TP 2 : Introduction à Numpy - Matplotlib

Introduction

Ce TP débute par un tutoriel d'introduction aux modules NUMPY et MATPLOTLIB de PYTHON. Il a pour but de présenter l'essentiel grâce à des exemples à saisir dans la console (sous IDLE3, IPYTHON3, PYZO ou autre IDE), et dont analysera les résultats. Il se termine par quelques exercices de représentation graphique. On pourra créer des fonctions lorsque le programme doit être exécuté plusieurs fois en variant certains paramètres.

Le chargement des bibliothèques se fait habituellement avec les alias `np` et `plt` :

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> a = np.cos(np.pi)
...
>>> plt.plot(t,cos(t))
```

Mais pour ce texte, dans la suite on utilisera simplement les fonctions sans alias :

```
>>> from numpy import *
>>> from matplotlib.pyplot import *
>>> a = cos(pi)
...
>>> plot(t,cos(t))
```

Il est aussi possible d'importer directement une synthèse cohérente de NUMPY, SCIPY et MATPLOTLIB simplement par

```
>>> from pylab import *
```

Le cœur de la philosophie de NUMPY réside dans l'utilisation d'objets de type tableau (`array`) multidimensionnel, que l'on qualifiera de *vecteurs* (ligne ou colonne de taille `n`, une dimension) ou de *matrices* (multidimensionnel, ex : en 2 dimensions de tailles `n` × `m`). Nombre de fonctions pré-compilées et optimisées sont associées à ce type et permettent de simplifier l'écriture du code et en accélérer la réalisation. En particulier, les opérations courantes et les fonctions usuelles s'appliquent élément par élément sur un objet `array` sans avoir à le parcourir explicitement par une boucle ou des boucles imbriquées. Ainsi, on sera ammené lorsque cela est nécessaire, à abandonner le type `list` pour le type `array` pour bénéficier de cette efficacité. Notons toutefois que **cette conversion est parfois implicite avec certaines fonctions**. Voir par exemple :

```
>>> L = [1, 2, 3]
>>> exp(L)
>>> L**2
>>> array(L)**2
```

Ce document n'est évidemment pas exhaustif, on prendra l'habitude de chercher les informations dans la documentation locale ou en ligne :

— http ://www.numpy.org/ pour NUMPY et SCIPY ;

— http ://matplotlib.org/ pour les graphiques avec MATPLOTLIB.

Table des matières

I Expressions arithmétiques et fonctions usuelles **2**

II Vecteurs et matrices (array) - NUMPY **2**
1 Générer des vecteurs ou des matrices . . . 2
2 Manipuler les éléments d'une matrice . . . 3
3 Opérations élément par élément . . . 3
4 Fonctions usuelles appliquées à une matrice . . . 4
5 Multiplication matricielle . . . 4
6 Opérateurs relationnels . . . 4

III Graphiques : MATPLOTLIB **4**
1 Graphiques à 2 dimensions . . . 4
a La commande `plot` . . . 4
b Titres, légendes et annotations . . . 5
c Sauvegarde de la figure . . . 5
2 Manipuler plusieurs graphiques . . . 5
3 Graphiques à 3 dimensions . . . 6
a Les courbes non planes . . . 6
b Les surfaces (fonctions de 2 variables) . . . 6

IV EXERCICES **8**
1 Courbes planes . . . 8
2 Surfaces . . . 8
3 Représentation d'une fractale : l'ensemble de Julia . . . 8

I. Expressions arithmétiques et fonctions usuelles

On peut construire des expressions avec les opérateurs usuels, qui respectent l'ordre de priorité classique :

- ** élévation à une puissance
- * multiplication
- + addition
- − soustraction

Nous verrons dans un prochain paragraphe comment ces opérations sont utilisables sur les matrices.
NUMPY fournit les fonctions mathématiques usuelles que l'on trouve sur toute calculatrice moderne, nous ne citons ici que les plus classiques :

`sqrt`	racine carrée
`log`	logarithme
`log10`	logarithme décimal
`sin`	sinus
`cos`	cosinus
`tan`	tangente
`arctan`	arc tangente
`exp`	exponentielle
`cosh`	cosinus hyperbolique
`floor`	partie entière
`round`	arrondi à l'entier le plus proche
`abs`	valeur absolue ou module pour les complexes
`real`	partie réelle d'un complexe
`imag`	partie imaginaire d'un complexe

Ces fonctions peuvent agir sur les nombres complexes. Tester par exemple

```
>>> sqrt(-1)
>>> sqrt(1j**2)
```

II. Vecteurs et matrices (array) - NUMPY

II.1. Générer des vecteurs ou des matrices

Un vecteur est une matrice à une seule ligne (ou une seule colonne après transposition). Il y a plusieurs façons d'entrer un vecteur ou une matrice :

- Entrer au clavier une liste explicite d'éléments et la transformer en objet de type `array` (plus exactement `ndarray`).
- Générer cette matrice par une instruction automatique (taille quelconque) ;
- Charger cette matrice par lecture d'un fichier de données externe (`readlines`, `csv.reader`...).

La place mémoire nécessaire à une matrice est automatiquement allouée au moment de sa création.

Pour entrer une petite matrice explicitement :

```
>>> V = array([1, 2, 3])
>>> A = array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> A
>>> print(A)
```

On accède aux éléments avec la même logique qu'en mathématique pour les matrices :

```
>>> V[0]
>>> A[0,0]
```

Pour générer une matrice de taille quelconque remplie de zéros ou de 1 :

```
>>> A = zeros((3,2))
>>> A = ones((2,4))
```

On peut aussi générer une **matrice identité** :

```
>>> I = eye(3,3)
>>> I = eye(3,4)
```

On peut aussi générer des **vecteurs** particuliers à l'aide de la fonction `arange` (contraction de `array(range(...))`) qui étend le `range` natif sous PYTHON :

```
>>> x = arange(1,5)
>>> array(range(1,5))
>>> x = arange(1,5,0.2)
```

On notera que cette commande produit toujours un vecteur ligne. Il est bien sûr possible d'utiliser un incrément négatif,

```
>>> y = arange(6,1,-1)
```

Une alternative consiste à spécifier uniquement les valeurs minimum et maximum et le nombre de valeurs désirés, grâce à `linspace` :

```
>>> k = linspace(1,5,5)
```

On peut aisément **créer des tables** en utilisant ces commandes, ce qui est utile notamment pour créer rapidement des représentations graphiques (section suivante). On peut aussi créer un découpage logarithmique :

```
>>> k = logspace(1,5,5)
```

ce qui est utile pour les représentations graphique avec axe logarithmique.

II.2. Manipuler les éléments d'une matrice

On peut accéder aux éléments individuels d'une matrice en précisant leurs numéros de ligne et de colonne dans des parenthèses. Prenons la matrice

```
>>> A = array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

La commande suivante permet de remplacer a_{33} par $a_{13} + a_{31}$:

```
>>> A[2,2] = A[0,2] + A[2,0]
```

On peut **extraire une ligne ou une colonne** d'une matrice :

```
>>> v = A[0,:]
```

permet de mettre dans le vecteur v la première ligne de la matrice A, alors que

```
>>> v = A[:,1]
```

permettra d'extraire la deuxième colonne de A.
De manière générale un indice de ligne ou de colonne peut être un vecteur :

```
>>> x = linspace(0,8,5)
>>> v = array([3, 0, 1, 2, 4])
>>> x[v]
```

On a effectué une **permutation** des éléments de x via v. On peut aussi **extraire une sous-matrice** d'une matrice plus grande :

```
>>> B = copy(A[0:2,:])
```

sélectionne les deux premières lignes et toutes les colonnes de A et les stocke dans une matrice B. On a ici utilisé `copy` pour éviter que `B` reste couplé à `A` (même emplacement mémoire). Tester l'effet d'une modification de A dans ce cas puis sans `copy`.
Il est aussi possible de construire une matrice à partir de matrices plus petites. Par exemple, pour **ajouter une nouvelle ligne** à la matrice A :

```
>>> vl = array([[10, 11, 12]])
>>> B = concatenate((A,vl))
>>> C = concatenate((A,vl.T), axis=1)
```

La seconde instruction utilise la **transposée** `l.T` de `l`, et concatène selon l'axe des colonnes (par défaut sur les lignes, `axis=1`).
On peut aussi facilement modifier les sous-matrices d'une matrice (comme on l'a déjà vu plus haut pour le cas d'une ligne ou d'une colonne) :

```
>>> A[0:2,0:2] = eye(2,2)
```

Ici on a remplacé la sous-matrice principale d'ordre 2 de A par la matrice identité.
Enfin, on peut obtenir la **taille d'une matrice** (nombre de lignes et de colonnes),

```
>>> shape(A)
```

et l'injecter dans des variables :

```
>>> [n,m] = shape(A)
```

II.3. Opérations élément par élément

Les opérateurs usuels $+$, $-$, $*$, $/$ et $**$ **agissent élément par élément**. C'est la grande force de Numpy, et plus généralement des langages de calcul matriciel (Scilab, Matlab, ...). Ces opérations sont valides dès que les **dimensions des matrices sont les mêmes**. Par exemple avec les matrices de l'exemple précédent l'addition

```
>>> A + x
```

n'est pas possible car A et x ne sont pas conformes. Par contre les opérations

```
>>> A**2
>>> A + A**2
>>> A*A
>>> A/A
>>> A**A
```

sont bien réalisées, élément par élément.
Toutefois, ces opérations sont acceptées si l'un des opérandes est un scalaire :

```
>>> A+1
>>> 2*A
>>> A/2
```

II.4. Fonctions usuelles appliquées à une matrice

Les fonctions usuelles s'appliquant sur réels et complexes s'**appliquent aussi élément par élément** sur les vecteurs et matrices, ce qui est fort utile pour les représentations graphiques ou les exploitations de données. Par exemple :

```
>>> cos(A)
```

II.5. Multiplication matricielle

La multiplication matricielle est bien sûr possible dès que les dimensions des opérandes sont compatibles : le nombre de colonnes de l'opérande de gauche doit être égal au nombre de lignes de l'opérande de droite. Par exemple l'opération suivante n'est pas valide

```
>>> dot(A,x)
```

mais les suivantes le sont

```
>>> B = A+1
>>> dot(A,B)
>>> dot(eye(3,3),B)-B
>>> dot(x,v)
```

La dernière commande donne le produit scalaire entre les vecteurs `x` et `v`.

II.6. Opérateurs relationnels

Six opérateurs relationnels sont disponibles pour comparer deux matrices de dimensions égales :

`<`	plus petit que
`<=`	plus petit ou égal
`>`	plus grand
`>=`	plus grand ou égal
`==`	égal
`!=`	différent

NUMPY compare les paires d'éléments correspondants. Le résultat est une matrice de constantes booléennes, de valeurs `False` ou `True`. Par exemple

```
>>> (A-1)<=0
```

renvoie une matrice indiquant les éléments négatifs de `A-1`. Ceci peut permettre d'exprimer une expression logique dans une instruction conditionnelle de façon condensée.

III. Graphiques : MATPLOTLIB

Cette section donne un bref apercu des possibilités graphiques de MATPLOTLIB. Il est possible de générer des graphiques à deux et à trois dimensions, et d'agir facilement sur les attributs du graphe (couleurs, type de ligne, axes, annotations...). Les commandes présentées ci-dessous sont utiles notamment pour être introduites dans des scripts et automatiser la production de figure. De plus leur syntaxe a l'avantage d'être très proche de celle de Matlab.

III.1. Graphiques à 2 dimensions

a. La commande `plot`

Pour générer un graphe en deux dimensions, la commande générale est `plot` :

```
>>> t = arange(0,2*pi,pi/4)
>>> plot(t,sin(t))
```

À ce stade, l'image est créée mais ne s'affiche pas. On peut alors l'afficher :

```
>>> show()
```

Sur la figure, quelques boutons permettent un minimum d'interactivité sur le graphe (zoom, fenêtre...) ainsi qu'une sauvegarde directe à la souris. Pour modifier ou ajouter un élément sur le graphe, il est alors nécessaire de fermer la figure et répéter la séquence d'instructions qui l'ont créée [1].
La syntaxe est claire : `plot(x,y)` permet de tracer une courbe reliant les points dont les **abscisses** sont données dans le vecteur x et les **ordonnées** dans y. Une première remarque s'impose : les points sont a priori reliés par des segments de droite, et plus on prendra de points intermédiaires, plus le tracé sera fidèle à l'allure théorique. Par exemple :

```
>>> t = arange(0,2*pi,pi/16)
>>> plot(t,sin(t))
```

Ici on a tracé le graphe d'une fonction de x, mais on peut aussi tracer une **courbe paramétrée** du type

$$\begin{cases} x &= f(t) \\ y &= g(t) \end{cases} \quad \text{pour} \quad t \in [a,b]\,.$$

Par exemple pour un cercle :

1. d'où l'intérêt de sauvegarder les instructions dans un script : `ma_figure.py`.

```
>>> t = linspace(0,2*pi,64)
>>> plot(cos(t),sin(t))
>>> axis("equal")
>>> xlim(-1.5,1.5)
```

La commande `axis("equal")` permet d'imposer une échelle identique sur les deux axes de coordonnées. La commande `xlim(-1.5,1.5)` permet de fixer l'intervalle de visualisation sur l'axe des abscisses.

Il est possible de **superposer deux courbes** sur le même graphique :

```
>>> plot(t,cos(t),t,sin(t))
```

Il est possible de **changer les couleurs et les type de lignes** utilisés :

```
>>> plot(t,cos(t),'g',t,sin(t),'r')
```

On peut aussi ne faire apparaître que les points et ne pas les relier par des segments de droite. Dans ce cas on peut repérer chaque point par un symbole (point, rond, étoile, croix...). On peut aussi modifier le type de ligne reliant les points :

```
>>> plot(t,cos(t),'g^:',t,sin(t),'ro')
```

Donc on peut modifier couleur et types de tracé (ligne ou symbole) en faisant suivre chaque couple x, y d'une chaîne de caractères entourée d'apostrophes (le même caractère que pour la transposition des matrices) composée de deux symboles précisant la couleur et le symbole. Le tableau ci-dessous donne quelques symboles et couleurs possibles (non exhaustif) :

Color	Marker	Linetype
`y` yellow	`.` dot	`-` Ligne continue (Solid line, default)
`m` magenta	`o` circle	`- -` tirets (Dashed line)
`c` cyan	`x` cross	`:` pointillés (Dotted line)
`r` red	`+` plus	`-.` tirets-points (Dash-dotted line)
`g` green	`*` star	
`b` blue	`d,D` diamond	
`w` white	$\wedge$ upward-pointing triangle	
`k` black	`v` downward-pointing triangle	

On peut aussi changer la taille du trait avec `linewidth`, et plein d'autres choses. Pour en savoir plus, il suffit de taper

```
>>> help(plot)
```

b. *Titres, légendes et annotations*

En particulier, on prendra soin de toujours légender un graphe (axes, identification des courbes, titre), et éventuellement ajouter une trame pour se repérer plus facilement :

```
>>> plot(t,cos(t),label="cos")
>>> plot(t,sin(t),label="sin")
>>> xlabel('abscisses')
>>> ylabel('ordonnées')
>>> title('Fonctions sinus et cosinus')
>>> legend()
```

On pourra aussi ajouter des indications sur le graphe (texte et flèches) avec `text` et `annotate`.

c. *Sauvegarde de la figure*

On peut aussi sauvegarder la figure (en `.png` ou `.pdf` par exemple), pour la voir ou la conserver.

```
>>> savefig('ma_figure.png')
>>> show()
```

L'image `ma_figure.png` est alors apparue dans le *répertoire courant*[2]. Avant la sauvegarde, on peut aussi en modifier les dimensions :

```
>>> figure(figsize=(10,6))
>>> savefig('ma_figure.pdf', dpi=200)
```

Elle aura pour taille 10*200 × 6*200 -> 2000×1200. L'argument `dpi` (*dot per inch*) est facultatif et sa valeur par défaut est 100.

III.2. Manipuler plusieurs graphiques

On peut utiliser plusieurs fenêtres graphiques en même temps, à condition de stoquer les figures dans des variables :

```
>>> fig1 = figure()
>>> plot(t,cos(t))
>>> fig2 = figure()
>>> plot(t,sin(t))
>>> show()
```

2. Lorsque python est lancé depuis la console, il s'agit du répertoire de lancement.

Une même fenêtre peut intégrer **plusieurs graphes** non superposés grâce à la commande `subplot` :

```
>>> subplot(2,1,1)
>>> plot(t,cos(t))
>>> subplot(2,1,2)
>>> plot(t,sin(t))
```

La commande `subplot(n,m,k)` permet de **subdiviser** la fenêtre en $n*m$ cases, n lignes et m colonnes (comme pour les matrices). Ces cases sont numérotées de gauche à droite et de haut en bas. La valeur de k permet de spécifier dans quelle case on désire faire un graphique.

III.3. Graphiques à 3 dimensions

a. Les courbes non planes

Une courbe non plane peut s'exprimer sous la forme d'une équation intrinsèque, mais c'est une forme non directement exploitable pour la représentation graphique. On préfèrera la forme paramétrée, qu'on obtient par exemple naturellement lorsqu'on cherche un mouvement par intégration du principe fondamental de la dynamique. Par exemple une trajectoire hélicoïdale aura une forme du type :

$$\left\{\begin{array}{rcl} x(t) & = & \cos(t) \\ y(t) & = & \sin(t) \\ z(t) & = & t \end{array}\right. \quad \text{avec} \quad (t,s) \in [a,b]\times[c,d].$$

Il est alors possible de tracer des courbes dans l'espace avec la librairie `Axes3D` :

```
>>> from mpl_toolkits.mplot3d import Axes3D
>>> gca(projection='3d').plot(cos(t),sin(t),t,label="helice")
>>> show()
```

Le graphe est alors interactif de telle sorte que l'on peut modifier l'angle de vue de la courbe et la « faire tourner » à l'aide du clic gauche de la souris (on voit s'afficher les valeurs des angles latitude (*elevation*) et azimuth. .

b. Les surfaces (fonctions de 2 variables)

On peut générer des surfaces (*mapping*) via des graphes de fonctions de deux variables

$$z = f(x,y)\,,$$

ou bien des graphes paramétriques

$$\left\{\begin{array}{rcl} x & = & f(t,s) \\ y & = & g(t,s) \\ z & = & h(t,s) \end{array}\right. \quad \text{avec} \quad (t,s) \in [a,b]\times[c,d].$$

Voici un premier exemple permettant de tracer le graphe de la fonction

$$z = f(x,y) = \cos\left(\pi\sqrt{x^2+y^2}\right)\,,$$

sur le domaine de variation $(x,y) \in [-1,1]\times[-2,2]$.

Création d'une grille de points (grid) de type matrice

Comme pour les courbes en 2D, il faut échantillonner les variables. Par exemple :

```
>>> x = linspace(-1,1,100)
>>> y = linspace(-2,2,200)
```

Cependant, à ce stade Python ne sait pas évaluer directement toutes les valeurs $f(x_i,y_j)$ pour tous les couples (x_i,y_j) simplement à partir des vecteurs x et y. Il faudrait construire un objet représentant une "grille" ou un "maillage" (une matrice de taille 200×100 en l'occurence) qui contienne tous les couples (x_i,y_j). En pratique cela se fait en construisant deux matrices 20×10 qui répètent chacune les valeurs des vecteurs x et y :

```
>>> [X,Y] = meshgrid(x,y)
```

On peut ensuite calculer en une seule instruction les valeurs de f aux points (x_i,y_j), ce qui crée aussi par une matrice 20×10 :

```
>>> Z = cos(pi*sqrt( X**2 + Y**2))
```

Graphe d'une surface en lignes de niveaux

Une surface est couramment représentée via un principe de lignes de niveau, comme dans le cas d'une carte topographique :

```
>>> contour(X, Y, Z)
```

On peut choisir le nombre de niveaux donc de contours, et même les valeurs des contours à représenter (qui ne sont donc pas forcément équidistants). On peut aussi choisir l'échelle de couleur (*colormap*, argument `cmap`). Les noms des différentes échelles disponibles sont accessibles via `colormap()`.
On gagne en général en visibilité en remplissant les interstices entre les niveaux par la couleur choisie (*fill*) :

```
>>> contourf(X, Y, Z, cmap=cm.coolwarm)
```

Enfin, lorsque l'on ne prète pas une attention particulière aux valeurs prises par les coordonnées x et y, on peut les ommettre avec la fonction

```
>>> imshow(Z)
```

qui créée un dégradé de couleur.

Graphe d'une surface en 3D

Pour représenter ces surfaces en trois dimensions, on utilise aussi la librairie `Axes3D`. On utilise ensuite X et Y avec la commande `plot_surface`, qui créée un « grillage » à partir des points de coordonnées x et y, et projette les valeurs de z sur une **échelle de couleurs** avec remplissage.

```
>>> from mpl_toolkits.mplot3d import Axes3D
>>> gca(projection='3d').plot_surface(X, Y, Z, cmap=cm.coolwarm)
>>> show()
```

Il est aussi possible de paramétrer le point de view via

```
>>> view_init(elev=10., azim=45.)
```

ce qui est utile lorsque la figure est réalisée par un script, en mode non interactif.

Si l'on ne souhaite pas avoir de remplissage par une échelle de couleurs, on peut préférer `plot_wireframe`, qui n'affiche que le « grillage ».

IV. EXERCICES

IV.1. Courbes planes

1. Tracer une ellipse sous forme paramétrée par la variable temps :

$$\left\{ \begin{array}{rcl} x(t) & = & X_m \cos(\omega t) \\ y(t) & = & Y_m \cos(\omega t + \varphi) \end{array} \right.$$

Explorer l'influence de la constante φ.

2. Tracer une ellipse paramétrée sous forme polaire :

$$r = \frac{p}{1 + e\cos(\theta - \theta_0)} \quad \text{avec} \quad e < 1$$

Explorer l'influence de la constante θ_0. Que se passe-t-il pour $e > 1$?

IV.2. Surfaces

3. Représenter un massif montagneux de profil gaussien :

$$z = e^{-(x^2+y^2)} \cdot (1 + \cos(2\pi(x+y))) \,.$$

4. Tracer une sphère paramétrée en coordonnées sphériques :

$$\left\{ \begin{array}{rcl} x & = & R\cos\varphi\,\sin\theta \\ y & = & R\sin\varphi\,\sin\theta \\ z & = & R\cos\theta \end{array} \right.$$

5. Expérimenter des surfaces plus générales où le rayon est une fonction $R(\varphi, \theta)$, par exemple

$$R(\varphi, \theta) = 2 + \sin(4\varphi + 4\theta) \,,$$

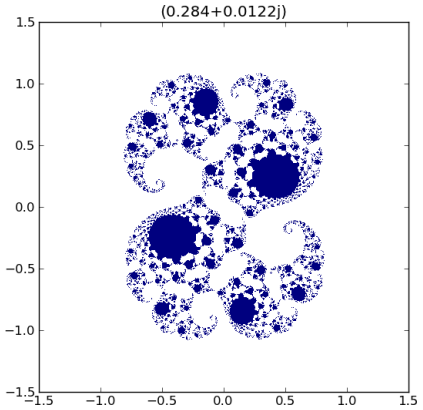
où toute autre fonction issue de votre imagination !

IV.3. Représentation d'une fractale : l'ensemble de Julia

On considère un nombre complexe c et le polynôme $P(z) = z^2 + c$. On défini la suite $(z_n)_{n\in\mathbb{N}}$ par

$$z_{n+1} = P(z_n) = z_n^2 + c \quad \text{avec} \quad z_0 \in \mathbb{C} \,.$$

L'*ensemble de Julia rempli* associé à P et paramétré par c est l'ensemble des points z_0 du plan complexe (premier terme de la suite) tels que la suite $(z_n)_{n\in\mathbb{N}}$ soit bornée. C'est la plupart du temps un objet fractal.



On souhaite obtenir une représentation approchée de cet ensemble dans le cas où

$$c = 0,284 + i\,0,0122$$

qui correspond à l'image ci-contre. Pour ce faire, on se limite aux suites bornées par la valeur 2 ($|z_n| \leq 2$), dans la limite d'un nombre d'itérations maximal N imposé (ex : $N = 100$).

6. Ecrire un programme qui réalise successivement les étapes suivantes. On oublira pas de commenter le code pour la lisibilité (`# ...`). On pourra s'aider de la fiche *Python, Numpy, Pylab pour le calcul numérique*.

a) Construire un pavage du plan complexe sur le domaine $(x, y) \in [-1,2; 1,2]\times[-1,2; 1,2]$, avec (x, y) les parties réelle et imaginaire de z_0 : $z_0 = x + i\,y$. On définira ainsi deux *vecteurs* xx et yy de taille $d = 500$ prenant des valeurs équidistantes sur ces intervalles (ex : `np.linspace`).

b) Construire une matrice M (tableau à 2 dimensions) de taille $d \times d$ dont les éléments, qui correspondent aux points de coordonnées (x, y) du pavage, seront affectés de la valeur 0 si la suite $(z_n)_{n\in\mathbb{N}}$ avec $z_0 = x + i\,y$ est non bornée par 2, ou de la valeur 1 si elle l'est (`np.empty`).

c) Représenter graphiquement la matrice M (`pl.imshow`).

d) Calculer l'aire relative occupée par l'ensemble de Julia rempli (pourcentage d'occupation du domaine).