TP5 : comparaison d'algorithmes

La **comparaison d'algorithmes** cherche à répondre à la question suivante : Lorsqu'un même problème peut être résolu, soit en utilisant l'algorithme A, soit en utilisant l'algorithme B, *quel algorithme est préférable*, en termes de fiabilité, de rapidité de calcul, etc ? Deux approches sont possibles :

Sur un plan théorique. On doit chercher à comprendre le *nombre d'opérations mathématiques* nécessaires à chaque algorithme pour résoudre le problème, en fonction de la taille N des données en entrée. Ce domaine, à la croisée des mathématiques et de l'informatique, s'appelle l'**étude de complexité**.

Sur un plan pratique. On peut toujours implémenter les deux algorithmes, et regarder lequel est le plus rapide. (Ici aussi, en faisant varier la taille N des données en entrée.) Cette approche est plus concrète. Par contre, sa conclusion peut être biaisée car elle dépend aussi de la machine utilisée, du langage choisi, de la qualité de l'implémentation... C'est pourquoi, généralement, le mieux est de combiner les deux approches.

L'objectif de ce TP est de ***comparer les 2 algorithmes vus en TP pour tracer des courbes de Bézier***. Afin d'avoir tous la même base de travail, les 2 implémentations vous sont déjà fournies ci-après. Les 2 fonctions s'appellent de la même manière :

```
traceBezierXXXX(P,K)
```

où *XXXX* est le nom détaillé de chaque algorithme, et les arguments d'appel sont :

- Un np.array P contenant les coordonnées des $N+1$ points de contrôle $P_0, \dots, P_N$.
- Un nombre entier K gérant la *précision* requise pour le tracé. Par exemple : $K = 10 \rightarrow$ 10 valeurs de $t \rightarrow$ tracé imprécis. $K = 1000 \rightarrow$ 1000 valeurs de $t \rightarrow$ tracé très précis.

Algorithme A : Casteljau (TP2).

```
# algorithme de Casteljau calculant F_P(t) pour UNE valeur de t
def CasteljauUnPoint(t,P):
    P = P.copy()  # (nécessaire afin de ne pas écraser le P en dehors de la fonction)
    N = P.shape[0]-1
    for i in range(N):
        for j in range(N-i):
            P[j] = (1-t)*P[j] + t*P[j+1]
    return P[0]

# Calcul et traçage de F_P(t) en K valeurs différentes de t
def traceBezierCasteljau(P,K):
    allt = np.linspace(0,1,K)
    allFt = np.array( [CasteljauUnPoint(t,P) for t in allt] )
    plt.plot(allFt[:,0],allFt[:,1])
```

Algorithme B : Formule explicite de Bézier (TP4).

```
# Formule explicite calculant F_P(t) pour UNE valeur de t
from scipy.special import binom
def BezierFormuleUnPoint(t,P):
    N = P.shape[0]-1
    Ft = np.zeros(2)
    for i in range(N+1):
        Ft += binom(N,i) * t**i * (1-t)**(N-i) * P[i]
    return Ft

# Calcul et traçage de F_P(t) en K valeurs différentes de t
def traceBezierFormule(P,K):
    allt = np.linspace(0,1,K)
    allFt = np.array( [BezierFormuleUnPoint(t,P) for t in allt] )
    plt.plot(allFt[:,0],allFt[:,1])
```

Exercices

Exercice 1 (Sur papier : étude théorique de complexité).
Les 2 algorithmes étudiés effectuent le même genre d'opérations « de base », qui consistent à effectuer des *sommes pondérées entre points du plan*. Appellons donc **opération élémentaire** le fait de réaliser UN calcul du type

$$x.Q + y.R$$

où Q et R sont des points du plan donnés, et x et y des nombres réels. Le but de cet exercice est d'estimer **combien d'opérations élémentaires** sont nécessaires à chacun des 2 algorithmes.

1. Étude de l'algorithme A.
   (a) Combien d'opérations élémentaires sont réalisées par la fonction *CasteljauUnPoint*, en fonction de N l'ordre de la courbe ? (Faites un dessin, ça aide !)
   (b) Combien de fois la fonction *CasteljauUnPoint* est-elle appelée par la fonction *traceBezierCasteljau* ?
   (c) Déduisez-en le nombre total d'opérations élémentaires en fonction de N et K.
2. Étude de l'algorithme B.
   (a) Combien d'opérations élémentaires sont réalisées par la fonction *BezierFormuleUnPoint*, en fonction de N l'ordre de la courbe ?
   (b) Combien de fois la fonction *BezierFormuleUnPoint* est-elle appelée par la fonction *traceBezierFormule* ?
   (c) Déduisez-en le nombre total d'opérations élémentaires en fonction de N et K.

Exercice 2 (Étude empirique des temps de calcul).
Pour confirmer les prédictions théoriques de l'exercice 1, on va maintenant *mesurer les temps de calcul* des 2 algorithmes, et comment ils évoluent en fonction des nombres N et K.
Pour mesurer le temps de calcul associé à une série d'instructions, on peut appeler la fonction Python *time()* (qui se trouve dans le module du même nom), une fois *avant* et une fois *après* l'exécution des instructions en question :

```
from time import time
debut = time()
...(instructions)...
fin = time()
duree = fin-debut
```

Questions :

1. **Temps de calcul en fonction de N.** On veut mesurer les temps de calcul nécessaires aux ordres $N = 2, 5, 10, 20, 30, 40, 50$.
   (a) Pour chaque valeur de N testée, tirer $N+1$ points de contrôle de manière aléatoire. Tracer la courbe de Bézier correspondante avec l'algorithme A et mesurer le temps écoulé. Faire de même pour l'algorithme B.
   (b) Représenter la courbe du temps de calcul en fonction de N, pour chacun des 2 algorithmes, et déduire quel est l'algorithme le plus efficace.

   Conseils :
   - Utilisez une valeur pour K fixe et pas trop grande, par exemple $K = 100$.
   - Commentez la ligne *plt.plot()* des algorithmes, qui fausse les temps de calcul. Nous voulons estimer le temps pris par les *calculs*, et non par les opérations de traçage.

2. **Temps de calcul en fonction de K.** Même principe que la question 1, mais cette fois-ci avec N fixé (par exemple $N = 5$) et la précision K qui varie de 100 en 100 jusqu'à 1000.

3. Les courbes obtenues sont-elles cohérentes avec les prédictions de l'exercice 1 ?

4. Raffinement. Afin d'avoir des estimations plus fiables des temps de calcul, effectuez 10 traçages (par exemple) pour chaque valeur de N et K testée, et renvoyez le temps de calcul *moyen* sur ces 10 traçages.

Exercice 3 (pour les fans de Numpy uniquement : vectoriser son code).
Les implémentations en Python des 2 algorithmes présentés ci-dessus utilisent exclusivement des *boucles for*. Or lorsqu'on utilise Numpy, des gains de temps de calcul importants sont réalisables en écrivant son code de manière *vectorisée*. (Voir le TP4 pour l'explication détaillée.) Pour vous en convaincre, essayez d'écrire des versions vectorisées de l'algorithme B (comme expliqué au TP4) puis de l'algorithme A, et mesurez l'amélioration des temps de calcul.
La morale ici, c'est que le temps d'exécution ne dépend pas uniquement de l'algorithme choisi, mais aussi de la manière dont il a été codé. Il faut exploiter au mieux les possibilités de chaque langage !