Chapitre 7
Matrices, courbes et images

1 Tracé de courbes

Dans cette partie, on explique comment tracer une courbe en Python, par exemple, la courbe représentative d'une fonction.

Comme la courbe contient une infinité de points, il n'est pas possible de les stocker ou de les calculer tous. On ne place alors sur le graphique qu'un nombre fini de points, que l'on relie ensuite par des segments de droite.

Bien sûr, plus on choisit de points, plus le tracé est lisse et précis, mais plus le temps de calcul est long. En pratique, on peut choisir un millier de points pour le tracé d'une courbe.

1.1 Tracé

On commence par importer la bibliothèque `matplotlib.pyplot`, qui permet de produire des éléments graphiques.

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

Pour tracer la courbe, on doit disposer de la liste `abscisses` de toutes les abscisses des points à placer ainsi que de la liste `ordonnees` de toutes leurs ordonnées, dans l'ordre dans lequel on souhaite les placer.

Le tracé s'effectue alors à l'aide de la commande `plot` (*plot* pour *tracer* en anglais) du module `matplotlib.pyplot`, dont le premier argument sera placé en abscisses et le second argument en ordonnées.

```
plt.plot(abscisses, ordonnees)
```

Plaçons par exemple les points de coordonnées $(1, 5)$, $(2, 3)$, $(3, -2)$ et $(2, 0)$, dans cet ordre, sur un graphique.

```
abscisses = [1, 2, 3, 2]
ordonnees = [5, 3, -2, 0]
plt.plot(abscisses, ordonnees)
```

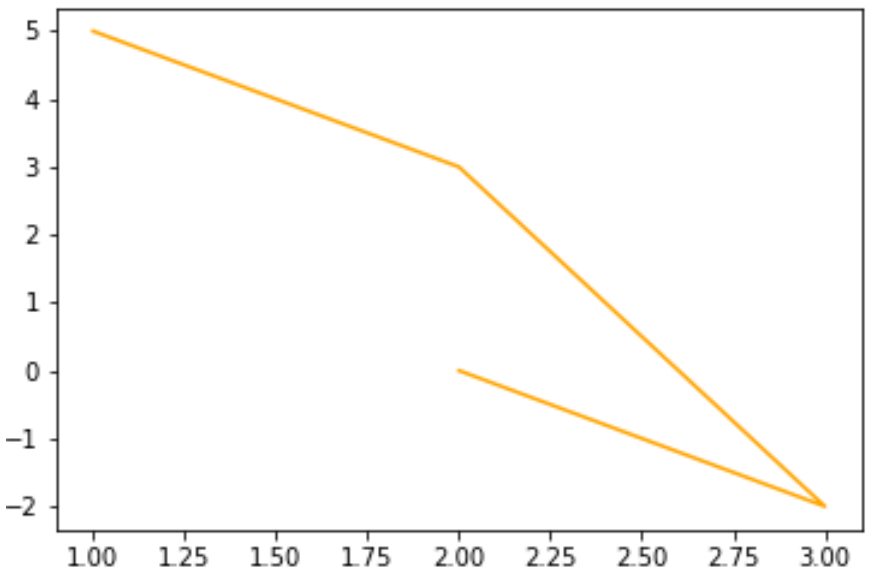


FIGURE 1 – Par défaut, les points sont reliés par des segments de droite.

Notons que la courbe représentée dans l'exemple précédent n'est pas le graphe d'une fonction.

Dans la suite, intéressons-nous au cas très important du tracé du graphe d'une fonction f sur un intervalle $[a; b]$.

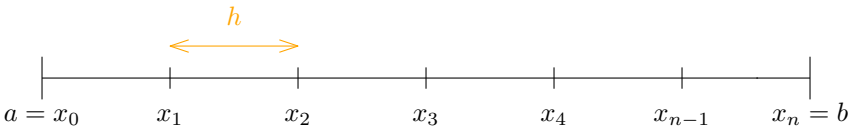
1.2 Création de la liste des abscisses

La liste des abscisses des points à tracer peut être n'importe quelle liste de flottants et on peut la construire de diverses manières (comme nous l'avons vu dans le chapitre 5).

Toutefois, pour tracer le graphe de la fonction f sur l'intervalle $[a;b]$, on choisira souvent une liste de points régulièrement espacés de l'intervalle $[a;b]$. On subdivise pour cela le segment $[a;b]$ en n petits sous-segments $[x_k;x_{k+1}]$ de longueur $h = \dfrac{b-a}{n}$ en posant, pour tout $k \in [\![0;n]\!]$:

$$x_k = a + k\frac{b-a}{n} = a + kh.$$

Le réel h est appelé le **pas** de la subdivision.



On prend alors pour liste d'abscisses la liste $[x_0, x_1, \ldots, x_n]$, qui contient les $n+1$ points de la subdivision.

Cette liste se construit de deux manières différentes.

⋆ À partir d'une liste vide, en ajoutant successivement à la liste les points de la subdivision :

```
h = (b - a)/n # h est le pas de la subdivision.
abscisses = []
for k in range(n + 1): # k va de 0 à n.
    abscisses.append(a + k * h)
```

⋆ En compréhension :

```
h = (b - a)/n # h est toujours le pas de la subdivision.
abscisses = [a + k * h for k in range(n + 1)]
```

1.3 Création de la liste des ordonnées

Une fois définie la liste `abscisses` des abscisses, on crée en compréhension la liste de toutes les images par la fonction f des points de la liste `abscisses` :

```
ordonnees = [f(x) for x in abscisses]
```

On peut alors tracer le graphe de la fonction f à l'aide de ces deux listes et de la fonction `plot`.

Testons ceci sur notre fonction préférée :

$$f : \mathbb{R} \longrightarrow \mathbb{R}$$
$$x \longmapsto \begin{cases} x^2 \sin\left(\dfrac{1}{x}\right) & \text{si } x \neq 0 \\ 0 & \text{sinon.} \end{cases}$$

Commençons par la définir en Python.

```
import numpy as np

def fonction(x):
    if x != 0:
        return (x ** 2) * np.sin(1 / x)
    else:
        return 0
```

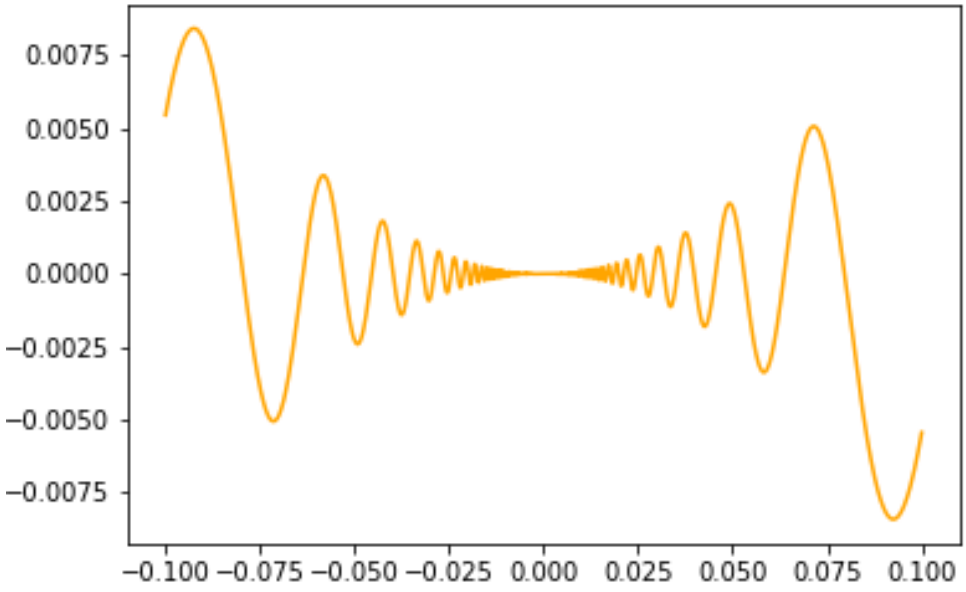
Traçons alors le graphe de la fonction f sur l'intervalle $\left[-\frac{1}{10}; \frac{1}{10}\right]$, avec mille-et-un points.

```
a = -0.1
b = 0.1
h = (b - a)/1000 # On divise l'intervalle en mille sous-segments.

abscisses = [a + k * h for k in range(1001)]

ordonnees = [fonction(x) for x in abscisses]

plt.plot(abscisses, ordonnees)
```



2 Matrices : tableaux à deux entrées

2.1 Définition

Une **matrice** est un tableau de nombres, à deux entrées, chaque élément de la matrice étant donc indexé par un numéro de ligne et un numéro de colonne.

Les matrices sont un objet très important en mathématiques, comme nous le verrons dans les chapitres d'algèbre linéaire. En informatique, elles apparaitront dans plusieurs contextes également, notamment pour représenter une image (voir la section 3) ou pour représenter un graphe.

2.1.1 Représentation informatique

En Python, une matrice est représentée comme la liste de ses lignes, chaque ligne étant elle-même une liste (possédant autant d'éléments que la matrice possède de colonnes).

Par exemple, la liste de listes suivante :

```
[[3, -2, 0, 1], [4, 9, -1, 3], [0, 2, -1, -7]]
```

représente la matrice $\begin{pmatrix} 3 & -2 & 0 & 1 \\ 4 & 9 & -1 & 3 \\ 0 & 2 & -1 & -7 \end{pmatrix}$.

2.1.2 Taille de la matrice

Si `LL` est une liste de listes représentant une matrice, alors on obtient la **taille** de la matrice correspondante de la manière suivante :

- ⋆ son nombre de lignes est la longueur de la liste `LL`, c'est-à-dire `len(LL)` ;
- ⋆ son nombre de colonnes est la longueur de la première ligne, c'est-à-dire `len(LL[0])`.

```
LL = [[3, -2, 0, 1], [4, 9, -1, 3], [0, 2, -1, -7]]

n = len(LL) # Nombre de lignes de la matrice.

LL[0]

p = len(LL[0]) # Nombre de colonnes de la matrice.
```

2.1.3 Création d'une matrice de taille fixée

Il est très courant d'avoir besoin, en initialisation d'un programme, de créer une matrice de taille fixée remplie de zéros, pour la modifier ensuite au fur et à mesure du programme (comme à la partie suivante).

Soient n et p deux entiers naturels non nuls. La matrice nulle à n lignes et p colonnes est représentée de la manière suivante :

```
[p * [0] for ligne in range(n)] # L'indice ligne prend n valeurs,
                               # de $0$ à $n - 1$.
```

En effet, cette liste contient n listes, chacune d'elles étant la liste `p * [0]`, qui contient p éléments tous égaux à zéro.

2.2 Accès aux éléments et modification d'éléments

Soient n et p deux entiers naturels non nuls.
Soit `LL` la représentation, sous forme de liste de listes, d'une matrice à n lignes et p colonnes.

- ⋆ Les lignes de la matrice `LL` sont indexées de 0 à $n-1$ et les colonnes de la matrice sont indexées de 0 à $p-1$.

⋆ Pour tout $i \in [\![0; n-1]\!]$, alors la liste

```
LL[i]
```

est la ligne d'indice i de la matrice.

⋆ Pour tous $i \in [\![0; n-1]\!]$ et $j \in [\![0; p-1]\!]$, l'élément

```
LL[i][j]
```

est le coefficient de la matrice situé sur la ligne d'indice i et la colonne d'indice j : en effet, c'est l'élément d'indice j dans la liste `LL[i]`, c'est-à-dire dans la ligne d'indice i.

Reprenons l'exemple précédent.

```
LL = [[3, -2, 0, 1], [4, 9, -1, 3], [0, 2, -1, -7]]

LL[1]

LL[1][2]

LL[0][0]
```

Comme pour une liste quelconque, on peut modifier directement les éléments d'une matrice.

```
LL = [4 * [0] for i in range(5)] # Matrice à 5 lignes et 4 colonnes.

LL

LL[2][1] = 14

LL

LL[0] = [2, 4, 6, 8] # Modification de la première ligne.
```

2.3 Parcours d'un tableau

Afin de parcourir tous les coefficients d'une matrice, on imbrique deux boucles : l'une dont la variable parcourt les indices des lignes de la matrice et l'autre dont la variable parcourt les indices de ses colonnes.
Le parcours d'une matrice représentée comme la liste `LL` de ses lignes s'écrit alors de la manière suivante.

```
n = len(LL) # Nombre de lignes de la matrice.
p = len(LL[0]) # Nombre de colonnes.

for i in range(n): # L'indice i va de 0 à n - 1, c'est-à-dire que
                   # i parcourt les indices des lignes de la matrice LL.
    for j in range(p): # L'indice j va de 0 à p - 1,
                       # j parcourt les indices des colonnes de LL.
        bloc d'instructions
        dépendant de l'élément LL[i][j]
```

Présentons deux exemples de parcours de tableau par boucles imbriquées.

2.3.1 Somme des éléments d'un tableau

Pour sommer tous les éléments d'un tableau, on crée une variable, initialisée à 0 (pour la somme vide), qui contiendra la valeur de la somme de tous les éléments rencontrés. On parcourt alors le tableau et on ajoute à la somme la valeur de chaque élément considéré.

```
def somme(LL):
    """
    Argument : un tableau LL de nombres,
               représenté comme liste de listes.
    Sortie : la somme de tous les éléments de LL.
    """
    n = len(LL) # Nombre de lignes.
    p = len(LL[0]) # Nombre de colonnes.
    somme = 0
    for i in range(n): # i va de 0 à n - 1 :
                       # i parcourt les indices de lignes de LL.
        for j in range(p): # j va de 0 à p - 1 :
                           # j parcourt les indices de colonnes de LL.
            somme += LL[i][j]
    return somme
```

Testons cette fonction.

```
somme([[-1, 3, 5], [6, 0, -2]])
```

2.3.2 Somme de deux matrices

Si A et B sont deux matrices de même taille (c'est-à-dire possédant le même nombre de lignes et le même nombre de colonnes), alors la matrice $A + B$ est la matrice obtenue en additionnant les matrices A et B coefficient par coefficient.

Par exemple, on a :

$$\begin{pmatrix} 1 & 3 & -2 \\ 4 & 0 & 5 \end{pmatrix} + \begin{pmatrix} 6 & -4 & 7 \\ -1 & 2 & -1 \end{pmatrix} = \begin{pmatrix} 7 & -1 & 5 \\ 3 & 2 & 4 \end{pmatrix}.$$

Pour calculer la somme de deux matrices A et B de même taille, on crée une matrice de même taille que les deux matrices A et B, initialement nulle, qui sera la somme des matrices A et B.

On parcourt ensuite tous les indices de lignes et tous les indices de colonnes des matrices (ce sont les mêmes pour les deux matrices, car elles ont la même taille) : à chaque position, le coefficient de la matrice-résultat devient la somme des coefficients à cette position dans A et B.

On retourne la matrice ainsi construite à la fin du parcours.

```
def somme_matricielle(A, B):
    """
    Entrées : deux matrices A et B de même taille,
              représentées comme listes de listes.
    Sortie : la matrice A + B.
    """
    nA = len(A) # Nombre de lignes de la matrice A.
    pA = len(A[0]) # Nombre de colonnes de la matrice A.
    nB = len(B) # Nombre de lignes de la matrice B.
    pB = len(B[0]) # Nombre de colonnes de la matrice B.

    # On teste si les matrices A et B ont bien même taille.
    if nA != nB or pA != pB:
        return "Erreur, les matrices doivent avoir même taille !"

    else: # Dans ce cas, A et B ont bien la même taille.
        S = [pA * [0] for i in range(nA)]
        # S est initialement la matrice nulle
        # à nA lignes et pA colonnes,
        # et sera la somme des matrices A et B.
        for i in range(nA): # i parcourt les indices
                            # des lignes de A et B.
            for j in range(pA): # j parcourt les indices
                                # des colonnes de A et B.
                S[i][j] = A[i][j] + B[i][j]
                # Le coefficient à la ligne i et à la colonne j
                # dans la matrice S devient la somme des coefficients
                # à la ligne i et la colonne j dans A et B.
        return S
```

Testons cette fonction sur l'exemple précédent.

```
A = [[1, 3, -2], [4, 0, 5]]

B = [[6, -4, 7], [-1, 2, -1]]

somme_matricielle(A, B)
```

3 Traitement d'images

3.1 Images numériques

Dans cette séance, nous manipulerons des images en niveaux de gris.

Une telle image est représentée par la matrice des ses pixels. À chaque pixel (*picture element*) de l'image est associé un réel entre 0 et 1 : la valeur 0 correspond à la couleur noire, la valeur 1 à la

couleur blanche et les valeurs intermédiaires aux différents niveaux de gris.

FIGURE 2 – Meet Barbara !

De même, une image en couleur sera représentée par une matrice de triplets, chaque triplet donnant le code RVB (*rouge, vert, bleu*) d'un pixel de l'image.

3.2 Lecture, visualisation et sauvegarde d'une image

3.2.1 Mise en place

On commence par enregistrer dans une même dossier le fichier Python et l'image que l'on souhaite étudier.
À l'aide de la commande `cd` (*current directory*), on définit dans la console le chemin d'accès jusqu'à votre dossier comme le chemin d'accès par défaut.

```
cd C:/BCPST1/Nom/Dossier
```

Pour accéder à un fichier, on utilisera alors son chemin d'accès relatif à ce dossier.

Pour la lecture et l'enregistrement d'images, on utilisera les fonctions suivantes.

- ⋆ Dans la bibliothèque `matplotlib.image` (c'est la bibliothèque de manipulation d'images de `matplotlib`) : les fonctions `imread` et `imsave`.
- ⋆ Dans la bibliothèque `matplotlib.pyplot` : la fonction `imshow`.

Importons alors ces bibliothèques :

```
import matplotlib.image as mpimg

import matplotlib.pyplot as plt
```

3.2.2 Lecture d'une image

La commande `imread` de la bibliothèque `matplotlib.image` permet de lire une image comme la matrice de ses pixels : elle prend en argument une chaine de caractères, qui est le nom du fichier à lire, et renvoie la matrice des pixels de l'image.

```
tableau = mpimg.imread("barbara.png")

tableau
```

On remarque que le résultat n'est pas tout à fait une liste de liste, mais un tableau `numpy`, de type `numpy.ndarray`. On peut le convertir en liste de listes de la manière suivante.

```
image = [list(tableau[i]) for i in range(len(tableau))]
```

Cette conversion n'est toutefois pas nécessaire, car la syntaxe des tableaux `numpy` est, pour ce qu'on a vu, similaire à celle des listes de listes.

```
# Les dimensions de Barbara :
len(image)
len(image[0])

image[0][0] # Le premier pixel de Barbara.
```

3.2.3 Affichage d'une image

La commande `imshow` de la bibliothèque `matplotlib.pyplot` permet d'afficher une image : elle prend en argument une matrice de pixels et elle affiche l'image correspondante.

```
plt.imshow(image, cmap = 'gray') # Affiche l'image en niveaux de gris.
```

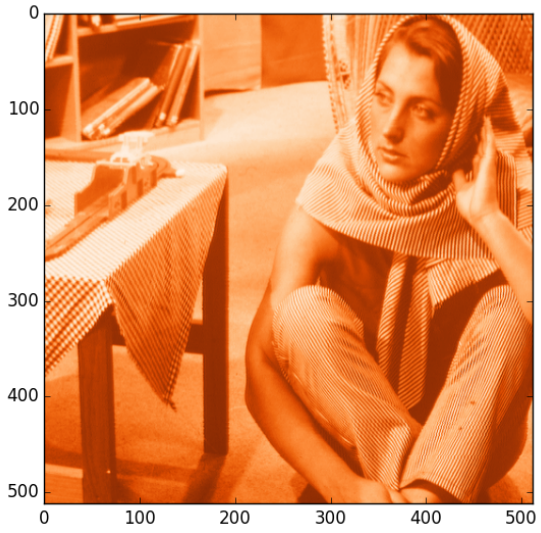


FIGURE 3 – Barbara en niveaux d'orange (`cmap = "Oranges_r"`).

3.2.4 Enregistrement d'une image

La fonction `imsave` de la bibliothèque `matplotlib.image` permet de faire l'opération inverse de la fonction `imread` : elle prend en arguments un nom de fichier et une matrice de pixels, et elle enregistre l'image correspondante au format PNG sous le nom de fichier donné.

```
image # L'image de Barbara.

image[0][0] = 0.14 # Espiègle, on modifie le premier pixel de Barbara !

mpimg.imsave("ann.png", image)

plt.imshow(mpimg.imread("ann.png"), cmap = 'gray') # Ça change tout !
```

3.3 Traitement d'images

Dans cette section et en TP, nous allons effectuer quelques transformations simples sur des images.

Les algorithmes que l'on présente s'opèrent tous sur les matrices de pixels correspondantes. Pour revenir à une image (au format PNG), on appliquera les fonctions de la sous-section précédente.

3.3.1 Négatif

Pour obtenir le négatif d'une image, on applique à chaque pixel la fonction suivante :

$$\begin{aligned} [0;1] &\longrightarrow [0;1] \\ x &\longmapsto 1-x, \end{aligned}$$

qui inverse le noir et le blanc.

```
def negatif(image):
    """
    Entrée : une matrice image,
             représentant une image en niveaux de gris.
    Sortie : la matrice du négatif de l'image.
    """
    n = len(image) # Nombre de lignes de l'image.
    p = len(image[0]) # Nombre de colonnes de l'image.

    neg = [p * [0] for i in range(n)] # neg sera le résultat,
    # une matrice de même taille que l'image (n lignes, p colonnes).

    for i in range(n): # L'indice i de ligne va de 0 à n - 1.
        for j in range(p): # L'indice j de colonne va de 0 à p - 1.
            neg[i][j] = 1 - image[i][j]

    return neg
```

3.3.2 Réduction

On souhaite réduire une image, en divisant sa longueur et sa largeur par un entier d.

La méthode la plus simple consiste à ne garder qu'une ligne sur d et qu'une colonne sur d dans l'image.

```
def reduction(image, d):
    """
    Entrées : une matrice image,
              représentant une image en niveaux de gris,
              un entier d.
    Sortie : la matrice de l'image réduite d'un facteur d
             en ligne et en colonne.
    """
    n = len(image) # Nombre de lignes de l'image.
    p = len(image[0]) # Nombre de colonnes de l'image.

    reduite = [(p // d) * [0] for i in range(n // d)]
    # On crée une matrice possédant d fois moins de lignes
    # et d fois moins de colonnes que la matrice de départ.

    for i in range(n // d): # i parcourt les indices de 0 à (n // d) - 1
                           # des lignes de la matrice reduite.
        for j in range(p // d): # j parcourt les indices de 0 à (p // d) - 1
                               # de ses colonnes.
            reduite[i][j] = image[d * i][d * j]

    return reduite
```

Une autre méthode consiste à donner à chaque pixel de l'image réduite la valeur moyenne des d^2 pixels correspondants dans l'image de départ (voir le TP7).