Національний технічний університет України «КПІ ім. Ігоря Сікорського»
Факультет Інформатики та Обчислювальної Техніки
Кафедра обчислювальної техніки

Лабораторна робота № 5

з дисципліни «Численні методи»

на тему

“Інтерполяційні поліноми”

Виконав:
студент гр. ІП-93
Завальнюк Максим

Викладач:
доц. Рибачук Л.В.

Київ – 2021

Зміст

1 Постановка задачі

Створити програму, яка для заданої функції по заданим точкам будує інтерполяційний поліном P_n(x) у формі Ньютона, а також здійснює інтерполяцію кубічними сплайнами.

Програма має розраховувати значення похибки ε = $|P_n(x) - y(x)|$.

Знайти кубічний інтерполяційний сплайн для заданої функції у SciPy. Вивести графік результатів.

2 Розв’язок

Функція: $\sin\frac{\alpha}{2} * x + \sqrt[3]{x * \alpha}$

Вузли та значення у них:

########## X and Y values ##########
[4, 6, 8, 10, 12]
[2.01001 3.03286 2.34793 3.75752 2.55094]

Поліном у формі Ньютона:

########## Newton eval ##########
+(2.01001) +(0.51142) (x-4) +(-0.21347) (x-4)(x-6) +(0.07921) (x-4)(x-6)(x-8) +(-0.02217) (x-4)(x-6)(x-8)(x-10) = y
-62.93331 + 38.19706 x**1 - 7.93542 x**2 + 0.69996 x**3 - 0.02217 x**4

Коефіцієнти сплайнів:

########## Spline coeffiecients ##########
[0.83831 -0.14234 0.23794 0.27758 -0. -0.49033 0.68047 -0.66065
-0.08172 0.19513 -0.22352 0.11011]

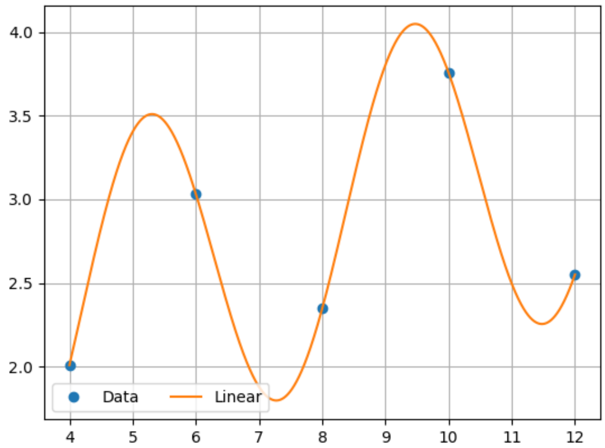
Рівняння сплайнів:

########## Spline equations ##########
2.010012986907738 +(0.83831)(x-4) +-0.0(x-4)**2 +-0.08172(x-4)**3
3.032859879450653 +(-0.14234)(x-6) +-0.49033(x-6)**2 +0.19513(x-6)**3
2.3479262226143818 +(0.23794)(x-8) +0.68047(x-8)**2 +-0.22352(x-8)**3
3.7575203461109754 +(0.27758)(x-10) +-0.66065(x-10)**2 +0.11011(x-10)**3

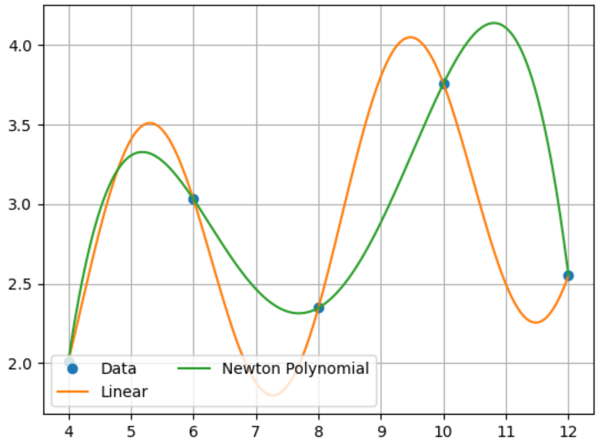
Похибки:

########## Faults ##########
########## Newton interpolation ##########
0.0
########## Cubic spline interpolation ##########
0.0001
########## SciPy cubic spline interpolation ##########
0.0

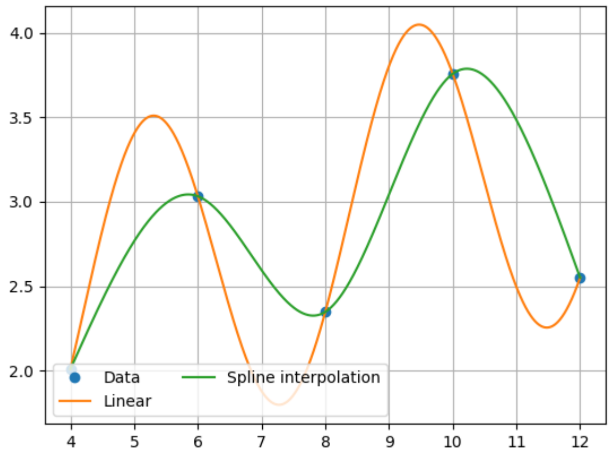
Графіки:



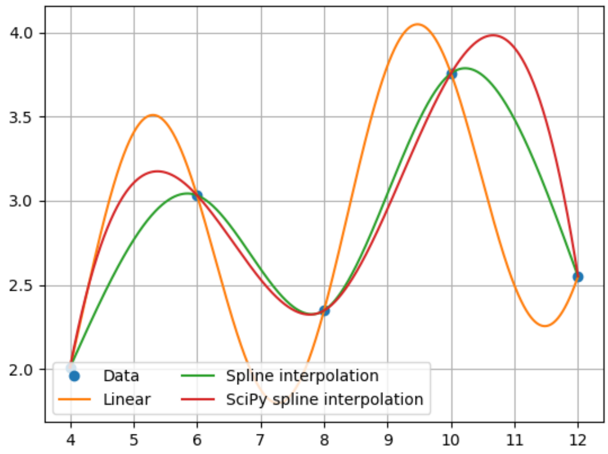
Поліном Ньютона:



Інтерполяція кубічними сплайнами:



Інтерполяція кубічними сплайнами із SciPy:



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
    """
    faults = {'newton': 0., 'spline': 0., 'scipy': 0.}
    for x_value in x_values:
        faults['newton'] += abs(solve_newton_polynomial(newton_coeffs, x_values,
x_value) - linear_function(x_value))
        faults['spline'] += abs(solve_spline_equation(x_values, y_values,
x_value, spline_coeffs, indexes) - linear_function(x_value))
        faults['scipy'] += abs(cubic_spline(x_value) - linear_function(x_value))
    print(template.substitute(string='Faults'))
    print(template.substitute(string='Newton interpolation'))
    print(round(faults['newton'], 5))
    print(template.substitute(string='Cubic spline interpolation'))
    print(round(faults['spline'], 4))
    print(template.substitute(string='SciPy cubic spline interpolation'))
    print(round(faults['scipy'], 5))


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()
```