МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»

Кафедра теоретических основ компьютерной безопасности и криптографии

Интерполяция таблично заданных функций

ОТЧЕТ ПО ДИСЦИПЛИНЕ

«ЧИСЛЕННЫЕ МЕТОДЫ РЕШЕНИЯ ЗАДАЧ»

студента 4 курса 431 группы

специальности 10.05.01 Компьютерная безопасность

факультета компьютерных наук и информационных технологий

Сенокосова Владислава Владимировича

Преподаватель

Аспирант ____________ В.М.Шкатов

подпись, дата

Саратов 2024

Вариант 19:

19	x: 0,234 0,649 1,382 2,672 2,849 y: 0,511 0,982 2,411 3,115 4,184

Задания

1. Реализовать интерполяционный многочлен Лагранжа
2. Найти таблицу конечных разностей
3. Найти таблицу разделенных разностей
4. Реализовать многочлен Ньютона
5. Реализовать систему линейного сплайна
6. Реализовать систему квадратичного сплайна

Задание 1: «Многочлен Лагранжа»

Пусть в точках $x_0, x_1,\dots, x_n$ таких, что $a \le x_0 <\dots< x_n \le b$ известны значения функции $y = f(x)$, то есть на отрезке $[a; b]$ задана табличная (сеточная) функция:

x	x_0	x_1	$\dots$	x_n
y	y_0	y_1	$\dots$	y_n

Определение. Функция $\varphi(x)$ называется интерполирующей (интерполяционной) для $f(x)$ на $[a; b]$, если ее значения $\varphi(x_0)$, $\varphi(x_1)$, ..., $\varphi(x_n)$ в заданных точках $x_0, x_1,\dots, x_n$, называемых узлами интерполяции, совпадают с заданными значениями функции $f(x)$, то есть с $y_0, y_1,\dots, y_n$ соответственно.

Будем строить многочлен n-степени $L_n(x)$ в виде линейной комбинации

$$L_n(x) = \sum_{i=0}^{n} p_i(x) f(x_i), \qquad (5.1)$$

где базисные многочлены имеют вид

$$p_i(x) = \frac{(x-x_0)(x-x_1)\dots(x-x_{i-1})(x-x_{i+1})(x-x_n)}{(x_i-x_0)(x_i-x_1)\dots(x_i-x_{i-1})(x_i-x_{i+1})(x_i-x_n)},$$

обладающий свойством: $L_n(x_i) = f(x_i), i = \overline{0,n}$, (5.2)

если известны значения функции $f(x)$ в точках $x_i, i = \overline{0,n}$.

Функция, реализующая интерполяционный многочлен Лагранжа представлена в следующем виде:

```python
def lagrange_interpol_polinom(system):
    # system => [[x1, x2, x3...], [y1, y2, y3...]]

    # Вычисление базисных полиномов
    x = sp.Symbol("x")
    base_polinoms_lst = []
    for i in range(len(system[0])):
        numerator, denominator = 1, 1
        for j, x_i in enumerate(system[0]):
            if j != i:
                numerator *= (x - x_i)
                denominator *= (system[0][i] - x_i)
        base_polinoms_lst.append(numerator / denominator)

    # Вычисление многочлена лагранжа
    L = 0
    for i, val in enumerate(system[1]):
        L += val * base_polinoms_lst[i]

    return L
```

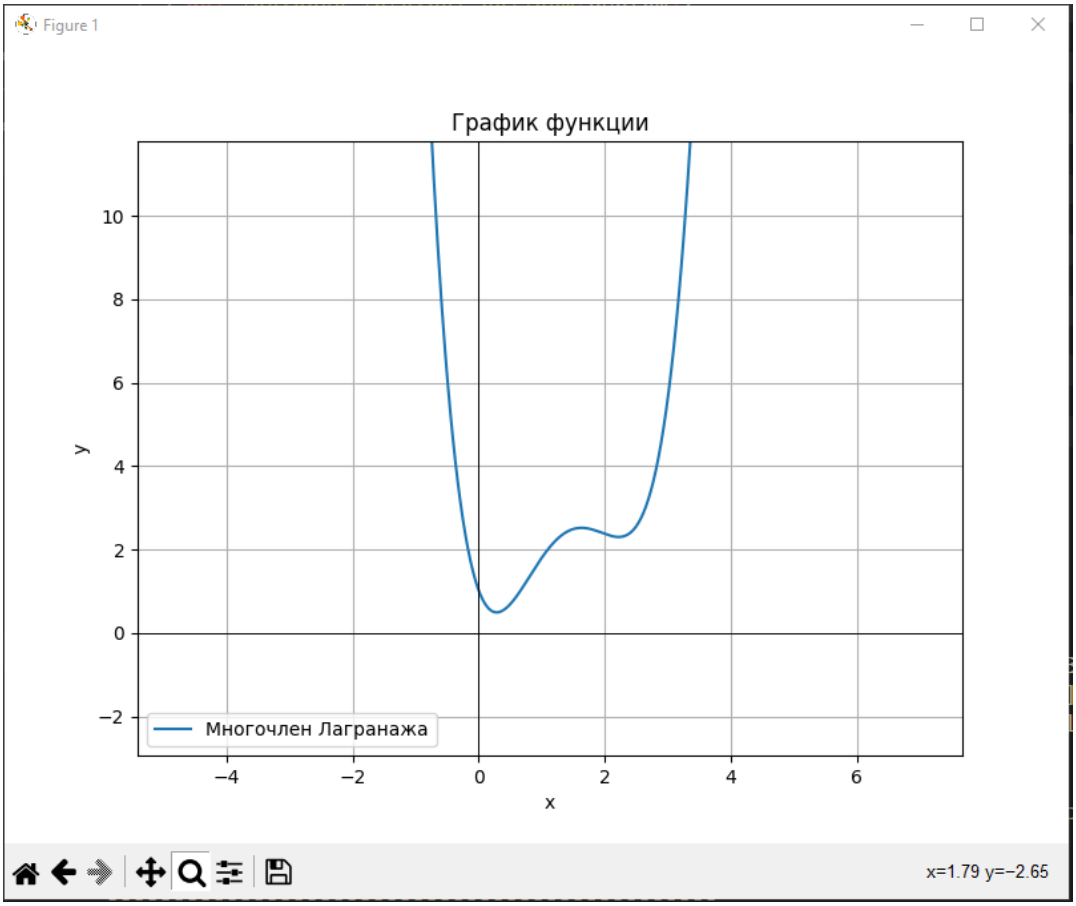
В результате работы получаем следующий многочлен для варианта 19:

$$0.168238551642772*(x - 2.849)*(x - 2.672)*(x - 1.382)*(x - 0.649) - 0.725338705421255*(x - 2.849)*(x - 2.672)*(x - 1.382)*(x - 0.234) + 1.51401969098111*(x - 2.849)*(x - 2.672)*(x - 0.649)*(x - 0.234) - 2.76608480073843*(x - 2.849)*(x - 1.382)*(x - 0.649)*(x - 0.234) + 2.80087625413611*(x - 2.672)*(x - 1.382)*(x - 0.649)*(x - 0.234)$$

В терминале при выводе:

```
Полученный многочлен Лагранжа:
------------------------------------------------
0.168238551642772*(x - 2.849)*(x - 2.672)*(x - 1.382)*(x - 0.649) - 0.725338705421255*(x - 2.849)*(x - 2.6
72)*(x - 1.382)*(x - 0.234) + 1.51401969098111*(x - 2.849)*(x - 2.672)*(x - 0.649)*(x - 0.234) - 2.7660848
0073843*(x - 2.849)*(x - 1.382)*(x - 0.649)*(x - 0.234) + 2.80087625413611*(x - 2.672)*(x - 1.382)*(x - 0.
649)*(x - 0.234)
```

График функции выглядит следующим образом:



Задание 2: «Таблица конечных разностей»

Функция, отвечающая за нахождение конечных разностей представлена следующим образом:

```
def finite_differences(system):
    # Вычисление таблицы конечных разностей

    row_y = deepcopy(system[1])
    arr_rows_y = []
    count_variables = len(row_y)

    for _ in range(count_variables - 1):
        buff_row = []
        for i in range(len(row_y) - 1):
            buff_row.append(round(row_y[i + 1] - row_y[i], 3))
        arr_rows_y.append(buff_row)
        row_y = buff_row

    return arr_rows_y
```

Результат выполнения функции:

```
Полученная таблица конечных разностей:
--------------------------------------------
[[0.471, 1.429, 0.704, 1.069], [0.958, -0.725, 0.365], [-1.683, 1.09], [2.773]]
--------------------------------------------
```

Задание 3: «Таблица разделенных разностей»

Функция, отвечающая за нахождение разделенных разностей представлена следующим образом:

```
def divided_differences(system):

    row_x = deepcopy(system[0])
    row_y = deepcopy(system[1])
    arr_rows_x = []
    for j in range(len(system[0]) - 1):
        buff_row = []
        for i in range(len(row_y) - 1):
            buff_row.append(round((row_y[i + 1] - row_y[i]) / (row_x[i + 1
+ j] - row_x[i]), 3))
        arr_rows_x.append(buff_row)
        row_y = buff_row

    return arr_rows_x
```

Результат выполнения функции:

```
Полученная таблица разделенных разностей:
--------------------------------------------
[[1.135, 1.95, 0.546, 6.04], [0.71, -0.694, 3.745], [-0.576, 2.018], [0.992]]
--------------------------------------------
```

Задание 4: «Многочлен Ньютона»

Пусть интерполируемая функция $y = f(x)$ задана таблично значениями $y_0, y_1, \ldots, y_n$ на системе равностоящих узлов $x_0, x_1, \ldots, x_n$: $\forall x_k$ можно представить в виде $x_k = x_0 + kh$, $k = \overline{0,n}$, $h > 0$, $f_k = f(x_k)$, h – шаг сетки.

Определение. Конечной разностью 1-го порядка называется

$\Delta^1 f_k = f_{k+1} - f_k \ (\Delta^0 f_k = f_k)$.

Конечная разность n-порядка:

$\Delta^n f_k = \Delta^{n-1} f_{k+1} - \Delta^{n-1} f_k$.

Свойства:

1. $\Delta^n P_n(x) = const$ (конечная разность n-го порядка от полинома n-й степени равно константе).

$\Delta^{(n+1)} P_n(x) = 0$ (конечная разность $(n+1)$-го порядка от полинома n-го порядка равна нулю).

2. Пусть $f(x)$ имеет все производные, тогда $\Delta^n f_k \approx f^{(n)}(x_k)h^n$.

Непосредственно через значения функции конечные разности можно представить рекуррентной формулой $\Delta^n f_k = \sum_{i=0}^{n}(-1)^i (C_n^i) f_{n+k-i}$.

Пусть $f(x)$ задана таблично и $x_k = x_0 + kh$, $k = \overline{0,n}$, $f_k = f(x_k)$.

Определение. Разделенной разностью $f(x_0,...,x_n)$ n-го порядка называется:

$$f(x_0,x_1,...,x_n) = \frac{f(x_1,x_2,...,x_n) - f(x_0,x_1,...,x_{n-1})}{x_n - x_0}.$$

Разделенная разность первого порядка: $f(x_0,x_1) = \dfrac{f(x_1) - f(x_0)}{x_1 - x_0}$.

Разделенная разность второго порядка:

$$f(x_0,x_1,x_2) = \frac{f(x_1,x_2) - f(x_0,x_1)}{x_2 - x_0}.$$

Функция, результатом которой является полином Ньютона, представлена следующим образом:

```
def newtown_polinom(system):
    # Вычисление многочлена ньютона
    x = sp.Symbol("x")

    # Выбираем начальное значение функции => y0
    polinom_newton = system[1][0]

    # Получаем таблицу разделенных разностей
    arr_rows_x = divided_differences(system)

    for i in range(len(system[0]) - 1):
        koef = arr_rows_x[i][0]
        j = 0
        while i >= 0:
            koef *= (x - system[0][j])
            i -= 1
            j += 1
```

```
    polinom_newton += koef

# Вычисление значение полинома в точке x1 + x2
x1 = system[0][1]
x2 = system[0][2]

val_func = polinom_newton.subs({x: x1 + x2})

return polinom_newton, val_func
```

В результате работы получаем следующий многочлен для варианта 19:

1.135*x + (0.134784 - 0.576*x)*(x - 1.382)*(x - 0.649) +

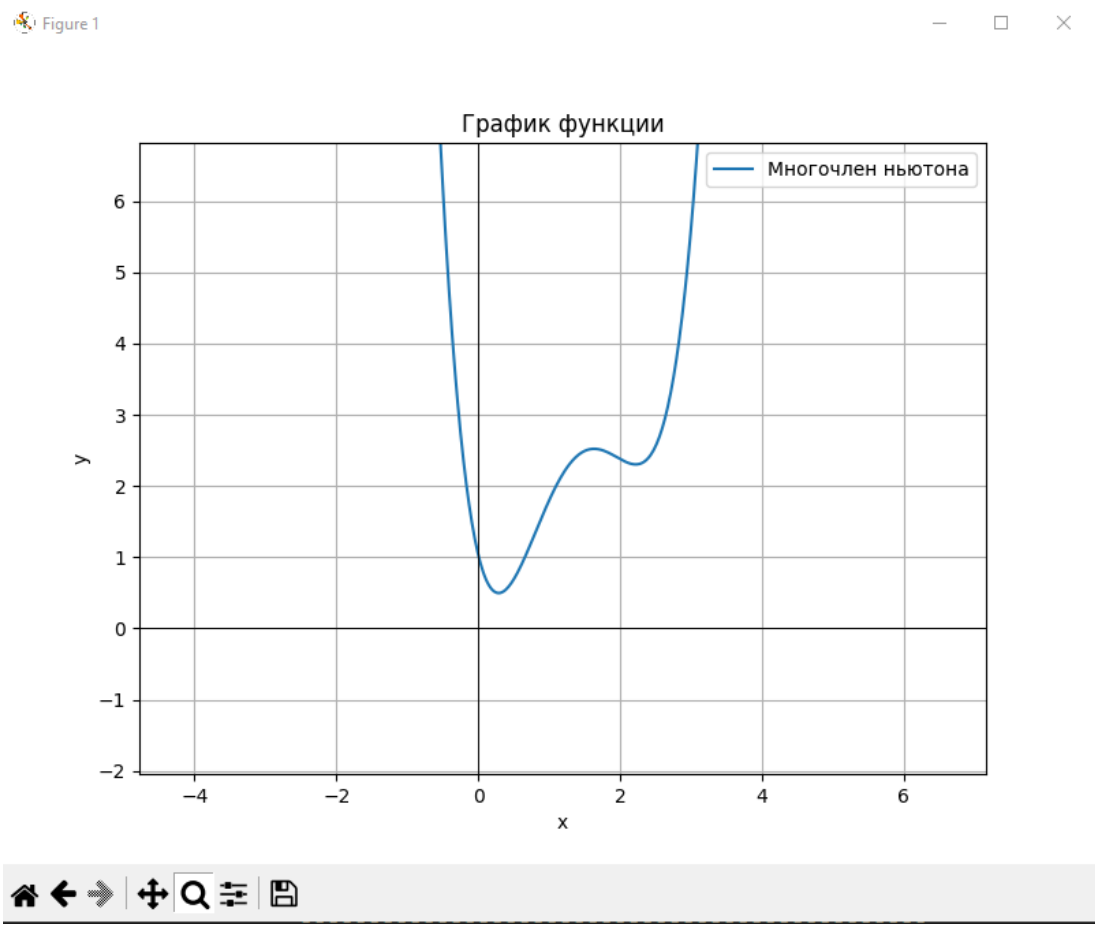
(0.71*x - 0.16614)*(x - 0.649) +

(0.992*x - 0.232128)*(x - 2.672)*(x - 1.382)*(x - 0.649) + 0.24541

В терминале при выводе:

```
Полученный многочлен ньютона:
1.135*x + (0.134784 - 0.576*x)*(x - 1.382)*(x - 0.649) + (0.71*x - 0.16614)*(x - 0.649) + (0.992*x - 0.232
128)*(x - 2.672)*(x - 1.382)*(x - 0.649) + 0.24541
Вычисление значения полинома в точке x1 + x2:
2.36059853053869
```

График выглядит следующим образом:



Задание 5: «Система уравнений линейного сплайна»

Пусть задана функция $y = f(x)$ таблично x_i, y_i $(i = \overline{0,n})$ $a \le x_0 \le x_1 \le \ldots \le x_n \le b$.

Требуется аппроксимировать функцию $f(x)$ кусочно-линейной функции $\varphi(x)$, исходя из условий интерполяций, т. е.

$$\varphi(x) = \begin{cases} a_1 x + b_1, x_0 \le x \le x_1 \\ a_2 x + b_2, x_1 \le x \le x_2 \\ \ldots \\ a_n x + b_n, x_{n-1} \le x \le x_n \end{cases}.$$

Для нахождения неизвестных параметров a_k, b_k $(k = \overline{1,n})$, получим систему уравнений:

$$\begin{cases} \begin{cases} a_1 x_0 + b_1 = y_0 \\ a_1 x_1 + b_1 = y_1 \end{cases} \\ \ldots \\ \begin{cases} a_n x_{n-1} + b_n = y_{n-1} \\ a_n x_n + b_n = y_n \end{cases} \end{cases}.$$

Каждая из n подсистем решается отдельно.

Функция, реализующая построение системы уравнений линейного сплайна представлена следующим образом:

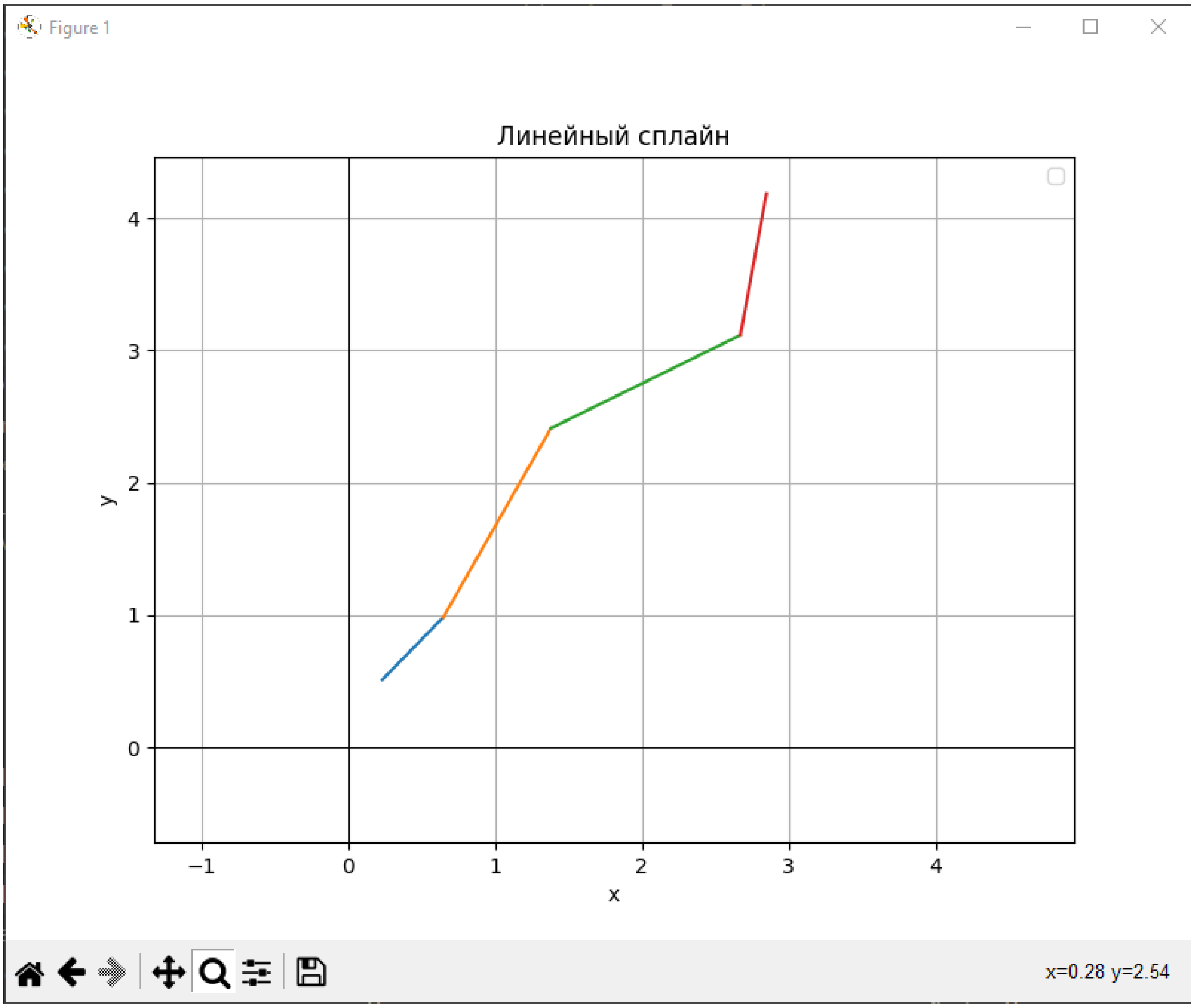
```
def build_linear_spline(system):
    # Построим интерполяционные сплайн линейный
    x = sp.Symbol("x")

    # Определяем количество параметров
    params = []
    for i in range(len(system[0]) - 1):
        params_for_system = []
        params_for_system.append(sp.Symbol(f"a{i + 1}"))
        params_for_system.append(sp.Symbol(f"b{i + 1}"))
        params.append(params_for_system)

    # Составляем системы
    systems = []
    j, i = 1, 0
    for _ in range(len(params)):
        buff_sys = []
```

```
        j -= 1
        for _ in range(len(params[0])):
            buff_sys.append(system[0][j] * params[i][0] + params[i][1] -
system[1][j])
            j += 1
        i += 1
        systems.append(buff_sys)

    # Находим корни систем
    roots = []

    # Берем произвольные стартовые приближения
    intervals = [10 for _ in range(len(params[0]))]

    for i in range(len(systems)):
        root, _ = method_newton(params[i], systems[i], intervals)
        roots.append(list(root))

    # Находим итоговую систему линейного сплайна и диапозоны для каждой из
строк системы

    linear_spline = []

    for pair_root in roots:
        linear_spline.append(pair_root[0] * x + pair_root[1])

    # Определяем диапозоны
    diaposon = []
    i = 1
    for _ in range(len(system[0]) - 1):
        buff = []
        i -= 1
        for _ in range(2):
            buff.append(system[0][i])
            i += 1
        diaposon.append(buff)

    # Объединяем наши уравнения и диапозоны в кортеж
    # (уравнение, диапозон принимаемых значений x)

    res_system = []
    for i in range(len(diaposon)):
        res_system.append((linear_spline[i], diaposon[i]))

    return res_system
```

В качестве системы уравнений выступает массив, в котором каждый элемент представлен парой, где первый элемент – это уравнение, а второй элемент – диапазон значений аргумента. В результате выполнения функции получаем следующий вывод:

```
Система линейного сплайна и значения параметров:
---------------------------------------------
[(1.13493975903614*x + 0.245424096385542, [0.234, 0.649]), (1.94952251023192*x - 0.283240109140519, [0.649
, 1.382]), (0.545736434108527*x + 1.65679224806202, [1.382, 2.672]), (6.03954802259886*x - 13.022672316384
2, [2.672, 2.849])]
```

Полученная система уравнений имеет следующий график:



Можно заметить, что эта функция приближенно напоминает исходную функцию.

Задание 6: «Система уравнений квадратичного сплайна»

Кусочно-квадратичная аппроксимация осуществляется аналогично кусочно-линейной аппроксимации. Каждое звено кусочно-квадратичной функции при $n = 2m$

$$\varphi(x) = \begin{cases} a_1x^2 + b_1x + c_1, x \in [x_0, x_2] \\ a_2x^2 + b_2x + c_2, x \in [x_2, x_4] \\ \ldots \\ a_nx^2 + b_nx + c_n, x \in [x_{2m-2}, x_{2m}] \end{cases}.$$

Тройка коэффициентов $a_k, b_k, c_k \, (k = \overline{1, m})$ может быть найдена последовательным решением трехмерных линейных систем, соответствующим выставленным интерполяционным условиям.

$$\begin{cases} a_kx_{2k-2}^2 + b_kx_{2k-2} + c_k = y_{2k-2} \\ a_kx_{2k-1}^2 + b_kx_{2k-1} + c_k = y_{2k-1} \\ a_kx_{2k}^2 + b_kx_{2k} + c_k = y_{2k} \end{cases}.$$

Функция, реализующая построение системы уравнений квадратичного сплайна представлена следующим образом:

```
def build_kvadro_spline(system):
    # Построим сплайн квадратичный
    x = sp.Symbol("x")

    # Определяем параметры
    params_kvadro = []

    # Определяем количество систем состоящих из 3 уравнений
    count_system = int(len(system[0]) / 3 + 0.99)

    for i in range(count_system):
        params_for_system = []
        params_for_system.append(sp.Symbol(f"a{i + 1}"))
        params_for_system.append(sp.Symbol(f"b{i + 1}"))
        params_for_system.append(sp.Symbol(f"c{i + 1}"))
        params_kvadro.append(params_for_system)

    # Составляем систему уравнений

    kvadro_system = []
    k = 0
    j = 0
    index_system = 0
    count_itr = len(params_kvadro[0])
    buff_row = None
```

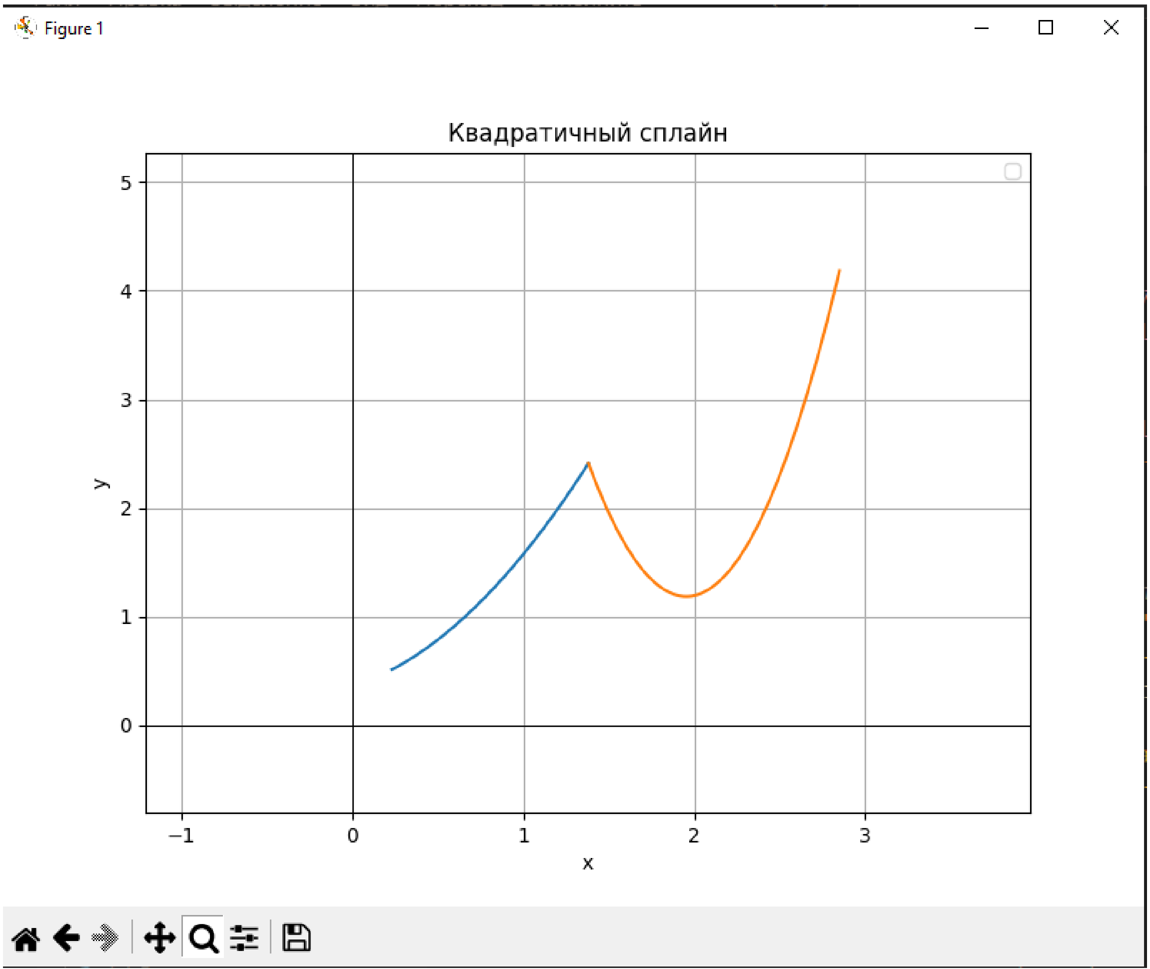
```
    for i in range(count_system):
        if buff_row == None:
            syst = []
        else:
            syst = [buff_row]
        for _ in range(count_itr):
            last_row = pow(system[0][k], 2) *
params_kvadro[index_system][0] \
                        + system[0][k] * params_kvadro[index_system][1] \
                        + params_kvadro[index_system][2] - system[1][j]
            syst.append(last_row)
            k += 1
            j += 1
        index_system += 1
        count_itr -= 1
        kvadro_system.append(syst)
        if index_system < len(params_kvadro):
            buff_row = pow(system[0][k - 1], 2) *
params_kvadro[index_system][0] \
                            + system[0][k - 1] *
params_kvadro[index_system][1] \
                            + params_kvadro[index_system][2] - system[1][j
- 1]

    # Вычисление корней систем

    roots = []
    intervals = [10 for i in range(len(params_kvadro[0]))]
    for i in range(len(params_kvadro)):
        root, _ = method_newton(params_kvadro[i], kvadro_system[i],
intervals)
        roots.append(list(root))

    # Вычисление итоговой системы уравнений

    res_func = []
    for row in roots:
        buff = 0
        for i in range(len(row)):
            buff += row[i] * x ** (2 - i)
        res_func.append(buff)

    # Объединяем с диапозоном значений x

    res = []
    start = 0
    end = len(res_func)
    for i in range(len(res_func)):
        res.append((res_func[i], system[0][start:end + 1]))
        start = end
        end = end + len(res_func)
```

```
    return res
```

В качестве системы уравнений выступает массив, в котором каждый элемент представлен парой, где первый элемент – это уравнение, а второй элемент – диапазон значений аргумента. В результате выполнения функции получаем следующий вывод:

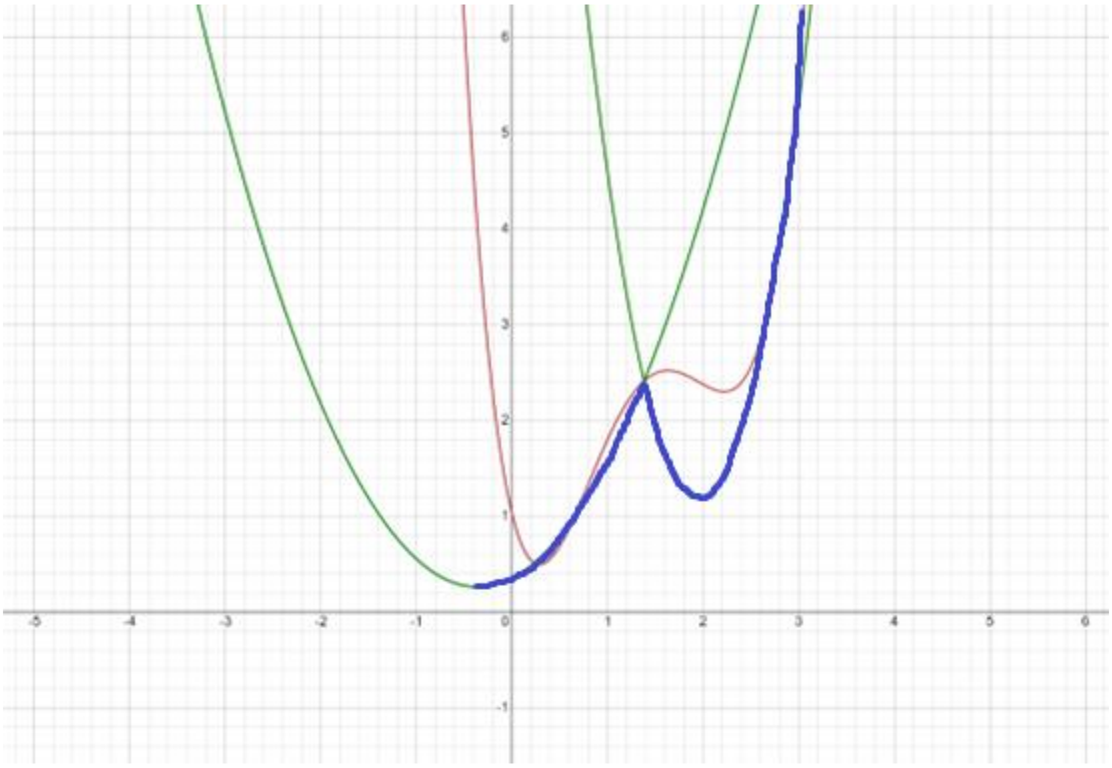
```
Система квадратичного сплайна и значения параметров:
----------------------------------------------
[(0.709566856442315*x**2 + 0.508392224797581*x + 0.353183176606011, [0.234, 0.649, 1.382]), (3.74492950817
338*x**2 - 14.6362077920264*x + 15.4857084226119, [1.382, 2.672, 2.849])]
```

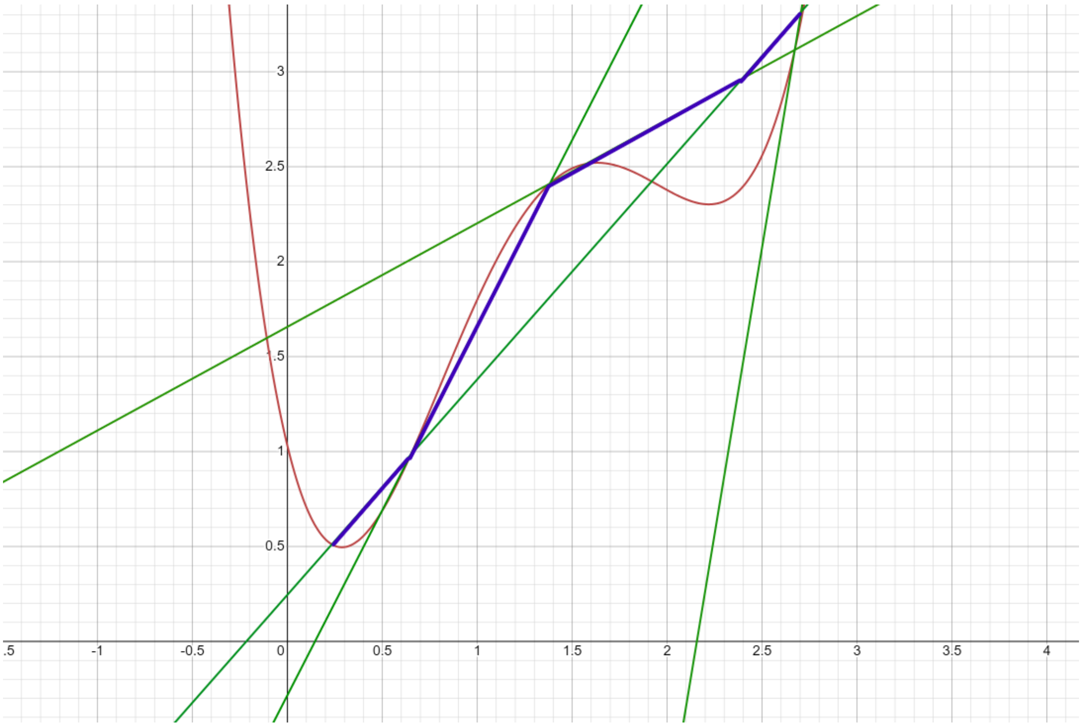
Полученная система уравнений имеет следующий график:



То есть приближение происходит при помощи квадратичных функций.

Отобразим все функции на одном графике. Для простоты анализа было принято решение разделить на два графика. Первый из них демонстрирует исходную функцию (красный) и квадратичный сплайн (зеленый – обведен синем), второй демонстрирует функцию (красный) и линейный сплайн (зеленый – обведен синем):





Выводы:

Эх, вот это я, сука, попал… как пять крон в говно.

-Золтан Хивай

ПРИЛОЖЕНИЕ

Листинг программы

```python
import sympy as sp
import matplotlib.pyplot as plt
import numpy as np
from copy import deepcopy
from lab4 import method_newton

def lagrange_interpol_polinom(system):
    # system => [[x1, x2, x3...], [y1, y2, y3...]]

    # Вычисление базисных полиномов
    x = sp.Symbol("x")
    base_polinoms_lst = []
    for i in range(len(system[0])):
        numerator, denominator = 1, 1
        for j, x_i in enumerate(system[0]):
            if j != i:
                numerator *= (x - x_i)
                denominator *= (system[0][i] - x_i)
        base_polinoms_lst.append(numerator / denominator)

    # Вычисление многочлена лагранжа
    L = 0
    for i, val in enumerate(system[1]):
        L += val * base_polinoms_lst[i]

    return L

def finite_differences(system):
    # Вычисление таблицы конечных разностей

    row_y = deepcopy(system[1])
    arr_rows_y = []
    count_variables = len(row_y)

    for _ in range(count_variables - 1):
        buff_row = []
        for i in range(len(row_y) - 1):
            buff_row.append(round(row_y[i + 1] - row_y[i], 3))
        arr_rows_y.append(buff_row)
        row_y = buff_row

    return arr_rows_y

def divided_differences(system):

    row_x = deepcopy(system[0])
    row_y = deepcopy(system[1])
```

```
    arr_rows_x = []
    for j in range(len(system[0]) - 1):
        buff_row = []
        for i in range(len(row_y) - 1):
            buff_row.append(round((row_y[i + 1] - row_y[i]) / (row_x[i + 1
+ j] - row_x[i]), 3))
        arr_rows_x.append(buff_row)
        row_y = buff_row

    return arr_rows_x

def newtown_polinom(system):
    # Вычисление многочлена ньютона
    x = sp.Symbol("x")

    # Выбираем начальное значение функции => y0
    polinom_newton = system[1][0]

    # Получаем таблицу разделенных разностей
    arr_rows_x = divided_differences(system)

    for i in range(len(system[0]) - 1):
        koef = arr_rows_x[i][0]
        j = 0
        while i >= 0:
            koef *= (x - system[0][j])
            i -= 1
            j += 1
        polinom_newton += koef

    # Вычисление значение полинома в точке x1 + x2
    x1 = system[0][1]
    x2 = system[0][2]

    val_func = polinom_newton.subs({x: x1 + x2})

    return polinom_newton, val_func

def build_linear_spline(system):
    # Построим интерполяционные сплайн линейный
    x = sp.Symbol("x")

    # Определяем количество параметров
    params = []
    for i in range(len(system[0]) - 1):
        params_for_system = []
        params_for_system.append(sp.Symbol(f"a{i + 1}"))
        params_for_system.append(sp.Symbol(f"b{i + 1}"))
        params.append(params_for_system)

    # Составляем системы
```

```python
    systems = []
    j, i = 1, 0
    for _ in range(len(params)):
        buff_sys = []
        j -= 1
        for _ in range(len(params[0])):
            buff_sys.append(system[0][j] * params[i][0] + params[i][1] -
system[1][j])
            j += 1
        i += 1
        systems.append(buff_sys)

    # Находим корни систем
    roots = []

    # Берем произвольные стартовые приближения
    intervals = [10 for _ in range(len(params[0]))]

    for i in range(len(systems)):
        root, _ = method_newton(params[i], systems[i], intervals)
        roots.append(list(root))

    # Находим итоговую систему линейного сплайна и диапозоны для каждой из
строк системы

    linear_spline = []

    for pair_root in roots:
        linear_spline.append(pair_root[0] * x + pair_root[1])

    # Определяем диапозоны
    diaposon = []
    i = 1
    for _ in range(len(system[0]) - 1):
        buff = []
        i -= 1
        for _ in range(2):
            buff.append(system[0][i])
            i += 1
        diaposon.append(buff)

    # Объединяем наши уравнения и диапозоны в кортеж
    # (уравнение, диапозон принимаемых значений x)

    res_system = []
    for i in range(len(diaposon)):
        res_system.append((linear_spline[i], diaposon[i]))

    return res_system

def build_kvadro_spline(system):
```

```python
    # Построим сплайн квадратичный
    x = sp.Symbol("x")

    # Определяем параметры
    params_kvadro = []

    # Определяем количество систем состоящих из 3 уравнений
    count_system = int(len(system[0]) / 3 + 0.99)

    for i in range(count_system):
        params_for_system = []
        params_for_system.append(sp.Symbol(f"a{i + 1}"))
        params_for_system.append(sp.Symbol(f"b{i + 1}"))
        params_for_system.append(sp.Symbol(f"c{i + 1}"))
        params_kvadro.append(params_for_system)

    # Составляем систему уравнений

    kvadro_system = []
    k = 0
    j = 0
    index_system = 0
    count_itr = len(params_kvadro[0])
    buff_row = None
    for i in range(count_system):
        if buff_row == None:
            syst = []
        else:
            syst = [buff_row]
        for _ in range(count_itr):
            last_row = pow(system[0][k], 2) * 
params_kvadro[index_system][0] \
                        + system[0][k] * params_kvadro[index_system][1] \
                        + params_kvadro[index_system][2] - system[1][j]
            syst.append(last_row)
            k += 1
            j += 1
        index_system += 1
        count_itr -= 1
        kvadro_system.append(syst)
        if index_system < len(params_kvadro):
            buff_row = pow(system[0][k - 1], 2) * 
params_kvadro[index_system][0] \
                            + system[0][k - 1] * 
params_kvadro[index_system][1] \
                            + params_kvadro[index_system][2] - system[1][j 
- 1]

    # Вычисление корней систем

    roots = []
```

```python
    intervals = [10 for i in range(len(params_kvadro[0]))]
    for i in range(len(params_kvadro)):
        root, _ = method_newton(params_kvadro[i], kvadro_system[i],
intervals)
        roots.append(list(root))

    # Вычисление итоговой системы уравнений

    res_func = []
    for row in roots:
        buff = 0
        for i in range(len(row)):
            buff += row[i] * x ** (2 - i)
        res_func.append(buff)

    # Объединяем с диапозоном значений x

    res = []
    start = 0
    end = len(res_func)
    for i in range(len(res_func)):
        res.append((res_func[i], system[0][start:end + 1]))
        start = end
        end = end + len(res_func)

    return res

def draw_func_system(system, name):
    # Определение переменной x как символ из библиотеки SymPy
    x = sp.Symbol('x')

    # Построение графиков
    plt.figure(figsize=(8, 6))

    for equation, x_range in system:
        # Создание массива значений x в указанном диапазоне
        x_values = np.linspace(x_range[0], x_range[-1], 100)

        # Преобразование уравнения в функцию
        equation_func = sp.lambdify(x, equation, 'numpy')

        # Вычисление значений y для каждого значения x
        y_values = equation_func(x_values)

        # Построение графика уравнения
        plt.plot(x_values, y_values)

    # Настройка осей координат
    plt.axhline(0, color='black',linewidth=0.5)
    plt.axvline(0, color='black',linewidth=0.5)
```

```python
    # Отображение четвертей координатной плоскости
    plt.axhline(0, color='black',linewidth=0.5)
    plt.axvline(0, color='black',linewidth=0.5)
    plt.xlim(-25, 25)
    plt.ylim(-25, 25)

    # Добавление подписей к осям
    plt.xlabel('x')
    plt.ylabel('y')
    # Добавление заголовка
    plt.title(name)

    # Добавление легенды
    plt.legend()

    # Отображение графика
    plt.grid(True)
    plt.show()

def draw_func(func, name):

    x = sp.Symbol('x')
    # Создание массива значений x
    x_values = np.linspace(-10, 10, 1000)

    # Вычисление значений функции для каждого значения x
    y_values = [func.subs({x: i})for i in x_values]

    # Построение графика функции
    plt.figure(figsize=(8, 6))
    plt.plot(x_values, y_values, label=f"{name}")

    # Настройка осей координат
    plt.axhline(0, color='black',linewidth=0.5)
    plt.axvline(0, color='black',linewidth=0.5)

    # Отображение четвертей координатной плоскости
    plt.axhline(0, color='black',linewidth=0.5)
    plt.axvline(0, color='black',linewidth=0.5)
    plt.xlim(-25, 25)
    plt.ylim(-25, 25)

    # Добавление подписей к осям
    plt.xlabel('x')
    plt.ylabel('y')

    # Добавление заголовка
    plt.title('График функции')

    # Добавление легенды
    plt.legend()
```

```python
    # Отображение графика
    plt.grid(True)
    plt.show()

if __name__ == "__main__":
    # Система из варианта 19
    system_1 = [[0.234, 0.649, 1.382, 2.672, 2.849],
                [0.511, 0.982, 2.411, 3.115, 4.184]]

    # Система из примера
    system_2 = [[0.351, 0.867, 3.315, 5.013, 6.432],
                [-0.572, -2.015, -3.342, -5.752, -6.911]]

    test_system = deepcopy(system_1)

    # Нахождение интерполяционного многочлена Лагранжа
    print("Полученный многочлен Лагранжа:")
    print("----------------------------------------------")
    polinom1 = lagrange_interpol_polinom(test_system)
    print(polinom1)
    draw_func(polinom1, "Многочлен Лагранажа")
    print("----------------------------------------------")

    # Построение таблицы конечных разностей
    print("Полученная таблица конечных разностей:")
    print("----------------------------------------------")
    print(finite_differences(test_system))
    print("----------------------------------------------")
    # Построение таблицы разделенных разностей
    print("Полученная таблица разделенных разностей:")
    print("----------------------------------------------")
    print(divided_differences(test_system))
    print("----------------------------------------------")
    # Построение полинома Ньютона
    print("----------------------------------------------")
    polinom2, val = newtown_polinom(test_system)
    print("Полученный многочлен ньютона:")
    print(polinom2)
    draw_func(polinom2, "Многочлен ньютона")
    print("Вычисление значения полинома в точке x1 + x2:")
    print(val)
    print("----------------------------------------------")
    # Построение линейного сплайна
    print("Система линейного сплайна и значения параметров:")
    print("----------------------------------------------")
    system1 = build_linear_spline(test_system)
    print(system1)
    draw_func_system(system1, "Линейный сплайн")
    print("----------------------------------------------")
    # Построение квадратичного сплайна
```

```
    print("Система квадратичного сплайна и значения параметров:")
    print("-----------------------------------------------")
    system2 = build_kvadro_spline(test_system)
    print(system2)
    draw_func_system(system2, "Квадратичный сплайн")
    print("-----------------------------------------------")
```