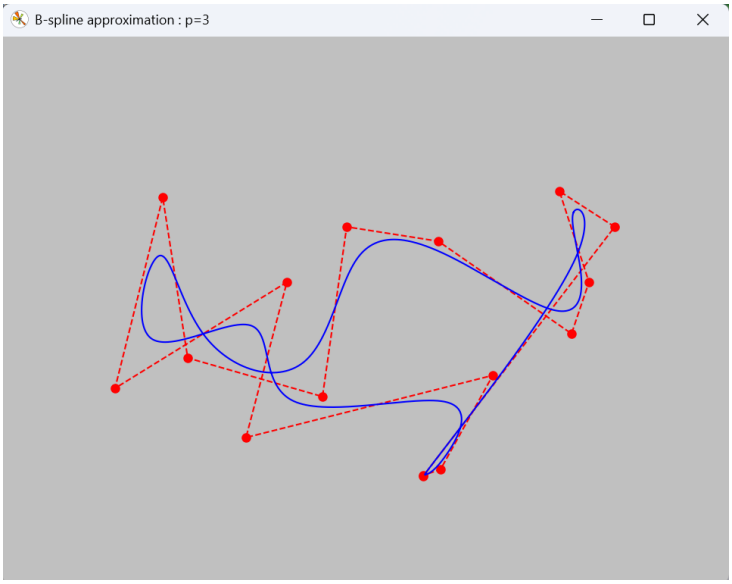
Notebook : Homework 4

B-splines

Janssen Tom

Mars 2026



Ce document a pour but d'aider les étudiants du cours de l'excellent cours de Méthodes Numériques, donné par M. Vincent Legat à l'Ecole Polytechnique de Louvain, en leur offrant une solution détaillée aux devoirs

1 Formule B-spline

Pour rappel, comme vu lors de la quatrième séance d'exo avec vos super tuteurs, une courbe paramétrée à l'aide des B-spline de degré p est définie par :

$$S(t) = \sum_{i=0}^{n} P_i B_i^p(t) \tag{1}$$

où les P_i sont les points de contrôle et B_i^p sont les fonctions de base. Ces dernières sont définies récursivement (comme à la page 30 des notes du cours) sur un vecteur de m+1 noeuds $T = \{T_0, T_1, \ldots, T_m\}$.

1: Formules des B-splines

Pour le degré $p = 0$:

$$B_i^0(t) = \begin{cases} 1 & \text{si } T_i \leq t < T_{i+1} \\ 0 & \text{sinon} \end{cases}$$

Pour le degré $p > 0$:

$$B_i^p(t) = \frac{t - T_i}{T_{i+p} - T_i} B_i^{p-1}(t) + \frac{T_{i+p+1} - t}{T_{i+p+1} - T_{i+1}} B_{i+1}^{p-1}(t)$$

2 L'implémentation "facile"

Ici on va implémenter le code pour une B-spline de degré 3 périodique (ou fermée).

2.1 Arguments de la fonction `bspline`

— **X, Y** : Coordonnées des points de contrôle.

— **t** : Paramètre sur lequel on va évaluer la courbe

— **T** : Le vecteur de nœuds, ici généré par range(-3, len(X)+4).

2.2 Gestion de la périodicité et continuité

Pour transformer une spline ouverte en une courbe fermée, il ne suffit pas de relier le dernier point au premier ; il faut assurer la continuité des dérivées premières et secondes.

Pour un degré $p = 3$, la théorie impose de dupliquer les p premiers points de contrôle à la fin de la séquence. En Python, cela se traduit par l'extension : `[*X, *X[0:3]]`.

Cette manipulation permet aux fonctions de base de "boucler" sur le vecteur de noeuds. On obtient ainsi une continuité C^2

```
from numpy import *
import numpy as np

def b(t, T, i, p):
    if p == 0:
        return (T[i] <= t) * (t < T[i+1])
    else:
        u = 0.0 if T[i+p] == T[i] else (t - T[i]) / (T[i+p] - T[i]) * b(t, T, i, p-1)
        u += 0.0 if T[i+p+1] == T[i+1] else (T[i+p+1] - t) / (T[i+p+1] - T[i+1]) * b(t, T, i+1, p-1)
        return u
```

```
11
12 def bspline(X, Y, t):
13     T = range(-3, len(X) + 4)
14
15     # extension periodique : duplication des p=3 premiers points pour
       assurer la continuite C2
16     X_ext = array([*X, *X[0:3]])
17     Y_ext = array([*Y, *Y[0:3]])
18
19     p = 3
20     n_basis = len(T) - 1 - p  # nombre de fonctions de base
21     B = zeros((n_basis, len(t)))  # Matrice d'influence (fonctions de
       base evaluees sur t)
22
23     # calcul des fonctions de base pour chaque point de controle et
       chaque valeur de t
24     for i in range(n_basis):
25         B[i, :] = b(t, T, i, p)
26
27     # Combinaison lineaire : projection des points de controle sur la
       base polynomiale
28     x_curve = X_ext @ B
29     y_curve = Y_ext @ B
30     return x_curve, y_curve
```

3 L'astuce du 11/10

Pour les plus futés, on peut remarquer que l'utilisation de noeuds équidistants permet de simplifier drastiquement le problème. Les fonctions de forme sont alors trouvables.

3.1 Exploitation du cas uniforme

Lorsque les noeuds sont équidistants et que le degré est fixé à $p = 3$, les fonctions de base ont la même forme sur chaque intervalle $[i, i+1[$. On peut alors utiliser une expression locale explicite au lieu de recalculer la récurrence à chaque évaluation.

Si $t \in [i, i+1[$, on pose

$$i = \lfloor t \rfloor, \qquad u = t - i \in [0, 1[.$$

Dans le cas uniforme cubique, les quatre fonctions de base non nulles sur cet intervalle sont des polynômes cubiques en u. On obtient alors les poids locaux

$$\frac{1}{6}\big[-u^3 + 3u^2 - 3u + 1,\ 3u^3 - 6u^2 + 4,\ -3u^3 + 3u^2 + 3u + 1,\ u^3\big],$$

ce qui se réécrit sous forme matricielle

$$S_i(u) = \frac{1}{6}\begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} P_i \\ P_{i+1} \\ P_{i+2} \\ P_{i+3} \end{bmatrix}.$$

Autrement dit, pour chaque valeur de t, on calcule les quatre poids associés à u, puis on forme une combinaison linéaire des quatre points de contrôle locaux.

3.2 Vectorisation

Pour un vecteur de paramètres t, on calcule d'abord les indices

$$i = \lfloor t \rfloor$$

et les coordonnées locales

$$u = t - i.$$

On construit ensuite la matrice

$$U = \begin{bmatrix} u_1^3 & u_1^2 & u_1 & 1 \\ u_2^3 & u_2^2 & u_2 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ u_N^3 & u_N^2 & u_N & 1 \end{bmatrix},$$

dont chaque ligne contient les puissances associées à une valeur de u. Le produit UM fournit alors directement, pour chaque point, les quatre poids de la combinaison locale.

```python
def bspline(X, Y, t):

    X = array(X)
    Y = array(Y)
    t = array(t)

    X_ext = concatenate([X, X[:3]])
    Y_ext = concatenate([Y, Y[:3]])

    M = array([
        [-1, 3, -3, 1],
        [3, -6, 3, 0],
        [-3, 0, 3, 0],
        [1, 4, 1, 0]
    ]) / 6.0

    indices = floor(t).astype(int) #pour obtenir les indices entiers des intervalles
    u = t - indices

    U = stack([u ** 3, u ** 2, u, ones_like(u)], axis=1) #on construit une matrice (N,4) avec une ligne par point

    basis_values = U @ M

    x = zeros(len(t), dtype=float)
    y = zeros(len(t), dtype=float)
    for j in range(4):
        idx = (indices + j) % len(X_ext)
        x += basis_values[:, j] * X_ext[idx]
        y += basis_values[:, j] * Y_ext[idx]

    return x, y
```

4 Conclusion

Cette approche évite le recalcul récursif des fonctions de base et permet une évaluation beaucoup plus efficace en pratique :)