**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ**

**Кафедра вычислительной математики**

**Отчёт**

**Лабораторная работа №1-4**

Вариант № 12

Снежко Льва Владимировича

студента 3 курса, 3 группы

специальности «Информатика»

дисциплина «Численные методы»

Преподаватель: Будник А.М.

Минск, 2025

**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 (полином 5 степени).

Для каждого из методов необходимо:

* Вычислить значения интерполяционного многочлена в точках ;
* Вычислить или оценить 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:

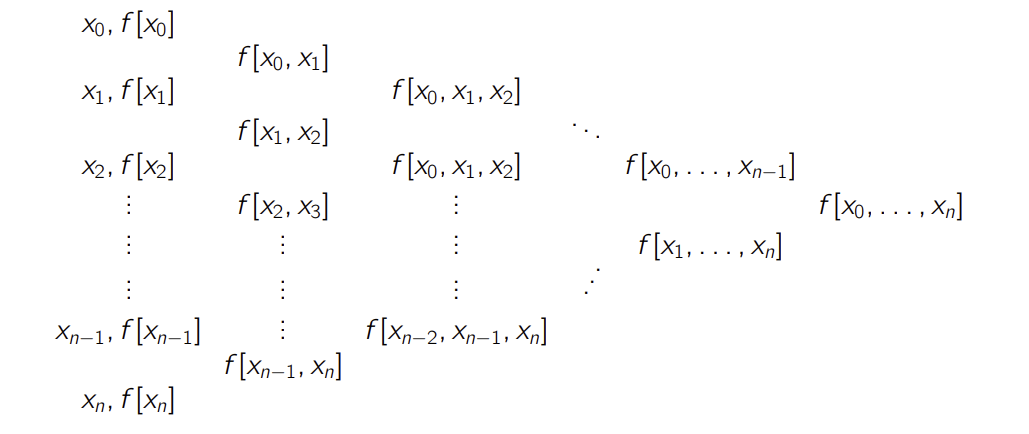
,

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**

# Таблица значений функции

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

table\_transposed = table.T

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(len(prev\_column)-1):

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)]

def extend\_divided\_difference(dd\_table, x\_vals, x\_star, f\_star):

"""

Расширяет таблицу разделённых разностей на одну точку x\_star, f\_star

и возвращает f[x0, ..., xn, x\*] (верхний элемент новой диагонали).

"""

# n = len(x\_vals)

column = [f\_star.copy()]

for k in range(len(x\_vals)):

numerator = column[-1] - dd\_table[k][-1]

denominator = x\_star - x\_vals[k]

column.append(numerator / denominator)

return column[-1]

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

def omega(x\_vals, x\_point):

res = 1

for x in x\_vals:

res \*= (x\_point - x)

return res

omegas = [omega(x\_vals, x\_point) for x\_point in x\_star]

P\_x\_star = [newton\_interpolation(x\_vals, f\_vals, x\_st, newton\_coeffs) for x\_st in x\_star]

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\_star,

"f(x)": f\_star,

"P(x) (полином)": P\_x\_star

}

n = len(newton\_coeffs)

dd\_frame = pd.DataFrame(dd\_table).T

dd\_frame.columns = [f"f[x0..x{i}]"

for i, n in zip(range(len(dd\_table)), reversed(range(len(dd\_table))))]

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)

# Истинная погрешность

r\_x\_stars = np.abs(f\_star - P\_x\_star)

error\_bound\_stars = []

for i in range(len(x\_star)):

error\_bound = abs(extend\_divided\_difference(dd\_table, x\_vals, x\_star[i], f\_star[i]) \* omegas[i])

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\_star,

"r истинная": r\_x\_stars,

"оценка погрешности": error\_bound\_stars,

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

})

# Вывод таблиц

display(table\_transposed)

display(df)

display(error\_table)

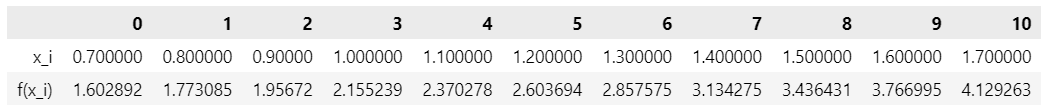
display(dd\_frame)

display(coeff\_frame)

display(omega\_frame)

**Результаты**

Таблица значений:







Коэффициенты интерполяционного многочлена (разделенные разности):

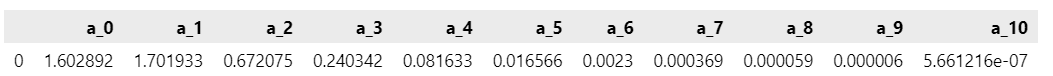
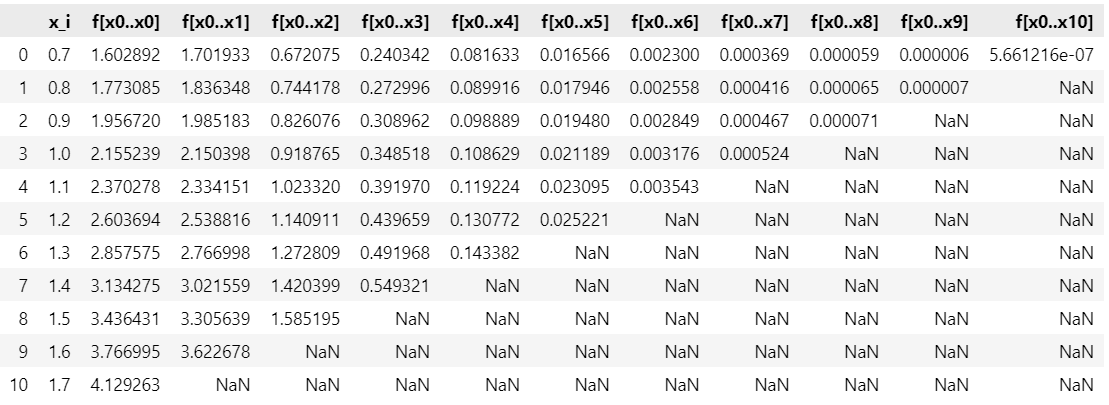
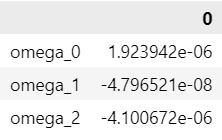


Таблица разделённых разностей



**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 (x\*\*). Разница погрешностей в зависимости от точки восстановления связана с возрастанием многочлена (x) и расположением контрольной точки относительно ближайшего узла.



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 = 0.7, b = 1.7.

**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\*h/3, x\_vals[n // 2] + h / 2, x\_vals[-1] - h / 3])

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

delta\_x\_star = [min([abs(x - x\_s) for x in x\_vals]) for x\_s 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\*)']))

print('\nМинимальное расстояние от контрольных точек до узлов интерполяции:\n')

print(tabulate(zip(x\_star, delta\_x\_star), headers=['x\*\_i', '\delta\_i']))

# 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 extend\_divided\_difference(dd\_table, x\_vals, x\_star, f\_star):

"""

Расширяет таблицу разделённых разностей на одну точку x\_star, f\_star

и возвращает f[x0, ..., xn, x\*] (верхний элемент новой диагонали).

"""

# n = len(x\_vals)

column = [f\_star.copy()]

for k in range(len(x\_vals)):

numerator = column[-1] - dd\_table[k][-1]

denominator = x\_star - x\_vals[k]

column.append(numerator / denominator)

return abs(column[-1])

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 = np.array([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).T

dd\_frame.columns = [f"f[x0..x{i}]"

for i, n in zip(range(len(dd\_table)), reversed(range(len(dd\_table))))]

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

coeff\_frame = pd.DataFrame(newton\_coeffs, index=[f"a\_{i}" for i in range(len(newton\_coeffs))]).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(a, b, 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\_v1 = [M\_max / factorial \* 2 \* ((b-a)/4)\*\*(n+1) for i in range(3)]

error\_bound\_stars\_v2 = [extend\_divided\_difference(dd\_table, x\_vals, x\_star[i], f\_star[i]) \* 2 \* ((b-a)/4)\*\*(n+1) for i in range(3)]

# Проверка выполнения неравенства

is\_error\_bound\_stars\_valid\_v1 = [

abs(r\_x\_stars[i]) <= error\_bound\_stars\_v1[i] for i in range(3)

]

is\_error\_bound\_stars\_valid\_v2 = [

abs(r\_x\_stars[i]) <= error\_bound\_stars\_v2[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 v1": error\_bound\_stars\_v1,

"оценка погрешности v2": error\_bound\_stars\_v2,

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

"Неравенство оценки v1 выполняется?": is\_error\_bound\_stars\_valid\_v1,

"Неравенство оценки v2 выполняется?": is\_error\_bound\_stars\_valid\_v2

})

# Вывод таблиц

display(table\_transposed)

display(df)

display(error\_table)

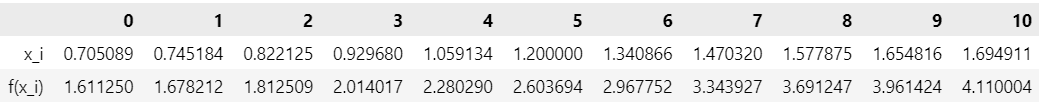
display(dd\_frame)

display(coeff\_frame)

display(omega\_frame)

**Результаты**

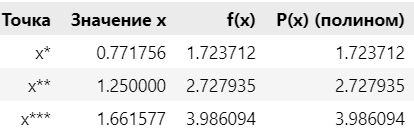
Таблица значений:

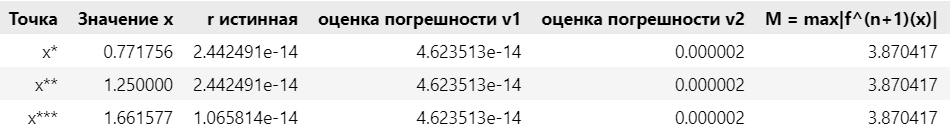


Коэффициенты интерполяционного многочлена (разделенные разности):



Интерполяция в контрольных точках:

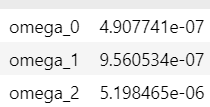




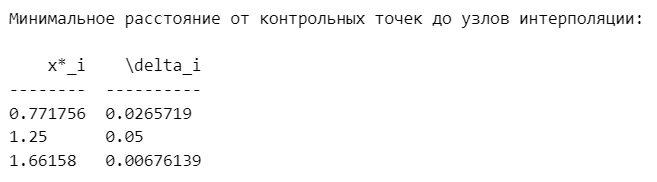
**Анализ**

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:



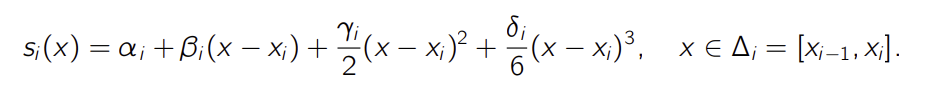
Наименьшая погрешность наблюдается в точке , для которой контрольная точка находится ближе остальных к ближайшему узлу интерполирования, что снижает интерполяционную ошибку.

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**

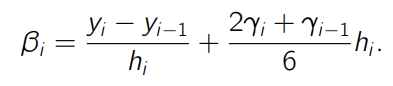
Будем строить кубический сплайн для равномерной сетки.

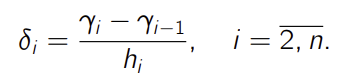
Каждый «кусок» сплайна будем искать в виде:



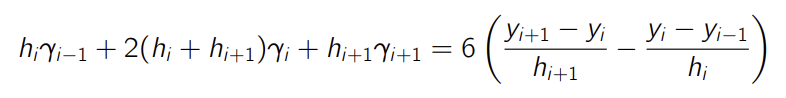
Коэффициенты вычисляем следующим образом:







Для нахождения необходимо решить СЛАУ (i = ):



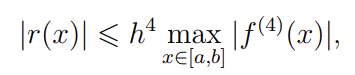
методом прогонки.

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

    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**

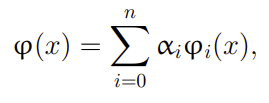


**Анализ**

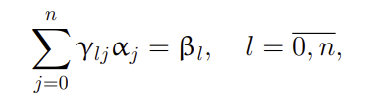
Различия в погрешностях объясняются поведением исходной функции и свойствами кубического сплайна. Истинная погрешность зависит от локальных особенностей функции. Вблизи быстрорастущей экспоненты (правая часть интервала) погрешность стремится к верхней оценке, тогда как в середине и начале интервала она значительно меньше.

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

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

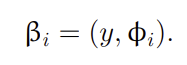
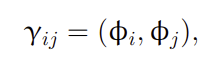
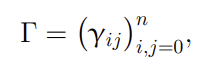


где - является решением системы линейных уравнений:



которая в матричном виде записывается просто как:



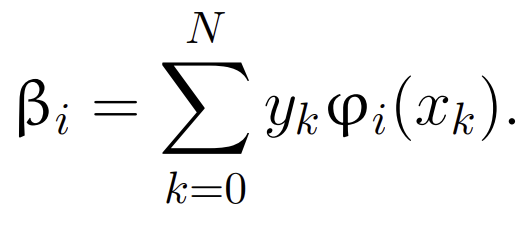
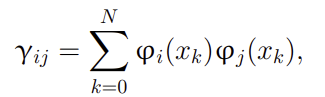


Положим .

Скалярное произведение будем вычислять по формуле:

(u, v) =

Т.е. имеем:



N = 10.

Погрешность будем вычислять по следующей формуле:

**Листинг кода**

import numpy as np

import sympy as sp

import pandas as pd

import matplotlib.pyplot as plt

# Параметры

j = 12

N = 10  # количество точек - 1

h = 1 / N

alpha\_j = 0.1 + 0.05 \* j

n = 6   # количество базисных функций: φ\_0 до φ\_5 => многочлен 5-й степени

# Функция f(x)

def f(x):

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

# Узлы интерполяции

x\_vals = np.array([alpha\_j + 0.1 \* i for i in range(N + 1)])

y\_vals = f(x\_vals)

# Проверочные точки

x\_star = x\_vals[0] + (2 / 3) \* h

x\_star2 = x\_vals[n // 2] + (1 / 2) \* h

x\_star3 = x\_vals[-1] - (1 / 3) \* h

TEST\_POINTS = [x\_star, x\_star2, x\_star3]

def phi(i, x):

    return x \*\* i

def calculate\_gram\_matrix(n, x\_vals):

    A = np.zeros((n, n))

    for i in range(n):

        for j in range(n):

            A[i, j] = sum(phi(i, xk) \* phi(j, xk) for xk in x\_vals)

    return A

def calculate\_beta(n, x\_vals, y\_vals):

    b = np.zeros(n)

    for i in range(n):

        b[i] = sum(fx \* phi(i, xk) for fx, xk in zip(y\_vals, x\_vals))

    return b

def solve\_system(A, b):

    return np.linalg.solve(A, b)

def approximate(alpha, x):

    return sum(alpha[i] \* phi(i, x) for i in range(len(alpha)))

def calculate\_error(alpha, x\_vals):

    return np.sqrt(sum((f(xk) - approximate(alpha, xk)) \*\* 2 for xk in x\_vals))

# Основная логика

A = calculate\_gram\_matrix(n, x\_vals)

b = calculate\_beta(n, x\_vals, y\_vals)

alpha = solve\_system(A, b)

df\_alpha = pd.DataFrame([ [round(a, 6) for a in alpha] ],

                        columns=[f"α{i}" for i in range(len(alpha))])

app\_x\_star = approximate(alpha, x\_star)

app\_x\_star2 = approximate(alpha, x\_star2)

app\_x\_star3 = approximate(alpha, x\_star3)

df = pd.DataFrame({

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

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

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

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

})

# # Истинная ошибка

r\_x\_star = round(abs(f(x\_star) - app\_x\_star),10)

r\_x\_star2 = round(abs(f(x\_star2) - app\_x\_star2), 10)

r\_x\_star3 = round(abs(f(x\_star3) - app\_x\_star3), 10)

error\_bound = calculate\_error(alpha, x\_vals)

is\_error\_bound\_valid = [

    r\_x\_star <= error\_bound,

    r\_x\_star2 <= error\_bound,

    r\_x\_star3 <= error\_bound

]

error\_table = pd.DataFrame({

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

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

    "r(ист)": [r\_x\_star, r\_x\_star2, r\_x\_star3],

    "Оценка погрешности Δ": [error\_bound] \* 3,

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

})

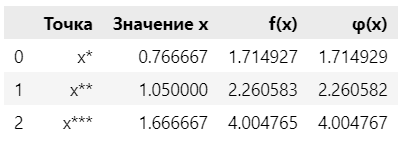
# Отображение таблиц

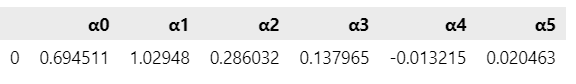
display(df)

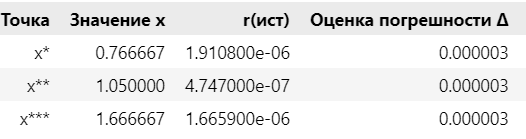
display(df\_alpha)

display(error\_table)

**Результаты**







**Анализ**

В ходе интерполяции с применением метода наименьших квадратов была проведена оценка точности аппроксимации на ряде контрольных точек. Вычисленные значения ошибок оказались порядка 10-7, что свидетельствует о высокой степени точности, достигаемой данным методом. Полученные результаты оказались лучше теоретических оценок, что указывает на эффективность выбранного подхода и точное воспроизведение исходной функции.

Замечено, что значения погрешности для всех тестовых точек имеют сопоставимый масштаб, что говорит о равномерном распределении ошибки по всему интервалу. Это подчеркивает не только точность, но и стабильность метода наименьших квадратов при аппроксимации.

Дополнительно устойчивость метода подтверждается отсутствием значительных расхождений между реальными и аппроксимированными значениями: ошибки во всех точках остаются устойчиво малы, что свидетельствует о надежности численных вычислений.

**Выводы**

Полином пятой степени, использованный для аппроксимации, продемонстрировал высокую эффективность при приближении исходной функции на отрезке [0,7;1,7]. Он не только обеспечивает точное, но и равномерное приближение по всему интервалу, что указывает на достаточность выбранной степени. Повышение степени полинома, скорее всего, не приведет к значимому приросту точности.

Таким образом, в рамках проведённого эксперимента метод наименьших квадратов проявил себя как надёжный и точный инструмент для аппроксимации. Использование полинома пятой степени обеспечило устойчивое и высокоточное приближение, превзошедшее теоретически ожидаемую точность, что подтверждает практическую ценность данного подхода в задачах численного анализа.