B-spline Fitting Analysis

October 15, 2021

Information: *Analysis and codes for B-spline fitting*

Written by: *Zihao Xu*

Lat update date: *Oct.15.2021*

1 B-spline Summary

The following summary referred to the book *An Introduction To Nurbs With Historical Perspective* by *David F. Rogers* heavily.

1.1 Definition

Letting $P(t)$ be the position vector along the curve as a function of the parameter t, a B-spline is given by

$$P(t) = \sum_{i=1}^{n+1} B_i N_{i,k}(t), \quad t_{min} \leq t < t_{max}, \quad 2 \leq k \leq n+1$$

- B_i are the position vectors of the $\boldsymbol{n+1}$ **control polygon vertices**
- $N_{i,k}$ are the **normalized B-spline basis functions**

1.1.1 Normalized B-spline basis functions

For the ith normalized B-spline basis function of order k (degree $k-1$), the basis functions $N_{i,k}(t)$ are defined by the **Cox-de Boor recursion formulas**.

$$N_{i,1}(t) = \begin{cases} 1 & \text{if } x_i \leq t < x_{i+1} \\ 0 & \text{otherwise} \end{cases}$$
$$N_{i,k}(t) = \frac{(t-x_i)N_{i,k-1}(t)}{x_{i+k-1}-x_i} + \frac{(x_{i+k}-t)N_{i+1,k-1}(t)}{x_{i+k}-x_{i+1}}$$

Remark that the convention

$$\frac{0}{0} = 0$$

is adopted in some cases.

1.1.2 Properties

- The sum of the B-spline basis functions for any parameter value t is 1.

$$\sum_{i=1}^{n+1} N_{i,k} \equiv 1$$

- Each basis function is positive or zero for all parameter values.

$$N_{i,k} \geq 0$$

- The maximum order (the number of coefficients defining the polynomial) of the curve is $n+1$. The maximum degree (the highest power defining the polynomial) is one less.
- The curve generally follows the shape of the control polygon.
- The curve is transformed by transforming the control polygon vertices.
- The curve lies within the convex hull of its control polygon.

1.2 Knot vectors

The choice of knot vector has a significant influence on the B-spline basis functions $N_{i,k}(t)$ and hence on the resulting B-spline curve. The only requirement for a knot vector is that it satisfy the relation $x_i \leq x_{i+1}$. Fundamentally, two types of knot vector are used, **periodic** and **open**, in two flavors, **uniform** and **nonuniform**.

1.2.1 Uniform knot vector

- In a uniform knot vector, individual knot values are evenly spaced. One example is,

$$\begin{bmatrix} -0.2 & -0.1 & 0 & 0.1 & 0.2 \end{bmatrix}$$

- In practice, uniform knots vectors generally **begin at zero** and are **incremented by 1** to some maximum value, or are **normalized in the range between 0 and 1**. Examples are

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0.25 & 0.5 & 0.75 & 1 \end{bmatrix}$$

- **Periodic uniform** knot vectors yield periodic uniform basis functions for which

$$N_{i,k}(t) = N_{i-1,k}(t-1) = N_{i+1,k}(t+1)$$

  Thus, each basis function is a translate of the other.

- An **open uniform** knot vector has multiplicity of knot values at the ends equal to the order k of the B-spline basis function. Internal knot values are evenly spaced. Formally, an open uniform knot vector is given by

$$\begin{aligned} x_i &= 0 & 1 \leq i \leq k \\ x_i &= i-k & k+1 \leq i \leq n+1 \\ x_i &= n-k+2 & n+2 \leq i \leq n+k+1 \end{aligned}$$

The resulting open uniform basis functions yield curves that behave most nearly like Bézier curves. When the number of control polygon vertices is equal to the order of the B-spline basis and an open uniform knot vector is used, the B-spline basis reduces to the Bernstein basis. In that case, the knot vector is just k zeros followed by k ones.

1.2.2 Nonuniform knot vector

- Nonuniform knot vectors may have either unequally spaced and/or multiple internal knot values. They may be periodic or open. Examples are

$$\begin{aligned} \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 2 & 2 & 2 \end{bmatrix} &\quad \text{open} \\ \begin{bmatrix} 0 & 1 & 2 & 2 & 3 & 4 \end{bmatrix} &\quad \text{periodic} \\ \begin{bmatrix} 0 & 0.28 & 0.5 & 0.72 & 1 \end{bmatrix} &\quad \text{periodic} \end{aligned}$$

1.3 B-spline Curve Controls

Because of the flexibility of B-spline basis functions and hence of the resulting B-spline curves, different types of control 'handles' are used to influence the shape of B-spline curves. Control is achieved by

- changing the type of knot vector and hence basis function: periodic uniform, open uniform or nonuniform
- changing the order k of the basis function
- changing the number and position of the control polygon vertices
- using multiple polygon vertices
- using multiple knot values in the knot vector

2 B-spline Curve Fitting

2.1 Problem definition

In this project, determining a control polygon that generates a B-spline curve for a set of **know data points** is considered.

If a data point lies on the B-spline curve, then it must satisfy

$$\begin{aligned}
D_1(t_1) &= N_{1,k}(t_1)B_1 + N_{2,k}(t_1)B_2 + \cdots + N_{n+1,k}(t_1)B_{n+1} \\
D_2(t_2) &= N_{1,k}(t_2)B_1 + N_{2,k}(t_2)B_2 + \cdots + N_{n+1,k}(t_2)B_{n+1} \\
&\vdots \\
D_j(t_j) &= N_{1,k}(t_j)B_1 + N_{2,k}(t_j)B_2 + \cdots + N_{n+1,k}(t_j)B_{n+1}
\end{aligned}$$

where $2 \le k \le n+1 \le j$. In matrix form,

$$\mathbf{D} = \mathbf{N}\mathbf{B}$$

where

$$\begin{aligned}
\mathbf{D}^T &= \begin{bmatrix} D_1(t_1) & D_2(t_2) & \cdots & D_j(t_j) \end{bmatrix} \\
\mathbf{B}^T &= \begin{bmatrix} B_1 & B_2 & \cdots & B_{n+1} \end{bmatrix} \\
\mathbf{N} &= \begin{bmatrix} N_{1,k}(t_1) & \cdots & N_{n+1,k}(t_1) \\ \vdots & \ddots & \vdots \\ N_{1,k}(t_j) & \cdots & N_{n+1,k}(t_j) \end{bmatrix}
\end{aligned}$$

The **parameter value** t_j for each data point is a measure of the distance of the data point along the B-spline curve. One useful approximation for this parameter value uses the chord length between data points. Specifically, for j data points the parameter value at the lth data point is

$$\begin{aligned}
t_1 &= 0 \\
\frac{t_l}{t_{max}} &= \frac{\sum_{s=2}^{l} |D_s - D_{s-1}|}{\sum_{s=2}^{j} |D_s - D_{s-1}|}
\end{aligned}$$

The maximum parameter value t_{max} is usually taken as the maximum value of the knot vector.

2.2 Solution

If $2 \le k \le n+1 = j$, then the matrix $\mathbf{N}$ is square and the control polygon is obtained directly by matrix inversion

$$\mathbf{B} = \mathbf{N}^{-1}\mathbf{D} \quad 2 \le k \le n+1 = j$$

In this case, the resulting B-spline curve passes through each data point. It may not be 'smooth' or 'fair' and develop unwanted wiggles or undulations.

For a fairer or smoother curve, specify fewer control polygon points than data points, i.e. $2 \leq k \leq n+1 < j$. The problem is now overspecified and can only be solved in a mean sense.

$$
\begin{aligned}
\mathbf{D} &= \mathbf{N}\mathbf{B} \\
\mathbf{N}^T\mathbf{D} &= \mathbf{N}^T\mathbf{N}\mathbf{B} \\
\mathbf{B} &= (\mathbf{N}^T\mathbf{N})^{-1}\mathbf{N}^T\mathbf{D}
\end{aligned}
$$

3 B-spline fitting in python

Here is the python implementation of the algorithm mentioned above. First import all the required packages.

[1]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
```

3.1 Uniform knot vectors

First define the function to create **uniform knot vectors** according to spline order, number of control points, normalize or not and open or periodic.

[2]:
```
def KnotVector(k, num, normalize=True, periodic=False):
    """ Create an uniform knot vector.
        k:            curve degree
        num:          number of control points
        normalize:    whether to normalize the knot vector, default is True
        perioic:      whether the knot vector is periodic or not, default is␣
 ↪False
    """
    # Get the length of the required knot vector
    n = num - 1
    knot_num = n + k + 1
    # Check whether open or periodic
    if periodic:
        # Periodic uniform knot vectors
        Knots = np.arange(0., knot_num, 1.)
    else:
        # Open uniform knot vectors
        Knots = np.zeros(knot_num)
        for ii in range(knot_num):
            if 0 <= ii and ii <= k - 1:
                Knots[ii] = 0.
            elif k <= ii and ii <= n:
                Knots[ii] = ii + 1 - k
            else:
                Knots[ii] = n - k + 2
    # Check whether to normalize or not
    if normalize:
        Knots = Knots / np.max(Knots)
    return Knots
```

3.2 B-spline basis functions

Next build the function to get the **normalized B-spline basis functions** from knot vectors.

[3]:
```python
def CoxDeBoorRecursion(i, k, Knots, t):
    """ Compute the corrsponding B-spline basis functions.
        i:          position index
        k:          order of basis function
        Knots:      the knot vector
        t:          parameter value
    """
    if k == 1:
        if Knots[i] <= t and t < Knots[i+1]:
            return 1.
        else:
            return 0.
    else:
        num1 = (t - Knots[i]) * CoxDeBoorRecursion(i, k-1, Knots, t)
        den1 = Knots[i+k-1] - Knots[i]
        num2 = (Knots[i+k] - t) * CoxDeBoorRecursion(i+1, k-1, Knots, t)
        den2 = Knots[i+k] - Knots[i+1]
        if den1 == 0:
            part1 = 0.
        else:
            part1 = num1 / den1
        if den2 == 0:
            part2 = 0.
        else:
            part2 = num2 / den2
        return part1 + part2
```

Also construct a function to plot out the basis functions for a sanity check.

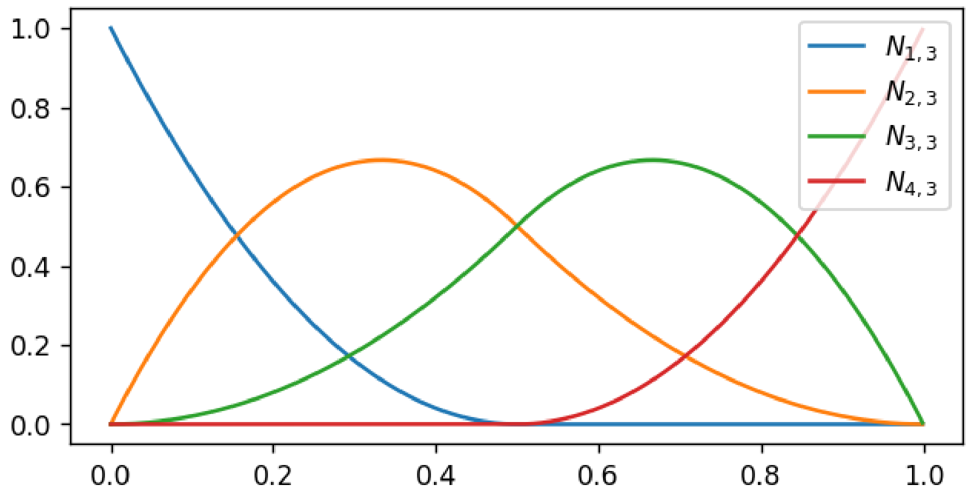
[4]:
```python
def BSplineBasis(i, k, Knots, T):
    """ Return a series of basis function value
        i:          position index
        k:          order of basis function
        Knots:      the knot vector
        t:          a series of parameter value
    """
    values = []
    for t in T:
        values.append(CoxDeBoorRecursion(i, k, Knots, t))
    return np.asarray(values)
```

Do a sanity check for the above functions.

[5]:
```python
# Curve order
k = 3
# Number of control points
num = 4
# Create an normalized open uniform knot vector
```

```
Knots = KnotVector(k, num)
print("The normalized open uniform knot vector is:\n", Knots)
# Plot out the basis function on the interval [0, 1]
T = np.arange(0.,1.,0.001)
fig, ax = plt.subplots(figsize=(6,3), dpi=125)
for ii in range(num):
    ax.plot(T, BSplineBasis(ii,k,Knots, T), label="$N_{%d,%d}$"%(ii+1,k))
plt.legend(loc='upper right')
plt.show()
```

```
The normalized open uniform knot vector is:
 [0.  0.  0.  0.5 1.  1.  1. ]
```



3.3 Forward B-spline computation

Construct a function to compute the B-spline curve as a benchmark.

```
[6]: def PointsToSpline(Points, k, Knots, T):
         """ Get the B-spline curve from given control points, knot vectors
             and parameter values
             Points:   Control points
             k:        Curve degree
             Knots:    Knot vector
             T:        Parameter value interval
         """
         # Number of control points
         num = Points.shape[0]
         n = num - 1
         # Dimension of points on the curve
         dim = Points.shape[1]
```

```
    # Compute the curve point by point
    curve = []
    for t in T:
        point = np.zeros(dim)
        for ii in range(0, n+1):
            point += Points[ii]*CoxDeBoorRecursion(ii, k, Knots, t)
        curve.append(point)
    return np.array(curve)
```

3.4 B-spline curve fitting

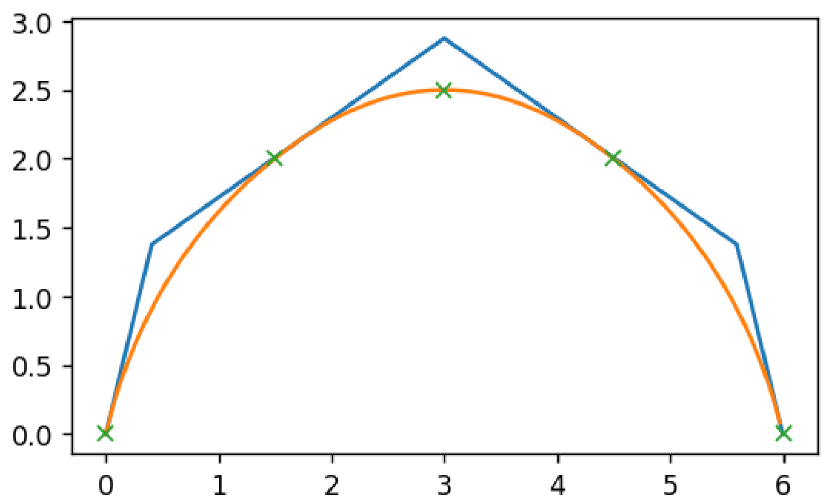
Finally, use the algorithm mentioned before to fit B-spline curve to some known points. Notice the Moore-Penrose pseudo inverse is computed by numpy.linalg.pinv method.

[7]:
```
def CurveToPoints(Curve, k, num):
    """ Fit a B-spline curve to a given curve
        Curve:    The given curve to be fit
        k:        Curve degree
        num:      Number of control points
    """
    # Create an open uniform knot vector
    Knots = KnotVector(k, num)
    # Assign the parameter values
    Distances = [0.]
    num_points = Curve.shape[0]
    distance = 0
    for ii in range(num_points - 1):
        distance += np.sqrt(np.sum(np.power(Curve[ii+1,:]-Curve[ii,:],2)))
        Distances.append(distance)
    Distances = np.asarray(Distances)
    Distances /= np.max(Distances)
    Distances[-1] *= 1.0 - 1e-5
    # Build the N matrix
    N_Mat = np.zeros([num_points, num])
    for ii in range(num_points):
        for jj in range(num):
            N_Mat[ii,jj] = CoxDeBoorRecursion(jj, k, Knots, Distances[ii])
    # Solve the equation
    if num_points == num:
        N_Inv = np.linalg.inv(N_Mat)
    else:
        N_Inv = np.linalg.pinv(N_Mat)
    Points = np.dot(N_Inv, Curve)
    # Output the knot vector and control points
    return Knots, Points
```

Now let's check the fitting with an example.

[8]:

```python
# Construct a curve to be fit
Curve = np.array([[0, 0], [1.5, 2], [3, 2.5], [4.5, 2], [6, 0]])
# Curve degree
k = 3
# Number of control points
num = 5
# Fit with B-spline curve
Knots, Points = CurveToPoints(Curve, k, num)
# Draw out the spline with computed control points
T = np.arange(0., 1., 0.01)
Spline = PointsToSpline(Points, k, Knots, T)
fig, ax = plt.subplots(figsize=(5,3), dpi=125)
ax.plot(Points[:, 0], Points[:, 1])
ax.plot(Spline[:, 0], Spline[:, 1])
ax.plot(Curve[:, 0], Curve[:, 1], "x")
plt.show()
```



4 Future Work

Here are the possible future works I can think of after the discussion in the morning. List there here for reference.

4.1 Algorithm

- **Dimension Reduction**: For efficient computation of the control points, the matrix to be inverted can not be to large, which requires a re-sampling to the smoothed curve (which might include thousands of points) while maintaining the features of the input curve.
- **Find a way to determine the number of control points**: To fit a B-spline to a known curve, the curve degree and the number of control points are needed. We can always use cubic B-spline curves so that we do not need to change the curve degree while a cubic B-spline curve is usually flexible enough. However, we must find a way to determine the number of control points according to some features of the input curve.
- **Support closed B-spline curve**: In this project, some closed curves might be the inputs so the algorithm should be able to support closed B-spline curve fitting.

4.2 Test

- More B-spline curves should be created to test this fitting algorithm.
- Find out the possible limitations of current algorithm (e.g. when is the matrix inverse not available).

4.3 Future Implementation

- Some packages here are embedded in Python. Substitutions are required if the final project is going to be implemented in Java.