Numerical Analysis for finding a short and smooth path for robot traversal between obstacles

Rishabh Bajpai

Katyayani Trivedi

Abstract:

In this paper, we have tried to make robot traversal between obstacles more efficient by finding the path that is shorter as well as smoother compared to the ones obtained by other obstacle-avoiding algorithms. We have incorporated shortest path finding algorithms and curve fitting methods in our code. The robot will use 2D co-ordinate system.

Keywords:

Python3.6, shortest path Algorithm, Cubic spline interpolation, iterations, intersection of lines, sets, graph, weight

Introduction:

In traditional methods, people have used obstacle-detecting sensors to sense the presence of any obstacle in the robot's path. This method was quite time-taking as the robot has to sense every obstacle and then choose a path accordingly.

Even if we have predefined data of the position of the obstacles, the path that the robot follows comes out to be sharp which further reduces the speed and efficiency, thus increasing time of travel.

Hence, for best locomotion of a robot, it should have constant velocity all along which can only be ensured with the help of smooth curves. So the path it follows should be short as well as smooth.

Getting the data:

Step 1: Input the co-ordinates of the starting and end point in a list.

Step 2: Enter the number of obstacles.

Step 3: Through iterations all the co-ordinates of the obstacles are put in a list

#Python code

```
x= [xstart, obs1,obs2,…….,obsn,xend]

y= [ystart, obs1,obs2,…….,obsn,yend]
```

Graph creation:

Step 1: Create vertices for starting point, ending point and obstacles.

#Python code

```
for i in range(2*count+2):

    G.add_vertex(ch)
    m=ord(ch)
    m+=1
    ch=chr(m)
```

Step 2: Create edges and calculate weights.

#Python code

```
care =np.zeros((len(x),len(x)))
for i in range(2*count+2):
    for j in range(2*count+2):
        temp=0
        if(i!=j):
            if (i%2!=0)and(j==(i+1)):
                continue
            else:
                for k in range(1,2*count+1):
                    for l in range(1,2*count + 1):
                        temp_x =[]
                        temp_x.append(x[i])
                        temp_x.append(x[j])
                        temp_x.append(x[k])
                        temp_x.append(x[l])

                        temp_y =[]
                        temp_y.append(y[i])
                        temp_y.append(y[j])
                        temp_y.append(y[k])
                        temp_y.append(y[l])
                        for i1 in range
                                (len(temp_x)):

                            # Find the minimum
                              element in remaining
                            # unsorted array
                            min_idx = i1
                            for j1 in range(i1 + 1,
                                    len(temp_x)):
                                if temp_x[min_idx]
```

```
                     > temp_x[j1]:
                  min_idx = j1

              # Swap the found minimum
element with
              # the first element
              temp_x[i1], temp_x[min_idx]
= temp_x[min_idx], temp_x[i1]
              temp_y[i1], temp_y[min_idx]
= temp_y[min_idx], temp_y[i1]

            fx1=temp_x[1]*(y[i]-
y[j])+temp_y[1]*(x[j]-x[i])-y[i]*(x[j]-x[i])-
x[i]*(y[i]-y[j])
            fx2=temp_x[2]*(y[i]-
y[j])+temp_y[2]*(x[j]-x[i])-y[i]*(x[j]-x[i])-
x[i]*(y[i]-y[j])
            print(i,j,k,l,fx1,fx2)
            if fx1*fx2<0:
              ch_e1 = chr(i + 65)
              ch_e2 = chr(j + 65)
              G.del_edge(ch_e1,ch_e2)
              temp=1
              care[i][j]=1

      if temp==0:
        if care[i][j]!=1:

          ch_e1 = chr(i + 65)
          ch_e2 = chr(j + 65)
          distance = pow((pow((x[i] - x[j]),
2) + pow((y[i] - y[j]), 2)), 0.5)
          G.add_edge(ch_e1, ch_e2, distance)
```

Note: In calculating weights of different edges, we may have some extra edges (paths), which are not desired. So to remove redundancy we use an array which stores all invalid paths (care array).

Shortest path algorithm:

We have many shortest path algorithms:

Dijkstra's algorithm solves the single-source shortest path problem.

Bellman–Ford algorithm solves the single-source problem if edge weights may be negative.

A search algorithm solves for single pair shortest path using heuristics to try to speed up the search.

Floyd–Warshall algorithm solves all pair of shortest paths.

Johnson's algorithm solves all pair of shortest paths, and may be faster than Floyd–Warshall on sparse graphs.

Viterbi algorithm solves the shortest stochastic path problem with an additional probabilistic weight on each node.

But for this problem none of the above algorithm is going to give desired results.

So we have modified Dijkstra's algorithm:

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a *SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.

In this algorithm, the value of weight, number of vertices and all edges are provided by user. But in real time situation Robot should calculate all these things by itself. So we have modified the algorithm accordingly.

We have created tree for implementing the algorithm.

From a map of all the possible paths from the starting point to the ending point, a path with the minimum sum of the weights is chosen.

As an output of the algorithm, we are getting a list of vertices in sequential order.

The list we get is in the notation of vertices (e.g. A, B, C). So, we have to convert them to their respective co-ordinates.

Smoothing the path:

As we know that the slope of a cubic spline at any point is defined and continues, so we have considered it for smoothing the curve.

If there are more than a few data points, a cubic spline is hard to beat as a global interpolant. It is considerably “stiffer” than a polynomial in the sense that it has less tendency to oscillate between data points.

The mechanical model of a cubic spline is a thin, elastic beam that is attached with pins to the data points. Because the beam is unloaded between the pins, each segment of the spline curve is a cubic polynomial—recall from beam theory that $d4y/dx4 = q/(E\, I)$, so that $y(x)$ is a cubic since $q = 0$. At the pins, the slope and bending moment (and hence the second derivative) are continuous. There is no bending moment at the two end pins; consequently, the second derivative of the spline is zero at the end points. Because these end conditions occur naturally in the beam model, the resulting curve is known as the *natural cubic spline*. The pins (i.e., the data points) are called the *knots* of the spline.

# Python code

```
# module cubicSpline
''' k = curvatures(xData,yData).
Returns the curvatures of cubic spline at its
knots.
y = evalSpline(xData,yData,k,x).
Evaluates cubic spline at x. The curvatures k can
be
computed with the function 'curvatures'.
'''
import numpy as np
from LUdecomp3 import *
def curvatures(xData,yData):
    n = len(xData) - 1
    c = np.zeros(n)
    d = np.ones(n+1)
    e = np.zeros(n)
    k = np.zeros(n+1)
    c[0:n-1] = xData[0:n-1] - xData[1:n]
    d[1:n] = 2.0*(xData[0:n-1] - xData[2:n+1])
    e[1:n] = xData[1:n] - xData[2:n+1]
    k[1:n] =6.0*(yData[0:n-1] - yData[1:n]) \
    /(xData[0:n-1] - xData[1:n]) \
    -6.0*(yData[1:n] - yData[2:n+1]) \
    /(xData[1:n] - xData[2:n+1])
    LUdecomp3(c,d,e)
    LUsolve3(c,d,e,k)
return k

def evalSpline(xData,yData,k,x):

    def findSegment(xData,x):
        iLeft = 0
```

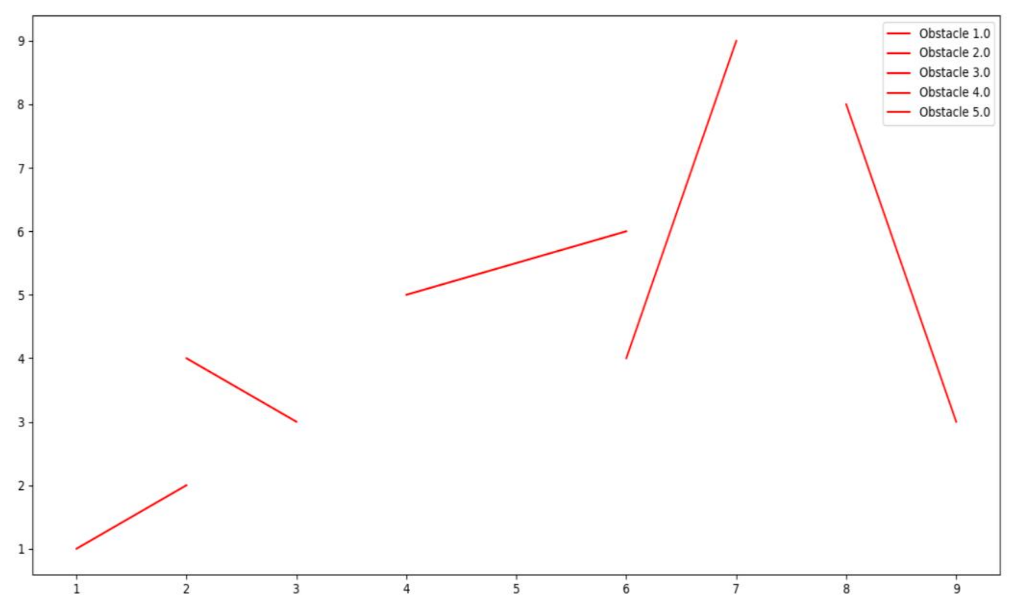
```
            iRight = len(xData)- 1
            while 1:
            if (iRight-iLeft) <= 1:
               return iLeft
            i =(iLeft + iRight)/2

            if x < xData[i]:
                iRight = i
            else:
                iLeft = i
            i = findSegment(xData,x)
            h = xData[i] - xData[i+1]
            y = ((x - xData[i+1])**3/h - (x -
            xData[i+1])*h)*k[i]/6.0 \
            - ((x - xData[i])**3/h - (x -
            xData[i])*h)*k[i+1]/6.0 \
            + (yData[i]*(x - xData[i+1]) \
            - yData[i+1]*(x - xData[i]))/h
    return y
```

Plotting the graph:

Graph 1: The data from the initial lists is used to plot a graph of all the obstacles.

#Python Code

```
    for i in range(int((len(x)-2)/2)):
        i=2*i+1
        x1 = np.linspace(x[i], x[i+1], num=50)
        y1 = y[i] + ((y[1+i] - y[i+0]) / (x[i+1] -
    x[i+0])) * (x1 - x[i+0])
        c='Obstacle '+str((i-1)/2+1)
        plt.plot(x1, y1,label=c,color='red')
```



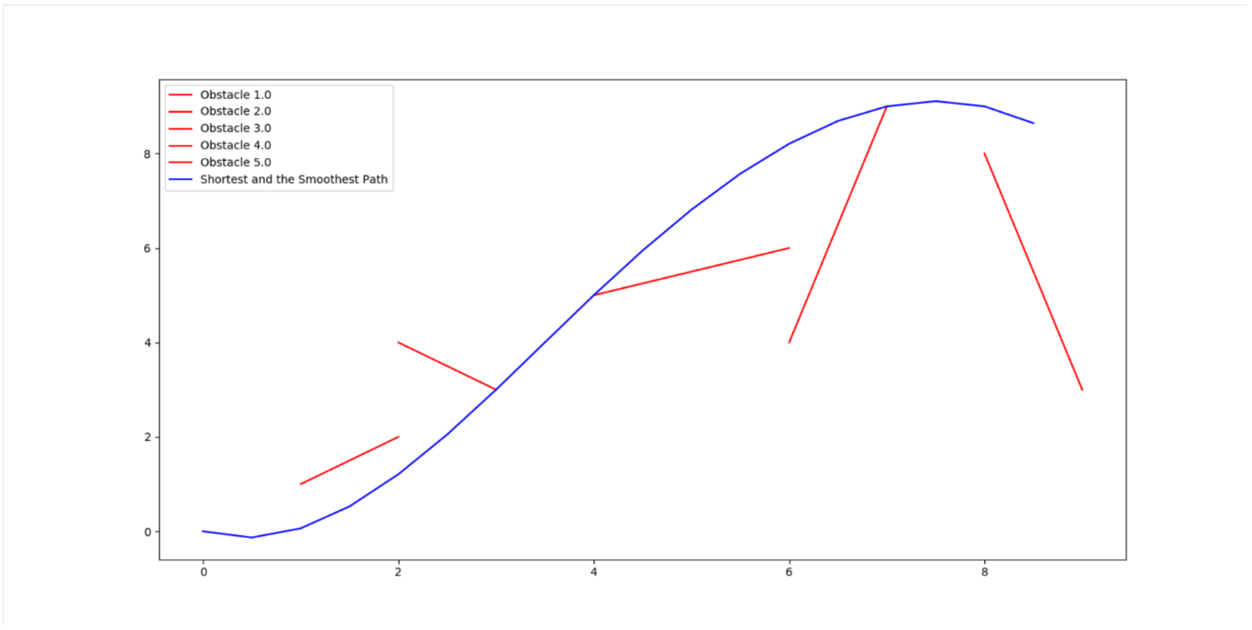
Graph 2: This graph shows the final path that is obtained indicated by blue color.

If the number of turns in the final path is less than 4, then there is no need to make it smooth and straight lines are used to join them.

#Python code

```
if len(ans_x)<4:
        print("The path is almost straight, so no
                   need to make it smooth")
for i in range(len(ans_x)-1):
           x1 = np.linspace(ans_x[i], ans_x[i+1] +
                   0.05, num=50)
           y1 = ans_y[i] + ((ans_y[1+i] - ans_y[i+0])
/ (ans_x[i+1] - ans_x[i+0])) * (x1 - ans_x[i+0])

plt.plot(x1, y1,color='blue')
```

```
else:
    tck = interpolate.splrep(ans_x, ans_y, s=0)
    xnew = np.arange(ans_x[0], (ans_x[len(ans_x) -
1] + 1), 0.5)
    ynew = interpolate.splev(xnew, tck, der=0)
    plt.plot(xnew, ynew, label='Shortest and the
Smoothest Path',color='blue')


plt.legend()
plt.show()
```



Final Output path

Limitations:

For some values of co-ordinates the program may fail, due to division of some number with zero.

Conclusion:

In this paper an attempt has been made to find out a path for robot traversal between obstacles, that is both short and smooth. We have seen how the traditional Dijkistra algorithm couldn’t work efficiently here, and thus it has been replaced by a modified version. For smoothing out the curves, cubic spline interpolation has been incorporated.

References:

1. https://github.com/
2. Numerical Methods in Engineering with Python3 – Jaan Kiusalaas
3. Numerical Methods for Engineers – Steven C. Chapra, Raymond P. Canale
4. https://matplotlib.org/tutorials/index.html
5. https://math.stackexchange.com/