# Courbes de Bézier

## 1. Introduction

Les courbes de Bézier sous couramment utilisées pour tracer des courbes définies par des points de contrôle, en particulier dans les logiciels de CAO. Après avoir défini les courbes de Bézier, on verra comment les tracer au moyen de segments rectilignes.

Pour alléger les notations, on se limite aux courbes dans un espace à 2 dimensions (courbes planes), mais tous les résultats et algorithmes s'appliquent également dans un espace à 3 dimensions.

## 2. Définitions

### 2.a. Courbe paramétrée

Par convention, on notera de la manière suivante un point $P$ de l'espace à 2 dimensions dont les coordonnées dans un repère cartésien sont $(x, y)$ :

$$P = (x, y) \tag{1}$$

En python, on représentera un point par une liste `[x,y]`.

Une courbe paramétrée définie sur un intervalle $[a, b]$ est une application continue :

$$P : [a, b] \to R^2 \tag{2}$$

$$t \to P(t) = (x(t), y(t)) \tag{3}$$

Les fonctions $x(t), y(t)$ sont continues. Si elles sont de classe $C^1$, le vecteur dont les composantes sont $(x'(t), y'(t))$ est tangent à la courbe et varie continûment le long de la courbe. La courbe est alors de classe $C^1$. Si ces fonctions sont de classe $C^2$, le rayon de courbure varie continûment.

### 2.b. Courbes polynomiales

Une courbe paramétrée est *polynomiale de degré* $d$ si les fonctions $x(t), y(t)$ sont des polynômes de degré $d$. Une courbe cubique est une courbe polynomiale de degré $d = 3$.

### 2.c. Courbe de Bézier cubique

Une courbe de Bézier cubique est une courbe polynomiale de degré 3 définie par 4 *points de contrôle*, $(P_0, P_1, P_2, P_3)$. L'intervalle de définition est $[a, b] = [0, 1]$. La courbe commence au point $P_0$ (pour $t = 0$) et finit au point $P_3$ (pour $t = 1$). La dérivée au point $P_0$ est contrôlée par le point $P_1$. Le vecteur dérivée en $t = 0$ est :

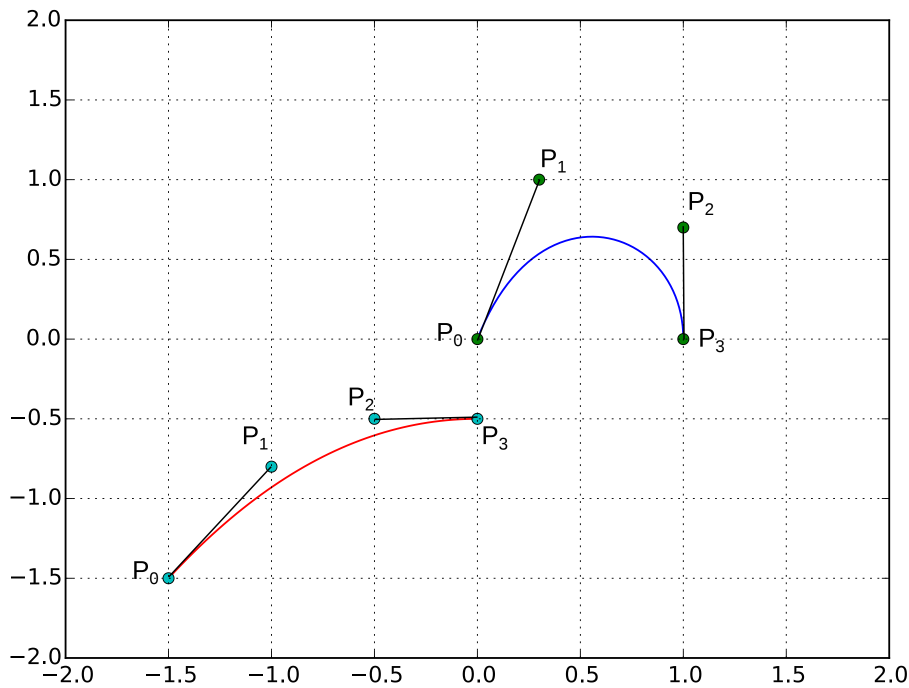
$$D_0 = 3(P_1 - P_0) \tag{4}$$

La droite $(P_0, P_1)$ est donc tangente à la courbe au point $P_0$.

De manière similaire, la dérivée au point $P_3$ est :

$$D_3 = 3(P_3 - P_2) \tag{5}$$

La figure suivante montre deux exemples de courbe de Bézier cubique :




On démontre ([1]) que l'unique courbe polynomiale de degré 3 vérifiant ces deux propriétés est :

$$P(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3 \tag{6}$$

Pour calculer un point de paramètre $t$ de la courbe, il suffit de calculer les coordonnées du point $P$ en utilisant la formule ci-dessus.

Il existe aussi une méthode géométrique pour obtenir ce point, appelé l'algorithme de *Casteljau*. On commence par considérer les trois segments $P_0P_1$, $P_1P_2$ et $P_2P_3$. Pour un paramètre $t \in (0,1)$, on calcule pour chacun de ces segments le barycentre défini par :

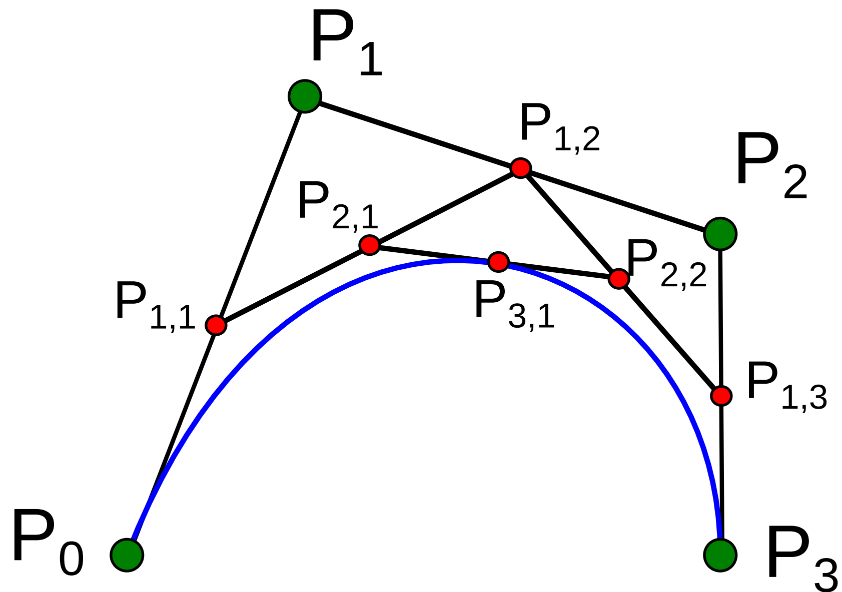
$$P_{1,i}(t) = (1-t)P_i + tP_{i+1} \tag{7}$$

La figure suivante montre la construction dans le cas $t = 0,5$. Les trois points ainsi obtenus définissent deux segments, dont on obtient les deux barycentres suivants :

$$P_{2,i}(t) = (1-t)P_{1,i} + tP_{2,i+1} \tag{8}$$

Ces deux points définissent un segment dont le barycentre suivant donne le point de paramètre $t$ sur la courbe de Bézier :

$$P(t) = P_{3,1}(t) = (1-t)P_{2,1} + tP_{2,2} \tag{9}$$




### 2.d. Courbe de Bézier d'ordre n

Une courbe de Bézier d'ordre $n$ est une courbe polynomiale de degré $n$ définie par $n+1$ points de contrôle. L'algorithme de Casteljau permet de définir une courbe de Bézier d'ordre n. Il suffit de calculer par récurrence les points suivants :

$$P_{j,i}(t) = (1-t)P_{j-1,i} + tP_{j-1,i+1} \quad (10)$$

À la n-ième itération, on obtient le point $P(t)$ de la courbe de Bézier.

Si l'on considère par exemple une courbe de Bézier d'ordre 5, elle est définie par 6 points de contrôle. Les dérivées première et seconde au point $P_0$ sont contrôlées par les points $P_1$ et $P_2$.

# 3. Échantillonnage d'une courbe de Bézier

## 3.a. Problème

On dispose d'une fonction graphique élémentaire permettant de tracer une ligne formée de segments rectilignes à partir d'une liste de points. La fonction `plot` de `matplotlib.pyplot` permet de faire ce type de tracé. Plus fondamentalement, les processeurs graphiques (GPU) disposent d'une fonction pour tracer très rapidement des segments rectilignes (les segments sont traités en parallèle) ; il est donc à la charge du programmeur de transformer une courbe de Bézier en segments rectilignes qui réalisent une approximation de cette courbe.

L'échantillonnage consiste à calculer les points de la courbe pour $N+1$ valeurs du paramètre $t$ :

$$t_0, t_1, ...t_N \tag{11}$$

avec $t_0 = 0$ et $t_N = 1$. On obtient ainsi $N$ segments rectilignes à tracer. Les points doivent être assez rapprochés pour que la ligne obtenue soit visuellement très proche de la courbe de Bézier.

## 3.b. Échantillonnage régulier

L'intervalle $[0,1]$ est divisé en $N$ intervalles égaux de longueur $1/N$ :

$$t_k = \frac{k}{N} \tag{12}$$

Pour une courbe de Bézier cubique, on peut utiliser directement la formule (6) pour calculer les points.

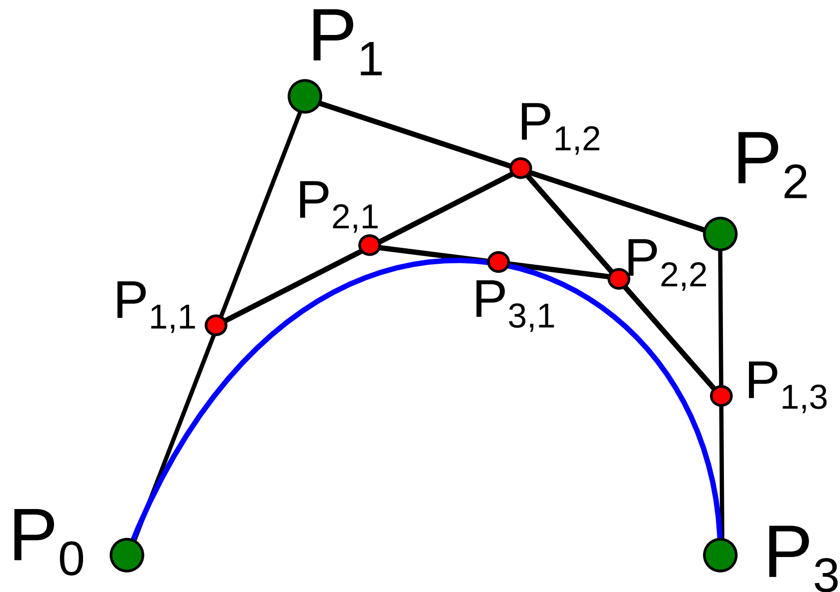
Plus généralement, pour une courbe de Bézier d'ordre $n$, on calcule chaque point en utilisant l'algorithme de Casteljau.

## 3.c. Échantillonnage récursif d'une courbe de Bézier cubique

L'inconvénient de la méthode précédente est la nécessité de choisir un $N$ assez grand pour que la ligne brisée obtenue soit assez proche de la courbe. Par exemple, $N = 100$ donne un bon résultat pour les courbes à forte courbure. Cependant, lorsque les 4 points de contrôle sont quasi alignés, la courbure est faible et un nombre plus petit de points est suffisant.

L'algorithme de Casteljau permet de définir un algorithme récursif, selon le principe *diviser pour régner* ([2]). Considérons le cas $t = 0,5$.

On démontre que la courbe comprise entre les points $P_0$ et $P_{3,1}$ est la courbe de Bézier cubique définie par les points de contrôle $P_0, P_{1,1}, P_{2,1}, P_{3,1}$. De même, la courbe comprise entre les points $P_{3,1}$ et $P_3$ est la courbe de Bézier cubique définie par les points de contrôle $P_{3,1}, P_{2,2}, P_{1,3}, P_3$.




La construction permet donc de scinder la courbe de Bézier en deux courbes de Bézier. L'opération est répétée récursivement. La récursion se termine lorsque les 4 points de contrôle peuvent être considérés comme assez proches de l'alignement pour qu'on puisse représenter la courbe par un segment rectiligne. Il faut donc définir une condition pour stopper la récursion. On peut pour cela calculer les vecteurs unitaires

$$\overrightarrow{u_{01}} = \overrightarrow{P_0P_1} \tag{13}$$

$$\overrightarrow{u_{12}} = \overrightarrow{P_1P_2} \tag{14}$$

$$\overrightarrow{u_{23}} = \overrightarrow{P_2P_3} \tag{15}$$

Pour une tolérance $\epsilon$, on peut stopper la récursion lorsque la condition suivante est remplie :

$$|2 - \overrightarrow{u_{01}} \cdot \overrightarrow{u_{12}} - \overrightarrow{u_{12}} \cdot \overrightarrow{u_{23}}| < \epsilon \tag{16}$$

En effet, les deux produits scalaires sont égaux à 1 lorsque les points sont alignés.

On doit cependant tenir compte du fait que les point $P_1$ et $P_2$ peuvent être confondus. Dans ce cas, on utilise la condition suivante :

$$|1 - \overrightarrow{u_{01}} \cdot \overrightarrow{u_{23}}| < \epsilon \tag{17}$$

Lorsque cette condition est remplie, on ajoute le point $P_0$ à la liste de points.

Cette méthode a l'avantage de fournir une densité de points élevée là où la courbe présente une forte courbure, une densité beaucoup plus faible là où la courbe est assimilable à un segment rectiligne. La méthode peut facilement se généraliser aux courbes de Bézier de degré $n$.

## 4. Implémentation en Python

Voici les modules à importer pour ce travail :

```
import math
from matplotlib.pyplot import *
```

### 4.a. Courbe de Bézier cubique, calcul direct

Les points sont représentés par une liste `[x,y]`.

On commence par définir deux fonctions pour effectuer les combinaisons linéaires des coordonnées de deux points. Par convention, les points sont notés en majuscules, les nombres réels en minuscule.

- `combinaison_lineaire(A,B,u,v)` : renvoie le point $uA + vB$.
- `interpolation_lineaire(A,B,t)` : renvoie le point $tA + (1 - t)B$.

La fonction `point_bezier_3(points_control,t)` calcule et renvoie le point de paramètre $t$ pour une courbe de Bézier définie par 4 points de contrôle. Ceux-ci sont fournis sous forme d'une liste de points. Le calcul est fait en utilisant la formule (6). En veillera à minimiser le nombre de multiplications effectuées au cours de ce calcul.

La fonction `courbe_bezier_3(points_control,N)` calcule les points de la courbe de Bézier en subdivisant l'intervalle $[0, 1]$ en $N$ intervalles égaux. La fonction renvoie une liste de points, qui doivent être bien sûr classés par valeurs de $t$ croissantes.

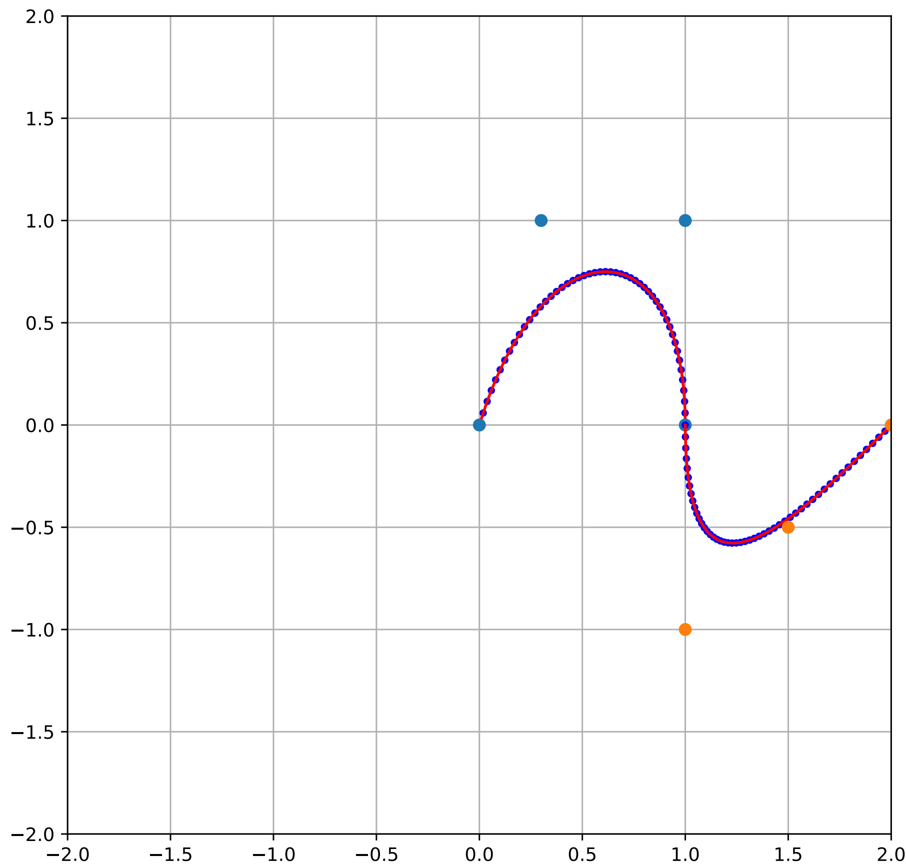
La liste de points obtenue avec la fonction précédente ne peut pas être tracée directement avec la fonction `plot` de `matplotlib.pyplot`. On écrit donc une fonction `plot_points(points_co`
Le style est ".", "o" ou "-" (tracé de segments).

```
def combinaison_lineaire(A,B,u,v):
    return [A[0]*u+B[0]*v,A[1]*u+B[1]*v]

def interpolation_lineaire(A,B,t):
    return combinaison_lineaire(A,B,t,1-t)

def point_bezier_3(points_control,t):
    x=(1-t)**2
    y=t*t
    A = combinaison_lineaire(points_control[0],points_control[1],(1-t)*x,3*t*x)
    B = combinaison_lineaire(points_control[2],points_control[3],3*y*(1-t),y*t)
    return [A[0]+B[0],A[1]+B[1]]

def courbe_bezier_3(points_control,N):
    if len(points_control) != 4:
        raise SystemExit("4 points de controle")
```

```
    dt = 1.0/N
    t = dt
    points_courbe = [points_control[0]]
    while t < 1.0:
        points_courbe.append(point_bezier_3(points_control,t))
        t += dt
    points_courbe.append(points_control[3])
    return points_courbe

def plot_points(points_courbe,style='-'):
    x = []
    y = []
    for p in points_courbe:
        x.append(p[0])
        y.append(p[1])
    plot(x,y,style)
```

Voici un exemple avec deux courbes de Bézier représentées par 50 segments rectilignes :

```
figure(figsize=(8,8))
P0 = [0,0]
P1 = [0.3,1]
P2 = [1,1]
P3 = [1,0]
points = courbe_bezier_3([P0,P1,P2,P3],50)
plot_points(points,style='b.')
plot_points(points,style='r-')
plot_points([P0,P1,P2,P3],style='o')
P4 = [1,-1]
P5 = [1.5,-0.5]
P6 = [2,0]
points = courbe_bezier_3([P3,P4,P5,P6],50)
plot_points(points,style='b.')
plot_points(points,style='r-')
plot_points([P4,P5,P6],style='o')
axis([-2,2,-2,2])
grid()
```




Le point $P_3$ de la première est confondu avec le point $P_0$ de la seconde. Les deux courbes sont donc raccordées en ce point. On a choisit les points de contrôle de part et d'autre de ce point commun pour que la dérivée première soit identique de part et d'autre (raccord de classe $C^1$). En respectant cette condition, il est donc possible de contruire une courbe de classe $C^1$ en raccordant plusieurs courbes de Bézier cubiques. Les logiciels de dessin (par exemple Inkscape) ont une fonction permettant de tracer directement à la souris des courbes de Bézier cubiques raccordées de cette manière.

### 4.b. Courbe de Bézier d'ordre n, algorithme de Casteljau

L'algorithme de Casteljau repose sur une opération de réduction, qui consiste à appliquer la transformation suivante pour $N$ points (indice i variant de $0$ à $N-2$).

$$P_{j,i}(t) = (1-t)P_{j-1,i} + tP_{j-1,i+1} \tag{18}$$

On obtient ainsi $N-1$ points qui sont des interpolations linéaires de paramètre $1-t$ sur les segments.

Initialement, il y a $n+1$ points de contrôle. Une première réduction donne $n$ points. La seconde réduction donne $n-1$ points. On continue les réductions jusqu'à obtenir un seul point : on obtient ainsi le point de paramètre $t$ de la courbe de Bézier.

La fonction reduction(points_control,t) effectue une réduction et renvoie les points sous forme d'une liste.

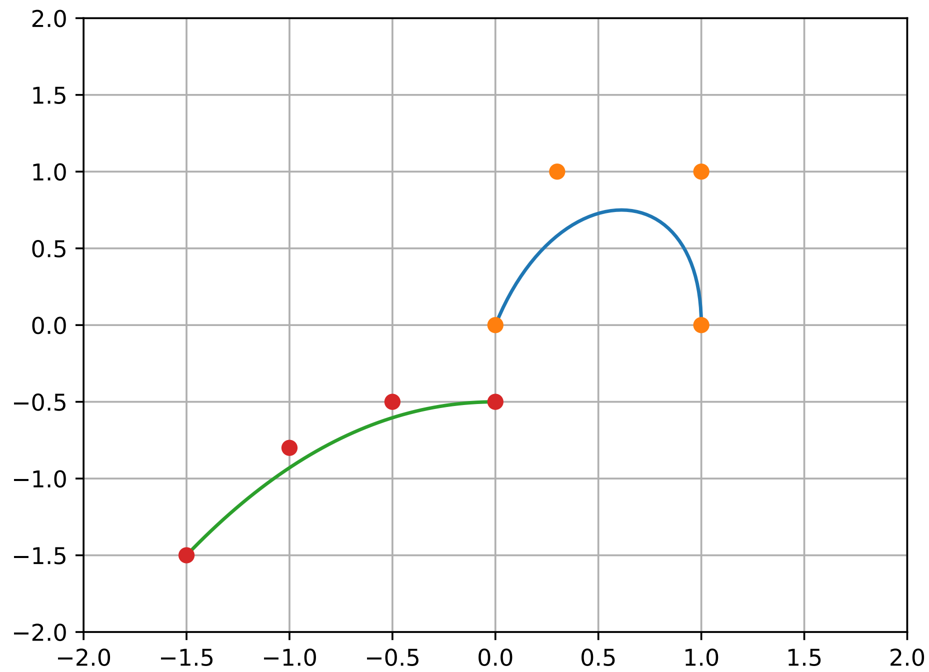
La fonction point_bezier_n(points_control,t) calcule et renvoie le point de paramètre $t$ pour une courbe de Bézier définie par les points de contrôle fournis en argument. Le degré de la courbe est égal à la taille de la liste de points de contrôle moins un.

La fonction courbe_bezier_n(points_control,N) calcule les points de la courbe de Bézier en subdivisant l'intervalle $[0, 1]$ en $N$ intervalles égaux.

```
def reduction(points_control,t):
    points_sortie=[]
    N = len(points_control)
    for i in range(N-1):
        points_sortie.append(interpolation_lineaire(points_control[i],points_control[i+1
    return points_sortie

def point_bezier_n(points_control,t):
    n = len(points_control)
    while n > 1:
        points_control = reduction(points_control,t)
        n = len(points_control)
    return points_control[0]

def courbe_bezier_n(points_control,N):
    n = len(points_control)
    dt = 1.0/N
    t = dt
    points_courbe = [points_control[0]]
    while t < 1.0:
        points_courbe.append(point_bezier_n(points_control,t))
        t += dt
    points_courbe.append(points_control[n-1])
    return points_courbe
```

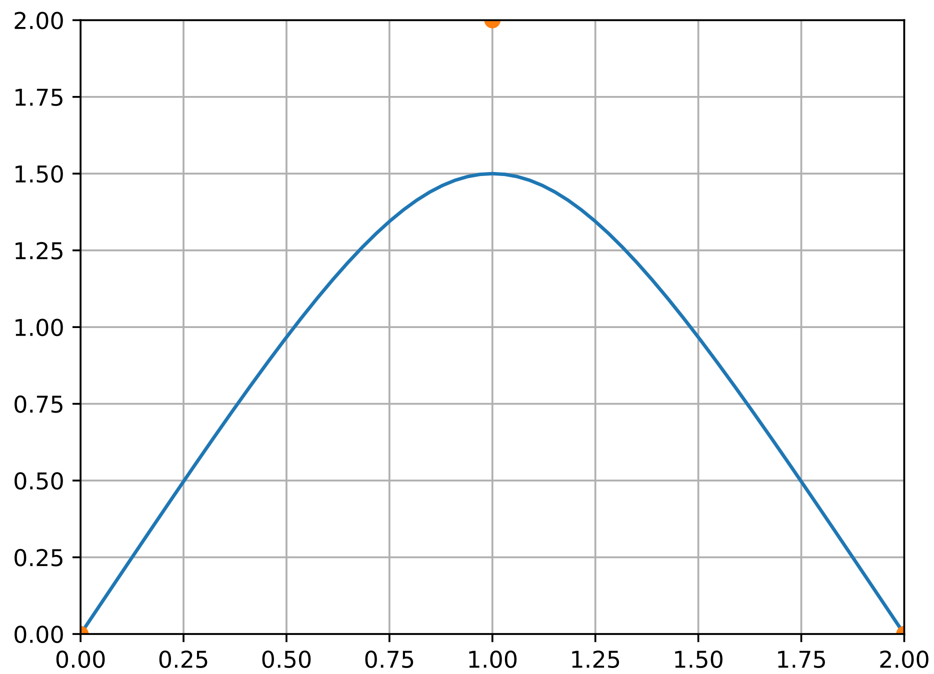
Voici comme premier exemple une courbe de Bézier cubique :

```
figure()
plot_points(courbe_bezier_n([P0,P1,P2,P3],50),style='-')
plot_points([P0,P1,P2,P3],style='o')
A0=[-1.5,-1.5]
A1=[-1,-0.8]
A2=[-0.5,-0.5]
A3=[0,-0.5]
plot_points(courbe_bezier_n([A0,A1,A2,A3],50),style='-')
plot_points([A0,A1,A2,A3],style='o')
axis([-2,2,-2,2])
grid()
```



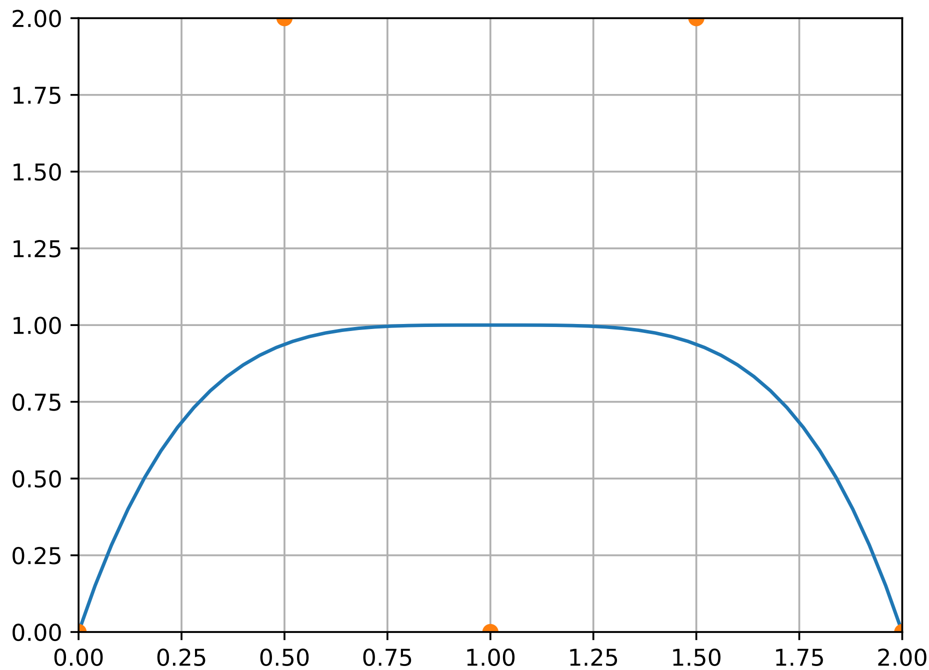

Il est possible d'utiliser deux points de contrôle confondus, comme dans cet exemple :

```
P0=[0,0]
P1=[1,2]
P2=[1,2]
P3=[2,0]
figure()
plot_points(courbe_bezier_n([P0,P1,P2,P3],50),style='-')
plot_points([P0,P1,P2,P3],style='o')
axis([0,2,0,2])
grid()
```




En gardant les deux points extrêmes, on ajoute un point de contrôle pour obtenir une courbe de Bézier d'ordre 4 :

```
P0=[0,0]
P1=[0.5,2]
P2=[1,0]
P3=[1.5,2]
P4=[2,0]
figure()
plot_points(courbe_bezier_n([P0,P1,P2,P3,P4],50),style='-')
plot_points([P0,P1,P2,P3,P4],style='o')
axis([0,2,0,2])
grid()
```




### 4.c. Courbe de Bézier cubique, algorithme récursif

En commence par écrire deux fonctions qui serviront pour détecter la fin de la récursion.

La fonction `vecteur_unitaire(P1,P2)` renvoie le vecteur unitaire défini par deux points. Si les deux points sont confondus, elle renvoie `False`.

La fonction `test_alignement_4pts(points,epsilon)` teste les 4 points fournis dans la liste `points` avec la condition (16) ou (17) . Elle revoit `True` si la condition est vérifiée, `False` sinon.

La fonction `division_courbe_bezier_3(points_control)` divise la courbe de Bézier cubique en deux courbes de Bézier cubiques, en utilisant la construction de Casteljau avec $t = 0,5$. Elle renvoie `(points_control_1,points_control_2)`, c'est-à-dire les deux listes de points de contrôle.

La fonction `courbe_bezier_3_recursif(points_control,epsilon,pile_points_cou` implémente l'algorithme récursif. Elle scinde la courbe de Bézier en deux parties (en appelant la fonction précédente), puis s'appelle récursivement sur ces deux parties. La récursion est stoppée lorsque les 4 points sont assez proches de l'alignement (avec la tolérance `epsilon`). Le point $P_0$ est alors ajouté à la pile de points transmise en troisième argument.

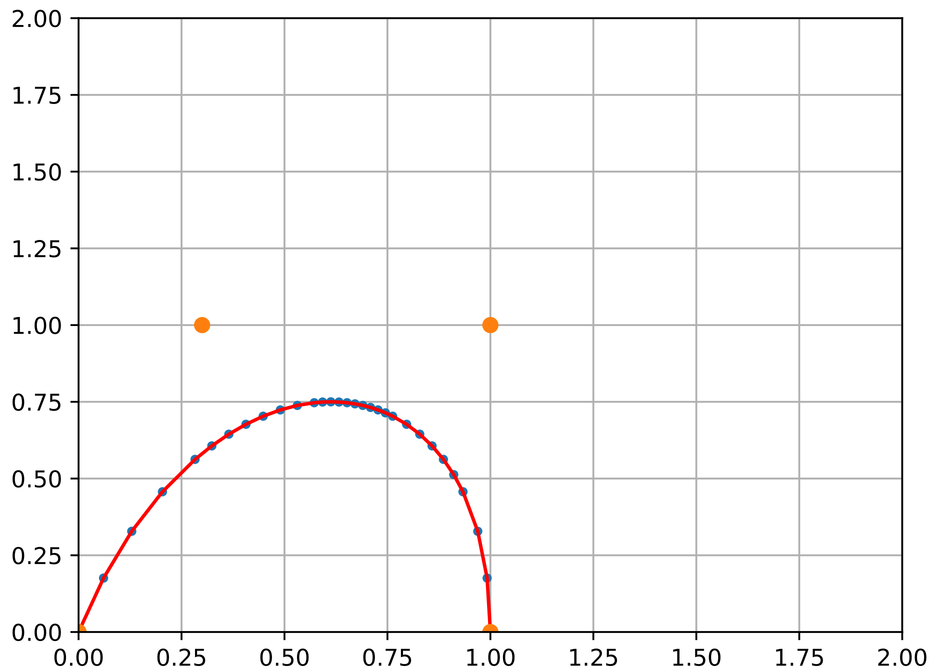
La fonction `courbe_bezier_3_recursif_init(points_control,epsilon)` démarre la récursion. Elle doit donc créer une pile de points vides, et renvoyer cette pile de points.

```
def vecteur_unitaire(P1,P2):
    Ux = P2[0]-P1[0]
    Uy = P2[1]-P1[1]
    norme = math.sqrt(Ux*Ux+Uy*Uy)
    if norme!=0:
        return [Ux/norme,Uy/norme]
```

```
    else:
        return False

def test_alignement_4pts(points,epsilon):
    U1 = vecteur_unitaire(points[0],points[1])
    U2 = vecteur_unitaire(points[1],points[2])
    U3 = vecteur_unitaire(points[2],points[3])
    if U2:
        x = 2.0-(U1[0]*U2[0]+U1[1]*U2[1]+U2[0]*U3[0]+U2[1]*U3[1])
    else:
        x = 1.0-(U1[0]*U3[0]+U1[1]*U3[1])
    if abs(x) < epsilon:
        return True
    else:
        return False

def division_courbe_bezier_3(points_control):
    P01 = interpolation_lineaire(points_control[0],points_control[1],0.5)
    P12 = interpolation_lineaire(points_control[1],points_control[2],0.5)
    P23 = interpolation_lineaire(points_control[2],points_control[3],0.5)
    P01_12 = interpolation_lineaire(P01,P12,0.5)
    P12_23 = interpolation_lineaire(P12,P23,0.5)
    Q = interpolation_lineaire(P01_12,P12_23,0.5)
    return ([points_control[0],P01,P01_12,Q],[Q,P12_23,P23,points_control[3]])

def courbe_bezier_3_recursif(points_control,epsilon,pile_points_courbe):
    if test_alignement_4pts(points_control,epsilon):
        pile_points_courbe.append(points_control[0])
    else:
        (points_1,points_2) = division_courbe_bezier_3(points_control)
        courbe_bezier_3_recursif(points_1,epsilon,pile_points_courbe)
        courbe_bezier_3_recursif(points_2,epsilon,pile_points_courbe)

def courbe_bezier_3_recursif_init(points_control,epsilon):
    pile_points_courbe = []
    courbe_bezier_3_recursif(points_control,epsilon,pile_points_courbe)
    pile_points_courbe.append(points_control[-1])
    return pile_points_courbe
```

Voici comme exemple une courbe qui a été tracée plus haut avec 50 points obtenus par échantillonnage régulier de $t$.

```
P0 = [0,0]
P1 = [0.3,1]
P2 = [1,1]
P3 = [1,0]
figure()
epsilon=5e-3
points = courbe_bezier_3_recursif_init([P0,P1,P2,P3],epsilon)
plot_points(points,style='.')
plot_points(points,style='r-')
plot_points([P0,P1,P2,P3],style='o')
axis([0,2,0,2])
grid()
```

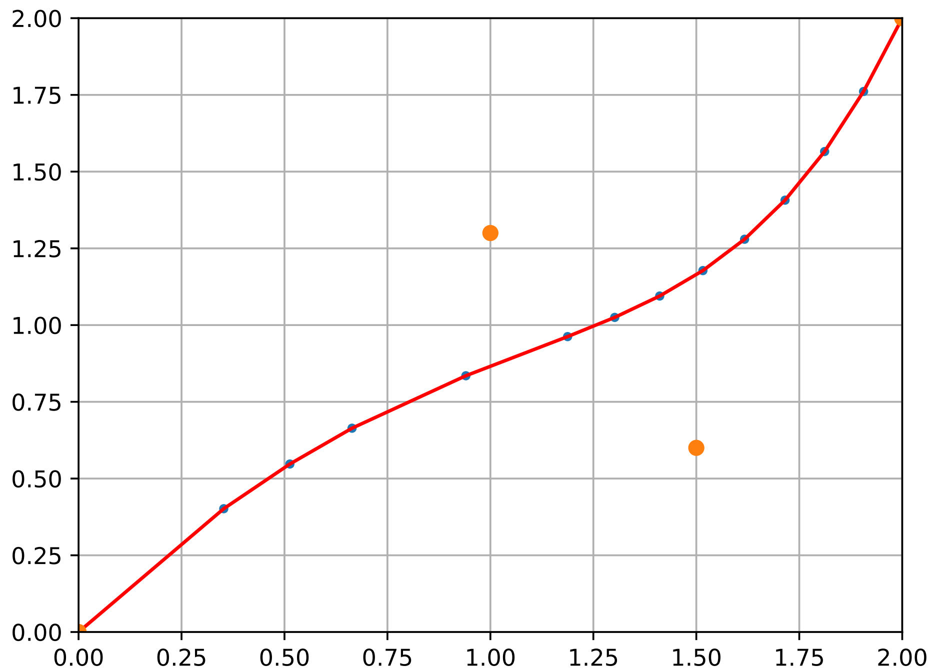



Voyons le nombre de points calculés :

```
print(len(points))
--> 31
```

Voici un autre exemple avec une courbure plus faible :

```
P0 = [0,0]
P1 = [1,1.3]
P2 = [1.5,0.6]
P3 = [2,2]
figure()
epsilon=5e-3
points = courbe_bezier_3_recursif_init([P0,P1,P2,P3],epsilon)
plot_points(points,style='.')
plot_points(points,style='r-')
plot_points([P0,P1,P2,P3],style='o')
axis([0,2,0,2])
grid()
```




```
print(len(points))
--> 14
```

### 4.d. Courbe de classe C1

En complément, on se propose de construire une courbe de classe $C^1$ en raccordant des courbes de bézier cubiques.

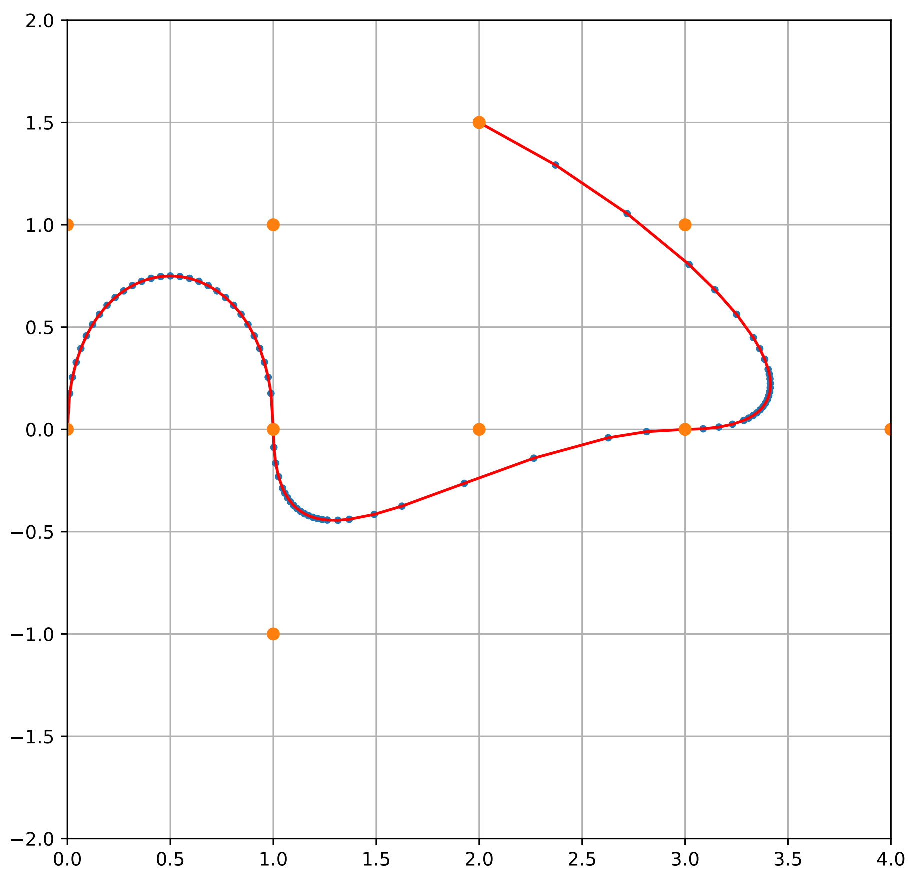
On définit pour cela une classe `Courbe`. Pour initialiser la courbe, on définit une courbe de Bézier en donnant les 4 points de contrôle. Pour ajouter un segment de Bézier cubique, il suffit de donner les points $P_2$ et $P_3$. Le point $P_0$ est identique au dernier point du segment précédent. Le point $P_1$ est calculé pour assurer la continuité de la dérivée.

```
class Courbe:
    def __init__(self,P0,P1,P2,P3,epsilon):
        self.points_courbe = []
        self.epsilon = epsilon
        self.bezier_3_recursif_init([P0,P1,P2,P3])
        self.points_control = [P0,P1,P2,P3]
    def bezier_3_recursif_init(self,points_control):
        courbe_bezier_3_recursif(points_control,self.epsilon,self.points_courbe)
        self.P2 = points_control[2]
        self.P3 = points_control[3]
    def ajouter(self,P2,P3):
        P0 = self.P3
        P1 = [P0[0]+P0[0]-self.P2[0],P0[1]+P0[1]-self.P2[1]]
        self.bezier_3_recursif_init([P0,P1,P2,P3])
        self.points_control.append(P1)
        self.points_control.append(P2)
        self.points_control.append(P3)
        self.dernier_point = P3
```

```
    def liste_points(self):
        self.points_courbe.append(self.dernier_point)
        return self.points_courbe
    def liste_points_control(self):
        self.points_control.append(self.dernier_point)
        return self.points_control
```

Exemple :

```
courbe = Courbe([0,0],[0,1],[1,1],[1,0],5e-3)
courbe.ajouter([2,0],[3,0])
courbe.ajouter([3,1],[2,1.5])
points = courbe.liste_points()
points_control = courbe.liste_points_control()
figure(figsize=(8,8))
plot_points(points,style='.')
plot_points(points,style='r-')
plot_points(points_control,'o')
axis([0,4,-2,2])
grid()
```



**Références :**

[1] R. Malgouyres, *Algorithmes pour la synthèse d'images et l'animation 3D*, (Dunod, 2055)
[2] P. Shirley, S. Marschner, *Fundamentals of computer graphics*, (A K Peters, 2009)