Polytech Clermont

IMDS 4A

Equation de Burgers

Rapport de projet réalisé de septembre 2022 à mars 2023

Présenté par : Félix Baubriaud et Merwan Benbadra

Date de soutenance : 24 mars 2023

Non confidentiel

Tuteur Polytech Clermont : Thierry Dubois

Année 2022/2023

Résumé

L'équation de Burgers est une équation au dérivées partielles permettant de modéliser des problèmes dans des domaines variés. Différents procédés mathématiques sont utilisés pour résoudre l'équation, tels que la discrétisation, le schéma de Crank-Nicolson et la méthode de Newton. Une fois tous les calculs écrit correctement, il est possible de résoudre numériquement le problème et d'afficher la courbe représentative du résultat en utilisant un langage comme Python. On peut également comparer l'approximation obtenue par nos méthodes avec une solution exacte donnée pour une certaine condition initiale.

Mots-clés

-Différences finies -Equation -Discrétisation -Dérivés partielles

-Approximation -Implémentation -Graphique -Modélisation

-Erreurs -Récurrence -Système -Programmation

Abstract

Burgers equation is a partial differential equations that enables to models problems in various field. There are different mathematical process that are used to solve this equation such as discretization, Crank-Nicolson method and Newton method. Moreover, it is possible to solve numerically this problem and to show the representative curve of the solution using Python for instance. We can also compare the exact solution with the approximation get with our methods.

Keywords

-Finite difference -Equation -Discretization -Partial derivative

-Approximation -Implementation -Graph -Modelization

-Errors -recurrence -System -Programming

REMERCIEMENTS

Nous tenons à remercier toutes les personnes qui nous ont aidé pour ce projet.

Nous remercions tout d'abord notre tuteur M. Thierry Dubois, qui nous a indiqué la marche à suivre pour effectuer ce projet au mieux et nous a accompagné tout au long de l'année. Ses conseils et corrections nous ont permis d'avancer efficacement dans nos recherches et de toujours avoir un but précis.

Nous remercions également notre professeur M. François Bouchon, avec qui nous avons étudié un certain nombre de prérequis pour le projet en cours de différence finies et calcul différentiel.

Enfin nous remercions l'enseignant chercheur M. Rachid Touzani, aujourd'hui à la retraite, pour son cours d'analyse numérique qui nous a également été très utile pour venir à bout de ce projet.

TABLE DE MATIERES

INTRODUCTION

L'équation de Burgers est une équation aux dérivées partielles provenant de la mécanique des fluides. Elle permet par exemple de modéliser des problèmes de dynamique des gaz, de l'acoustique ou de trafic routier.

L'équation de Burgers doit son nom à Johannes Martinus Burgers (physicien néerlandais). Il est né en 1895 aux Pays-Bas et meurt en 1981 à Washington. Il s'est intéressé principalement à la mécanique des fluides, la mécanique du solide et à la science des matériaux.

En 1918, il obtient un doctorat en mathématiques et physique à l'université de Leiden. Il enseigne pendant 37 ans l'aérodynamique et l'hydrodynamique au Pays-Bas à l'université. Cette équation a été introduite par Bateman en 1915 qui a fourni l'état d'équilibre de la solution, en 1948 Burgers propose de faire de l'équation de Burgers un modèle mathématique de la turbulence. Il s'agit de l'état d'un fluide, liquide ou gaz pour lequel la vitesse est tourbillonnaire.

En 1964, il est récompensé pour ses travaux dans le domaine de la rhéologie (science qui s'intéresse aux déformations et écoulements de la matière).

Le but de ce projet est, dans un premier temps, de résoudre l'équation de Burgers en utilisant des outils mathématiques appropriés, et dans un deuxième temps d'implémenter un programme permettant de visualiser le résultat obtenu. Nous allons donc diviser ce rapport en 5 parties : d'abord nous présenterons l'équation de Burgers. Ensuite, nous proposerons un schéma discrétisé du problème puis nous mettrons en œuvre sur ce dernier la méthode de Newton. Enfin, nous détaillerons l'implémentation du problème et nous étudierons les résultats obtenus.

1 PRESENTATION DU PROBLEME

On s'intéresse aux équations de la forme :

$$(\mathcal{S})\begin{cases}\partial_t u(x,t) + u(x,t)\partial_x u(x,t) = \nu\partial^2_{xx}u(x,t),\ \forall x \in (0,L),\ \forall t \in (0,T] \\ u(x,0) = u_0(x),\ \forall x \in (0,L) \\ u(0,t) = u(L,t) = 0, \forall t \in (0,T]\end{cases}$$

où $\nu > 0$ est un paramètre fixé (appelé la viscosité) et u_0 une fonction donnée.

L correspond à la longueur du domaine et T à la durée mesurée.

L'objectif est de trouver une fonction u dépendant du temps t et de la position x, vérifiant le système d'équation ci-dessus.

Remarque : on a $u(x,t)\partial_x u(x,t) = \frac{1}{2}\partial_x(u(x,t)^2)$.

Dans ce projet, nous étudions le problème en dimension 1, ce qui signifie que le domaine sur lequel on mesure notre fonction u est un segment de longueur L comme le montre l'équation. Si le domaine était une aire, on aurait $x = (x_1, x_2) \in \Omega \subset \mathbb{R}^2$. De même, si on voulait modéliser l'équation dans un domaine en 3 dimensions, on aurait $x = (x_1, x_2, x_3) \in \Omega \subset \mathbb{R}^3$. Cependant, passer en dimension 2 ou 3 complique les calculs, c'est pourquoi nous gardons $x \in (0, L) \subset \mathbb{R}$.

2 SCHEMA NUMERIQUE

Pour résoudre ce problème, nous allons proposer un schéma de Crank-Nicolson en temps et centré d'ordre 2 en espace.

2.1 DISCRETISATION

On considère pour tout $i \in 0, \dots, I$, $x_i = ih$ avec I le nombre de points de discrétisation en espace et $h = \frac{L}{I}$ le pas d'espace.

De plus, on pose pour tout $k \in 0, \dots, K$, $t^{(k)} = k\delta t$ avec K le nombre de points de discrétisation en temps et $\delta t = \frac{T}{K}$ le pas de temps.

L'équation de Burgers devient

$$(\mathcal{S}) \quad \begin{cases} \partial_{t^{(k)}} u\left(x_i, t^{(k)}\right) + u\left(x_i, t^{(k)}\right) \partial_{x_i} u\left(x_i, t^{(k)}\right) = \nu \partial^2_{x_i x_i} u\left(x_i, t^{(k)}\right), \forall i \in 0, \dots, I, \forall k \in 0, \dots, K \\ u(x_i, 0) = u_0(x_i), \forall i \in \{0, \dots, I\} \\ u\left(0, t^{(k)}\right) = u\left(I, t^{(k)}\right) = 0, \forall k \in \{0, \dots, K\} \end{cases}$$

2.2 APPROXIMATIONS UTILISEES

Nous allons expliquer ce que sont les classes de fonction. En effet il est important de comprendre cette notion car les approximations de dérivées s'appliquent pour certaines classes de fonction.

On commence par les fonctions d'une variable réelle.

Soit f une fonction dont l'ensemble de définition est un intervalle I inclus dans R.

Si cette fonction est dérivable sur I, donc dérivable en chaque point de I ainsi que sa dérivée alors elle est de classe C^1.

Pour les fonctions de plusieurs variables, on dit qu'une fonction est de classe C^1 si toutes ses dérivées partielles existent et sont continues.

Ainsi pour montrer qu'une fonction est de classe C^1, il faut au préalable calculer les dérivées partielles, et montrer qu'elles sont continues.

Nous allons illustrer cela sur un exemple :

$$f(x,y) = \begin{cases} \frac{x^2y^3}{x^2+y^2} si(x,y) \neq (0,0) \\ 0 \qquad\qquad \text{sinon} \end{cases}$$

On peut montrer que cette fonction est bien de classe C^1, en effet ses dérivées partielles sont toutes continues. Notons que nous sommes dans des espaces vectoriels de dimensions finies, on peut donc choisir la norme qui permet de démontrer la continuité. En effet toutes les normes sont équivalentes dans ces espaces.

Nous utilisons les propriétés suivantes, donnant une approximation de chacune des dérivées partielles présentes dans l'équation de Burgers :

$$\left| \partial_{t^{(k)}} u\left(x_i, t^{(k)}\right) - \frac{u\left(x_i, t^{(k+1)}\right) - u\left(x_i, t^{(k)}\right)}{\delta t} \right| \leq \frac{M_2}{2} \delta t,$$

avec $M_2 = \max_{k \in \{0,\dots,K\}} \left| \partial^2_{t^{(k)} t^{(k)}} u\left(x_i, t^{(k)}\right) \right|,$

valable pour u de classe C^2 sur un intervalle compact.

$$\left| \partial_{x_i} u\left(x_i, t^{(k)}\right) - \frac{u\left(x_{i+1}, t^{(k)}\right) - u\left(x_{i-1}, t^{(k)}\right)}{2h} \right| \leq \frac{M_3}{6} h^2,$$

avec $M_3 = \max_{i \in \{0,\dots,I\}} \left| \partial^3_{x_i x_i x_i} u\left(x_i, t^{(k)}\right) \right|,$

valable pour u de classe C^3 sur un intervalle compact.

$$\left|\partial^2_{x_i x_i} u\left(x_i, t^{(k)}\right) - \frac{u\left(x_{i+1}, t^{(k)}\right) - 2u\left(x_i, t^{(k)}\right) + u\left(x_{i-1}, t^{(k)}\right)}{h^2}\right| \leq \frac{M_4}{12} h^2,$$

$$avec\ M_4 = \max_{i\in\{0,\dots,I\}} \left|\partial^4_{x_i x_i x_i x_i} u\left(x_i, t^{(k)}\right)\right|,$$

valable pour u de classe C^4 sur un intervalle compact.

Ces approximations sont obtenues avec les formules de Taylor Lagrange.

2.3 CRANK-NICOLSON

Le schéma numérique de Crank-Nicolson s'obtient en prenant comme approximation des dérivés la somme de la moitié au temps k et de l'autre moitié au temps $k+1$.

On pose $u_i^{(k)} \approx u\left(x_i, t^{(k)}\right)$.

Le schéma de Crank-Nicolson s'écrit

$$(\mathcal{S}) \quad \begin{cases} \dfrac{u_i^{(k+1)} - u_i^{(k)}}{\delta t} + \dfrac{1}{2}\left(u_i^{(k)} \dfrac{u_{i+1}^{(k)} - u_{i-1}^{(k)}}{2h} + u_i^{(k+1)} \dfrac{u_{i+1}^{(k+1)} - u_{i-1}^{(k+1)}}{2h}\right) \\ = \dfrac{1}{2}\nu\left(\dfrac{u_{i+1}^{(k)} - 2u_i^{(k)} + u_{i-1}^{(k)}}{h^2} + \dfrac{u_{i+1}^{(k+1)} - 2u_i^{(k+1)} + u_{i-1}^{(k+1)}}{h^2}\right), \ \forall i \in \{1, \dots, I-1\}, \ \forall k \in \{0, \dots, K\} \\ u_i^{(0)} = u_0(x_i), \qquad \forall i \in \{0, \dots, I\} \\ u_0^{(k)} = u_I^{(k)} = 0, \qquad \forall k \in \{0, \dots, K\} \end{cases}$$

ce qui peut se réécrire (en rassemblant les termes en k+1 d'un côté et en k de l'autre)

$$(\mathcal{S}) \quad \begin{cases} \dfrac{u_i^{(k+1)}}{\delta t} + \dfrac{1}{2} u_i^{(k+1)} \dfrac{u_{i+1}^{(k+1)} - u_{i-1}^{(k+1)}}{2h} - \dfrac{1}{2}\nu \dfrac{u_{i+1}^{(k+1)} - 2u_i^{(k+1)} + u_{i-1}^{(k+1)}}{h^2} \\ = \dfrac{u_i^{(k)}}{\delta t} - \dfrac{1}{2} u_i^{(k)} \dfrac{u_{i+1}^{(k)} - u_{i-1}^{(k)}}{2h} + \dfrac{1}{2}\nu \dfrac{u_{i+1}^{(k)} - 2u_i^{(k)} + u_{i-1}^{(k)}}{h^2}, \ \forall i \in \{1, \dots, I-1\}, \ \forall k \in \{0, \dots, K\} \\ u_i^{(0)} = u_0(x_i), \qquad \forall i \in \{0, \dots, I\} \\ u_0^{(k)} = u_I^{(k)} = 0, \qquad \forall k \in \{0, \dots, K\} \end{cases}$$

ou encore (en rassemblant maintenant les termes en i, i+1 et i-1 puis en repassant tout à gauche)

$$(\mathcal{S}) \quad \begin{cases} \left(\frac{1}{\delta t} + \frac{u_{i+1}^{(k+1)} - u_{i-1}^{(k+1)}}{4h} + \frac{\nu}{h^2}\right) u_i^{(k+1)} - \frac{\nu}{2h^2} u_{i+1}^{(k+1)} - \frac{\nu}{2h^2} u_{i-1}^{(k+1)} \\ + \left(\frac{\nu}{h^2} + \frac{u_{i+1}^{(k)} - u_{i-1}^{(k)}}{4h} - \frac{1}{\delta t}\right) u_i^{(k)} - \frac{\nu}{2h^2} u_{i+1}^{(k)} - \frac{\nu}{2h^2} u_{i-1}^{(k)} = 0\,,\ \forall i \in \{1, \dots, I-1\},\ \forall k \in \{0, \dots, K\} \\ u_i^{(0)} = u_0(x_i),\ \forall i \in \{0, \dots, I\} \\ u_0^{(k)} = u_I^{(k)} = 0,\ \forall k \in \{0, \dots, K\} \end{cases}$$

3 METHODE DE NEWTON

3.1 PRESENTATION DE LA METHODE

La première équation peut se mettre sous la forme $F\left(U^{(k+1)}\right) = 0_{\mathbb{R}^{I-1}}$

avec $U^{(k+1)} = \left(u_1^{(k+1)}, \ \dots \ , u_{I-1}^{(k+1)}\right)$,

et $F\left(U^{(k+1)}\right) = \left(f_1\left(U^{(k+1)}\right), \ \dots \ , f_{I-1}\left(U^{(k+1)}\right)\right)$,

où $\forall i \in \{1, \ \dots \ , I-1\}, f_i\left(U^{(k+1)}\right) = \left(\frac{1}{\delta t} + \frac{u_{i+1}^{(k+1)} - u_{i-1}^{(k+1)}}{4h} + \frac{v}{h^2}\right) u_i^{(k+1)} - \frac{v}{2h^2} u_{i+1}^{(k+1)} - \frac{v}{2h^2} u_{i-1}^{(k+1)} + \left(\frac{v}{h^2} + \frac{u_{i+1}^{(k)} - u_{i-1}^{(k)}}{4h} - \frac{1}{\delta t}\right) u_i^{(k)} - \frac{v}{2h^2} u_{i+1}^{(k)} - \frac{v}{2h^2} u_{i-1}^{(k)}.$

Or, pour résoudre une équation du type $F(x) = 0_{\mathbb{R}^{I-1}}$ avec $F : \mathbb{R}^{I-1} \to \mathbb{R}^{I-1}$, on peut faire appel à la méthode de Newton.

Remarque : les $u_i^{(k)}, i \in \{0, \dots, I\}$ sont ici connus, donc sont considérés comme des constantes. En effet, on part de $U^{(0)} = (u_0(x_0), \dots, u_0(x_I))$ connu et on calcule les $U^{(k)}, k \in \{1, \dots, K\}$ avec la méthode de Newton.

Cette méthode s'écrit :

$$\begin{cases} V^{(0)} = U^{(k)} \\ J_F\left(V^{(n)}\right)\delta V = -F\left(V^{(n)}\right), \forall n > 0 \\ V^{(n+1)} = V^{(n)} + \delta V \end{cases}$$

Avec $J_F : \mathbb{R}^{I-1} \to \mathcal{M}_{I-1}(\mathbb{R})$ la Jacobienne de F et $V, \delta V \in \mathbb{R}^{I-1}$ des variables temporaires.

La première équation peut se récrire $\left(v_1^{(0)}, \dots, v_{I-1}^{(0)}\right) = \left(u_1^{(k)} \dots, u_{I-1}^{(k)}\right)$.

On applique cette méthode pour tout $k \in \{0, \dots, K-1\}$ afin de trouver $U^{(k+1)}$, qui se verra affecter la valeur $V^{(n)}$ pour n suffisamment grand pour que la méthode de Newton converge :

$$U^{(k+1)} = \lim_{n \to +\infty} V^{(n)}$$

3.2 CALCUL DE LA JACOBIENNE

Nous allons donner des explications sur la Jacobienne de manière générale. Cette matrice s'obtient grâce aux calculs de dérivées partielles. Prenons un exemple :

$$f(x, y) = (3x, 2y)$$

Voici la matrice jacobienne obtenue :

$$J = \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}$$

Si on considère une fonction de $\mathbb{R}^m$ dans $\mathbb{R}^n$ de manière générale, cette matrice a m lignes et n colonnes. Pour les calculs de Jacobienne, on dispose de plusieurs propriétés :

- Pour la composé de deux fonctions, la jacobienne est le produit des deux jacobiennes.
- Pour obtenir la jacobienne de la réciproque, on inverse la jacobienne de la fonction d'origine.

Dans notre problème, la Jacobienne de F est définie par :

$$J_F\left(V^{(n)}\right) = \begin{pmatrix} \partial_1 f_1\left(V^{(n)}\right) & \cdots & \partial_{I-1} f_1\left(V^{(n)}\right) \\ \vdots & \ddots & \vdots \\ \partial_1 f_{I-1}\left(V^{(n)}\right) & \cdots & \partial_{I-1} f_{I-1}\left(V^{(n)}\right) \end{pmatrix} \in \mathcal{M}_{I-1}(\mathbb{R})$$

Ici, on a

$$\forall i \in \{1, \dots, I-1\},\ f_i\left(V^{(n)}\right) = \left(\frac{1}{\delta t} + \frac{v_{i+1}^{(n)} - v_{i-1}^{(n)}}{4h} + \frac{\upsilon}{h^2}\right) u_i^{(n)} - \frac{\nu}{2h^2} v_{i+1}^{(n)} - \frac{\nu}{2h^2} v_{i-1}^{(n)}$$

$$+ \left(\frac{\upsilon}{h^2} + \frac{u_{i+1}^{(k)} - u_{i-1}^{(k)}}{4h} - \frac{1}{\delta t}\right) u_i^{(k)} - \frac{\upsilon}{2h^2} u_{i+1}^{(k)} - \frac{\upsilon}{2h^2} u_{i-1}^{(k)}.$$

Donc pour tout $i \in \{1, \dots, I-1\}$, on a

$\partial_j f_i\left(V^{(n)}\right) = 0\ si\ j \notin \{\mathrm{i}-1, \mathrm{i}, \mathrm{i}+1\}$

$$\partial_i f_i\left(V^{(n)}\right) = \frac{1}{\delta t} + \frac{v_{i+1}^{(n)} - v_{i-1}^{(n)}}{4h} + \frac{v}{h^2}$$

(cette formule ne pose pas de problème pour $i = 1$ et $i = I - 1$ car $v_0^{(n)} = v_I^{(n)} = 0$)

et

$$\partial_{i+1} f_i\left(V^{(n)}\right) = -\frac{\nu}{2h^2} + \frac{v_i^{(n)}}{4h}$$

$$\partial_{i-1} f_i\left(V^{(n)}\right) = -\frac{\nu}{2h^2} - \frac{v_i^{(n)}}{4h}.$$

Ainsi, notre Jacobienne ressemble à

$$J_F\left(V^{(n)}\right) = \begin{pmatrix} \frac{1}{\delta t} + \frac{v_2^{(n)} - v_0^{(n)}}{4h} + \frac{v}{h^2} & -\frac{v}{2h^2} + \frac{v_1^{(n)}}{4h} & 0 & \cdots & & & \cdots & 0 \\ -\frac{v}{2h^2} - \frac{v_2^{(n)}}{4h} & & \ddots & & & & & \vdots \\ 0 & \ddots & & & \ddots & \ddots & & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & 0 \\ & & & & & \ddots & & -\frac{v}{2h^2} + \frac{v_{I-2}^{(n)}}{4h} \\ 0 & \cdots & & & \cdots & 0 & -\frac{v}{2h^2} - \frac{v_{I-1}^{(n)}}{4h} & \frac{1}{\delta t} + \frac{v_I^{(n)} - v_{I-2}^{(n)}}{4h} + \frac{v}{h^2} \end{pmatrix}$$

4 IMPLEMENTATION INFORMATIQUE

Afin de vérifier que notre approximation est correcte, nous avons implémenté un programme en python (via le logiciel Spyder) permettant de visualiser sur un même graphique la solution exacte du problème et l'approximation que nous obtenons grâce au calcul ci-dessus.

4.1 RETRANSCRIPTION DES DONNEES

Le programme utilise les librairies *numpy* (pour les vecteurs, matrices et fonctions prédéfinies), *matplotlib.pyplot* (pour les graphiques) et *scipy.sparse.linalg.dsolve* (pour résoudre une équation du type Ax = B avec A une matrice, x et B un vecteur).

Figure 1 : Importation des librairies

```
#librairie necessaires
import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse.linalg.dsolve import spsolve
```

Nous commençons par déclarer toutes nos variables, à savoir la viscosité, la longueur du domaine, la durée mesurée, le nombre de points de discrétisation en espace et en temps ainsi que le pas de discrétisation en espace et en temps. On définit également un ϵ comme la différence minimale entre deux termes consécutifs dans la méthode de Newton pour sortir de la boucle. $m > 1$ est un paramètre de la fonction au temps initial et de la solution associée (données).

Figure 2 : Définition des paramètres

```
#Donnees
v = 0.01 #viscosite
L = 2 #longueur du domaine
I = 100 #Nombre de points de discretisation en espace
h = L/I #pas de discretisation en espace
T = 20 #duree mesuree
K = 1000 #Nombre de points de discretisation en temps
dt = T/K #pas de discretisation en temps
eps = 10**(-8) #seuil a obtenir pour la methode de Newton
m = 2 #m>1 parametre pour la condition initiale et la solution
```

Ensuite, nous écrivons les fonctions que nous allons utiliser. La première est la fonction u_0 qui renvoie la valeur de u au temps initiale $T = 0$. La deuxième est la solution finale qu'on est censé retrouver. Ces deux fonctions sont données. Elles prennent en paramètre un vecteur numpy qui peut être de taille variable et retourne un vecteur de même taille.

On déclare en dessous la fonction F, qui correspond à la fonction définie à la partie 3.1. Elle prend en paramètre deux vecteurs numpy de taille $I-1$: le $V^{(n)}$ de la méthode de Newton, et $U^{(k)}$ le terme précédent dans le temps. Enfin, la fonction J correspondant à la Jacobienne de F. Elle prend en paramètre le vecteur numpy $V^{(n)}$ de taille $I-1$ et renvoie une matrice numpy de taille $(I-1)\times(I-1)$. À noter que ces deux fonctions utilisent les valeurs $v_0^{(n)}$, $v_I^{(n)}$, $u_0^{(k)}$ et $u_I^{(k)}$ qui sont toutes nulles, mais doivent être utilisées afin d'écrire une formule générale pour tout $i \in \{1, \dots, I-1\}$. Il est donc nécessaire d'ajouter ces valeurs aux vecteurs passés en paramètre.

Figure 3 : Définition des fonctions

```
def u0(x): #condition initiale
    return 2*v*np.pi*np.sin(np.pi*x)/(m+np.cos(np.pi*x))

def sol(x,t): #solution exacte a retrouver
    return 2*v*np.pi*np.exp(-np.pi**2*v*t)*np.sin(np.pi*x)/(m+np.exp(-np.pi**2*v*t)*np.cos(np.pi*x))

def F(Uold, V): #R^(I-1)² --> R^(I-1)
    #On ajoute aux vecteurs les limites aux bords du domaine (=0)
    Uold = np.concatenate([0, Uold, 0], axis = None) #vecteur de dimension I+1
    V = np.concatenate([0, V, 0], axis = None) #vecteur de dimension I+1

    return (1/dt + (V[2:I+1] - V[0:I-1])/(4*h) + v/(h**2))*V[1:I] - v/(2*h**2)*V[2:I+1] - v/(2*h**2)*V[0:I-1] \
           + (v/(h**2) + (Uold[2:I+1] - Uold[0:I-1])/(4*h) - 1/dt)*Uold[1:I] - v/(2*h**2)*Uold[2:I+1] - v/(2*h**2)*Uold[0:I-1]

def J(V): #R^(I-1) --> M_(I-1)(R)
    #On ajoute au vecteur les limites aux bords du domaine (=0)
    V = np.concatenate([0, V, 0], axis = None) #vecteur de dimension I+1

    #Construction de la Jacobienne J(uk) de dimension (I-1,I-1)
    J = np.diag(1/dt + (V[2:I+1] - V[0:I-1])/(4*h) + v/h**2) #diagonale
    J += np.diag((-v/(2*h**2) - V[2:I]/(4*h))*np.ones(I-2), -1) #sous-diagonale
    J += np.diag((-v/(2*h**2) + V[1:I-1]/(4*h))*np.ones(I-2), 1) #sur-diagonale
    return J;
```

4.2 INITIALISATION DU PROGRAMME

On ne garde pas en mémoire tous les $U^{(k)}$ mais seulement le terme courant, qu'on appelle Unew, et le terme précédent Uold. Ces vecteurs seront toujours de taille $I-1$. En effet, les valeurs au bord sont connues (elles valent 0), donc il suffit pour l'affichage d'ajouter ces valeurs au bord comme dans les fonctions F et J. Pour le temps $T = 0$, on initialise Unew avec la fonction u_0 que l'on applique sur un vecteur comprenant $I-1$ valeurs réparties uniformément sur l'intervalle $]0, L[$.

Comme la fonction que l'on cherche à approximer dépend de deux paramètres (la position et le temps), on a choisi de la représenter par un graphe de u en fonction de la position, qui évolue

au cours du temps grâce à un affichage interactif. On trace sur ce graphe à la fois la solution exacte et l'approximation afin de mieux comparer. On garde également quelques affichages permanents de couleurs différentes. En même temps, dans la console, on affiche les résultats obtenus pour différents temps.

Figure 4 : Initialisation des variables

```
#vecteur u0
Unew = u0(np.arange(h,L,h)) #vecteur de dimension I-1
Uold = np.copy(Unew) #representera le terme precedent

x = np.arange(0,L+h,h) #vecteur de dimension I+1

#Mise en place du graphique
plt.figure(figsize=(12,8)) #taille de la fenetre
plt.title('Equation de Burgers')
plt.xlabel('position')
plt.ylabel('valeur de u', rotation=90)
plt.plot(x, u0(x), 'm', linewidth=3, label='fonction au temps initial')

#'''
#Mise en place du graphique interactif
lineSol, = plt.plot(x,u0(x), 'r', linewidth=3, label='solution')
lineApp, = plt.plot(x,np.concatenate([0, Unew, 0], axis = None), 'k--', linewidth=2, label='approximation')
#'''

plt.legend() #affichage de la legende

C = ['b','c--','y','g--','c','b--','g','y--'] #Couleurs des courbes
c = 0

print('U[0] = ',np.concatenate([0, Unew, 0], axis = None)) #affichage dans la console de de u au temps 0
print("\n")
```

4.3 APPLICATION DE LA METHODE

Pour calculer tous les autres termes $U^{(k)}$ jusqu'à $k = K$, on utilise une boucle *for*. Dans cette boucle, on va effectuer la méthode de Newton, ce qui donne lieu à une deuxième boucle *while* car on répète les mêmes opérations jusqu'à ce que la méthode de Newton converge, c'est-à-dire jusqu'à ce que la différence entre deux termes consécutifs soit inférieure à ϵ. On déclare avant d'entrer dans cette boucle une variable n pour compter le nombre d'itération, et δV qui est initialisé au vecteur $(1, \dots, 1)$ pour entrer dans le *while*, car δV est la différence entre deux termes consécutifs. On initialise $V^{(0)}$ à Uold puis on calcule dans la boucle les $V^{(n)}$ en suivant la méthode de Newton. On calcule ainsi tous les $U^{(k)}$ qui sont le résultat obtenu par la méthode de Newton ($U^{(k)} = \lim_{n \to +\infty} V^{(n)}$).

Figure 5 : Calcul des termes à chaque temps

```
for k in range(1,K+1): #boucle dans laquelle on va calculer les vecteurs uk
    n = 0 #compte le nombre d'itération pour que la méthode de Newton converge
    dV = np.ones(I-1) #pour rentrer dans la boucle du while
    V = np.copy(Uold) #On choisit comme point de depart pour la méthode de Newton le terme precedent (dans le temps)

    while max(abs(dV))>eps: #boucle dans laquelle on va effectuer les iterations de Newton
        dV = spsolve(J(V),-F(Uold,V)) #resolution de J*du = -F (multiplié par dt)
        V += dV
        n += 1

    Unew = V #le terme suivant correspond au resultat obtenu par la methode de Newton
    Uold = Unew #On garde en memoire U au temps precedent
```

Toujours dans la boucle for, on affiche alors pour certaines valeurs de k les résultats obtenus et les résultats qu'on est censé obtenir. On obtient l'erreur en calculant la norme de la différence entre l'approximation et la solution. On met également à jour le graphique interactif, et pour certaine valeur de k on garde une trace de la courbe.

Figure 6 : Affichages

```
    #Affichage de 8 itération de temps dans la console (reparties uniformement)
    if k%int(K/8)==0:
        print('temps t = ', k*dt)
        print('approximation : ', np.concatenate([0, Unew, 0], axis = None))
        print('solution : ', sol(x,k*dt))
        #norme de la différence entre l'approximation et la solution
        print('ecart : ', np.linalg.norm(np.concatenate([0, Unew, 0], axis = None) - sol(x,k*dt)))
        print('nb itération n = ', n)
        print('\n')

    #Mise a jour du graphique : 4 affichages persistants repartis uniformement dans le temps
    if k%int(K/4)==0:
        #affichage de la solution exacte (trait continu)
        plt.plot(x,sol(x,k*dt), C[c], linewidth=3, label='solution au temps t = %1.2f' %(k*dt))
        #affichage de l'approximation en pointillé
        plt.plot(x, np.concatenate([0, Unew, 0], axis = None),  C[c+1], linewidth=2, label='approximation au temps t = %1.2f' %(k*dt))
        plt.legend() #affichage de la legende
        c += 2

    #'''
    #graphique interactif
    lineSol.set_xdata(x)
    lineSol.set_ydata(sol(x,k*dt))
    lineApp.set_xdata(x)
    lineApp.set_ydata(np.concatenate([0, Unew, 0], axis = None))
    plt.pause(1/K) #plus K est grand, moins la pause entre deux iterations sera longue
    #'''
```

5 PRESENTATION DES RESULTATS

Dans notre programme, nous avons défini comme condition au temps initiale la fonction

$$u(x,0) = \frac{2\upsilon\pi \sin(\pi x)}{m+cos(\pi x)}, m > 1,$$

associée à la solution exacte donnée

$$u(x,t) = \frac{2\upsilon\pi e^{-\pi^2 \upsilon t} \sin(\pi x)}{m + e^{-\pi^2 \upsilon t} \cos(\pi x)}.$$

On peut comparer la solution avec les résultats obtenus par notre approximation en faisant varier la longueur du domaine (L), la durée mesuré (T), le nombre de points de discrétisation en espace (I) et en temps (K), le paramètre de la condition initiale (m), la viscosité (υ) et le seuil de la méthode de Newton (ϵ). Voici quelques exemples :

Figure 7 : Graphique de la solution et de l'approximation pour les paramètres $L = 1,\ T = 10,\ I = 1000,\ K = 1000,\ m = 2,\ \upsilon = 0.01,\ \epsilon = 10^{-8}$

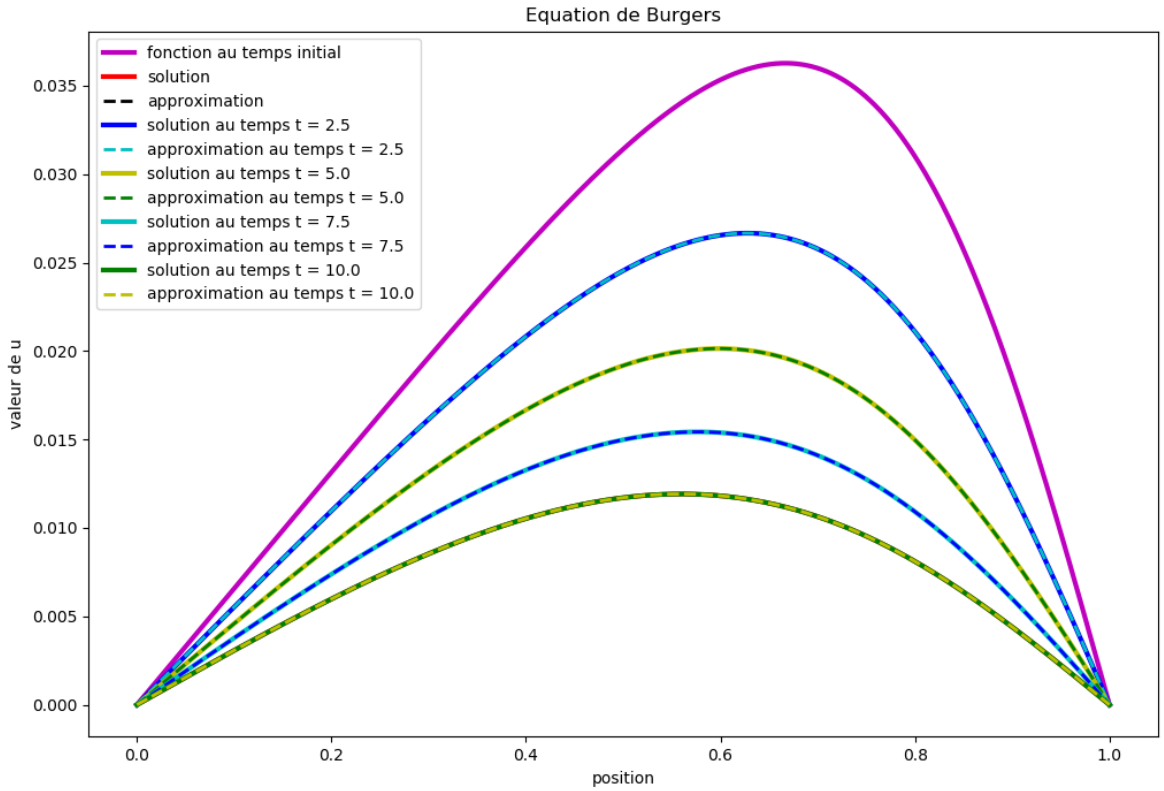

Figure 8 : Ecarts entre la solution de l'approximation à différents temps pour les paramètres $L = 1,\ T = 10,\ I = 1000,\ K = 1000,\ m = 2,\ \upsilon = 0.01,\ \epsilon = 10^{-8}$

Temps	$t = 2.5$	$t = 5$	$t = 7.5$	$t = 10$
$\|\|solution - approximation\|\|$	4.53×10^{-7}	4.60×10^{-7}	4.08×10^{-7}	3.10×10^{-7}

On observe une très faible différence entre la solution et l'approximation, quelque soit t. Ceci peut être dû au fait qu'on ait pris un grand nombre de points de discrétisation.

Figure 9 : Graphique de la solution et de l'approximation pour les paramètres $L = 1,\ T = 4,\ I = 40,\ K = 40,\ m = 2,\ \upsilon = 0.05,\ \epsilon = 10^{-8}$

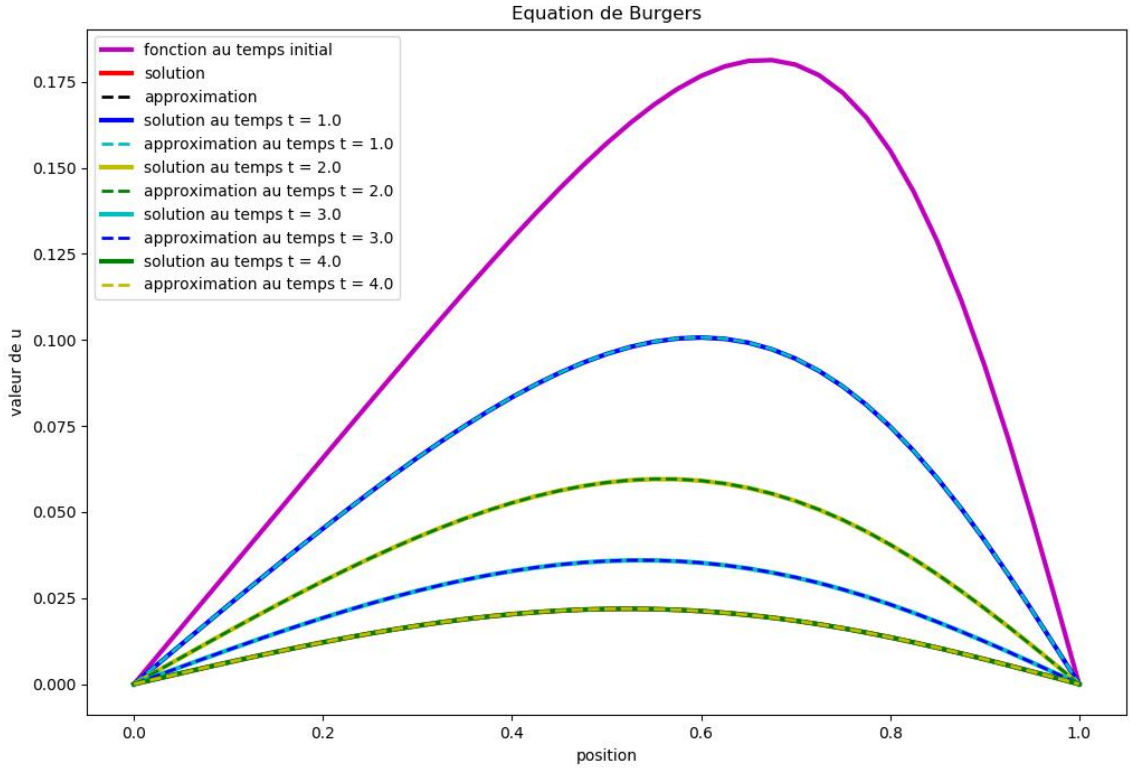

Figure 10 : Ecarts entre la solution de l'approximation à différents temps pour les paramètres $L = 1,\ T = 4,\ I = 40,\ K = 40,\ m = 2,\ \upsilon = 0.05,\ \epsilon = 10^{-8}$

Temps	$t = 1$	$t = 2$	$t = 3$	$t = 4$
$\|\|solution - approximation\|\|$	1.96×10^{-4}	1.47×10^{-4}	1.09×10^{-4}	8.00×10^{-5}

L'approximation est toujours très bonne en baissant le nombre de points de discrétisation. En augmentant la viscosité, on peut voir que la fonction prend des valeurs plus grandes mais décroit plus vite vers 0 avec le temps.

Figure 11 : Graphique de la solution et de l'approximation pour les paramètres $L = 1,\ T = 1,\ I = 8,\ K = 100,\ m = 5,\ \upsilon = 0.1,\ \epsilon = 10^{-8}$

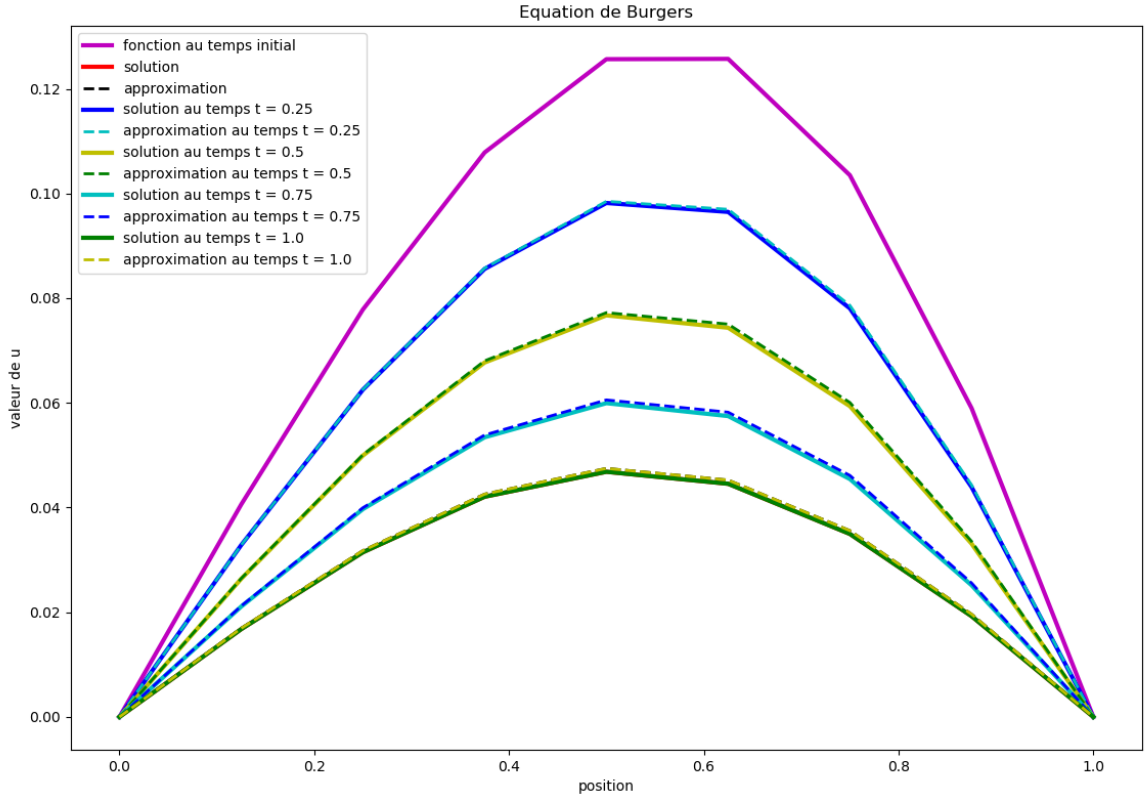

Figure 12 : Ecarts entre la solution de l'approximation à différents temps pour les paramètres $L = 1,\ T = 1,\ I = 8,\ K = 100,\ m = 5,\ \upsilon = 0.1,\ \epsilon = 10^{-8}$

Temps	$t = 0.24$	$t = 0.48$	$t = 0.72$	$t = 0.96$
$\|\|solution - approximation\|\|$	9.01×10^{-4}	1.24×10^{-3}	1.34×10^{-3}	1.35×10^{-3}

En choisissant un très petit nombre de points de discrétisation en espace, on peut observer sur le graphique un petit écart entre la solution et l'approximation. L'erreur reste tout de même de l'ordre de 10^{-3}. En augmentant m , on peut voir que la fonction décroit à partir d'une valeur de la position plus petite.

Figure 13 : Graphique de la solution et de l'approximation pour les paramètres $L = 4,\ T = 0.4,\ I = 100,\ K = 8,\ m = 5,\ \upsilon = 0.5,\ \epsilon = 10^{-8}$

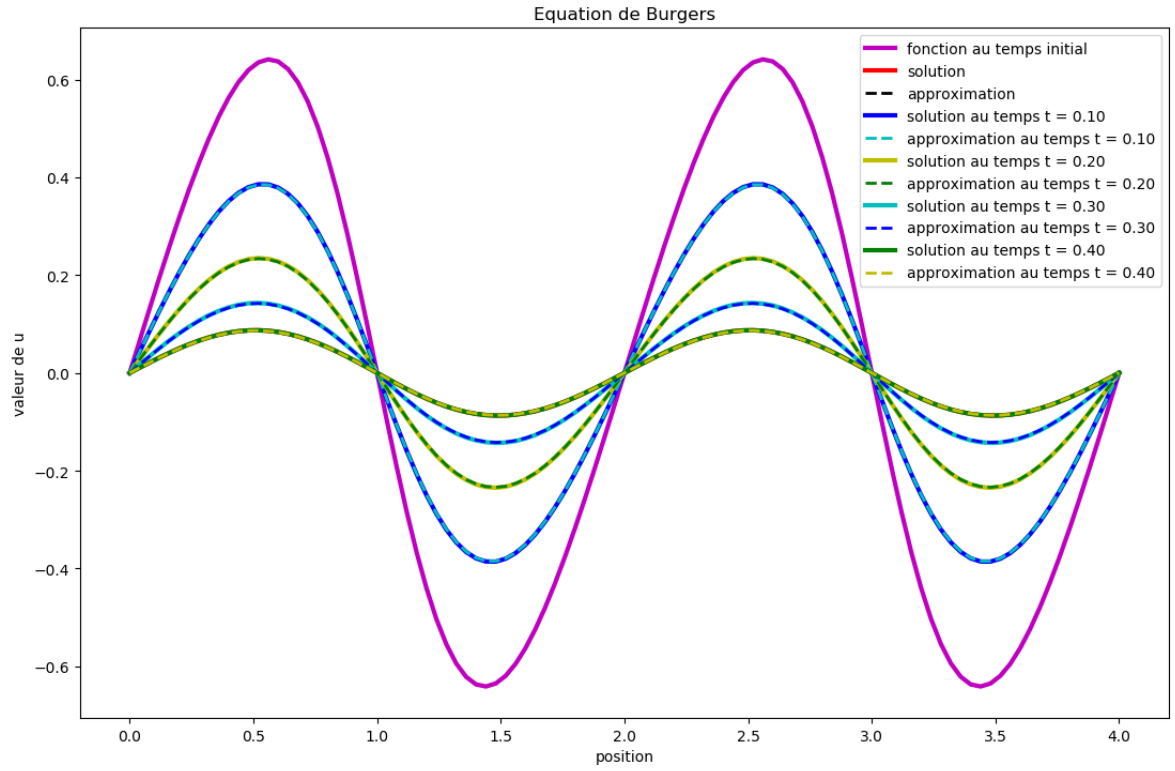

Figure 14 : Ecarts entre la solution de l'approximation à différents temps pour les paramètres $L = 4,\ T = 0.4,\ I = 100,\ K = 8,\ m = 5,\ \upsilon = 0.5,\ \epsilon = 10^{-8}$

Temps	$t = 0.10$	$t = 0.20$	$t = 0.30$	$t = 0.40$
$\|\|solution - approximation\|\|$	6.12×10^{-3}	6.87×10^{-3}	6.06×10^{-3}	4.85×10^{-3}

La fonction solution du système est 2-périodique par rapport à la position, on peut le voir en augmentant la longueur du domaine. Cela ne se voit pas sur le graphique mais choisir un très petit nombre de points de discrétisation en temps génère aussi un écart avec la solution plus élevé.

Figure 15 : Graphique de la solution et de l'approximation pour les paramètres $L = 2,\ T = 20,\ I = 1000,\ K = 320,\ m = 2,\ \upsilon = 0.01,\ \epsilon = 10^{-15}$

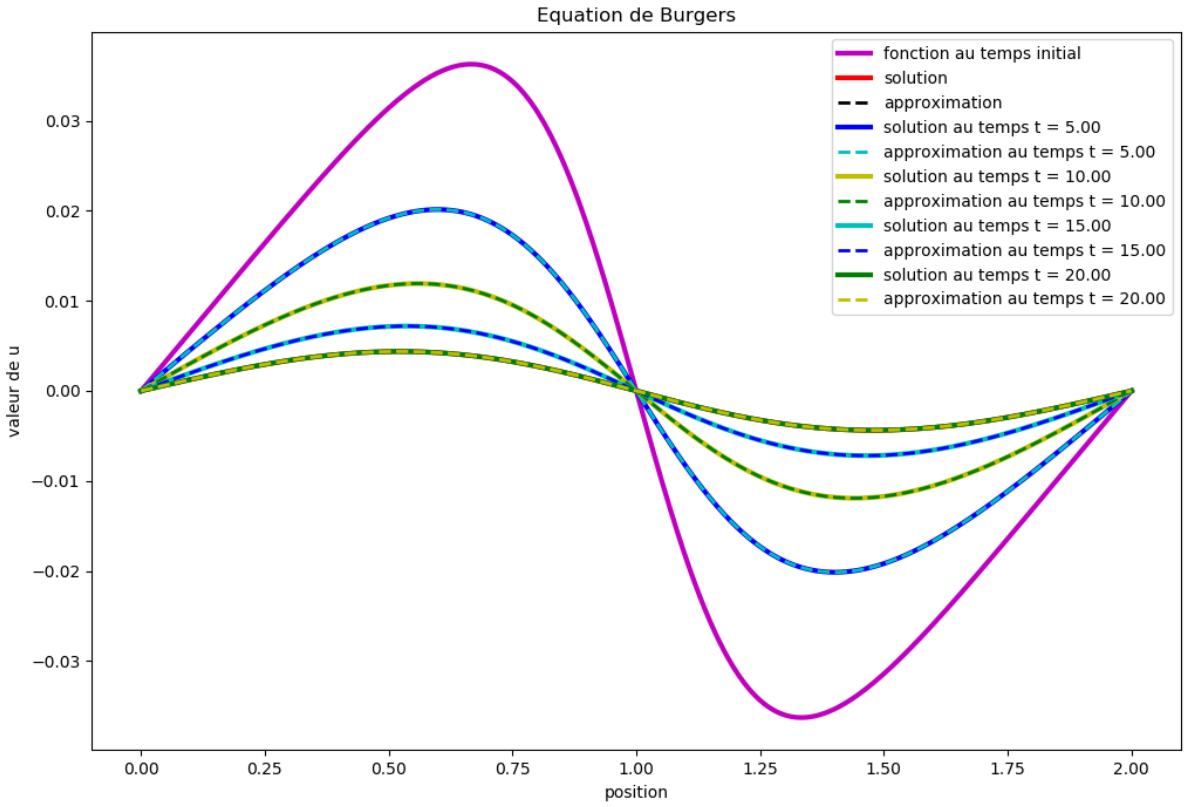

Figure 16 : Ecarts entre la solution de l'approximation à différents temps pour les paramètres $L = 2,\ T = 20,\ I = 1000,\ K = 320,\ m = 2,\ \upsilon = 0.01,\ \epsilon = 10^{-15}$

Temps	$t = 5$	$t = 10$	$t = 15$	$t = 20$
$\|solution - approximation\|$	2.54×10^{-7}	1.37×10^{-7}	5.89×10^{-8}	2.30×10^{-8}

En augmentant les points de discrétisation, l'écart avec la solution est de l'ordre de 10^{-7} voir 10^{-8}. Même en diminuant la valeur de ϵ, on remarque que le nombre d'itérations dans les méthodes de Newton utilisées à chaque itération de temps est minime (ici toujours égale à 3). Cela montre que la méthode de Newton sur ce problème converge très rapidement.

CONCLUSION

L'équation de Burgers peut être résolue à l'aide de différents outils mathématiques. Ici, nous avons d'abord écrit la forme discrétisée du problème. Nous avons ensuite utilisé des approximations de dérivée obtenues à partir des équations de Taylor, en vue d'obtenir un schéma de Crank-Nicolson en temps et centré d'ordre 2 en espace. A partir de ce schéma, nous avons pu mettre l'équation sous la forme d'une fonction qu'on cherche à annuler. Nous l'avons alors résolue en utilisant la méthode de Newton. Enfin, nous sommes passés à l'implémentation en Python de ce problème afin de résoudre numériquement l'équation de Burgers, puis nous avons pu comparer nos résultats avec une solution exacte du problème.

Les résultats sont satisfaisants : la fonction obtenue converge bel et bien vers la solution exacte au fur et à mesure qu'on augmente le nombre de points de discrétisation. De plus, cette convergence est très rapide. En effet, il n'est pas nécessaire d'avoir un très grand nombre de points de discrétisation pour avoir une approximation correcte. L'erreur entre l'approximation obtenue et la solution est négligeable pour un nombre de points de discrétisation suffisant.

Ce projet a permis de se familiariser avec une équation au dérivée partielle particulière. Cela nous a fait approfondir les connaissances acquises en cours de différences finies. Nous avons également utilisé des théorèmes mathématiques vus précédemment tout au long de notre cursus. Résoudre une équation de ce type est donc un bon exercice, reprenant beaucoup de points essentiels à connaître en mathématiques appliquées.

Dans notre programme, nous avons testé notre approximation avec une seule condition initiale, associée à une solution connu. Il existe d'autres solutions à ce problème associées chacune à des conditions initiales différentes. D'autre part, la position x est un réel tout au long du projet, ce qui signifie que le domaine étudié est un simple segment. Une suite possible du projet aurait été d'étudier le problème en dimension supérieure.

BIBLIOGRAPHIE

Sources utilisées pour écrire l'introduction :

- https://fr.wikipedia.org/wiki/%C3%89quation_de_Burgers

  Dernière mise à jour de la page : 2 mars 2022
  Dernière consultation : 16 mars 2023

- https://history.aip.org/phn/11806008.html
  Dernière mise à jour de la page : 2023
  Dernière consultation : 16 mars 2023

- https://fr.wikipedia.org/wiki/M%C3%A9daille_Bingham

  Dernière mise à jour de la page : 30 juillet 2018
  Dernière consultation : 16 mars 2023

Sources utilisées pour écrire en LaTeX :

- https://devmath.fr/tools/latex-symbols-list/

  Dernière mise à jour de la page : 2022
  Dernière consultation : 16 mars 2023

- https://www.math-linux.com/latex-4/faq/latex-faq/article/latex-derivee-limite-somme-produit-et-integrale

  Dernière mise à jour de la page : 5 décembre 2020
  Dernière consultation : 16 mars 2023

Sources utilisées pour écrire en Python :

- https://stackoverflow.com/questions/43757820/how-to-add-a-variable-to-python-plt-title

  Dernière mise à jour de la page : 9 décembre 2019
  Dernière consultation : 16 mars 2023

- https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html

  Dernière mise à jour de la page : 2023
  Dernière consultation : 16 mars 2023

- http://www.python-simple.com/python-matplotlib/couleurs-matplotlib.php

  Dernière mise à jour de la page : 19 février 2023
  Dernière consultation : 16 mars 2023

- https://stackoverflow.com/questions/12978518/scipy-sparse-dia-matrix-solver

  Dernière mise à jour de la page : 17 novembre 2012
  Dernière consultation : 16 mars 2023