Національний технічний університет України «КПІ ім. Ігоря Сікорського»

Факультет Інформатики та Обчислювальної Техніки

Кафедра обчислювальної техніки

Лабораторна робота № 6

з дисципліни «Чисельні методи»

на тему

**“Інтерполяційні поліноми”**

Виконав:

студент гр. ІП-93

Завальнюк Максим

Викладач:

доц. Рибачук Л.В.

Київ – 2021

### Зміст

[Зміст 2](#_Toc67845271)

[1 Постановка задачі 3](#_Toc67845272)

[2 Розв’язок 4](#_Toc67845273)

[3 Розв’язок за допомогою NumPy 9](#_Toc67845274)

[4 Лістинг програми 10](#_Toc67845275)

### 1 Постановка задачі

Створити програму, яка для заданої функції по заданим точкам будує інтерполяційний

поліном (x) у формі Ньютона, а також здійснює інтерполяцію кубічними сплайнами.

Програма має розрахувавати значення похибки ε = .

Знайти кубічний інтерполяційний сплайн для заданої функції у SciPy. Вивести

графік результатів.

### 2 Розв’язок

Допрограмовий етап

Рівняння: 10 \* -3 \* + 7 \* -27 = 0 (1)

Спираючись на наслідок з теореми 1 (про число коренів алгебраїчного рівняння) та теореми 2 (про властивість парної спряженості комплексних коренів рівняння), що алгебраїчне рівняння непарного степеню має хоча б один дійсний корінь, можемо зробити висновок, про існування хоча б одного дійсного кореня у нашого рівняння (1), так як воно має степінь 5.

За допомогою теореми 3 (про оцінку модулів коренів рівняння) та наслідку з неї, про визначення верхніх та нижніх меж додатних та від’ємних коренів визначимо дані межі.

A = 27, B = 10, = 27, = 10

r = = = 0.729

R = 1 + = 1 + = 3.7

0.729 < < 3.7, i = 1, 2, …, n

Або за наслідком можемо розкрити модулі і отримати відповідно межі додатних та від’ємних коренів:

0.729 < < 3.7, i = 1, 2, …, n

-3.7 < < 0.729, i = 1, 2, …, n

Встановимо більш точні межі дійсних коренів для даного рівняння, використовуючи теорему 4 (теорема Лагранжа про верхню межу додатних коренів рівняння) та теорему 5 (про нижню і верхню межі додатних та від’ємних коренів алгебраїчного рівняння).

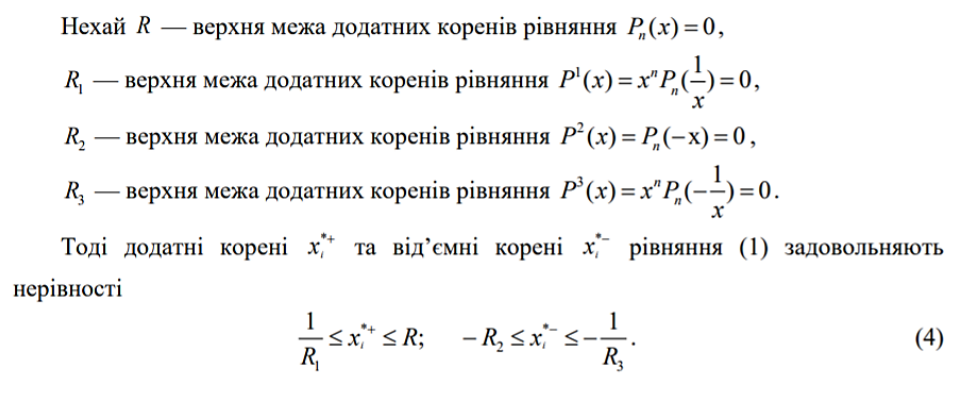
У нашому випадку рівняння вже задовольняє умови використання для теореми 4, = = 10 > 0, знайдемо − перший від’ємний коефіцієнт в послідовності , …, та C − найбільше з абсолютних значень від’ємних коефіцієнтів рівняння.

i = 4, C = 27

Тоді за верхню межу додатних коренів рівняння може бути прийняте число:

R = 1 + = 1 + = 2.64317

Далі за допомогою теореми 5 визначимо нижню межу додатних коренів, та обидві межі для від’ємних шляхом використання теореми 4 для наступних рівнянь:

Побудуємо рівняння = () = 0 отримаємо

(10) = 0, 10 -3x + 7 - 27 = 0 або 27 - 7 +3x -10 = 0

= 27, i = 2, C = 10

= 1 + = 1 + = 1.718145

Побудуємо рівняння = (-x) = 0 отримаємо

10 – 3 + 7 – 27 = 0 або -10 – 3 + 7 – 27 = 0 або 10 + 3 - 7 + 27 = 0

= 10, i = 3, C = 7

= 1 + = 1 + = 1.83666

Побудуємо рівняння = (-) = 0 отримаємо

(10) = 0, -10 -3x + 7 - 27 = 0 або 27 - 7 +3x +10 = 0

= 27, i = 2, C = 7

= 1 + = 1 + = 1.637644

Маємо такі межі для додатних коренів:

≤ ≤ R, ≤ ≤ 2.64317, 0.582023 ≤ ≤ 2. 64317

Маємо такі межі для від’ємних коренів:

- ≤ ≤ -, -1.83666 ≤ ≤ -0.610633

Визначимо кількість дійсних додатних та від’ємних коренів за допомогою теореми 6 (теорема Декарта про кількість дійсних коренів алгебраїчних рівнянь).

Число змін знаків у рівнянні = 0 рівне = 3, тобто кількість додатних коренів рівняння (1) рівна 3-м або менше від нього на парне число, тобто рівна 1.

Число змін знаків у рівнянні = 0 рівне = 2, тобто кількість від’ємних коренів рівняння (1) рівна 2-м або менше від нього на парне число, тобто рівна 0 (від’ємних дійсних коренів многочлен не має).

Скористаємося теоремою 7 (теорема Гюа про необхідну умову дійсності всіх коренів алгебраїчного рівняння) Взявши за k = 1, отримаємо:

9 > 10 \* 0. Але Теорема Гюа є лише необхідною умовою дійсності всіх коренів алгебраїчного рівняння.

### 3 Розв’язок за допомогою SciPy

Нижче наведено розв’язок системи у SciPy:

cs = CubicSpline(x\_values.copy(), y\_values.copy())

### 4 Лістинг програми

from scipy.interpolate import CubicSpline  
import numpy as np  
import matplotlib.pyplot as plt  
from string import Template  
from math import sin  
  
template = Template(**'#'** \* 10 + **' $string '** + **'#'** \* 10)  
  
  
def linear\_function(x: int, alpha=3) -> float:  
 *"""  
 Main linear function  
 :param x: x value  
 :param alpha: some value  
 :return: result y = f(x)  
 """* y\_value = sin(alpha / 2 \* x) + (x \* alpha) \*\* (1 / 3)  
 return y\_value  
  
  
def create\_indexes(x\_values: list) -> dict:  
 *"""  
 Function that creates indexes for spline coefficients matrix  
 :param x\_values: our nodes  
 :return: dictionary with indexes  
 """* indexes = {}  
 length = len(x\_values)  
 for i in range(length - 1):  
 indexes[**f'b**{i + 1}**'**] = i  
 indexes[**f'c**{i + 1}**'**] = i + length - 1  
 indexes[**f'd**{i + 1}**'**] = i + length \* 2 - 2  
 indexes[**'y'**] = (length - 1) \* 3  
 return indexes  
  
  
def get\_coeffs\_for\_newton\_polynomial(x\_elements: list, y\_elements: list) -> list:  
 *"""  
 Creates pyramid and extracts coefficients  
 :param x\_elements: our nodes  
 :param y\_elements: results of f(x)  
 :return: list of coefficients  
 """* length = len(y\_elements)  
 pyramid = []  
 for \_ in range(length):  
 tmp\_array = []  
 for \_ in range(length):  
 tmp\_array.append(0)  
 pyramid.append(tmp\_array)  
 print(template.substitute(string=**'Zero pyramid'**))  
 print(np.matrix(pyramid))  
 for index in range(length):  
 pyramid[index][0] = y\_elements[index]  
 print(template.substitute(string=**'Pyramid with first y\_elements'**))  
 print(np.matrix(pyramid))  
 for step in range(1, length):  
 for index in range(length - step):  
 pyramid[index][step] = (pyramid[index + 1][step - 1] - pyramid[index][step - 1]) / (  
 x\_elements[index + step] - x\_elements[index])  
 print(template.substitute(string=**'Final pyramid'**))  
 print(np.matrix(pyramid))  
 return pyramid[0] *# return first row*def print\_newton\_polynomial(x\_values: list, newton\_coeffs: list) -> None:  
 *"""  
 Function that prints our newton polynomial  
 :param x\_values: our nodes  
 :param newton\_coeffs: coefficients for newton polynomial  
 :return: nothing to return  
 """* print(template.substitute(string=**'Newton eval'**))  
 for i in range(len(newton\_coeffs)):  
 print(**f' +(**{round(newton\_coeffs[i], 5)}**) '**, end=**''**)  
 for j in range(i):  
 print(**f'(x-**{x\_values[j]}**)'**, end=**''**)  
 print(**' = y'**, end=**'**\n**'**)  
  
 *# Create polynomial with NumPy* final\_polynomial = np.polynomial.Polynomial([0.]) *# our target polynomial* length = len(newton\_coeffs)  
 for i in range(length):  
 polynomial = np.polynomial.Polynomial([1.]) *# create a dummy polynomial* for j in range(i):  
 p\_temp = np.polynomial.Polynomial([-x\_values[j], 1.]) *# (x\_elements - x\_j)* polynomial = np.polymul(polynomial, p\_temp) *# multiply dummy with expression* polynomial \*= newton\_coeffs[i] *# apply coefficient* final\_polynomial = np.polyadd(final\_polynomial, polynomial) *# add to target polynomial* final\_polynomial[0].coef = np.round\_(final\_polynomial[0].coef, decimals=5)  
 print(final\_polynomial[0])  
  
  
def solve\_newton\_polynomial(newton\_coeffs: list, x\_values: list, x\_value: float) -> float:  
 *"""  
 Function that calculate newton polynomial at x\_valuew  
 :param newton\_coeffs: coefficients for newton polynomial  
 :param x\_values: our nodes  
 :param x\_value: current x  
 :return: f(x)  
 """* length = len(x\_values) - 1  
 result = newton\_coeffs[length]  
 for k in range(1, length + 1):  
 result = newton\_coeffs[length - k] + (x\_value - x\_values[length - k]) \* result  
 return result  
  
  
def show\_plot(x\_values: list, y\_values: list, newton\_coeffs=None, spline\_coeffs=None, indexes=None, cubic\_spline=None) -> None:  
 *"""  
 Function for creating plots  
 :param x\_values: our nodes  
 :param y\_values: values at this nodes  
 :param newton\_coeffs: coefficients for newton polynomial  
 :param spline\_coeffs: coefficients for spline equations  
 :param indexes: dictionary with indexes for spline equations  
 :param cubic\_spline: CubicSpline class from SciPy  
 :return: nothing to return  
 """* x\_axis = np.linspace(4, 12, num=10000)  
 x\_axis\_2 = np.linspace(4, 12, num=2000)  
 fig, ax = plt.subplots()  
 ax.plot(x\_values, y\_values, **'o'**, label=**'Data'**)  
 ax.plot(x\_axis, [linear\_function(x) for x in x\_axis], label=**'Linear'**)  
 if cubic\_spline is not None:  
 ax.plot(x\_axis\_2, [solve\_spline\_equation(x\_values, y\_values, x, spline\_coeffs, indexes) for x in x\_axis\_2],  
 label=**'Spline interpolation'**)  
 ax.plot(x\_axis\_2, cubic\_spline(x\_axis\_2), label=**'SciPy spline interpolation'**)  
 elif newton\_coeffs is not None:  
 ax.plot(x\_axis\_2, [solve\_newton\_polynomial(newton\_coeffs, x\_values, x) for x in x\_axis\_2],  
 label=**'Newton Polynomial'**)  
 elif spline\_coeffs is not None:  
 ax.plot(x\_axis\_2, [solve\_spline\_equation(x\_values, y\_values, x, spline\_coeffs, indexes) for x in x\_axis\_2],  
 label=**'Spline interpolation'**)  
 ax.legend(loc=**'lower left'**, ncol=2)  
 plt.grid()  
 plt.show()  
  
  
def create\_matrix(x\_values: list, y\_values: list, indexes: dict) -> [list, list]:  
 *"""  
 Function that create matrix to find coefficients for spline equations  
 :param x\_values: our nodes  
 :param y\_values: values at this nodes  
 :param indexes: indexes for matrix  
 :return: matrix a and vector b  
 """* matrix\_a = []  
 indexes\_length = len(indexes)  
 *# I* for i in range(1, len(x\_values)):  
 row = np.zeros(indexes\_length)  
 h = x\_values[i] - x\_values[i - 1]  
 row[indexes[**f'b**{i}**'**]] = h  
 row[indexes[**f'c**{i}**'**]] = h \*\* 2  
 row[indexes[**f'd**{i}**'**]] = h \*\* 3  
 row[indexes[**'y'**]] = y\_values[i] - y\_values[i - 1]  
 matrix\_a.append(row)  
 *# II* for i in range(1, len(x\_values) - 1):  
 row = np.zeros(indexes\_length)  
 h = x\_values[i] - x\_values[i - 1]  
 row[indexes[**f'b**{i + 1}**'**]] = 1  
 row[indexes[**f'b**{i}**'**]] = -1  
 row[indexes[**f'c**{i}**'**]] = -2 \* h  
 row[indexes[**f'd**{i}**'**]] = -3 \* h \*\* 2  
 row[indexes[**'y'**]] = 0  
 matrix\_a.append(row)  
 *# III* for i in range(1, len(x\_values) - 1):  
 row = np.zeros(indexes\_length)  
 h = x\_values[i] - x\_values[i - 1]  
 row[indexes[**f'c**{i + 1}**'**]] = 1  
 row[indexes[**f'c**{i}**'**]] = -1  
 row[indexes[**f'd**{i}**'**]] = -3 \* h  
 row[indexes[**'y'**]] = 0  
 matrix\_a.append(row)  
 *# IV* row = np.zeros(indexes\_length)  
 row[indexes[**f'c**{len(x\_values) - 1}**'**]] = 1  
 row[indexes[**f'd**{len(x\_values) - 1}**'**]] = 3 \* (x\_values[-1] - x\_values[-2])  
 row[indexes[**'y'**]] = 0  
 matrix\_a.append(row)  
 row = np.zeros(indexes\_length)  
 row[indexes[**'c1'**]] = 1  
 row[indexes[**'y'**]] = 0  
 matrix\_a.append(row)  
 vector\_b = np.zeros(indexes\_length - 1)  
 for i in range(len(matrix\_a)):  
 vector\_b[i] = matrix\_a[i][-1]  
 matrix\_a = np.delete(matrix\_a, np.s\_[-1:], axis=1)  
 print(template.substitute(string=**'Matrix A and vector B'**))  
 print(np.matrix(matrix\_a))  
 print(vector\_b)  
 return matrix\_a, vector\_b  
  
  
def solve\_kramer\_method(matrix\_a: list, vector\_b: list, matrix\_c: list) -> list:  
 *"""  
 Kramer function to find spline coeffiecents  
 :param matrix\_a: matrix a  
 :param vector\_b: vector b  
 :param matrix\_c: matrix a copy  
 :return: list of spline coefficients  
 """* spline\_coeffs = []  
 for i in range(0, len(vector\_b)):  
 for j in range(0, len(vector\_b)):  
 matrix\_c[j][i] = vector\_b[j]  
 if i > 0:  
 matrix\_c[j][i - 1] = matrix\_a[j][i - 1]  
 spline\_coeffs.append(np.linalg.det(matrix\_c) / np.linalg.det(matrix\_a))  
 spline\_coeffs = np.array(spline\_coeffs).round(5)  
 return spline\_coeffs  
  
  
def print\_spline\_equations(x\_values: list, y\_values: list, spline\_coeffs: list, indexes: dict) -> None:  
 *"""  
 Function that print spline equations  
 :param x\_values: our nodes  
 :param y\_values: values at this nodes  
 :param spline\_coeffs: coefficients for spline equations  
 :param indexes: indexes for spline equations  
 :return: nothing to return  
 """* print(template.substitute(string=**'Spline equations'**))  
 for i in range(len(x\_values) - 1):  
 print(  
 **f"**{y\_values[i]} **+(**{spline\_coeffs[indexes[**f'b**{i + 1}**'**]]}**)(x-**{x\_values[i]}**) +**{spline\_coeffs[indexes[**f'c**{i + 1}**'**]]}**(x-**{x\_values[i]}**)\*\*2 +**{spline\_coeffs[indexes[**f'd**{i + 1}**'**]]}**(x-**{x\_values[i]}**)\*\*3"**)  
  
  
def solve\_spline\_equation(x\_values: list, y\_values: list, x\_value: float, spline\_coeffs: list, indexes: dict) -> float:  
 *"""  
 Function to get value at x point of spline equation  
 :param x\_values: our nodes  
 :param y\_values: values at this nodes  
 :param x\_value: current x point  
 :param spline\_coeffs: coefficients for spline equations  
 :param indexes: dictionary with indexes for spline equations  
 :return: f(x)  
 """* for i in range(len(x\_values) - 1):  
 if x\_values[i] <= x\_value <= x\_values[i + 1]:  
 return y\_values[i] + spline\_coeffs[indexes[**f'b**{i + 1}**'**]] \* (x\_value - x\_values[i]) + spline\_coeffs[  
 indexes[**f'c**{i + 1}**'**]] \* (x\_value - x\_values[i]) \*\* 2 + spline\_coeffs[indexes[**f'd**{i + 1}**'**]] \* (  
 x\_value - x\_values[i]) \*\* 3  
  
  
def get\_faults(x\_values: list, y\_values: list, newton\_coeffs: list, spline\_coeffs: list, indexes: dict, cubic\_spline: CubicSpline) -> None:  
 *"""  
 Function that prints faults for math functions  
 :param x\_values: our nodes  
 :param y\_values: values at this nodes  
 :param newton\_coeffs: coefficients for newton polynomial  
 :param spline\_coeffs: coefficients for spline equations  
 :param indexes: dictionary with indexes for spline equations  
 :param cubic\_spline: CubicSpline class from SciPy  
 :return: nothing to return  
 """* x\_axis = np.arange(4, 12.5, 0.5)  
 print(template.substitute(string=**'Faults'**))  
 for x\_value in x\_axis:  
 print(**'X value:'**, x\_value)  
 print(**'Newton polynomial'**, round(abs(solve\_newton\_polynomial(newton\_coeffs, x\_values, x\_value) - linear\_function(x\_value)), 5))  
 print(**'Cubic spline'**, round(abs(solve\_spline\_equation(x\_values, y\_values, x\_value, spline\_coeffs, indexes) - linear\_function(x\_value)), 4))  
 print(**'SciPy cubic spline'**, round(abs(cubic\_spline(x\_value) - linear\_function(x\_value)), 5))  
 print()  
  
  
def main():  
 *"""Main function"""* k = 10 - 1  
 x\_values = [-5 + k, -3 + k, -1 + k, 1 + k, 3 + k]  
 y\_values = [linear\_function(x) for x in x\_values]  
 print(template.substitute(string=**'X and Y values'**))  
 print(x\_values)  
 print(np.array(y\_values).round(5))  
 show\_plot(x\_values=x\_values.copy(), y\_values=y\_values.copy())  
 *# Newton Polynomial* newton\_coeffs = get\_coeffs\_for\_newton\_polynomial(x\_values.copy(), y\_values.copy())  
 print(template.substitute(string=**'Coefficients'**))  
 print(np.array(newton\_coeffs).round(5))  
 print\_newton\_polynomial(x\_values.copy(), newton\_coeffs.copy())  
 show\_plot(x\_values=x\_values.copy(), y\_values=y\_values.copy(), newton\_coeffs=newton\_coeffs.copy())  
 *# Cubic spline  
 # x\_values\_2 = [2, 3, 5, 7]  
 # y\_values\_2 = [4, -2, 6, -3]* indexes = create\_indexes(x\_values.copy())  
 print(template.substitute(string=**'Indexes'**))  
 print(indexes)  
 matrix\_a, vector\_b = create\_matrix(x\_values.copy(), y\_values.copy(), indexes.copy())  
 matrix\_c = matrix\_a.copy()  
 spline\_coeffs = solve\_kramer\_method(matrix\_a, vector\_b, matrix\_c)  
 print(template.substitute(string=**'Spline coeffiecients'**))  
 print(spline\_coeffs)  
 print\_spline\_equations(x\_values.copy(), y\_values.copy(), spline\_coeffs.copy(), indexes.copy())  
 show\_plot(x\_values=x\_values.copy(), y\_values=y\_values.copy(), spline\_coeffs=spline\_coeffs.copy(),  
 indexes=indexes.copy())  
 *# SciPy spline* cs = CubicSpline(x\_values.copy(), y\_values.copy())  
 show\_plot(x\_values=x\_values.copy(), y\_values=y\_values.copy(), spline\_coeffs=spline\_coeffs.copy(), indexes=indexes.copy(), cubic\_spline=cs)  
 *# Get faults* get\_faults(x\_values.copy(), y\_values.copy(), newton\_coeffs.copy(), spline\_coeffs.copy(), indexes.copy(), cs)  
  
  
main()