**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ**

**Кафедра вычислительной математики**

**Отчёт**

**Лабораторная работа №1**

“ Интерполирование функций”

Вариант № 12

Снежко Льва Владимировича

студента 3 курса, 3 группы

специальности «Информатика»

дисциплина «Численные методы»

Преподаватель: Будник А.М.

Минск, 2025

Содержание:

Постановка задачи...................................................................................................3

Алгоритм решения..................................................................................................4

Листинг программы.................................................................................................6

Результат и его анализ.............................................................................................8

**Постановка задачи**

Рассмотрим набор различный точек на отрезке [a, b]:

, .

,

Требуется восстановить значение функции в других точках :

[a, b] = [], где

= [j = 12 – номер варианта] = .

Таким образом имеем:

[a, b] = [0.7, 1.7]

Необходимо интерполировать эту функцию:

1. Многочленом Ньютона на равномерной сетке;
2. Многочленом Ньютона на Чебышёвской сетке;
3. Кубическим сплайном;
4. Методом наименьших квадратов.

Для каждого из методов необходимо:

* Вычислить значения интерполяционного многочлена в точках ;
* Вычислить остаток интерполирования в точках ;
* Вычислить истинную погрешность ;
* Сравнить и проанализировать полученные результаты.

**Алгоритм решения**

1. **Интерполирование многочленом Ньютона на равномерной сетке**

Пусть

*где*

*, т.е.*

3 точки восстановления:

,

*,*

Интерполяционный многочлен в форме Ньютона будем искать в следующем виде:

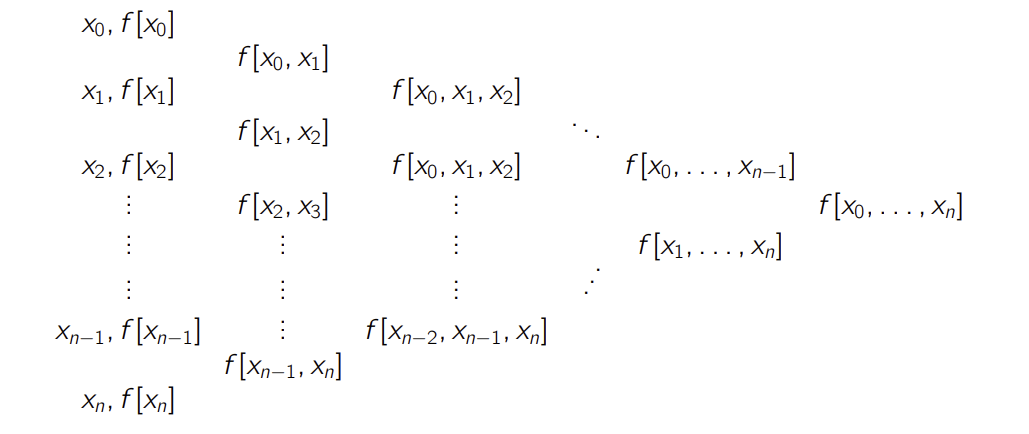
,

где ,

Коэффициенты ИМ удобно вычислять по определению разделённых разностей:

,

путём построения треугольной таблицы следующего вида:



Остаток интерполирования в точках вычислим по следующей формуле:

Истинную погрешность вычислим так:

)

**Листинг кода**

%pip install tabulate

%pip install sympy

import numpy as np

import math

from tabulate import tabulate

import matplotlib.pyplot as plt

import pandas as pd

import sympy as sp

j = 12

n = 10

alpha\_j = 0.1 + 0.05\*j

h = 1/n

def f(x):

    return alpha\_j \* math.exp(x) + (1 - alpha\_j) \* math.sin(x)

# Шаг 1. Построим исходную таблицу

x\_vals = np.array([alpha\_j + i \* h for i in range(n+1)])

f\_vals = np.array([f(x\_) for x\_ in x\_vals])

x\_star = np.array([x\_vals[0] + 2/3\*h, x\_vals[n // 2] / 2 + 0.5 \* h, x\_vals[-1] - 1/3\*h])

f\_star = np.array([f(x\_) for x\_ in x\_star])

print(tabulate(zip(x\_vals, y\_vals), headers=['x', 'f(x)']))

print('\nСпециальные точки')

print(tabulate(zip(x\_star, f\_star), headers=['x\*', 'f(x\*)']))

# Таблица значений функции

table = pd.DataFrame({"x\_i": x\_vals, "f(x\_i)": f\_vals})

table\_transposed = table.T

# Точки для проверки интерполяции

x\_star1 = x\_star[0]

x\_star2 = x\_star[1]

x\_star3 = x\_star[2]

f\_x\_star = f\_star[0]

f\_x\_star2 = f\_star[1]

f\_x\_star3 = f\_star[2]

def compute\_newton\_coefficients(x\_vals, y\_vals):

"""

Возвращает список коэффициентов интерполяционного многочлена Ньютона

с использованием рекурсивного определения разделённых разностей.

"""

n = len(x\_vals)

# Создаём таблицу размером n x n

dd\_table = [y\_vals.copy()] # f[x\_i]

for level in range(1, n):

prev\_column = dd\_table[-1]

curr\_column = []

for i in range(n - level):

numerator = prev\_column[i + 1] - prev\_column[i]

denominator = x\_vals[i + level] - x\_vals[i]

curr\_column.append(numerator / denominator)

dd\_table.append(curr\_column)

# Коэффициенты Ньютона — это верхние элементы каждого столбца

return [dd\_table[i][0] for i in range(n)]

newton\_coeffs = compute\_newton\_coefficients(x\_vals, f\_vals)

def newton\_interpolation(x\_vals, y\_vals, x, coef):

"""

Вычисляет значение интерполяционного многочлена Ньютона в точке x

с использованием рекурсивной формулы:

P\_{n+1}(x) = P\_n(x) + alpha\_{n+1} \* omega\_{n+1}(x)

"""

result = coef[0]

omega = 1.0

for i in range(1, len(coef)):

omega \*= (x - x\_vals[i - 1])

result += coef[i] \* omega

return result

P\_x\_star = newton\_interpolation(x\_vals, f\_vals, x\_star1, newton\_coeffs)

P\_x\_star2 = newton\_interpolation(x\_vals, f\_vals, x\_star2, newton\_coeffs)

P\_x\_star3 = newton\_interpolation(x\_vals, f\_vals, x\_star3, newton\_coeffs)

# Результаты интерполяции

data = {

"Точка": ["x\*", "x\*\*", "x\*\*\*"],

"Значение x": [x\_star1, x\_star2, x\_star3],

"f(x)": [f\_x\_star, f\_x\_star2, f\_x\_star3],

"P(x) (полином)": [P\_x\_star, P\_x\_star2, P\_x\_star3]

}

df = pd.DataFrame(data)

# Производная (n+1)-го порядка

x = sp.Symbol('x')

f\_sym = alpha\_j \* sp.exp(x) + (1 - alpha\_j) \* sp.sin(x)

f\_derivative = sp.diff(f\_sym, x, n + 1)

# Максимум абсолютного значения производной на отрезке [0.7, 1.7]

f\_derivative\_abs = sp.lambdify(x, sp.Abs(f\_derivative), 'numpy')

x\_test = np.linspace(0.7, 1.7, 1000)

M\_max = np.max(f\_derivative\_abs(x\_test))

# Истинная погрешность

r\_x\_star = f\_x\_star - P\_x\_star

r\_x\_star2 = f\_x\_star2 - P\_x\_star2

r\_x\_star3 = f\_x\_star3 - P\_x\_star3

# Оценка погрешности по неравенству

factorial = math.factorial(n + 1)

x\_stars = [x\_star1, x\_star2, x\_star3]

r\_x\_stars = [r\_x\_star, r\_x\_star2, r\_x\_star3]

error\_bound\_stars = []

for x\_val in x\_stars:

prod\_term = np.prod([abs(x\_val - xi) for xi in x\_vals])

error\_bound = M\_max / factorial \* prod\_term

error\_bound\_stars.append(error\_bound)

# Проверка выполнения неравенства

is\_error\_bound\_stars\_valid = [

abs(r\_x\_stars[i]) <= error\_bound\_stars[i] for i in range(3)

]

# Таблица ошибок

error\_table = pd.DataFrame({

"Точка": ["x\*", "x\*\*", "x\*\*\*"],

"Значение x": [x\_star1, x\_star2, x\_star3],

"r истинная": [abs(r) for r in r\_x\_stars],

"оценка погрешности": error\_bound\_stars,

"M = max|f^(n+1)(x)|": [M\_max] \* 3,

"Неравенство выполняется?": is\_error\_bound\_stars\_valid

})

# Вывод таблиц

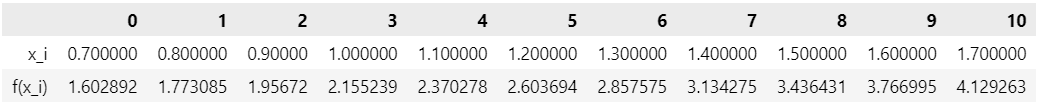
display(table\_transposed)

display(df)

display(error\_table)

**Результаты**

Таблица значений:



|  | **Точка** | **Значение x** | **f(x)** | **P(x) (полином)** |
| --- | --- | --- | --- | --- |
| 0 | x\* | 0.766667 | 1.714927 | 1.714927 |
| 1 | x\*\* | 0.650000 | 1.522435 | 1.522435 |
| 2 | x\*\*\* | 1.666667 | 4.004765 | 4.004765 |

|  | **Точка** | **Значение x** | **r истинная** | **оценка погрешности** | **M = max|f^(n+1)(x)|** | **Неравенство выполняется?** |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | x\* | 0.766667 | 1.021405e-13 | 1.343614e-13 | 2.78765 | True |
| 1 | x\*\* | 0.650000 | 3.516742e-12 | 4.688495e-12 | 2.78765 | True |
| 2 | x\*\*\* | 1.666667 | 2.433609e-13 | 2.863766e-13 | 2.78765 | True |

Коэффициенты интерполяционного многочлена (разделенные разности):

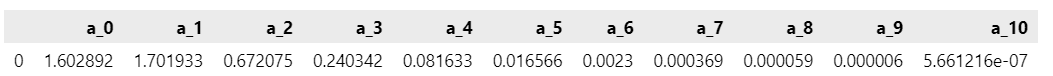
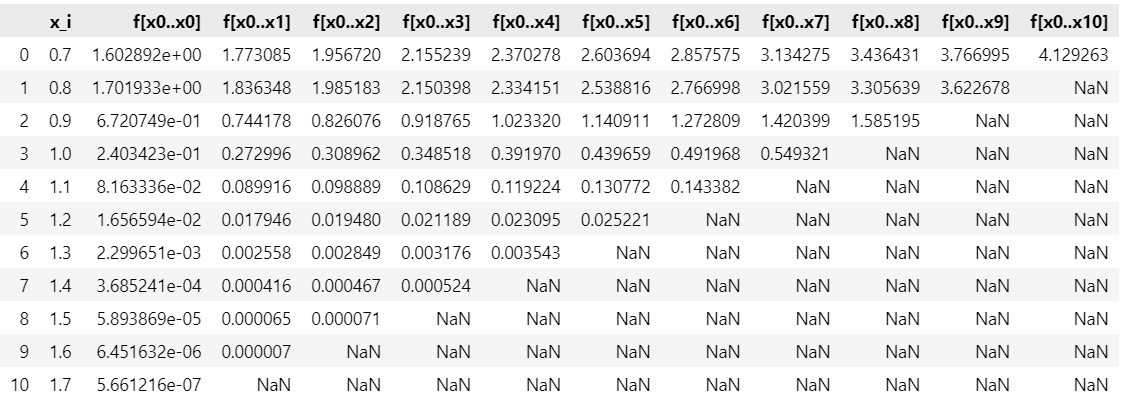


Таблица разделённых разностей



**Анализ**

Порядок не превышает погрешности интерполяции в контрольных точках. Видно, что погрешность возрастает в точках, близких к краям отрезка и имеет наиболее точное значение в точке находящейся близко к середине отрезка. Разница погрешностей в зависимости от точки восстановления связана с возрастанием многочлена (x) и расположением контрольной точки относительно ближайшего узла.

( = -0.000002;

( = 0.000064;

( = 0.000123.

1. **Интерполирование многочленом Ньютона на Чебышёвской сетке**

Пусть теперь

*где*

*i = ,*

*a = 0,7,*

*b = 1,7,*

*n = 10, т. е.*

*i = ,*

Точки восстановления те же:

* ,
* *,*

Остаток интерполирования в точках оценим по следующим формулам:

,

,

n = 10,

a = 0.5, b = 1.5.

**Листинг кода**

import numpy as np

import math

from tabulate import tabulate

import matplotlib.pyplot as plt

import pandas as pd

import sympy as sp

j = 12

n = 10

a = 0.7

b = 1.7

alpha\_j = 0.1 + 0.05\*j

h = 1/n

def f(x):

    return alpha\_j \* math.exp(x) + (1 - alpha\_j) \* math.sin(x)

x\_vals = np.array(sorted([(a+b) / 2 + (b-a) / 2 \* math.cos(math.pi \* (2\*i+1) / (2\*n+2)) for i in range(n+1)]))

f\_vals = np.array([f(x\_) for x\_ in x\_vals])

x\_star = np.array([x\_vals[0] + 2/3\*h, x\_vals[n // 2] + h / 2, x\_vals[n] - h / 3])

f\_star = np.array([f(x\_) for x\_ in x\_star])

print(tabulate(zip(x\_vals, f\_vals), headers=['x', 'f(x)']))

print('\nСпециальные точки')

print(tabulate(zip(x\_star, f\_star), headers=['x\*', 'f(x\*)']))

# Таблица значений функции

table = pd.DataFrame({"x\_i": x\_vals, "f(x\_i)": f\_vals})

table\_transposed = table.T

# Точки для проверки интерполяции

x\_star1 = x\_star[0]

x\_star2 = x\_star[1]

x\_star3 = x\_star[2]

f\_x\_star = f\_star[0]

f\_x\_star2 = f\_star[1]

f\_x\_star3 = f\_star[2]

def compute\_newton\_coefficients(x\_vals, y\_vals):

    """

    Возвращает список коэффициентов интерполяционного многочлена Ньютона

    с использованием рекурсивного определения разделённых разностей.

    """

    n = len(x\_vals)

    # Создаём таблицу размером n x n

    dd\_table = [y\_vals.copy()]  # f[x\_i]

    for level in range(1, n):

        prev\_column = dd\_table[-1]

        curr\_column = []

        for i in range(n - level):

            numerator = prev\_column[i + 1] - prev\_column[i]

            denominator = x\_vals[i + level] - x\_vals[i]

            curr\_column.append(numerator / denominator)

        dd\_table.append(curr\_column)

    # Коэффициенты Ньютона — это верхние элементы каждого столбца

    return dd\_table, [dd\_table[i][0] for i in range(n)]

dd\_table, newton\_coeffs = compute\_newton\_coefficients(x\_vals, f\_vals)

def newton\_interpolation(x\_vals, y\_vals, x, coef):

    """

    Вычисляет значение интерполяционного многочлена Ньютона в точке x

    с использованием рекурсивной формулы:

    P\_{n+1}(x) = P\_n(x) + alpha\_{n+1} \* omega\_{n+1}(x)

    """

    result = coef[0]

    omega = 1.0

    for i in range(1, len(coef)):

        omega \*= (x - x\_vals[i - 1])

        result += coef[i] \* omega

    return result, omega

omegas = [1.0,1.0,1.0]

P\_x\_star, omegas[0] = newton\_interpolation(x\_vals, f\_vals, x\_star1, newton\_coeffs)

P\_x\_star2, omegas[1] = newton\_interpolation(x\_vals, f\_vals, x\_star2, newton\_coeffs)

P\_x\_star3, omegas[2] = newton\_interpolation(x\_vals, f\_vals, x\_star3, newton\_coeffs)

omega\_frame = pd.DataFrame(omegas, index=[f'omega\_{i}' for i in range(len(omegas))])

# Результаты интерполяции

data = {

    "Точка": ["x\*", "x\*\*", "x\*\*\*"],

    "Значение x": [x\_star1, x\_star2, x\_star3],

    "f(x)": [f\_x\_star, f\_x\_star2, f\_x\_star3],

    "P(x) (полином)": [P\_x\_star, P\_x\_star2, P\_x\_star3]

}

n = len(newton\_coeffs)

dd\_frame = pd.DataFrame(dd\_table, columns=[f"f[x0..x{i}]" for i in range(len(newton\_coeffs))])

dd\_frame.insert(0, "x\_i", x\_vals)

coeff\_frame = pd.DataFrame(newton\_coeffs, index=[f"a\_{i}" for i in range(n)]).T

df = pd.DataFrame(data)

# Производная (n+1)-го порядка

x = sp.Symbol('x')

f\_sym = alpha\_j \* sp.exp(x) + (1 - alpha\_j) \* sp.sin(x)

f\_derivative = sp.diff(f\_sym, x, n + 1)

# Максимум абсолютного значения производной на отрезке [0.7, 1.7]

f\_derivative\_abs = sp.lambdify(x, sp.Abs(f\_derivative), 'numpy')

x\_test = np.linspace(0.7, 1.7, 1000)

M\_max = np.max(f\_derivative\_abs(x\_test))

# Истинная погрешность

r\_x\_star = f\_x\_star - P\_x\_star

r\_x\_star2 = f\_x\_star2 - P\_x\_star2

r\_x\_star3 = f\_x\_star3 - P\_x\_star3

# Оценка погрешности по неравенству

factorial = math.factorial(n + 1)

x\_stars = [x\_star1, x\_star2, x\_star3]

r\_x\_stars = [r\_x\_star, r\_x\_star2, r\_x\_star3]

error\_bound\_stars = []

for x\_val in x\_stars:

    prod\_term = np.prod([abs(x\_val - xi) for xi in x\_vals])

    error\_bound = M\_max / factorial \* prod\_term

    error\_bound\_stars.append(error\_bound)

# Проверка выполнения неравенства

is\_error\_bound\_stars\_valid = [

    abs(r\_x\_stars[i]) <= error\_bound\_stars[i] for i in range(3)

]

# Таблица ошибок

error\_table = pd.DataFrame({

    "Точка": ["x\*", "x\*\*", "x\*\*\*"],

    "Значение x": [x\_star1, x\_star2, x\_star3],

    "r истинная": [abs(r) for r in r\_x\_stars],

    "оценка погрешности": error\_bound\_stars,

    "M = max|f^(n+1)(x)|": [M\_max] \* 3,

    "Неравенство выполняется?": is\_error\_bound\_stars\_valid

})

# Вывод таблиц

display(table\_transposed)

display(df)

display(error\_table)

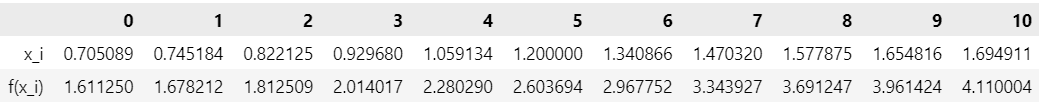
display(dd\_frame)

display(coeff\_frame)

display(omega\_frame)

**Результаты**

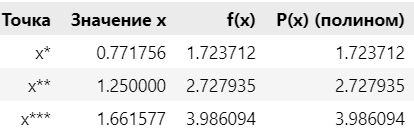
Таблица значений:

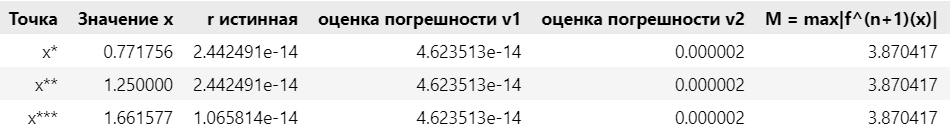


Коэффициенты интерполяционного многочлена (разделенные разности):



Интерполяция в контрольных точках:

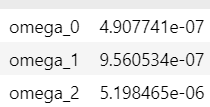




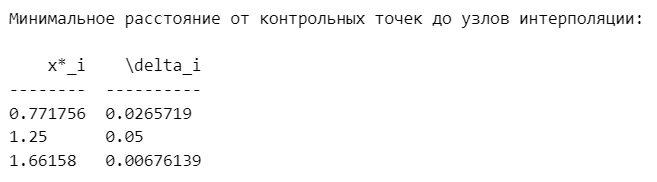
**Анализ**

Нетрудно видеть, что для не превосходит оценки сверху. Также можно заметить, что истинная погрешность в контрольных точках на чебышевской сетке улучшилась по сравнению с равномерной сеткой.

Многочлен в контрольных точках принимает значения:



Давайте проанализируем расположение точек восстановления относительно ближайших узлов:



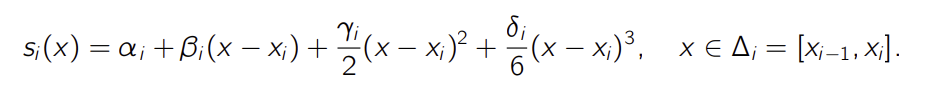
Наименьшая погрешность наблюдается в точке , для которой контрольная точка находится ближе остальных к ближайшему узлу интерполирования, что снижает интерполяционную ошибку.

Наибольшая погрешность соответствует точкам , Заметим, что погрешность интерполяции в этих точках – самая высокая. Можно сделать следующий вывод: чем дальше контрольная точка от узла, тем выше ошибка интерполяции.

1. **Интерполирование с помощью кубического сплайна**

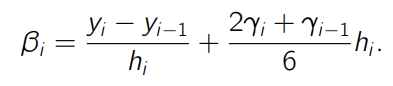
Будем строить кубический сплайн для равномерной сетки.

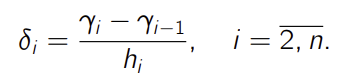
Каждый «кусок» сплайна будем искать в виде:



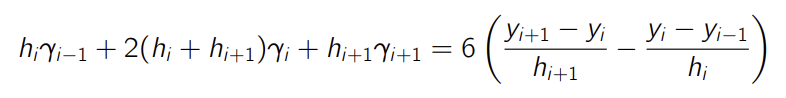
Коэффициенты вычисляем следующим образом:







Для нахождения необходимо решить СЛАУ (i = ):



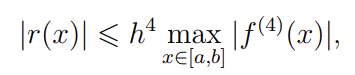
методом прогонки.

В качестве граничных условий берём:

*,*

В нашей задаче мы имеем фиксированное h = 0.1.

Погрешность интерполирования естественным кубическим сплайном на всем отрезке интерполирования может быть оценена следующей константой:



**Листинг кода**

import numpy as np

import math

from tabulate import tabulate

import matplotlib.pyplot as plt

import pandas as pd

import sympy as sp

j = 12

n = 10

alpha\_j = 0.1 + 0.05\*j

a = alpha\_j

b = 1+alpha\_j

h = 1/n

def f(x):

return alpha\_j \* math.exp(x) + (1 - alpha\_j) \* math.sin(x)

x\_vals = np.array([alpha\_j + i \* h for i in range(n+1)])

f\_vals = np.array([f(x\_) for x\_ in x\_vals])

x\_star = np.array([x\_vals[0] + 2/3\*h, x\_vals[n // 2] + 0.5 \* h, x\_vals[-1] - h/3])

f\_star = np.array([f(x\_) for x\_ in x\_star])

print(tabulate(zip(x\_vals, f\_vals), headers=['x', 'f(x)']))

print('\nСпециальные точки')

print(tabulate(zip(x\_star, f\_star), headers=['x\*', 'f(x\*)']))

x\_sym = sp.Symbol('x')

f\_sym = alpha\_j \* sp.exp(x\_sym) + (1 - alpha\_j) \* sp.sin(x\_sym)

f\_diff\_2 = sp.lambdify(x\_sym, sp.diff(f\_sym, x\_sym, 2), 'numpy')

f\_diff\_4\_abs = sp.lambdify(x\_sym, sp.Abs(sp.diff(f\_sym, x\_sym, 4)),'numpy')

def solve\_spline\_system(x, y):

a\_coef = [0] \* n

b\_coef = [0] \* n

for i in range(1, n):

temp1 = 6 \* ((f\_vals[i + 1] - f\_vals[i]) - (f\_vals[i] - f\_vals[i - 1])) / h

temp2 = 4 \* h

a\_coef[i] = -h / temp2

b\_coef[i] = (temp1 - h \* b\_coef[i - 1]) / temp2

for i in range(n - 1, 0, -1):

gamma[i] = a\_coef[i] \* gamma[i + 1] + b\_coef[i]

gamma[0] = f\_diff\_2(x\_vals[0])

gamma[n] = f\_diff\_2(x\_vals[-1])

return gamma

gamma = solve\_spline\_system(x\_vals, f\_vals)

print(tabulate(enumerate(gamma), headers=['gamma\_i']))

def build\_cubic\_spline(x, y, gamma):

# Коэффициенты для каждого интервала [x\_i, x\_{i+1}]

a = y

b = np.zeros(n+1)

c = [g / 2 for g in gamma]

d = np.zeros(n+1)

for i in range(1, n+1):

b[i] = (y[i] - y[i-1]) / h + (2 \* gamma[i] + gamma[i-1]) \* h / 6

for i in range(1, n+1):

d[i] = (gamma[i] - gamma[i-1]) / (6 \* h)

def spline\_func(xq):

# Найти соответствующий интервал

i = np.searchsorted(x, xq)

dx = xq - x[i]

return a[i] + b[i]\*dx + c[i]\*dx\*\*2 + d[i]\*dx\*\*3

# for i in range(1, n+1):

# if x[i-1] <= xq and xq <= x[i]:

# dx = xq - x[i]

# result = a[i] + b[i]\*dx + c[i]\*dx\*\*2 + d[i]\*dx\*\*3

# return result

# return 0.0

return spline\_func

spline = build\_cubic\_spline(x\_vals, f\_vals, gamma)

print(spline)

x\_segment = np.linspace(a,b, 1000)

error\_bound = h\*\*4 \* np.max(f\_diff\_4\_abs(x\_segment))

f\_interpolated = [spline(x\_) for x\_ in x\_star]

true\_error = abs(f\_star - f\_interpolated)

data = pd.DataFrame({

'Точка' : x\_star,

'Значение функции' : f\_star,

'Интерполированное значение' : f\_interpolated,

'Истинная погрешность' : true\_error,

'Оценка погрешности' : error\_bound

})

display(data)

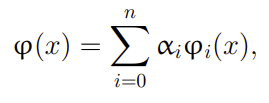
**Результаты**



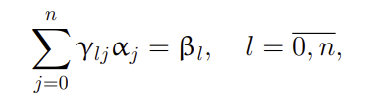
**Анализ**

1. **Интерполирование с помощью метода наименьших квадратов**

Будем строить многочлен МНК 5 степени, он имеет следующий вид:

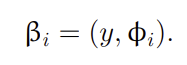
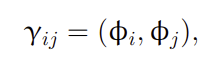
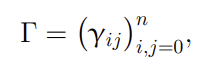


где - является решением системы линейных уравнений:



которая в матричном виде записывается просто как:



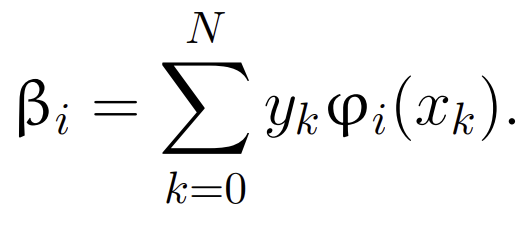
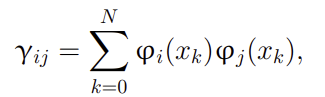


Положим .

Скалярное произведение будем вычислять по формуле:

(u, v) =

Т.е. имеем:



N = 10.

Погрешность будем вычислять по следующей формуле:

**Листинг кода**

public class MNK {

final static double[] X\_VALUES = {0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5};

final static double X\_1 = 0.5 + 0.1 \* 2 / 3;

final static double X\_2 = 1 + 0.1 \* 0.5;

final static double X\_3 = 1.5 - 0.1 \* 1 / 3;

final static double[] TEST\_POINTS = {X\_1, X\_2, X\_3};

static double[] Y\_VALUES = new double[X\_VALUES.length];

static int n = 6;

static int N = 10;

public static void main(String[] args) {

for (int i = 0; i <= N; i++) {

Y\_VALUES[i] = calculateF(X\_VALUES[i]);

}

double[][] gramMatrix = calculateGramMatrix(n);

double[] beta = calculateBeta(n);

double[] alpha = solveSystem(gramMatrix, beta);

System.out.println("Coefficients (alpha): " + Arrays.toString(alpha));

for (double x : TEST\_POINTS) {

double approximatedValue = approximateFunction(alpha, x);

System.out.println("Approximated φ(" + x + ") = " + approximatedValue);

System.out.println("r ист " + (calculateF(x) - approximatedValue));

}

double error = calculateError(alpha);

System.out.println("Approximation error Δ = " + error);

}

private static double[][] calculateGramMatrix(int n) {

double[][] matrix = new double[n][n];

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

matrix[i][j] = 0;

for (int k = 0; k <= N; k++) {

matrix[i][j] += phi(i, X\_VALUES[k]) \* phi(j, X\_VALUES[k]);

}

}

}

return matrix;

}

private static double[] calculateBeta(int n) {

double[] beta = new double[n];

for (int i = 0; i < n; i++) {

beta[i] = 0;

for (int k = 0; k <= N; k++) {

beta[i] += Y\_VALUES[k] \* phi(i, X\_VALUES[k]);

}

}

return beta;

}

private static double phi(int i, double x) {

return Math.pow(x, i);

}

private static double calculateF(double x) {

return 0.5 \* Math.exp(x) + 0.5 \* Math.sin(x);

}

private static double[] solveSystem(double[][] A, double[] b) {

int size = A.length;

double[] x = new double[size];

for (int i = 0; i < size; i++) {

int maxRow = i;

for (int k = i; k < size; k++) {

if (Math.abs(A[k][i]) > Math.abs(A[maxRow][i])) {

maxRow = k;

}

}

if (maxRow != i) {

double[] tempA = A[i];

A[i] = A[maxRow];

A[maxRow] = tempA;

double tempB = b[i];

b[i] = b[maxRow];

b[maxRow] = tempB;

}

for (int j = i + 1; j < size; j++) {

double factor = A[j][i] / A[i][i];

for (int k = i; k < size; k++) {

A[j][k] -= factor \* A[i][k];

}

b[j] -= factor \* b[i];

}

}

for (int i = size - 1; i >= 0; i--) {

double sum = 0;

for (int j = i + 1; j < size; j++) {

sum += A[i][j] \* x[j];

}

x[i] = (b[i] - sum) / A[i][i];

}

return x;

}

private static double approximateFunction(double[] alpha, double x) {

double sum = 0;

for (int i = 0; i < alpha.length; i++) {

sum += alpha[i] \* phi(i, x);

}

return sum;

}

private static double calculateError(double[] alpha) {

double error = 0;

for (int k = 0; k <= N; k++) {

double approximation = approximateFunction(alpha, X\_VALUES[k]);

error += Math.pow(Y\_VALUES[k] - approximation, 2);

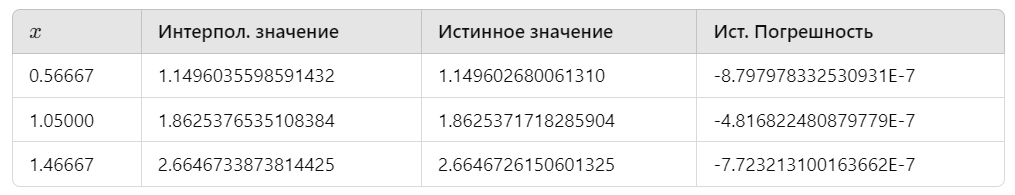
}

return Math.sqrt(error);

}

}

**Результаты**



Оценка погрешности: Δ = 1.5313332717189941E−6.

= [0.499242,1.005162,0.235934,0.019566,0.006303,0.013671].

**Анализ**

Видно, что истинная погрешность для каждой из точек восстановления имеет одинаковый порядок (1e-7) и не превосходит теоретическую погрешность (1e-6).

**Выводы**