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

Институт информационных технологий и прикладной математики

«Кафедра вычислительной математики и программирования»

**Лабораторная работа по предмету "Численные методы" №4**

Студент: Федоров М.А.

Преподаватель: Ревизников Д.Л.

Группа: М8О-307Б-22

Дата:

Оценка:

Подпись:

Оглавление

[**Задача 4.1** 3](#_Toc199248904)

[**Общий алгоритм решения методом Эйлера** 3](#_Toc199248905)

[**Программное решение** 4](#_Toc199248906)

[**Пример работы** 6](#_Toc199248907)

[**Общий алгоритм решения методом Рунге-Кутты 4-го порядка** 7](#_Toc199248908)

[**Программное решение** 7](#_Toc199248909)

[**Пример работы** 10](#_Toc199248910)

[**Общий алгоритм решения методом Адамса** 11](#_Toc199248911)

[**Программное решение** 11](#_Toc199248912)

[**Пример работы** 14](#_Toc199248913)

[**Задача 4.2** 15](#_Toc199248914)

[**Общий алгоритм решения методом стрельбы** 15](#_Toc199248915)

[**Программное решение** 16](#_Toc199248916)

[**Пример работы** 19](#_Toc199248917)

[**Общий алгоритм решения конечно-разностным методом** 19](#_Toc199248918)

[**Программное решение** 20](#_Toc199248919)

[**Пример работы** 23](#_Toc199248920)

[**Вывод** 23](#_Toc199248921)

# **Задача 4.1**

Реализовать методы Эйлера, Рунге-Кутты и Адамса 4-го порядка в виде программ, задавая в качестве входных данных шаг сетки *h*. 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.

Задача Коши:

,





Точное решение:



# **Общий алгоритм решения методом Эйлера**

График функции y(h), которая является решением задачи Коши, представляет собой гладкую кривую, проходящую через точку (x0, y0), согласно условию y(x0)= y0 , и имеет в этой точке касательную. Тангенс угла наклона касательной к оси Ох равен значению производной от решения в точке x0 и равен значению правой части дифференциального уравнения в точке (x0, y0), согласно выражению y’(x0)=f(x0, y0). Учитывая, что Δy=y1−y0 и заменяя производную y’(x0) на правую часть дифференциального уравнения , получаем соотношение y1=y0+hf(x0, y0). Переход к произвольным индексам дает формулу метода Эйлера:

yk+1=yk+hf(xk, yk).

Основной прием, используемый при решении задач Коши для ОДУ n-го порядка, заключается в введении новых переменных и сведении задачи для ОДУ высокого порядка к решению системы ОДУ первого порядка. Введем новые переменные:

Изображение выглядит как Шрифт, снимок экрана, дизайн

Контент, сгенерированный ИИ, может содержать ошибки.

тогда задачу можно переписать в виде системы n ОДУ первого порядка.

Изображение выглядит как текст, Шрифт, снимок экрана, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, Шрифт, снимок экрана, типография

Контент, сгенерированный ИИ, может содержать ошибки.

Для решения системы к каждому ее отдельному элементу могут применятся любые алгоритмы решения задачи Коши.

# **Программное решение**

ef eiler(step: float):

    # y''=(y-(x+1)\*y')/x^2

    # x лежит в [1; 2]

    x, y, z = sp.symbols('x y z', real=True)

    double\_step = 2 \* step

    p = 1

    x\_start = 1

    x\_end = 2

    y\_start = 2 + sp.exp(1)

    z\_start = 1

    y\_accurate = x + 1 + x\*sp.exp(1/x)

    y\_accurate\_lamb = sp.lambdify(x, y\_accurate)

    x\_\_ = []

    y\_\_ = []

    y\_2h = []

    x\_accurate\_vals = np.linspace(x\_start, x\_end, 400)

    y\_accurate\_vals = y\_accurate\_lamb(x\_accurate\_vals)

    for i in np.linspace(x\_start, x\_end, int((x\_end - x\_start)/step)+1):

        x\_\_.append(i)

    y\_diff\_func = z

    z\_diff\_func = (y-(x+1)\*z)/(x\*\*2)

    y\_diff\_lamb = sp.lambdify([x, y, z], y\_diff\_func)

    z\_diff\_lamb = sp.lambdify([x, y, z], z\_diff\_func)

    x\_prev = x\_start

    y\_prev = y\_start

    z\_prev = z\_start

    y\_\_.append(y\_start)

    for i in range(int((x\_end - x\_start) / step)):

        y\_next = y\_prev + step \* y\_diff\_lamb(x\_prev, y\_prev, z\_prev)

        z\_next = z\_prev + step \* z\_diff\_lamb(x\_prev, y\_prev, z\_prev)

        x\_next = x\_prev + step

        x\_prev = x\_next

        y\_prev = y\_next

        z\_prev = z\_next

        y\_\_.append(y\_next)

        print("y:", y\_next, "\tz:", z\_next)

    x\_prev = x\_start

    y\_prev = y\_start

    z\_prev = z\_start

    y\_2h.append(y\_start)

    for i in range(int((x\_end - x\_start) / double\_step)):

        y\_next = y\_prev + double\_step \* y\_diff\_lamb(x\_prev, y\_prev, z\_prev)

        z\_next = z\_prev + double\_step \* z\_diff\_lamb(x\_prev, y\_prev, z\_prev)

        x\_next = x\_prev + double\_step

        x\_prev = x\_next

        y\_prev = y\_next

        z\_prev = z\_next

        y\_2h.append(y\_next)

    for i in range(len(y\_2h)):

        R = (y\_\_[2\*i] - y\_2h[i]) / (2\*\*p - 1)

        print("Член погрешности на шаге", 2\*i, "равен:", R.evalf(), "\tразность с точным решением:", (y\_accurate\_lamb(x\_\_[2\*i]) - y\_\_[2\*i]).evalf())

    plt.figure()

    plt.plot(x\_accurate\_vals, y\_accurate\_vals, c='green') # точные значения

    plt.scatter(x\_\_, y\_\_, c='red') # приближения

    plt.xlabel('x')

    plt.ylabel('y')

    plt.show()

# **Пример работы**

График: Изображение выглядит как линия, График, диаграмма, текст

Контент, сгенерированный ИИ, может содержать ошибки.

Вывод программы:

y: 2.1 + E z: 0.1\*E + 1

y: 2.2 + 1.01\*E z: 0.165289256198347\*E + 1

y: 2.3 + 1.02652892561983\*E z: 0.210175619834711\*E + 1

y: 2.4 + 1.04754648760331\*E z: 0.242313193799208\*E + 1

y: 2.5 + 1.07177780698323\*E z: 0.266088439844371\*E + 1

y: 2.6 + 1.09838665096766\*E z: 0.284157626838696\*E + 1

y: 2.7 + 1.12680241365153\*E z: 0.298203596416315\*E + 1

y: 2.8 + 1.15662277329316\*E z: 0.309333447742526\*E + 1

y: 2.9 + 1.18755611806742\*E z: 0.318299161310862\*E + 1

y: 3.0 + 1.2193860341985\*E z: 0.325625713949807\*E + 1

Член погрешности на шаге 0 равен: 0 разность с точным решением: -5.88653899022988e-16

Член погрешности на шаге 2 равен: 0.0271828182845905 разность с точным решением: 0.0157064223277538

Член погрешности на шаге 4 равен: 0.0205134801207576 разность с точным решением: 0.0122913166544334

Член погрешности на шаге 6 равен: 0.00769571521336761 разность с точным решением: 0.00346905794416746

Член погрешности на шаге 8 равен: -0.00572846753893990 разность с точным решением: -0.00679046948449210

Член погрешности на шаге 10 равен: -0.0184963910195828 разность с точным решением: -0.0171923572382769

# **Общий алгоритм решения методом Рунге-Кутты 4-го порядка**

Семейство явных методов Рунге-Кутты р-го порядка записывается в виде совокупности формул:

Изображение выглядит как текст, Шрифт, рукописный текст, белый

Контент, сгенерированный ИИ, может содержать ошибки.

Параметры ai, bij, ci подбираются так, чтобы значение yk+1 совпадало со значением разложения в точке xk+1 точного решения в ряд Тейлора с погрешностью O(hp+1). Метод Рунге-Кутты 4-го порядка обладает следующими параметрами и формулами:

Изображение выглядит как Шрифт, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, рукописный текст, Шрифт, каллиграфия

Контент, сгенерированный ИИ, может содержать ошибки.

# **Программное решение**

def runge\_kutt(step: float):

    # y''=(y-(x+1)\*y')/x^2

    # x лежит в [1; 2]

    x, y, z = sp.symbols('x y z', real=True)

    double\_step = step\*2

    p = 4

    x\_start = 1

    x\_end = 2

    y\_start = 2 + sp.exp(1)

    z\_start = 1

    y\_accurate = x + 1 + x\*sp.exp(1/x)

    y\_accurate\_lamb = sp.lambdify(x, y\_accurate)

    x\_\_ = []

    y\_\_ = []

    y\_2h = []

    x\_accurate\_vals = np.linspace(x\_start, x\_end, 400)

    y\_accurate\_vals = y\_accurate\_lamb(x\_accurate\_vals)

    for i in np.linspace(x\_start, x\_end, int((x\_end - x\_start)/step)+1):

        x\_\_.append(i)

    y\_diff\_func = z

    z\_diff\_func = (y-(x+1)\*z)/(x\*\*2)

    y\_diff\_lamb = sp.lambdify([x, y, z], y\_diff\_func)

    z\_diff\_lamb = sp.lambdify([x, y, z], z\_diff\_func)

    x\_prev = x\_start

    y\_prev = y\_start

    z\_prev = z\_start

    y\_\_.append(y\_start)

    for i in range(int((x\_end - x\_start) / step)):

        k1\_y = step \* y\_diff\_lamb(x\_prev, y\_prev, z\_prev)

        k2\_y = step \* y\_diff\_lamb(x\_prev + (1/2)\*step, y\_prev + (1/2)\*k1\_y, z\_prev + (1/2)\*k1\_y)

        k3\_y = step \* y\_diff\_lamb(x\_prev + (1/2)\*step, y\_prev + (1/2)\*k2\_y, z\_prev + (1/2)\*k2\_y)

        k4\_y = step \* y\_diff\_lamb(x\_prev + step, y\_prev + k3\_y, z\_prev + k3\_y)

        k1\_z = step \* z\_diff\_lamb(x\_prev, y\_prev, z\_prev)

        k2\_z = step \* z\_diff\_lamb(x\_prev + (1/2)\*step, y\_prev + (1/2)\*k1\_z, z\_prev + (1/2)\*k1\_z)

        k3\_z = step \* z\_diff\_lamb(x\_prev + (1/2)\*step, y\_prev + (1/2)\*k2\_z, z\_prev + (1/2)\*k2\_z)

        k4\_z = step \* z\_diff\_lamb(x\_prev + step, y\_prev + k3\_z, z\_prev + k3\_z)

        y\_next = y\_prev + (1/6)\*(k1\_y + 2\*k2\_y + 2\*k3\_y + k4\_y)

        z\_next = z\_prev + (1/6)\*(k1\_z + 2\*k2\_z + 2\*k3\_z + k4\_z)

        x\_next = x\_prev + step

        x\_prev = x\_next

        y\_prev = y\_next

        z\_prev = z\_next

        y\_\_.append(y\_next)

        print("y:", y\_next, "\tz:", z\_next)

    x\_prev = x\_start

    y\_prev = y\_start

    z\_prev = z\_start

    y\_2h.append(y\_start)

    for i in range(int((x\_end - x\_start) / double\_step)):

        k1\_y = double\_step \* y\_diff\_lamb(x\_prev, y\_prev, z\_prev)

        k2\_y = double\_step \* y\_diff\_lamb(x\_prev + (1/2)\*double\_step, y\_prev + (1/2)\*k1\_y, z\_prev + (1/2)\*k1\_y)

        k3\_y = double\_step \* y\_diff\_lamb(x\_prev + (1/2)\*double\_step, y\_prev + (1/2)\*k2\_y, z\_prev + (1/2)\*k2\_y)

        k4\_y = double\_step \* y\_diff\_lamb(x\_prev + double\_step, y\_prev + k3\_y, z\_prev + k3\_y)

        k1\_z = double\_step \* z\_diff\_lamb(x\_prev, y\_prev, z\_prev)

        k2\_z = double\_step \* z\_diff\_lamb(x\_prev + (1/2)\*double\_step, y\_prev + (1/2)\*k1\_z, z\_prev + (1/2)\*k1\_z)

        k3\_z = double\_step \* z\_diff\_lamb(x\_prev + (1/2)\*double\_step, y\_prev + (1/2)\*k2\_z, z\_prev + (1/2)\*k2\_z)

        k4\_z = double\_step \* z\_diff\_lamb(x\_prev + double\_step, y\_prev + k3\_z, z\_prev + k3\_z)

        y\_next = y\_prev + (1/6)\*(k1\_y + 2\*k2\_y + 2\*k3\_y + k4\_y)

        z\_next = z\_prev + (1/6)\*(k1\_z + 2\*k2\_z + 2\*k3\_z + k4\_z)

        x\_next = x\_prev + double\_step

        x\_prev = x\_next

        y\_prev = y\_next

        z\_prev = z\_next

        y\_2h.append(y\_next)

    for i in range(len(y\_2h)):

        R = (y\_\_[2\*i] - y\_2h[i]) / (2\*\*p - 1)

        print("Член погрешности на шаге", 2\*i, "равен:", R.evalf(), "\tразность с точным решением:", (y\_accurate\_lamb(x\_\_[2\*i]) - y\_\_[2\*i]).evalf())

    plt.figure()

    plt.plot(x\_accurate\_vals, y\_accurate\_vals, c='green') # точные значения

    plt.scatter(x\_\_, y\_\_, c='red') # приближения

    plt.xlabel('x')

    plt.ylabel('y')

    plt.show()

# **Пример работы**

График: Изображение выглядит как линия, График, диаграмма, текст

Контент, сгенерированный ИИ, может содержать ошибки.

Вывод программы:

y: 2.10517083333333 + E z: 0.0866456775547685\*E + 0.995736586645678

y: 2.20989327993135 + 1.00911259811317\*E z: 0.145652083774663\*E + 0.993203077403169

y: 2.31434927525107 + 1.02443094914048\*E z: 0.187612493614351\*E + 0.991716080302968

y: 2.4186488818466 + 1.04416231143765\*E z: 0.218507982914893\*E + 0.990891852145234

y: 2.52286180367992 + 1.0671429780908\*E z: 0.241916966542604\*E + 0.990502445908023

y: 2.62703377133478 + 1.09258558705955\*E z: 0.26008395243844\*E + 0.990406181223987

y: 2.73119561475259 + 1.11993883307413\*E z: 0.27447319196591\*E + 0.990511664605727

y: 2.83536855194556 + 1.14880540740085\*E z: 0.2860721698009\*E + 0.990758095702772

y: 2.93956740650237 + 1.17889185589228\*E z: 0.295566051482799\*E + 0.991103968979531

y: 3.04380263683992 + 1.20997678383177\*E z: 0.303442095478113\*E + 0.991520327764819

Член погрешности на шаге 0 равен: 0 разность с точным решением: -5.88653899022988e-16

Член погрешности на шаге 2 равен: 0.000884259319494546 разность с точным решением: 0.00822535081992626

Член погрешности на шаге 4 равен: 0.000514437347017559 разность с точным решением: 0.00284157938323949

Член погрешности на шаге 6 равен: -0.000263678957940658 разность с точным решением: -0.00779578678346139

Член погрешности на шаге 8 равен: -0.00121103520551737 разность с точным решением: -0.0209092177785432

Член погрешности на шаге 10 равен: -0.00224600525267003 разность с точным решением: -0.0354179997868791

# **Общий алгоритм решения методом Адамса**

Многошаговые методы решения задачи Коши характеризуются тем, что решение в текущем узле зависит от данных не в одном предыдущем узле, как это имеет место в одношаговых методах, а от нескольких предыдущих узлах. Многие многошаговые методы различного порядка точности можно конструировать с помощью квадратурного способа (т.е. с использованием эквивалентного интегрального уравнения). Решение дифференциального уравнения y’=f(x,y) удовлетворяет интегральному соотношению:

Изображение выглядит как текст, Шрифт, белый, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Если решение задачи Коши получено в узлах вплоть до k-го, то можно аппроксимировать подынтегральную функцию, например: интерполяционным многочленом какой-либо степени. При использовании интерполяционного многочлена 3-ей степени построенного по значениям подынтегральной функции в последних четырех узлах получим метод Адамса четвертого порядка точности:



где fk - значение подынтегральной функции в узле xk.

Метод Адамса (4.25) как и все многошаговые методы не является самостартующим, то есть для того, чтобы использовать метод Адамса необходимо иметь решения в первых четырех узлах. В узле x0 решение y0 известно из начальных условий, а в других трех узлах решения можно получить с помощью подходящего одношагового метода, например: метода Рунге-Кутты четвертого порядка.

# **Программное решение**

def adams(step: float):

    # y''=(y-(x+1)\*y')/x^2

    # x лежит в [1; 2]

    x, y, z = sp.symbols('x y z', real=True)

    double\_step = 2\*step

    p = 4

    x\_start = 1

    x\_end = 2

    y\_start = 2 + sp.exp(1)

    z\_start = 1

    y\_accurate = x + 1 + x\*sp.exp(1/x)

    y\_accurate\_lamb = sp.lambdify(x, y\_accurate)

    x\_\_ = []

    y\_\_ = []

    y\_2h = []

    x\_accurate\_vals = np.linspace(x\_start, x\_end, 400)

    y\_accurate\_vals = y\_accurate\_lamb(x\_accurate\_vals)

    for i in np.linspace(x\_start, x\_end, int((x\_end - x\_start)/step)+1):

        x\_\_.append(i)

    y\_diff\_func = z

    z\_diff\_func = (y-(x+1)\*z)/(x\*\*2)

    y\_diff\_lamb = sp.lambdify([x, y, z], y\_diff\_func)

    z\_diff\_lamb = sp.lambdify([x, y, z], z\_diff\_func)

    last\_4\_x = [x\_start]

    last\_4\_y = [y\_start]

    last\_4\_z = [z\_start]

    y\_\_.append(y\_start)

    while len(last\_4\_x) < 4:

        k1\_y = step \* y\_diff\_lamb(last\_4\_x[-1], last\_4\_y[-1], last\_4\_z[-1])

        k2\_y = step \* y\_diff\_lamb(last\_4\_x[-1] + (1/2)\*step, last\_4\_y[-1] + (1/2)\*k1\_y, last\_4\_z[-1] + (1/2)\*k1\_y)

        k3\_y = step \* y\_diff\_lamb(last\_4\_x[-1] + (1/2)\*step, last\_4\_y[-1] + (1/2)\*k2\_y, last\_4\_z[-1] + (1/2)\*k2\_y)

        k4\_y = step \* y\_diff\_lamb(last\_4\_x[-1] + step, last\_4\_y[-1] + k3\_y, last\_4\_z[-1] + k3\_y)

        k1\_z = step \* z\_diff\_lamb(last\_4\_x[-1], last\_4\_y[-1], last\_4\_z[-1])

        k2\_z = step \* z\_diff\_lamb(last\_4\_x[-1] + (1/2)\*step, last\_4\_y[-1] + (1/2)\*k1\_z, last\_4\_z[-1] + (1/2)\*k1\_z)

        k3\_z = step \* z\_diff\_lamb(last\_4\_x[-1] + (1/2)\*step, last\_4\_y[-1] + (1/2)\*k2\_z, last\_4\_z[-1] + (1/2)\*k2\_z)

        k4\_z = step \* z\_diff\_lamb(last\_4\_x[-1] + step, last\_4\_y[-1] + k3\_z, last\_4\_z[-1] + k3\_z)

        y\_\_.append(last\_4\_y[-1] + (1/6)\*(k1\_y + 2\*k2\_y + 2\*k3\_y + k4\_y))

        last\_4\_y.append(last\_4\_y[-1] + (1/6)\*(k1\_y + 2\*k2\_y + 2\*k3\_y + k4\_y))

        last\_4\_z.append(last\_4\_z[-1] + (1/6)\*(k1\_z + 2\*k2\_z + 2\*k3\_z + k4\_z))

        last\_4\_x.append(last\_4\_x[-1] + step)

        print("y:", last\_4\_y[-1] + (1/6)\*(k1\_y + 2\*k2\_y + 2\*k3\_y + k4\_y), "\tz:", last\_4\_z[-1] + (1/6)\*(k1\_z + 2\*k2\_z + 2\*k3\_z + k4\_z))

    while(np.abs(last\_4\_x[-1] - x\_end) > 0.000001 and last\_4\_x[-1] < x\_end):

        y\_next = last\_4\_y[-1] + (step/24)\*(55\*y\_diff\_lamb(last\_4\_x[-1], last\_4\_y[-1], last\_4\_z[-1])  -  59\*y\_diff\_lamb(last\_4\_x[-2], last\_4\_y[-2], last\_4\_z[-2])  +  37\*y\_diff\_lamb(last\_4\_x[-3], last\_4\_y[-3], last\_4\_z[-3])  -  9\*y\_diff\_lamb(last\_4\_x[-4], last\_4\_y[-4], last\_4\_z[-4]))

        z\_next = last\_4\_z[-1] + (step/24)\*(55\*z\_diff\_lamb(last\_4\_x[-1], last\_4\_y[-1], last\_4\_z[-1])  -  59\*z\_diff\_lamb(last\_4\_x[-2], last\_4\_y[-2], last\_4\_z[-2])  +  37\*z\_diff\_lamb(last\_4\_x[-3], last\_4\_y[-3], last\_4\_z[-3])  -  9\*z\_diff\_lamb(last\_4\_x[-4], last\_4\_y[-4], last\_4\_z[-4]))

        x\_next = last\_4\_x[-1] + step

        last\_4\_y.append(y\_next)

        last\_4\_z.append(z\_next)

        last\_4\_x.append(x\_next)

        last\_4\_y.pop(0)

        last\_4\_z.pop(0)

        last\_4\_x.pop(0)

        y\_\_.append(y\_next)

        print("y:", y\_next, "\tz:", z\_next, "\tx:", last\_4\_x[-1])

    last\_4\_x = [x\_start]

    last\_4\_y = [y\_start]

    last\_4\_z = [z\_start]

    y\_2h.append(y\_start)

    while len(last\_4\_x) < 4:

        k1\_y = double\_step \* y\_diff\_lamb(last\_4\_x[-1], last\_4\_y[-1], last\_4\_z[-1])

        k2\_y = double\_step \* y\_diff\_lamb(last\_4\_x[-1] + (1/2)\*double\_step, last\_4\_y[-1] + (1/2)\*k1\_y, last\_4\_z[-1] + (1/2)\*k1\_y)

        k3\_y = double\_step \* y\_diff\_lamb(last\_4\_x[-1] + (1/2)\*double\_step, last\_4\_y[-1] + (1/2)\*k2\_y, last\_4\_z[-1] + (1/2)\*k2\_y)

        k4\_y = double\_step \* y\_diff\_lamb(last\_4\_x[-1] + double\_step, last\_4\_y[-1] + k3\_y, last\_4\_z[-1] + k3\_y)

        k1\_z = double\_step \* z\_diff\_lamb(last\_4\_x[-1], last\_4\_y[-1], last\_4\_z[-1])

        k2\_z = double\_step \* z\_diff\_lamb(last\_4\_x[-1] + (1/2)\*double\_step, last\_4\_y[-1] + (1/2)\*k1\_z, last\_4\_z[-1] + (1/2)\*k1\_z)

        k3\_z = double\_step \* z\_diff\_lamb(last\_4\_x[-1] + (1/2)\*double\_step, last\_4\_y[-1] + (1/2)\*k2\_z, last\_4\_z[-1] + (1/2)\*k2\_z)

        k4\_z = double\_step \* z\_diff\_lamb(last\_4\_x[-1] + double\_step, last\_4\_y[-1] + k3\_z, last\_4\_z[-1] + k3\_z)

        y\_2h.append(last\_4\_y[-1] + (1/6)\*(k1\_y + 2\*k2\_y + 2\*k3\_y + k4\_y))

        last\_4\_y.append(last\_4\_y[-1] + (1/6)\*(k1\_y + 2\*k2\_y + 2\*k3\_y + k4\_y))

        last\_4\_z.append(last\_4\_z[-1] + (1/6)\*(k1\_z + 2\*k2\_z + 2\*k3\_z + k4\_z))

        last\_4\_x.append(last\_4\_x[-1] + double\_step)

        print("y:", last\_4\_y[-1] + (1/6)\*(k1\_y + 2\*k2\_y + 2\*k3\_y + k4\_y), "\tz:", last\_4\_z[-1] + (1/6)\*(k1\_z + 2\*k2\_z + 2\*k3\_z + k4\_z))

    while(np.abs(last\_4\_x[-1] - x\_end) > 0.000001 and last\_4\_x[-1] < x\_end):

        y\_next = last\_4\_y[-1] + (double\_step/24)\*(55\*y\_diff\_lamb(last\_4\_x[-1], last\_4\_y[-1], last\_4\_z[-1])  -  59\*y\_diff\_lamb(last\_4\_x[-2], last\_4\_y[-2], last\_4\_z[-2])  +  37\*y\_diff\_lamb(last\_4\_x[-3], last\_4\_y[-3], last\_4\_z[-3])  -  9\*y\_diff\_lamb(last\_4\_x[-4], last\_4\_y[-4], last\_4\_z[-4]))

        z\_next = last\_4\_z[-1] + (double\_step/24)\*(55\*z\_diff\_lamb(last\_4\_x[-1], last\_4\_y[-1], last\_4\_z[-1])  -  59\*z\_diff\_lamb(last\_4\_x[-2], last\_4\_y[-2], last\_4\_z[-2])  +  37\*z\_diff\_lamb(last\_4\_x[-3], last\_4\_y[-3], last\_4\_z[-3])  -  9\*z\_diff\_lamb(last\_4\_x[-4], last\_4\_y[-4], last\_4\_z[-4]))

        x\_next = last\_4\_x[-1] + double\_step

        last\_4\_y.append(y\_next)

        last\_4\_z.append(z\_next)

        last\_4\_x.append(x\_next)

        last\_4\_y.pop(0)

        last\_4\_z.pop(0)

        last\_4\_x.pop(0)

        y\_2h.append(y\_next)

    for i in range(len(y\_2h)):

        R = (y\_\_[2\*i] - y\_2h[i]) / (2\*\*p - 1)

        print("Член погрешности на шаге", 2\*i, "равен:", R.evalf(), "\tразность с точным решением:", (y\_accurate\_lamb(x\_\_[2\*i]) - y\_\_[2\*i]).evalf())

    # print(len(x\_\_), len(y\_\_))

    plt.figure()

    plt.plot(x\_accurate\_vals, y\_accurate\_vals, c='green') # точные значения

    plt.scatter(x\_\_, y\_\_, c='red') # приближения

    plt.xlabel('x')

    plt.ylabel('y')

    plt.show()

# **Пример работы**

График:

Изображение выглядит как линия, График, диаграмма, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

Вывод программы:

y: 2.21034166666667 + E z: 0.173291355109537\*E + 0.991473173291355

y: 2.31461572652936 + 1.01822519622633\*E z: 0.204658489994558\*E + 0.990669568160661

y: 2.41880527057079 + 1.0397493001678\*E z: 0.229572903454039\*E + 0.990229083202766

y: 2.4134645109001 + 1.04497721695553\*E z: 0.21717160777172\*E + 0.993803292927133 x: 1.4000000000000004

y: 2.51319291488853 + 1.06782978906331\*E z: 0.240540022964089\*E + 0.994479989392973 x: 1.5000000000000004

y: 2.61242904992392 + 1.09302716370603\*E z: 0.258512236841295\*E + 0.996075155210667 x: 1.6000000000000005

y: 2.71224193024883 + 1.11958195002442\*E z: 0.273263785421185\*E + 0.996639732621431 x: 1.7000000000000006

y: 2.81181810153193 + 1.14759329420882\*E z: 0.28506022887335\*E + 0.997463791783453 x: 1.8000000000000007

y: 2.91166487303895 + 1.1765759683948\*E z: 0.294872773023892\*E + 0.997902024391859 x: 1.9000000000000008

y: 3.01143671194072 + 1.20650763065229\*E z: 0.3029660218143\*E + 0.998398723650383 x: 2.000000000000001

y: 2.4428 + E z: 0.303872053872054\*E + 0.970538720538721

y: 2.65767727272727 + 1.06727727272727\*E z: 0.295283236751491\*E + 0.975979934233903

y: 2.87375923016666 + 1.13265298134405\*E z: 0.302912958020189\*E + 0.978982650784056

Член погрешности на шаге 0 равен: 0 разность с точным решением: -5.88653899022988e-16

Член погрешности на шаге 2 равен: 0.000884259319494546 разность с точным решением: 0.00822535081992626

Член погрешности на шаге 4 равен: 0.000316488807994630 разность с точным решением: 0.00581080746858342

Член погрешности на шаге 6 равен: -0.00115730506706548 разность с точным решением: 0.00560860485341088

Член погрешности на шаге 8 равен: -0.00145701656115303 разность с точным решением: 0.00593609789901149

Член погрешности на шаге 10 равен: -0.000839368512162965 разность с точным решением: 0.00637806116023543

# **Задача 4.2**

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.

Краевая задача:

(x2+1)y″-2y=0

y′(0)=2

y(1)=3+(П/2)

Точное решение:

y(x)=x2+x+1+(x2+1)arctg(x)

# **Общий алгоритм решения методом стрельбы**

Примером краевой задачи является двухточечная краевая задача для обыкновенного дифференциального уравнения второго порядка:

y’’=f(x, y, y’)

с граничными условиями, заданными на концах отрезка [a,b]:

Изображение выглядит как Шрифт, текст, типография, дизайн

Контент, сгенерированный ИИ, может содержать ошибки.(4.1)

Следует найти такое решение y(x) на этом отрезке, которое принимает на концах отрезка значения y0, y1. Кроме граничных условий (4.1), называемых граничными условиями первого рода, используются еще условия на производные от решения на концах - граничные условия второго рода:



Суть метода стрельбы заключена в многократном решении задачи Коши для приближенного нахождения решения краевой задачи. Пусть надо решить краевую задачу на отрезке [a, b]. Вместо исходной задачи формулируется задача Коши с начальными условиями:



где η - некоторое значение тангенса угла наклона касательной к решению в точке x = b. Положим сначала некоторое начальное значение параметру η =η0, после чего решим каким либо методом задачу Коши. Пусть y=y0(x, y0, η0) решение этой задачи на интервале [a ,b], тогда сравнивая значение функции y0(b, y0, η0) со значением y1 в правом конце отрезка можно получить информацию для корректировки угла наклона касательной к решению в левом конце отрезка. Решая задачу Коши для нового значения η =η1, получим другое решение со значением y1(b, y0, η1) на правом конце. Таким образом, значение решения на правом конце y(b, y0, η0) будет являться функцией одной переменной η. Решение исходной задачи эквивалентно нахождению корня уравнения:

Изображение выглядит как Шрифт, текст, белый, каллиграфия

Контент, сгенерированный ИИ, может содержать ошибки.

Следующее значение искомого корня определяется по соотношению:

(4.2)

Итерации по формуле (4.2) выполняются до удовлетворения заданной точности.

# **Программное решение**

def shooting():

    # y'' = 2y/(x\*\*2+1)

    # y'(0) = 2

    # y(1) = 3+(П/2)

    # x в [0; 1]

    x, y, z= sp.symbols('x y z', real=True)

    accuracy = 0.00001

    def f(t):

        return t\*\*2 + t + 1 + (t\*\*2 + 1) \* np.arctan(t)

    def solving(step):

        x\_start = 0

        x\_end = 1

        y\_end = 3 + np.pi/2

        z\_start = 2

        n\_mas = [0., 0.2] # y(a) <-------- подбираемые параметры для решения задачи

        solutions = []

        y\_\_ = []

        x\_\_ = np.linspace(x\_start, x\_end, int((x\_end - x\_start)/np.abs(step))+1)

        y\_diff\_func = z

        z\_diff\_func = 2\*y/(1+x\*\*2)

        y\_diff\_lamb = sp.lambdify([x, y, z], y\_diff\_func)

        z\_diff\_lamb = sp.lambdify([x, y, z], z\_diff\_func)

        for j in range(2):

            y\_\_.clear()

            x\_prev = x\_start

            y\_prev = n\_mas[j]

            z\_prev = z\_start

            for i in range(int((x\_end - x\_start) / np.abs(step)) + 1):

                k1\_y = step \* y\_diff\_lamb(x\_prev, y\_prev, z\_prev)

                k2\_y = step \* y\_diff\_lamb(x\_prev + (1/2)\*step, y\_prev + (1/2)\*k1\_y, z\_prev + (1/2)\*k1\_y)

                k3\_y = step \* y\_diff\_lamb(x\_prev + (1/2)\*step, y\_prev + (1/2)\*k2\_y, z\_prev + (1/2)\*k2\_y)

                k4\_y = step \* y\_diff\_lamb(x\_prev + step, y\_prev + k3\_y, z\_prev + k3\_y)

                k1\_z = step \* z\_diff\_lamb(x\_prev, y\_prev, z\_prev)

                k2\_z = step \* z\_diff\_lamb(x\_prev + (1/2)\*step, y\_prev + (1/2)\*k1\_z, z\_prev + (1/2)\*k1\_z)

                k3\_z = step \* z\_diff\_lamb(x\_prev + (1/2)\*step, y\_prev + (1/2)\*k2\_z, z\_prev + (1/2)\*k2\_z)

                k4\_z = step \* z\_diff\_lamb(x\_prev + step, y\_prev + k3\_z, z\_prev + k3\_z)

                y\_next = y\_prev + (1/6)\*(k1\_y + 2\*k2\_y + 2\*k3\_y + k4\_y)

                z\_next = z\_prev + (1/6)\*(k1\_z + 2\*k2\_z + 2\*k3\_z + k4\_z)

                x\_next = x\_prev + step

                x\_prev = x\_next

                y\_prev = y\_next

                z\_prev = z\_next

                # y\_\_.insert(0, y\_next)

                y\_\_.append(y\_next)

            solutions.append(y\_\_[-1])

        while np.abs(solutions[-1] - y\_end) > accuracy:

            n\_mas.append(n\_mas[-1] - ((n\_mas[-1] - n\_mas[-2]) / (solutions[-1] - solutions[-2])) \* (solutions[-1] - y\_end))

            y\_\_.clear()

            x\_prev = x\_start

            y\_prev = n\_mas[-1]

            z\_prev = z\_start

            # y\_\_.append(y\_end)

            for i in range(int((x\_end - x\_start) / np.abs(step)) + 1):

                k1\_y = step \* y\_diff\_lamb(x\_prev, y\_prev, z\_prev)

                k2\_y = step \* y\_diff\_lamb(x\_prev + (1/2)\*step, y\_prev + (1/2)\*k1\_y, z\_prev + (1/2)\*k1\_y)

                k3\_y = step \* y\_diff\_lamb(x\_prev + (1/2)\*step, y\_prev + (1/2)\*k2\_y, z\_prev + (1/2)\*k2\_y)

                k4\_y = step \* y\_diff\_lamb(x\_prev + step, y\_prev + k3\_y, z\_prev + k3\_y)

                k1\_z = step \* z\_diff\_lamb(x\_prev, y\_prev, z\_prev)

                k2\_z = step \* z\_diff\_lamb(x\_prev + (1/2)\*step, y\_prev + (1/2)\*k1\_z, z\_prev + (1/2)\*k1\_z)

                k3\_z = step \* z\_diff\_lamb(x\_prev + (1/2)\*step, y\_prev + (1/2)\*k2\_z, z\_prev + (1/2)\*k2\_z)

                k4\_z = step \* z\_diff\_lamb(x\_prev + step, y\_prev + k3\_z, z\_prev + k3\_z)

                y\_next = y\_prev + (1/6)\*(k1\_y + 2\*k2\_y + 2\*k3\_y + k4\_y)

                z\_next = z\_prev + (1/6)\*(k1\_z + 2\*k2\_z + 2\*k3\_z + k4\_z)

                x\_next = x\_prev + step

                x\_prev = x\_next

                y\_prev = y\_next

                z\_prev = z\_next

                # y\_\_.insert(0, y\_next)

                y\_\_.append(y\_next)

                # print("y:", y\_next, "\tz:", z\_next)

            solutions.append(y\_\_[-1])

        # print("Все подбираемые y(a):", n\_mas)

        # print("Все вычисленные значения y(b):", solutions)

        # print("Точное значение y(b):", y\_end)

        return x\_\_, y\_\_

    x\_h, y\_h = solving(0.1)

    x\_2h, y\_2h = solving(0.2)

    # Вычисление погрешности методом Рунге-Ромберга (порядок 2)

    for i in range(len(y\_2h)):

        print("x=", x\_2h[i], "Остаточный член погрешности (Рунге-Ромберг):", np.abs(y\_h[2\*i] - y\_2h[i]) / 3, "\t\tРазность с точным решением", np.abs(f(x\_2h[i]) - y\_h[2\*i]))

    plt.scatter(x\_h, y\_h, label="Численное решение")

    plt.plot(x\_h, f(x\_h), c="green", label="Точное решение")

    plt.xlabel("x")

    plt.ylabel("y(x)")

    plt.title("Конечно-разностное решение")

    plt.grid(True)

    plt.legend()

    plt.show()

# **Пример работы**

График:

Изображение выглядит как текст, линия, График, диаграмма

Контент, сгенерированный ИИ, может содержать ошибки.

Вывод программы:

x= 0.0 Остаточный член погрешности (Рунге-Ромберг): 0.0014474716806354888 Разность с точным решением 0.056963481558711004

x= 0.2 Остаточный член погрешности (Рунге-Ромберг): 0.0051686817033030534 Разность с точным решением 0.026165875198258748

x= 0.4 Остаточный член погрешности (Рунге-Ромберг): 0.008599920022108595 Разность с точным решением 0.00471472539303508

x= 0.6000000000000001 Остаточный член погрешности (Рунге-Ромберг): 0.009137322467584438 Разность с точным решением 0.007629675350287801

x= 0.8 Остаточный член погрешности (Рунге-Ромберг): 0.006426375542495884 Разность с точным решением 0.009689242170067836

x= 1.0 Остаточный член погрешности (Рунге-Ромберг): 2.9605947323337506e-16 Разность с точным решением 2.6645352591003757e-15

# **Общий алгоритм решения конечно-разностным методом**

Рассмотрим двухточечную краевую задачу для линейного дифференциального

уравнения второго порядка на отрезке [a ,b]:

Изображение выглядит как Шрифт, текст, белый, рукописный текст

Контент, сгенерированный ИИ, может содержать ошибки.

Введем разностную аппроксимацию производных следующим образом:

Изображение выглядит как Шрифт, рукописный текст, текст, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Подставляя аппроксимации производных, получим систему уравнений для нахождения yk:

Изображение выглядит как Шрифт, текст, линия, диаграмма

Контент, сгенерированный ИИ, может содержать ошибки.

Приводя подобные и учитывая, что при задании граничных условий первого рода два неизвестных y0, yN уже фактически определены, получим систему линейных алгебраических уравнений с трехдиагональной матрицей коэффициентов:

Изображение выглядит как текст, Шрифт, линия, белый

Контент, сгенерированный ИИ, может содержать ошибки.

В итоге, задача сводится к решению системы, например, методом прогонки.

В случае использования граничных условий второго и третьего рода аппроксимация производных проводится с помощью односторонних разностей первого и второго порядков:

Изображение выглядит как текст, Шрифт, рукописный текст, линия

Контент, сгенерированный ИИ, может содержать ошибки.

# **Программное решение**

def end\_diffirence():

    # y'' = 2y/(x\*\*2+1)

    # y'(0) = 2

    # y(1) = 3+(П/2)

    # x в [0; 1]

    def p(x):

        return 0

    def q(x):

        return 2 / (1 + x\*\*2)

    def r(x):

        return 0

    # Точное решение

    def f(x):

        return x\*\*2 + x + 1 + (x\*\*2 + 1) \* np.arctan(x)

    def solving(h):

        a, b = 0.0, 1.0

        N = int((b - a) / h) # число разбиений

        x = np.linspace(a, b, N + 1)

        dy\_a = 2.0  # y'(0) = 2

        y\_b = 3 + np.pi / 2  # y(1) = 3 + π/2

        A = np.zeros(N)

        B = np.zeros(N)

        C = np.zeros(N)

        D = np.zeros(N)

        B[0] = -1/h

        C[0] = 1/h

        D[0] = dy\_a

        for i in range(1, N - 1):

            A[i] = 1 / h\*\*2 - p(x[i]) / (2 \* h)

            B[i] = -2 / h\*\*2 - q(x[i])

            C[i] = 1 / h\*\*2 + p(x[i]) / (2 \* h)

            D[i] = r(x[i])

        A[N - 1] = 1 / h\*\*2 - p(x[N - 1]) / (2 \* h)

        B[N - 1] = -2 / h\*\*2 + q(x[N - 1])

        C[N - 1] = 0  # поскольку y\_N известно

        D[N - 1] = r(x[N - 1]) - (1 / h\*\*2 + p(x[N - 1]) / (2 \* h)) \* y\_b

        # Решение СЛАУ методом прогонки

        for i in range(1, N):

            m = A[i] / B[i - 1]

            B[i] = B[i] - m \* C[i - 1]

            D[i] = D[i] - m \* D[i - 1]

        # Обратный ход

        y\_2h = np.zeros((N + 1) // 2)

        y = np.zeros(N + 1)

        y[N - 1] = D[N - 1] / B[N - 1]  # Начинаем с предпоследней точки

        for i in range(N - 2, -1, -1):

            y[i] = (D[i] - C[i] \* y[i + 1]) / B[i]

        # y\_N = y(1)

        y[N] = y\_b

        return x, y

    x\_h, y\_h = solving(0.1)

    x\_2h, y\_2h = solving(0.2)

    # Вычисление погрешности методом Рунге-Ромберга (порядок 2)

    for i in range(len(y\_2h)):

        print("x=", x\_2h[i], "Остаточный член погрешности (Рунге-Ромберг):", np.abs(y\_h[2\*i] - y\_2h[i]) / 3, "\t\tРазность с точным решением", np.abs(f(x\_2h[i]) - y\_h[2\*i]))

    plt.scatter(x\_h, y\_h, label="Численное решение")

    plt.plot(x\_h, f(x\_h), c="green", label="Точное решение")

    plt.xlabel("x")

    plt.ylabel("y(x)")

    plt.title("Конечно-разностное решение")

    plt.grid(True)

    plt.legend()

    plt.show()

# **Пример работы**

График: Изображение выглядит как текст, линия, График, диаграмма

Контент, сгенерированный ИИ, может содержать ошибки.

Вывод программы:

x= 0.0 Остаточный член погрешности (Рунге-Ромберг): 0.0860766627965377 Разность с точным решением 0.12104398800481908

x= 0.2 Остаточный член погрешности (Рунге-Ромберг): 0.07735690049947615 Разность с точным решением 0.10191189265212763

x= 0.4 Остаточный член погрешности (Рунге-Ромберг): 0.07467561083860315 Разность с точным решением 0.09072697538169683

x= 0.6000000000000001 Остаточный член погрешности (Рунге-Ромберг): 0.07727177585169558 Разность с точным решением 0.08595646260907186

x= 0.8 Остаточный член погрешности (Рунге-Ромберг): 0.08453259064690173 Разность с точным решением 0.08639063444278072

x= 1.0 Остаточный член погрешности (Рунге-Ромберг): 0.0 Разность с точным решением 0.0

# **Вывод**

В ходе выполнения лабораторной работы я научился реализовывать алгоритмы решения различных задач, связанных с дифференциальными уравнениями.