NM: Assignment1

Class	
Created	
Materials	
Reviewed	☐
Type	

Info:

- Hany Hamed
- BS4-Robotics
- Fall 2021, Innopolis Univeristy

Sources:

- Formulation and implementation are based on:
  - www.math.uh.edu/~jingqiu/math4364/spline.pdf or https://web.archive.org/web/20150702075205/https://www.math.uh.edu/~jingqiu/math4364/spline.pdf
  - https://www.uio.no/studier/emner/matnat/math/MAT-INF4130/h17/book2017.pdf chapter 1
  - http://www.thevisualroom.com/tri_diagonal_matrix.html
- Lecture materials
- YouTube Explanation: https://www.youtube.com/watch?v=LaolbjAzZvg
- https://towardsdatascience.com/numerical-interpolation-natural-cubic-spline-52c1157b98ac
- https://people.cs.clemson.edu/~dhouse/courses/405/notes/splines.pdf
- https://timodenk.com/blog/cubic-spline-interpolation/
- Hornbeck, H., 2020. Fast Cubic Spline Interpolation. arXiv preprint arXiv:2001.09253.

Task:

Implement a function for Spline 3rd order (Cubic) interpolation

Explanation:

Cubic Spline Formulation:

The formulation for the Cubic Spline:

$$S_{3,n}(x) = \begin{cases} p_1(x) & = a_1 + b_1 x + c_1 x^2 + d_1 x^3, & x \in [x_0, x_1], \\ p_2(x) & = a_2 + b_2 x + c_2 x^2 + d_2 x^3, & x \in [x_1, x_2], \\ & \vdots & \\ p_n(x) & = a_n + b_n x + c_n x^2 + d_n x^3, & x \in [x_{n-1}, x_n]; \end{cases}$$

Interpolation using Cubic spline:

$$S_{3,n}(x_i) = f_i, \quad i = 0, 1, \ldots, n.$$

Conditions:

Smoothness conditions:

$$p_i'(x_i) = p_{i+1}'(x_i), \quad p_i''(x_i) = p_{i+1}''(x_i), \quad i = 1, 2, \ldots, n-1.$$

Interpolation conditions:

$$p_i(x_{i-1}) = f_{i-1}, \quad p_i(x_i) = f_i, \quad i = 1, 2, \ldots, n.$$

Boundary conditions:

Here, the coding implementation is based on the natural cubic spline, thus the following conditions hold:

$$p_1''(x_0) = 0, \quad p_n''(x_n) = 0.$$

But the rest of the formulation, for now, is using the complete cubic spline:

$$p_1'(x_0) = f'(x_0), \quad p_n'(x_n) = f'(x_n)$$

Process:

Let us define (the 2nd derivative of the spline).

$$z_i = S_{3,n}''(x_i), \; 0 \le i \le n.$$

For each interior knot

$$p_i''(x_i) = z_i = p_{i+1}''(x_i), \quad 1 \le i \le n-1.$$

$P_{i+1}(x_i)$ is a cubic polynomial on $[x_i, x_{i+1}]$, thus the 2nd derivative is linear $P_{i+1}''(x_i) = z_i$ and $P_{i+1}''(x_{i+1}) = z_{i+1}$ and we can get the following:

$$p_{i+1}''(x) = \frac{z_i}{h_{i+1}}(x_{i+1} - x) + \frac{z_{i+1}}{h_{i+1}}(x - x_i),$$

h_{i+1} in the code is defined by delta_x

$$h_{i+1} \equiv x_{i+1} - x_i$$

By integrating the above equation twice, such that C and D are the constants of the integration

$$p_{i+1}(x) = \frac{z_i}{6h_{i+1}}(x_{i+1} - x)^3 + \frac{z_{i+1}}{6h_{i+1}}(x - x_i)^3 + C(x - x_i) + D(x_{i+1} - x) \tag{1}$$

By substitution of Interpolation conditions:

$$p_{i+1}(x_i) = f_i \qquad p_{i+1}(x_{i+1}) = f_{i+1}$$

$$\frac{z_i}{6h_{i+1}}(x_{i+1}-x_i)^3 + D(x_{i+1}-x_i) = f_i \qquad \frac{z_{i+1}}{6h_{i+1}}(x_{i+1}-x_i)^3 + C(x_{i+1}-x_i) = f_{i+1}$$

That follows:

$$C = \left(\frac{f_{i+1}}{h_{i+1}} - \frac{z_{i+1}h_{i+1}}{6}\right), \quad D = \left(\frac{f_i}{h_{i+1}} - \frac{z_i h_{i+1}}{6}\right)$$

Substitution in (1):

$$\begin{aligned} p_{i+1}(x) &= \frac{z_i}{6h_{i+1}}(x_{i+1}-x)^3 + \frac{z_{i+1}}{6h_{i+1}}(x-x_i)^3 + \left(\frac{f_{i+1}}{h_{i+1}} - \frac{z_{i+1}h_{i+1}}{6}\right)(x-x_i) \\ &\quad + \left(\frac{f_i}{h_{i+1}} - \frac{z_i h_{i+1}}{6}\right)(x_{i+1}-x). \end{aligned}$$

1st derivative:

$$\begin{aligned} p'_{i+1}(x) &= -\frac{z_i}{2h_{i+1}}(x_{i+1}-x)^2 + \frac{z_{i+1}}{2h_{i+1}}(x-x_i)^2 + \left(\frac{f_{i+1}}{h_{i+1}} - \frac{z_{i+1}h_{i+1}}{6}\right) \\ &\quad - \left(\frac{f_i}{h_{i+1}} - \frac{z_i h_{i+1}}{6}\right), \end{aligned}$$

After simplification (2)

$$p'_{i+1}(x_i) = -\frac{h_{i+1}}{3}z_i - \frac{h_{i+1}}{6}z_{i+1} - \frac{f_i}{h_{i+1}} + \frac{f_{i+1}}{h_{i+1}}$$

And for the same for i index (3)

$$p'_i(x_i) = \frac{h_i}{6}z_{i-1} + \frac{h_i}{3}z_i - \frac{f_{i-1}}{h_i} + \frac{f_i}{h_i}.$$

and according to the following continuity condition

$$p'_i(x_i) = p'_{i+1}(x_i)$$

We have the following:

$$h_i z_{i-1} + 2(h_i + h_{i+1})z_i + h_{i+1}z_{i+1} = \frac{6}{h_{i+1}}(f_{i+1}-f_i) - \frac{6}{h_i}(f_i - f_{i-1}), \quad 1 \le i \le n-1$$

This is the system of n-1 linear equations for n+1 unknowns, in order to get the rest of the sufficient linear equations, we apply the boundary conditions for the complete spline

i = 0 and substitute in (2)

$$2h_1 z_0 + h_1 z_1 = \frac{6}{h_1}(f_1 - f_0) - 6f'_0$$

i = n and substitute in (3)

$$h_n z_{n-1} + 2h_n z_n = 6f'_n - \frac{6}{h_n}(f_n - f_{n-1})$$

By combining the last three equations, we get a system of linear equations: $A\mathbf{z} = \mathbf{d}$

Such that

$$\mathbf{z} = [z_0, z_1, \cdots, z_n]^T, \quad \mathbf{d} = [d_0, d_1, \cdots, d_n]^T,$$

$$d_i = \begin{cases} \frac{6}{h_1}(f_1 - f_0) - 6f'_0, & i = 0, \\ \frac{6}{h_{i+1}}(f_{i+1} - f_i) - \frac{6}{h_i}(f_i - f_{i-1}), & 1 \le i \le n-1, \\ 6f'_n - \frac{6}{h_n}(f_n - f_{n-1}), & i = n, \end{cases}$$

A is a tridiagonal matrix

$$A = \begin{bmatrix} 2h_1 & h_1 & & & \\ h_1 & 2(h_1+h_2) & h_2 & & \\ & h_2 & 2(h_2+h_3) & h_3 & \\ & \ddots & \ddots & \ddots & \\ & & h_{n-1} & 2(h_{n-1}+h_n) & h_n \\ & & & h_n & 2h_n \end{bmatrix}$$

And the tridiagonal systems of equations can be solved by Tri-Diagonal Matrix Algorithm (TDMA) or Thomas Algorithm

Code

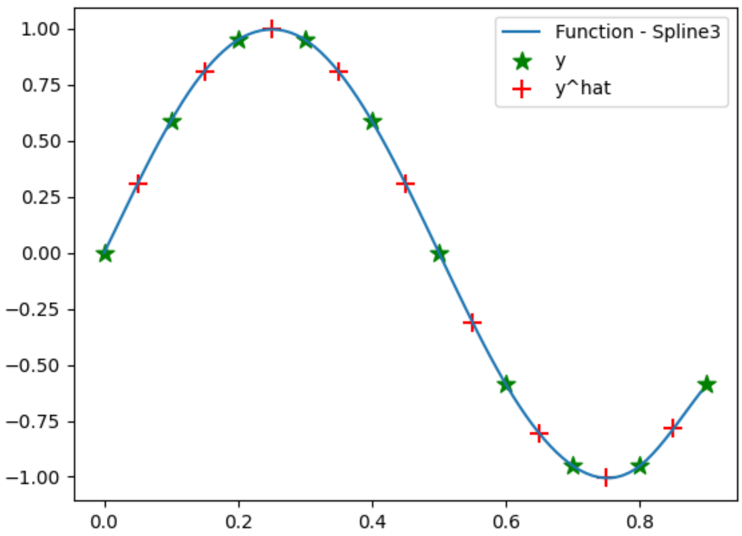
```
# Class implementation
class Spline3:
    def __init__(self):
        self.coeff = None

    def fit(self, X, Y):
        self.X = X
        self.Y = Y

        delta_x = self._diff(X)
        delta_y = self._diff(Y)

        # Initiate the slices of the matrices
        dim = len(X)
        diagonal_i = [None for _ in range(dim)] # Slice from the Diagonal matrix for i
        diagonal_i1 = [None for _ in range(dim-1)]  # Slice from the Diagonal matrix for i-1
        self.C = [None for _ in range(dim)]

        # fill diagonals matrices
        diagonal_i[0] = sqrt(2*delta_x[0])
        diagonal_i1[0] = 0.0

        # Solve [L][y] = [B]
        # "The natural spline is defined as setting the second derivative of the first and the last polynomial equal to zero in the in
        B0 = 0.0 # natural boundary condition related to the 2nd derivative of the first polynomial = 0
        self.C[0] = B0 / diagonal_i[0]
```

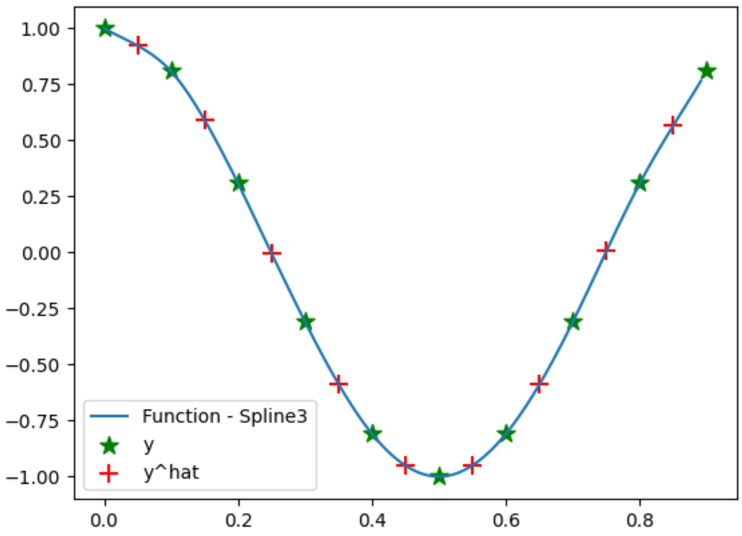
```
        for i in range(1, dim-1, 1):
            diagonal_i1[i] = delta_x[i-1] / diagonal_i[i-1]
            diagonal_i[i] = sqrt(2*(delta_x[i-1]+delta_x[i]) - diagonal_i1[i-1] * diagonal_i1[i-1])
            Bi = 6*(delta_y[i]/delta_x[i] - delta_y[i-1]/delta_x[i-1])
            self.C[i] = (Bi - diagonal_i1[i-1]*self.C[i-1])/diagonal_i[i]

        # Last polynomial
        i = dim - 1
        diagonal_i1[i-1] = delta_x[-1] / diagonal_i[i-1]
        diagonal_i[i] = sqrt(2*delta_x[-1] - diagonal_i1[i-1] * diagonal_i1[i-1])
        Bi = 0.0 # # natural boundary condition related to the 2nd derivative of the last polynomial = 0
        self.Z[i] = (Bi - diagonal_i1[i-1]*self.Z[i-1])/diagonal_i[i]


        # solve [L.T][x] = [y]
        i = dim-1
        self.Z[i] = self.Z[i] / diagonal_i[i]
        for i in range(dim-2, -1, -1):
            self.Z[i] = (self.Z[i] - diagonal_i1[i-1]*self.Z[i+1])/diagonal_i[i]


        self.coeff = {"X": self.X, "Y": self.Y, "Z": self.Z}    # not (x,y,z), Z is not the 3rd dimension in the euclidean space, it i

    def _diff(self, x):
        new_x = [x[i] - x[i-1] for i in range(1, len(x))]
        return new_x

    def compute(self, x, eps=0.00001):
        # Find the nearest neighbors for the interpolated point
        index = 0
        for i in range(len(self.X)):
            if(self.X[i] - x > eps):
                index = i
                break

        x1, x0 = self.X[index], self.X[index-1]   # Neighbours from x-axis
        y1, y0 = self.Y[index], self.Y[index-1]   # Y-axis correspondeing values to the neighbours
        z1, z0 = self.Z[index], self.Z[index-1]
        h1 = x1 - x0                             # difference between the x-axis

        # calculate cubic
        y = z0/(6*h1)*(x1-x)**3 + \
            z1/(6*h1)*(x-x0)**3 + \
            (y1/h1 - z1*h1/6)*(x-x0) + \
            (y0/h1 - z0*h1/6)*(x1-x)
        return y

    def get_coeff(self):
        return self.coeff

    def func_out(self, a, b, num_bins=100):
        bin_dim = (b-a)/100
        x = [a+i*bin_dim for i in range(num_bins)]
        bins = [self.compute(i) for i in x]
        return x, bins
```

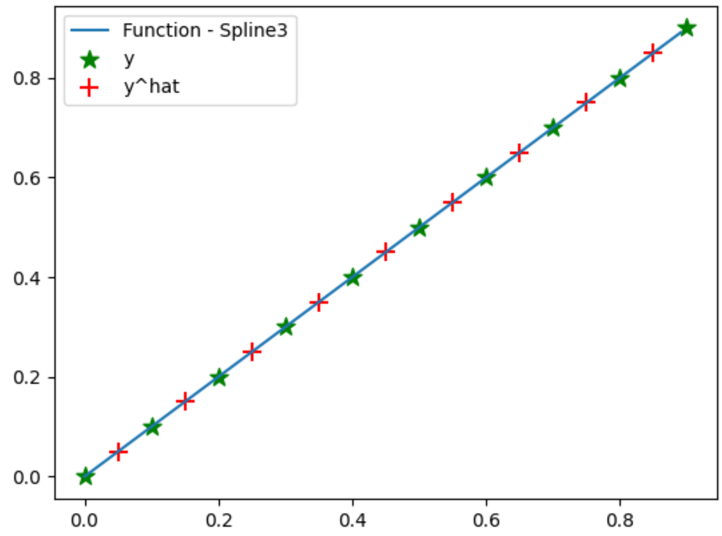
```
# Input data
n = 0
m = 0
k = 0
x = []
y = []
x_hat = []

# Using input from cli
# n = int(input())
# for i in range(n):
#     x_i = float(input())
#     x.append(x_i)

# m = int(input())
# for i in range(m):
#     y.append([])
#     for j in range(n):
#         y_j_i = float(input()) # y_j^i
#         y[-1].append(y_j_i)

# k = int(input())
# for i in range(k):
#     x_hat_i = float(input())
#     x_hat.append(x_hat_i)

# Using txt file
with open("input.txt", "r") as f:
```

```
    lines = f.readlines()

    n = int(lines[0].strip())
    line = lines[1].strip().split(" ")
    for i in range(n):
        x_i = float(line[i])
        x.append(x_i)

    m = int(lines[2].strip())
    for i in range(m):
        line = lines[3+i].strip().split(" ")
        y.append([])
        for j in range(n):
            y_j_i = float(line[j]) # y_j^i
            y[-1].append(y_j_i)

    k = int(lines[3+m].strip())
    line = lines[4+m].strip().split(" ")
    for i in range(k):
        x_hat_i = float(line[i])
        x_hat.append(x_hat_i)
```

```
# Fit (compute coefficients) and interpolate the data
with open("output.txt", "w") as f:
    for i in range(m):
        X = x
        Y = y[i]
        spline3 = Spline3()
        spline3.fit(X, Y)
        outputs = []
        for j in range(k):
            X_hat = x_hat[j]
            out = spline3.compute(X_hat)
            outputs.append(out)
        f.write(" ".join([str(o) for o in outputs]))
        f.write("\n")
        print(" ".join([str(o) for o in outputs]))
        if(PLOT):
            plt.scatter(X, Y, label="y", marker="*", s=100, c="g")
            plt.scatter(x_hat, outputs, label="y^hat", marker="+", s=100, c="r")

        if(PLOT):
            function_intrep = spline3.func_out(X[0], X[-1])
            plt.plot(*function_intrep, label="Function - Spline3")
            plt.legend()
            plt.show()
```








```
# Calculation of the error
for i in range(m):
    error = 0
    out = None
    out_correct = None
    with open("output.txt", "r") as output:
        out = output.readlines()[i].strip().split(" ")
    with open("output_correct.txt", "r") as output:
        out_correct = output.readlines()[i].strip().split(" ")
    errors = [float(out[i]) - float(out_correct[i]) for i in range(k)]
    error = sum(errors)
    mean = error/k
    std = sqrt(sum([(e-mean)**2 for e in errors])/k)
    print(f"Error for {i+1}th set = {error}\n {errors}\n {mean}+-{std}")
```

- Error for 1st set = 0.008192203025349065
    - [-0.0016426072569069583, 0.0027929221862741382, -0.002548429505472516, 0.0003396771690166167, -0.00019379194236129882, -0.00014746258937303747, 0.0008440768639047525, -0.0031963105198169472, 0.011944128620084316]

- $\mu + -\sigma = 0.0009102447805943406 + -0.004259654552161143$

- Error for 2nd set = -0.015768264450830224
    - [-0.009091463371454722, -0.0010719874520043193, -0.0018690863513000843, 0.0006948898753855737, -3.0853049121648546e-05, 0.00014710882624158206, -0.0004002448114652779, 0.0015179096192236919, -0.005664537736335018]
    - $\mu + -\sigma = -0.0017520293834255806 + -0.0032442698651418225$
- Error for 3rd set = -2.7755575615628914e-16
    - [0.0, -5.551115123125783e-17, 0.0, 0.0, 0.0, 0.0, 0.0, -1.1102230246251565e-16, -1.1102230246251565e-16]
    - $\mu + -\sigma = -3.0839528461809905e - 17 + -4.6156379789502494e - 17$