TP 7 : Numpy, Matplotlib, et des dessins récursifs

1 Introduction aux modules Numpy et Matplotlib

1.1 Module Numpy

- Importation : `import numpy as np`
- Module pour l'algèbre linéaire, basé sur une bibliothèque efficace. Les données sont des tableaux *homogènes*.
- Création : `np.zeros(5)`, `np.zeros((4,3))`, `np.zeros(8, dtype=np.int64)` (par défaut, flottant).
- Conversion : `np.array(..)`.
- Accès : `M[i]`, `M[i][j]` ou `M[i,j]`, etc...
- Modification similaire.
- Opérations vectorielles : `+`, `-`, `*`, `/` terme à terme entre tableaux de même taille.
- si `v` tableau et `x` scalaire, `x*v` nouveau tableau où les entrées sont multipliées par `x`, `x+v` nouveau tableau où les entrées sont translatées de `x`, etc..
- Pratiques : `np.linspace(a,b,n)` (tableau Numpy de n points régulièrement espacés dans $[a, b]$) et `np.arange(a,b,p)` (produit un tableau Numpy contenant les points $a, a+p, a+2p, \ldots, < b$).

1.2 Module Matplotlib

- Importation : `import matplotlib.pyplot as plt`
- Tracé (sans affichage) : `plt.plot(X,Y)`, avec `X` et `Y` de même taille.
- Affichage : `plt.show()`.

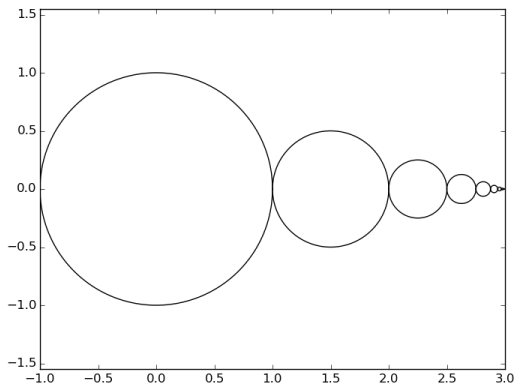
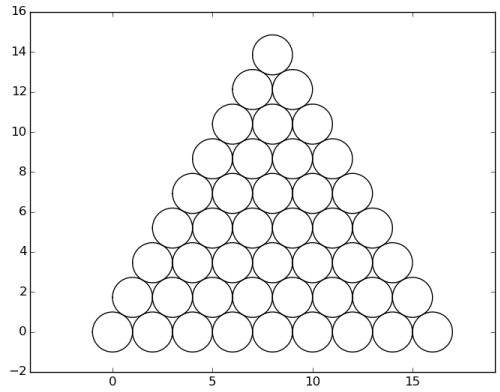
2 Quelques tracés

Sur le site web se trouve une annexe où les squelettes des fonctions à écrire sont précisées, ainsi que les importations suivantes :

```
import matplotlib.pyplot as plt
import numpy as np
fro math import *
```

Exercice 1. Écrire une fonction `cercle(x,y,r)` prenant en entrée trois réels, avec $r > 0$, et traçant (sans l'afficher) le cercle de rayon (x, y) et de rayon r. *Indication :* un paramétrage de ce cercle est $(x(t), y(t)) = (x + r\cos(t), y + r\sin(t))$ pour $0 \leq t \leq 2\pi$.

Exercice 2. À l'aide de la fonction précédente, écrire deux fonctions `fig1(n)` et `fig2(n)` permettant de produire des figures comme ci-dessous (on pourra commencer par `plt.axis('equal')` pour que les cercles ne soient pas des ellipses). *Remarque :* dans la figure de gauche, le cercle initial est de rayon 1, le suivant de rayon 1/2, etc... Dans la figure de droite, tous les cercles sont de rayon 1. Faire un peu de géométrie au brouillon avant d'écrire votre fonction !

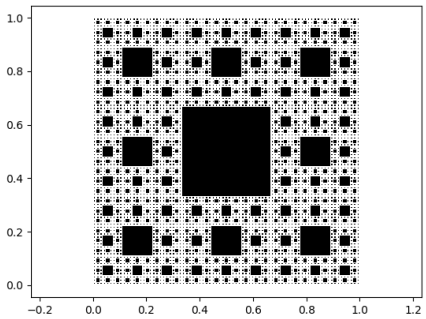



3 Fractales en Python

Le but de cette section est de composer des figures complexes, à l'aide de fonctions récursives.

3.1 Tapis de Sierpinski

Le tapis de Sierpinski est un objet fractal, dont une approximation est la suivante :



L'algorithme permettant d'obtenir cette figure au rang n (sur l'image, $n = 4$), est le suivant. Pour travailler sur le carré $[x, x + c] \times [y, y + c]$,

— on colorie le carré central de côté $c/3$;

— Si $n > 0$, on rappelle récursivement la fonction sur les 8 petits carrés de côté $c/3$ découpant le carré initial, en excluant le carré déja rempli, avec paramètre $n - 1$.

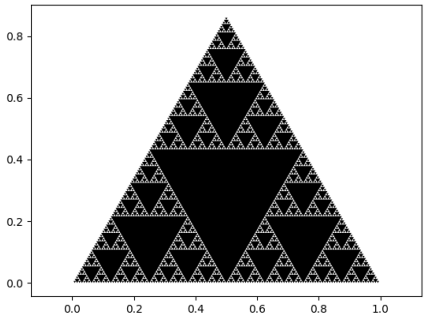
Exercice 3. Compléter le code suivant (présent dans l'annexe) pour obtenir une approximation du carré de sierpinski.

```
def tapis_sierpinski(n):
    plt.axis('equal')
    def aux(x,y,c,n): #travaille sur le carré [x,x+c]*[y,y+c]
        c2=c/3
        plt.fill([x+c2,x+c2,x+2*c2,x+2*c2],[y+c2,y+2*c2, y+2*c2, y+c2],"k") #carré central
        if n>0:
            ....
    aux(0,0,1,n) #appel à aux sur le carré [0,1]*[0,1]
    plt.show()
```

La fonction `plt.fill` permet de remplir un polygone, dont les abscisses et ordonnées sont décrites dans deux listes de même taille.

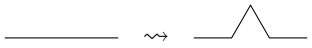
3.2 Triangle de Sierpinski

En suivant la même idée que pour le tapis de Sierpinski, produire la figure suivante (compléter la fonction `triangle_sierpinski` de l'annexe). *Remarque :* il est commode de travailler avec des tableaux Numpy, pour construire facilement des milieux ! Votre fonction auxiliaire prendra en entrée 3 sommets d'un triangle, ainsi que l'entier n.

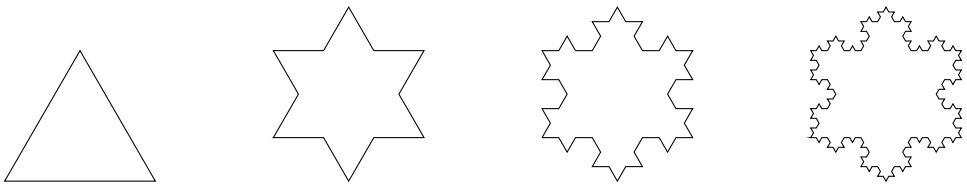


3.3 Flocon de Von Koch

Le but de cet exercice est de tracer une ligne brisée qui s'approche de l'objet fractal appelé le Flocon de Von Koch. La figure ci-dessous montre l'étape élémentaire de tracé de la ligne brisée : on remplace un segment de droite par le même segment, avec un triangle équilatéral qui a « poussé » du milieu vers l'extérieur.



Pour tracer le flocon, on part d'un triangle équilatéral, et on applique cette transformation sur ses trois côtés. On obtient ainsi une ligne brisée constituée de 12 segments. On réitère le procédé sur ces 12 segments, etc... La figure ci-dessous montre les étapes 0, 1, 2 et 3.

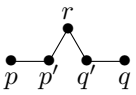


Étape 0 Étape 1 Étape 2 Étape 3

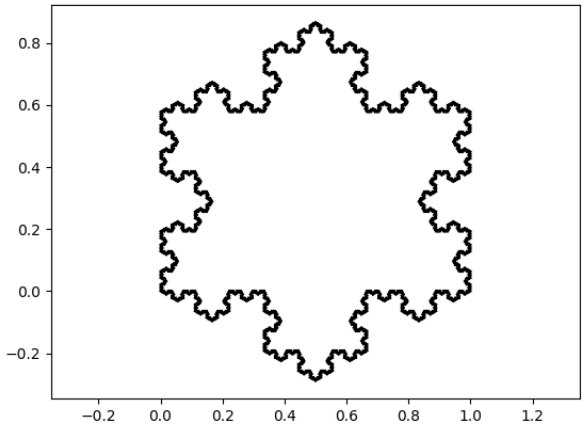
On se propose d'écrire une fonction récursive en Python, permettant de tracer la figure obtenue après n itérations, en partant d'un triangle initial de côté 1. La fonction `flocon(n)` suivante trace le flocon, à partir d'une fonction `koch(p,q,n)` qui travaille sur un segment. Écrire cette fonction.

```
def flocon(n):
    plt.axis("equal")
    def koch(p,q,n):
        ...
    p1,p2,p3 = np.array([0.,0.]), np.array([0.5,sqrt(3)/2]),np.array([1.,0.])
    koch(p1,p2,n)
    koch(p2,p3,n)
    koch(p3,p1,n)
    plt.show()
```

Indications : le cas de base de la fonction `koch(p,q,n)` ($n = 0$) consiste simplement à tracer le segment $[p, q]$. Sinon, Il s'agit de faire 4 appels récursifs. Le point le plus difficile à calculer est la *pointe* du triangle équilatéral qui pousse depuis le segment. On pourra observer que dans la figure suivante, le vecteur $\overrightarrow{p'r}$ s'obtient par rotation du vecteur $\overrightarrow{p'q'}$ d'un angle $\pi/3$, et que l'image du vecteur (x, y) par la rotation d'angle θ (autour de l'origine) est $(\cos(\theta)x - \sin(\theta)y, \sin(\theta)x + \cos(\theta)y)$



Voici le résultat de `flocon(5)` :



4 Une section pour les plus rapides : l'algorithme de Caslteljau

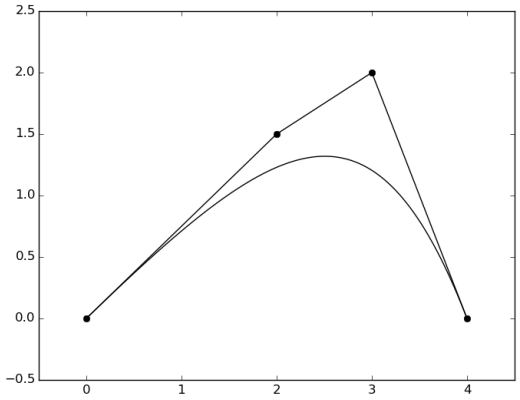


FIGURE 1: Une courbe de Bézier de degré 3

Si vous avez déja utilisé un logiciel basique de dessin [1], vous avez sûrement tracé des courbes à l'aide d'un outil pas facile à faire marcher : il faut donner deux points (« l'origine » et la « destination » de la courbe), ainsi que deux « points de contrôle » qui ne sont pas sur la courbe mais qui orientent sa forme. La figure 1 présente une telle courbe : on commence en $(0, 0)$ et on termine en $(4, 0)$, avec points de contrôle $(2, 1.5)$ et $(3, 2)$.

Bien qu'à ma connaissance, ce ne soit pas possible dans les logiciels basiques de dessin, on peut utiliser plus de deux points de contrôle. La figure 2 présente une courbe de Bézier à 6 points (dont 4 points de contrôle). Rien n'empêche les points de former un polygone non convexe, voir aussi la figure 2 : à droite, on a pris les mêmes points que pour la figure 1, en inversant la « destination » et le deuxième point de contrôle.

Passons maintenant à la définition des courbes de Bézier : pour n un entier naturel, on définit les polynômes de Bernstein de degré n comme :

$$B_{n,i}(t) = \binom{n}{i} t^i (1-t)^{n-i} \qquad \text{pour tout } i \text{ entre 0 et } n.$$

1. comme Paint !

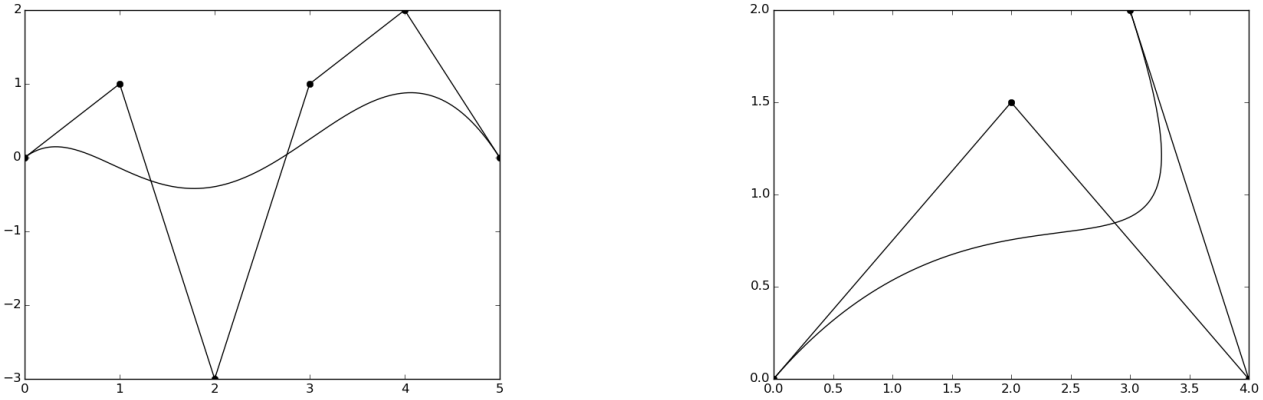


FIGURE 2: Une courbe de Bézier de degré 5, et une de degré 3 formée sur un polygone non convexe

Étant donnés $n+1$ points $P_0, \dots, P_n$ du plan, on définit la courbe de Béziers contrôlée par les (P_i) par

$$M(t) = \sum_{i=0}^{n} B_{n,i}(t) P_i \qquad \text{pour } t \in [0,1]$$

Ceci a bien un sens : comme $\sum_{i=0}^{n} B_{n,i}(t) = 1$, le point $M(t)$ est barycentre des P_i, avec les poids $B_{n,i}(t)$. Par exemple pour $n = 3$, on a

$$M(t) = (1-t^3)P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3$$

On remarque aussi que $M(0) = P_0$ et $M(1) = P_n$. De plus la tangente à $M(t)$ suit la direction [2] $\overrightarrow{P_0P_1}$ en $t = 0$, et la direction $\overrightarrow{P_{n-1}P_n}$ en $t = 1$.

S'il est possible de calculer des points de la courbe de Bézier via la formule précédente, on montre ici un algorithme récursif permettant d'obtenir une approximation de la courbe en ne calculant que des milieux : l'algorithme de Casteljau. L'algorithme est basé sur la propriété suivante, détaillée en figure 3

Soient donc P_0, P_1, P_2 et P_3 un ensemble de 4 points de $\mathbb{R}^2$. On considère la courbe de Bézier (de degré 3) définie par ses 4 points. Notons :

— M le milieu du segment $[P_1, P_2]$;
— A_1 le milieu du segment $[P_0, P_1]$;
— A_2 le milieu du segment $[A_1, M]$;
— B_2 le milieu du segment $[P_2, P_3]$;
— B_1 le milieu du segment $[M, B_2]$;
— $A_3 = B_0$ le milieu du segment $[A_2, B_1]$.

Alors la courbe de Bézier contrôlée par les points P_0, P_1, P_2 et P_3 est exactement la réunion des deux courbes de Bézier contrôlées par $A_0 = P_0$, A_1, A_2 et A_3 et par $A_3 = B_0$, B_1, B_2 et $B_3 = P_3$

Cette construction est l'algorithme de Casteljau. Remarquez que le point $A_3 = B_0$ appartient à la courbe (il correspond au point $M\left(\frac{1}{2}\right)$), et que la ligne brisée formée des deux suites de segments $[A_0, A_1, A_2, A_3]$ et $[B_0, B_1, B_2, B_3]$ est une approximation bien plus précise de la courbe que n'est la ligne brisée formée par les points $[P_0, P_1, P_2, P_3]$. On peut donc construire récursivement une approximation de la courbe de Bézier : tant que les segments de la ligne brisée sont de longueur supérieure à une certaine borne, on applique l'algorithme de Casteljau.

Question 1. Écrire une fonction `milieu(p,q)` prenant en entrée deux points (représentés par des tableaux Numpy de taille 2) et renvoyant le couple associé au milieu du segment $[p, q]$.

Question 2. En déduire une fonction `etape_casteljau(P)` prenant en entrée une liste de 4 points du plan (représentés par des tableaux Numpy) et retournant deux listes de la forme $[A_0, A_1, A_2, A_3]$ et $[B_0, B_1, B_2, B_3]$ comme détaillé dans l'algorithme de Casteljau, les éléments sont des tableaux Numpy de taille 2.

2. ce qui est cohérent avec les figures !

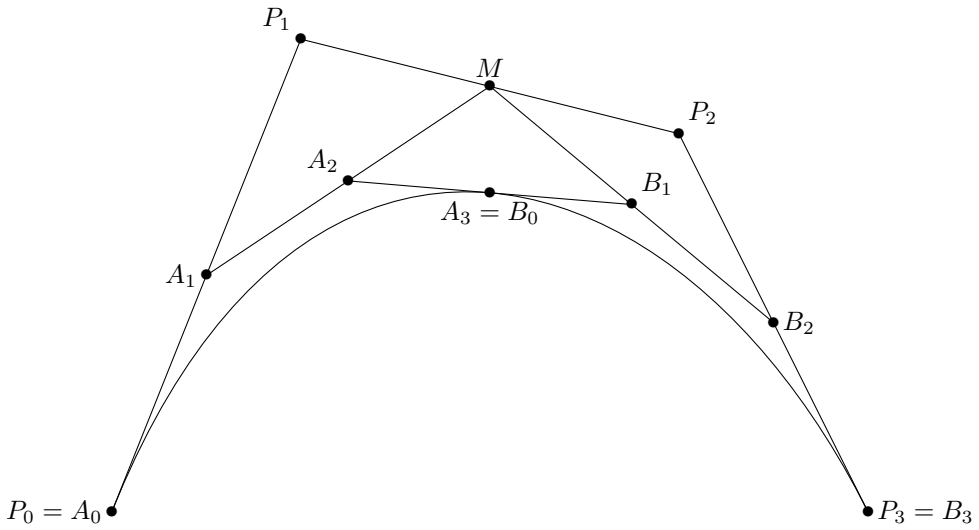


FIGURE 3: Une étape de l'algorithme de Casteljau

Question 3. En déduire une fonction (récursive) `bezier_casteljau(P)` prenant en entrée une liste de 4 points du plan (représentés par des tableaux Numpy) et traçant une approximation de la courbe de Bézier contrôlée par les points de `P`. Une condition d'arrêt sera par exemple la suivante : la distance entre chacun des trois couples de points successifs de `P` est inférieure à une certaine borne (comme 0.1). Dans ce cas on trace simplement la ligne brisée constituée des points de `P`. Ne pas hésiter à introduire des fonctions annexes : `distance`, `cond_arret`, etc...

> *Remarquez que les courbes de Bézier sont vraiment utilisées : toutes les lettres de ce texte sont en fait formées de courbes de Bézier ! Un intérêt est le fait que zoomer sur une lettre dans un fichier pdf ne produit pas de résultat tout « pixellisé » : les courbes de Bézier sont à la base du dessin dit « vectoriel ».*

```
>>> bezier_casteljau([np.array([0.,0.]), np.array([2.,1.5]), np.array([3.,2.]), np.array([4.,0.])])
>>> plt.show()
```

Le code précédent produit :