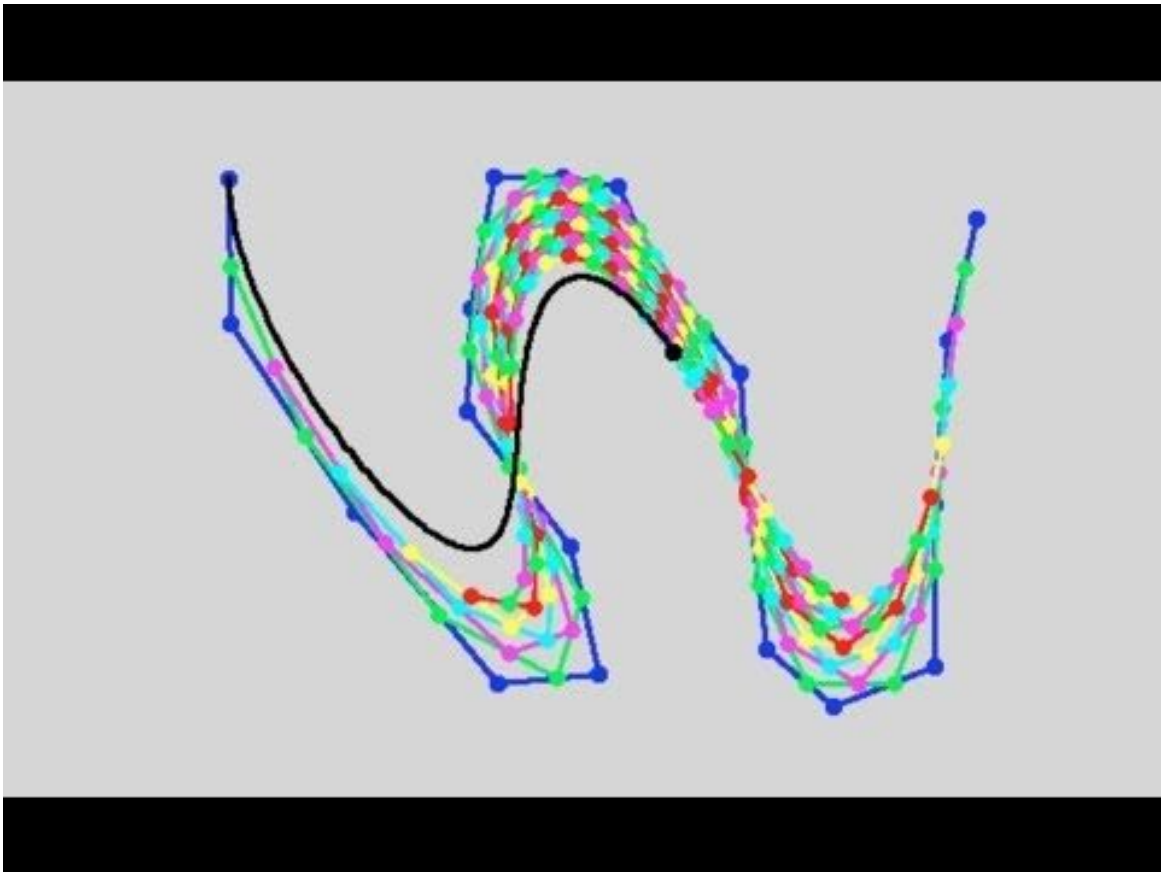
Advances in Robotics and Control

Assignment 4

B-Spline Curve

Nivedita Rufus, 2019702002



Question 1

Introduction

Let U be a set of ***m + 1*** non-decreasing numbers, $u_0 <= u_1 <= u_2 <= ... <= u_m$. The u_i***'s*** are called knots, the set ***U*** the knot vector, and the half-open interval $[u_i, u_{i+1})$ the i-th knot span. To define B-spline basis functions, we need one more parameter, the degree of these basis functions, ***p***. The i-th B-spline basis function of degree ***p***, written as $N_{i,p}(u)$, is defined recursively as follows:

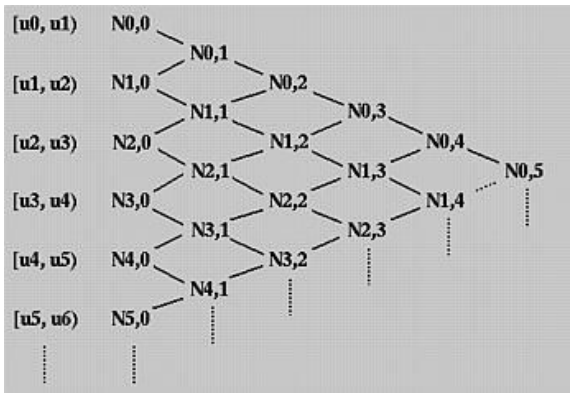
$$N_{i,0}(u) = \begin{cases} 1 & \text{if } u_i \leq u < u_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

$$N_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u)$$

For example:

$$N_{0,1}(u) = \frac{u - u_0}{u_1 - u_0} N_{0,0}(u) + \frac{u_2 - u}{u_2 - u_1} N_{1,0}(u)$$

The same can be extended to calculate the basis functions of higher-orders respectively,



Question: Write a code to plot basis functions of B-spline, given the degree p and knot vector U.

Steps of Approach

- Define parameters degree and knot vector of ***m+1*** elements which are nondecreasing in nature.
- Compute $N_{0,0}(u)$ for the defined knot vector.
- This is done in the following code block.

```
m = len(knot_vector)
n = m - degree - 1
u_vector = np.linspace(knot_vector[0], knot_vector[m-1], num_points)

N = []
for i in range(0, m-1):
      N_0 = []
      for u in u_vector:
            if(u>=knot_vector[i] and u<knot_vector[i+1]):
                  x = 1
            else:
                  x = 0
            N_0.append(x)
      N_0 = np.array(N_0)
      if(degree == 0):
            ax.step(u_vector, N_0, label = "N("+str(i)+","+str(degree)+")")
            plt.legend()
      N.append(N_0)
```

- Now compute the remaining basis functions for the higher-orders in adherance to the degree defined in the begining according to the formula,

$$N_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u)$$

Code:

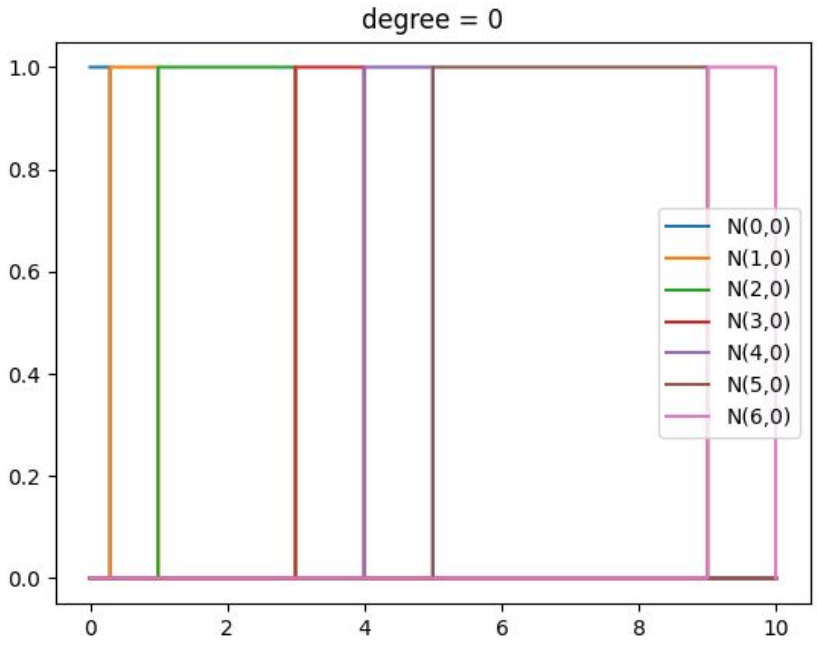
```
for p in range(1, (degree + 1)):
      points = []
      n = m - p -1
      for i in range(0, n):
            temp = []
            for j in range(0, len(u_vector)):
                  # print(p,i,j,n)
                  x = ((u_vector[j] - knot_vector[i])/(knot_vector[i+p] -
knot_vector[i]))*N[i][j]
                  y = ((knot_vector[i+p+1] -
u_vector[j])/(knot_vector[i+p+1] - knot_vector[i+1]))*N[i+1][j]

                  temp.append((x+y))
            points.append(np.array(temp))
      points = np.array(points)
      N = points
```

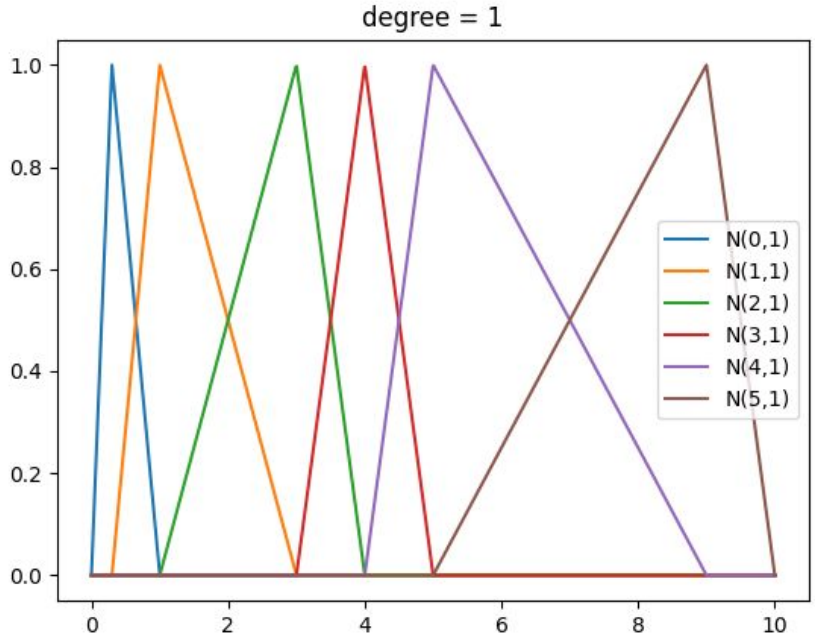
- Finally, plot the the coreesponding basis functions.

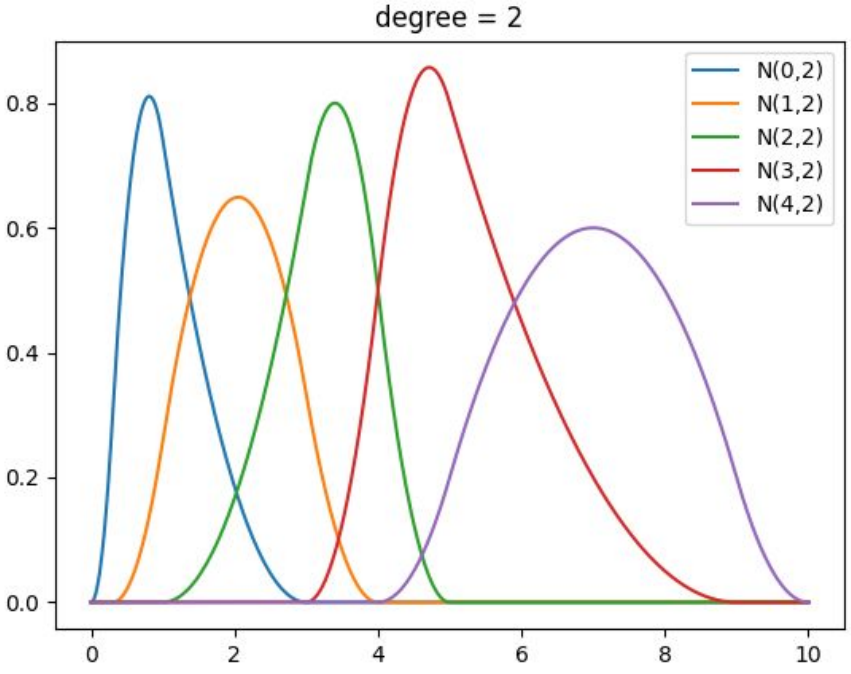
```
for i in range(len(N)):
      ax.plot(u_vector, N[i], label = "N("+str(i)+","+str(degree)+")")
      plt.legend()
```

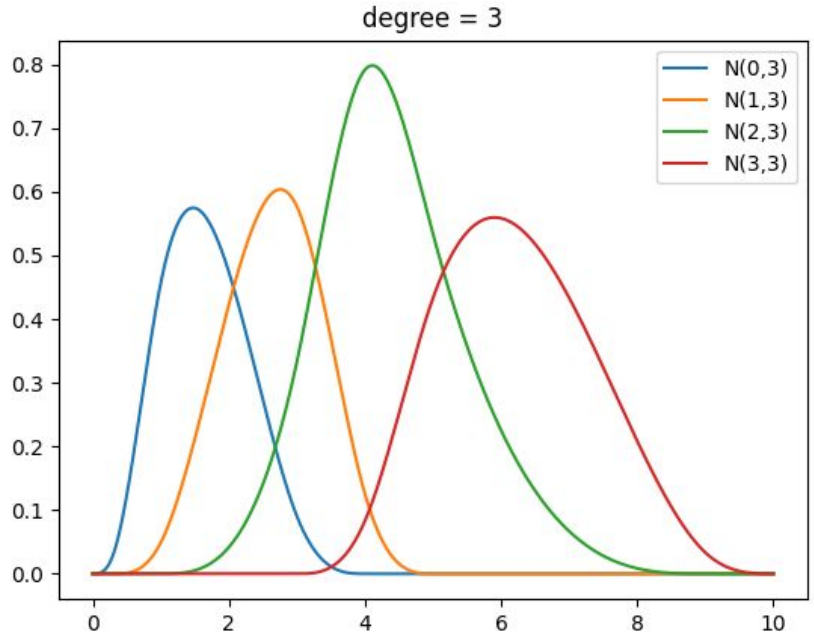
Results

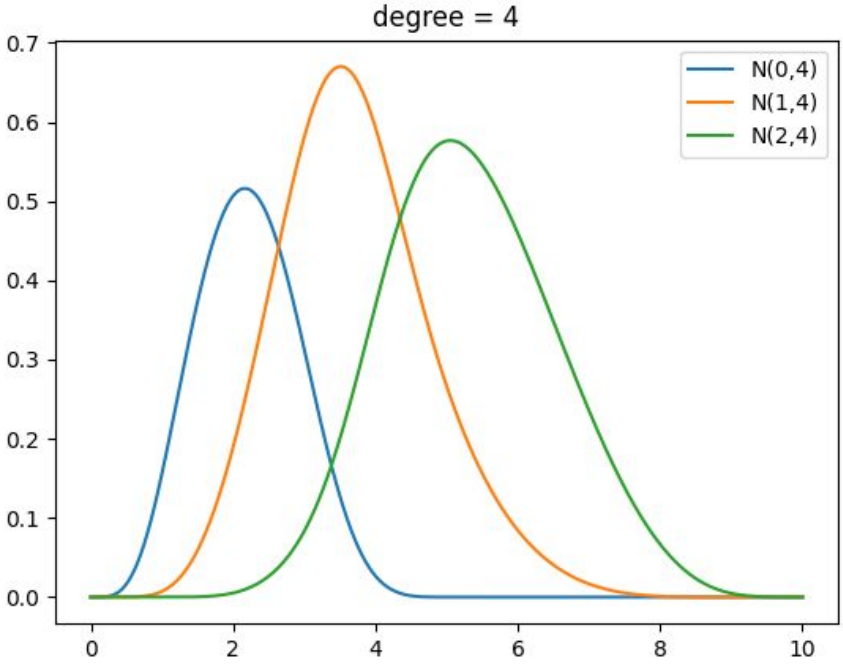
Knot vector = [0, 0.3, 1, 3, 4, 5, 9, 10]

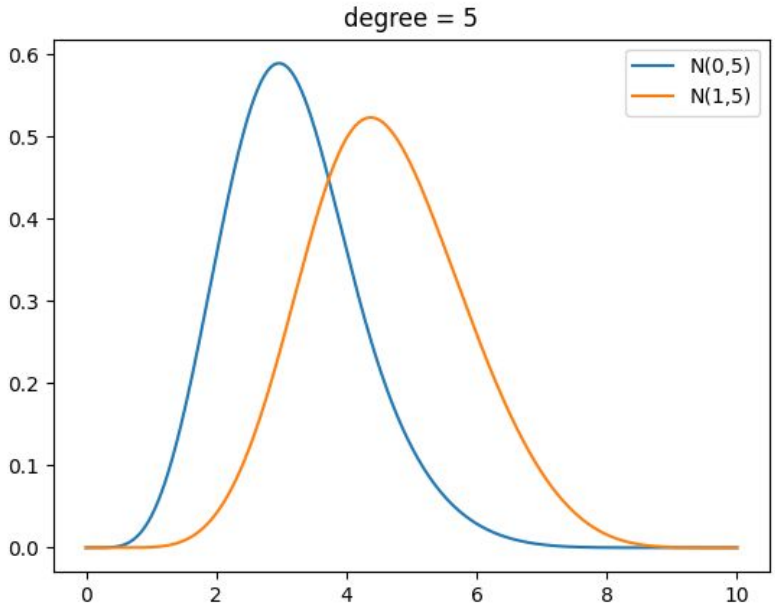


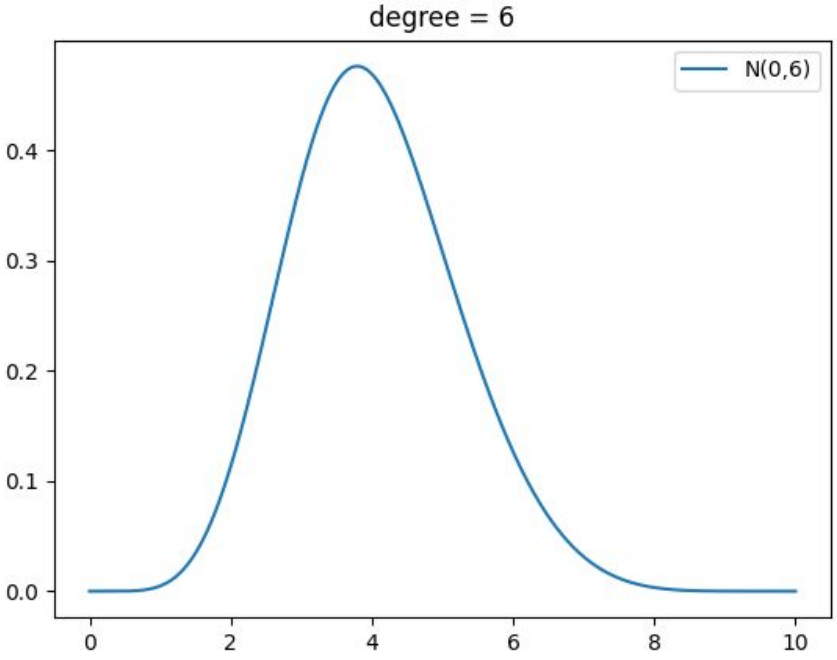

degree = 1
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10
N(0,1)
N(1,1)
N(2,1)
N(3,1)
N(4,1)
N(5,1)


degree = 2
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10
N(0,2)
N(1,2)
N(2,2)
N(3,2)
N(4,2)

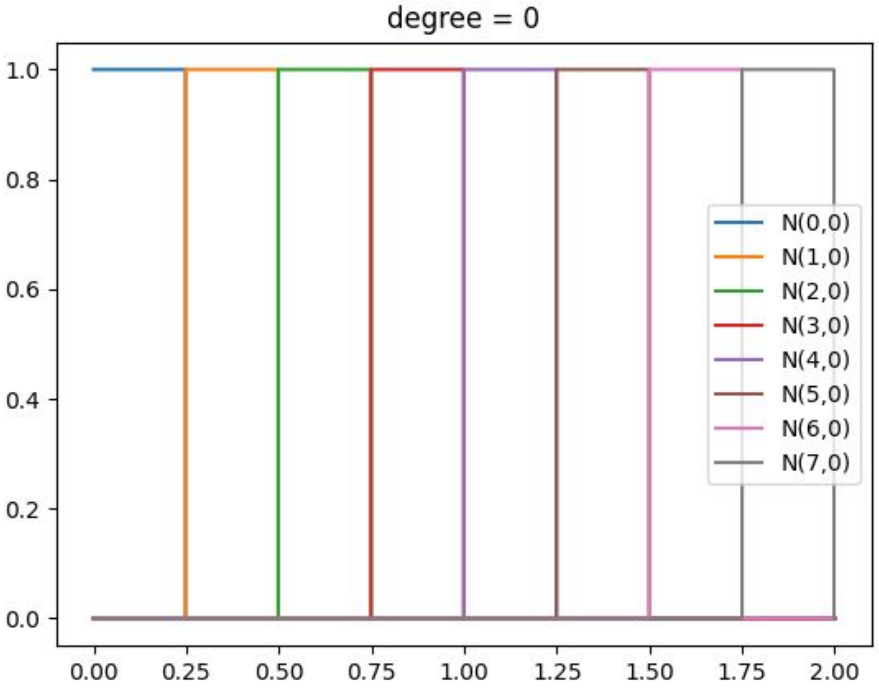

degree = 3
N(0,3)
N(1,3)
N(2,3)
N(3,3)
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
2
4
6
8
10

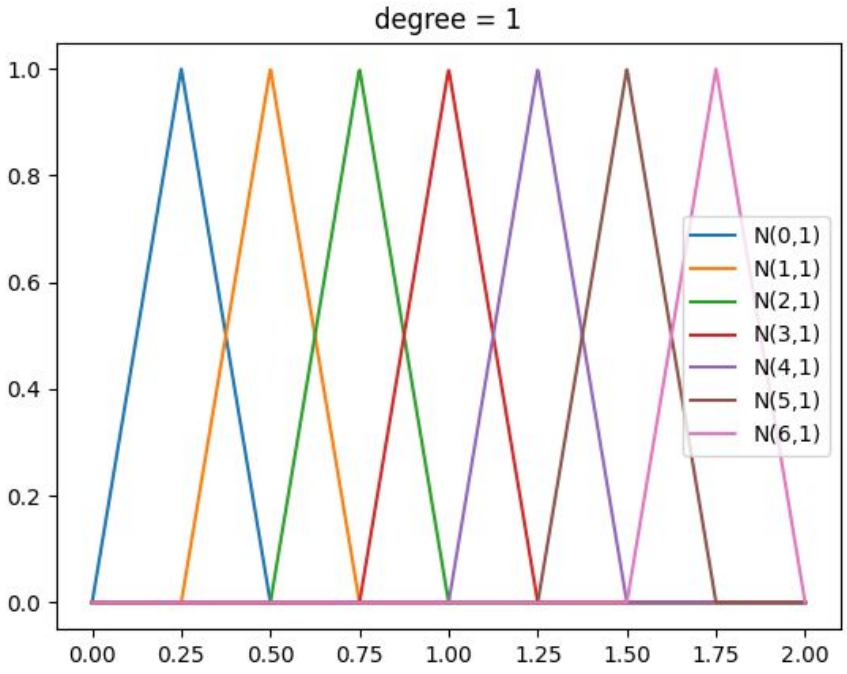

degree = 4
N(0,4)
N(1,4)
N(2,4)
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
2
4
6
8
10

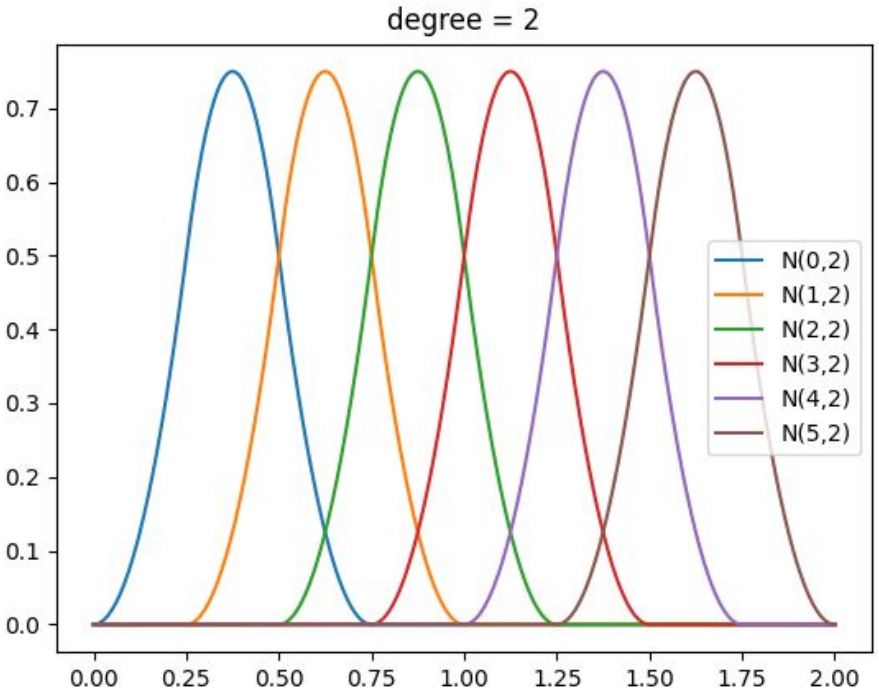

degree = 5
N(0,5)
N(1,5)
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
2
4
6
8
10

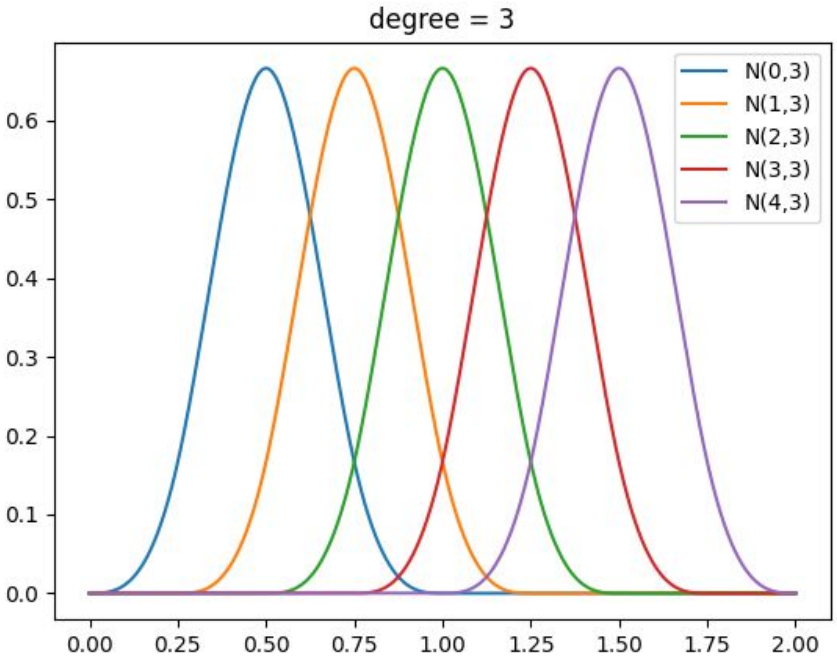

degree = 6
N(0,6)
0.4
0.3
0.2
0.1
0.0
0
2
4
6
8
10

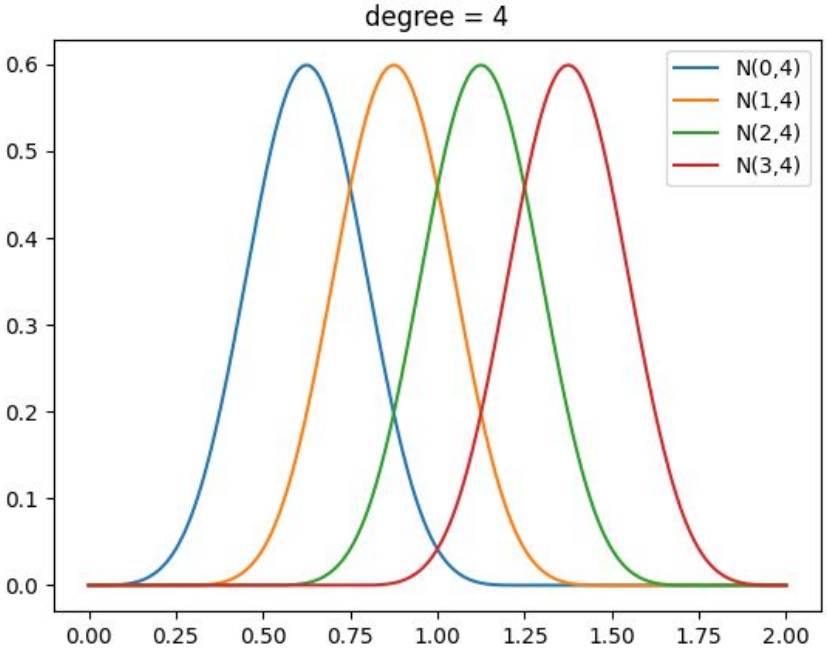
Knot vector = [0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2]

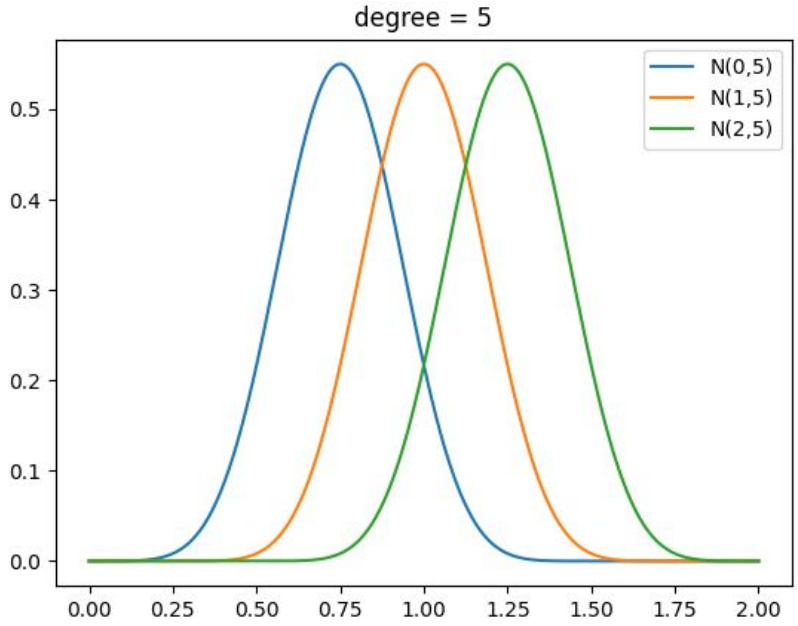


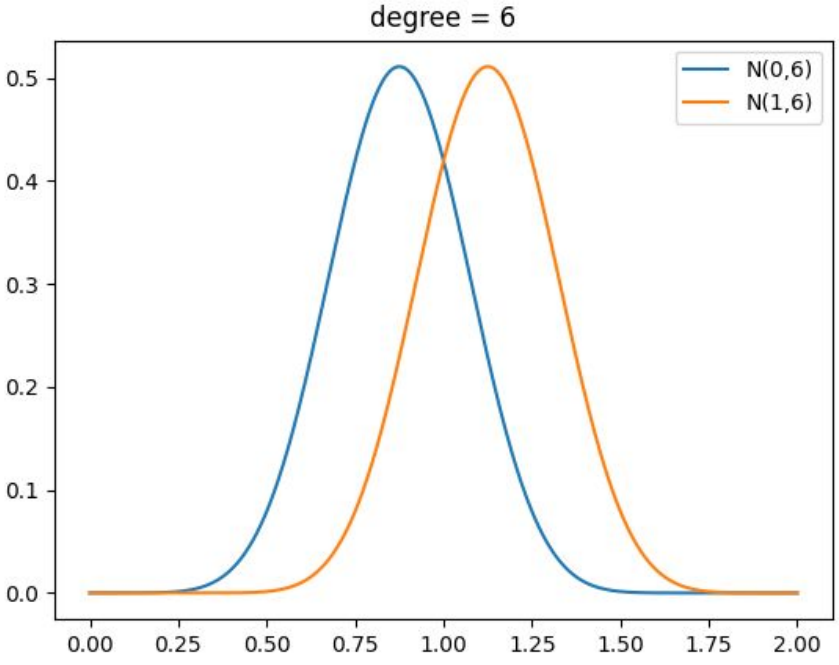

degree = 1
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
N(0,1)
N(1,1)
N(2,1)
N(3,1)
N(4,1)
N(5,1)
N(6,1)

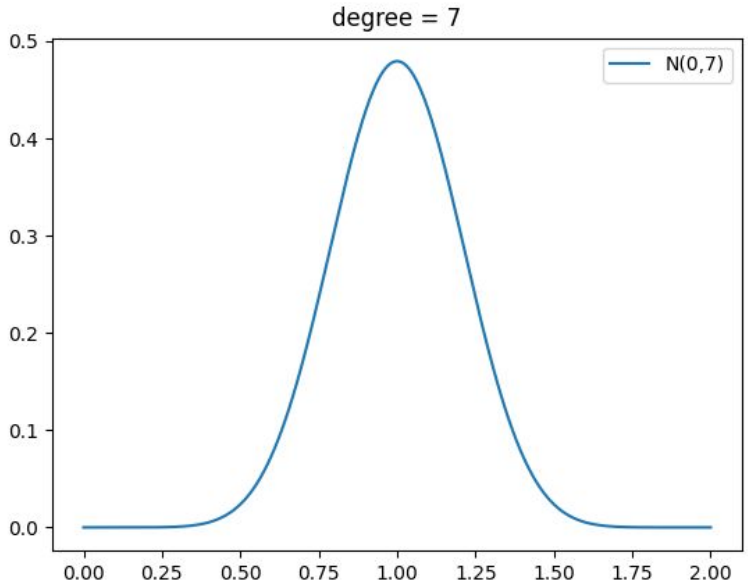

degree = 2
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
N(0,2)
N(1,2)
N(2,2)
N(3,2)
N(4,2)
N(5,2)


degree = 3
N(0,3)
N(1,3)
N(2,3)
N(3,3)
N(4,3)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00


degree = 4
N(0,4)
N(1,4)
N(2,4)
N(3,4)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00


degree = 5
N(0,5)
N(1,5)
N(2,5)
0.0
0.1
0.2
0.3
0.4
0.5
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00


degree = 6
N(0,6)
N(1,6)
0.0
0.1
0.2
0.3
0.4
0.5
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00



Question 2

Introduction

Moving control points is the most obvious way of changing the shape of a B-spline curve. The local modification scheme states that changing the position of control point P_i only affects the curve $C(u)$ on interval $[u_i, u_{i+p+1})$, where p is the degree of a B-spline curve. In fact, the shape change is translational in the direction of the control point being moved. More precisely, if control point P_i is moved in certain direction to a new position Qi , then point $C(u)$, where u is in $[u_i, u_{i+p+1})$, will be moved in the same direction from P_i to Q_i. However, the distance moved is different from point to point.

Suppose $C(u)$ is a given B-spline curve of degree p defined as follows:

$$\mathbf{C}(u) = \sum_{i=0}^{n} N_{i,p}(u)\mathbf{P}_i$$

Let control point P_i be moved to a new position $P_i + v$. Then, the new B-spline curve $D(u)$ of degree p is the following:

$$\begin{aligned}
\mathbf{D}(u) &= \sum_{k=0}^{i-1} N_{k,p}(u)\mathbf{P}_k + N_{i,p}(u)(\mathbf{P}_i + \mathbf{v}) + \sum_{k=i+1}^{n} N_{k,p}(u)\mathbf{P}_k \\
&= \sum_{k=0}^{n} N_{k,p}(u)\mathbf{P}_k + N_{i,p}(u)\mathbf{v} \\
&= \mathbf{C}(u) + N_{i,p}(u)\mathbf{v}
\end{aligned}$$

Question: For a robot agent of circular shape (diameter = 1 unit) a straight path is designed with a spline with two control points (top left, bottom right) in a map considering no obstacles. But, when the agent started moving from the initial point (0,0) and reached at (4,2) it is found that there are two obstacles (circular and polygonal shape) as shown in the Figure. Use the property of local modification of Spline (by using knot insertion, control point modification etc.) for modifying the path with minimal deviation from the straight line path. The path needs to be at least C^1 continuous at any point.

Steps of Approach

So, the task is given as to modify the path so as to avoid colliding with the obstacles:

1. Circular Obstacle with given centre and radius
2. Trapezoidal Obstacle with given vertices

The modification of the path is done according to control point modification as discussed in the introduction of this question. The path with minimal deviation is chosen after checking for collision.

This approach is briefed below along with code snippets:

- The first step is to generate the path from the start to end if the obstacles were not present.
- Now, we need to find the segments of the generated path which collide with the obstacles. This will be useful in shifting the path that will have the minimum deviation from the original path. This is done in the code snippet.

Collision with Circle:

```
def CollisionCircle(x, y, radius, center_x, center_y, radiusob):
    dist = np.hypot((x-center_x),(y-center_y))
    t = radius + radiusob
    if  t < dist:
        return False
    else:
        return True
```

Collision with trapezium:

```
def CollisionTrapezium(a, b, x, y, radius):
    x_pts = np.linspace(a[0], b[0], 50)
    y_pts = np.linspace(a[1], b[1], 50)
    c = False
    for i in range(0,x_pts.size):
        dist = np.hypot((x_pts[i]-x), (y_pts[i]-y))
        if radius > dist:
            c = True
    return c
```

- We identify the segments participating in collision in this code snippet:

```
def get_points_with_circle(pts, radius, x, y, ob_radius):
    cir_pts = []
    cir_idx = []
    for i in range(0, len(pts)):
        if(CollisionCircle(pts[i][0], pts[i][1], radius, x, y, ob_radius)):
```

```
            cir_pts.append(pts[i])
            cir_idx.append(i)
    return cir_pts, cir_idx

def get_points_with_trapezium(pts):
    trapezium_pts = []
    trapezium_idx = []
    for i in range(0,len(pts)):
        if(CollisionTrapezium([12,-3], [14,-3],pts[i][0],pts[i][1],0.5) or
CollisionTrapezium([14,-3], [16,-7], pts[i][0],pts[i][1],0.5) or
CollisionTrapezium([16,-7],[10,-6], pts[i][0],pts[i][1],0.5) or
CollisionTrapezium([10,-6],[12,-3],pts[i][0],pts[i][1],0.5)):
            trapezium_pts.append(pts[i])
            trapezium_idx.append(i)
    return trapezium_pts, trapezium_idx
```

- We now generate new control points for the identified segments using the following formula:

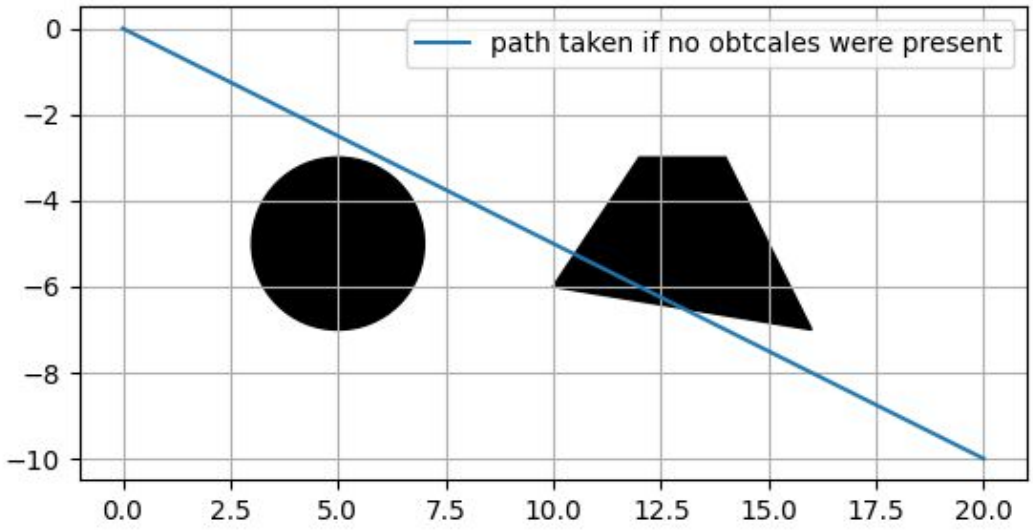
$$\begin{aligned}\mathbf{D}(u) &= \sum_{k=0}^{i-1} N_{k,p}(u)\mathbf{P}_k + N_{i,p}(u)(\mathbf{P}_i + \mathbf{v}) + \sum_{k=i+1}^{n} N_{k,p}(u)\mathbf{P}_k \\ &= \sum_{k=0}^{n} N_{k,p}(u)\mathbf{P}_k + N_{i,p}(u)\mathbf{v} \\ &= \mathbf{C}(u) + N_{i,p}(u)\mathbf{v}\end{aligned}$$

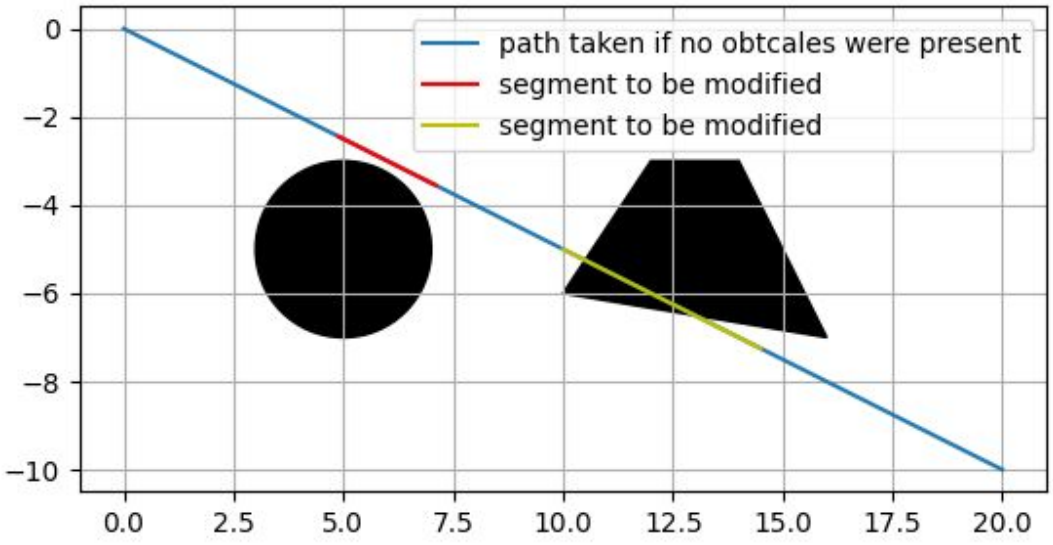
- To ensure that the modified path is at least C^1 continuous the degree chosen for the path is 3. To select the most optimal trajectory , we chose that path with least mean square distance from the original path, this ensures a path with minimum deviation.

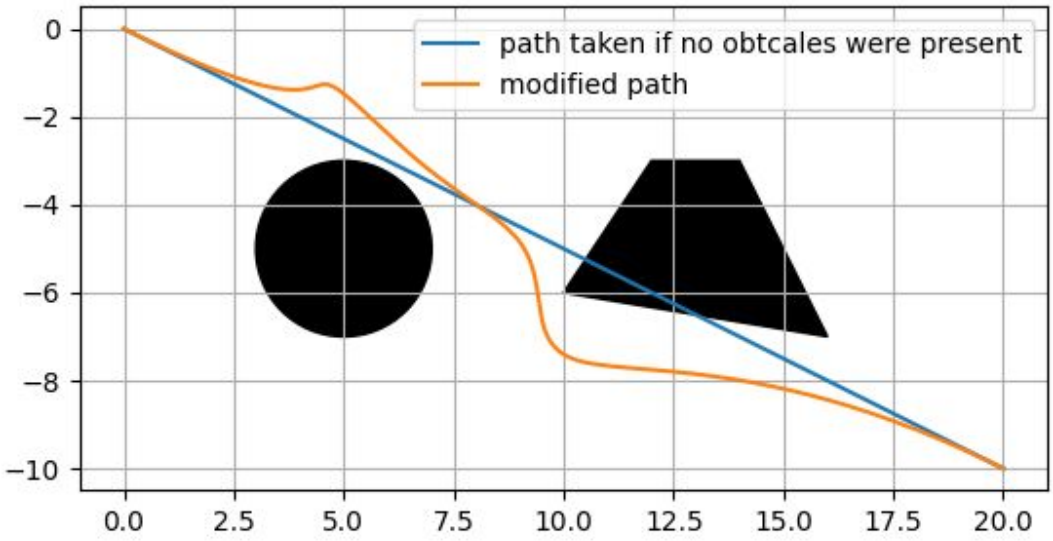
```
def get_optimal_trajectory(seg_points, new_knots, pts):
    length = np.linspace(0,11,30)
    breadth = np.linspace(0,11, 30)
    min_cost = 1000000000000
```

```
    for x1 in length:
        for y1 in breadth:
            for x2 in length:
                for y2 in breadth:
                    print(x1,y1,x2,y2)
                    control_points = [[4,-2], seg_points[0].tolist(),
[x1,-y1], seg_points[1].tolist(), seg_points[2].tolist(), [x2,-y2],
seg_points[3].tolist(), [20,-10]]
                    knots = [0, 0, 0, 0, (new_knots[0]/20),
(new_knots[1]/20), (new_knots[2]/20), (new_knots[3]/20), 1, 1, 1, 1]
                    curve, curve_points = generate_bspline(3,
control_points,knots)
                    dist = np.linalg.norm(curve_points-pts)
                    c = False
                    for i in range(0,curve_points.shape[0]):
                        if(CollisionTrapezium([10,-6],
[16,-7],curve_points[i][0],curve_points[i][1],0.5) or
CollisionTrapezium([12,-3],[16,-7],curve_points[i][0],curve_points[i][1],0.
5) or CollisionCircle(curve_points[i][0], curve_points[i][1], 0.5, 5, -5,
2)):
                            c = True
                            break
                    if dist<min_cost and c==False:
                        min_params = [x1,-y1, x2,-y2]
                        min_cost = dist
    return min_params, min_cost
```

Results


path taken if no obtcales were present
0
−2
−4
−6
−8
−10
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


path taken if no obtcales were present
segment to be modified
segment to be modified
0
−2
−4
−6
−8
−10
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


path taken if no obtcales were present
modified path
0
−2
−4
−6
−8
−10
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0