Université de Strasbourg

UFR de Math Info

Rapport de stage

Master 1 Calcul scientifique et mathématique de l'information

Sujet:

Classification Cahn-Hilliard

Ikram KERMEZLI

Contents

1 Introduction

Le modèle de Cahn-Hilliard est un modèle d'équation aux dérivées partielles parabolique du quatrième ordre, introduit en 1958 par Cahn et Hilliard, [3]. Il joue un rôle important dans la science des matériaux. Il décrit des caractéristiques qualitatives de systèmes de deux phases en séparation, ayant les mêmes propriétés dans toutes les directions et de température constante. C'est un modèle d'évolution en temps.

Le modèle de Cahn-Hilliard a été mise en place pour décrire la séparation de phase avec conservation de masse. Cependant, l'utilisation du modèle de Cahn-Hilliard a été élargie à divers domaines de la chimie, de la physiques, de la biologie et au secteur de l'ingénierie, la retouche d'image, les flux des fluides multiphases, les micro-structures avec des inhomogénités élastiques, la simulation de la croissance tumorale, l'optimisation de topologie, la dynamique des populations, la théorie de la formation de galaxies, et même les caractéristiques des anneaux de Saturne, [2].

L'objectif de ce stage est de résoudre l'équation de Cahn-Hilliard en cherchant le meilleur contrôle pour approcher un état de la solution voulu. Ce rapport est organisé comme suit, dans la section 2 on présente le modèle de Cahn-Hilliard et différentes façons de discrétiser les equations différentielles du modèle. Dans la section 3 on présente l'implémentation de l'algorithme pour résoudre les équations et trouver le contrôle optimal. Puis on donne dans la section 4 les résultats numérique de l'application sur des données générés aléatoirement et les détails sur l'algorithme de Newton utilisé dans la résolution de l'équation CH. En fin, on donnera une conclusion du stage et des perspectives pour une éventuelle suite du projet.

2 Le modèle de Cahn-Hilliard

Le modèle de Cahn-Hilliard est défini comme suit, [1][2]:

$$\left\{\begin{array}{lcl} \partial_t u & = & \Delta(\phi'(u)+\gamma\Delta u) \\ u(t=0,.) & = & u_0(x) \end{array}\right. \quad (P)$$

avec u et u_0 deux fonctions de $\Omega \subset \mathbb{R}^2 \times [0,T]$ dans $\mathbb{R}$, $\gamma > 0$. On réécrit l'équation de Cahn-Hilliard comme suit:

$$\left\{\begin{array}{lcll} \partial_t u & = & \Delta w & \\ w & = & \phi'(u)+\gamma\Delta u & \quad (P') \\ u(0,.) & = & u_0(x) & \forall x \in \Omega \end{array}\right.$$

avec conditions périodiques, c'est à dire que le bord gauche est identifié au bord droit, et le bord du haut est identifié au bord du bas.

La formulation (P') s'appelle le "formulation mixte", elle permet de simplifier l'équation en deux EDP d'ordre 2, contrairement à la première ou on avait une seule d'ordre 4 et donc plus compliqué à résoudre.

2.1 Première version: Schéma implicite d'Euler

Pour résoudre numériquement ces EDP, on effectue une discrétisation implicite par différence finis de (P'), Pour notre étude on prend $\Omega = [0,1]\times[0,1]$ un carré de longueur 1, puis on le discrétise en divisant chaque coté par le nombre de points $N-1$, on obtient le pas d'espace $\Delta x = \frac{1}{N-1}$ et de même pour avoir le pas Δy. On trouve Δt avec la condition cfl :

$$\Delta t = cfl \times \Delta x$$

On pose:

$$\begin{array}{lcl} U^n_{i,j} & \approx & u(t_n, x_{i,j}) \\ W^n_{i,j} & \approx & w(t_n, x_{i,j}) \end{array}$$

Où :

$$\begin{array}{lcl} x_{i,j} & = & (i\Delta x, j\Delta y) \\ t_n & = & n\Delta t \end{array}$$

i et j allant de 1 à N. On discrétise avec les notations précédentes:

$$\left\{\begin{array}{lcl} \frac{U^{n+1}_{i,j}-U^n_{i,j}}{\Delta t} & = & \frac{W^{n+1}_{i-1,j}-2W^{n+1}_{i,j}+W^{n+1}_{i+1,j}}{\Delta x^2} + \frac{W^{n+1}_{i,j-1}-2W^{n+1}_{i,j}+W^{n+1}_{i,j+1}}{\Delta y^2}, \\ W^{n+1}_{i,j} & = & \phi'(U^{n+1}_{i,j}) + \gamma(\frac{U^{n+1}_{i-1,j}-2U^{n+1}_{i,j}+U^{n+1}_{i+1,j}}{\Delta x^2} + \frac{U^{n+1}_{i,j-1}-2U^{n+1}_{i,j}+U^{n+1}_{i,j+1}}{\Delta y^2}) \end{array}\right.$$

On obtient le système d'équations:

$$\left\{\begin{array}{lcl} U^{n+1}_{i,j} - \frac{\Delta t}{\Delta x^2}(W^{n+1}_{i-1,j}+W^{n+1}_{i+1,j}+W^{n+1}_{i,j-1}+W^{n+1}_{i,j+1}-4W^{n+1}_{i,j}) & = & U^n_{i,j}, \\ W^{n+1}_{i,j} - \phi'(U^{n+1}_{i,j}) - \frac{\gamma}{\Delta x^2}(U^{n+1}_{i-1,j}+U^{n+1}_{i+1,j}+U^{n+1}_{i,j-1}+U^{n+1}_{i,j+1}-4U^{n+1}_{i,j}) & = & 0. \end{array}\right. \quad (P'')$$

Le système discret s'écrit alors :

$$BX^{n+1} + G(X^{n+1}) = F^n \quad (P'')$$

Où B une matrice de la forme :

$$B = \begin{pmatrix} Id & \frac{\Delta t}{\Delta x^2} A \\ \frac{-\gamma}{\Delta x^2} A & Id \end{pmatrix} \in \mathbb{R}^{\mathbb{N},\mathbb{N}}$$

et A est la matrice de la discrétisation du laplacien et F est le vecteur second membre qui s'écrit :

$$F = \begin{pmatrix} U^n_{1,1} \\ \vdots \\ U^n_{N,N} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \in \mathbb{R}$$

L'inconnu du système est le vecteur X^{n+1} tel que :

$$X^{n+1} = \begin{pmatrix} U^{n+1}_{1,1} \\ \vdots \\ U^{n+1}_{N,N} \\ W^{n+1}_{1,1} \\ \vdots \\ W^{n+1}_{N,N} \end{pmatrix} \in \mathbb{R}$$

Et $G(X^{n+1})$ un vecteur bloc non-linéaire en U:

$$G(X^{n+1}) = \begin{pmatrix} 0 \\ \phi'(U^{n+1}) \end{pmatrix} \in \mathbb{R}$$

On obtient alors un système non-linéaire qu'on résout avec l'algorithme de Newton (voir section 4).

2.2 Deuxième version: θ-Schéma

Le schéma précédent n'est que d'ordre 1 en temps, pour améliorer ça on propose d'utiliser le schéma de Crank-Nicolson pouvant être un cas particulier du θ-schéma avec $\theta = 1/2$.

La θ-méthode consiste à approcher la dérivée spatiale au temps $n+1$ à l'aide de termes évalués à n (partie explicite) et de termes évalués à $n+1$ (partie implicite). Cette stratégie peut être appliquée, que la dérivée en espace soit approchée avec un schéma centré ou tout autre schéma. On effectue la discrétisation de l'équation et on construit le schéma comme suit:

$$\begin{cases} \frac{U^{n+1}-U^n}{\Delta t} &= \theta \mathcal{L}(W^{n+1}) + (1-\theta)\mathcal{L}(W^n) \\ W^{n+1} &= \phi'(U^{n+1}) + \gamma \mathcal{L}(U^{n+1}) \end{cases}$$

Où $\mathcal{L}$ représente la discrétisation du laplacien et

$$\phi'(U^{n+1}) \begin{pmatrix} \phi'(U_{1,1}) \\ \vdots \\ \phi'(U_{N,N}) \end{pmatrix}$$

$\theta = 0$ correspond au schéma d'Euler explicite,$\theta = 1$ au schéma d'Euler implicite, tandis que $\theta = 1/2$ donne lieu au schéma de Crank-Nicolson d'ordre 2 en temps($O(\Delta t^2)$), c'est à dire que ce schéma permet d'obtenir une erreur

$$\| u(t^n, x^n) - u^n \| \simeq O(\Delta t^2).$$

2.3 Schéma avec pas de temps adaptatif

Cette méthode consiste à changer le pas de temps Δt à chaque itération en l'adaptant sous condition de convergence ou non de l'itération précédente, on pose un critère de convergence pour l'algorithme de Newton, puis on test:

- Si le critère est satisfait, on calcule un nouveau Δt :

$$\Delta t = \alpha \times \Delta t$$

,

- Si le critère de convergence n'est pas satisfait, on calcule un nouveau pas de temps, mais cette fois en le diminuant :

$$\Delta t = \beta \times \Delta t$$

α et β étant des coefficients défini tel que :

$$\begin{cases} \alpha > 1 \\ \beta < 1 \end{cases}$$

Ce-ci permet une convergence plus rapide et un temps de calcul très petit par rapport à l'algorithme du solveur Cahn-Hilliard sans pas de temps adaptatif.

3 Implémentation

3.1 Solveur Cahn-Hilliard

Pour résoudre l'EDP du modèle de Cahn-Hilliard, on implémente un solveur qui utilise l'algorithme de Newton a chaque itération, pour cela, on défini un X_{init}, qu'on prend aléatoirement avec la fonction `numpy.rand.random()`, un T final, un nombre maximal d'itérations `iter_max` et la fonction J et son gradient dJ. Premièrement, la fonction J est définie d'après le system d'équations (P'') par :

$$J(X) = BX^{n+1} + G(X^{n+1}) - F^n$$

Pour simplifier l'implémentation, on utilise les fonctions `numpy.roll()` et `numpy.concatenate()`, c'est à dire, on calcul la fonction J en deux partie, la première $res(U)$ et la deuxième en $res(W)$, qu'on concatène pour retourner un seul vecteur $J(X)$:

- On effectue le calcul du laplacien pour chacun des vecteur inconnus U^{n+1} et W^{n+1} avec la fonction `Laplacien(x)` qui utilise `numpy.roll()`.
- Puis on calcul
$$res(U^{n+1}) = U^{n+1} - \Delta t \Delta W - U^n$$
et
$$res(W^{n+1}) = \phi'(U^{n+1}) - W - \gamma \Delta U^{n+1}$$
- on retourne le résultat avec `numpy.concatenate()` en pensant à bien effectuer un `numpy.reshape()` des vecteur $res(U)$ et $res(W)$ pour obtenir un vecteur de taille N^2.

Le `numpy.reshape()` permet d'utiliser la fonction J dans l'algorithme de Newton. Avant d'utiliser la fonction `numpy.roll()` il faut faire un `numpy.reshape()` sur le vecteur X^{n+1} pour faciliter la manipulation et le calcul du laplacien.

Une fois la fonction J calculée, on doit aussi obtenir le gradient de la fonction, on utilise pour cela la bibliothèque `autograd.numpy` en faisant un `from autograd import jacobian`, la fonction `jacobian()` retourne le gradient, on fait appel à cette fonction dans `dJ(X)` qu'on utilise directement dans l'algorithme de Newton.

Pour utiliser le schéma de Crank-Nicolson, une modification de la précédente fonction J est requise, il suffit juste de changer le $res(U^{n+1})$ comme ceci:

$$res(U^{n+1}) = U^{n+1} - \theta(\Delta t \Delta W^{n+1}) - (1-\theta)(\Delta t \Delta W^n) - U^n$$

En posant $\theta = \frac{1}{2}$. La fonction `dJ` se calcule de la même manière.

Le schéma du pas de temps adaptatif est implémenter dans la boucle de l'algorithme de résolution de l'équation de Cahn-Hilliard. Pour un temps final T fixé et un nombre d'itérations nb maximal, on procède comme ceci:

1) On effectue l'appel à de l'algorithme de Newton,

2) On test la convergence de ce dernier:

   . Si l'algorithme converge on augmente le pas de temps Δt d'un certain pourcentage fixé :

   $$\Delta t = \alpha \times \Delta t$$

   et on met à jour les compteurs de temps t et du nombre d'itérations.

   . Si l'algorithme ne converge pas, on met à jour le pas de temps seulement :

   $$\Delta t = \beta \times \Delta t$$

   et on revient à l'étape 1).

α et β sont des constantes tel que:

$$\begin{cases} \alpha & > 1 \\ \beta & < 1 \end{cases}$$

3.2 Passage en Pytorch

Pour pouvoir résoudre l'EDP en introduisant le contrôle on a besoin de trouver le contrôle optimal V qui minimise la quantité :

$$\| u(T, x, V) - \hat{u} \|^2$$

et pour se faire, on propose d'utiliser la bibliothèque Pytorch qui permet de créer un graphe et donc de faire la différenciation de l'optimisation de graphe.

C'est à dire que ça permet d'avoir un graphe sur lequel on va chercher le paramètre V et de calculer la dérivé d e la quantité a minimiser par rapport à ce paramètre.

Pour cela, il faut d'abord réécrire les fonctions précédentes et passer de `numpy` à `torch`. La bibliothèque `torch` va permettre de manipuler des tenseurs, contrairement à `numpy` avec la quelle on manipule des vecteurs.

Il existe plusieurs fonctions qui font la même chose dans les deux bibliothèques, sauf qu'elle ne s'utilise pas pareil et ont des syntaxe différentes, pour écrire les algorithme précédent en `torch`, voici les fonctions qu'on change:

. La fonction `numpy.copy()` devient `torch.clone()`.

. `numpy.array()`, pour transformer une liste en vecteur, devient `torch.tensor()` pour changer la liste en tenseur.

. La fonction `roll()` est la même et s'utilise de la même manière aussi, sauf que pour `numpy` le 3-ème argument qui détermine l'axe c'est `axis=ax` et pour `torch` c'est `dims=dimension`.

. La fonction `numpy.concatenate()`, vu dans la section 3.1, devient `torch.cat()` et pour le 2-ème argument `axis=ax` en `numpy` cela devient `dims=dimension` en `torch`.

. La fonction `jacobian()` simple d'utilisation en `numpy` qui retourne la matrice jacobienne d'une fonction donnée en argument; elle devient `torch.autograd.functional.jacobian()` et on lui passe 3 arguments qui sont: la fonction, la variable de la fonction et un booléen pour `create_graph=booleen`, ici on utilise `True`.

. Pour pouvoir afficher la solution avec `matplotlib.imshow()` il faut d'abord convertir le tenseur en un vecteur `numpy` en utilisant `detach().cpu().numpy()`.

3.3 Algorithme de la descente du gradient

Après avoir mis en point l'algorithme de résolution de Cahn-Hilliard et réécris le programme en `torch`, on passe à l'algorithme de la descente du gradient qui résout l'EDP en trouvant le contrôle V qui permet d'obtenir un résultat attendu. En introduisant le contrôle, l'EDP s'écrit comme suit:

$$\begin{cases} \partial_t u & = & \Delta w - v \\ w & = & \phi'(u) + \gamma \Delta u \\ u(0,.) & = & u_0(x) \qquad\qquad \forall x \in \Omega \end{cases}$$

On refait la discrétisation implicite par différence finis et on obtient le même schéma, sauf pour le vecteur second membre F^n on a:

$$F = \begin{pmatrix} U^n_{1,1} - V^n_{1,1} \\ \vdots \\ U^n_{N,N} - V^n_{N,N} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \in \mathbb{R}$$

Ce qui change dans l'implémentation c'est le $res(U^{n+1})$ dans la fonction J:

$$res(U^{n+1}) = U^{n+1} - \theta(\Delta t \Delta W^{n+1}) - (1-\theta)(\Delta t \Delta W^n) - U^n - V$$

On propose de définir le contrôle V par 3 constantes, c'est a dire qu'on cherche le contrôle dans un espace tel que:

$$H = \left\{ V = v_0 f_0(x) + v_1 f_1(x) + v_2 f(t) / (v_0, v_1, v_2) \in \mathbb{R}^3 \right\}.$$

Où f_0 et f_1 sont des fonctions constantes par morceaux, de $\mathbb{R}^2$ dans $\mathbb{R}$; on les appelle aussi base de Haar et f une fonction qui dépend du temps.

Et donc on cherche à minimiser la quantité :

$$\| u(T, x, V) - \hat{u} \|^2$$

dans l'espace H, pour cela on propose d'utiliser l'algorithme de la descente du gradient défini comme ceci:

1) On fixe un V_0 initial et un nombre d'itérations maximal,

2) On défini l'optimiser:

```
optimizer = torch.optim.Adam([V], lr=lr)
```

où `lr` étant une constante fixé au début et V est la variable.

3) A chaque itération on résout l'équation de Cahn-Hilliard avec l'algorithme de CH, on calcul la fonction a minimiser:

```
loss = torch.norm(U-y)**2
```

où `U` est la solution de l'algorithme de CH et `y` est la solution cible qu'on veut approcher avec le V. on rajoute les lignes:

```
loss.backward()
```

```
optimizer.step()
```

et la variable V est actualisée, ceci jusqu'à un certain nombre d'itérations.

On peut trouver l'implémentation du programme complet en Python ici.

4 Tests Numériques

4.1 Algorithme de Newton

Soit $f : I \to \mathbb{R}^n$ une fonction régulière et non-linéaire, on veut résoudre $f(x) = 0$ avec $x \in I$, on propose d'utiliser l'algorithme de Newton détaillé ci-après:

1) On prend un x^0 initial qui ne soit pas très loin de la solution x^*,

2) On définit la suite: $x^{k+1} = x^k + \nabla f(x^k)^{-1} * f(x^k)$.

l'implémentation de l'algorithme en Python est accessible ici.

Elle prend en paramètres la fonction f et son gradient, le x^0, un nombre maximal d'itérations et un critère d'arrêt ε. Elle retourne la solution x^* et la liste des itérés avec laquelle on va vérifier que $\| f(x^k) \|$ tend vers 0.

On s'assure que l'algorithme donne bien la solution de $f(x) = 0$ en choisissant quelques fonctions pour tester:

1-er exemple on prend une fonction f de $\mathbb{R}^2$ dans $\mathbb{R}^2$: $f(x, y) = (\cos(x), \sin(y))$ avec

$$\nabla f(x) = \begin{pmatrix} -\sin(x) & 0 \\ 0 & \cos(y) \end{pmatrix}$$

On a: $f(x^*, y^*) = 0 \Rightarrow x^* = \frac{\pi}{2} + k\pi$ et $y^* = \pi + k\pi$ avec $k \in Z$ en choisissant $(x^0, y^0) = (1, 1)$ on obtient avec le solver: $x^* = (1.57, 0)$ en 5 itérations.

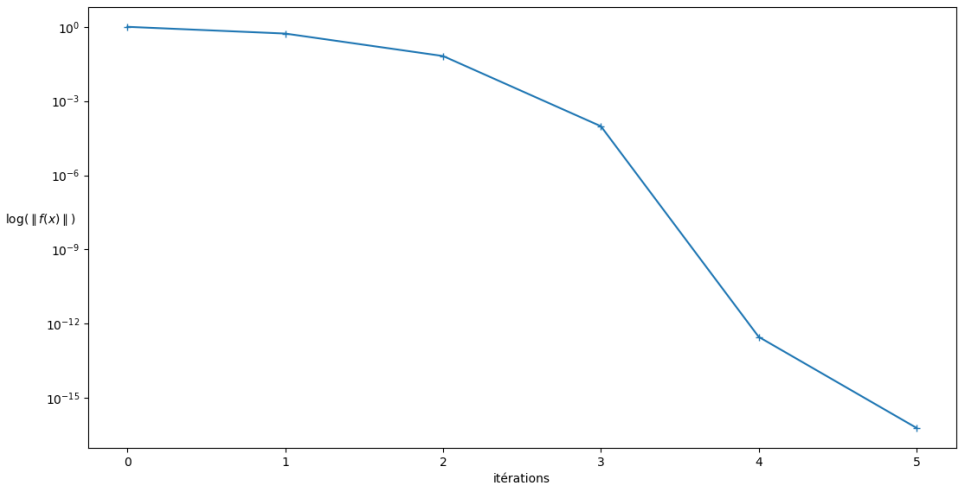

Figure 1: Graphe de $\log(\| f(x^k, y^k) \|)$, $f(x, y) = (\cos(x), \sin(y))$

2-ème exemple on prend une autre fonction f de $\mathbb{R}^3$ dans $\mathbb{R}^3$: $f(x,y,z) = (x^2, y^2, -z^2)$ avec

$$\nabla f(x,y,z) = \begin{pmatrix} 2x & 0 & 0 \\ 0 & 2y & 0 \\ 0 & 0 & -2z \end{pmatrix}$$

On a : $f(x^*, y^*, z^*) = 0 => (x^*, y^*, z^*) = (0,0,0)$;

on obtient avec le solver $(x^*, y^*, z^*) = (1.3 \times 10^{-17}, 1.3 \times 10^{-17}, 1.3 \times 10^{-10})$ qu'on considère égale à 0. On trace le graphe de $\log(\| f(x^k, y^k, z^k) \|)$ en fonction des itérations, on voit bien que c'est décroissant jusqu'à ce qu'on obtient des valeurs nulles:

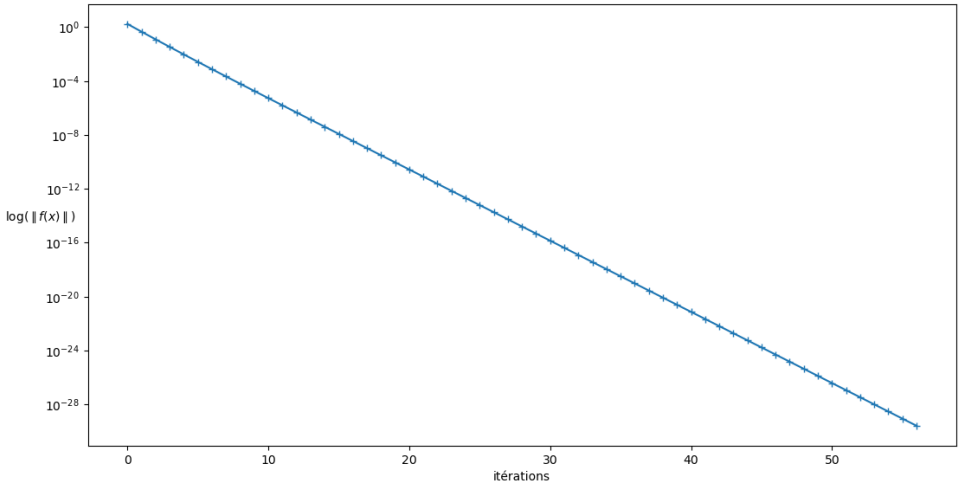

Figure 2: Graphe de $\log(\| f(x^k, y^k, z^k) \|)$, $f(x,y,z) = (x^2, y^2, -z^2)$

4.2 Comparaison des algorithmes sans et avec pas de temps adaptatif

Pour illustrer la différence entre les deux version de l'algorithme de résolution de l'équation CH, on effectue deux résolution et on compare les résultats obtenus ci-dessous:

Premièrement, l'algorithme sans pas adaptatif, c'est à dire que le pas de temps Δt est constant pendant toute l'exécution de l'algorithme, on prend comme paramètres :

$$\begin{cases} \text{nombre de points} &= 20 \\ \Delta x = \Delta y &= 5.3\text{e-}2 \\ \gamma &= 1\text{e-}4 \\ \Delta t_0 &= 2.77\text{e-}4 \\ \text{T final} &= 0.1 \end{cases}$$

On obtient:

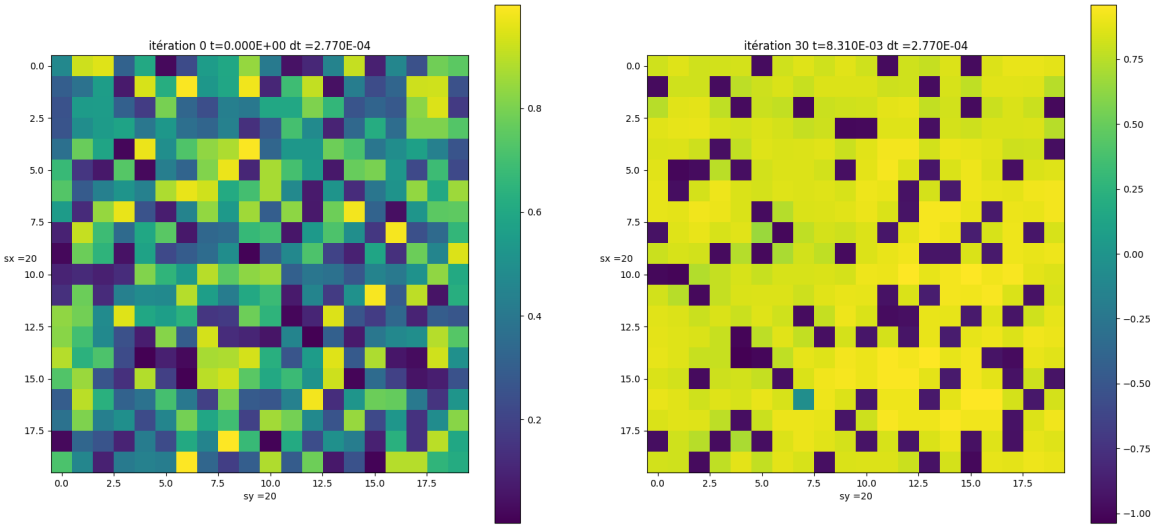

Figure 3: Solution sans pas de temps adaptatif

résultats :

$$\begin{cases} \text{temps de calcul} &= 3 \text{ minutes et } 10,17 \text{ secondes} \\ \text{temps final t} &= 8.31\text{e-}3 \\ \text{nombre d'itérations} &= 30 \\ \text{minimum de la solution} &= -1.037 \\ \text{maximum de la solution} &= 9.592\text{e-}1 \end{cases}$$

On remarque que l'algorithme a atteint le nombre maximal d'itération 30 sans arriver à T qui est de 0.1, donc il prend plus de temps puisque à chaque itération il ajoute au temps une très petite quantité et donc il prend du temps a converger.

Puis, l'algorithme avec pas adaptatif où le pas de temps Δt augmente ou diminue à chaque itération,selon la convergence ou non de l'algorithme de Newton, on prend les paramètres :

$$\begin{cases} \text{nombre de points} &= 20 \\ \Delta x = \Delta y &= 5.3\text{e-}2 \\ \gamma &= 1\text{e-}4 \\ \Delta t_0 &= 1\text{e-}3 \\ \text{T final} &= 5 \end{cases}$$

On obtient ceci :

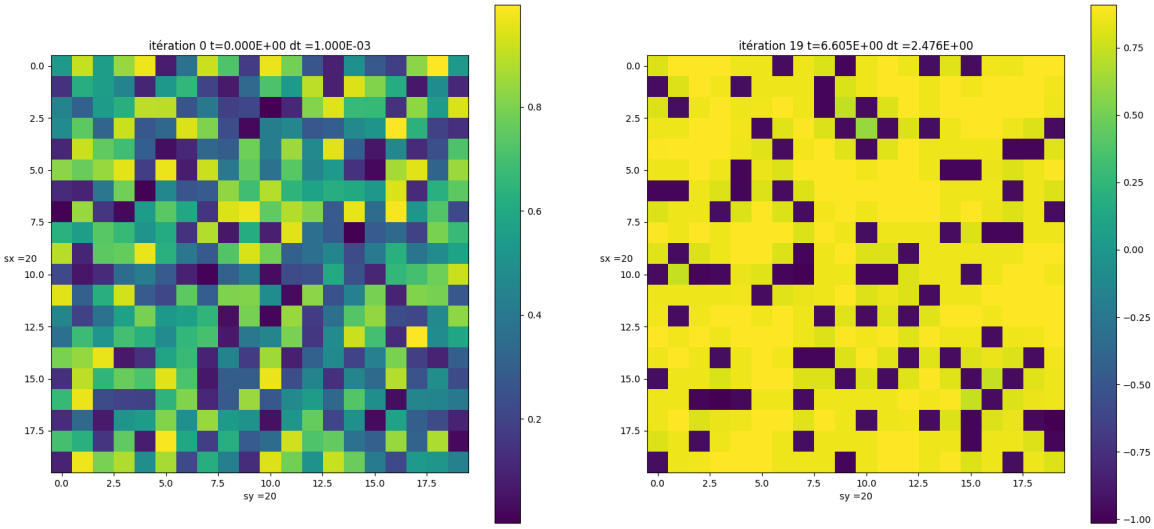

Figure 4: Solution avec pas de temps adaptatif

résultats:

$$\begin{cases} \text{temps de calcul} &= 3 \text{ minutes et } 11,67 \text{ secondes} \\ \text{temps final t} &= 6.61 \\ \text{nombre d'itérations} &= 19 \\ \text{minimum de la solution} &= -1.014 \\ \text{maximum de la solution} &= 9.089\text{e-}1 \end{cases}$$

On obtient la solution cette fois en seulement 19 itérations pour un temps final T beaucoup plus grand, ceci est du au fait que le pas de temps augmente a chaque convergence de l'algorithme de Newton et donc il s'adapte pour obtenir une convergence plus rapide.

4.3 Graphe de la fonction ϕ

Potentiel de Cahn-Hilliard Cette fonction est définie de $\mathbb{R}$ dans $\mathbb{R}$ comme suit:

$$\phi(x) = (1/4) * (x^2 - 1)^2$$

son graphe est de la forme suivante :

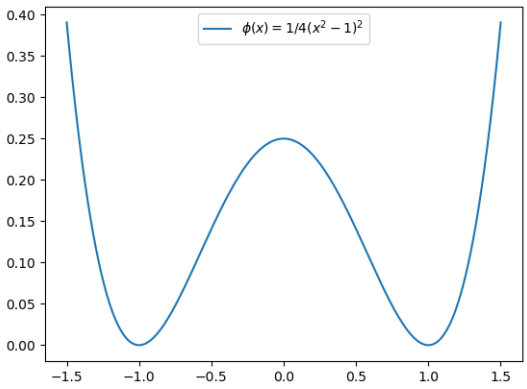

Figure 5: Graphe de la fonction potentiel de Cahn-Hilliard

On voit que la fonction atteint son minimum en -1 et 1, ce qui signifie que sa dérivé ϕ' s'annule en ces deux points.

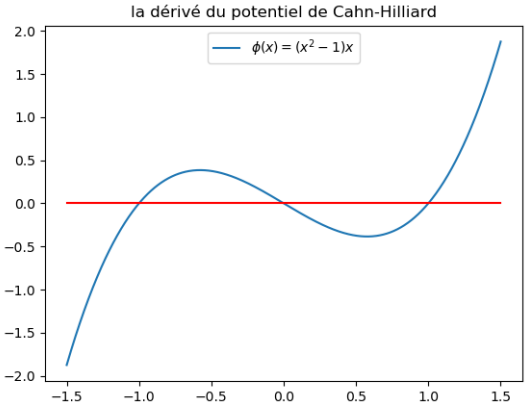

Figure 6: Graphe de la dérivé du potentiel de Cahn-Hilliard

En utilisant cette fonction dans le modèle Cahn-Hilliard, la solution U converge et ses valeurs minimum et maximum sont entre ces deux valeurs -1 et 1. On prend un exemple pour illustrer:

$$\begin{cases} \text{nombre de points} &= 12 \\ \Delta x = \Delta y &= 0.091 \\ \gamma &= \text{1e-3} \\ \Delta t_0 &= \text{1e-3} \\ \alpha &= 1.6 \\ \beta &= 0.8 \\ \text{T final} &= 0.5 \end{cases}$$

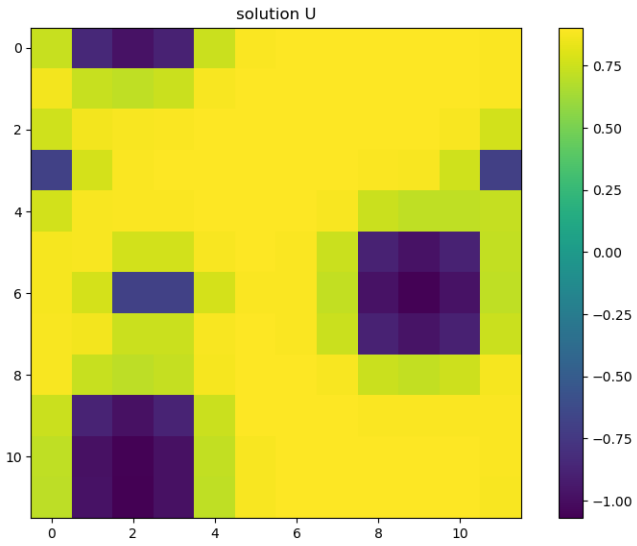

Figure 7: Solution Cahn-Hilliard avec potentiel de Cahn-Hilliard

résultats:

$$\begin{cases} \text{temps de calcul} &= 60.97 \text{ secondes} \\ \text{temps final t} &= 5.02\text{e-}1 \\ \text{nombre d'itérations} &= 20 \\ \text{minimum de la solution} &= -1.038 \\ \text{maximum de la solution} &= 9.441\text{e-}1 \end{cases}$$

Sur le graphe de la solution U, les valeurs prise sont très proches des points ou le minimums de la fonction ϕ est atteint.

Potentiel de Ginzburg-Landau Fonction définie de de $\mathbb{R}$ dans $\mathbb{R}$ comme suit:

$$\phi(x) = x^4 - x^2$$

son graphe est de la forme suivante :

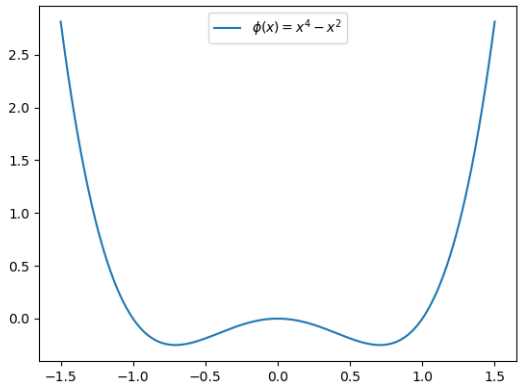

Figure 8: Graphe de la fonction potentiel de Ginzburg-Landau

On voit que la fonction atteint son minimum en -0.7 et 0.7, donc sa dérivée ϕ' s'annule en ces deux points.

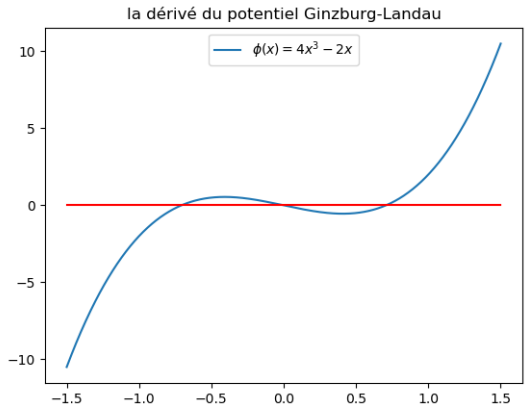

Figure 9: Graphe de la dérivé du potentiel de Ginzburg-Landau

Avec cette fonction, la solution U du modèle de Cahn-Hilliard converge et prend ses valeurs minimum et maximum dans l'intervalle $]-0.7, 0.7[$. En prenant un exemple, avec les mêmes paramètres initial, on obtient:

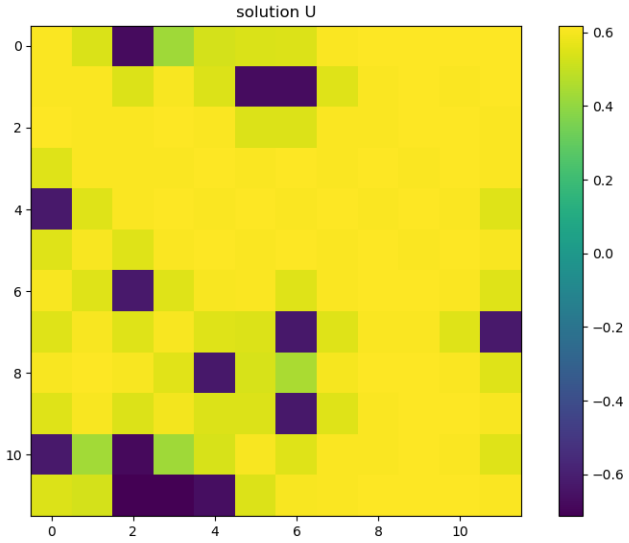

Figure 10: Solution Cahn-Hilliard avec potentiel de Ginzburg-Landau

résultats :

$$\begin{cases} \text{temps de calcul} &= 12.22 \text{ secondes} \\ \text{temps final t} &= 6\text{e-}1 \\ \text{nombre d'itérations} &= 11 \\ \text{minimum de la solution} &= -7.012\text{e-}1 \\ \text{maximum de la solution} &= 5.950\text{e-}1 \end{cases}$$

On trouve ici le code en Python pour obtenir les graphes des potentiels de Cahn-Hilliard et de Ginzburg-Landau.

4.4 Paramètre de pénalisation γ

C'est un paramètre qui pénalise la condition de $\Delta u = 0$ dans l'équation:

$$\partial_t u = \Delta(\phi'(u) + \gamma \Delta u)$$

L'algorithme qu'on met en place vas devoir minimiser la quantité :

$$\int_\Omega \phi'(u) + \gamma |\nabla u|^2 \, dx$$

Et donc pour minimiser la somme de deux termes positif, il faut que chaque terme soir petit, cependant si on choisit un γ qui tend vers 0, l'algorithme ne va pas fournir d'effort pour minimiser $|\nabla u|$, et il va se concentrer plutôt sur la minimisation du terme $\phi'(u)$

Par contre quand γ est assez grand (par exemple 1 ou 10^{-1}), l'algorithme va chercher des solutions $u(x)$ tel que :

$$\begin{cases} \phi(u) & \to 0 \\ |\nabla u| & \to 0 \end{cases}$$

et faire tendre $|\nabla u| \to 0$ revient à chercher une solution constante par morceaux.

On prend deux exemples pour illustrer le rôle du paramètre, on utilise le potentiel de Cahn-Hilliard, avec en premier un γ assez petit:

$$\begin{cases} \text{nombre de points} &= 10 \\ \Delta x = \Delta y &= \text{1.11e-1} \\ \gamma &= \text{1e-4} \\ \Delta t_0 &= \text{1e-3} \\ \alpha &= 1.6 \\ \beta &= 0.8 \\ \text{T final} &= 0.1 \end{cases}$$

On obtient:

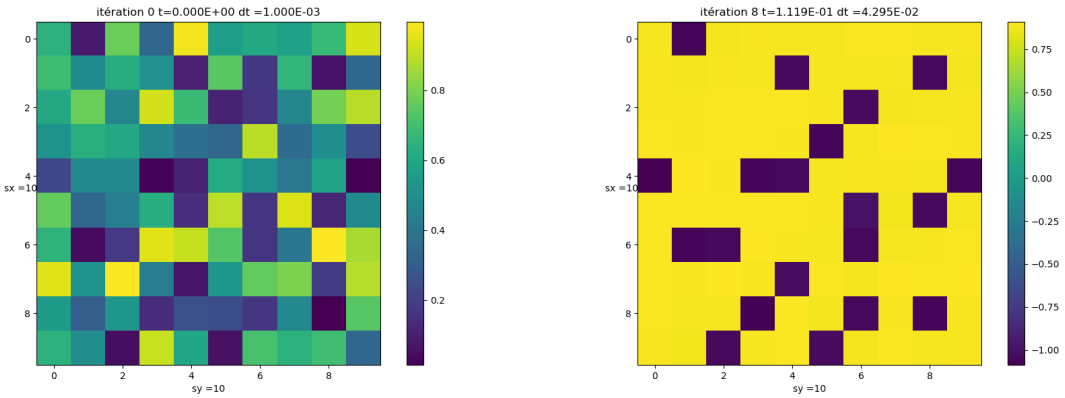

Figure 11: simulation avec $\gamma =$ 1e-4

résultats :

$$\begin{cases} \text{temps de calcul} &= 7.23 \text{ secondes} \\ \text{temps final t} &= 1.12\text{e-}1 \\ \text{nombre d'itérations} &= 7 \\ \text{minimum de la solution} &= -1.08 \\ \text{maximum de la solution} &= 9.113\text{e-}1 \end{cases}$$

Avec γ assez petit, l'algorithme converge en 7 itérations, la résolution est assez rapide puisque l'algorithme ne s'est pas trop attarder sur le 2-ème terme de l'équation.

On prend un deuxième exemple avec un γ plus grand:

$$\begin{cases} \text{nombre de points} &= 10 \\ \Delta x = \Delta y &= 1.11\text{e-}1 \\ \gamma &= 1\text{e-}2 \\ \Delta t_0 &= 1\text{e-}3 \\ \alpha &= 1.6 \\ \beta &= 0.8 \\ \text{T final} &= 2 \end{cases}$$

On obtient:

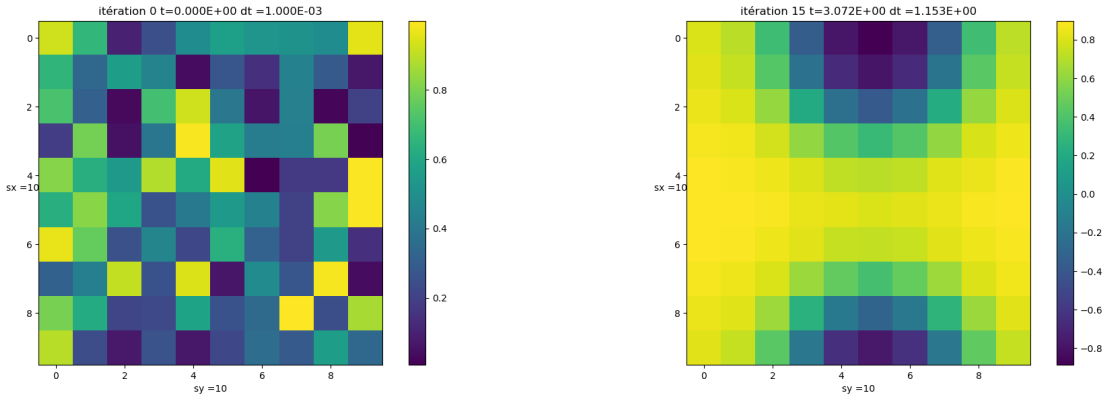

Figure 12: simulation avec $\gamma = 1\text{e-}2$

résultats :

$$\begin{cases} \text{temps de calcul} &= 10.84 \text{ secondes} \\ \text{temps final t} &= 3.07 \\ \text{nombre d'itérations} &= 14 \\ \text{minimum de la solution} &= -1.023 \\ \text{maximum de la solution} &= 9.94\text{e-}1 \end{cases}$$

On remarque que cette fois l'algorithme à pris un peu plus de temps pour converger, 14 itérations, et ce due au fait que γ est assez grand pour que l'algorithme ne néglige pas le terme $|\nabla u|$ et donc il prend un peu plus de temps à trouver la solution qui satisfait les deux condition en même temps.

On l'appelle aussi paramètre de raideur, quand on fait une interprétation plutôt visuelle. Sur les illustrations précédentes, on voit qu'il y a deux état constant qui valent 1 et -1.

Quand γ est grand la zone qui sépare les deux états est plus douce, ça donne une sorte de marche.

Quand γ est petit, la zone qui sépare les deux état varie fortement et devient une pente vertical.

Pour la suite on utilise $\gamma =$ 1e-3 ou 1e-4 pour pouvoir faire de la classification et avoir des motifs plus apparents en traçant la solution.

4.5 Résolution de l'équation avec le contrôle V

Pour cette dernière étape de la simulation, on résout le système d'EDP en prenant un $u_0(x)$ qui soit un peu proche de $\hat{u}$, on fixe les paramètres comme suit:

$$\begin{cases} \text{nombre de points} &= 8 \\ \Delta x = \Delta y &= \text{1.43e-1} \\ \gamma &= \text{1e-4} \\ \Delta t_0 &= \text{1e-3} \\ \alpha &= 1.6 \\ \beta &= 0.8 \\ \text{T final} &= 0.1 \end{cases}$$

Pour le choix de la solution cible $\hat{u}$ on propose d'utiliser la bibliothèque `sklearn.datasets` et d'importer la database `load_digits`, ce sont les images des chiffre de 0 à 9 stoker dans des matrice de taille 8×8, d'où le choix précédent $n = 8$ c'est pour permettre de calculer la norme : $\| u(T, x, v) - \hat{u} \|^2$.

L'exécution prend assez de temps du fait qu'il y a plusieurs algorithme à exécuter. On obtient les résultats suivant:

- D'abord le résultat de l'algorithme du gradient pour obtenir le `Vopt`:

A la fin de ce dernier, on a le graphe de la `loss` en fonction des itérations, dans cet exemple le nombre d'itérations est de 10:

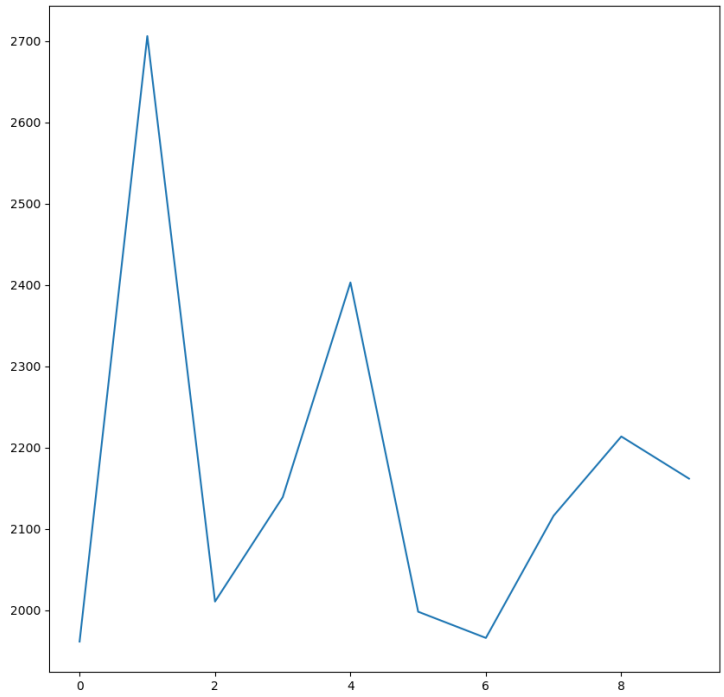

Figure 13: la loss en fonction des itérations

On remarque que la loss n'est pas stable, elle est croissante au début mais décroit vite après quelque itérations.

On obtient aussi le V optimal:

$$V = (0.0409, -0.6593, 0.1405)$$

avec lequel on passe à la deuxième étape,

- l'exécution du solveur CH :

En 8 itérations, on peut avoir la comparaison suivante :

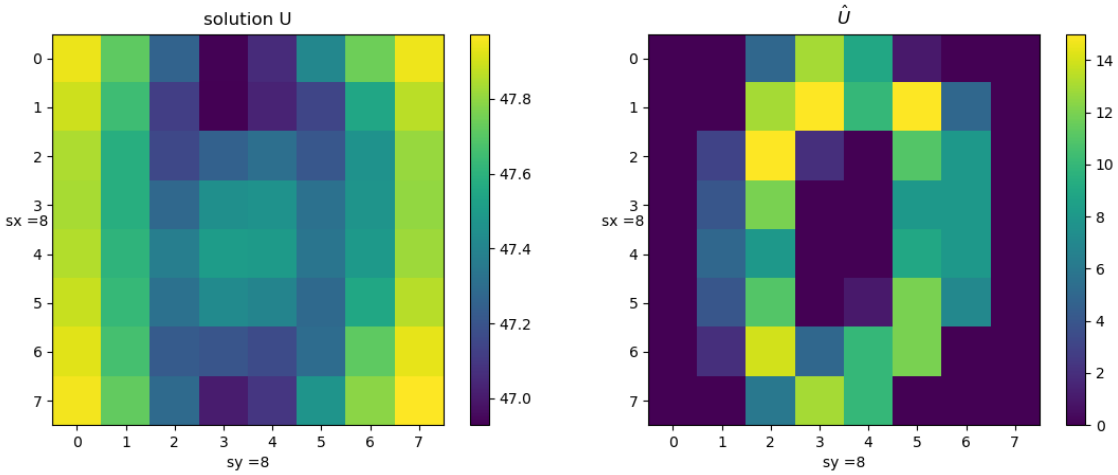

Figure 14: simulation avec le V optimal

Certes, l'échelle n'est pas la même, mais on remarque que les deux graphe on un peu la même forme. En effectuant la simulation avec plusieurs itérations, l'algorithme a réussi à garder la forme voulu de la solution.

Pour s'assurer que la solution est convergente, on regarde les graphes en semilog de $J(X)$ en fonction des itérations de l'algorithme de Newton:

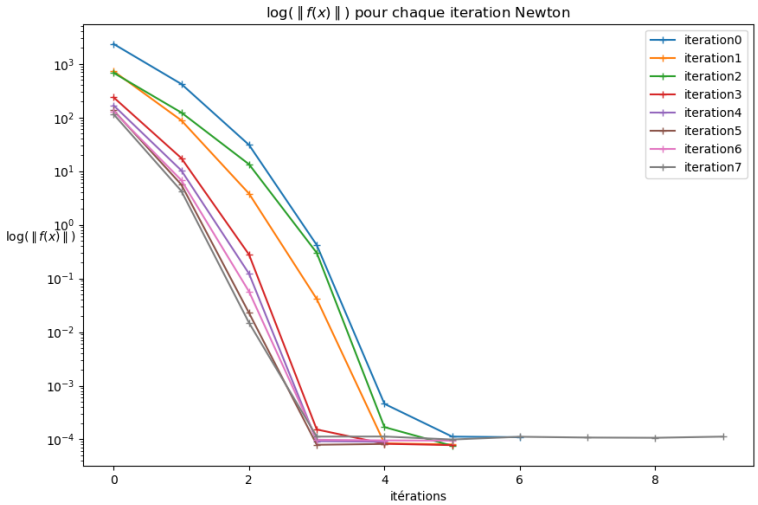

Figure 15: Itération 0 de la descente du gradient

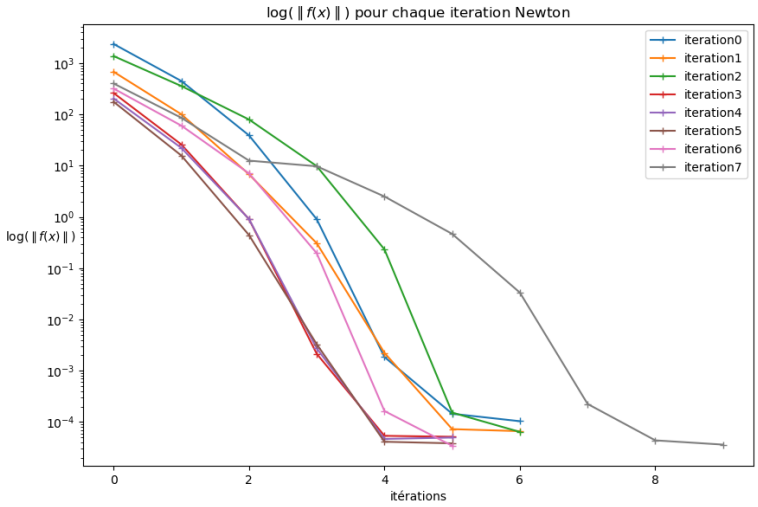

Figure 16: Itération 4 de la descente du gradient

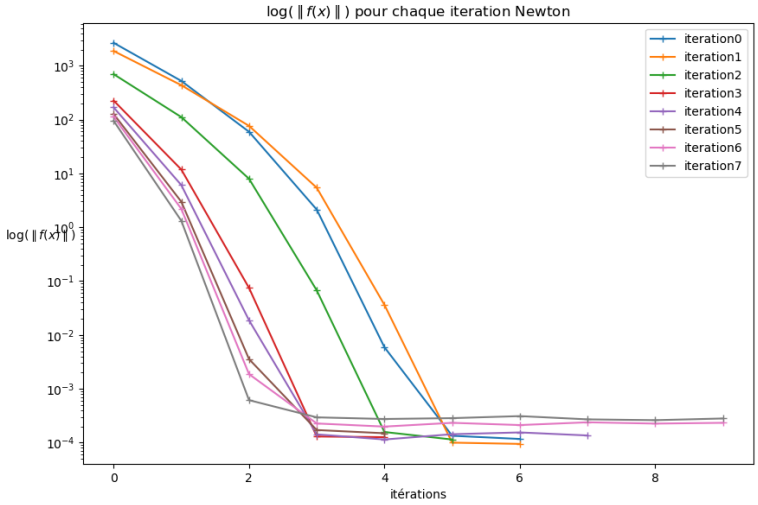

Figure 17: Itération 9 de la descente du gradient

Les 3 figure ci-dessus montrent bien que la solution est convergente, du fait qu'on a $\log(\| J(U) \|)$ qui décroit et atteint 10^{-4}, donc l'équation $J(U) = 0$ est résolution et on a la solution numérique du système d'équations de CH.

5 Conclusion

Durant ce stage, on a effectuer plusieurs étapes. Tout d'abord on a fait l'étude et la discrétisation du modèle de Cahn-Hilliard, de différentes façons, on a fait la mise en point d'un algorithme de Newton qui a permis de résoudre l'EDP et d'obtenir la convergence de la solution du modèle. Puis on a proposer une façon d'aborder le modèle en rajoutant un contrôle pour permettre d'obtenir une solution contrôlée par un résultat voulu au départ.

Même si la dernière étape n'a pas totalement abouti, le stage a permis d'approcher le modèle de Cahn-Hilliard sous d'une manière intéressante et qui peut être développée et utilisée dans la classification, ou dans le filtrage d'image par exemple.

Dans un futur travail, il serait prenant d'approfondir l'étude en abordant d'une autre façon le modèle de Cahn-Hilliars et d'essayer d'introduire et de définir le contrôle de plusieurs manières.

References

[1] C.M. Elliott, D.A. French and F.A. Milner, *A Second Order Splitting Method for the Cahn-Hiiliard Equation* , Numerische Mathematik, volume 54,pages: 575-590, 1989.

[2] Junseok Kim, Seunggyu Lee, Yongho Choi, Seok-Min Lee and Darae Jeong, *Basic Principles and Practical Applications of the Cahn–Hilliard Equation*, Hindawi Publishing Corporation, volume 2016, 27 Oct 2016.

[3] J.W. Cahn and J. E. Hilliard, *Free energy of a nonuniform system I, Interfacial free energy*, J. Chem. Phys. 28 (1958), 258-267.