**Московский Авиационный Институт**

(Национальный Исследовательский Университет)

Институт №8 «Компьютерные науки и прикладная математика»

**Лабораторная работа №3**

**«Численные методы»**

**Вариант 9**

**Выполнила студентка группы**: М8О-305Б-21

Бондарева Елена Евгеньевна

# Преподаватель: Филиппов Глеб Сергеевич

Оценка: !

Дата: !

Москва, 2024

**Задание 3.1**

Используя таблицу значений  функции , вычисленных в точках  построить интерполяционные многочлены Лагранжа и Ньютона, проходящие через точки . Вычислить значение погрешности интерполяции в точке .

**9.** , a)  ; б) ; .

**Теоретические сведения**

Пусть на отрезке задано множество несовпадающих точек (интерполяционных узлов), в которых известны значения функции . Приближающая функция 𝜑(𝑥, 𝑎) такая, что выполняются равенства

называется интерполяционной.

Наиболее часто в качестве приближающей функции используют многочлены степени 𝑛:

Произвольный многочлен может быть записан в виде

Здесь – многочлены степени 𝑛, так называемые лагранжевы многочлены влияния, которые удовлетворяют условию и, соответственно,

а интерполяционный многочлен запишется в виде

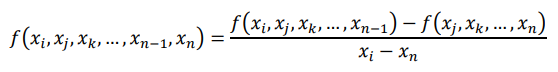
Интерполяционный многочлен, записанный в этой форме, называется интерполяционным многочленом Лагранжа.

Недостатком интерполяционного многочлена Лагранжа является необходимость полного пересчета всех коэффициентов в случае добавления дополнительных интерполяционных узлов. Чтобы избежать указанного недостатка используют интерполяционный многочлен в форме Ньютона.

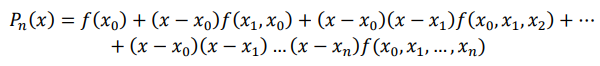
Введем понятие разделенной разности. Разделенные разности нулевого порядка совпадают со значениями функции в узлах. Разделенные разности первого порядка обозначаются и определяются через разделенные разности нулевого порядка:

разделенные разности второго порядка определяются через разделенные разности первого порядка:

Разделенная разность 𝑛 − 𝑘 + 2 определяется соотношениями



**Интерполяционный многочлен Ньютона** может быть представлен в виде:



Отметим, что при добавлении новых узлов первые члены многочлена Ньютона остаются неизменными.

Для повышения точности интерполяции в сумму могут быть добавлены новые члены, что требует подключения дополнительных интерполяционных узлов. При этом безразлично, в каком порядке подключаются новые узлы. Этим формула Ньютона выгодно отличается от формулы Лагранжа.

**Код программы:**

import numpy as np

import math

import matplotlib.pyplot as plt

def f(x):

    return np.arccos(x)

def omega(x, X):

    w = 1

    for i in range(len(X)):

        w \*= (x - X[i])

    return w

def L(x, X):

    l = 0.0

    for i in range(len(X)):

        l += (f(X[i]) \* omega(x, X)) / ((x - X[i]) \* omega(X[i], X))

    return l

def X\_new(i, k, X):

    return [X[j] for j in range(i, k)]

def separate(X):

    if len(X) == 2:

        return (f(X[0]) - f(X[1])) / (X[0] - X[1])

    else:

        return (separate(X\_new(0, len(X) - 1, X)) - separate(X\_new(1, len(X), X))) / (X[0] - X[len(X) - 1])

def xxx(x, i, X):

    res = 1

    for j in range(i):

        res \*= (x - X[j])

    return res

def P(x, X):

    p = f(X[0])

    for i in range(1, len(X)):

        X\_ = X\_new(0, i + 1, X)

        p += xxx(x, i, X) \* separate(X\_)

    return p

def main():

    X\_a = np.array([-0.4, -0.1, 0.5, 0.8])

    X\_b = np.array([-0.4, 0.0, 0.3, 0.5])

    X = 0.1

    Yi = np.array([])

    print("Выберите входные данные (a или b):")

    while True:

        bl = input()

        if bl == "a":

            print("a) Xi = [-0.4, -0.1, 0.2, 0.5]")

            print("Xi = ", X\_a)

            for i in range(0, 4):

                 Yi = np.append(Yi, f(X\_a[i]))

            print("Yi = ", Yi)

            print("\nМногочлен Лагранжа:")

            print("f(x\*) = ", f(X))

            print("L(x\*) = ", L(X, X\_a))

            print("Абсолютная погрешность delta = ", f(X) - L(X, X\_a))

            print("\nМногочлен Ньютона")

            print("f(x\*) = ", f(X))

            print("P(x\*) = ", P(X, X\_a))

            print("Абсолютная погрешность delta = ", f(X) - P(X, X\_a))

            # ГРАФИК ФУНКЦИИ

            xmin = -0.5

            xmax = 0.5

            dx = 0.01

            xarr = np.arange(xmin, xmax, dx)

            ylist = [f(x) for x in xarr]

            y\_X\_a = [f(x) for x in X\_a]

            Larr = X\_a

            Llist = [L(x + 0.1, X\_a) for x in Larr]

            Parr = X\_a

            Plist = [P(x, X\_a) for x in Parr]

            fig = plt.figure(figsize=(8, 6))

            grid = plt.grid(True)

            plt.title('f(x)')

            plt.plot(xarr, ylist)

            plt.plot(Larr, Llist)

            plt.plot(Parr, Plist)

            plt.plot(X\_a, y\_X\_a, '\*')

            plt.xlabel('X')

            plt.ylabel('Y')

            plt.legend(['f(x)', 'L(x)', 'P(x)', 'Xa', 'Ox', 'Oy'])

            plt.show()

            break;

        elif bl == "b":

            print("b) Xi = [-0.4, 0.0, 0.2, 0.5]")

            print("Xi = ", X\_b)

            for i in range(0, 4):

                Yi = np.append(Yi, f(X\_b[i]))

            print("Yi = ", Yi)

            print("\nМногочлен Лагранжа:")

            print("f(x\*) = ", f(X))

            print("L(x\*) = ", L(X, X\_b))

            print("Абсолютная погрешность delta = ", f(X) - L(X, X\_b))

            print("\nМногочлен Ньютона")

            print("f(x\*) = ", f(X))

            print("P(x\*) = ", P(X, X\_b))

            print("Абсолютная погрешность delta = ", f(X) - P(X, X\_b))

            xmin = -0.5

            xmax = 0.5

            dx = 0.01

            xarr = np.arange(xmin, xmax, dx)

            ylist = [f(x) for x in xarr]

            y\_X\_b = [f(x) for x in X\_b]

            Larr = X\_b

            Llist = [L(x + 0.1, X\_b) for x in Larr]

            Parr = X\_b

            Plist = [P(x, X\_b) for x in Parr]

            fig = plt.figure(figsize=(8, 6))

            grid = plt.grid(True)

            plt.title('f(x)')

            plt.plot(xarr, ylist)

            plt.plot(Larr, Llist)

            plt.plot(Parr, Plist)

            plt.plot(X\_b, y\_X\_b, '\*')

            plt.xlabel('X')

            plt.ylabel('Y')

            plt.legend(['f(x)', 'L(x)', 'P(x)', 'Xb', 'Ox', 'Oy'])

            plt.show()

            break;

        else:

            print("error")

            break;

if \_\_name\_\_ == '\_\_main\_\_':

    main()

**Вывод программы:**

**1)**

Выберите входные данные (a или b):

a

a) Xi = [-0.4, -0.1, 0.2, 0.5]

Xi = [-0.4 -0.1 0.5 0.8]

Yi = [1.98231317 1.67096375 1.04719755 0.64350111]

Многочлен Лагранжа:

f(x\*) = 1.4706289056333368

L(x\*) = nan

Абсолютная погрешность delta = nan

Многочлен Ньютона

f(x\*) = 1.4706289056333368

P(x\*) = 1.4744689029731688

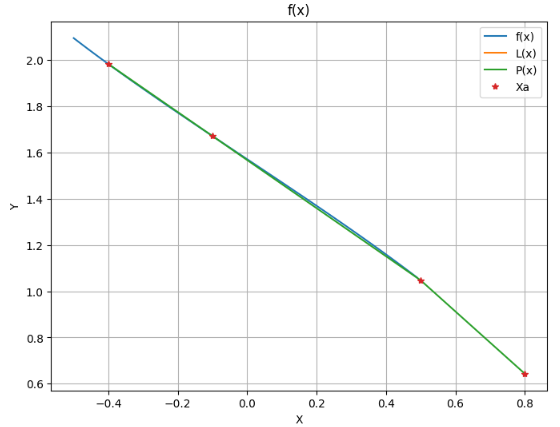
Абсолютная погрешность delta = -0.0038399973398319798

<ipython-input-4-019f8f7ed785>:17: RuntimeWarning: divide by zero encountered in scalar divide

l += (f(X[i]) \* omega(x, X)) / ((x - X[i]) \* omega(X[i], X))

<ipython-input-4-019f8f7ed785>:17: RuntimeWarning: invalid value encountered in scalar add

l += (f(X[i]) \* omega(x, X)) / ((x - X[i]) \* omega(X[i], X))

****

**2)**

Выберите входные данные (a или b):

b

b) Xi = [-0.4, 0.0, 0.2, 0.5]

Xi = [-0.4 0. 0.3 0.5]

Yi = [1.98231317 1.57079633 1.26610367 1.04719755]

Многочлен Лагранжа:

f(x\*) = 1.4706289056333368

L(x\*) = nan

Абсолютная погрешность delta = nan

Многочлен Ньютона

f(x\*) = 1.4706289056333368

P(x\*) = 1.4708182016614235

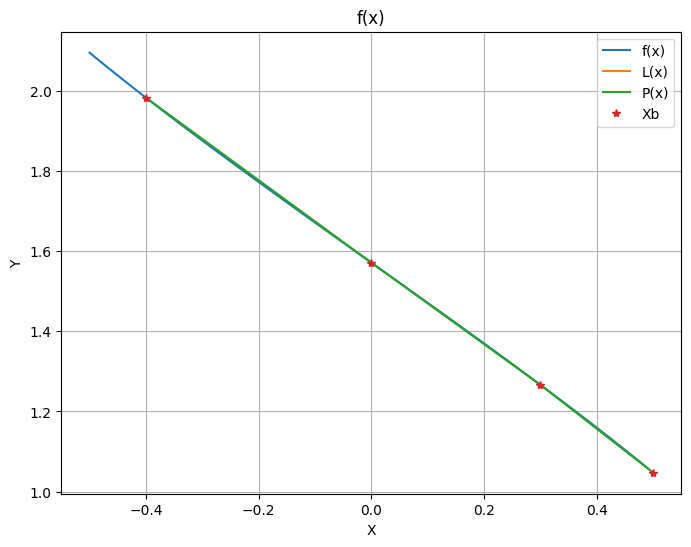
Абсолютная погрешность delta = -0.0001892960280867051

<ipython-input-5-019f8f7ed785>:17: RuntimeWarning: divide by zero encountered in scalar divide

l += (f(X[i]) \* omega(x, X)) / ((x - X[i]) \* omega(X[i], X))

<ipython-input-5-019f8f7ed785>:17: RuntimeWarning: invalid value encountered in scalar add

l += (f(X[i]) \* omega(x, X)) / ((x - X[i]) \* omega(X[i], X))

****

**Вывод:**

В ходе лабораторной работы были изучены интерполяционные многочлены Лагранжа и Ньютона, которые используются для приближения функций по известным значениям в узлах. Было установлено, что многочлены Лагранжа и Ньютона дают достаточно точные результаты при интерполяции функций, однако многочлен Ньютона имеет ряд преимуществ перед многочленом Лагранжа. Одним из таких преимуществ является возможность использования разделенных разностей, которые позволяют быстрее и проще вычислять коэффициенты многочлена.

**Задание 3.2**

Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при  и . Вычислить значение функции в точке .

0.1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
|  | -0.4 | -0.1 | 0.2 | 0.5 | 0.8 |
|  | 1.9823 | 1.6710 | 1.3694 | 1.0472 | 0.64350 |

**Теоретические сведения**

Использование одной интерполяционной формулы на большом числе узлов нецелесообразно. Интерполяционный многочлен может проявить свои колебательные свойства, его значения между узлами могут сильно отличаться от значений интерполируемой функции. Одна из возможностей преодоления этого недостатка заключается в применении сплайн-интерполяции. Суть сплайн-интерполяции заключается в определении интерполирующей функции по формулам одного типа для различных непересекающихся промежутков и в стыковке значений функции и её производных на их границах.

Наиболее широко применяемым является случай, когда между любыми двумя точками разбиения исходного отрезка строится многочлен n-й степени:

который в узлах интерполяции принимает значения аппрокcимируемой функции и непрерывен вместе со своими (𝑛 − 1) производными. Такой кусочно-непрерывный интерполяционный многочлен называется сплайном. Его коэффициенты находятся из условий равенства в узлах сетки значений сплайна и приближаемой функции, а также равенства (𝑛 − 1) производных соответствующих многочленов. На практике наиболее часто используется интерполяционный многочлен третьей степени, который удобно представить как

Для построения кубического сплайна необходимо построить n многочленов третьей степени, т.е. определить неизвестных . Эти коэффициенты ищутся из условий в узлах сетки.

**Код программы:**

import numpy as np

import matplotlib.pyplot as plt

def Lab3\_2(a, b):

    p = np.zeros(len(b))

    q = np.zeros(len(b))

    p[0] = -a[0][1]/a[0][0]

    q[0] = b[0]/a[0][0]

    for i in range(1, len(p)-1):

        p[i] = -a[i][i+1]/(a[i][i] + a[i][i-1]\*p[i-1])

        q[i] = (b[i] - a[i][i-1]\*q[i-1])/(a[i][i] + a[i][i-1]\*p[i-1])

    i = len(a)-1

    p[-1] = 0

    q[-1] = (b[-1] - a[-1][-2]\*q[-2])/(a[-1][-1] + a[-1][-2]\*p[-2])

    x = np.zeros(len(b))

    x[-1] = q[-1]

    for i in reversed(range(len(b)-1)):

        x[i] = p[i]\*x[i+1] + q[i]

    return x

def S(x, X, F, bl):

    h = np.array([])

    n = len(X)

    for i in range(1, n):

        h = np.append(h, X[i] - X[i - 1])

    c\_syst = np.array([])

    rows = np.array([])

    for i in range(n - 2):

        if i == 0:

            rows = np.append(rows, 2 \* (h[i] + h[i + 1]))

        elif i == 1:

            rows = np.append(rows, h[i+1])

        else:

            rows = np.append(rows, 0)

    c\_syst = np.append(c\_syst, rows)

    for i in range(1, n - 3):

        rows = np.array([])

        for j in range(n - 2):

            if i - 1 == j:

                rows = np.append(rows, h[i-1])

            elif i == j:

                rows = np.append(rows, 2 \* (h[i - 1] + h[i]))

            elif i + 1 == j:

                rows = np.append(rows, h[i])

            else:

                rows = np.append(rows, 0)

        c\_syst = np.append(c\_syst, rows)

    rows = np.array([])

    for i in range(2, n):

        if i == n - 2:

            rows = np.append(rows, h[i])

        elif i == n - 1:

            rows = np.append(rows, 2 \* (h[i - 2] + h[i - 1]))

        else:

            rows = np.append(rows, 0)

    c\_syst = np.append(c\_syst, rows)

    b = np.array([])

    for i in range(2, n):

        b = np.append(b, 3 \* ((F[i] - F[i - 1]) / h[i - 1] - (F[i - 1] - F[i - 2]) / h[i - 2]))

    c\_matrix = np.zeros((3, 3))

    k = 0

    for i in range(3):

        for j in range(3):

            c\_matrix[i][j] = c\_syst[k]

            k += 1

    c = np.array([])

    c = np.append(c, 0)

    c = np.append(c, Lab3\_2(c\_matrix, b))

    a = np.array([])

    b = np.array([])

    d = np.array([])

    for i in range(n - 1):

        a = np.append(a, F[i])

        if i == n - 2:

            b = np.append(b, (F[i + 1] - F[i]) / h[i] - (2 / 3) \* h[i] \* c[i])

            d = np.append(d, - c[i] / (3 \* h[i]))

        else:

            b = np.append(b, (F[i + 1] - F[i]) / h[i] - (1 / 3) \* h[i] \* (c[i + 1] + 2 \* c[i]))

            d = np.append(d, (c[i + 1] - c[i]) / (3 \* h[i]))

    if bl == 1:

        print("\nКоэффициенты сплайнов:")

        print("a = ", a)

        print("b = ", b)

        print("c = ", c)

        print("d = ", d)

    for i in range(n - 1):

        if (x >= X[i]) & (x <= X[i + 1]):

            res = a[i] + b[i] \* (x - X[i]) + c[i] \* (x - X[i]) \*\* 2 + d[i] \* (x - X[i]) \*\* 3

            break

    return res

def main():

    x = 0.1

    X = np.array([ -0.4, -0.1, 0.2, 0.5, 0.8])

    F = np.array([1.9823, 1.6710, 1.3694, 1.0472, 0.64350])

    print('f(x\*) = ', S(x, X, F, 1))

    xmin = -0.4

    xmax = 0.5

    dx = 0.001

    xarr = np.arange(xmin, xmax, dx)

    ylist = [S(x\_, X, F, 0) for x\_ in xarr]

    fig = plt.figure(figsize=(6, 4))

    grid = plt.grid(True)

    plt.title('S(x)')

    plt.plot(xarr, ylist)

    plt.plot(X, F, '\*')

    plt.xlabel('X')

    plt.ylabel('Y')

    plt.legend(['S(x)', 'Xi'])

    plt.show()

main()

**Вывод программы:**

Коэффициенты сплайнов:

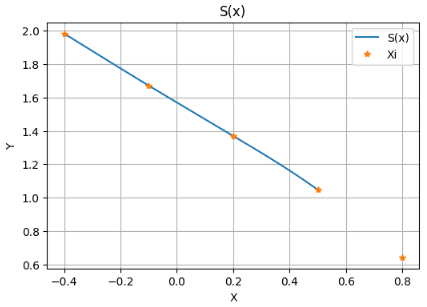
a = [1.9823 1.671 1.3694 1.0472]

b = [-1.04638095 -1.0202381 -1.00166667 -1.21109524]

c = [ 0. 0.08714286 -0.0252381 -0.67285714]

d = [ 0.0968254 -0.12486772 -0.71957672 0.74761905]

f(x\*) = 1.4694391534391535

****

**Вывод:**

В результате выполнения лабораторной работы был изучен кубический сплайн для приближения функций, который используется для интерполяции данных на отрезках. Было установлено, что кубический сплайн позволяет достаточно точно аппроксимировать функцию на всем интервале, а также обладает свойством гладкости и непрерывности первой производной в узлах. Кроме того, была использована методика вычисления коэффициентов сплайна, которая включает решение системы линейных уравнений методом прогонки.

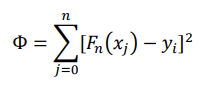
**Задание 3.3**

Для таблично заданной функции путем решения нормальной системы МНК найти приближающие многочлены a) 1-ой и б) 2-ой степени. Для каждого из приближающих многочленов вычислить сумму квадратов ошибок. Построить графики приближаемой функции и приближающих многочленов.

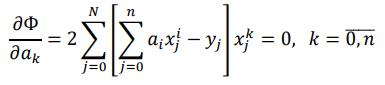
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
|  | -0.7 | -0.4 | -0.1 | 0.2 | 0.5 | 0.8 |
|  | 2.3462 | 1.9823 | 1.671 | 1.3694 | 1.0472 | 0.6435 |

**Теоретические сведения**

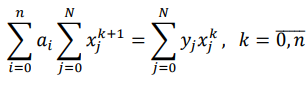
Пусть задана таблично в узлах функция . При этом значения функции определены с некоторой погрешностью, также из физических соображений известен вид функции, которой должны приближенно удовлетворять табличные точки, например: многочлен степени 𝑛, у которого неизвестны коэффициенты . Неизвестные коэффициенты будем находить из условия минимума квадратичного отклонения многочлена от таблично заданной функции



Минимума Ф можно добиться только за счет изменения коэффициентов многочлена . Необходимые условия экстремума имеют вид



Эту систему для удобства преобразуют к следующему виду:



Эта система называется нормальной системой метода наименьших квадратов (МНК) и представляет собой систему линейных алгебраических уравнений относительно коэффициентов . Решив систему, построим многочлен , приближающий функцию 𝑓(𝑥) и минимизирующий квадратичное отклонение.

**Код программы:**

**I:**

import copy, math, cmath

import matplotlib.pyplot as plt

import numpy as np

def matrixmult(A, B):

    C = [[0.0 for col in range(len(B[0]))] for row in range(len(A))]

    for i in range(len(A)):

        for j in range(len(B[0])):

            for k in range(len(B)):

                C[i][j] += A[i][k]\*B[k][j]

    return C

def pivot\_matrix(M):

    m = len(M)

    MCopy = copy.deepcopy(M)

    id\_mat = [[float(i==j) for i in range(m)] for j in range(m)]

    row\_exchanges = 0

    for i in range(m):

        maxElem = abs(MCopy[i][i])

        maxRow = i

        for k in range(i+1, m):

            if(abs(MCopy[k][i]) > maxElem):

                maxElem = abs(MCopy[k][i])

                maxRow = k

        if i != maxRow:

            id\_mat[i], id\_mat[maxRow] = id\_mat[maxRow], id\_mat[i]

            MCopy[i], MCopy[maxRow] = MCopy[maxRow], MCopy[i]

            row\_exchanges += 1

    return id\_mat, row\_exchanges

def lup\_decomposition(A):

    n = len(A)

    L = [[0.0] \* n for i in range(n)]

    U = [[0.0] \* n for i in range(n)]

    P, rowExc = pivot\_matrix(A)

    PA = matrixmult(P, A)

    for j in range(n):

        L[j][j] = 1.0

        for i in range(j+1):

            s = sum(L[i][k] \* U[k][j] for k in range(i))

            U[i][j] = PA[i][j] - s

        for i in range(j, n):

            s = sum(L[i][k] \* U[k][j] for k in range(j))

            L[i][j] = (PA[i][j] - s) / U[j][j]

    return (P, L, U, rowExc)

def lup\_solve(P,L,U,B):

    n = len(P)

    Bt = matrixmult(P, [[i] for i in B])

    Y = [0.0 for i in range(n)]

    for i in range(n):

        Y[i] = Bt[i][0]/L[i][i]

        for k in range(i):

            Y[i] -= Y[k]\*L[i][k]

    X = [0.0 for i in range(n)]

    for i in range(n-1,-1,-1):

        s = sum(X[k]\*U[i][k] for k in range(i+1,n))

        X[i] = (Y[i] - s)/U[i][i]

    return X

def lsm1(x,y):

    n = len(x)

    A = [[n, sum(x)],

         [sum(x), sum(el\*\*2 for el in x)]]

    P, L, U, \_ = lup\_decomposition(A)

    ynew = [sum(y),sum(x[i]\*y[i] for i in range(n))]

    a = lup\_solve(P,L,U,ynew)

    return lambda x: a[0] + a[1]\*x

def main():

    x = [-0.7, -0.4, -0.1, 0.2, 0.5, 0.8]

    y = [2.3462, 1.9823, 1.671, 1.3694, 1.0472, 0.6435]

    func = lsm1(x,y)

    xnew = np.linspace(min(x),max(x),100)

    ynew = [func(el) for el in xnew]

    print(x)

    print(y)

    print(xnew)

    print(ynew)

    print("\nГрафик:")

    plt.plot(x,y,'o',xnew,ynew)

    plt.grid(True)

    plt.show()

    print("\nВывод суммы квадратов ошибок:")

    print(sum((func(x[i])-y[i])\*\*2 for i in range(len(x))))

if \_\_name\_\_ == '\_\_main\_\_':

      main()

**Вывод программы (I):**

[-0.7, -0.4, -0.1, 0.2, 0.5, 0.8]

[2.3462, 1.9823, 1.671, 1.3694, 1.0472, 0.6435]

[-0.7 -0.68484848 -0.66969697 -0.65454545 -0.63939394 -0.62424242

-0.60909091 -0.59393939 -0.57878788 -0.56363636 -0.54848485 -0.53333333

-0.51818182 -0.5030303 -0.48787879 -0.47272727 -0.45757576 -0.44242424

-0.42727273 -0.41212121 -0.3969697 -0.38181818 -0.36666667 -0.35151515

-0.33636364 -0.32121212 -0.30606061 -0.29090909 -0.27575758 -0.26060606

-0.24545455 -0.23030303 -0.21515152 -0.2 -0.18484848 -0.16969697

-0.15454545 -0.13939394 -0.12424242 -0.10909091 -0.09393939 -0.07878788

-0.06363636 -0.04848485 -0.03333333 -0.01818182 -0.0030303 0.01212121

0.02727273 0.04242424 0.05757576 0.07272727 0.08787879 0.1030303

0.11818182 0.13333333 0.14848485 0.16363636 0.17878788 0.19393939

0.20909091 0.22424242 0.23939394 0.25454545 0.26969697 0.28484848

0.3 0.31515152 0.33030303 0.34545455 0.36060606 0.37575758

0.39090909 0.40606061 0.42121212 0.43636364 0.45151515 0.46666667

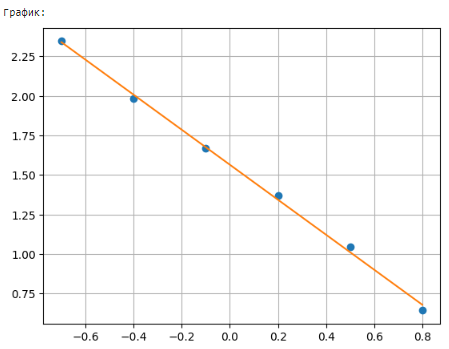
0.48181818 0.4969697 0.51212121 0.52727273 0.54242424 0.55757576

0.57272727 0.58787879 0.6030303 0.61818182 0.63333333 0.64848485

0.66363636 0.67878788 0.69393939 0.70909091 0.72424242 0.73939394

0.75454545 0.76969697 0.78484848 0.8 ]

[2.3399619047619042, 2.3231936507936504, 2.3064253968253965, 2.2896571428571426, 2.2728888888888887, 2.2561206349206344, 2.2393523809523805, 2.2225841269841267, 2.205815873015873, 2.189047619047619, 2.1722793650793646, 2.1555111111111107, 2.138742857142857, 2.121974603174603, 2.105206349206349, 2.0884380952380948, 2.071669841269841, 2.054901587301587, 2.038133333333333, 2.0213650793650793, 2.004596825396825, 1.9878285714285713, 1.9710603174603172, 1.9542920634920633, 1.9375238095238094, 1.9207555555555553, 1.9039873015873015, 1.8872190476190474, 1.8704507936507935, 1.8536825396825396, 1.8369142857142855, 1.8201460317460316, 1.8033777777777775, 1.7866095238095236, 1.7698412698412698, 1.753073015873016, 1.7363047619047618, 1.7195365079365077, 1.7027682539682538, 1.686, 1.6692317460317458, 1.652463492063492, 1.635695238095238, 1.618926984126984, 1.6021587301587301, 1.585390476190476, 1.5686222222222221, 1.5518539682539683, 1.5350857142857142, 1.5183174603174603, 1.5015492063492064, 1.4847809523809523, 1.4680126984126984, 1.4512444444444443, 1.4344761904761905, 1.4177079365079366, 1.4009396825396825, 1.3841714285714286, 1.3674031746031747, 1.3506349206349206, 1.3338666666666668, 1.3170984126984127, 1.3003301587301588, 1.2835619047619047, 1.2667936507936508, 1.250025396825397, 1.233257142857143, 1.216488888888889, 1.199720634920635, 1.1829523809523812, 1.166184126984127, 1.1494158730158732, 1.1326476190476193, 1.115879365079365, 1.0991111111111111, 1.0823428571428573, 1.0655746031746034, 1.0488063492063493, 1.0320380952380954, 1.0152698412698413, 0.9985015873015874, 0.9817333333333336, 0.9649650793650796, 0.9481968253968257, 0.9314285714285717, 0.9146603174603177, 0.8978920634920639, 0.8811238095238099, 0.8643555555555557, 0.8475873015873018, 0.8308190476190478, 0.8140507936507938, 0.7972825396825399, 0.7805142857142859, 0.763746031746032, 0.7469777777777781, 0.7302095238095241, 0.7134412698412702, 0.6966730158730162, 0.6799047619047622]

****

Вывод суммы квадратов ошибок:

0.003940351047619047

**Код программы:**

**II:**

def lup\_decomposition(A):

    n = len(A)

    L = [[0.0] \* n for i in range(n)]

    U = [[0.0] \* n for i in range(n)]

    P, rowExc = pivot\_matrix(A)

    PA = matrixmult(P, A)

    for j in range(n):

        L[j][j] = 1.0

        for i in range(j+1):

            s = sum(L[i][k] \* U[k][j] for k in range(i))

            U[i][j] = PA[i][j] - s

        for i in range(j, n):

            s = sum(L[i][k] \* U[k][j] for k in range(j))

            L[i][j] = (PA[i][j] - s) / U[j][j]

    return (P, L, U, rowExc)

def lup\_solve(P,L,U,B):

    n = len(P)

    Bt = matrixmult(P, [[i] for i in B])

    Y = [0.0 for i in range(n)]

    for i in range(n):

        Y[i] = Bt[i][0]/L[i][i]

        for k in range(i):

            Y[i] -= Y[k]\*L[i][k]

    X = [0.0 for i in range(n)]

    for i in range(n-1,-1,-1):

        s = sum(X[k]\*U[i][k] for k in range(i+1,n))

        X[i] = (Y[i] - s)/U[i][i]

    return X

def lsm2(x,y):

    n = len(x)

    A = [[n, sum(x), sum(el\*\*2 for el in x)],

         [sum(x), sum(el\*\*2 for el in x), sum(el\*\*3 for el in x)],

         [sum(el\*\*2 for el in x), sum(el\*\*3 for el in x), sum(el\*\*4 for el in x)]]

    P, L, U, \_ = lup\_decomposition(A)

    ynew = [sum(y),sum(x[i]\*y[i] for i in range(n)),sum(y[i]\*x[i]\*\*2 for i in range(n))]

    a = lup\_solve(P,L,U,ynew)

    return lambda x: a[0] + a[1]\*x + a[2]\*x\*\*2

def main():

    x = [-0.7, -0.4, -0.1, 0.2, 0.5, 0.8]

    y = [2.3462, 1.9823, 1.671, 1.3694, 1.0472, 0.6435]

    func = lsm2(x,y)

    xnew = np.linspace(min(x),max(x),100)

    ynew = [func(el) for el in xnew]

    print(x)

    print(y)

    print(xnew)

    print(ynew)

    print("\nГрафик:")

    plt.plot(x,y,'o',xnew,ynew)

    plt.grid(True)

    plt.show()

    print("\nВывод суммы квадратов ошибок:")

    print(sum((func(x[i])-y[i])\*\*2 for i in range(len(x))))

if \_\_name\_\_ == '\_\_main\_\_':

      main()

**Вывод программы (II):**

[-0.7, -0.4, -0.1, 0.2, 0.5, 0.8]

[2.3462, 1.9823, 1.671, 1.3694, 1.0472, 0.6435]

[-0.7 -0.68484848 -0.66969697 -0.65454545 -0.63939394 -0.62424242

-0.60909091 -0.59393939 -0.57878788 -0.56363636 -0.54848485 -0.53333333

-0.51818182 -0.5030303 -0.48787879 -0.47272727 -0.45757576 -0.44242424

-0.42727273 -0.41212121 -0.3969697 -0.38181818 -0.36666667 -0.35151515

-0.33636364 -0.32121212 -0.30606061 -0.29090909 -0.27575758 -0.26060606

-0.24545455 -0.23030303 -0.21515152 -0.2 -0.18484848 -0.16969697

-0.15454545 -0.13939394 -0.12424242 -0.10909091 -0.09393939 -0.07878788

-0.06363636 -0.04848485 -0.03333333 -0.01818182 -0.0030303 0.01212121

0.02727273 0.04242424 0.05757576 0.07272727 0.08787879 0.1030303

0.11818182 0.13333333 0.14848485 0.16363636 0.17878788 0.19393939

0.20909091 0.22424242 0.23939394 0.25454545 0.26969697 0.28484848

0.3 0.31515152 0.33030303 0.34545455 0.36060606 0.37575758

0.39090909 0.40606061 0.42121212 0.43636364 0.45151515 0.46666667

0.48181818 0.4969697 0.51212121 0.52727273 0.54242424 0.55757576

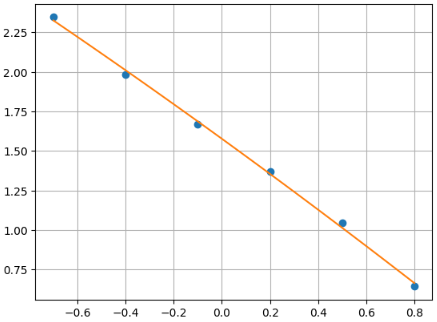
0.57272727 0.58787879 0.6030303 0.61818182 0.63333333 0.64848485

0.66363636 0.67878788 0.69393939 0.70909091 0.72424242 0.73939394

0.75454545 0.76969697 0.78484848 0.8 ]

[2.3255214285714287, 2.3098360998148877, 2.294128670543822, 2.2783991407582316, 2.2626475104581165, 2.2468737796434763, 2.231077948314312, 2.2152600164706224, 2.1994199841124082, 2.183557851239669, 2.1676736178524054, 2.151767283950617, 2.1358388495343035, 2.119888314603466, 2.1039156791581033, 2.0879209431982155, 2.071904106723803, 2.0558651697348664, 2.0398041322314047, 2.0237209942134182, 2.007615755680907, 1.9914884166338709, 1.97533897707231, 1.9591674369962244, 1.9429737964056142, 1.926758055300479, 1.9105202136808195, 1.8942602715466348, 1.8779782288979254, 1.8616740857346914, 1.8453478420569325, 1.8289994978646489, 1.8126290531578406, 1.7962365079365077, 1.7798218622006496, 1.763385115950267, 1.7469262691853598, 1.7304453219059275, 1.7139422741119708, 1.697417125803489, 1.6808698769804826, 1.6643005276429514, 1.6477090777908956, 1.631095527424315, 1.6144598765432094, 1.5978021251475794, 1.5811222732374242, 1.5644203208127447, 1.5476962678735402, 1.530950114419811, 1.5141818604515571, 1.4973915059687783, 1.480579050971475, 1.4637444954596466, 1.4468878394332936, 1.430009082892416, 1.4131082258370133, 1.396185268267086, 1.3792402101826342, 1.3622730515836574, 1.3452837924701557, 1.3282724328421296, 1.3112389726995783, 1.2941834120425026, 1.2771057508709023, 1.2600059891847768, 1.2428841269841269, 1.225740164268952, 1.2085741010392523, 1.1913859372950282, 1.1741756730362791, 1.1569433082630054, 1.1396888429752066, 1.122412277172883, 1.105113610856035, 1.087792844024662, 1.0704499766787643, 1.0530850088183419, 1.035697940443395, 1.0182887715539233, 1.0008575021499264, 0.983404132231405, 0.9659286617983589, 0.948431090850788, 0.9309114193886923, 0.9133696474120718, 0.8958057749209267, 0.8782198019152567, 0.8606117283950618, 0.8429815543603424, 0.8253292798110982, 0.8076549047473291, 0.7899584291690355, 0.772239853076217, 0.7544991764688738, 0.7367363993470057, 0.718951521710613, 0.7011445435596956, 0.6833154648942532, 0.6654642857142863]

График:



Вывод суммы квадратов ошибок:

0.0032396991428571102

**Вывод:**

В ходе лабораторной работе было изучено построение приближающих многочленов первой и второй степени методом наименьших квадратов. Было выявлено, что многочлен первой степени может быть использован для линейного приближения данных, в то время как многочлен второй степени обеспечивает более точное приближение и учитывает кривизну функции.

**Задание 3.4**

Вычислить первую и вторую производную от таблично заданной функции в точке .

 1.0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
|  | -1.0 | 0.0 | 1.0 | 2.0 | 3.0 |
|  | 2.3562 | 1.5708 | 0.7854 | 0.46365 | 0.32175 |

**Теоретические сведения**

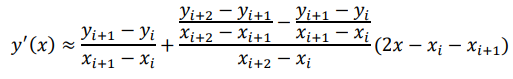
Формулы численного дифференцирования в основном используются при нахождении производных от функции 𝑦 = 𝑓(𝑥), заданной таблично. Исходная функция на отрезках , заменяется некоторой приближающей, легко вычисляемой функцией , где 𝑅(𝑥) – остаточный член приближения, – набор коэффициентов, вообще говоря, различный для каждого из рассматриваемых отрезков, и полагают, что . Наиболее часто в качестве приближающей функции берется интерполяционный многочлен , а производные соответствующих порядков определяются дифференцированием многочлена.

При решении практических задач, как правило, используются аппроксимации первых и вторых производных.

В первом приближении, таблично заданная функция может быть аппроксимирована отрезками прямой. В этом случае:

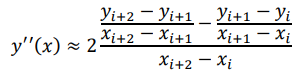


При использовании для аппроксимации таблично заданной функции интерполяционного многочлена второй степени имеем:



При равностоящих точках разбиения, данная формула обеспечивает второй порядок точности.

Для вычисления второй производной, необходимо использовать интерполяционный многочлен, как минимум второй степени. После дифференцирования многочлена получаем



**Код программы:**

def main():

    X = [-1.0, 0.0, 1.0, 2.0, 3.0]

    Y = [2.3562, 1.5708, 0.7854, 0.46365, 0.32175]

    Xp = 1.0

    Yd = (Y[2] - Y[1])/(X[2]-X[1])

    print("The left-hand derivative 1:")

    print(Yd)

    Yd = (Y[3] - Y[2])/(X[3]-X[2])

    print("The right-hand derivative 1:")

    print(Yd)

    Yd = (Y[2] - Y[1])/(X[2]-X[1]) + (((Y[3]-Y[2])/(X[3]-X[2])-(Y[2]-Y[1])/(X[2]-X[1]))/(X[3]-X[1]))\*(2\*Xp-X[1]-X[2])

    print("The first derivative:")

    print(Yd)

    Yd = 2\*(((Y[3]-Y[2])/(X[3]-X[2])-(Y[2]-Y[1])/(X[2]-X[1]))/(X[3]-X[1]))

    print("The second derivative:")

    print(Yd)

if \_\_name\_\_ == '\_\_main\_\_':

      main()

**Вывод программы:**

The left-hand derivative 1:

-0.7854

The right-hand derivative 1:

-0.32175

The first derivative:

-0.5535749999999999

The second derivative:

0.46365

**Вывод:**

В результате выполнения лабораторной работы были изучены численные методы нахождения первой и второй производной таблично заданной функции в заданной точке. Были рассмотрены методы конечных разностей, которые позволяют вычислять производные по определенным формулам, используя значения функции в соседних точках. Было установлено, что точность вычисления производных зависит от шага сетки и от порядка используемых разностных формул.

**Задание 3.5**

Вычислить определенный интеграл , методами прямоугольников, трапеций, Симпсона с шагами . Оценить погрешность вычислений, используя Ме­тод Рунге-Ромберга:

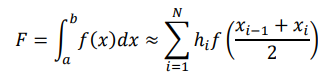
, ;

**Теоретические сведения**

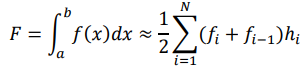
Формулы численного интегрирования используются в тех случаях, когда вычислить аналитически определенный интеграл не удается. Рассмотрим наиболее простой и часто применяемый способ, когда подынтегральную функцию заменяют на интерполяционный многочлен.

При использовании интерполяционных многочленов различной степени, получают формулы численного интегрирования различного порядка точности.

Заменим подынтегральную функцию, интерполяционным многочленом Лагранжа нулевой степени, проходящим через середину отрезка – точку , получим формулу прямоугольников:



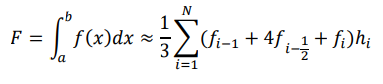
В случае таблично заданных функций удобно в качестве узлов интерполяции выбрать начало и конец отрезка интегрирования, т.е. заменить функцию 𝑓(𝑥) многочленом Лагранжа первой степени



Эта формула носит название формулы трапеций.

Для повышения порядка точности формулы численного интегрирования заменим подынтегральную кривую параболой – интерполяционным многочленом второй степени, выбрав в качестве узлов интерполяции концы и середину отрезка интегрирования: .

Для случая , получим формулу Симпсона:



Метод Рунге-Ромберга-Ричардсона позволяет получать более высокий порядок точности вычисления. Если имеются результаты вычисления определенного интеграла порядка точности p на сетке с шагом и на сетке с шагом , то



**Код программы:**

import numpy as np

def y(x):

    return x/(x\*\*2+9)

def rectangle(x0, xk, f, h):

    X = np.arange(x0, xk, h)

    return h \* sum([f(X[i] + h / 2) for i in range(len(X))])  # метод прямоугольников

def trapeze(x0, xk, f, h):

    X = np.arange(x0, xk, h)

    return h \* ((f(X[0]) + f(xk)) / 2 + sum([f(X[i]) for i in range(1, len(X))]))   # метод трапеций

def Simpson(x0, xk, f, h):

    res = 0

    x = x0 + h

    while x < xk:

        res += f(x - h) + 4 \* f(x) + f(x + h)

        x += 2 \* h

    return (h / 3) \* res    # метод Симпсона

def RR(F1, F2, h1, h2, p):

    if h1 < h2:

        return F1 + (F1 - F2) / ((h2 / h1) \*\* p - 1)

    return F2 + (F2 - F1) / ((h1 / h2) \*\* p - 1)       # метод Рунге-Ромберга

def main():

    x0 = 0; xk = 2; h1 = 0.5; h2 = 0.25

    p = 2

    p1 = rectangle(x0, xk, y, h1)

    t1 = trapeze(x0, xk, y, h1)

    s1 = Simpson(x0, xk, y, h1)

    p2 = rectangle(x0, xk, y, h2)

    t2 = trapeze(x0, xk, y, h2)

    s2 = Simpson(x0, xk, y, h2)

    rp = RR(p1, p2, h1, h2, p)

    rt = RR(t1, t2, h1, h2, p)

    rs = RR(s1, s2, h1, h2, 4)

    print("Начальные данные:\nX0 = 0, Xk = 2, h1 = 0.5, h2 = 0.25\n")

    print('Для h1 = {}:'.format(h1))

    print("Метод прямоугольников: {}".format(p1))

    print("Метод трапеций: {}".format(t1))

    print("Метод Симпсона: {}".format(s1))

    print('\nДля h2 = {}:'.format(h2))

    print("Метод прямоугольников: {}".format(p2))

    print("Метод трапеций: {}".format(t2))

    print("Метод Симпсона: {}".format(s2))

    print("\nЗначение по методу Рунге-Ромбергу для прямоугольника:", rp)

    print("Погрешность: {0} и {1}".format(abs(rp - p1), abs(rp - p2)))

    print("\nЗначение по методу Рунге-Ромбергу для трапеции:", rt)

    print("Погрешность: {0} и {1}".format(abs(rt - t1), abs(rt - t2)))

    print("\nЗначение по методу Рунге-Ромбергу для Симпсона:", rs)

    print("Погрешность: {0} и {1}".format(abs(rs - s1), abs(rs - s2)))

if \_\_name\_\_ == '\_\_main\_\_':

      main()

**Вывод программы:**

Начальные данные:

X0 = 0, Xk = 2, h1 = 0.5, h2 = 0.25

Для h1 = 0.5:

Метод прямоугольников: 0.1847192474595487

Метод трапеций: 0.18215523215523216

Метод Симпсона: 0.1838992838992839

Для h2 = 0.25:

Метод прямоугольников: 0.18407516757447664

Метод трапеций: 0.1834372398073904

Метод Симпсона: 0.18386457569144316

Значение по методу Рунге-Ромбергу для прямоугольника: 0.18386047427945262

Погрешность: 0.0008587731800960841 и 0.00021469329502402101

Значение по методу Рунге-Ромбергу для трапеции: 0.18386457569144316

Погрешность: 0.0017093435362109943 и 0.0004273358840527486

Значение по методу Рунге-Ромбергу для Симпсона: 0.18386226181092044

Погрешность: 3.702208836345311e-05 и 2.3138805227140846e-06

**Вывод:**

В ходе лабораторной работы были изучены численные методы нахождения определенного интеграла функции с помощью методов прямоугольников, трапеций и Симпсона. Было установлено, что эти методы основаны на аппроксимации функции на заданном интервале при помощи простых геометрических фигур. Были проанализированы достоинства и недостатки каждого метода, в том числе точность вычисления, зависимость от количества разбиений интервала и трудоемкость. Также были разработаны алгоритмы вычисления определенного интеграла с использованием каждого из методов.