TP2 : algorithme de Casteljau (cas général)

Élément de cours : l'algorithme de Castejau général

Rappel du TP1 : l'*algorithme de Casteljau* est une des méthodes classiques pour construire une courbe de Bézier à partir de ses points de contrôle. Lors du TP1, nous avions traité les cas $N = 1$ (2 points de contrôle), $N = 2$ (3 points de contrôle) et $N = 3$ (4 points de contrôle).
Le but de ce TP2 est d'implémenter la version *générale* de l'algorithme de Casteljau, pour n'importe quel ordre N de la courbe, autrement dit, pour n'importe quel nombre $N + 1$ de points de contrôle.
Voici la définition générale de l'algorithme, et une illustration :

Courbes de Bézier : algorithme de Casteljau

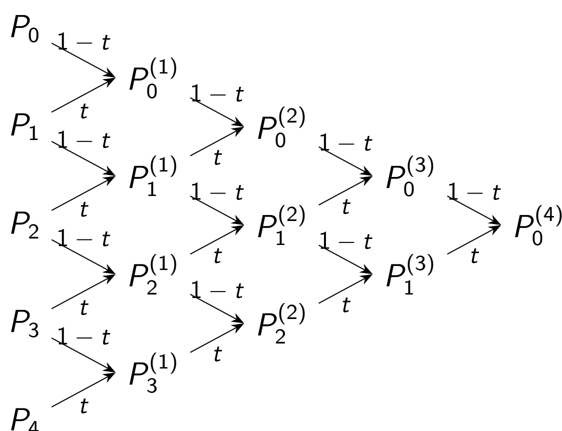
Soient $N + 1$ points de contrôle $(P_0, \dots, P_N)$, et un nombre $t \in [0, 1]$ *fixé*. L'algorithme suivant permet de calculer les coordonnées du point $F_{P_0,\dots,P_N}(t)$:

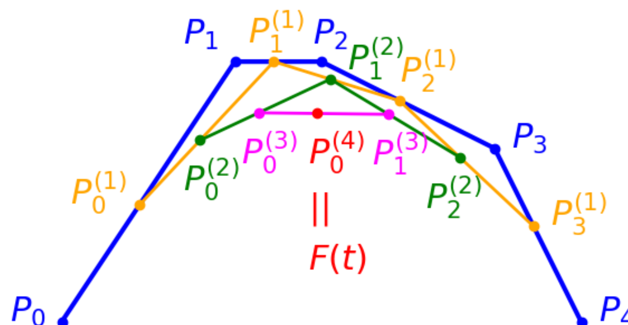
- À l'étape 1, les points de contrôle initiaux (P_k) sont recombinés pour former N nouveaux *points intermédiaires* notés $(P_k^{(1)})$, via la formule

$$P_k^{(1)} = (1 - t)P_k + tP_{k+1}$$

- À l'étape 2, les points $(P_k^{(1)})$ sont à leur tour recombinés, suivant la même formule, pour former $N - 1$ nouveaux points intermédiaires $(P_k^{(2)})$.
- À l'étape 3, les points $(P_k^{(2)})$ sont recombinés pour former $N - 2$ nouveaux points $(P_k^{(3)})$.
- ...
- À l'étape N, il ne reste plus qu'un unique point $P_0^{(N)}$. Il correspond à $F_{P_0,\dots,P_N}(t)$.

Ci-dessous, la procédure est illustrée pour $N = 4$, c'est-à-dire 5 *points de contrôle initiaux $P_0, \dots, P_4$:*





Exercices

Exercice 1 (Représentation des points avec Numpy)**.** Comme dans le TP précédent, un point du plan sera représenté par un *np.array* à 2 éléments. Par exemple, pour le point $A(-1,3)$:

```
A = np.array( [-1,3] )
```

Mais de plus, lorsqu'on implémente une version générale de l'algorithme de Casteljau, *le nombre de points de contrôle n'est pas connu d'avance*. Il nous faut donc un moyen de représenter un *ensemble* de points de contrôle, en nombre arbitraire. Pour cela, le choix suivant sera le plus pratique :

Consigne :
Un ensemble de $N+1$ points de contrôle sera représenté par un *np.array* de taille $(N+1,2)$.

Questions.

1. Rentrez, au format requis, l'ensemble de $N+1=4$ points de contrôle ci-dessous :

$$P_0(-1,3), \quad P_1(2,6), \quad P_2(7,8), \quad P_3(4,1)$$

```
P = TODO()
print(P)
# Vérification (simplement recopier et exécuter la ligne)
assert isinstance(P,np.ndarray) and P.shape==(4,2), "P n'a pas le bon format"
```

2. À partir du *np.array P* défini au-dessus, trouvez une commande Numpy qui renvoie le nombre contenu à la troisième ligne, première colonne du tableau P.

```
nombre = P[TODO()]
print(nombre)
assert nombre==7, "pas la bonne valeur"
```

3. À partir du *np.array P* défini au-dessus, trouvez une commande Numpy qui renvoie les deux coordonnées du point P_1 (2ème point de contrôle). Le résultat doit être au format que nous avons choisi pour représenter un point du plan, c'est-à-dire un *np.array* de taille $(2,)$.

```
P1 = P[TODO()]
print(P1)
assert np.array_equal(P1,np.array([2,6])), "mauvais format pour P1"
```

4. À partir du *np.array P* défini au-dessus, trouvez une commande Numpy, en une seule ligne, renvoyant la liste de tous les points P_i *sauf le dernier*. Le résultat doit être au même format, c'est-à-dire un *np.array* de taille $(N,2)$.

   Indice : il peut être utile de revoir le TP1 du cours Bases Maths 2, sur le *slicing* en Numpy.

```
Psaufdernier = P[TODO()]
print(Psaufdernier)
assert Psaufdernier.shape==(3,2), "taille (N,2) attendue pour Psaufdernier"
```

5. À partir du $np.array$ P défini au-dessus, trouvez une commande Numpy, en une seule ligne, permettant de renvoyer toutes les *abscisses* des points P_i. Testez-la sur plusieurs choix de P, en faisant varier le nombre de points.

```
P_x = TODO()
print(P_x)
```

6. Déduisez-en une commande Matplotlib permettant de représenter graphiquement le polygone de contrôle constitué des points P_i. Testez-la sur plusieurs choix de P, en faisant varier le nombre de points.

```
plt.plot(TODO())
```

Exercice 2 (Algorithme de Casteljau, cas général)**.** Le but de cet exercice est d'implémenter l'algorithme de Casteljau général, dont la description est donnée en première page. Les points de contrôle, ainsi que les points intermédiaires, seront représentés au format Numpy exposé à l'Exercice 1.

1. Écrivez une fonction qui implémente la première étape de l'algorithme de Casteljau :

```
def CasteljauEtape(t,P):
    return TODO()  # la nouvelle liste de points intermédiaires
```

   Cette fonction doit prendre en entrée un nombre réel $t \in [0, 1]$, et une liste de $N + 1$ points de contrôle $(P_k)_{k=0...N}$, stockés dans un $np.array$ P comme vu à l'Exercice 1.
   Elle doit renvoyer un $np.array$ représentant un ensemble de N points (c'est-à-dire un point de moins !), correspondant aux premiers points intermédiaires de l'algorithme de Casteljau (ceux notés $(P_k^{(1)})$ dans la description de l'algorithme de Casteljau en première page).

2. Déduisez-en une fonction qui implémente l'algorithme de Casteljau :

```
def bezierCasteljau(t,P):
    TODO()          # (implémentation libre)
    return TODO()   # le point F(t) recherché, qui correspond à P^(N)_0
```

   Vous êtes libres de choisir les détails de l'implémentation (boucle for, boucle $while$, ou appels récursifs). Testez votre fonction pour différents choix de t et des points de contrôle.

3. Rajoutez ***une*** ligne à votre fonction $bezierCasteljau$ afin de *visualiser* l'ensemble des points de contrôle intermédiaires impliqués dans la construction (comme dans la figure de droite en première page).

4. Vérification : choisissez un ensemble de $N+1 = 5$ points de contrôle, visualisez-les, et tracez la totalité courbe de Bézier $F(t)$ correspondante. Recommencez pour d'autres points de contrôle, avec plus ou moins de points (moins de 20 quand-même !).

5. *(Question facultative, pour gagner du temps vous pouvez traiter directement l'exercice 3.)* Effectuez une animation de la construction de la courbe et de ses points de contrôle, avec les mêmes consignes qu'au TP1, exercice 4.

Exercice 3 (Étude de complexité).

On rappelle que la **complexité** d'un algorithme est *l'estimation du nombre d'opérations requis pour cet algorithme*, en fonction de la taille des données en entrée.

Ici, la taille des données en entrée est N, l'ordre de la courbe à construire. Quant au nombre d'opérations à estimer, on le définit de la sorte : $V_N =$ nombre de calculs du type « $(1-t)A+tB$ » nécessaires afin de calculer le point $F(t)$ pour une courbe de Bézier d'ordre N.

1. En ce qui concerne la fonction *bezierCasteljau*, démontrez que $V_N = V_{N-1} + N$. Déduisez-en l'expression de V_N en fonction de N.
2. Un étudiant a proposé le code suivant pour construire une courbe de Bézier :

```
def bezierRec(t,P):
    if P.shape[0]==1:
        return P[0]
    else:
        return (1-t)*bezierRec(t,P[:-1]) + t*bezierRec(t,P[1:])
```

   Démontrez que ce code est correct mathématiquement, c'est-à-dire qu'il calcule bien le point $F(t)$ associé aux points de contrôle $P_0, \dots, P_N$. (En cas de doute, retournez voir le TD1.)
3. Démontrez que, pour ce code, le nombre d'opérations V_N vérifie $V_N = 2V_{N-1} + 1$. Déduisez-en la valeur de V_N en fonction de N. *(Vous pouvez ignorer le « +1 » pour simplifier la réponse.)*
4. Comparez la complexité de cet algorithme avec celle de *bezierCasteljau*. Lequel est le plus favorable ?
5. Vérification expérimentale : choisissez 15 points de contrôle et affichez la courbe de Bézier correspondante, d'abord avec la fonction *bezierCasteljau*, et ensuite avec la fonction *bezierRec*. Recommencez pour 20, puis 25 points de contrôle. L'une des deux implémentations est-elle plus rapide que l'autre ?