**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ**

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский Авиационный Институт»**

**(Национальный Исследовательский Университет)**

**Институт: №8 «Информационные технологии   
и прикладная математика»   
Кафедра: 806 «Вычислительная математика   
и программирование»**

Лабораторная работа №4   
по курсу «Численные методы»

Группа: М8О-307Б-22

Студент(ка): П. В. Лебедько

Преподаватель: Д. Л. Ревизников

Оценка:

Дата: 02.05.2025

Москва, 2025

**ОГЛАВЛЕНИЕ**

[**1** **Задание 1** 3](#_Toc197110593)

[**Задание** 3](#_Toc197110594)

[**Вариант** 3](#_Toc197110595)

[**Ход лабораторной работы** 3](#_Toc197110596)

[Результаты 6](#_Toc197110597)

[**2** **Задание 2** 7](#_Toc197110598)

[**Задание** 7](#_Toc197110599)

[**Вариант** 7](#_Toc197110600)

[**Ход лабораторной работы** 7](#_Toc197110601)

[Метод стрельбы 7](#_Toc197110602)

[Конечно-разностный метод 8](#_Toc197110603)

[Результаты 10](#_Toc197110604)

[**3** **Выводы** 11](#_Toc197110605)

# **Задание 1**

## **Задание**

Реализовать методы Эйлера, Рунге-Кутты и Адамса 4-го порядка в виде программ, задавая в качестве входных данных шаг сетки . С использованием разработанного программного обеспечения решить задачу Коши для ОДУ 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге – Ромберга и путем сравнения с точным решением.

## **Вариант**

**Вариант 17**

|  |  |
| --- | --- |
| , |  |

## **Ход лабораторной работы**

Необходимо преобразовать исходное уравнение второго порядка в систему ОДУ первого порядка:

Хранить переменные системы y и z будем в одной массиве, чтобы с помощью программы можно было решать ОДУ любого порядка. В соответствие с этим введем начальные данные и реализуем вектор-функцию f(x):

# Функции

def f(x: float, y: np.ndarray):

    return np.array([

        y[1],

        ((x + 1) \* y[1] - y[0]) / x

    ])

def getTrueY(x):

    return x + 1 + np.exp(x)

# Интервал

a = 1

b = 2

# Начальные условия

y0 = np.array([2 + np.e, 1 + np.e])

Реализуем методы Рунге-Кутты, Эйлера и Адамса:

# Рунге-Кутт 4го порядка

p = 4

as\_ = [0, 0.5, 0.5, 1]

bs = [[0.5], [0, 0.5], [0, 0, 1]]

cs = [1/6, 1/3, 1/3, 1/6]

def getKs(x: float, y: np.ndarray, h):

    dim = y.shape[0]

    Ks = np.empty((p, dim))

    for i in range(p):

        newX = x + as\_[i] \* h

        newY = np.copy(y)

        for j in range(i):

            newY += bs[i - 1][j] \* Ks[j]

        K = h \* f(newX, newY)

        if debug: print(f"\tK{i + 1} = {K}")

        Ks[i] = K

    return Ks

def getDeltaY(x: float, y: np.ndarray, h):

    Ks = getKs(x, y, h)

    dim = Ks.shape[1]

    sum\_ = np.zeros(dim)

    for i in range(p):

        sum\_ += cs[i] \* Ks[i]

    if debug: print(f"\tdeltaY = {sum\_}")

    return sum\_

def RungeKutta(xs: list, y0: np.ndarray, h):

    N = len(xs) - 1

    dim = y0.shape[0]

    ys = np.empty((N + 1, dim))

    ys[0] = y0

    if debug: print(f"N = {N}, dim = {dim}")

    for k in range(1, N + 1):

        if debug: print(f"Шаг {k}")

        ys[k] = ys[k - 1] + getDeltaY(xs[k - 1], ys[k - 1], h)

        if debug: print(f"\ty = {ys[k]}")

    return ys

def Euler(xs: list, y0: np.ndarray, h):

    N = len(xs) - 1

    dim = y0.shape[0]

    ys = np.empty((N + 1, dim))

    ys[0] = y0

    for k in range(N):

        ys[k + 1] = ys[k] + h \* f(xs[k], ys[k])

    return ys

def Adams(xs: list, y0s: np.ndarray, h):

    N = len(xs) - 1

    dim = y0s.shape[1]

    ys = np.empty((N + 1, dim))

    fs = np.empty((N + 1, dim))

    for i in range(4):

        ys[i] = np.copy(y0s[i])

        fs[i] = f(xs[i], ys[i])

    for k in range(4, N + 1):

        ys[k] = ys[k - 1] + h/24 \* (55 \* fs[k - 1] - 59 \* fs[k - 2] + 37 \* fs[k - 3] - 9 \* fs[k - 4])

        fs[k] = f(xs[k], ys[k])

    return ys

А также метод Рунге для оценки погрешности. Он будет принимать массивы рассчитанных значений ys для шага h и ys2 для шага h/2:

def RungeError(ys: np.ndarray, ys2: np.ndarray, p):

    k = 2

    error = 0

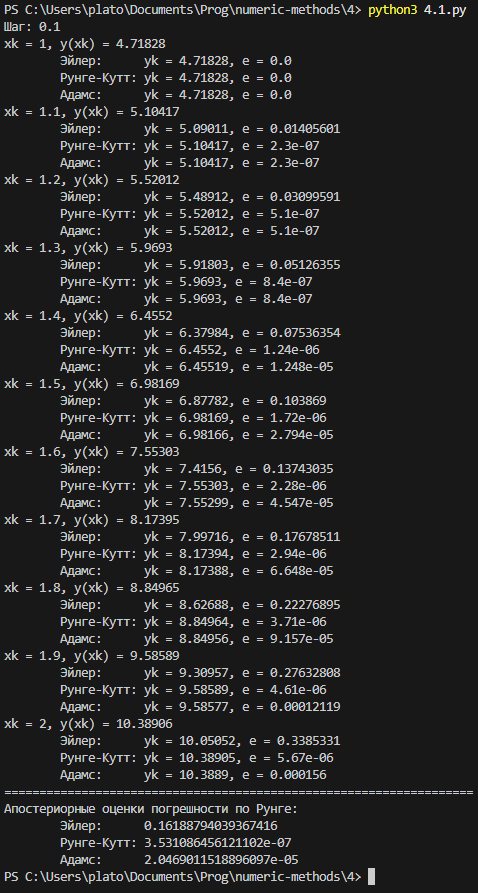
    for i in range(ys.shape[0]):

        error = max(error, abs(ys2[i \* 2][0] - ys[i][0]) / (k \*\* p - 1))

    return error

Остается только провести вычисления.

### Результаты



Заметим, что Рунге-Кутт дает самые точные результаты. Это ожидаемо, поскольку этот метод имеет четвертый порядок точности, что выше методов Эйлера и Адамса.

# **Задание 2**

## **Задание**

Реализовать метод стрельбы и конечно-разностный метод решения краевой задачи для ОДУ в виде программ. С использованием разработанного программного обеспечения решить краевую задачу для обыкновенного дифференциального уравнения 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге – Ромберга и путем сравнения с точным решением.

## **Вариант**

**Вариант 17**

|  |  |
| --- | --- |
| (x2-1) y″+(x-3)y′-y=0,  y′(0)=0,  y′ (1)+y(1)= – 0.75 |  |

## **Ход лабораторной работы**

### Метод стрельбы

Необходимо преобразовать исходную краевую задачу второго порядка в задачу Коши:

Так же определим вектор-функцию f:

    return np.array([

        y[1],

        y[1] / 2

        if x == 1 else

        (y[0] - (x - 3) \* y[1]) / (x \*\* 2 - 1)

    ])

Причем начальные условия заданы только для z(0) = 0. Начальное условие y(0) будем подбирать с помощью метода секущих из условия на значение в точке 1:

def getTrueY(x):

    return x - 3 + 1 / (x + 1)

def y1(ys: np.ndarray):

    return -0.75 - ys[-1][1]

Так как мы знаем истинное решение ОДУ, можно посчитать, что на самом деле y(0) = -2. Введем данное условие в алгоритм Рунге-Кутта и сравним его решение с решением, полученным методом стрельбы также через алгоритм Рунге-Кутта.

Метод Рунге-Кутта берем из предыдущего задания, реализуем метод стрельбы:

def Shooting(xs, y0, h, eps):

    eta0 = randint(-2166, 2166)

    eta1 = randint(-2166, 2166)

    ys0 = RungeKutta(xs, np.array([eta0, y0]), h)

    ys1 = RungeKutta(xs, np.array([eta1, y0]), h)

    F0 = ys0[-1][0] - y1(ys0)

    F1 = ys1[-1][0] - y1(ys1)

    iter = 1

    while True:

        eta = eta1 - (eta1 - eta0) / (F1 - F0) \* F1

        ys = RungeKutta(xs, np.array([eta, y0]), h)

        F0 = F1

        F1 = ys[-1][0] - y1(ys)

        if abs(F1) < eps:

            return ys, iter, eta

        iter += 1

        eta0 = eta1

        eta1 = eta

### Конечно-разностный метод

Так как граничные условия в задаче не первого рода, будем аппроксимировать производные с помощью односторонних разностей первого порядка (для сохранения трех диагональной структуры). Определим реализацию конечно-разностного метода, вручную задавая коэффициенты для первой и последней строки системы:

def FiniteDifference(n, xs, h, A\_b1, A\_c1, A\_an, A\_bn, b1, bn):

    A = np.zeros((n, 3))

    b = np.empty(n)

    A[0][1] = A\_b1

    A[0][2] = A\_c1

    A[n - 1][0] = A\_an

    A[n - 1][1] = A\_bn

    b[0] = b1

    b[n - 1] = bn

    for k in range(1, n - 1):

        A[k][0] = 1 - p(xs[k]) \* h / 2

        A[k][1] = -2 + h \*\* 2 \* q(xs[k])

        A[k][2] = 1 + p(xs[k]) \* h / 2

        b[k] = h \*\* 2 \* f(xs[k])

    ys = tridiagonalMatrixAlgorithm(A, b)

    return ys

Определим функции p(x), q(x), f(x) по исходной задаче:

def p(x):

    return (x - 3) / (x \*\* 2 - 1)

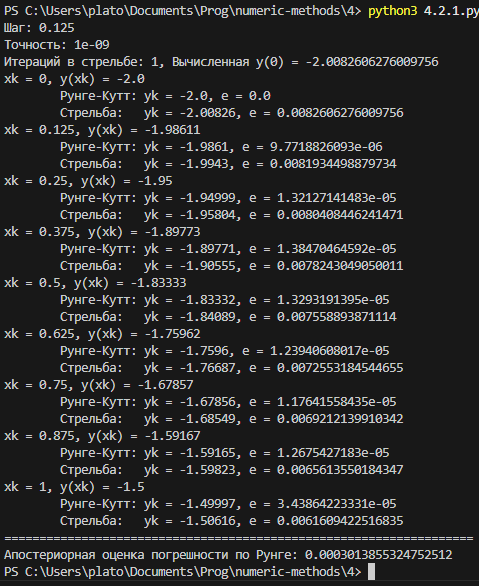
def q(x):

    return -1 / (x \*\* 2 - 1)

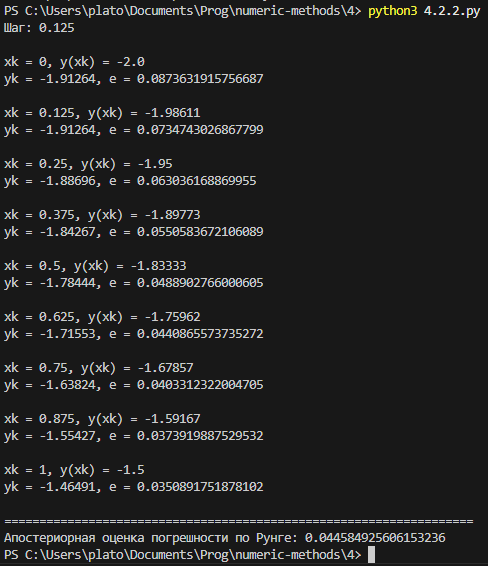
def f(x):

    return 0

### Результаты



Метод стрельбы за одну итерацию смог вычислить y(0) = -2.008, то есть решение с точностью до двух порядков. И приблизительно на два порядка отличается полученное от решение от решения Рунге-Кутты, которое знало y(0) = -2.



Заметно, что конечно-разностный метод дает менее точные результаты, поскольку, из-за приближения производных разностями первого порядка, имеет точность первого порядка.

# **Выводы**

В ходе лабораторной работы я реализовал численные методы для решения задачи Коши для ОДУ второго порядка, а также для решения краевой задачи.