Tris efficaces et courbes de Béziers

1 Tris efficaces

Exercice 1. *Visualisation de l'animation.* Télécharger le fichier `animation.py` sur le site web, et jouer avec. Il y a les tris du cours, et le tri par tas aussi. Attention à ne pas tout casser !

Exercice 2. *Comparaisons des tris.* Le but est de comparer les tris sur des listes générés aléatoirement. Travaillez avec le fichier `tri_cours.py`. La fonction `randint` du module `random` permet d'obtenir des entiers aléatoires : `randint(a,b)` fournit un entier aléatoire de $[\![a, b]\!]$.

```
>>> from random import randint
>>> randint(0,10000)
8310
```

La fonction `clock` du module `time` permet de mesurer le temps. Elle s'utilise comme suit pour mesurer le temps d'exécution d'un script :

```
from time import clock
t=clock()
[script]
t2=clock()-t #t2 contient le temps d'exécution du script.
```

Enfin, la fonction `plot` de `matplotlib.pyplot` permet de tracer des graphe, mais vous le saviez déja. Pour chacun des 5 tris du cours, et pour toute taille de liste t entre 10 et 200 par pas de 10, mesurer la moyenne sur 10 essais du temps d'exécution de chacun des tris sur une liste d'entiers aléatoires de $[\![0, 10000]\!]$ de taille t, et tracer le résultat. Voici un squelette pour faire ceci efficacement.

```
tris=[tri_selection, tri_insertion, tri_bulles, tri_rapide, tri_fusion]
noms_tris=["selection", "insertion", "bulles", "rapide", "fusion"]
temps_execution=[ [] for _ in range(5)] #liste contenant des listes de temps d'exécution
tailles=list(range(10,210,10)) #de 10 à 200 par pas de 10
for t in taille:
   for i in range(5):
       [...] #calcul des temps d'exécution
for i in range(5):
    plt.plot(tailles, temps_execution[i], label=noms_tris[i]) #tracé
plt.legend(loc="upper left") #positionnement de la légende.
plt.show()
```

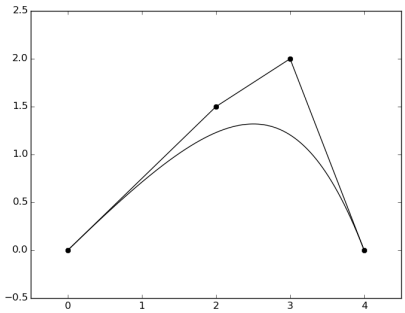

FIGURE 1: Une courbe de Bézier de degré 3

2 Introduction aux courbes de Bézier générales

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.

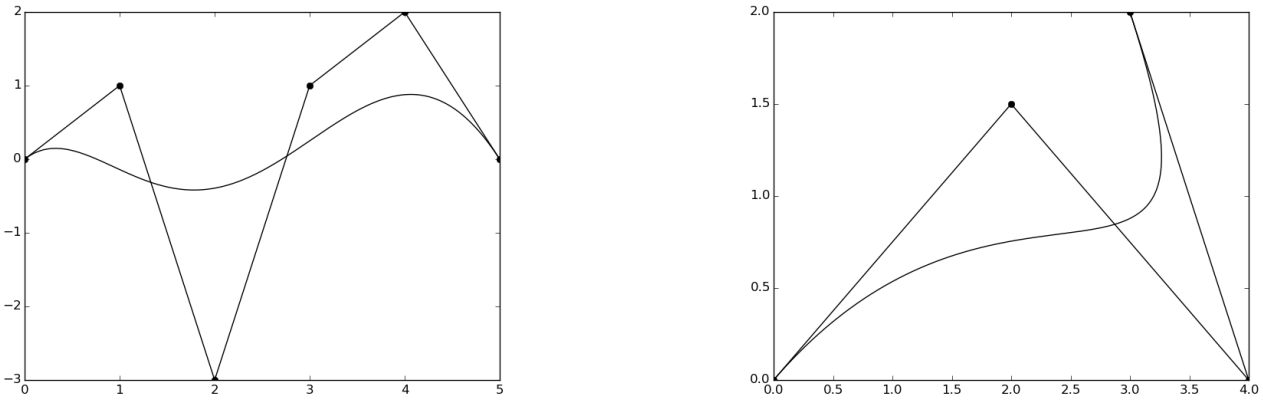

FIGURE 2: Une courbe de Bézier de degré 5, et une de degré 3 formée sur un polygone non convexe

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

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

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 paramétrée par les (P_i) par

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

Cette définition a bien un sens : comme $\sum_{i=0}^{n} B_{n,i}(t) = 1$ pour tout t, on exprime simplement que $M(t)$ est barycentre des P_i. De plus, $B_{n,i}(t) \geq 0$ pour $t \in [0, 1]$, donc $M(t)$ est dans l'enveloppe convexe formée par les points P_i. 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 cette partie est de tracer des courbes de Bézier en se donnant un ensemble de points. On commence par importer les modules dont on a besoin :

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

1. comme Paint !
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 !

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 (petite) liste de points P_i (une liste de couples, donc), et retournant deux listes X, Y, telles que X et Y soient constituées d'abscisses et d'ordonnées de 1000 points successifs de la courbe de Bézier associée aux points de `P`. On pourra utiliser `np.linspace(0,1,1000)` pour produire une liste (un tableau Numpy, plutôt) constitué de 1000 flottants régulièrement espacés dans $[0,1]$.

Question 4. En déduire comment tracer une courbe de Bézier, étant donnée une liste de points. L'appliquer par exemple à `[(0,0), (2,1.5), (3,2), (4,0)]` pour retrouver le résultat de la figure 1.

Question 5. Améliorer vos graphiques en traçant les points P_i et les segments P_iP_{i+1}. 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 verte en rajoutant `color="green"`.

3 Algorithme de Casteljau pour le tracé des courbes de Bézier de degré 3

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 construction suivante, détaillée en figure 3.

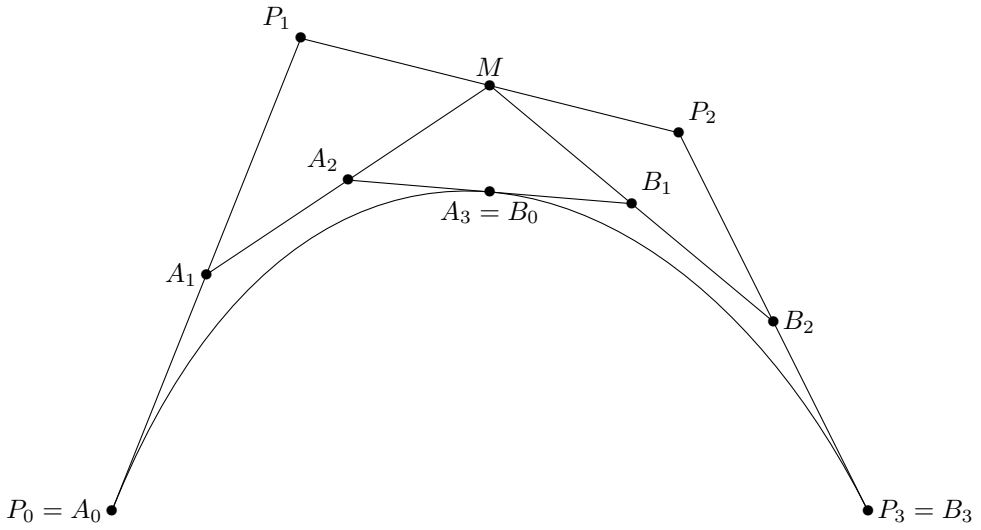

FIGURE 3: Une étape de l'algorithme de Casteljau

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 respectivement 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 6. Si ce n'est pas déja fait, écrivez une fonction `trace_ligne_brisee(P)` traçant les segments $[P_i, P_{i+1}]$ de la liste `P`.

Question 7. Écrire une fonction `milieu(p,q)` prenant en entrée deux points (représentés par des couples de flottants) et renvoyant le couple associé au milieu du segment $[p, q]$.

Question 8. En déduire une fonction `etape_casteljau(P)` prenant en entrée une liste de 4 points du plan (représentés par des couples de flottants) 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.

Question 9. En déduire une fonction (récursive) `bezier_casteljau(P)` prenant en entrée une liste de 4 points du plan (représentés par des couples de flottants) 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`.

Question 10. Démontrez que la courbe de Bézier initiale est bien l'union de celles contrôlées par $[A_0, A_1, A_2, A_3]$ et $[B_0, B_1, B_2, B_3]$.

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 ».