МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«Национальный исследовательский университет ИТМО»

ФАКУЛЬТЕТ ПРОГРАММНОЙ ИНЖЕНЕРИИ И КОМПЬЮТЕРНОЙ ТЕХНИКИ

**ЛАБОРАТОРНАЯ РАБОТА №5**

по дисциплине

«ВЫЧИСЛИТЕЛЬНАЯ МАТЕМАТИКА»

Вариант № 13

***Выполнил:***

Студент группы P3218

Рамеев Тимур

Ильгизович

***Преподаватель:***

Бострикова Дарья

Константиновна

# Содержание

[Содержание 2](#_Toc177337947)

[Цель работы 3](#_Toc177337948)

[Используемые методы 3](#_Toc177337949)

[Многочлен Лагранжа 3](#_Toc177337950)

[Многочлен Ньютона с разделенными разностями 3](#_Toc177337951)

[Многочлен Ньютона с конечными разностями 4](#_Toc177337952)

[Многочлен Гаусса 5](#_Toc177337953)

[Вычислительная часть 6](#_Toc177337954)

[Интерполяционная формула Ньютона 6](#_Toc177337955)

[Интерполяционная формула Гаусса 6](#_Toc177337956)

[Реализация и пример работы 7](#_Toc177337957)

[Вывод 8](#_Toc177337958)

# Цель работы

Цель работы: решение задачи интерполяции, нахождение значения функции при заданных значениях аргумента, отличных от узловых точек

# Используемые методы

## Многочлен Лагранжа

Наиболее простой способ получения интерполяционного многочлена – многочлен Лагранжа.

Главным минусом этого метода является необходимость в пересчете всех коэффициентов при добавлении новых узлов интерполяции.

## Многочлен Ньютона с разделенными разностями

Многочлен Ньютона с разделенными разностями используется при интерполяции функции на неравномерных сетках.

В определении многочлена используются разделенные разности. Разделенная разность k-го порядка:

При этом разделенная разность 0-го порядка определяется как:

Наконец, непосредственно многочлен Ньютона:

## Многочлен Ньютона с конечными разностями

При наличии равномерной сетки можно использовать многочлен Ньютона с конечными разностями.

Конечная разность k-го порядка:

При этом конечная разность 0-го порядка определяется как:

В зависимости от положения аргумента функции в сетке используют одну из 2-х интерполяционных формул Ньютона:

Если x расположен в левой половине сетки, то используется 1-я интерполяционная формула (вперед):

, где

– шаг сетки

При этом если , товместо можно взять .

Если расположен в правой половине сетки, то используется 2-я интерполяционная формула (назад):

, где

– шаг сетки

При этом если , товместо можно взять .

## Многочлен Гаусса

В случаях, когда аргумент расположен близко к середине сетки (назовем ее ), следует использовать многочлен Гаусса

Если :

Если :

, где

– шаг сетки

# Вычислительная часть

## Интерполяционная формула Ньютона

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| № |  |  |  |  |  |  |  |  |
| 0 | 1,10 | 0,2234 | 1,0204 | 0,0002 | 0,0132 | -0,0368 | 0,0762 | -0,1313 |
| 1 | 1,25 | 1,2438 | 1,0206 | 0,0134 | -0,0236 | 0,0394 | -0,0551 |  |
| 2 | 1,40 | 2,2644 | 1,0340 | -0,0102 | 0,0158 | -0,0157 |  |  |
| 3 | 1,55 | 3,2984 | 1,0238 | 0,0056 | 0,0001 |  |  |  |
| 4 | 1,70 | 4,3222 | 1,0294 | 0,0057 |  |  |  |  |
| 5 | 1,85 | 5,3516 | 1,0351 |  |  |  |  |  |
| 6 | 2,00 | 6,3867 |  |  |  |  |  |  |

Функция задана равномерной сеткой с шагом

Требуется вычислить значение функции в точке

Так как воспользуемся первой интерполяционной формулой Ньютона для интерполирования вперед:

## Интерполяционная формула Гаусса

Требуется вычислить значение функции в точке

Так как ( – центральная точка ) воспользуемся второй интерполяционной формулой Гаусса:

# Реализация и пример работы

from math import sin, cos

from backend.Lab5.utils import StatusInterpol, StatusMethod

import numpy as np

import json

*def* find\_interpolation\_type1(*argument*, *x\_values*, *y\_values*):

*# Некая валидация на сервере*

    n = len(x\_values)

    try:

        local\_x\_values = []

        local\_y\_values = []

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

            if x\_values[i] == "" and y\_values[i] == "":

                n -= 1

            if x\_values[i] != "" and y\_values[i] != "":

                local\_x\_values.append(*float*(x\_values[i].replace(',', '.')))

                local\_y\_values.append(*float*(y\_values[i].replace(',', '.')))

        argument = *float*(argument.replace(',', '.'))

    except *ValueError*:

        return { "status" : json.dumps(StatusInterpol.ERROR), "error" : "Некорректный формат данных" }

    except (*IndexError*, *TypeError*):

        return { "status" : json.dumps(StatusInterpol.ERROR), "error" : "Количество X координат и Y координат должно быть одинаковым"}

    if len(local\_x\_values) < 2:

        return { "status" : json.dumps(StatusInterpol.ERROR), "error" : "Количество точек для интерполяции должно быть больше 2"}

    if len(local\_x\_values) != len(local\_y\_values) or len(local\_x\_values) != n:

        print(x\_values, y\_values)

        return { "status" : json.dumps(StatusInterpol.ERROR), "error" : "Количество X координат и Y координат должно быть одинаковым"}

    if argument in local\_x\_values:

        return { "status" : json.dumps(StatusInterpol.ERROR), "error" : "Подскажи, пожалуйста, деловая колбаса, а зачем мне считать значение в узле интерполяции???"}

*# Сортировка точек*

    dict\_points = *dict*(sorted(*dict*(zip(local\_x\_values, local\_y\_values)).items()))

    if len(dict\_points) != len(local\_x\_values):

        return { "status" : json.dumps(StatusInterpol.ERROR), "error" : "Координата X должна быть уникальной, братишка"}

    points = (np.array(*list*(dict\_points.keys())), *list*(dict\_points.values()))

    return {

        "status" : json.dumps(StatusInterpol.FIRST\_TYPE\_OK),

        "lagrange\_result" : lagrange\_method(points, argument),

        "newton\_separated\_result" : newton\_separated\_differences\_method(points, argument),

        "newton\_finite\_result" : newton\_finite\_differences\_method(points, argument),

        "interpol\_points" : {"x\_values" : points[0].tolist(), "y\_values" : points[1]}

    }

*def* find\_interpolation\_type2(*argument*, *number\_of\_equation*, *left\_border*, *right\_border*, *number\_of\_points*):

*# Некая валидация на сервере*

    try:

        left\_border = *float*(left\_border.replace(',', '.'))

        right\_border = *float*(right\_border.replace(',', '.'))

        argument = *float*(argument.replace(',', '.'))

        number\_of\_points = *int*(number\_of\_points)

        number\_of\_equation = *int*(number\_of\_equation)

    except *ValueError*:

        return { "status" : json.dumps(StatusInterpol.ERROR), "error" : "Некорректный формат данных" }

    if number\_of\_points < 2:

        return { "status" : json.dumps(StatusInterpol.ERROR), "error" : "Количество точек для интерполяции должно быть больше 2"}

    points = get\_points(number\_of\_equation, left\_border, right\_border, number\_of\_points)

    return {

        "status" : json.dumps(StatusInterpol.SECOND\_TYPE\_OK),

        "lagrange\_result" : lagrange\_method(points, argument),

        "newton\_separated\_result" : newton\_separated\_differences\_method(points, argument),

        "newton\_finite\_result" : newton\_finite\_differences\_method(points, argument),

        "true\_value" : find\_true\_value(argument, number\_of\_equation, points),

        "interpol\_points" : {"x\_values" : points[0].tolist(), "y\_values" : points[1]}

    }

*def* lagrange\_method(*points*, *argument*):

    x\_values, y\_values = points

*# Получаем коэффициенты c\_i*

    c\_values = *list*()

    for i in range(len(x\_values)):

        current\_c = y\_values[i]

        for j in range(len(x\_values)):

            if j != i:

                current\_c /= x\_values[i] - x\_values[j]

        c\_values.append(current\_c)

*# Получаем конкретные значения аппроксимируещей функции для графика*

    graph\_x\_values = find\_graph\_points(x\_values)

    graph\_y\_values = *list*()

    for i in range(len(graph\_x\_values)):

        current\_y = 0

        for j in range(len(c\_values)):

            current\_summand = c\_values[j]

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

                if j != k:

                    current\_summand \*= (graph\_x\_values[i] - x\_values[k])

            current\_y += current\_summand

        graph\_y\_values.append(current\_y)

*# Поиск значения аппроксимирующей функции в заданной точке*

    argument\_value = 0

    for j in range(len(c\_values)):

        current\_summand = c\_values[j]

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

            if j != k:

                current\_summand \*= (argument - x\_values[k])

        argument\_value += current\_summand

    return {"status" : json.dumps(StatusMethod.OK), "x\_values" : graph\_x\_values, "y\_values" : graph\_y\_values, "argument\_value" : argument\_value }

*def* newton\_separated\_differences\_method(*points*, *argument*):

    x\_values, \_ = points

    f\_values = find\_separate\_differences(points)

*# Получаем конкретные значения аппроксимируещей функции для графика*

    graph\_x\_values = find\_graph\_points(x\_values)

    graph\_y\_values = *list*()

    for i in range(len(graph\_x\_values)):

        current\_y\_value = f\_values[0]

        for k in range(1, len(f\_values)):

            multiplier = 1

            for j in range(k):

                multiplier \*= graph\_x\_values[i] - x\_values[j]

            current\_y\_value += f\_values[k] \* multiplier

        graph\_y\_values.append(current\_y\_value)

*# Поиск значения аппроксимирующей функции в заданной точке*

    argument\_value = f\_values[0]

    for k in range(1, len(f\_values)):

        multiplier = 1

        for j in range(k):

            multiplier \*= argument - x\_values[j]

        argument\_value += f\_values[k] \* multiplier

    return {"status" : json.dumps(StatusMethod.OK), "x\_values" : graph\_x\_values, "y\_values" : graph\_y\_values, "argument\_value" : argument\_value }

*def* newton\_finite\_differences\_method(*points*, *argument*):

    x\_values, \_ = points

*# Проверка возможности применить метод*

    h = x\_values[1] - x\_values[0]

    for i in range(1, len(x\_values)):

        if x\_values[i] - x\_values[i - 1] - h > 0.000001:

            return {"status" : json.dumps(StatusMethod.ERROR), "error" : "Метод применим только для функций заданных на равномерной сетке"}

    diffs, diffs\_with\_x = find\_finite\_differences(points)

*# Получаем конкретные значения аппроксимируещей функции для графика*

    graph\_x\_values = find\_graph\_points(x\_values)

    graph\_y\_values = *list*()

    middle = (x\_values.max() + x\_values.min()) / 2

    for x in graph\_x\_values:

        graph\_y\_values.append(find\_newton\_func\_value(x, x\_values, middle, diffs))

*# Поиск значения аппроксимирующей функции в заданной точке*

    argument\_value = find\_newton\_func\_value(argument, x\_values, middle, diffs)

    return {"status" : json.dumps(StatusMethod.OK), "x\_values" : graph\_x\_values, "y\_values" : graph\_y\_values, "argument\_value" : argument\_value, "diffs\_table" : [[round(x, 2) for x in rows] for rows in diffs\_with\_x]}

*# Поиск разделенных разностей порядка len(points)*

*def* find\_separate\_differences(*points*):

    x\_values, y\_values = points

    n = len(y\_values)

    diffs = np.zeros((n, n))

    diffs[0] = y\_values

    for i in range(1, n + 1):

        for j in range(n - i):

            diffs[i][j] = (diffs[i - 1][j + 1] - diffs[i - 1][j]) / (x\_values[j + i] - x\_values[j])

    f\_values = np.ravel(np.compress([1], diffs, *axis*=1))

    return f\_values

*# Поиск конечных разностей порядка len(points)*

*def* find\_finite\_differences(*points*):

    x\_values, y\_values = points

    n = len(y\_values)

    diffs = np.zeros((n, n))

    diffs[0] = y\_values

    for i in range(1, n):

        for j in range(n - i):

            diffs[i][j] = diffs[i - 1][j + 1] - diffs[i - 1][j]

    diffs\_with\_x = np.append([x\_values], diffs, *axis*=0)

    return np.transpose(diffs), np.transpose(diffs\_with\_x)

*# Находит значение формулы Ньютона, принимая на вход таблицу конечных разностей*

*def* find\_newton\_func\_value(*x*, *interpol\_values*, *middle\_point*, *diffs*):

    x\_start = 0

    diffs\_i = *list*()

    h = interpol\_values[1] - interpol\_values[0]

    if x < middle\_point:

*# Первая формула Ньютона*

        for i in range(1, len(interpol\_values)):

            if x < interpol\_values[i]:

                x\_start = interpol\_values[i - 1]

                diffs\_i = diffs[i - 1]

                break

        t = (x - x\_start) / h

        t\_multipliesrs = np.append(1, find\_t\_multipliers(t, len(diffs\_i), *type\_*=1))

    else:

*# Вторая формула Ньютона*

        for i in range(len(interpol\_values) - 2, -1, -1):

            if x > interpol\_values[i]:

                x\_start = interpol\_values[i + 1]

                diffs\_i = find\_diffs\_i\_back\_newton(i + 1, diffs)

                break

        t = (x - x\_start) / h

        t\_multipliesrs = np.append(1, find\_t\_multipliers(t, len(diffs\_i), *type\_*=2))

    return np.sum(t\_multipliesrs \* diffs\_i)

*# Поиск разностей в таблице разностей для интерполирования назад (Метод Ньютона для равноудаленных узлов)*

*def* find\_diffs\_i\_back\_newton(*slice\_number*, *diffs*):

    ret\_arr = np.ndarray(0)

    for i in range(slice\_number, -1, -1):

        ret\_arr = np.append(ret\_arr, diffs[i][slice\_number - i])

    return ret\_arr

*# Высчитывает дробь - t-множитель, принимая на вход само значение t (Метод Ньютона для равноудаленных узлов)*

*def* find\_t\_multipliers(*t*, *len\_*, *type\_*):

    ret\_arr = np.ndarray(0)

    for i in range(1, len\_):

        top = t

        bottom = 1

        for j in range(1, i):

            bottom \*= (j + 1)

            if type\_ == 1:

*# Первая формула Ньютона*

                top \*= (t - j)

            else:

*# Вторая формула Ньютона*

                top \*= (t + j)

        ret\_arr = np.append(ret\_arr, top / bottom)

    return ret\_arr

*def* find\_true\_value(*argument*, *number\_of\_equation*, *points*):

    x\_values, \_ = points

    method = define\_method(number\_of\_equation)

    argument\_value = method(argument)

    graph\_x\_values = find\_graph\_points(x\_values)

    graph\_y\_values = *list*()

    for x in graph\_x\_values:

        graph\_y\_values.append(method(x))

    return {"status" : json.dumps(StatusMethod.OK), "x\_values" : graph\_x\_values, "y\_values" : graph\_y\_values, "argument\_value" : argument\_value }

*# Возвращает начальный набор точек для интерполирования функций*

*def* get\_points(*number\_of\_equation*, *left\_border*, *right\_border*, *number\_of\_points*):

    find\_y = define\_method(number\_of\_equation)

    x\_values = np.linspace(left\_border, right\_border, number\_of\_points)

    y\_values = [find\_y(x) for x in x\_values]

    return x\_values, y\_values

*# Возвращает X\_points для отправки на фронт для графика*

*def* find\_graph\_points(*x\_values*):

    left\_border = min(x\_values)

    right\_border = max(x\_values)

    graph\_left\_border = left\_border - (left\_border + right\_border) / 4

    graph\_right\_border = right\_border + (left\_border + right\_border) / 4

    return np.linspace(graph\_left\_border, graph\_right\_border, 1000).tolist()

*def* first\_equation(*x*):

    return 2 \* x - 3

*def* second\_equation(*x*):

    return x\*\*3 + 4.81 \* x\*\*2 - 17.37 \* x + 5.38

*def* third\_equation(*x*):

    return 5 \* sin(x) - 2 \* cos(x)

*def* define\_method(*number\_of\_equation*):

    if  number\_of\_equation == 1:

        return first\_equation

    elif  number\_of\_equation == 2:

        return second\_equation

    elif  number\_of\_equation == 3:

        return third\_equation

    else:

        return None

# Вывод

Мы изучили несколько методов интерполяции функций - интерполяция многочленом Лагранжа, Ньютона, Гаусса – с их помощью интерполировали функцию по нескольким точкам. Также мы разработали программную реализацию этих методов