TP4 : implémenter la formule explicite de Bézier

Vous avez pu découvrir lors du TD2 la formule explicite pour les courbes de Bézier :

Courbes de Bézier, formule explicite

$$F_{P_0,\dots,P_N}(t) = \sum_{k=0}^{N} \binom{N}{k} t^k (1-t)^{N-k} P_k \tag{1}$$

Le but de ce TP est de continuer à vous approprier cette formule, et de l'implémenter en Python.

Exercice 1 (Ordre 2).

1. *(Question papier)* Rappelez la formule explicite pour une courbe de Bézier d'ordre 2 :

$$F_{A,B,C}(t) = \dots$$

2. Écrivez une fonction Python qui implémente ce calcul, pour trois points de contrôle A, B, C au format Numpy, et *UN* nombre t entre 0 et 1 :

```python
def formuleBezier2(t,A,B,C):
    return TODO()
```

3. Testez votre fonction, en affichant la totalité d'une courbe de Bézier d'ordre 2 :

```python
A,B,C = np.array([0,1]),np.array([3,3]),np.array([5,0])  # (par exemple)
allt = np.linspace(0,1,100)      # toutes les valeurs de t testées
allFt = np.zeros((len(allt),2)) # va stocker le point F(t) en chaque valeur t testée
for i in range(len(allt)):
    allFt[i] = formuleBezier2(allt[i],A,B,C)
plt.figure()
plt.plot(TODO())                # affiche la courbe F
```

Exercice 2 (Ordre 3).

1. *(Question papier)* Rappelez la formule explicite pour une courbe de Bézier d'ordre 3 :

$$F_{A,B,C,D}(t) = \dots$$

2. Écrivez une fonction Python qui implémente ce calcul, pour quatre points de contrôle A, B, C, D au format Numpy, et *UN* nombre t entre 0 et 1 :

```python
def formuleBezier3(t,A,B,C,D):
    return TODO()
```

3. Testez votre fonction, en affichant la totalité d'une courbe de Bézier d'ordre 3 (sur le même modèle qu'à l'exercice précédent).

Exercice 3 (Ordre quelconque).

1. On rappelle la définition du coefficient binomial :

$$\binom{N}{k} = \frac{N.(N-1)\dots(N-k+1)}{k.(k-1)\dots 1}$$

   Écrivez une fonction Python *binom(N,k)* qui calcule ce coefficient binomial.

2. La formule explicite pour une courbe de Bézier à l'ordre N est :

$$\begin{aligned} F_{P_0,\dots,P_N}(t) &= (1-t)^N P_0 + \binom{N}{1} t(1-t)^{N-1} P_1 + \binom{N}{2} t^2(1-t)^{N-2} P_2 + \dots \\ &= \sum_{k=0}^{N} \binom{N}{k} t^k (1-t)^{N-k} P_k \end{aligned}$$

   Écrivez une fonction Python qui implémente ce calcul, pour un nombre arbritraire de points de contrôle stockés dans un np.array *P* (comme dans les TP précédents), et *UN* nombre t entre 0 et 1 :

```python
def formuleBezierGenerale(t,P):
    Ft = np.zeros(2)
    for k in TODO():
        TODO()
    return Ft
```

3. Testez votre fonction, en affichant la totalité d'une courbe de Bézier d'ordre N quelconque (sur le même modèle qu'à l'exercice précédent).

Exercice 4 (Vectoriser son code).

Python est un *langage interprété* (il n'y a pas de compilation avant l'exécution), ce qui se traduit concrètement par une certaine lenteur, comparé à un langage compilé comme le C.
Toutefois, Numpy permet de compenser cette lenteur en utilisant un style de codage appelé la **vectorisation**. L'idée est de remplacer le maximum de boucles *for* (qui sont du Python natif, donc lentes) par les fonctions « globales » de Numpy (addition, multiplication, etc.) qui agissent *directement sur l'ensemble d'un np.array*.
En effet, ces fonctions « globales » de Numpy font appel à des librairies extérieures précompilées très rapides (elles-mêmes codées en C), donc leur exécution se fait « à la vitesse du C ».

Voici un exemple. On se donne deux points A et B, et 100 valeurs de test t entre 0 et 1 :

```
A,B = np.array([0,1]),np.array([3,3])        # (par exemple)
allt = np.linspace(0,1,100)       # toutes les valeurs t à tester
```

Le but est de calculer un np.array $allFt$ de taille $(100, 2)$, qui contienne les 100 points

$$F(t) = (1 - t)A + tB$$

correspondant aux 100 valeurs de t testées.
Voici une version non vectorisée (=Python basique) de ce calcul :

```
allFt = np.zeros((len(allt),2)) # va stocker le point F(t) en chaque valeur t testée
for i in range(len(allt)):
    t = allt[i]
    allFt[i] = (1-t)*A + t*B
```

Et voici une version vectorisée (=Numpy) du même cacul :

```
allFt = (1-allt[:,None])*A[None,:] + allt[:,None]*B[None,:]
```

La version non vectorisée utilise une boucle for qui parcourt le np.array $allt$, alors que la version vectorisée utilise la multiplication $*$ « globale » définie directement sur la *totalité* du np.array $allt$.

Questions.

1. 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
   ```

   Utilisez cette technique pour mesurer le temps de calcul nécessaire à la version non vectorisée, et le temps de calcul nécessaire à la version vectorisée. Enfin, calculez le rapport des 2 temps de calcul. Que concluez-vous ?

2. Dans la version vectorisée, la commande

   ```
   allt[:,None]*B[None,:]
   ```

   utilise une spécificité de Numpy appelée le **broadcasting**[1]. En résumant,

   - le tableau $allt$ est mis en colonne puis « répété » autant de fois que nécessaire (ici, 2 fois) le long de la 2è dimension pour correspondre à la longueur de B.

1. https://numpy.org/doc/stable/user/basics.broadcasting.html

- le tableau B est mis en ligne puis « répété » autant de fois que nécessaire (ici, 100 fois) le long de la 1ère dimension pour correspondre à la longueur de $allt$.
- Le résultat, $allt[:,None]*B[None,:]$, possède donc 100 lignes (comme $allt$) et 2 colonnes (comme B, une abscisse et une ordonnée) et contient tous les produits possibles entre un élément de $allt$ et un élément de B.

(a) Pour mieux comprendre le mécanisme, affichez la taille ($shape$) et le contenu des tableaux suivants :

$$allt,\quad allt[:,None],\quad B,\quad B[None,:],\quad allt[:,None]*B[None,:]$$

(b) Que se passerait-il si vous tapiez simplement ?

```
allFt = (1-allt)*A + allt*B
```

3. En vous inspirant de l'exemple précédent, écrivez une fonction

```
def formuleBezierVectorisee(allt,P):
    TODO() #(plusieurs lignes)
    return allFt
```

qui calcule la formule générale de Bézier *directement sur le np.array* $allt$, avec du code vectorisé.

4. Comparez les temps de calcul des deux versions afin de calculer $allFt$ ($formuleBezierGenerale$ + boucle for de l'exercice 3, et $formuleBezierVectorisee$ de l'exercice 4). Que concluez-vous ?