Report for Week 10

November 5, 2021

Information: *A brief summary for the work I've done related to the current project this semester.*

Written by: *Zihao Xu*

E-mail: *xu1376@purdue.edu*

1 Recap on Previous Works

1.1 Problem Statement

- While the goal is to **get precise descriptions with parameters strictly defining the geometric shapes from hand sketches**, the current sketch interface cannot fit free-form curves well.
- The existing sketch interface provides basic fitting using Bezier curves.
  - It cannot ensures the second order continuity of the fitted curves in some cases where a long free-form curve is separated to several parts and get fitted part by part.
  - In the meantime, it lacks the ability to appropriately determine the order of curves and number of control points, which are essential for precision and efficiency since the input curves would have different complexities.
- A stable, efficient and comprehensive free-form curve fitting technique is required for any further development.

1.2 Analyzing Related Works

- As I'm new to geometrics in computer aided design and have no prior knowledge about the free-form curves, I first read through the well-known book for new learners - *An Introduction to NURBS With Historical Perspective* written by *David F. Rogers* - to get a basic idea of how these curves are represented in computers.
- After that, I searched for existing free-form curve fitting techniques since I thought this is well-solved problem. To make myself familiar with the research works related to this topic, the papers I've read through are shown below. These are not formal references as this is a simple summary and I'm not writing this report in LaTeX.
  - **BendSketch: Modeling Freeform Surfaces Through 2D Sketching** written by *C.Li*, *H.Pan*, *Y.Liu*, *X.Tong*, *A.Sheffer* and *W.Wang*. This paper proposes new methods of modeling complex freeform shapes by sketching sparse 2D strokes and enabled the generation of surfaces with complex curvature patterns, while it simply re-sampled the curves instead of fitting the curves.

  - **FiberMesh: Designing Freeform Surfaces with 3D Curves** written by *A.Nealen*, *T.Igarashi*, *O.Sorkine*, *M.Alexa*. This paper presented a system for designing freeform surfaces with a collection of 3D curves. However, as far as I could understand, the models were not strictly defined by a series of parameters though it provided a powerful tool to explore various design ideas.
  - **A Method to Generate Freeform Curves from a Hand-drawn Sketch** written by *T.Kuragano* and *A.Yamaguchi*. This paper developed a method to extract five degree Bezier curves based on a hand-drawn sketch while a lot of technique mentioned were specific to hand-drawn sketches on paper.
  - **An interactive sketch-based CAD interface realizing geometrical and topological editing across multiple objects based on fuzzy logic**, which is an extended version of *Over-sketching Operation to Realize Geometrical and Topological Editing across Multiple Objects in Sketch-based CAD Interface* written by *T.Ito*, *T.Kaneko*, *Y.Tanaka*, *S.Saga*. This is a fairly new paper which focused on realizing sketch-based editing operation targeting multiple geometric objects simultaneously. It mentioned the curve fitting based on fuzzy logic but did not give the details.
  - **Advanced drawing beautification with ShipShape** written by *J.Fiser*, *P.Asente*, *S.Schiller*. *D.Sykora*. This paper presented a system for beautifying freehand sketches that provides multiple suggestion, taking into account implicit geometric relations to automatically rectify the output image. I looked into this paper because it mentioned working with Bezier curves while it skipped it as embedded work in Adobe.
  - **Freehand drawing system using a fuzzy logic concept** written by *C.L. Philip Chen*, *S.Xie*. This paper developed a pen-base drawing system using a fuzzy logic which could infer human drawing intentions and generate the corresponding geometric primitives and smooth B-spline curves. It showed an ordinary way of B-spline fitting given a fixed curve order.
  - **Fuzzy Spline Interpolation and Its Application to On-Line Freehand Curve Identification** written by *S.Saga* and *H.Makino*. This is the original paper introducing the fuzzy spline interpolation to handle curves which involve vagueness in their positional information.

- About how to determine the number of control points and resample the curve for B-spline fitting, I read through the following paper and would like to check if I can make use of the curvature information.

  - **Curvature based sampling of curves and surfaces** written by *L.Pagani* and *P.J.Scott*. This paper proposed a sampling method that enables the reconstruction of a curve or surface taking into account the regularity of the sampling and the complexity of the object.

1.3 Basic B-spline Fitting Analysis

Here is the B-spline fitting analysis I made to get myself familiar with the fitting processes and build a foundation for further development. The following summary referred to the book *An Introduction To Nurbs With Historical Perspective* by *David F. Rogers* heavily.

1.3.1 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 **$n+1$ control polygon vertices**
- $N_{i,k}$ are the **normalized B-spline basis functions**

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.

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.

1.3.2 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**.

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

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

  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.

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

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

1.3.3 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

1.3.4 B-spline Curve Fitting Algorithm

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.

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.

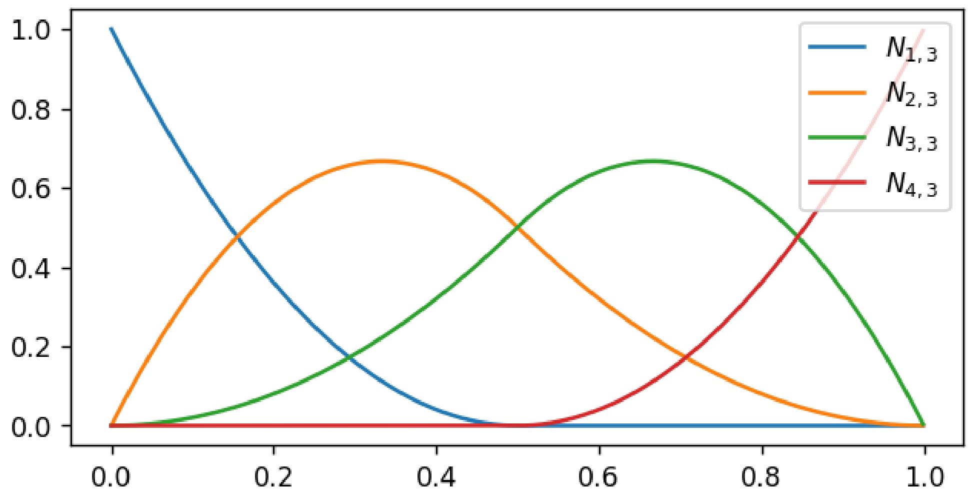
$$
\begin{aligned}
\mathbf{D} &= \mathbf{N}\mathbf{B} \\
\mathbf{N}^T\mathbf{D} &= \mathbf{N}^T\mathbf{N}\mathbf{B} \\
\mathbf{B} &= (\mathbf{N}^T\mathbf{N})^{-1}\mathbf{N}^T\mathbf{D}
\end{aligned}
$$

1.3.5 B-spline fitting in python

Here is the python implementation of the algorithm mentioned above. The explicit codings of the functions have been shown in the previous report and here is a simple show of usage. First check the basis functions used in curve fitting.

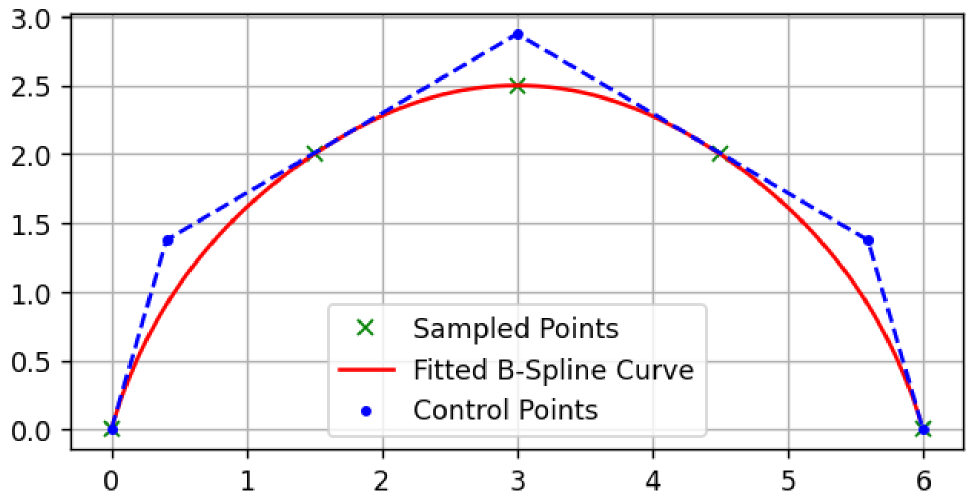
[1]:
```python
import numpy as np
import matplotlib.pyplot as plt
import B_spline_Fitting as bsf
# Curve order
k = 3
# Number of control points
num = 4
# Create an normalized open uniform knot vector
Knots = bsf.KnotVector(k, num)
print("The normalized open uniform knot vector is:\n", Knots)
# Plot out 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,
            bsf.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. ]
```



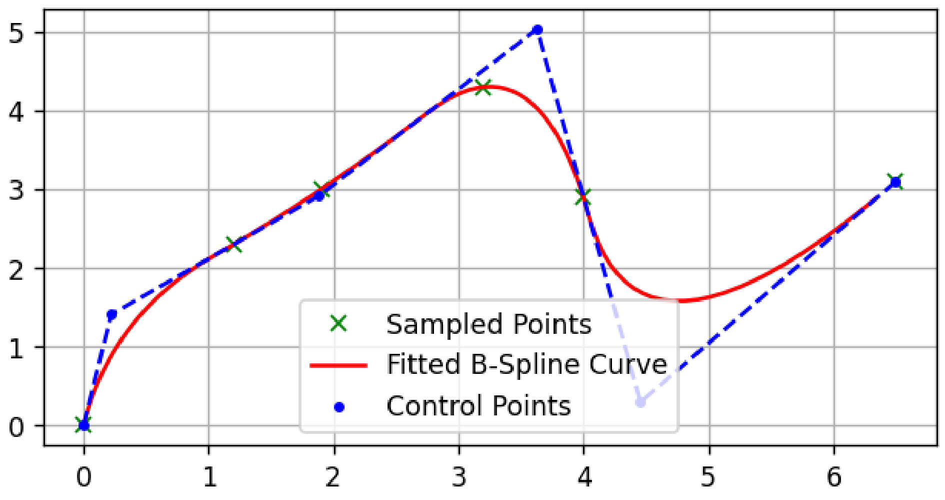
Second, check the fitting with examples.

[2]:
```
# 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 = bsf.CurveToPoints(Curve, k, num)
# Draw out the spline with computed control points
T = np.arange(0., 1., 0.01)
Spline = bsf.PointsToSpline(Points, k, Knots, T)
fig, ax = plt.subplots(figsize=(6, 3), dpi=125)
ax.plot(Curve[:, 0], Curve[:, 1], "gx", label='Sampled Points')
ax.plot(Spline[:, 0], Spline[:, 1], 'r', label='Fitted B-Spline Curve')
ax.plot(Points[:, 0], Points[:, 1], '.b', label='Control Points')
ax.plot(Points[:, 0], Points[:, 1], '--b')
plt.grid()
plt.legend(loc='best')
plt.show()
```



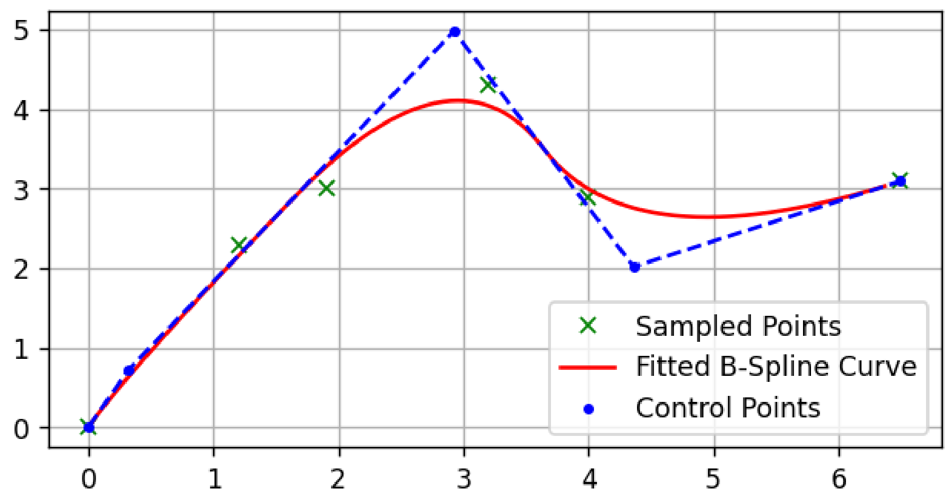
Another example using B-Spline fitting is shown below.

[3]:
```python
# Construct a curve to be fit
Curve = np.array([[0, 0], [1.2, 2.3], [1.9, 3], [3.2, 4.3], [4, 2.9],
                  [6.5, 3.1]])
# Curve degree
k = 3
# Number of control points
num = 6
# Fit with B-spline curve
Knots, Points = bsf.CurveToPoints(Curve, k, num)
# Draw out the spline with computed control points
T = np.arange(0., 1., 0.01)
Spline = bsf.PointsToSpline(Points, k, Knots, T)
fig, ax = plt.subplots(figsize=(6, 3), dpi=125)
ax.plot(Curve[:, 0], Curve[:, 1], "gx", label='Sampled Points')
ax.plot(Spline[:, 0], Spline[:, 1], 'r', label='Fitted B-Spline Curve')
ax.plot(Points[:, 0], Points[:, 1], '.b', label='Control Points')
ax.plot(Points[:, 0], Points[:, 1], '--b')
plt.grid()
plt.legend(loc='best')
plt.show()
```



It's easy to observe that selecting the right curve degree and number of control points is essential. Let's try with five control points and the curve degree to be three.

[4]:
```
# Construct a curve to be fit
Curve = np.array([[0, 0], [1.2, 2.3], [1.9, 3], [3.2, 4.3], [4, 2.9],
                  [6.5, 3.1]])
# Curve degree
k = 3
# Number of control points
num = 5
# Fit with B-spline curve
Knots, Points = bsf.CurveToPoints(Curve, k, num)
# Draw out the spline with computed control points
T = np.arange(0., 1., 0.01)
Spline = bsf.PointsToSpline(Points, k, Knots, T)
fig, ax = plt.subplots(figsize=(6, 3), dpi=125)
ax.plot(Curve[:, 0], Curve[:, 1], "gx", label='Sampled Points')
ax.plot(Spline[:, 0], Spline[:, 1], 'r', label='Fitted B-Spline Curve')
ax.plot(Points[:, 0], Points[:, 1], '.b', label='Control Points')
ax.plot(Points[:, 0], Points[:, 1], '--b')
plt.grid()
plt.legend(loc='best')
plt.show()
```



1.3.6 Deficiencies of this basic B-spline fitting

- **Requires 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.
- **No way to determine the number of control points itself**: 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.
- **Cannot 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.

2 B-Spline Fitting methods via Scipy

B-Spline fitting is embedded in the **Scipy** package and is very useful if we consider using a backend server different than server support by Java scripts. The official document for B-Spline fitting is given here.

The core function is **scipy.interpolate.splprep()** and here are the commonly used arguments.

- **x**: A list of sample vector arrays representing the curve.
- **u**: An array of parameter values. If not given, these values are calculated automatically using the chord length.
- **k**: Degree of the spline. Cubic splines are recommended.
- **s**: A smoothing condition. Large s means more smoothing while smaller values of s indicate less smoothing.
- **per**: If non-zero, data points are considered periodic with period x[m-1] - x[0] and a smooth periodic spline approximation is returned. Enable this option when fitting closed curves.

The commonly used outputs are as follows.

- **t,c,k**: A tuple (t,c,k) containing the vector of knots, the B-spline coefficients, and the degree of the spline.
- **u**: An array of parameter values.

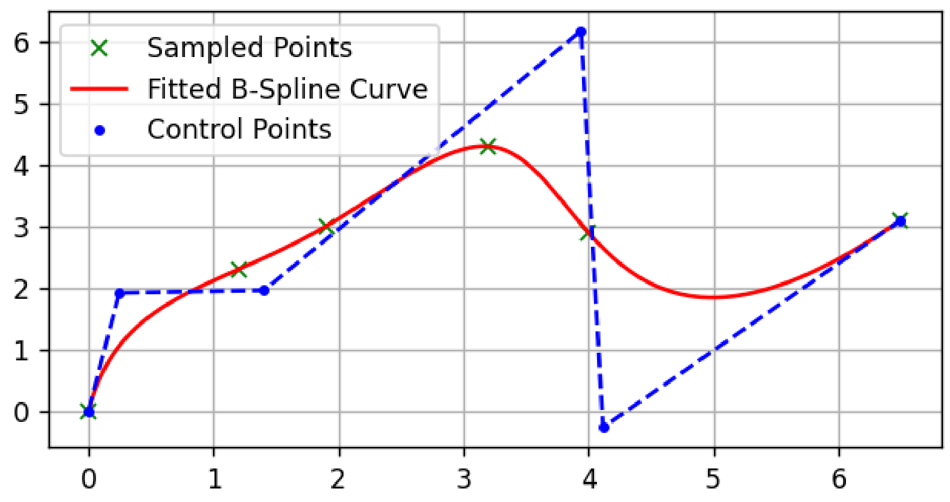
To check the performance of fitted splines, call the method **scipy.interpolate.splev()**.

It needs to be mentioned that the number of control points is exactly the same as the number of input position pairs.

Still take the data points above as an example.

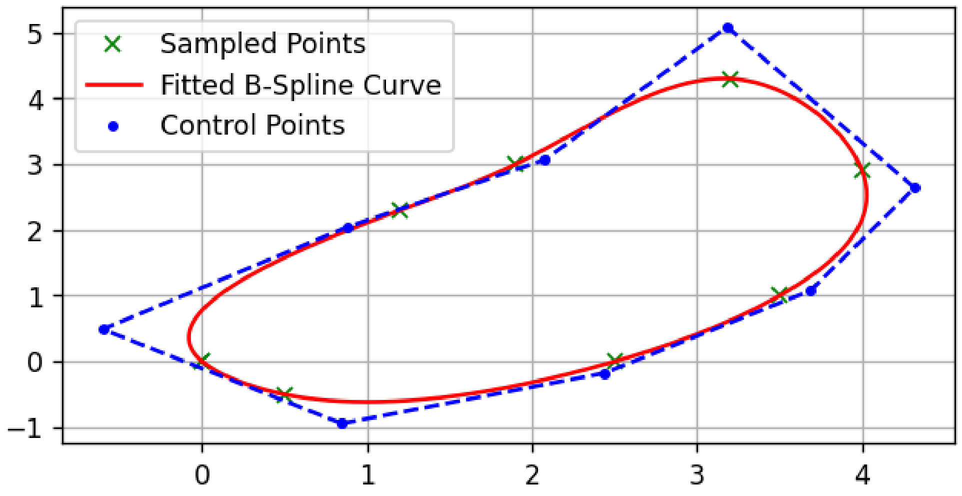
[5]:
```python
import scipy.interpolate as interpolate

# Construct a curve to be fit
Curve = np.array([[0, 0], [1.2, 2.3], [1.9, 3], [3.2, 4.3], [4, 2.9],
                  [6.5, 3.1]])
# Fit with B-spline curve
tck, u = interpolate.splprep(Curve.T, u=None, s=0, k=3)
# Extract the control pointss
Points = tck[1]
# Draw out the spline with computed control points
T = np.linspace(u.min(), u.max(), 1000)
Spline = interpolate.splev(T, tck)
fig, ax = plt.subplots(figsize=(6, 3), dpi=125)
ax.plot(Curve[:, 0], Curve[:, 1], "gx", label='Sampled Points')
ax.plot(Spline[0], Spline[1], 'r', label='Fitted B-Spline Curve')
ax.plot(Points[0], Points[1], '.b', label='Control Points')
ax.plot(Points[0], Points[1], '--b')
plt.grid()
plt.legend(loc='best')
plt.show()
```



This function also supports fitting closed curves.

[6]:
```python
# Construct a curve to be fit
Curve = np.array([[0, 0], [1.2, 2.3], [1.9, 3], [3.2, 4.3], [4, 2.9], [3.5, 1],
                  [2.5, 0], [0.5, -0.5], [0, 0]])
# Fit with B-spline curve
tck, u = interpolate.splprep(Curve.T, u=None, s=0, k=3, per=1)
# Extract the control pointss
Points = tck[1]
num = len(Points[0])
# Draw out the spline with computed control points
T = np.linspace(u.min(), u.max(), 1000)
Spline = interpolate.splev(T, tck)
fig, ax = plt.subplots(figsize=(6, 3), dpi=125)
ax.plot(Curve[:, 0], Curve[:, 1], "gx", label='Sampled Points')
ax.plot(Spline[0], Spline[1], 'r', label='Fitted B-Spline Curve')
ax.plot(Points[0][0:num - 2],
        Points[1][0:num - 2],
        '.b',
        label='Control Points')
ax.plot(Points[0][0:num - 2], Points[1][0:num - 2], '--b')
plt.grid()
plt.legend(loc='best')
plt.show()
```


Sampled Points
Fitted B-Spline Curve
Control Points
5
4
3
2
1
0
−1
0
1
2
3
4

3 Curve Sampling for Fitting

3.1 Problem Statement

Given a parametric curve described by point positions, we need to first resample the curve. As mentioned above, the number of control points obtained from the function embedded in Scipy package is exactly the same as number of sampled points (for open curves). Therefore, it is essential to appropriately sample the input curve as we would like to use as few control points as possible to describe a curve while also maintaining its characteristics.

Take the spline curve generated above as an example.

[7]:
```python
Curve_ori = np.concatenate(
    (Spline[0].reshape(-1, 1), Spline[1].reshape(-1, 1)), axis=1)
Curve_ori[-1, :] = Curve_ori[0, :]
print("There are %d input points." % Curve_ori.shape[0])
```

```
There are 1000 input points.
```

Suppose we have 1000 position pairs describing this curve. Let's see how much time it would take to directly fit this curve.

[8]:
```python
import time
# Start time
start = time.time()
# Fit with B-spline curve
tck, u = interpolate.splprep(Curve_ori.T, u=None, s=0, k=3, per=1)
# End time
end = time.time()
print("The fitting takes %f seconds." % (end - start))
# Extract the control pointss
Points = tck[1]
num = len(Points[0])
print("Obtain %d control points." % num)
```

```
The fitting takes 0.001206 seconds.
Obtain 1002 control points.
```

3.2 Resampling the curves

I would like to implement the method presented in this paper.The method takes into account both the regularity of the sampling and the complexity if the object. A higher density of samples is assigned where there are some significant features, described by the curvature.

The original sampling method is used for parametric curves with mathematical representations. The arc length parametrisation in the paper is computed as

$$l(t) = \int_{t_0}^{t} \|\mathbf{r}'(u)\| du$$

In my implementation, given sampled data points, the backward Euler method is used.

$$\mathbf{r}'[t_i] = \frac{\mathbf{r}[t_i] - \mathbf{r}[t_{i-1}]}{t_i - t_{i-1}}$$

The curvature parametrisation in the paper is computed as

$$k(t) = \frac{\|\mathbf{r}'(t) \times \mathbf{r}''(t)\|}{\|\mathbf{r}'(t)\|^3}$$

$$k_p(t) = \int_{t_0}^{t} k(u) dl(u) = \int_{t_0}^{t} k(u) \|\mathbf{r}'(u)\| du$$

Similarly, the backward Euler method is used in the implementation.

$$\begin{aligned}\mathbf{r}''[t_i] &= \frac{\mathbf{r}'[t_i] - \mathbf{r}'[t_{i-1}]}{t_i - t_{i-1}} \\ &= \frac{\mathbf{r}[t_i] - 2\mathbf{r}[t_{i-1}] + \mathbf{r}[t_{i-2}]}{(t_i - t_{i-1})^2}\end{aligned}$$

[9]:
```
from numpy.linalg import norm


def resample(curve, num=0, closed=False):
    """ Resample a curve to the desired number of points
        curve:   2D sampled data points
        num:     Number of resampled data points
                 If set to 0, automatically determine the number
        closed:  Input closed curves would have a repeating data point
                 at the end. Set to true to implicate this.
    """
    cv = curve.copy()
    # The initial parametrisation is uniform on [0,1]
    du = 1 / (cv.shape[0] - 1)
    # Calculate the approximated arc length parametrisation
    l = np.zeros(cv.shape[0])
    distance = 0
    for ii in range(1, cv.shape[0]):
        cv_dot = (cv[ii, :] - cv[ii - 1, :]) / du
        distance += norm(cv_dot, 2) * du
        l[ii] = distance
    # Calculate the approximated curvature parametrisation
    k = np.zeros(cv.shape[0])
    distance = 0
    cv_dot = (cv[1, :] - cv[0, :]) / du
    cv_ddot = (cv[1, :] - cv[0, :]) / (du**2)
    curvature = np.abs(np.cross(cv_dot, cv_ddot)) / (norm(cv_dot, 2)**3)
    distance += (l[1] - l[0]) * curvature
    k[1] = distance
    for ii in range(2, cv.shape[0]):
        cv_dot = (cv[ii, :] - cv[ii - 1, :]) / du
        cv_ddot = (cv[ii, :] - 2 * cv[ii - 1, :] + cv[ii - 2]) / (du**2)
        curvature = np.abs(np.cross(cv_dot, cv_ddot)) / (norm(cv_dot, 2)**3)
        distance += (l[ii] - l[ii - 1]) * curvature
        k[ii] = distance
    # The overall parametrisation
    p = l + 2 * k
    # Check whether the number of resampled points is assigned
    if num == 0:
        num = int(p[-1] // 3 + 1)
    # Add an additional point for closed point
    if closed:
        num += 1
    # Select the resampled points
    pts = np.zeros((num, 2))
    pts[0, :] = cv[0, :]
    pts[num - 1, :] = cv[-1, :]
    for ii in range(1, num - 1):
        for jj in range(0, cv.shape[0]):
            if p[jj] >= p[-1] / (num - 1.0) * ii:
                pts[ii, :] = cv[jj, :]
                break
    # Return the resampled points
    return (pts)
```

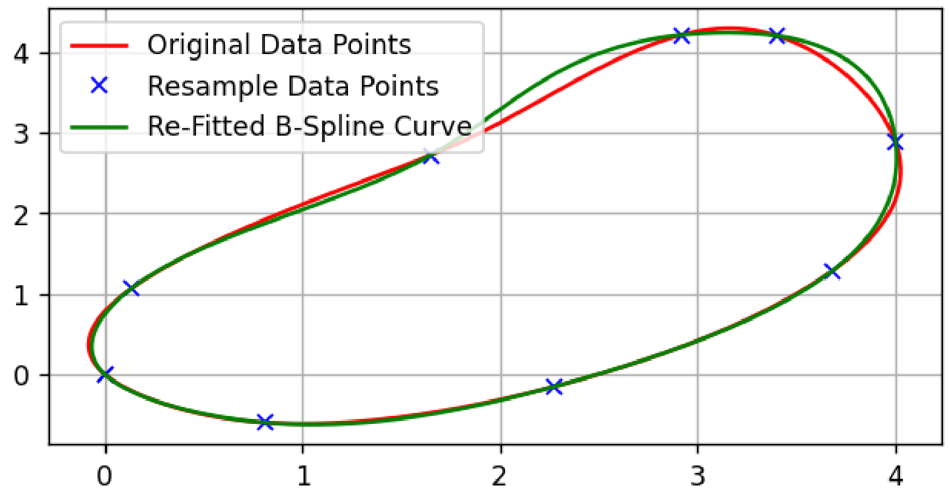
Let's see how this sampling method performs and if the re-fitted B-Spline curve is the same as the previous one.

[10]:
```
re_points = resample(Curve_ori, closed=True)
print("The resampled data points are:\n", re_points)
```

```
The resampled data points are:
 [[-7.97972799e-17  2.77555756e-16]
 [ 1.31896720e-01  1.05934332e+00]
 [ 1.64955895e+00  2.72257923e+00]
 [ 2.91602792e+00  4.21670078e+00]
 [ 3.40356234e+00  4.20657534e+00]
 [ 4.00197212e+00  2.88661486e+00]
 [ 3.67988374e+00  1.28197662e+00]
 [ 2.27581643e+00 -1.53804134e-01]
 [ 8.12200440e-01 -6.00934067e-01]
 [-7.97972799e-17  2.77555756e-16]]
```

[11]:
```
# Re-fit the B-Spline Curve from resampled points
tck, u = interpolate.splprep(re_points.T, u=None, s=0, k=3, per=1)
T = np.linspace(u.min(), u.max(), 1000)
Spline = interpolate.splev(T, tck)
# Draw out the spline with computed control points
```

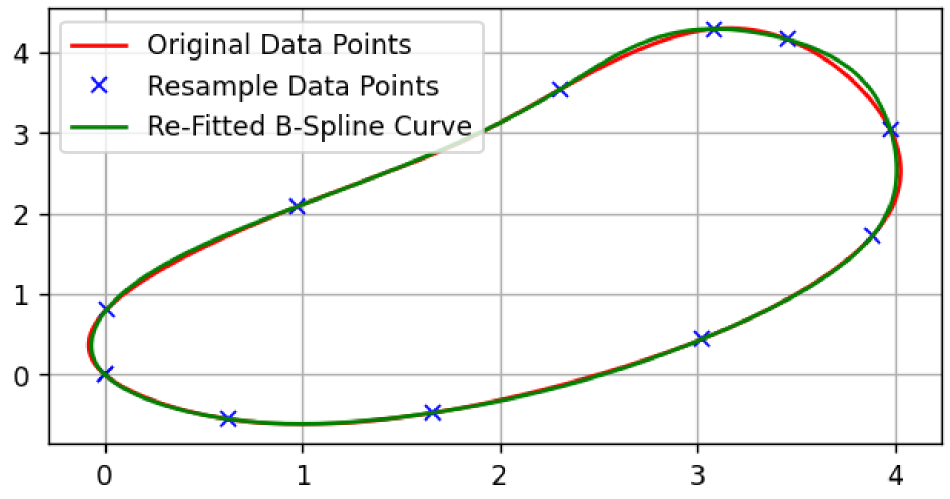
```
fig, ax = plt.subplots(figsize=(6, 3), dpi=125)
ax.plot(Curve_ori[:, 0], Curve_ori[:, 1], "r", label='Original Data Points')
ax.plot(re_points[:, 0], re_points[:, 1], 'bx', label='Resample Data Points')
ax.plot(Spline[0], Spline[1], 'g', label='Re-Fitted B-Spline Curve')
plt.grid()
plt.legend(loc='best')
plt.show()
```



After several trials, it had been found that more control points are needed for precision after resampling. Possible future works may look into how to improve the re-fitting precision with the same number of control points.

[12]:
```
re_points = resample(Curve_ori, num=11, closed=True)
print("The resampled data points are:\n", re_points)
# Re-fit the B-Spline Curve from resampled points
tck, u = interpolate.splprep(re_points.T, u=None, s=0, k=3, per=1)
T = np.linspace(u.min(), u.max(), 1000)
Spline = interpolate.splev(T, tck)
# Draw out the spline with computed control points
fig, ax = plt.subplots(figsize=(6, 3), dpi=125)
ax.plot(Curve_ori[:, 0], Curve_ori[:, 1], "r", label='Original Data Points')
ax.plot(re_points[:, 0], re_points[:, 1], 'bx', label='Resample Data Points')
ax.plot(Spline[0], Spline[1], 'g', label='Re-Fitted B-Spline Curve')
plt.grid()
plt.legend(loc='best')
plt.show()
```

```
The resampled data points are:
 [[-7.97972799e-17  2.77555756e-16]
 [ 1.14462321e-02  8.09783395e-01]
 [ 9.75168946e-01  2.08654362e+00]
 [ 2.30358169e+00  3.53941031e+00]
 [ 3.08235154e+00  4.29251986e+00]
 [ 3.45442536e+00  4.16228057e+00]
 [ 3.97492279e+00  3.04190077e+00]
 [ 3.88309099e+00  1.71616993e+00]
 [ 3.02376943e+00  4.40075144e-01]
 [ 1.65653597e+00 -4.77797038e-01]
 [ 6.25202394e-01 -5.53713406e-01]
 [-7.97972799e-17  2.77555756e-16]]
```



3.3 Future Work

- More curves should be tested with this sampling method
- Use the mean square error to evaluate the performance of the re-fitting process
- Look for possible developments to this sampling method if the performance is not satisfying enough
- Embed this backend server to the existing sketch interface.