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 №6**

по дисциплине

«ВЫЧИСЛИТЕЛЬНАЯ МАТЕМАТИКА»

Вариант № 13

***Выполнил:***

Студент группы P3218

Рамеев Тимур

Ильгизович

***Преподаватель:***

Бострикова Дарья

Константиновна

# Содержание

[Содержание 2](#_Toc178189826)

[Цель работы 3](#_Toc178189827)

[Используемые методы 3](#_Toc178189828)

[Метод Эйлера 3](#_Toc178189829)

[Усовершенствованный метод Эйлера 3](#_Toc178189830)

[Метод Милна 3](#_Toc178189831)

[Реализация и пример работы 4](#_Toc178189832)

[Вывод 5](#_Toc178189833)

# Цель работы

Решить задачу Коши для обыкновенных дифференциальных уравнений численными методами.

# Используемые методы

Представим решаемое уравнение в виде и рассмотрим методы его вычисления.

## Метод Эйлера

Наиболее простой способ одношагового численного решения обыкновенных дифференциальных уравнений – метод Эйлера:

Фактически он основан на приближении следующего по итерации значения функции суммой прошлого значения и дифференциала в прошлой точке

Порядок точности:

## Усовершенствованный метод Эйлера

Усовершенствуем метод Эйлера рассмотрением дифференциала не только в прошлой точке, но и в текущей приближаемой. Взяв среднее арифметическое от них, мы повысим точность приближения.

Порядок точности:

## Метод Милна

Рассмотрим также один из многошаговых методов прогноза и коррекции – метод Милна. В этом методе значение функции в одной и той же точке вычисляется несколько раз. Изначально вычисляется прогноз , а затем он корректируется последовательным вычислением . корректирование прекращается, когда разница между последними 2-я значениями становится удовлетворительно мала.

Предиктор:

Корректор:

Заметим, что для использования метода Милна нужно иметь первые 4 вычисленных значения функции. Их можно получить одношаговыми методами.

Порядок точности:

# Реализация и пример работы

Программа, реализующая решение обыкновенных дифференциальных уравнений методом Эйлера, модифицированным методом Эйлера и метод Милна была написана на языке Python.

Исходный код:

import numpy as np

import json

from backend.Lab6.utils import Method, StatusMethod, FindAccYValueRetCode

*# from utils import Method, StatusMethod, FindAccYValueRetCode*

from math import sin, e, pow, cos, isnan

ONE\_STEP\_CONSTRAINT = 20

MILN\_CONSTRAINT = 30

MAX\_POINTS = 200

CAN\_NOT\_COMPUTE\_OTHER\_POINTS\_MESSAGE = "Предупреждение: Остальные точки посчитать не представляется возможным!"

FAILED\_TO\_MEET\_ACCURACY\_MESSAGE = "Предупреждение: Я не могу достигнуть нужной точности!"

OK\_MESSAGE = "Метод отработал корректно!"

*def* solve\_differential\_equation(*accuracy*, *y\_value*, *h\_value*, *left\_border*, *right\_border*, *equation\_number*):

*# Некая валидация на сервере*

    try:

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

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

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

        h\_value = *float*(h\_value.replace(',', '.'))

        y\_value = *float*(y\_value.replace(',', '.'))

        equation\_number = *int*(equation\_number)

    except *ValueError*:

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

    if left\_border >= right\_border:

        return { "status" : json.dumps(StatusMethod.ERROR), "error" : "Левая граница интервала должна быть меньше правой"}

    if (right\_border - left\_border) / h\_value > MAX\_POINTS:

        return { "status" : json.dumps(StatusMethod.ERROR), "error" : "Слишком маленький шаг для такого промежутка, пожалуйста, поменять входные данные"}

    if accuracy < 0 or accuracy > 0.1:

        return { "status" : json.dumps(StatusMethod.ERROR), "error" : "Точность не соотвествует заданному условию, пожалуйста, убедитесь, что она больше 0 и меньше 0.1" }

    if not equation\_number in [1, 2, 3]:

        return { "status" : json.dumps(StatusMethod.ERROR), "error" : "Некорректный номер уравнения, проверьте еще раз"}

    if right\_border - left\_border < h\_value:

        return { "status" : json.dumps(StatusMethod.ERROR), "error" : "Шаг не должен быть больше интервала"}

    return {

        "status" : json.dumps(StatusMethod.OK),

        "euler\_result" : one\_step\_method(accuracy, y\_value, h\_value, left\_border, right\_border, equation\_number, Method.EULER),

        "euler\_improved\_result" : one\_step\_method(accuracy, y\_value, h\_value, left\_border, right\_border, equation\_number, Method.EULER\_IMPROVED),

        "miln\_result" : many\_step\_method(accuracy, y\_value, h\_value, left\_border, right\_border, equation\_number, Method.MILN),

        "true\_result" : find\_true\_values(left\_border, right\_border, y\_value, h\_value, equation\_number)

    }

*# Основонй метод*

*def* one\_step\_method(*accuracy*, *y\_value*, *h\_value*, *left*, *right*, *equation\_number*, *method\_number*):

    message = OK\_MESSAGE

    x\_values = [left]

    y\_values = [y\_value]

    current\_h\_value = h\_value

    find\_f\_value = define\_equation(equation\_number)

    counter = 1

    if method\_number == Method.EULER:

        method = euler\_method

        p\_value = 1

    elif method\_number == Method.EULER\_IMPROVED:

        method = euler\_improved\_method

        p\_value = 2

    else:

        return None

    while x\_values[-1] + accuracy \* 0.01 < right:

        current\_y\_value, current\_h\_value, current\_counter, ret\_code = find\_accurate\_y\_value(find\_f\_value, left, right, x\_values[-1] + current\_h\_value, y\_value, current\_h\_value, accuracy, counter, method, p\_value)

        if ret\_code == FindAccYValueRetCode.TOO\_LOW\_H\_VALUE or ret\_code == FindAccYValueRetCode.TOO\_MUCH\_ITERATIONS:

            message =  FAILED\_TO\_MEET\_ACCURACY\_MESSAGE

        if isnan(current\_y\_value) or current\_y\_value == *float*("inf") or current\_y\_value == *float*("-inf"):

            message = CAN\_NOT\_COMPUTE\_OTHER\_POINTS\_MESSAGE

            break

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

        x\_values.append(x\_values[-1] + current\_h\_value)

        counter = current\_counter

    x\_values\_noda, y\_values\_noda = find\_noda\_y(x\_values, y\_values, h\_value)

    return {

        "x\_values" : [] if message == CAN\_NOT\_COMPUTE\_OTHER\_POINTS\_MESSAGE else x\_values,

        "y\_values" : [] if message == CAN\_NOT\_COMPUTE\_OTHER\_POINTS\_MESSAGE else y\_values,

        "h\_value" : current\_h\_value,

        "message" : message,

        "x\_values\_noda" : x\_values\_noda,

        "y\_values\_noda" : y\_values\_noda

    }

*def* many\_step\_method(*accuracy*, *y\_value*, *h\_value*, *left*, *right*, *equation\_number*, *method\_number*):

    message = OK\_MESSAGE

    find\_f\_value = define\_equation(equation\_number)

    if method\_number == Method.MILN:

        method = miln\_method

    else:

        return None

    current\_step = h\_value

    initial\_values = init\_y\_values\_via\_one\_step\_method(find\_f\_value, left, y\_value, current\_step, 4, Method.EULER\_IMPROVED)

    method\_x\_values, method\_y\_values = method(find\_f\_value, accuracy, current\_step, left, right, initial\_values)

    true\_y\_values = find\_true\_values(left, right, y\_value, current\_step, equation\_number)["y\_values\_noda"]

    if len(method\_y\_values) == len(true\_y\_values):

        while not(check\_accuracy\_many\_step\_methods(method\_y\_values, true\_y\_values, accuracy)) and len(method\_y\_values) < MAX\_POINTS:

            current\_step /= 2

            initial\_values = init\_y\_values\_via\_one\_step\_method(find\_f\_value, left, y\_value, current\_step, 4, Method.EULER\_IMPROVED)

            method\_x\_values, method\_y\_values = method(find\_f\_value, accuracy, current\_step, left, right, initial\_values)

            true\_y\_values = find\_true\_values(left, right, y\_value, current\_step, equation\_number)["y\_values\_noda"]

            if len(method\_y\_values) != len(true\_y\_values):

                message = CAN\_NOT\_COMPUTE\_OTHER\_POINTS\_MESSAGE

                break

    else:

        message = CAN\_NOT\_COMPUTE\_OTHER\_POINTS\_MESSAGE

    x\_values\_noda, y\_values\_noda = find\_noda\_y(method\_x\_values, method\_y\_values, h\_value)

    return {

        "x\_values" : [] if message == CAN\_NOT\_COMPUTE\_OTHER\_POINTS\_MESSAGE else method\_x\_values,

        "y\_values" : [] if message == CAN\_NOT\_COMPUTE\_OTHER\_POINTS\_MESSAGE else method\_y\_values,

        "x\_values\_noda" : x\_values\_noda,

        "y\_values\_noda" : y\_values\_noda,

        "h\_value" : current\_step,

        "message" : message

    }

*def* init\_y\_values\_via\_one\_step\_method(*find\_f\_value*, *left*, *y\_value*, *h\_value*, *size\_of\_init\_arrray*, *type\_of\_method*):

    if type\_of\_method == Method.EULER:

        method = euler\_method

    elif type\_of\_method == Method.EULER\_IMPROVED:

        method = euler\_improved\_method

    else:

        return None

    ret\_arr = [y\_value]

    for i in range(size\_of\_init\_arrray - 1):

        ret\_arr.append(method(find\_f\_value, left, left + h\_value \* (i + 1), y\_value, h\_value))

    return ret\_arr

*def* check\_accuracy\_many\_step\_methods(*point\_set\_1*, *point\_set\_2*, *accuracy*):

    for i in range(len(point\_set\_1)):

        if abs(point\_set\_1[i] - point\_set\_2[i]) > accuracy:

            return False

    return True

*def* euler\_method(*find\_f\_value*, *left*, *right*, *y\_value*, *h\_value*):

    x\_values = np.arange(left, right + h\_value \* 0.01, h\_value)

    previous\_y\_value = y\_value

    for x\_value in x\_values[:-1]:

        previous\_y\_value = previous\_y\_value + h\_value \* find\_f\_value(x\_value, previous\_y\_value)

    return previous\_y\_value

*def* euler\_improved\_method(*find\_f\_value*, *left*, *right*, *y\_value*, *h\_value*):

    x\_values = np.arange(left, right + h\_value \* 0.01, h\_value)

    previous\_y\_value = y\_value

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

        previous\_y\_value = previous\_y\_value + h\_value / 2 \* (find\_f\_value(x\_values[i], previous\_y\_value) + find\_f\_value(x\_values[i + 1], previous\_y\_value + h\_value \* find\_f\_value(x\_values[i], previous\_y\_value)))

    return previous\_y\_value

*def* find\_accurate\_y\_value(*equation*, *left*, *right*, *current\_right*, *y\_value*, *h\_value*, *accuracy*, *counter*, *find\_y\_i*, *p\_value*):

    y\_first = find\_y\_i(equation, left, current\_right, y\_value, h\_value)

    if (right - left) / h\_value > MAX\_POINTS:

        return y\_first, h\_value, counter, FindAccYValueRetCode.TOO\_LOW\_H\_VALUE

    if counter >= ONE\_STEP\_CONSTRAINT:

        return y\_first, h\_value, counter, FindAccYValueRetCode.TOO\_MUCH\_ITERATIONS

    y\_second = find\_y\_i(equation, left, current\_right, y\_value, h\_value / 2)

    if check\_accuracy\_one\_step\_methods(y\_first, y\_second, p\_value, accuracy):

        return find\_accurate\_y\_value(equation, left, right, current\_right - h\_value / 2, y\_value, h\_value / 2, accuracy, counter + 1, find\_y\_i, p\_value)

    else:

        return y\_second, h\_value, counter, FindAccYValueRetCode.OK

*def* check\_accuracy\_one\_step\_methods(*y\_first*, *y\_second*, *p\_value*, *accuracy*):

    return abs(y\_first - y\_second) / (2\*\*p\_value - 1) > accuracy

*def* miln\_method(*find\_f\_value*, *accuracy*, *h\_value*, *left*, *right*, *initial\_values*):

    f\_values = []

    x\_values = np.arange(left, right + h\_value \* 0.01, h\_value)

    y\_values = initial\_values

    if len(y\_values) < 3:

        return x\_values.tolist()[:len(y\_values)], y\_values

    for i in range(1, 3):

        f\_values.append(find\_f\_value(x\_values[i], y\_values[i]))

    for i in range(4, len(x\_values)):

        f\_values.append(find\_f\_value(x\_values[i - 1], y\_values[i - 1]))

        current\_y\_predicted = y\_values[i - 4] + 4 \* h\_value / 3 \* (2 \* f\_values[i - 4] - f\_values[i - 3] + 2 \* f\_values[i - 2])

        current\_y\_adjustable = y\_values[i - 2] + h\_value / 3 \* (f\_values[i - 3] + 4 \* f\_values[i - 2] + find\_f\_value(x\_values[i], current\_y\_predicted))

        counter = 0

        while counter < MILN\_CONSTRAINT and abs(current\_y\_adjustable - current\_y\_predicted) > accuracy:

            current\_y\_predicted = current\_y\_adjustable

            current\_y\_adjustable = y\_values[i - 2] + h\_value / 3 \* (f\_values[i - 3] + 4 \* f\_values[i - 2] + find\_f\_value(x\_values[i], current\_y\_predicted))

            counter += 1

        if isnan(current\_y\_adjustable) or current\_y\_adjustable == *float*("inf") or current\_y\_adjustable == *float*("-inf"):

            return x\_values.tolist()[:len(y\_values)], y\_values

        y\_values.append(current\_y\_adjustable)

    return x\_values.tolist(), y\_values

*# Поиск свободного коэффициента*

*def* find\_c1(*x*, *y*):

    return -pow(e, x) / y - x \* pow(e, x)

*def* find\_c2(*x*, *y*):

    return y - pow(x, 3) / 3 - pow(x, 2) / 2

*def* find\_c3(*x*, *y*):

    return (y + x \* sin(x) / 2 + x \* cos(x) / 2 + cos(x) / 2) / pow(e, x)

*def* find\_true\_value(*left\_x\_value*, *left\_y\_value*, *x*, *equation\_number*):

    if(equation\_number == 1):

        c = find\_c1(left\_x\_value, left\_y\_value)

        return -pow(e, x) / (x \* pow(e, x) + c)

    elif(equation\_number == 2):

        c = find\_c2(left\_x\_value, left\_y\_value)

        return pow(x, 3) / 3 + pow(x, 2) / 2 + c

    elif(equation\_number == 3):

        c = find\_c3(left\_x\_value, left\_y\_value)

        return - x \* sin(x) / 2 - x \* cos(x) / 2 - cos(x) / 2 + c \* pow(e, x)

    else:

        return None

*# Основонй метод*

*def* find\_true\_values(*left*, *right*, *y\_value*, *h\_value*, *equation\_number*):

    x\_values = np.arange(left, right + h\_value \* 0.01, h\_value)

    y\_values = *list*()

    x\_values\_for\_graph = np.linspace(left, right, 1000)

    y\_values\_for\_graph = *list*()

    for x in x\_values:

        y\_values.append(find\_true\_value(left, y\_value, x, equation\_number))

    for x in x\_values\_for\_graph:

        y\_values\_for\_graph.append(find\_true\_value(left, y\_value, x, equation\_number))

    return {

        "x\_values\_noda" : x\_values.tolist(),

        "y\_values\_noda" : y\_values,

        "x\_values" : x\_values\_for\_graph.tolist(),

        "y\_values" : y\_values\_for\_graph,

    }

*def* define\_equation(*equation\_number*):

    if (equation\_number == 1):

        return first\_equation\_solve

    elif(equation\_number == 2):

        return second\_equation\_solve

    elif(equation\_number == 3):

        return third\_equation\_solve

    else:

        return None

*def* first\_equation\_solve(*x*, *y*):

    return y + (1 + x) \* y\*\*2

*def* second\_equation\_solve(*x*, *y*):

    return x\*\*2 + x

*def* third\_equation\_solve(*x*, *y*):

    return y + x \* sin(x)

*def* find\_noda\_y(*x\_values*, *y\_values*, *h\_value*):

    left = x\_values[0]

    right = x\_values[-1]

    x\_values\_noda = np.arange(left, right + h\_value \* 0.01, h\_value)

    y\_values\_noda = *list*()

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

        min\_delta = 10000000

        current\_j = 0

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

            current\_delta =  abs(x\_values[j] - x\_values\_noda[i])

            if current\_delta < min\_delta:

                min\_delta = current\_delta

                current\_j = j

        y\_values\_noda.append(y\_values[current\_j])

    return x\_values\_noda.tolist(), y\_values\_noda

# Вывод

Мы изучили несколько методов решения обыкновенных дифференциальных уравнений - метод Эйлера, модифицированный метод Эйлера, метод Милна - и разработали их программную реализацию