Analyse Nmérique TP1

DUQUENOY Cyrile *

Université d'Aix-Marseille — L3 de Mathématiques
2021-2022 — Second semestre

tp1, Analyse Numérique : Systèmes Tridiagonaux

1 Exercice 1

On considère l'équation de Poisson sur $[0,1]$ donnéé par :

$$-u'' = f,\ u(0) = u(1) = 0,$$

que l'on discrétise avec le schéma aux différences finies :

$$-\tfrac{u_{j+1}-2u_j+u_{j-1}}{h^2} = f(x_j)$$

On prends ici $f(x) = x(1-x)$.

1.1 Rappel de Cours : discrétisation de l'équation de la chaleur

Soit $f \in C([0,1])$. On cherche u tel que :

$$\begin{gathered} -u''(x) = f(x),\ x \in]0,1[, \\ u(0) = u(1) = 0 \end{gathered}$$

*Université Aix Marseille

On prends un maillage avec $x_i = ih$ et $h = 1/N$.

On cherche $u_i \simeq u(x_i)$.
$u''(x_i) \simeq a_0 u(x_i) + a_1 u(x_{i+1}) + a_2 u(x_{i-1})$.

Développement de Taylor :
$u(x_{i+1}) = u(x_i + h) = u(x_i) + hu'(x_i) + \frac{h^2}{2} u''(x_i) + \frac{h^3}{6} u^{(3)}(x_i) + O(h^4)$.

$u(x_{i-h}) = u(x_i - h) = u(x_i) - hu'(x_i) + \frac{h^2}{2} u''(x_i) - \frac{h^3}{6} u^{(3)}(x_i) + O(h^4)$.

$u(x_{i+1}) + u(x_{i-1}) = 2u(x_i) + h^2 u''(x_1) + O(h^4)$.

On a alors :

$$u''(x_1) = \frac{u(x_{i+1}) + u(x_{i-1}) - 2u(x_i)}{h^2} + O(h^2)$$

On peut alors définir le schéma numérique :

$$-u_{i+1} + 2u_i - u_{i-1} = h^2 f(x_i),\ i \in [1, N-1].$$

On définit ensuite $AU_h = F_h$ avec A matrice carré d'ordre $N-1$, $U_h = (u_1, ..., u_{N-1})^t$ et $F_h = (f(x_1), ..., f(x_{N-1}))^t$, où A est tridiagonale. La diagonale de A est constituée de 2, les deux sous-diagonales sont constituées de -1.

1.1.1 Solution exacte

On a $f(x) = x(1-x) = -u''$.
La solution exacte est donnée en intégrant 2 fois $-f$.
Ce qui donne, avec les conditions initiales, $u(0) = u(1) = 0$: $u(x) = \frac{1}{12}(x^4 - 2x^3 + x)$.

1.1.2 Schéma numérique à l'aide de python

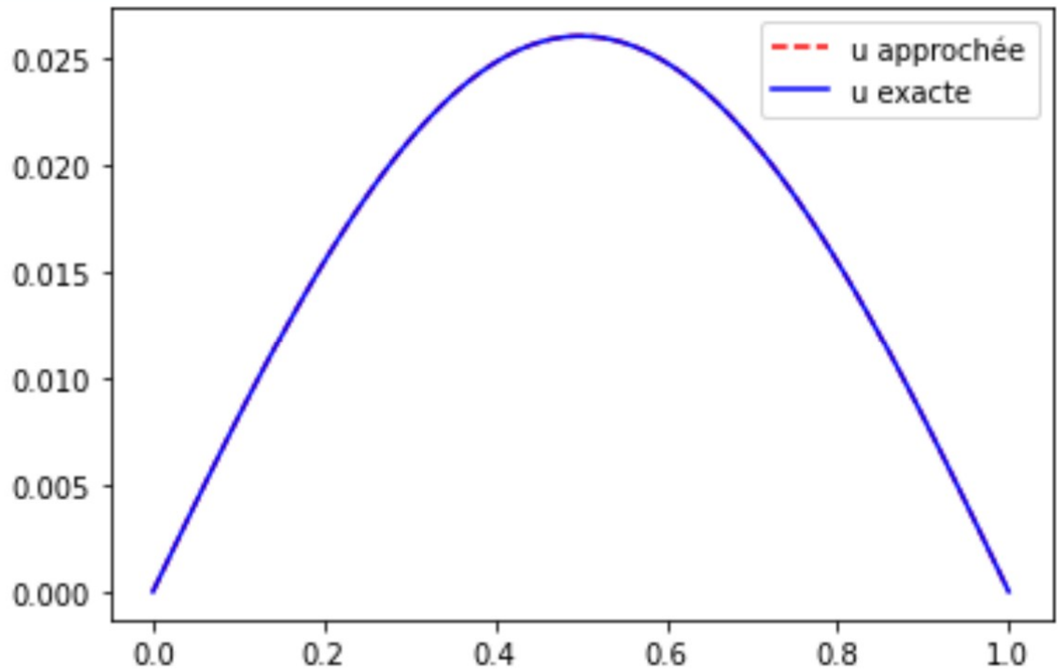
A l'aide de la solution exacte et de la solution approchée, on va calculer l'erreur numérique et comparer avec l'estimation :

$$\|e_h\|_{h,\infty} \leq \frac{h^2}{96} \|f''\|_\infty.$$

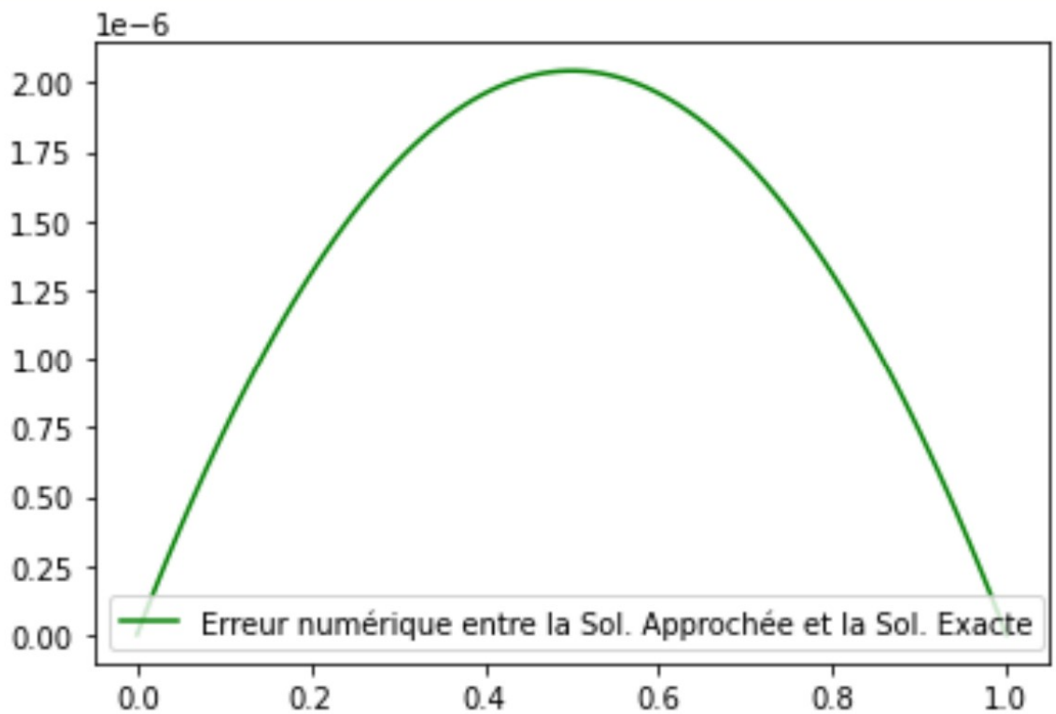
Ci-dessous le code python :

```python
import numpy as np
import math
import matplotlib.pyplot as plt

N=102
h=1/(N-1)
def Mat(N):
    #h=1/N
    d=(2/(h*h))*np.ones(N-2)
    d1= (-1/(h*h))*np.ones(N-3)
    A=np.diag(d,0)+np.diag(d1,1)+np.diag(d1,-1)
    return A

#Notre fonction f
def f(x):
    return x*(1-x)

#Calcul de la solution exacte
def sol_exacte (t):
    z=(1/12)*(t**4 - 2*(t**3) + t)
    return z

def tab(N):
    i=h
    x=0
    y=[]
    while i< (N-1)*(h) :
        y.append(i)
        i=i+h
    return y

x=tab(N)
print('x : ', x)
print('')
y=np.linspace(0,1,N)
print('y : ', y)

def schema(A):
    F=[]
    sol=[]
    for k in x:
        F.append(f(k))

    for k in y :
        sol.append(sol_exacte(k))
    A=np.linalg.inv(Mat(N))
    U=np.dot(F,A)
    V=[0]
    for k in U :
        V.append(k)
    V.append(0)
    U=V
    return U,sol


def erreur(s1,s2):
    E=[]
    t=0
    for k in s1:
        E.append(k- s2[t])
        t+=1
    return E

A=Mat(N)
U=schema(A)[0]
sol=schema(A)[1]

print('U : ', U)
print('')
print('Sol. Exacte : ', sol)


E=erreur(U, sol)

plt.plot(y,schema(A)[0], '--r', label='u approchée')
plt.legend()
plt.plot(y,schema(A)[1], '-b', label='u exacte')
```

On obtient alors le graphe suivant :



On calcule ensuite l'erreur numérique, (on en donne le graphe) :



1.2 On prends maintenant $f(x) = sin(p\pi x)$, $p \in \mathbb{N}^*$

1.2.1 Solution exacte

De la même manière que pour la fonction précédente,

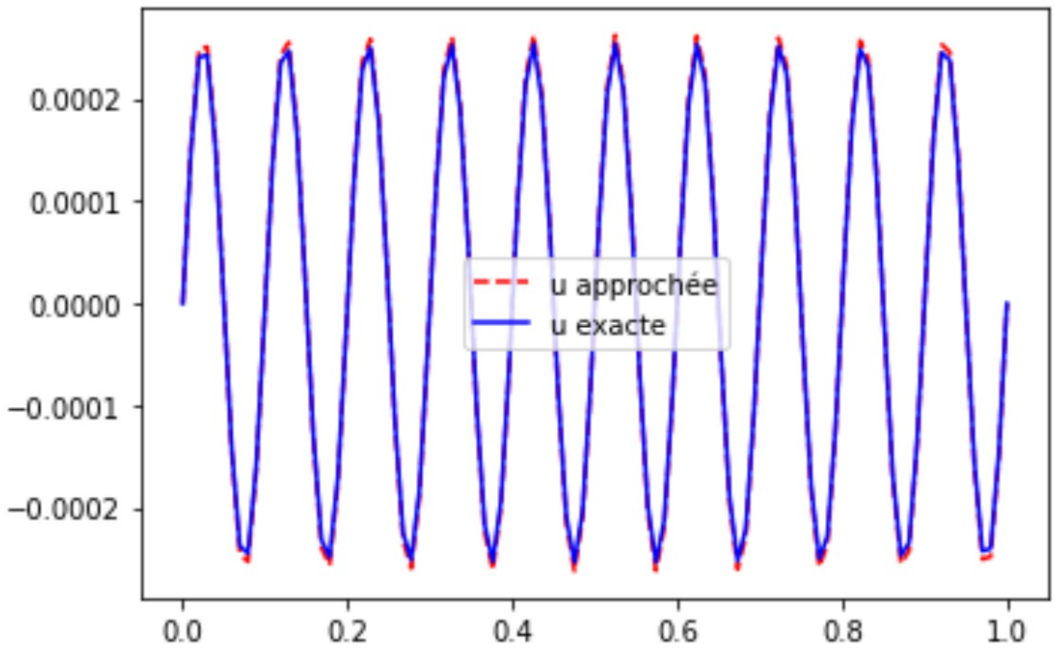
$$u(x) = \frac{sin(p\pi x)}{(p\pi)^2}.$$

1.2.2 Schéma numérique

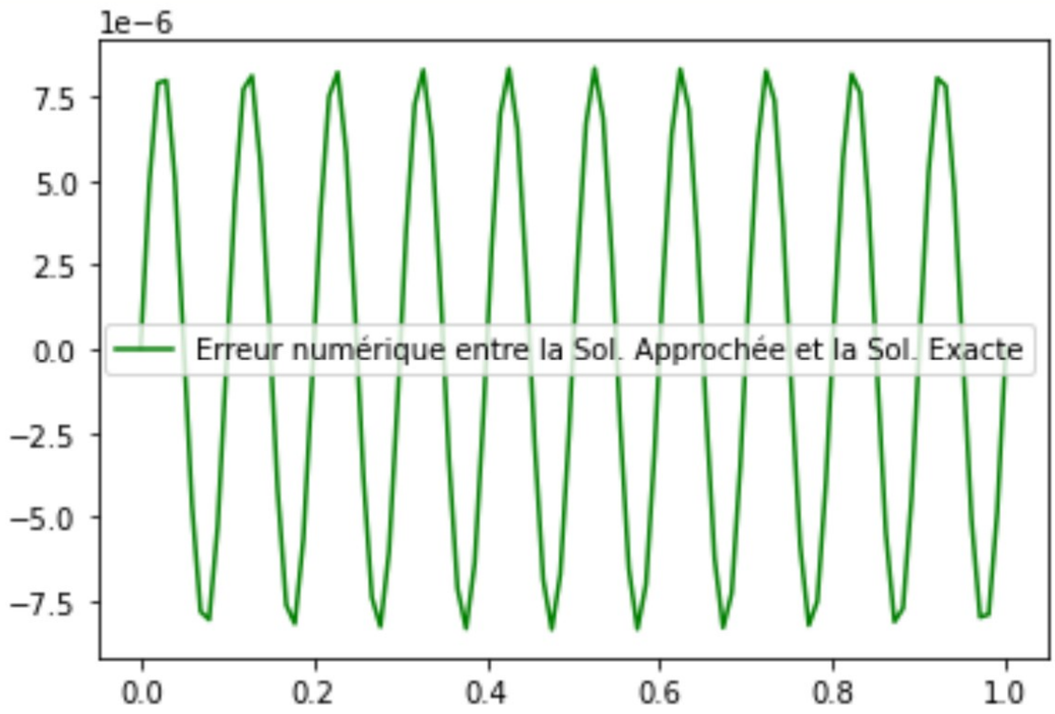
Ci-dessous le code python :

```python
def f2(t,p):
    return math.sin((math.pi)*p*t)

def sol2(p,t):
    return (1/((p*math.pi)*(p*math.pi)))*(math.sin(p*(math.pi)*t))

def schema1(A,p):
    F=[]
    sol=[]
    for k in x:
        F.append(f2(k,p))

    for k in y :
        sol.append(sol2(p,k))
    A=np.linalg.inv(Mat(N))
    U=np.dot(F,A)
    V=[0]
    for k in U :
        V.append(k)
    V.append(0)
    U=V
    return U,sol

p=20
A=Mat(N)
U1=schema1(A,p)[0]
sol1=schema1(A,p)[1]

print('U1 : ', U1)
print('')
print('Sol1. Exacte : ', sol1)

E1=erreur(U1, sol1)

plt.plot(y,schema1(A,p)[0], '--r', label='u approchée')
plt.legend()
plt.plot(y,schema1(A,p)[1], '-b', label='u exacte')
plt.legend()
plt.show()

plt.plot(y,E1, '-g', label='Erreur numérique entre la Sol. Approchée et la Sol. Exacte')
plt.legend()
plt.show()
```

On obtient alors le graphe suivant :



On calcule ensuite l'erreur numérique, (on en donne le graphe) :



1.3 Comparatif erreur numérique avec $\|e_h\|_{h,\infty} \leq \frac{h^2}{96}\|f''\|_\infty$.

Pour $f(x) = x(1-x)$, la norme infini de f'' est 2. Ainsi $\frac{h^2}{96}\|f''\|_\infty = \frac{h^2}{48}$
On vérifie l'inégalité à l'aide de cette fonction python :

```
#_Compare_#
def compare(E):
    print('')
    for k in E:
        if k > (h*h)/48 :
            print('Faux')
            return False
        else :
            print('Vrai')
            return True
```

Le résultat est 'Vrai'. L'inégalité est vérifiée.

Pour $f(x) = sin(p\pi x)$, la norme infini de f'' est majorée par $(p\pi)^2$. Ainsi $\frac{h^2}{96}\|f''\|_\infty = \frac{(hp\pi)^2}{96}$.
Le résultat est 'Vrai'. L'inégalité est vérifiée.

Conclusion : L'erreur de consistance tend vers 0 comme h^2. Le schéma est consistant d'ordre 2.

1.4 Un second problème

On considère maintenant le problème :

$$-u'' + ku = f$$

$$u(0) = u(1) = 0$$

avec $k \in \mathbb{R}^+$.

1.4.1 Résultats pour $f(x) = x(1-x)$

1.4.2 Schéma numérique

2 Exercice 2 : Factorisation LU d'une matrice tridiagonale

On considère A une matrice carré tridiagonale dont la diagonale est $a = (a_1, ..., a_n)$, la diagonale supérieure est $b = (b_1, ..., b_{n-1})$ et la diagonale inférieure est $c = (c_1, ..., c_{n-1})$.
Le but est de factoriser A sous la forme LU (factorisation LU).

On détermine un algorithme pouvant y parvenir. C'est l'algorithme dit de Thomas.

On veut $A = LU$.
On note $g = (g_1, ..., g_n)$ diagonale de U, $h = (h_1, ..., h_{n-1})$ la diagonale supérieure de U, $f = (f_1, ... f_{n-1})$ la sous diagonale de L.
Le produit LU nous amène à :

$$\begin{gathered} h_i = b_i \text{ pour } i = 1, ..., n-1 \\ g_1 = a_1 \\ g_i = a_1 - b_{i-1}\frac{c_{i-1}}{g_{i-1}} \text{ pour } i = 2, ..., n \\ f_i = \frac{c_i}{g_i} \text{ pour } i = 1, ..., n-1 \end{gathered}$$

Numériquement, cela revient à créer à l'aide de python une fonction LU qui dépend de a,b,c qui va construire L et U à l'aide de l'algorithme que l'on à donné ci-dessus, puis à renvoyer L,U. On s'assura que la fonction fonctionne correctement en comparant A avec LU où A sera généré de manière aléatoire. i.e. a,b,c générés aléatoirement.

```python
import numpy as np
import math
import matplotlib.pyplot as plt
from math import log
from math import exp
from math import sqrt

N=3

a=np.random.rand(N)
b=np.random.rand(N-1)
c=np.random.rand(N-1)

def LU(a,b,c):  #Complexité en O(n)
    N=np.size(a)
    U,L=[]*2,[]*2
    f=[]
    g=[]
    h=[]
    #Construction du U
    for k in range(len(b)):
        h.append(b[k])
    for k in range(len(a)):
        if k==0:
            g.append(a[k])
        else :
            g.append(a[k] - ( b[k-1] * (c[k-1] / g[k-1])))
    U=np.diag(g)+np.diag(h,1)
    #Construction de L
    x=[1]*N
    for k in range(len(c)):
        f.append(  (c[k]) / (g[k])   )
    L=np.diag(x)+np.diag(f,-1)
    return L,U

L,U=LU(a,b,c)
A=np.diag(a)+np.diag(b,1)+np.diag(c,-1)

#print(A, np.dot(L,U))

print('A-LU : ', A-np.dot(L,U))
print()
```

```
A-LU :  [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

Le code est correct, on peut remarquer au passage que la complexité de LU est de $O(n)$.

2.1 Algorithme de remontée

On se place dans le cas où l'on doit résoudre un système linéaire de la forme $Ux = y$, avec U de même forme que U dans la factorisation LU.

En notant $x = (x_1, ..., x_n)$ et $y = (y_1, ..., y_n)$, on a :

$$x_i = y_i - \frac{h_i x_{i+1}}{g_i} \text{ pour } i = 1, ..., n-1$$
$$x_n = \frac{y_n}{g_n}$$

Numériquement, cela revient à créer une fonction 'remonte' qui dépend de U et y qui va exploiter l'algorithme donné juste au dessus pour retourner x. Avec y et U générés aléatoirement.

```python
def remonte (U, y ) :
    g=[]
    h=[]
    for k in range(0,N-1):
        h.append(U[k][k+1])
    #del h[0]
    print('h :', h)
    print('')
    for k in range(len(y)):
        g.append(U[k][k])

    n = np.size(g)
    x = [1] * n
    x[n-1] = y[n-1] / g[n-1]
    for i in range  (n-2 , -1 , -1) :
        x[i] = ( y[i] - x[i+1] * h[i] ) / g[i]
    return x


print('U', U)
x=remonte(U,y)
print('x :',x)
print('')
```

2.2 Mise en application

Reprenons la matrice du Laplacien vu dans le premier exercice avec l'équation de la chaleur. (On travaillera sur des matrices de taillle 3x3 pour plus de lisibilité dans les print sur le terminal).

Ci-dessous le code Python permettant la mise en application des algorithmes de descente et de remontée sur la matrice du Laplacien. Le vecteur y est donnée aléatoirement. Le but est de résoudre $Ax = y$. On cherche donc x.

```
#________________________________Application_____________________________#
print('APPLICATION')
print('')


h=1/(N-1)

def Laplacien(a,b,c,y):
    print('y :', y)
    L,u=LU(a,b,c)
    Y=np.dot(np.linalg.inv(L),y)
    n=np.size(a)
    g=[]
    for i in range(0,n):
        g.append(U[i][i])
    h=[]
    for k in range(0,n-1):
        h.append(U[k][k+1])
    X=remonte(U,Y)
    X1=np.dot(L,U)
    print('Véif Calcul')
    print('On tombe bien sur le y choisi')
    print('Vérif LUX=Y :', np.dot(X1,X))
    return X

#------------------------------------TEST-----------------------------------#

y=[1,2,3]
d=(2/(h*h))*np.ones(N)
d1= (-1/(h*h))*np.ones(N-1)
d2= (-1/(h*h))*np.ones(N-1)

print(Laplacien(d,d1,d2,y))
print('')
```

On teste ensuite le code en s'assurant que le produit LUx vaut bien Ax.
Avec les résultats obtenus :

```
APPLICATION

y : [1, 2, 3]
h : [0.2280904811083002, 0.3057239014605102]

Véif Calcul
On tombe bien sur le y choisi
Vérif LUX=Y : [1. 2. 3.]
[-15.965993271413991, 54.26579390199927, -4.542886454740309]
```

2.3 Inverse d'une matrice sans utiliser np.linalg.inv()

On a vu que A pouvait se factoriser de la forme LU. Comme A est inversible, alors $A^{-1} = U^{-1}L^{-1}$.
On peut alors construire 3 fonctions python qui vont inverser U,V pour les deux premières, la troisième inverser A en faisant le produit $U^{-1}L^{-1}$.
Ci-dessous le code python :

```python
#___________________________Inverse_Matrice _Sans _Numpy_____________________#
def inverse_L(L):
    n=np.size(L[0])
    f=[]
    for i in range (0,n-1):
        f.append(L[i+1][i])
    I=np.zeros((n,n))+np.diag(np.ones(n))
    for j in range(1,n):
        I[j]=I[j]-[i*f[j-1] for i in I[j-1]]
    print("vérification de l'inverse")
    print(np.dot(L,I))
    return I

def inverse_U(U):
    n=np.size(U[0])
    g=[]
    for i in range (0,n):
        g.append(U[i][i])
    h=[]
    for k in range (0,n-1):
        h.append(U[k][k+1])
    I=np.zeros((n,n))+np.diag(np.ones(n))
    I[n-1]=[r/g[n-1] for r in I[n-1]]
    for j in range(n-2,-1,-1):
        I[j]=np.array([p/g[j] for p in I[j]])-[i*(h[j]/g[j]) for i in I[j+1]]
    print("vérification de l'inverse")
    print(np.dot(U,I))
    return I

def inverse_A(a,b,c):
    n=np.size(a)
    A=np.zeros((n,n))+np.diag(a)+np.diag(b,1)+np.diag(c,-1)
    L,U = LU(a,b,c)
    inv_u=inverse_U(U)
    inv_l=inverse_L(L)
    inv_a=np.dot(inv_u,inv_l)
    print('A')
    print(A)
    print("vérification de l'inverse")
    print(np.dot(A,inv_a))
    print()
    print("l'inverse de A est")
    return inv_a

print(inverse_A(d,d1,d2))
```

```
vérification de l'inverse
[[ 1.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  1.00000000e+00 -1.11022302e-16]
 [ 0.00000000e+00  0.00000000e+00  1.00000000e+00]]
vérification de l'inverse
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
A
[[ 8. -4.  0.]
 [-4.  8. -4.]
 [ 0. -4.  8.]]
vérification de l'inverse
[[ 1.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 5.55111512e-17  1.00000000e+00 -1.11022302e-16]
 [-5.55111512e-17 -1.11022302e-16  1.00000000e+00]]

l'inverse de A est
[[0.1875 0.125  0.0625]
 [0.125  0.25   0.125 ]
 [0.0625 0.125  0.1875]]
```

2.3.1 Remarque

On aurai pu utiliser cette fonction tiré d'internet qui retourne directement l'inverse d'une matrice. Comme il est tiré d'internet, nous l'avons pas utilisé.

```python
#-------------INVERSE---------------------#

def inverse(M,I):
    for fd in range(len(M)): #Pour chaque colonne de M
        fdScaler = 1.0/M[fd][fd] #Scalaire qui va diviser notre ligne
        for j in range(len(M)): #pour chaque colonne de M
            M[fd][j] *= fdScaler #On multiplie les coeff de la ligne par notre scalaire
            I[fd][j] *= fdScaler #De même pour l'identité
        for i in list(range(len(M)))[0:fd] + list(range(len(M)))[fd+1:]: #On parcours [fd+1,fd+2,...,2*fd]
            crScaler = M[i][fd]
            for j in range(len(M)):
                M[i][j]=M[i][j] - crScaler * M[fd][j]
                I[i][j]=I[i][j] - crScaler * I[fd][j]
    return I
```