Analyse numérique (XMS1ME090) – 2026-2027 – TP1 – Partie 1
Initiation et/ou rappels Python
Méthode des différences finies
Equation de Poisson 1D

Chaque programme python commencera par des appels aux bibliothèques qui seront utilisées (par exemple NumPy). On trouve dans le module NumPy les outils de manipulation de tableaux **et** de matrices. Il s'agit d'un module stable, bien testé et relativement bien documenté : `http://docs.scipy.org/doc/`, `http://docs.scipy.org/doc/numpy/reference/`

Pour l'importer, on recommande d'utiliser

```
import numpy as np
```

On aura aussi besoin des modules LinAlg et Random, qu'on recommande d'importer en utilisant la commande suivante :

```
import numpy.linalg as la
import numpy.random as rd
```

Toutes les fonctions numpy.linalg (resp. numpy.random) seront alors préfixées par la (resp. rd).

1 Les commentaires, les affichages

Il est important de pouvoir écrire des commentaires lors de l'élaboration d'un programme. Pour cela, sur une ligne, tout ce qui après le symbole # est non lu.

```
np.pi       # pour connaitre la valeur de pi
y = np.pi   # affecte pi à y
```

De même, il est important d'afficher des informations au cours de l'exécution d'un programme. Pour cela, on utilise la fonction **print**.

```
print('La valeur de y est : ',y)
[Sortie Python] La valeur de y est : 3.141592653589793
```

Pour choisir le nombre de décimales affichées, on utilise la fonction **format**.

```
print('La valeur de pi avec 7 décimales est', '{:.7f}'.format(np.pi))
[Sortie Python] La valeur de pi avec 7 décimales est 3.1415927

print('La valeur de pi en notation scientifique est', '{:.7e}'.format(np.pi))
[Sortie Python] La valeur de pi en notation scientifique est 3.1415927e+00
```

2 Variables prédéfinies

Par exemple **pi** dans la librairie NumPy :

```
np.pi
[Sortie Python] 3.141592653589793
```

et aussi des variables relatives à la représentation des réels. Python utilise un codage double précision des réels : codage binaire sur 64 bits (1 pour le signe, 11 pour l'exposant, 52 pour la mantisse), ce qui donne la représentation d'un réel x :

$$x = (\pm 1)\, 2^e \left(a_0 + a_1 \frac{1}{2} + a_2 \frac{1}{2^2} + \cdots + a_{52} \frac{1}{2^{52}} \right)$$

avec $a_0 = 1$, a_1,...,a_{52} égaux à 0 ou 1, et l'exposant $e \in [-2^{10}, 2^{10} - 1] = [-1024, 1023]$.

```
print(np.finfo(float).eps)
[Sortie Python] 2.22044604925e-16
```

eps est l'erreur relative entre deux réels qui ont la même représentation.

$$\texttt{eps} = 2^{-52} \simeq 10^{-16}$$

On l'appelle aussi "zéro machine". C'est le plus petit réel qui vérifie 1+eps>1. Pour des réels $x <$ eps, $1 + x$ et 1 sont codés identiquement :

```
eps=np.finfo(float).eps
1+eps/10
[Sortie Python] 1.0
```

Comme eps $\simeq 10^{-16}$, on peut considérer que les réels sont ainsi codés avec **16 chiffres significatifs**.

Il ne faut pas confondre le "zéro machine" **eps** avec le plus petit réel codable :

```
np.finfo(float).tiny
[Sortie Python] 2.2250738585072014e-308
```

Pour les grandes valeurs réelles, on a le plus grand réel codable :

```
np.finfo(float).max
[Sortie Python] 1.7976931348623157e+308
```

au delà duquel Python renvoie la variable **inf**, abréviation de infinity, on parle aussi dans ce cas de **overflow** :

```
v=1e+400
v
[Sortie Python] inf
```

Quand le nombre ne peut pas être évaluer, Pyhton renvoie la variable **nan** qui signifie "not a number" :

```
v-v
[Sortie Python] nan
```

3 Tableaux - Vecteurs - Matrices

Le module NumPy permet la manipulation simple et efficace de tableaux, de vecteurs et de matrices. **Les indices des tableaux, vecteurs et matrices commencent à 0 en Python**.
La ième composante d'un vecteur **x** est notée **x[i]** ; le coefficient de la ième ligne, jème colonne d'une matrice **A** est noté **A[i, j]**. Il n'y a pas à déclarer les tableaux, mais il est cependant recommandé de réserver une place mémoire pour les vecteurs et matrices en les initialisant à 0 (voir plus loin).

Tableaux ou Matrices Nous allons voir comment créer des tableaux avec la fonction

```
np.array()
```

de NumPy. Ces **tableaux** pourront être utilisés comme des **vecteurs** ou des **matrices** grâce à certaines fonctions de NumPy :

```
np.dot(), np.linalg.det(), np.linalg.inv(), np.linalg.eig(), etc.
```

qui permettent de réaliser des **calculs matriciels** utilisés en algèbre. Mais attention par défaut ce sont des **tableaux** c'est pour ça qu'il est par exemple possible de calculer le produit de 2 tableaux de dimension 1. Pour forcer l'utilisation de matrices il faut utiliser la commande :

```
np.matrix()
```

Cette commande permet aussi de transformer un **tableau** en **matrice**. Voici un exemple à tester pour comprendre cette subtilité :

```
x = np.array([2,5,3.2]) # un tableau de dim 1 x 3
y = np.array([1,10,0]) # un tableau de dim 1 x 3
print('Multiplication terme à terme : \n',x*y)
X = np.array([[2,5,3.2],[2,0,1.5],[1.5,2,4]]) # un tableau de dim 3 x 3
Y = np.array([[1,3,0],[4,1,2],[1,1,1]]) # un tableau de dim 3 x 3
print('Multiplication terme à terme \n: ',X*Y)
print('Produit matriciel : \n',np.matrix(X)*np.matrix(Y))
MX = np.matrix([[2,5,3.2],[2,0,1.5],[1.5,2,4]]) # une matrice de dim 3 x 3
MY = np.matrix([[1,3,0],[4,1,2],[1,1,1]]) # une matrice de dim 3 x 3
print('Produit matriciel : \n',MX*MY)
```

3.1 Création d'un tableau de dimension 1 / vecteur

Un **tableau à une ligne**, aussi utilisable comme un **vecteur** peut être défini de plusieurs façons :

(i) soit par une instruction de la forme

$$\mathbf{x = np.array([x1,x2,...,xn])},$$

où les éléments **xi** sont séparés par des virgules et mis entre **[ ]**.

(ii) soit par une subdivision uniforme d'un intervalle donné **[a, b]** en sous-intervalles de longueur donnée **h** :

```
x = np.arange(a,b,h) # de a (inclus) à b (exclus) par pas de h
x = np.arange(a,b) # par défaut h = 1
```

(iii) soit par une subdivision uniforme d'un intervalle donné **[a, b]** comprenant un nombre de points donné **n** :

```
x = np.linspace(a,b,n) # x[i]=a+(i-1)h, h=(b-a)/(n-1), i = 0,...,n-1
x = np.linspace(a,b) # par défaut n = 50
```

(iv) soit par initialisation à un vecteur avec des 0 ou des 1, de longueur donnée **n** :

```
x = np.zeros(n) # vecteur ligne, x[i]=0, i = 0,...,n-1
x = np.ones(n); # vecteur ligne, x[i]=1, i = 0,...,n-1
```

3.2 Opérations sur les tableaux de dimension 1 / vecteurs

Quelques opérations sur les tableaux / vecteurs :

u+v	:	addition de deux tableaux / vecteurs u et v de même longueur,
np.dot(u,v)	:	produit scalaire de deux **vecteurs** u et v,
u*v	:	multiplie deux **tableaux** u et v composantes par composantes,
u/v	:	divise deux à deux les composantes de deux **tableaux** u et v,
uv**	:	élève les composantes de u à la puissance des composantes de v,
np.abs(u)	:	vecteur donnant les valeurs absolues des composantes de u,
np.sum(u)	:	somme des composantes de u,
np.mean(u)	:	moyenne des composantes de u,
np.size(u)	:	longueur de u,
np.min(u)	:	plus petite composante de u,
np.max(u)	:	plus grande composante de u,
la.norm(u)	:	norme euclidienne du **vecteur** u,
u[n1 :n2]	:	extrait un sous-tableau / sous-vecteur de composantes u(n1) à u(n2**-1**) (**attention !**).

4 Création de tableaux / matrices

Un **tableau de n lignes et p colonnes** peut être défini :

(i) par une relation de la forme :

```
A = np.array([[a11,...,a1p],[a21,...,a2p],...,[an1,...,anp]])
```

où les aij sont les coefficients du tableau,

(ii) ou par initialisation à des tableaux particuliers :

```
A = np.zeros((n,p)) # n lignes et p colonnes, A(i,j) = 0
A = np.eye(n) # identité d'ordre n
A = np.ones((n,p)) # n lignes, p colonnes, A(i,j) = 1
A = rd.rand(n,p) # nombres aléatoires, 0 < A(i,j) < 1
A = np.diag(u) # diagonale, de diagonale égale au vecteur u
A = np.diag(u,1) # sur-diagonale égale à u, nulle ailleurs
```

Rappel : ici A est toujours un objet de type *array* donc un tableau. Attention aux opérations que vous faites dessus ! Pour passer en mode matriciel, il faut faire :

```
A = np.matrix(np.eye(n)) # MATRICE identité d'ordre n
```

5 Opérations sur les tableaux / matrices

Quelques opérations (en version tableau ou en version matricielle) :

np.shape(A)	:	donne les dimensions de A,
np.transpose(A)	:	transposée de A,
+	:	addition,
x = A[0, :]	:	première ligne de A,
y = A[:,1 :3]	:	deuxième et troisième colonnes de A,
w = A[3,i :j]	:	de l'indice i à l'indice (j-1) dans la ligne indice 3 (quatrième ligne),
u = A[:,2]	:	troisième colonne de A,
c*A	:	multiplie tous les éléments de A par le scalaire c.

Quelques opérations donnant des **résultats différents** en fonction de si A et B sont des matrices ou des tableaux :

A*B	:	donne le produit terme à terme de A et de B si A et B sont des tableaux, donne le produit matriciel de A et de B si A et B sont des matrices,
Am**	:	met à la puissance m (ici m est un scalaire) le tableau A si A est un tableau, donne la puissance m de la matrice A si A est une matrice.

Quelques opérations **matricielles** donnant les mêmes résultats pour tableau ou matrice :

la.eigvals(A)	:	vecteur donnant les valeurs propres de la matrice A,
la.det(A)	:	déterminant de la matrice A,
la.matrix_rank(A)	:	rang de la matrice A,
la.trace(A)	:	trace de la matrice A,
la.cond(A)	:	conditionnement de la matrice A,
la.solve(A, b)	:	solution du système matriciel $Ax = b$.

Exercice 1
A l'aide des fonctions **np.eye**, **np.diag** et **np.ones**, créer une matrice tridiagonale d'ordre 10 de la forme :

$$A = \begin{pmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & 0 & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & \ddots & 2 & -1 \\ & & & & -1 & 1 \end{pmatrix}$$

6 Scripts

Un script est un fichier contenant des instructions Python. **Un fichier script doit se terminer par le suffixe .py.** Voici un exemple de script défini dans un fichier premiertp.py :

```
# Mon premier programme Python
# premiertp.py
# Calcule le déterminant et les valeurs propres d'une matrice

import numpy as np        # importation du module numpy
import numpy.linalg as la # importation du module numpy.linalg
import numpy.random as rd # importation du module numpy.random


# Pour obtenir la dimension et l'affecter à la variable n :
n = int(input("Donner la dimension n de la matrice = "))

# Initialisation de la matrice avec des valeurs aléatoires
A = np.matrix(rd.rand(n, n))
print('A = ', A)

# Calcul du déterminant et des valeurs propres
DetA = la.det(A)
print('det(A) = ', DetA)
ValProp = la.eigvals(A)
print('Valeurs propres de A : ', ValProp)
```

Pyzo dispose d'un éditeur pour écrire et mettre au point les scripts. Pour **exécuter** le programme il suffit d'utiliser le menu `Run`, `Execute file`, ou le raccourci `ctrl+E`.

Les scripts sont **exécutés séquentiellement** dans la console Pyzo, c'est-à-dire qu'ils peuvent accéder aux variables qui s'y trouvent déjà, les modifier, en créer d'autres, etc.

Exercice 2
Ecrire, dans un fichier **TP1_1D.py**, le script suivant, et commencer à le compléter où cela est indiqué par la commande **TODO** :

```
# Fichier TP1_1D.py
# Programme de resolution du probleme aux limites mixtes en dimension 1
# -u''=f sur ]0,1], u(0)=ug, u'(1)=0
# par la méthode des différences finies (cf TD)

import numpy as np        # importation du module numpy
import numpy.linalg as la # importation du module numpy.linalg
```

```
8
9  print('Pour le second membre, choix de la fonction f')
10 print('Pour m=0 : f=0')
11 print('Pour m=1 : f=1')
12 print('Pour m=2 : f=x')
13 print('Pour m=3 : f=x^2')
14 print('Pour m=4 : f=4*pi^2*sin(2*pi*x)')
15 m = int(input("Choix de m = "))
16
17 print('Choix de la condition a gauche')
18 ug = float(input('ug = u(0) ='))
19
20 print('Choix du nombre N de points interieurs du maillage')
21 N = int(input('N = '))
22
23 # 1. Maillage - TODO
24 # Calculer le pas uniforme h de [0,1] subdivisé en N+1 intervalles.
25 # Définir la subdivision uniforme X de [0,1], de pas h.
26 # En déduire le vecteur Xh des (N+1) points du maillage où on cherche la
27 # solution (tous les points du maillage sauf 0).
28
29 # 2. Matrice du systeme lineaire - TODO
30 # Définir la matrice A à l'aide de l'exercice précédent (dimension N+1).
31 # Calculer son conditionnement.
```

7 Fonctions

Une fonction est un script admettant des variables d'entrées, par exemple $f(x, y) = x^2 + y^2$ admet x et y comme variables d'entrées, et peut renvoyer d'autres variables avec l'instruction `return`.

Voici la syntaxe d'une fonction "fonc" en Python :

```
1 def fonc(x, y, m):
2     z = x**2 + y **2 # attention à l'indentation du corps de la fonction !
3     t = x - y + m
4     return z, t
```

Les variables **x**, **y** et **m** sont les variables d'entrées, elles sont utilisées par la fonction sans être modifiées. Les variables **z** et **t** sont des variables locales, elles ne sont définies que dans la fonction `fonc`.

Utilisation de cette fonction : l'interpréteur exécute les lignes d'instructions d'un programme l'une après l'autre, dans l'ordre où elles apparaissent. Dans le script, la définition des fonctions doit donc précéder leur utilisation.

Un programme Python contient donc en général les blocs suivants, dans l'ordre :
- Quelques instructions d'initialisation (importation de modules, définition éventuelles de variables globales,...)
- Les définitions locales de fonctions
- Le corps principal du programme

8 Boucles et choix

8.1 Opérateurs logiques de comparaison

$<$	plus petit
$>$	plus grand
$<=$	plus petit ou égal
$>=$	plus grand ou égal
$==$	égal (compare si deux nombres sont égaux ou non)
!=	différent
and	et logique
or	ou logique
not	non (pour le contraire)
True	vrai
False	faux

8.2 Choix simple ou alternatif (if)

L'instruction conditionnelle en Python dans sa version la plus générale possède la syntaxe suivante :

```
if <condition1> :
   <instructions à exécuter si la condition1 est vraie>
elif <condition2> :
   <instructions à exécuter si la condition1 est fausse et si la condition2
est vraie>
elif <condition3> :
   <instructions à exécuter si les conditions 1 et 2 sont fausses et si la
condition3 est vraie>
   ...
else :
   <instructions à exécuter si aucune des conditions précédentes est vraie>
```

Il peut y avoir autant de elif qu'on le souhaite. On peut omettre la partie else et les elif. **Attention à l'indentation !**

Exemple :

```
a = np.log(2)
b = np.sqrt(2)/2
if (a > b) :
    print('a est plus grand que b')
else :
    print('a est plus petit que b')
```

Exercice 3

1. Définir dans le script **TP1_1D.py** la fonction **f(x,m)** donnée par :

$$f(x,m) = \begin{cases} 0 & \text{si } m = 0 \\ 1 & \text{si } m = 1 \\ x & \text{si } m = 2 \\ x^2 & \text{si } m = 3 \\ 4.\pi^2\, sin(2.\pi\, x) & \text{si } m = 4 \end{cases}$$

tek que que si x est un vecteur, le résultat y soit un vecteur de même dimension :

```
# Fonction f du second membre
def f(x, m):
    if (m == 0):
        y = np.zeros(np.size(x))
    elif (m == 1):
        y = np.ones(np.size(x))
    elif (m == 2):
        y = ...
    elif (m == 3):
        y = ...
    elif (m == 4):
        y = ...
    else :
        print('valeur de m indéfinie')
    return y
```

Attention, pour $m = 2$ pour avoir 2 objets en mémoire et non pas un seul objet avec deux alias, il faut faire :

```
y=np.copy(x) # cette commande est différente de celle : y = x !
```

2. Définir dans le script **TP1_1D.py** la fonction **solex(x,ug,m)** qui donne la solution exacte de l'équation $-u'' = f$ associée à la fonction **f(x,m)** et à la valeur **ug**. **Indication** : on calcule ces solutions "à la main" en intégrant deux fois la fonction f.

```
# Fonction définissant la solution exacte de l'équation
def solex(x, ug, m):
    if (m == 0):
        y = ug*np.ones(np.size(x))
    elif (m == 1):
        y = -0.5*x**2+x+ug
    elif (m == 2):
        y = ...
    elif (m == 3 ):
        y = ...
    elif (m == 4):
        y = ...
    else:
        print('valeur de m indéfinie')
    return y
```

3. Compléter, en appelant ces deux fonctions, le script **TP1_1D.py** à partir du calcul du second membre et jusqu'au calcul des erreurs :

```
# Second membre - TODO
b = ...

# Resolution du systeme lineaire
Uh = ...

# Calcul de la solution exacte aux points d'approximation
Uex = ....

# Calcul de l'erreur en norme infinie
err = ....
```

8.3 Boucle itérative pour (for)

Exemple :

```
A = np.zeros((4,5))
for i in np.arange(0, 4):
    for j in np.arange(4, -1, -1):
```

```
        A[i,j] = i+j**2
print('A = ', A)
```

8.4 Boucle itérative tant que (while)

En Python l'instruction while permet de répéter une opération tant qu'une condition (exprimée sous forme d'expression booléenne) est vérifiée. **Exemple :**

```
# Recherche de l'indice du 1er coefficient non nul d'un vecteur u
n=np.size(u)-1
test = False
i = -1
while ((not test) and (i < n)):
    i = i+1
    test = (np.abs(u[i]) > np.finfo(float).eps)

print('i = ', i)
```

9 Graphismes

Pour tracer des courbes, on utilisera le module Matplotlib qu'on importera de la manière suivante :

```
import matplotlib.pyplot as plt
```

On utilisera en particulier les fonctions `title`, `plot`, `xlabel`, `ylabel`, `legend`, `show`.

Exercice 4
Compléter le script **TP1_1D.py** pour tracer les graphes après le calcul des erreurs :

```
# Graphes
Xh = np.concatenate((np.array([0]),Xh)) # on complete avec 0
Uh = np.concatenate((np.array([ug]),Uh)) # on complete avec la valeur ug
# On trace le graphe de la fonction solex sur un maillage fin de 100 points
plt.plot(Xh,solex(Xh, ug, m), label = 'sol ex')
# et le graphe de la solution approchée obtenue
plt.plot(...)
# On ajoute un titre
plt.title('...')
# On ajoute les labels sur les axes
plt.xlabel('...')
plt.ylabel('...')
```

```
plt.legend() # Pour faire afficher les labels
plt.show() # Pour afficher la figure
```

10 Quelques fonctions mathématiques

Quelques fonctions usuelles :

- **np.sin, np.cos, np.tan, np.sinh, np.cosh, np.tanh,...**
- **np.arcsin, np.arccos, np.arctan, np.arcsinh, np.arccosh, np.arctanh, ...**
- **np.exp, np.log, np.log10, np.sqrt, ...**

et aussi :

np.fix(x)	:	plus proche entier inférieur ou égal à x en valeur absolue
np.floor(x)	:	partie entière de x
np.ceil(x)	:	plus proche entier plus grand ou égal à x
np.round(x)	:	entier le plus proche de x
np.mod(x, y)	:	le reste de la division euclidienne de x par y
np.sign	:	signe de x (vaut -1, 0 ou 1)

11 Equation de Poisson 1D

On considère une tige métallique de longueur $L = 1$ sur laquelle on applique une source de chaleur, représentée par une fonction f connue. On suppose que l'extrémité gauche est maintenue à une température u_g, et que l'extrémité droite est isolée. La température u dans l'intérieur de la tige (température d'état stationnaire) est solution du problème suivant :

$$\begin{cases} -u''(x) = f(x), \ x \in \Omega =]0, 1], \\ u(0) = u_g, \\ u'(1) = 0. \end{cases}$$

Exercice 5

Reprendre le script **TP1_1D.py**.

1. Vérifier que la résolution fonctionne bien : choisir les exemples de fonctions f pour lesquelles la méthode est exacte. Pour $h = 0.1$ (c'est-à-dire $N = 9$), compléter le tableau de la fiche de résultats avec les erreurs et les observations. Les solutions exactes et approchées doivent se superposer, et l'erreur doit être de l'ordre du zéro machine.

2. Etudier la convergence du schéma : pour chacun des exemples, compléter le tableau de la fiche de résultats avec les erreurs pour $h = 0.1, 0.01, 0.001$. Etudier l'ordre de convergence obtenu.

Analyse numérique (XMS1ME090) – 2026-2027 – TP1 – Partie 2
Méthode des différences finies
Equation de Laplace 2D

On considère une plaque carrée dont les côtés sont maintenus à des températures données (en ^oC) : $u_d(x)$ sur le côté supérieur et 0 sur les trois autres côtés. La température $u(x, y)$ dans l'intérieur de la plaque (température d'état stationnaire) est solution du problème suivant :

$$\begin{cases} \Delta u(x,y) = 0, \ \forall\,(x,y) \in \Omega =]0,1[^2, \\ u(0,y) = u(1,y) = u(x,0) = 0, \ \forall\, x \in [0,1], \ \forall\, y \in]0,1[, \\ u(x,1) = u_d(x), \ \forall\, x \in [0,1]. \end{cases}$$

Le carré Ω est maillé avec un pas uniforme $h = 1/(N+1)$ dans les deux directions :

$$\begin{cases} x_i = ih, \ i = 0, \ldots, N+1 \text{ dans la direction } x, \\ y_j = jh, \ j = 0, \ldots, N+1 \text{ dans la direction } y. \end{cases}$$

En appliquant la méthode des différences finies, on obtient que le vecteur U_h des valeurs $u_k = u_h(M_k) \simeq u(M_k)$ aux points M_k du maillage, situés à l'intérieur de Ω, numérotés de 1 à N^2, comme dans la figure suivante (exemple avec $N = 3$) :

7	8	9	
4	5	6	
1	2	3	

est solution d'un système linéaire $AU_h = b$, de dimension N^2, dont la matrice est symétrique et constituée de blocs carrés de dimension N. Par exemple pour $N = 3$:

$$A = \left(\begin{array}{ccc|ccc|ccc} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ \hline -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ \hline 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{array}\right) \quad \text{et} \quad b = \left(\begin{array}{c} 0 \\ 0 \\ 0 \\ \hline 0 \\ 0 \\ 0 \\ \hline u_d(x_1) \\ u_d(x_2) \\ u_d(x_3) \end{array}\right)$$

On va tester la résolution du problème pour la fonction u_d suivante :

$$u_d(x) = \sin(2\pi x)\sinh(2\pi)$$

pour laquelle on pourra vérifier que la solution exacte du problème est :

$$u(x, y) = \sin(2\pi x)\sinh(2\pi y)$$

1. Ecrire un script Python **TP1_2D.py** qui effectue les étapes suivantes :
   (a) Calcul d'une subdivision uniforme X de $(N + 2)$ points de $[0, 1]$
   (b) Calcul de la matrice A et du second membre b
   (c) Résolution du système linéaire : $U_h = A^{-1}b$
   (d) Rangement des valeurs de u_h aux points (x_i, y_j) du maillage dans une matrice Z_h :
   $$Z_h[i, j] = u_h(X[i], X[j]),\ i, j = 1, \ldots, N + 2$$
   Indication : utiliser les valeurs $U_h[k] = u_h(M_k)$ en parcourant le maillage ligne par ligne en suivant l'ordre des points M_k et en complétant avec les valeurs au bord.
   (e) Calcul de la solution exacte u aux points (x_i, y_j) du maillage dans une matrice Z :
   $$Z[i, j] = u(X[i], X[j]),\ i, j = 1, \ldots, N + 2$$
   (f) Calcul de l'erreur max :
   $$erreur = \max_{i,j} |u(x_i, y_j) - u_h(x_i, y_j)|$$
   (Remarque : pour calculer le maximum des coefficients d'une matrice A, utiliser la commande **amax(A)**).
2. Tracé du graphe de la fonction u_h : pour cela, on crée les matrices **coordX** et **coordY** telles que
   $$Z_h[i, j] = u_h(\text{coordX}[i, j], \text{coordY}[i, j]).$$
   Pour le tracé et surtout pour choisir la carte de couleur il faut ajouter

```
from matplotlib import cm
```

   et ensuite on peut faire :

```
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
surf = ax.plot_surface(coordX, coordY, Z, cmap=cm.jet,
                       linewidth=0, antialiased=False)
plt.show()
```

3. Faire plusieurs essais en faisant varier N. Mettre en évidence l'ordre de convergence.