**Московский авиационный институт (национальный исследовательский университет)**

Институт информационных технологий и прикладной математики

«Кафедра вычислительной математики и программирования»

**Лабораторная работа по предмету "Численные методы" №3**

Студент: Федоров М.А.

Преподаватель: Ревизников Д.Л.

Группа: М8О-307Б-22

Дата:

Оценка:

Подпись:

Оглавление

[**Задача 1.1** 3](#_Toc192860597)

[**Общий алгоритм решения** 3](#_Toc192860598)

[**Программное решение** 3](#_Toc192860599)

[**Пример работы** 5](#_Toc192860600)

[**Задача 1.2** 7](#_Toc192860601)

[**Общий алгоритм решения** 7](#_Toc192860602)

[**Программное решение** 7](#_Toc192860603)

[**Пример работы для системы из 5-и уравнений** 8](#_Toc192860604)

[**Задача 1.3** 8](#_Toc192860605)

[**Общий алгоритм решения метода итераций** 9](#_Toc192860606)

[**Программное решение** 9](#_Toc192860607)

[**Пример работы** 11](#_Toc192860608)

[**Общий алгоритм решения метода Зейделя** 11](#_Toc192860609)

[**Программное решение** 12](#_Toc192860610)

[**Пример работы** 13](#_Toc192860611)

[**Задача 1.4** 13](#_Toc192860612)

[**Общий алгоритм решения** 14](#_Toc192860613)

[**Программное решение** 15](#_Toc192860614)

[**Пример работы** 16](#_Toc192860615)

[**Задача 1.5** 17](#_Toc192860616)

[**Общий алгоритм решения** 17](#_Toc192860617)

[**Программное решение** 18](#_Toc192860618)

[**Пример работы** 20](#_Toc192860619)

[**Вывод** 21](#_Toc192860620)

# **Задача 3.1**

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

# **Общий алгоритм решения методом Лагранжа**

Наиболее часто в качестве приближающей функции используют многочлены степени n:

Изображение выглядит как Шрифт, белый, число, символ

Контент, сгенерированный ИИ, может содержать ошибки.

Подставляя в данную формулу значения узлов интерполяции и используя условие Pn(xi) = fi получаем систему линейных алгебраических уравнений относительно коэффициентов ai:

Изображение выглядит как Шрифт, текст, число, белый

Контент, сгенерированный ИИ, может содержать ошибки.

Произвольный многочлен может быть записан в виде:

Изображение выглядит как Шрифт, текст, символ, белый

Контент, сгенерированный ИИ, может содержать ошибки.

Здесь li(x) – многочлены степени n, так называемые лагранжевы многочлены

влияния, которые удовлетворяют условию:

Изображение выглядит как Шрифт, текст, линия, белый

Контент, сгенерированный ИИ, может содержать ошибки.

В таком случае li(x) может быть записан в виде:

Изображение выглядит как Шрифт, текст, линия, белый

Контент, сгенерированный ИИ, может содержать ошибки.

Тогда интерполяционный многочлен Лагранжа записывается как:

Изображение выглядит как Шрифт, белый, текст, число

Контент, сгенерированный ИИ, может содержать ошибки.

# **Программное решение**

def lagranj\_interpolation(x\_list: float):

    x = sp.symbols('x', real=True)

    y = 1/(x\*\*2)

    y\_lamb = sp.lambdify(x, y, 'numpy')

    x\_error = 0.8

    print("Исходная функция: y=", y)

    # получаем значения функции y(x) в каждой точке x

    y\_list= list()

    for i in x\_list:

        y\_list.append(y\_lamb(i))

    # составляем интерполяционный многочлен Лагранжа

    L = 0

    for i in range(len(y\_list)):

        li = 1

        for j in range(len(y\_list)):

            if j == i:

                continue

            li \*= (x - x\_list[j])/(x\_list[i] - x\_list[j])

        L += y\_list[i] \* li

        L = sp.simplify(L)

    L\_lamb = sp.lambdify(x, L, 'numpy')

    print("Интерполяционный многочлен Лагранжа имеет вид: L=", L)

    print("Проверим, что полученный многочлен и начальная функция имеют одинаковые значения в определенных точках:")

    for i in x\_list:

        print("При x=", i, ":\ty(x)=", y\_lamb(i), ",\tL(x)=", L\_lamb(i))

    print("Погрешность в точке", x\_error, ": ", np.abs(y\_lamb(x\_error)-L\_lamb(x\_error)))

# **Пример работы**

Исходная функция: y= x\*\*(-2)

Интерполяционный многочлен Лагранжа имеет вид: L= -237.270801373365\*x\*\*3 + 647.264226751406\*x\*\*2 - 554.8045876251\*x + 149.245087296369

Проверим, что полученный многочлен и начальная функция имеют одинаковые значения в определенных точках:

При x= 0.1 : y(x)= 99.99999999999999 , L(x)= 99.9999999999997

При x= 0.5 : y(x)= 4.0 , L(x)= 3.9999999999998863

При x= 0.9 : y(x)= 1.2345679012345678 , L(x)= 1.234567901234783

При x= 1.3 : y(x)= 0.5917159763313609 , L(x)= 0.5917159763322672

Погрешность в точке 0.8 : 3.39462798597404

# **Общий алгоритм решения методом Ньютона**

Введем понятие разделенной разности. Разделенные разности нулевого порядка

совпадают со значениями функции в узлах. Разделенные разности первого порядка обозначаются f(xi, xj) и определяются через разделенные разности нулевого порядка:

Изображение выглядит как Шрифт, линия, типография, каллиграфия

Контент, сгенерированный ИИ, может содержать ошибки.

Разделенные разности второго порядка определяются через разделенные разности первого порядка:

Изображение выглядит как Шрифт, текст, линия, рукописный текст

Контент, сгенерированный ИИ, может содержать ошибки.

Разделенная разность порядка n - k + 2 определяется соотношениями:

Изображение выглядит как текст, Шрифт, линия, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

Пусть известны значения аппроксимируемой функции f(x) в точках x0,x1,…,xn. Интерполяционный многочлен, значения которого в узлах интерполяции совпадают со значениями функции f(x), может быть записан в виде:



# **Программное решение**

def divided\_difference(function, start\_idex, end\_index, x\_list):

    ''' Как считаются разделенные разности:

    # разделенные разности 0-го порядка

    f0 = y\_lamb(x\_list[0])

    f1 = y\_lamb(x\_list[1])

    f2 = y\_lamb(x\_list[2])

    f3 = y\_lamb(x\_list[3])

    # разделенные разности 1-го порядка

    f01 = (f0 - f1)/(x\_list[0] - x\_list[1])

    f12 = (f1 - f2)/(x\_list[1] - x\_list[2])

    f23 = (f2 - f3)/(x\_list[2] - x\_list[3])

    # разделенные разности 2-го порядка

    f02 = (f01 - f12)/(x\_list[0] - x\_list[2])

    f13 = (f12 - f23)/(x\_list[1] - x\_list[3])

    # разделенные разности 3-го порядка

    f03 = (f02 - f13)/(x\_list[0] - x\_list[3])

    '''

    if start\_idex < end\_index:

        return (divided\_difference(function, start\_idex, end\_index-1, x\_list) - divided\_difference(function, start\_idex+1, end\_index, x\_list)) / (x\_list[start\_idex] - x\_list[end\_index])

    elif start\_idex == end\_index:

        return function(x\_list[start\_idex])

def newton\_interpolation(x\_list: float):

    x = sp.symbols('x', real=True)

    y = 1/(x\*\*2)

    y\_lamb = sp.lambdify(x, y, 'numpy')

    x\_error = 0.8

    print("Исходная функция: y=", y)

    # получаем значения функции y(x) в каждой точке x

    y\_list= list()

    for i in x\_list:

        y\_list.append(y\_lamb(i))

    # Пример итогового интерполяционного многочлена Ньютона:

    # P = f0+(x-x\_list[0])\*f01+(x-x\_list[0])\*(x-x\_list[1])\*f02+(x-x\_list[0])\*(x-x\_list[1])\*(x-x\_list[2])\*f03

    # Вычислим интерполяционный многочлен Ньютона

    P = divided\_difference(y\_lamb, 0, 0, x\_list)

    for i in range(1, 4):

        coef = 1

        for j in range(i):

            coef \*= x-x\_list[j]

        P += coef\*divided\_difference(y\_lamb, 0, i, x\_list)

    P = sp.simplify(P)

    P\_lamb = sp.lambdify(x, P, 'numpy')

    print("Интерполяционный многочлен Ньютона имеет вид: P=", P)

    print("Проверим, что полученный многочлен и начальная функция имеют одинаковые значения в определенных точках:")

    for i in x\_list:

        print("При x=", i, ":\ty(x)=", y\_lamb(i), ",\tP(x)=", P\_lamb(i))

    print("Погрешность в точке", x\_error, ": ", np.abs(y\_lamb(x\_error)-P\_lamb(x\_error)))

# **Пример работы**

Исходная функция: y= x\*\*(-2)

Интерполяционный многочлен Ньютона имеет вид: P= -237.270801373365\*x\*\*3 + 647.264226751406\*x\*\*2 - 554.8045876251\*x + 149.245087296369

Проверим, что полученный многочлен и начальная функция имеют одинаковые значения в определенных точках:

При x= 0.1 : y(x)= 99.99999999999999 , P(x)= 99.9999999999997

При x= 0.5 : y(x)= 4.0 , P(x)= 3.9999999999998863

При x= 0.9 : y(x)= 1.2345679012345678 , P(x)= 1.234567901234783

При x= 1.3 : y(x)= 0.5917159763313609 , P(x)= 0.5917159763322672

Погрешность в точке 0.8 : 3.39462798597404

# **Задача 3.2**

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

# **Общий алгоритм решения**

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 аппрокимируемой функции и непрерывен вместе со своими (n-1) производными. Такой кусочно-непрерывный интерполяционный многочлен называется сплайном.

На практике наиболее часто используется интерполяционный многочлен третьей степени, который удобно представить как:

Изображение выглядит как текст, Шрифт, рукописный текст, каллиграфия

Контент, сгенерированный ИИ, может содержать ошибки.

Для построения кубического сплайна необходимо построить n многочленов третьей степени, т. е. определить 4n неизвестных ai, bi, ci, di. Введем обозначение hi = xi – xi-1. Тогда коэффициенты ci можно найти при решении системы с трехдиагональной матрицей:

Изображение выглядит как текст, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Остальные коэффициенты сплайнов могут быть восстановлены по формулам:

Изображение выглядит как Шрифт, текст, рукописный текст, линия

Контент, сгенерированный ИИ, может содержать ошибки.

# **Программное решение**

def cube\_spline():

    xi = [0.1, 0.5, 0.9, 1.3, 1.7]

    fi = [100., 4., 1.2346, 0.59172, 0.34602]

    x\_calculate = 0.8

    n = len(xi)

    ai = np.zeros(n-1)

    bi = np.zeros(n-1)

    ci = np.zeros(n-1)

    di = np.zeros(n-1)

    Matrix = np.zeros((n, n))

    free\_numbers = np.zeros(n)

    h = lambda i: xi[i+1]-xi[i] # возможно тут должен быть abs

    delta = lambda i: (fi[i+1]-fi[i])/h(i)

    free\_numbers[0] = 0

    free\_numbers[n-1] = 0

    Matrix[0,0] = 1

    Matrix[n-1,n-1] = 1

    for i in range(1, n-1):

        Matrix[i, i-1] = h(i-1)

        Matrix[i, i] = 2\*(h(i-1)+h(i))

        Matrix[i, i+1] = h(i)

        free\_numbers[i] = 3\*(delta(i)-delta(i-1))

    ci = np.linalg.solve(Matrix, free\_numbers)

    for i in range(n-1):

        ai[i] = fi[i]

        bi[i] = delta(i)-(h(i)/3)\*(2\*ci[i]+ci[i+1])

        di[i] = (ci[i+1]-ci[i])/(3\*h(i))

    print("a = ", ai, "\nb = ", bi, "\nc = ", ci, "\nd = ", di)

    # считаем значение сплайна в заданной точке

    for i in range(n-1):

        if x\_calculate < xi[i+1]:

            x\_cur = x\_calculate - xi[i]

            print("Значение сплайна в точке x=", x\_calculate, ":\n", ai[i] + bi[i]\*x\_cur + ci[i]\*x\_cur\*\*2 + di[i]\*x\_cur\*\*3)

            break

    # строим плотную сетку и считаем значения сплайна во всех точках

    x\_dense = np.linspace(xi[0], xi[-1], 400)

    y\_dense = np.empty\_like(x\_dense)

    for i in range(len(ai)):

        mask = (x\_dense >= xi[i]) & (x\_dense <= xi[i+1])

        x\_cur = x\_dense[mask] - xi[i]

        y\_dense[mask] = ai[i] + bi[i]\*x\_cur + ci[i]\*x\_cur\*\*2 + di[i]\*x\_cur\*\*3

    # рисуем график

    plt.figure()

    plt.plot(x\_dense, y\_dense)

    plt.scatter(xi, fi)

    plt.xlabel('x')

    plt.ylabel('f(x)')

    plt.title('Natural Cubic Spline Interpolation')

    plt.show()

# **Пример работы**

График:

Изображение выглядит как текст, диаграмма, снимок экрана, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Вывод программы:

a = [100. 4. 1.2346 0.59172]

b = [-302.07259375 -115.8548125 24.75134375 -8.7126625 ]

c = [ 0. 465.54445312 -114.0290625 30.36904688 0. ]

d = [ 387.95371094 -482.97792969 120.33175781 -25.30753906]

Значение сплайна в точке x= 0.8 :

-1.8978470703124959

# **Задача 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.

# **Общий алгоритм решения**

Пусть задана таблично в узлах xj функция yj=f(xj) , j = 0,1,…,N . При этом значения функции yj 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:

Изображение выглядит как Шрифт, типография, текст, белый

Контент, сгенерированный ИИ, может содержать ошибки.

Необходимые условия экстремума имеют вид:

Изображение выглядит как Шрифт, текст, линия, белый

Контент, сгенерированный ИИ, может содержать ошибки.

Эту систему для удобства преобразуют к следующему виду:

Изображение выглядит как Шрифт, текст, белый, типография

Контент, сгенерированный ИИ, может содержать ошибки.

Решив систему, мы получим значения коэффициентов ai, с помощью которых можно построить многочлен Fn(x), приближающий функцию f(x) и минимизирующий квадратичное отклонение.

Стоит заметить, что в некоторых случаях использование многочлена степени n дает большую погрешность. Тогда, как вариант, можно использовать многочлен степени -n, т. е. Fn(x)=a0+a1\*x-1+a2\*x-2+…+an\*x-n. В таком случае система меняется не сильно: достаточно домножить все степени при x на -1.

# **Программное решение**

def mnk(): # для бОльшей точности можно домножить все степени при x на -1. Тогда получим многочлен F(x)=a\_0+a\_1\*x^-1+a\_2\*x^-2+...+a\_n\*x^-n

    xi = [0.1, 0.5, 0.9, 1.3, 1.7, 2.1]

    fi = [100., 4., 1.2346, 0.59172, 0.34602, 0.22676]

    x = sp.symbols('x', real=True)

    n = 2 # здесь задаем степень многочлена

    system\_matrix = np.zeros((n+1, n+1))

    free\_numbers = np.zeros(n+1)

    for i in range(n+1):

        free\_numbers[i] = sum(fi[k]\*xi[k]\*\*(i) for k in range(len(xi))) # можно домножить степень x на -1

        for j in range(n+1):

            system\_matrix[i, j] = sum(elem\*\*(i+j) for elem in xi) # можно домножить степень elem на -1, т. е. elem\*\*(-i-j)

    ai = np.linalg.solve(system\_matrix, free\_numbers)

    F = sum(ai[i]\*x\*\*(i) for i in range(len(ai))) # можно домножить степень x на -1

    F\_lambd = sp.lambdify(x, F, 'numpy')

    print("Приближающий многочлен степени", n, ":\nF(x)=", F)

    error = sum((F\_lambd(xi[i]) - fi[i])\*\*2 for i in range(len(fi)))

    print("Сумма квадратов ошибок:\n", error)

    x\_vals = np.linspace(xi[0], xi[-1], 400)

    y\_vals = F\_lambd(x\_vals)

    # рисуем график

    plt.figure()

    plt.plot(x\_vals, y\_vals)

    plt.scatter(xi, fi)

    plt.xlabel('x')

    plt.ylabel('F(x)')

    plt.title('МНК')

    plt.show()

# **Пример работы**

График при использовании многочленов степени n=2:

Изображение выглядит как текст, снимок экрана, диаграмма, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Вывод программы при использовании многочленов степени n=2:

Приближающий многочлен степени 2 :

F(x)= 54.629743303571\*x\*\*2 - 156.647650982142\*x + 98.4497298080354

Сумма квадратов ошибок:

1556.370861925466

График при использовании многочленов степени n=-2:

Изображение выглядит как текст, диаграмма, линия, График

Контент, сгенерированный ИИ, может содержать ошибки.

Вывод программы при использовании многочленов степени n=-2:

Приближающий многочлен степени 2 :

F(x)= 5.0057953615592e-6 + 3.12209211973074e-6/x + 0.99999963625488/x\*\*2

Сумма квадратов ошибок:

7.55716544923317e-10

# **Задача 3.4**

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

# **Общий алгоритм решения**

Формулы численного дифференцирования в основном используются при

нахождении производных от функции y = f(x), заданной таблично. Исходная функция yi = f(xi), i=0,1,…,M на отрезках [xj, xj+k] заменяется некоторой приближающей, легко вычисляемой функцией:



Где R(x) – остаточный член приближения,  - набор коэффициентов, различный для каждого из рассматриваемых отрезков, и полагают, что:



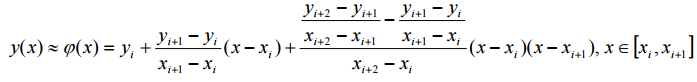
Наиболее часто в качестве приближающей функции берется интерполяционный многочлен:



А производные соответствующих порядков определяются дифференцированием многочлена. При решении практических задач, как правило, используются аппроксимации первых и вторых производных.

При использовании для аппроксимации таблично заданной функции

интерполяционного многочлена второй степени имеем:



Изображение выглядит как текст, Шрифт, линия, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, Шрифт, линия, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

# **Программное решение**

def differentiation():

    xi = [1., 1.2, 1.4, 1.6, 1.8]

    fi = [2., 2.1344, 2.4702, 2.9506, 3.5486]

    x = sp.symbols('x', real=True)

    x\_calculate = 1.4

    l = 0

    for i in range(1, len(xi)):

        if xi[i] > x\_calculate:

            l = i - 1

            break

    print("Границы отрезка: [", xi[l], ";", xi[l+1], "]")

    phi = fi[l] + (fi[l+1]-fi[l])/(xi[l+1]-xi[l])\*(x-xi[l]) + ((fi[l+2]-fi[l+1])/(xi[l+2]-xi[l+1])-(fi[l+1]-fi[l])/(xi[l+1]-xi[l]))/(xi[l+2]-xi[l])\*(x-xi[l])\*(x-xi[l+1])

    phi\_diff\_1 = sp.diff(phi, x)

    phi\_diff\_2 = sp.diff(phi\_diff\_1, x)

    print("Первая производная в точке x=", x\_calculate, ":\n", sp.lambdify(x, phi\_diff\_1)(x\_calculate))

    print("Вторая производная в точке x=", x\_calculate, ":\n", phi\_diff\_2)

# **Пример работы**

Границы отрезка: [ 1.4 ; 1.6 ]

Первая производная в точке x= 1.4 :

2.107999999999994

Вторая производная в точке x= 1.4 :

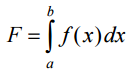
2.94000000000001

# **Задача 3.5**

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

# **Общий алгоритм решения**

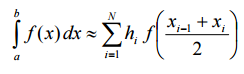
Формулы численного интегрирования используются в тех случаях, когда не удается вычислить аналитически определенный интеграл:



Отрезок [a, b] разбивают точками x0…xN, так что a = x0 ≤ x1 ≤ ... ≤ xN = b с достаточно мелким шагом hi = xi – xi-1 и на одном или нескольких отрезках hi подынтегральную функцию f(x) заменяют такой приближающей ϕ(x) , так что она, во-первых, близка f(x), а, во-вторых, интеграл от ϕ(x) легко вычисляется.

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

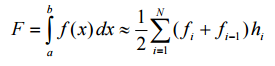
Заменим подынтегральную функцию, интерполяционным многочленом Лагранжа нулевой степени, проходящим через середину отрезка – точку xi-1/2 = (xi-1+xi)/2, получим формулу прямоугольников:



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:

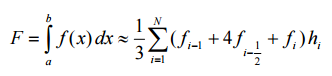


Эта формула носит название формулы трапеций.

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:

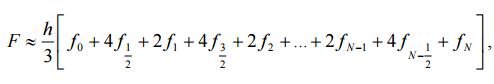


Для случая hi = (xi – xi-1)/2 получим формулу Симпсона (парабол):



В случае постоянного шага интегрирования hi = h, i=1,2,…,N формула Симпсона

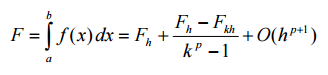
принимает вид:



при этом количество интервалов, на которое делится отрезок интегрирования, равно 2N.

Метод Рунге-Ромберга-Ричардсона позволяет получать более высокий порядок

точности вычисления. Если имеются результаты вычисления определенного интеграла на сетке с шагом h – F = Fh + O(hP) и на сетке с шагом kh – F = Fkh + O((kh)P), то:



Где О(hp+1) – остаточный член

# **Программное решение**

def rectangle\_integrate():

    x = sp.symbols('x', real=True)

    f = sp.sqrt(16-x\*\*2)

    f\_lambd = sp.lambdify(x, f, 'numpy')

    x\_0 = -2

    x\_k = 2

    h\_1 = 1.

    h\_2 = 0.5

    p = 2 # порядок аппроксимации

    N\_1 = np.abs(x\_k - x\_0) // h\_1

    N\_2 = np.abs(x\_k - x\_0) // h\_2

    xi\_1 = [x\_0+h\_1\*i for i in range(0, int(N\_1)+1)]

    xi\_2 = [x\_0+h\_2\*i for i in range(0, int(N\_2)+1)]

    print("Концы отрезков с шагом", h\_1, ":\n", xi\_1)

    print("Концы отрезков с шагом", h\_2, ":\n", xi\_2)

    result\_1 = sum(h\_1\*f\_lambd((xi\_1[i-1]+xi\_1[i])/2) for i in range(1, len(xi\_1)))

    result\_2 = sum(h\_2\*f\_lambd((xi\_2[i-1]+xi\_2[i])/2) for i in range(1, len(xi\_2)))

    print("Результат интегрирования формулой прямоугольников с шагом", h\_1, ":\n", result\_1)

    print("Результат интегрирования формулой прямоугольников с шагом", h\_2, ":\n", result\_2)

    runge\_romberg = result\_2 + (result\_2 - result\_1) / (2\*\*p - 1)

    print("Результат интегрирования после уточнения методом Рунге-Ромберга:\n", runge\_romberg)

    print("Разница между вычислениями:\n", np.abs(result\_2 - runge\_romberg))

def trapeze\_integrate():

    x = sp.symbols('x', real=True)

    f = sp.sqrt(16-x\*\*2)

    f\_lambd = sp.lambdify(x, f, 'numpy')

    x\_0 = -2

    x\_k = 2

    h\_1 = 1.

    h\_2 = 0.5

    p = 2 # порядок аппроксимации

    N\_1 = np.abs(x\_k - x\_0) // h\_1

    N\_2 = np.abs(x\_k - x\_0) // h\_2

    xi\_1 = [x\_0+h\_1\*i for i in range(0, int(N\_1)+1)]

    xi\_2 = [x\_0+h\_2\*i for i in range(0, int(N\_2)+1)]

    print("Концы отрезков с шагом", h\_1, ":\n", xi\_1)

    print("Концы отрезков с шагом", h\_2, ":\n", xi\_2)

    result\_1 = 0.5\*sum(h\_1\*(f\_lambd(xi\_1[i]) + f\_lambd(xi\_1[i-1])) for i in range(1, len(xi\_1)))

    result\_2 = 0.5\*sum(h\_2\*(f\_lambd(xi\_2[i]) + f\_lambd(xi\_2[i-1])) for i in range(1, len(xi\_2)))

    print("Результат интегрирования формулой трапеции с шагом", h\_1, ":\n", result\_1)

    print("Результат интегрирования формулой трапеции с шагом", h\_2, ":\n", result\_2)

    runge\_romberg = result\_2 + (result\_2 - result\_1) / (2\*\*p - 1)

    print("Результат интегрирования после уточнения методом Рунге-Ромберга:\n", runge\_romberg)

    print("Разница между вычислениями:\n", np.abs(result\_2 - runge\_romberg))

def simpson\_integrate():

    x = sp.symbols('x', real=True)

    f = sp.sqrt(16-x\*\*2)

    f\_lambd = sp.lambdify(x, f, 'numpy')

    x\_0 = -2

    x\_k = 2

    h\_1 = 1.

    h\_2 = 0.5

    p = 4 # порядок аппроксимации

    N\_1 = np.abs(x\_k - x\_0) // h\_1

    N\_2 = np.abs(x\_k - x\_0) // h\_2

    xi\_1 = [x\_0+h\_1\*i for i in range(0, int(N\_1)+1)]

    xi\_2 = [x\_0+h\_2\*i for i in range(0, int(N\_2)+1)]

    print("Концы отрезков с шагом", h\_1, ":\n", xi\_1)

    print("Концы отрезков с шагом", h\_2, ":\n", xi\_2)

    result\_1 = (h\_1/3)\*sum(f\_lambd(xi\_1[i-1])+4\*f\_lambd(xi\_1[i])+f\_lambd(xi\_1[i+1]) for i in range(1, len(xi\_1)-1, 2))

    result\_2 = (h\_2/3)\*sum(f\_lambd(xi\_2[i-1])+4\*f\_lambd(xi\_2[i])+f\_lambd(xi\_2[i+1]) for i in range(1, len(xi\_2)-1, 2))

    print("Результат интегрирования формулой Симпсона с шагом", h\_1, ":\n", result\_1)

    print("Результат интегрирования формулой Симпсона с шагом", h\_2, ":\n", result\_2)

    runge\_romberg = result\_2 + (result\_2 - result\_1) / (2\*\*p - 1)

    print("Результат интегрирования после уточнения методом Рунге-Ромберга:\n", runge\_romberg)

    print("Разница между вычислениями:\n", np.abs(result\_2 - runge\_romberg))

# **Пример работы**

Концы отрезков с шагом 1.0 :

[-2.0, -1.0, 0.0, 1.0, 2.0]

Концы отрезков с шагом 0.5 :

[-2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0]

Результат интегрирования формулой прямоугольников с шагом 1.0 :

15.353452420289436

Результат интегрирования формулой прямоугольников с шагом 0.5 :

15.317782947920461

Результат интегрирования после уточнения методом Рунге-Ромберга:

15.30589312379747

Разница между вычислениями:

0.01188982412299211

Концы отрезков с шагом 1.0 :

[-2.0, -1.0, 0.0, 1.0, 2.0]

Концы отрезков с шагом 0.5 :

[-2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0]

Результат интегрирования формулой трапеции с шагом 1.0 :

15.21006830755259

Результат интегрирования формулой трапеции с шагом 0.5 :

15.281760363921011

Результат интегрирования после уточнения методом Рунге-Ромберга:

15.305657716043818

Разница между вычислениями:

0.02389735212280719

Концы отрезков с шагом 1.0 :

[-2.0, -1.0, 0.0, 1.0, 2.0]

Концы отрезков с шагом 0.5 :

[-2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0]

Результат интегрирования формулой Симпсона с шагом 1.0 :

15.304023333311614

Результат интегрирования формулой Симпсона с шагом 0.5 :

15.30565771604382

Результат интегрирования после уточнения методом Рунге-Ромберга:

15.305766674892634

Разница между вычислениями:

0.00010895884881456652

# **Вывод**

В ходе выполнения лабораторной работы я научился решать задачи интерполяции и считать приближенные значения некоторых функций.