Récursivité en Python: TP

I. Algorithme de Casteljau pour le tracé de courbes de Bézier

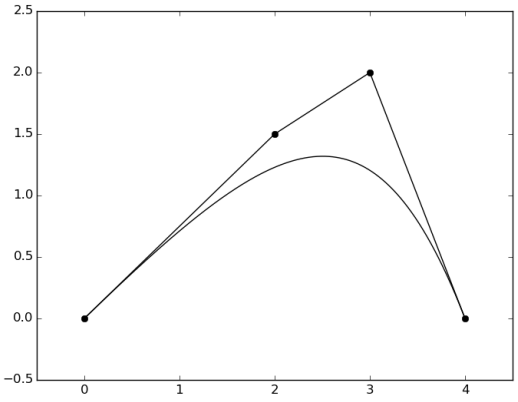

FIGURE 1: Une courbe de Bézier de degré 3

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.

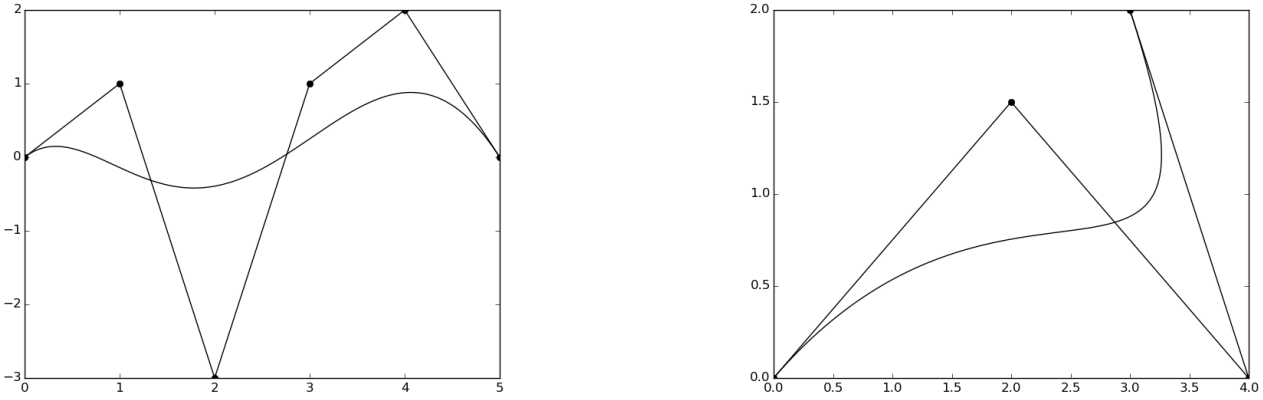

FIGURE 2: Une courbe de Bézier de degré 5, et une de degré 3 formée sur un polygone non convexe

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

1. comme Paint !

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

Ces polynômes sont très courants en mathématiques, vous avez d'ailleurs peut-être vus une démonstration du théorème de Weierstrass [2] basée sur ces polynômes.
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 [3] $\overrightarrow{P_0P_1}$ en $t = 0$, et la direction $\overrightarrow{P_{n-1}P_n}$ en $t = 1$.

Le but de cet exercice est de tracer des courbes de Bézier en se donnant un ensemble de points. Dans une première partie, on donne un tracé classique par calcul de points, dans la seconde on suit un algorithme récursif, qui effectue moins de calculs. Commencez par importer les modules :

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

I.1) Tracé basique

Question 1. Écrire une fonction `binome(n,p)` prenant en paramètre deux entiers n et p, et renvoyant $\binom{n}{p}$. On pourra utiliser la fonction `factorial(k)` pour le calcul de $k!$ (elle se trouve normalement dans le module `math`, dont on vient d'importer toutes les fonctions).

Question 2. En déduire une fonction `bernstein(n,i,t)` prenant en paramètres n et i deux entiers, ainsi que $t \in [0,1]$ et retournant $\binom{n}{i} t^i (1-t)^{n-i}$.

Question 3. Écrire une fonction `bezier(P)` prenant en paramètre une liste de points (une liste de couples, donc), et retournant une liste de 1000 points successifs sur la courbe de Bézier. On pourra utiliser `np.linspace(0,1,1000)` pour produire un tableau Numpy constitué de 1000 flottants régulièrement espacés dans $[0,1]$. On pourra de plus utiliser `np.array` pour convertir les points de `P` en tableaux Numpy, ce qui permet les opérations vectorielles. Par exemple avec `[(0,0), (2,1.5), (3,2), (4,0)]`, on obtient :

```
>>> bezier([(0,0), (2,1.5), (3,2), (4,0)])[0:5] #les 5 premiers points de la courbe !
[array([0., 0.]), array([0.006003, 0.0045015]), array([0.012, 0.00899697]),
array([0.01799099, 0.01348642]), array([0.02397599, 0.01796983])]
```

Question 4. Écrire une fonction `trace_ligne_brisee(L)` prenant en entrée une liste de points du plan, et reliant les points de `L` par des segments. Rappel : il suffit de répartir les absicsses et ordonnées dans deux listes `X` et `Y` et utiliser `plt.plot(X,Y)`. Jouer avec vos fonctions pour tracer des courbes de Bézier.

Question 5. Améliorer vos graphiques en traçant les points P_i et les segments P_iP_{i+1} (où les P_i sont seulement les points de contrôle). Étant données une liste d'abscisses `X` et une liste d'ordonnées `Y`, il suffit d'utiliser `plt.plot(X,Y)` pour relier les points (x_i, y_i) par une ligne brisée. `plt.plot(X,Y,'o')` « met des petits ronds » sur les points. `plt.plot(X,Y,'o-')` fait les deux. Vous pouvez mettre par exemple une couleur rouge en rajoutant `color="red"` ou `plt.plot(X,Y,'ro-')`

2. une fonction continue sur un segment est limite uniforme d'une suite de fonctions polynomiales. Pour f continue sur $[0,1]$, on montre que la suite des $t \mapsto \sum_{i=0}^{n} f(\frac{i}{n}) B_{n,i}(t)$ converge uniformément vers f lorsque n tend vers l'infini.
3. ce qui est cohérent avec les figures !

I.2) Algorithme de Casteljau

On décrit maintenant un algorithme capable de calculer très facilement des points d'une courbe de Bézier, sans avoir à prendre la valeur des polynômes de Bernstein en de multiples réels : la construction est très géométrique ! Cette technique mène à un algorithme récursif pour le tracé d'approximations de courbes de Bézier, en calculant seulement des milieux de segments et en traçant des lignes brisées. 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 $B_3 = P_3$

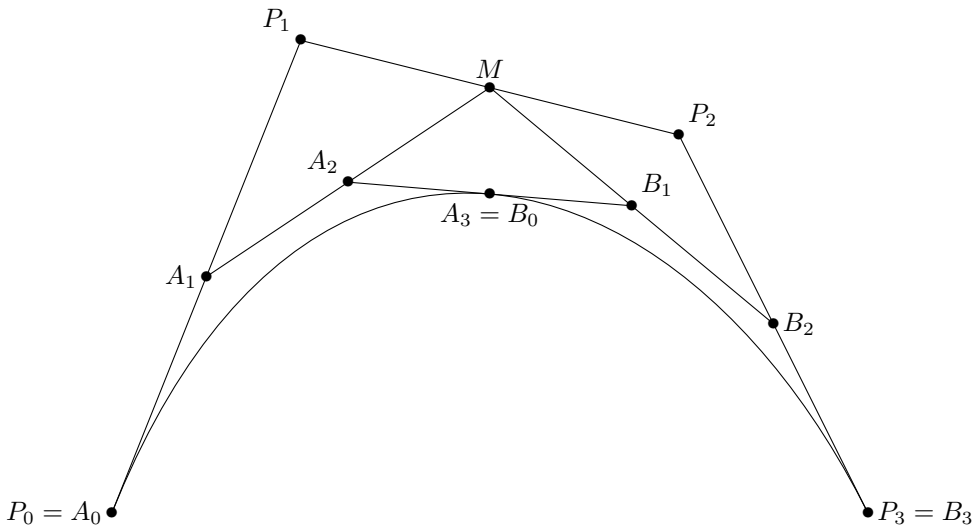

FIGURE 3: Une étape de l'algorithme de Casteljau

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 6. 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 7. 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.

Question 8. 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 deux 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`. Tester votre fonction avec des ensembles de 4 points convertis en tableaux Numpy !

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 ».

Résolution de Sudoku par backtracking

	9		2			6		5
3	2				7			
	7		9		5			8
	1							
		7					9	4
6								
		8						7
	3		4	9	1	5		
					3			

```
L=[[0, 9, 0, 2, 0, 0, 6, 0, 5], [3, 2, 0,
0, 0, 7, 0, 0, 0], [0, 7, 0, 9, 0, 5, 0, 0,
8], [0, 1, 0, 0, 0, 0, 0, 0, 0], [0, 0, 7,
0, 0, 0, 0, 9, 4], [6, 0, 0, 0, 0, 0, 0, 0,
0], [0, 0, 8, 0, 0, 0, 0, 0, 7], [0, 3, 0,
4, 9, 1, 5, 0, 0], [0, 0, 0, 0, 0, 3, 0, 0,
0]]
```

FIGURE 4: Une grille de Sudoku non encore remplie et sa représentation en Python

On se donne une grille de Sudoku sous la forme d'une liste de taille 9×9. Le but est d'y écrire des chiffres de $[\![1, 9]\!]$ de sorte que dans chaque ligne, chaque colonne, et chaque bloc de taille 3×3, chaque chiffre apparaisse une et une seule fois. Certaines cases sont naturellement déja remplies, et un Sudoku bien écrit ne possède normalement qu'une seule solution.

On va décrire une solution efficace au remplissage d'une grille de sudoku par backtracking. À partir de maintenant, la grille de sudoku est représentée en Python par une liste de listes, comme dans la figure précédente :

— la liste `L` est de taille 9.

— les « lignes » du sudoku sont les éléments de `L`, accessibles par `L[0]` (la ligne du haut) jusqu'à `L[8]`. Ce sont des listes.

— l'élément en case (i, j) (ligne i, colonne j, pour $0 \leq i, j \leq 8$, numérotées depuis le coin en haut à gauche) est accessible par `L[i][j]`. Par exemple, le 9 en haut du sudoku est l'élément `L[0][1]`.

— les cases non remplies sont associées au chiffre 0.

La technique du backtracking consiste simplement à essayer de remplir le sudoku en commençant par la première case jusqu'à la dernière. Si l'on découvre un conflit avec les règles, on est obligé de revenir en arrière. Le retour en arrière est considérablement simplifié par l'usage de la récursivité.

II.1) Détection de conflits

Question 1. Écrire une fonction `chiffres_ligne(L,i)` renvoyant la liste des nombres de 1 à 9 qui apparaîssent sur la ligne d'indice i. Par exemple, avec la grille initiale :

```
>>> chiffres_ligne(L,0)
[9, 2, 6, 5]
>>> chiffres_ligne(L,4)
[7, 9, 4]
```

Question 2. Faire de même avec `chiffres_colonne(L,j)`.

Question 3. Écrire maintenant une fonction `chiffres_bloc(L,i,j)` fournissant la liste des nombres de 1 à 9 apparaîssant dans le bloc de taille 3×3 auquel appartient la case (i, j). *Indication :* `i%3` fournit le résultat du reste dans la division euclidienne de i par 3.

```
>>> chiffres_bloc(L,4,7)
[9, 4]
>>> chiffres_bloc(L,4,5)
[]
```

Question 4. Déduire des questions précédentes une fonction `chiffres_conflit(L,i,j)` retournant la liste des chiffres qu'on ne peut pas écrire en case (i, j) sans contredire les règles du jeu. On ne se préoccupera pas du fait que `L[i][j]` puisse être dans la liste s'il est non nul, on n'appliquera cette fonction dans la suite que si `L[i][j]` est nul. Ça n'a aucune importance si certains chiffres apparaîssent plusieurs fois.

```
>>> chiffres_conflit(L,0,0)
[9, 2, 6, 5, 3, 6, 9, 3, 2, 7]
```

II.2) Passage à la case suivante

On essaye de remplir la grille par ligne croissante (de $i = 0$ à $i = 8$), puis par colonne croissante (de $j = 0$ à $j = 8$). En clair, on va d'abord essayer de mettre un chiffre en case $(0, 0)$, puis en case $(0, 1)$, etc... Si on a pu mettre un chiffre en case $(0, 8)$, on passe à la case $(1, 0)$, etc...

Question 5. Écrire une fonction `case_suivante(i,j)` permettant d'obtenir un couple d'indices indiquant les coordonnées de la case suivante.

II.3) La fonction principale

Question 6. On va donc écrire une fonction permettant de résoudre un Sudoku. Voici un squelette d'une fonction prenant en entrée une liste `L` représentant un Sudoku. Compléter-la !

```
def solution_sudoku(L):
    def aux(i,j):
        if i==9:
            [...]
        elif L[i][j]>0:
            [...]
        else:
            conflit=chiffres_conflit(L,i,j)
            [...]
         [...]
    return aux(0,0)
```

Explications : la fonction (récursive) `aux(i,j)` doit renvoyer `True` si l'on a réussi à compléter toute la grille à partir des hypothèses faites dans les cases précédant (i, j) (pour l'ordre de la question précédente) et `False` dans le cas contraire. Le principe est le suivant :

— il y a un cas de base tout en haut qui correspond au fait qu'on ait résolu le Sudoku ;

— si la case `L[i][j]` est déja remplie (c'était une des données du Sudoku), il n'y a rien à faire, on passe à la case suivante.

— sinon, on calcule les chiffres que l'on ne peut pas mettre en case (i, j), et on essaie successivement tous les autres : on écrit un chiffre en case (i, j) et on passe en case suivante par appel récursif. Déterminez ce qu'il faut faire suivant si l'appel récursif a renvoyé `True` ou `False`.

La fonction principale se contente de l'appel `aux(0,0)` : il faut essayer de tout remplir à partir du début !

8	9	1	2	3	4	6	7	5
3	2	5	6	8	7	4	1	9
4	7	6	9	1	5	3	2	8
9	1	4	7	5	2	8	6	3
2	5	7	3	6	8	1	9	4
6	8	3	1	4	9	7	5	2
1	4	8	5	2	6	9	3	7
7	3	2	4	9	1	5	8	6
5	6	9	8	7	3	2	4	1

FIGURE 5: La grille résolue, par backtracking.