**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ**

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский Авиационный Институт»**

**(Национальный Исследовательский Университет)**

**Институт: №8 «Компьютерные науки и прикладная математика»  
Кафедра: 806 «Вычислительная математика   
и программирование»**

Лабораторная работа № 4   
по курсу «Численные методы»

Группа: М8О-309Б-22

Студент: Концебалов О. С.

Преподаватель: Шабунина А. А.

Дата: 05.06.2025

Москва, 2025

**ОГЛАВЛЕНИЕ**

[**1**](#_goz6jjoqwyzv) **Тема** 3

[**2**](#_vuk8k1q4dmz5) **Задание** 3

[**3**](#_2ozmd7wlooac) **Теория** 4

[**4**](#_9z3sljvrrr44) **Демонстрация работы программы** 11

[**5**](#_84iii2jwnsps) **Исходный код программы** 14

[**6**](#_naq15jiogqdr) **Выводы** 23

# **Тема**

Численные методы решения обыкновенных дифференциальных уравнений.

# **Задание**

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.

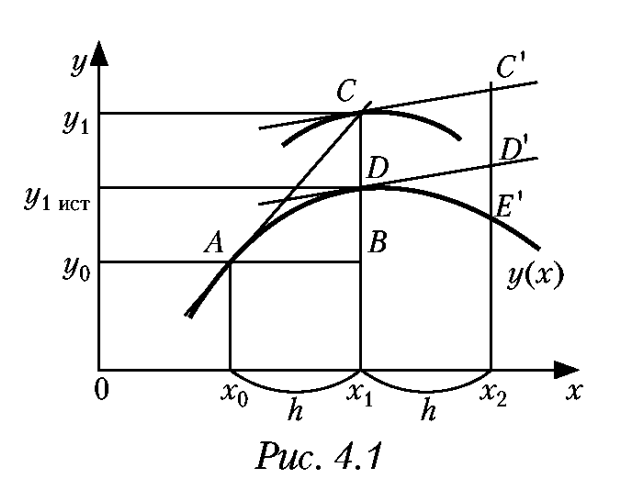
|  |  |  |
| --- | --- | --- |
| 9 |  |  |

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.

|  |  |  |
| --- | --- | --- |
| 9 | xy″-(2x+1)y′ +(x+1)y=0,  y′(0)=1,  y′ (1)-2y(1)=0 |  |

# **Теория**

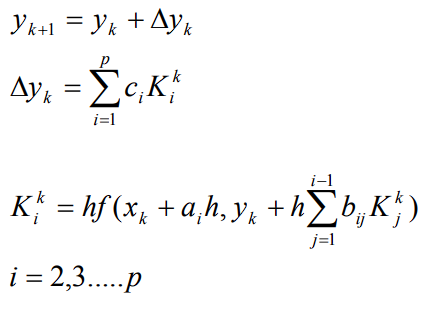
**Метод Эйлера (явный).** График функции, которая является решением задачи Коши, представляет собой гладкую кривую, проходящую через точку согласно условию , и имеет в этой точке касательную. Тангенс угла наклона касательной к оси Ох равен значению производной от решения в точке и равен значению правой части дифференциального уравнения в точке согласно выражению .



Из прямоугольного треугольника находим или . Учитывая, что , и заменяя производную на правую часть дифференциального уравнения, получаем

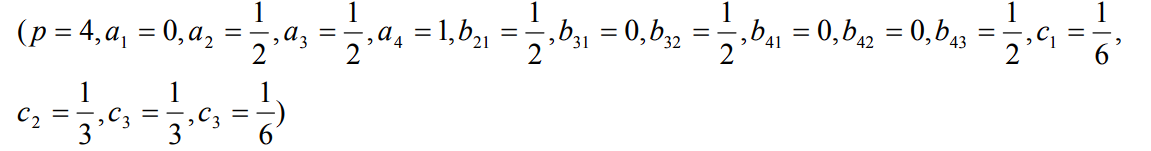
Переход в произвольные индексы дает формулу

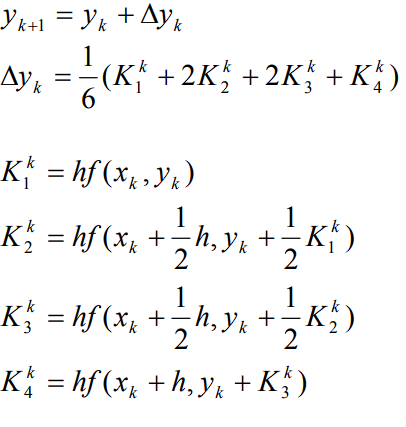
**Метод Рунге-Кутты.** Семейство явных методов Рунге-Кутты р-го порядка записывается в виде совокупности формул:



Параметры подбираются так, чтобы значение , рассчитанное по соотношению выше совпадало со значением разложения в точке точного решения в ряд Тейлора с погрешностью .

**Метод Рунге-Кутты четвертого порядка точности.**

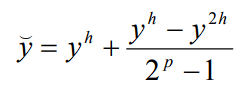




**Контроль точности на каждом шаге h.**

Основным способом контроля точности получаемого численного решения при решении задачи Коши является методы основанные на принципе Рунге-Ромберга-Ричардсона.

Пусть решение задачи Коши, полученное методом Рунге-Кутты p – го порядка точности с шагом в точке . Пусть решение той же задачи в точке , полученное тем же методом, но с шагом . Тогда выражение

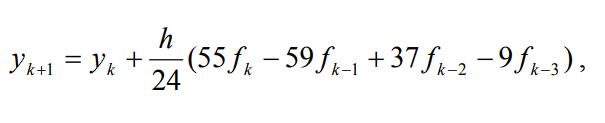


аппроксимирует точное решение в точке с -ым порядком.

Второе слагаемое в выражении оценивает главный член в погрешности решения . Контроль точности может быть организован следующим образом. Выбирается значение шага и дважды рассчитывается решение в точке , один раз с шагом , другой раз с шагом . Рассчитывается величина погрешности и сравнивается с заданной точностью ε. Если величина меньше ε, то можно продолжать вычисления с тем же шагом, в противном случае необходимо вернуться к решению в точке , уменьшить шаг и повторить вычисления.

**Метод Адамса.**

При использовании интерполяционного многочлена 3-ей степени построенного по значениям подынтегральной функции в последних четырех узлах получим метод Адамса четвертого порядка точности:



где значение подынтегральной функции в узле .

Метод Адамса, 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 решение известно из начальных условий, а в других трех узлах решения можно получить с помощью подходящего одношагового метода, например: метода Рунге-Кутты четвертого порядка.

**Метод стрельбы.** Суть метода заключена в многократном решении задачи Коши для приближенного нахождения решения краевой задачи. Пусть надо решить краевую задачу на отрезке . Вместо исходной задачи формулируется задача Коши с уравнением и с начальными условиями

Изображение выглядит как Шрифт, текст, белый, типография

Контент, сгенерированный ИИ, может содержать ошибки.

где - некоторое значение тангенса угла наклона касательной к решению в точке .

Положим сначала некоторое начальное значение параметру , после чего решим каким-либо методом задачу Коши. Пусть решение этой задачи на интервале , тогда сравнивая значение функции со значением в правом конце отрезка можно получить информацию для корректировки угла наклона касательной к решению в левом конце отрезка. Задачу можно сформулировать таким образом: требуется найти такое значение переменной , чтобы решение в правом конце отрезка совпало со значением .

Cледует использовать метод секущих, в котором производная от функции заменена ее разностным аналогом. Данный разностный аналог легко вычисляется по двум приближениям, например и . Следующее значение искомого корня определяется по соотношению

Изображение выглядит как текст, Шрифт, рукописный текст, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Итерации по этой формуле выполняются до удовлетворения заданной точности.

**Конечно-разностный метод.** Рассмотрим двухточечную краевую задачу для линейного дифференциального уравнения второго порядка на отрезке :

Изображение выглядит как Шрифт, текст, каллиграфия, белый

Контент, сгенерированный ИИ, может содержать ошибки.

Введем разностную сетку на отрезке Решение задачи будем искать в виде сеточной функции , предлагая, что решение существует и единственно. Введем разностную аппроксимацию производных следующим образом:

Изображение выглядит как Шрифт, рукописный текст, текст, каллиграфия

Контент, сгенерированный ИИ, может содержать ошибки.

Подставляя аппроксимации производных, получим систему уравнений для нахождения

Изображение выглядит как текст, Шрифт, линия, диаграмма

Контент, сгенерированный ИИ, может содержать ошибки.

Приводя подобные и учитывая, что при задании граничных условий первого рода два неизвестных 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 случае использования граничных условий второго и третьего рода аппроксимация производных проводится с помощью односторонних разностей первого и второго порядков.

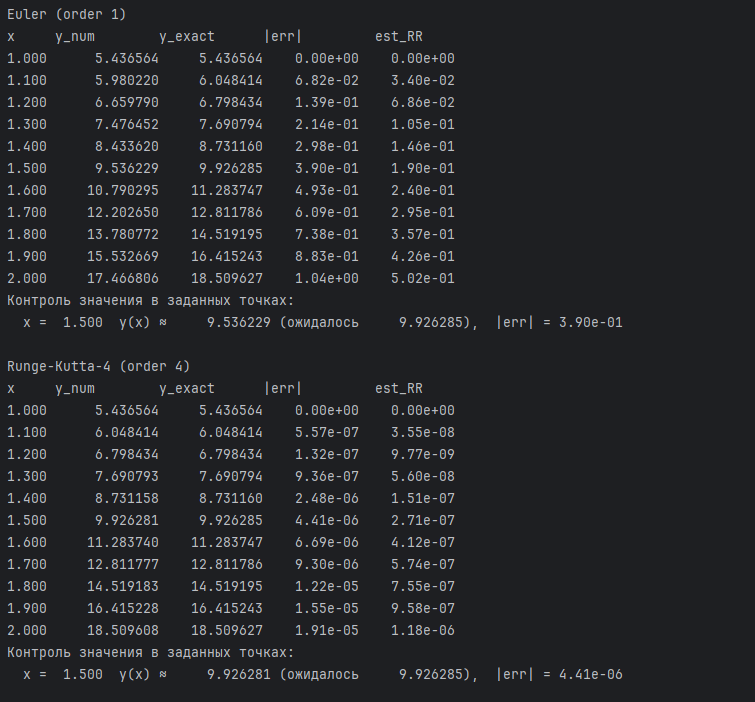
Изображение выглядит как текст, рукописный текст, Шрифт, число

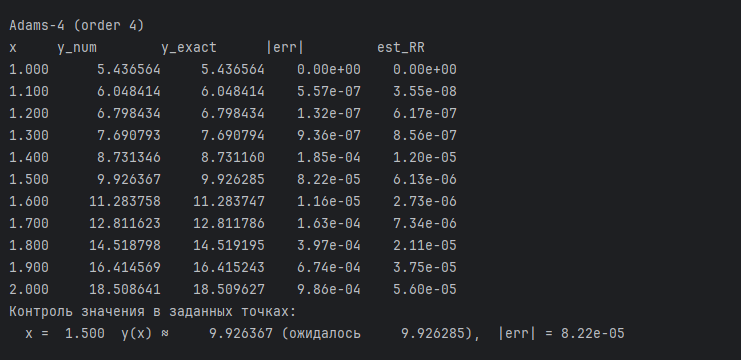
Контент, сгенерированный ИИ, может содержать ошибки.

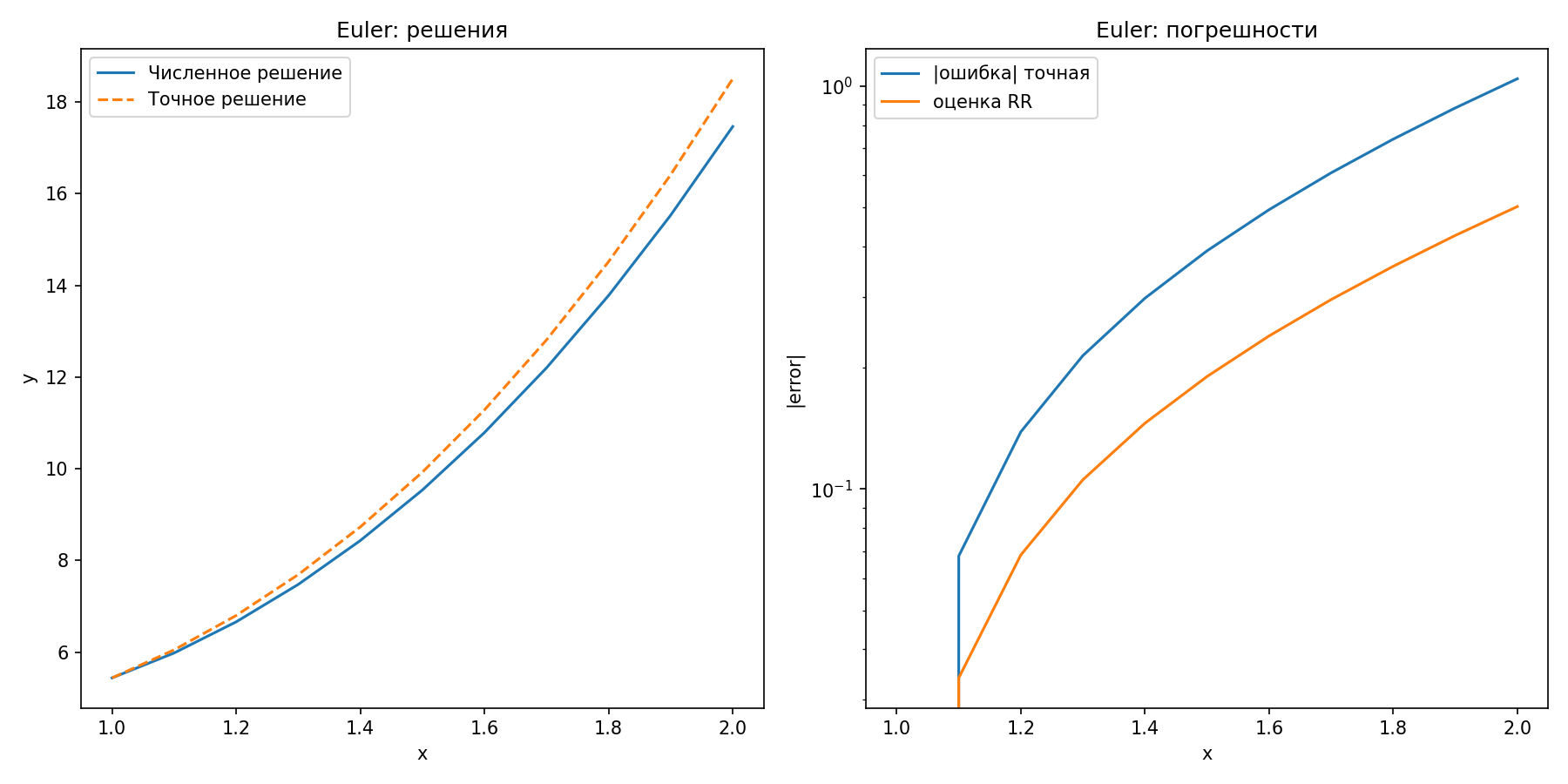
В случае использования двух первых формул линейная алгебраическая система аппроксимирует дифференциальную задачу в целом только с первым порядком (из-за аппроксимации в граничных точках), однако сохраняется трех диагональная структура матрицы коэффициентов. В случае использования двух последних формул второй порядок аппроксимации сохраняется везде, но матрица линейной системы не трехдиагональная.

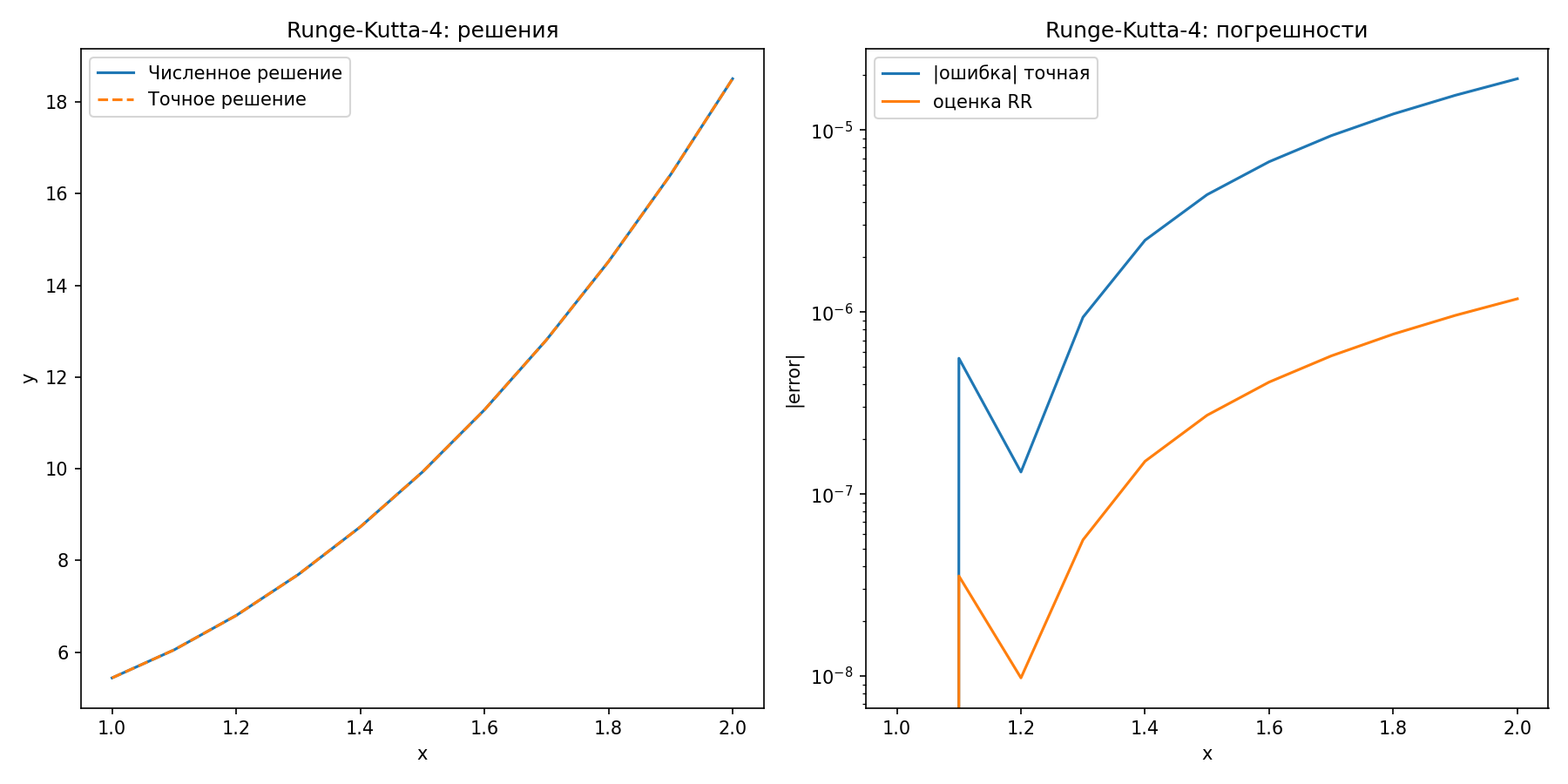
# **Демонстрация работы программы**

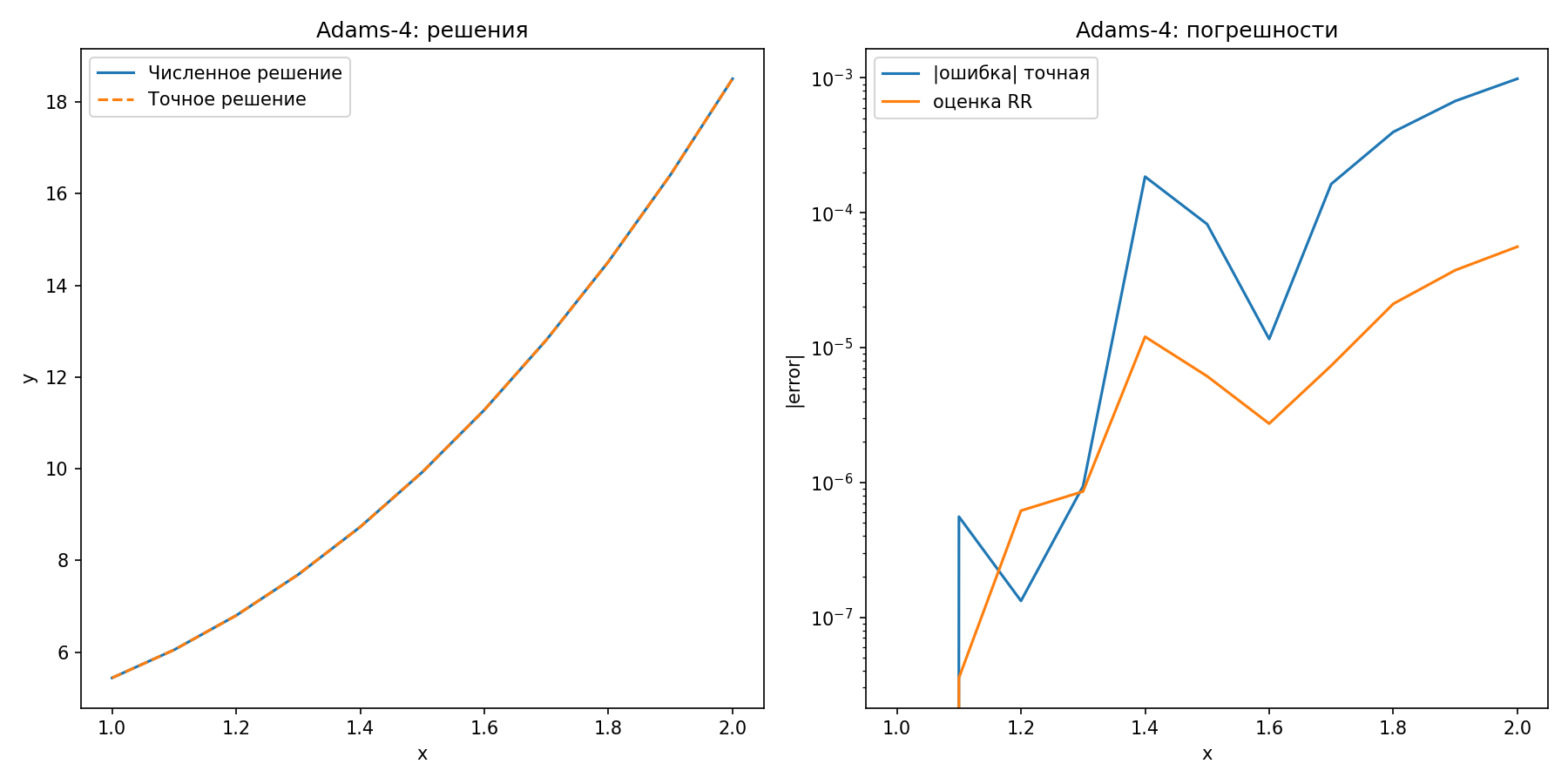
4.1) Численные методы решения задачи Коши



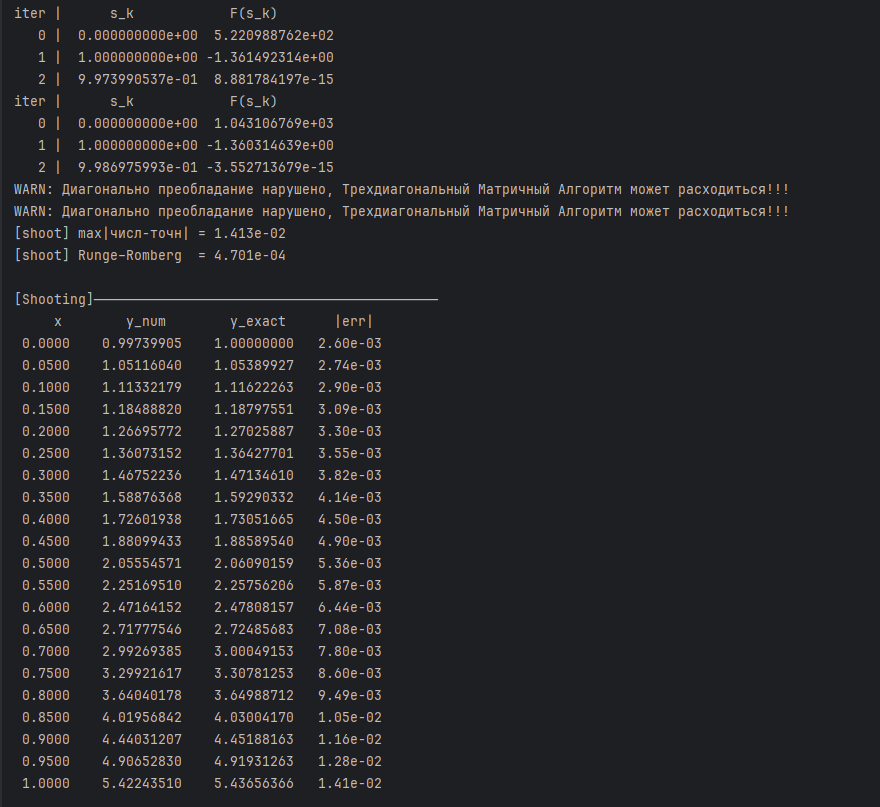


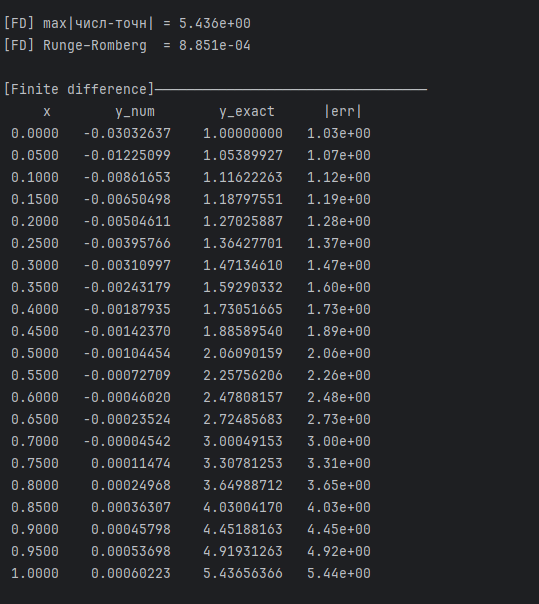


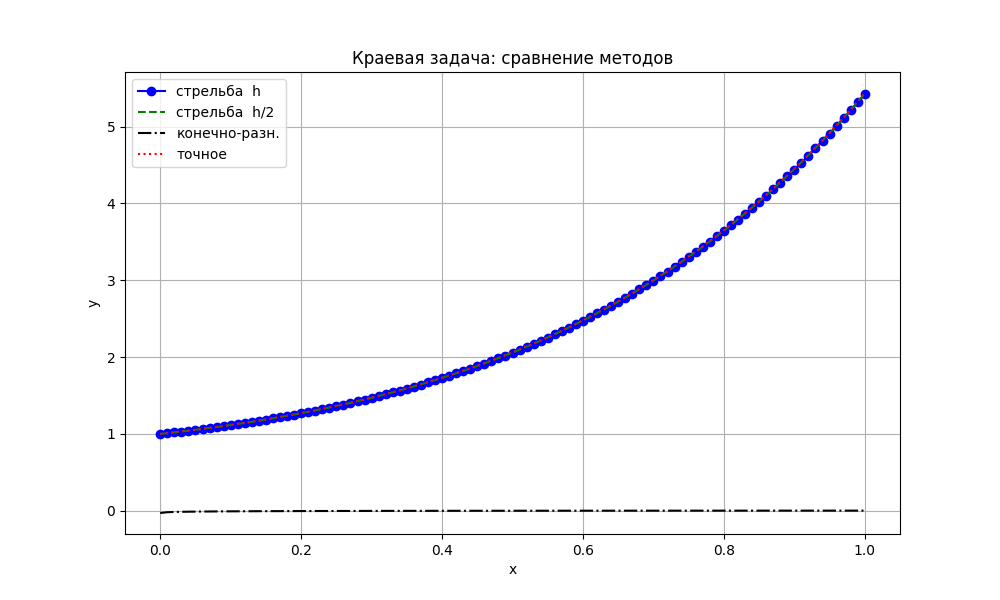




4.2) Методы решения краевой задачи







# **Исходный код программы**

Код был реализован на языке Python.

4.1) Численные методы решения задачи Коши

from \_\_future\_\_ import annotations  
  
import math  
from dataclasses import dataclass  
from typing import Callable, Tuple  
  
import matplotlib.pyplot as plt  
import numpy as np  
  
  
def f2(x, y1, y2):  
 # y'' = (1/x^(1/2))y' - (1/(4x^2))(x + x^(1/2) - 8)y  
 return (1.0 / np.sqrt(x)) \* y2 - (1.0 / (4.0 \* x \* x)) \* (x + np.sqrt(x) - 8.0) \* y1  
  
  
def y\_exact(x: float | np.ndarray) -> float | np.ndarray:  
 return (x \*\* 2 + 1.0 / x) \* np.exp(np.sqrt(x))  
  
  
def \_initial\_state() -> Tuple[float, float, float]:  
 # Начальные значения: x=1, y=2e, y'=2e  
 return 1.0, 2.0 \* math.e, 2.0 \* math.e  
  
  
def \_make\_grid(h: float, x\_end: float = 2.0, \*,  
 min\_nodes: int = 2) -> np.ndarray:  
 if h <= 0:  
 raise ValueError("шаг h должен быть > 0")  
  
 q, r = divmod(x\_end - 1.0, h)  
 if r > 1e-12 and abs(r - h) > 1e-12:  
 raise ValueError(  
 f"длина отрезка {x\_end - 1.0} не кратна шагу h={h} (остаток {r:g})"  
 )  
  
 n = int(round((x\_end - 1.0) / h))  
  
 if n + 1 < min\_nodes:  
 raise ValueError(  
 f"нужно ≥{min\_nodes} узлов, а получается {n + 1}"  
 )  
  
 return np.linspace(1.0, x\_end, n + 1)  
  
  
def euler(h: float, x\_end: float = 2.0):  
 # Явный метод Эйлера для системы уравнений  
 x = \_make\_grid(h, x\_end)  
 y1 = np.empty\_like(x)  
 y2 = np.empty\_like(x)  
  
 xi, y1i, y2i = \_initial\_state()  
 y1[0], y2[0] = y1i, y2i  
  
 for k in range(1, len(x)):  
 y1i, y2i = y1i + h \* y2i, y2i + h \* f2(xi, y1i, y2i)  
 xi += h  
 y1[k], y2[k] = y1i, y2i  
  
 return x, y1, y2  
  
  
def rk4(h: float, x\_end: float = 2.0):  
 # Метод Рунге–Кутта 4-го порядка  
 x = \_make\_grid(h, x\_end)  
 y1 = np.empty\_like(x)  
 y2 = np.empty\_like(x)  
  
 xi, y1i, y2i = \_initial\_state()  
 y1[0], y2[0] = y1i, y2i  
  
 for k in range(1, len(x)):  
 k1y1, k1y2 = y2i, f2(xi, y1i, y2i)  
  
 k2y1 = y2i + 0.5 \* h \* k1y2  
 k2y2 = f2(xi + 0.5 \* h, y1i + 0.5 \* h \* k1y1, y2i + 0.5 \* h \* k1y2)  
  
 k3y1 = y2i + 0.5 \* h \* k2y2  
 k3y2 = f2(xi + 0.5 \* h, y1i + 0.5 \* h \* k2y1, y2i + 0.5 \* h \* k2y2)  
  
 k4y1 = y2i + h \* k3y2  
 k4y2 = f2(xi + h, y1i + h \* k3y1, y2i + h \* k3y2)  
  
 y1i += (h / 6.0) \* (k1y1 + 2 \* k2y1 + 2 \* k3y1 + k4y1)  
 y2i += (h / 6.0) \* (k1y2 + 2 \* k2y2 + 2 \* k3y2 + k4y2)  
 xi += h  
  
 y1[k], y2[k] = y1i, y2i  
  
 return x, y1, y2  
  
  
def adams(h: float, x\_end: float = 2.0):  
 # Явный многошаговый метод Адамса 4-го порядка  
 min\_nodes = 5  
 x = \_make\_grid(h, x\_end, min\_nodes=min\_nodes)  
 y1 = np.empty\_like(x)  
 y2 = np.empty\_like(x)  
  
 xi, y1i, y2i = \_initial\_state()  
 y1[0], y2[0] = y1i, y2i  
  
 f\_hist = []  
 for k in range(1, 4):  
 k1y1, k1y2 = y2i, f2(xi, y1i, y2i)  
 k2y1 = y2i + 0.5 \* h \* k1y2  
 k2y2 = f2(xi + 0.5 \* h, y1i + 0.5 \* h \* k1y1, y2i + 0.5 \* h \* k1y2)  
 k3y1 = y2i + 0.5 \* h \* k2y2  
 k3y2 = f2(xi + 0.5 \* h, y1i + 0.5 \* h \* k2y1, y2i + 0.5 \* h \* k2y2)  
 k4y1 = y2i + h \* k3y2  
 k4y2 = f2(xi + h, y1i + h \* k3y1, y2i + h \* k3y2)  
 y1i += (h / 6.0) \* (k1y1 + 2 \* k2y1 + 2 \* k3y1 + k4y1)  
 y2i += (h / 6.0) \* (k1y2 + 2 \* k2y2 + 2 \* k3y2 + k4y2)  
 xi += h  
 y1[k], y2[k] = y1i, y2i  
 f\_hist.append(f2(xi, y1i, y2i))  
  
 for k in range(4, len(x)):  
 f\_n, f\_nm1, f\_nm2 = f\_hist[-1], f\_hist[-2], f\_hist[-3]  
 f\_nm3 = f2(x[k - 4], y1[k - 4], y2[k - 4])  
 y1i += (h / 24.0) \* (55 \* y2i - 59 \* y2[k - 2] + 37 \* y2[k - 3] - 9 \* y2[k - 4])  
 y2i += (h / 24.0) \* (55 \* f\_n - 59 \* f\_nm1 + 37 \* f\_nm2 - 9 \* f\_nm3)  
 xi += h  
 y1[k], y2[k] = y1i, y2i  
 f\_hist.append(f2(xi, y1i, y2i))  
  
 return x, y1, y2  
  
  
def compute\_errors(method: Callable[[float], Tuple[np.ndarray, np.ndarray, np.ndarray]], h: float, p: int):  
 x\_coarse, y1\_coarse, \_ = method(h)  
 x\_fine, y1\_fine, \_ = method(h / 2)  
  
 y\_fine\_on\_coarse = y1\_fine[::2]  
 exact = y\_exact(x\_coarse)  
  
 err\_exact = np.abs(y1\_coarse - exact)  
 err\_est = np.abs(y\_fine\_on\_coarse - y1\_coarse) / (2 \*\* p - 1)  
  
 return np.column\_stack((x\_coarse, y1\_coarse, exact, err\_exact, err\_est))  
  
  
def plot\_results(name: str, tbl: np.ndarray):  
 x, y\_num, y\_ex, err\_ex, err\_est = tbl.T  
  
 fig = plt.figure(figsize=(12, 6))  
  
 ax1 = fig.add\_subplot(1, 2, 1)  
 ax1.plot(x, y\_num, label="Численное решение")  
 ax1.plot(x, y\_ex, "--", label="Точное решение")  
 ax1.set\_xlabel("x")  
 ax1.set\_ylabel("y")  
 ax1.set\_title(f"{name}: решения")  
 ax1.legend()  
  
 ax2 = fig.add\_subplot(1, 2, 2)  
 ax2.semilogy(x, err\_ex, label="|ошибка| точная")  
 ax2.semilogy(x, err\_est, label="оценка RR")  
 ax2.set\_xlabel("x")  
 ax2.set\_ylabel("|error|")  
 ax2.set\_title(f"{name}: погрешности")  
 ax2.legend()  
  
 fig.tight\_layout()  
 fig.savefig(name.replace(" ", "\_") + ".png", dpi=150)  
  
  
@dataclass(slots=True)  
class Method:  
 func: Callable[[float], Tuple[np.ndarray, np.ndarray, np.ndarray]]  
 order: int  
  
  
METHODS = {  
 "Euler": Method(euler, 1),  
 "Runge-Kutta-4": Method(rk4, 4),  
 "Adams-4": Method(adams, 4),  
}  
  
CONTROL\_POINTS = [  
 (1.5, (1.5 \*\* 2 + 1 / 1.5) \* np.exp(np.sqrt(1.5))),  
]  
  
  
def check\_points(method\_func: Callable[[float], Tuple[np.ndarray, np.ndarray, np.ndarray]],  
 h: float):  
 x, y1, \_ = method\_func(h)  
  
 for xc, y\_expected in CONTROL\_POINTS:  
 y\_interp = np.interp(xc, x, y1)  
 err = abs(y\_interp - y\_expected)  
 print(f" x = {xc:6.3f} y(x) ≈ {y\_interp:12.6f} "  
 f"(ожидалось {y\_expected:12.6f}), |err| = {err:.2e}")  
  
  
def main():  
 h = 0.1  
  
 for name, m in METHODS.items():  
 tbl = compute\_errors(m.func, h, m.order)  
  
 print(f"\n{name} (order {m.order})")  
 print("x y\_num y\_exact |err| est\_RR")  
 for row in tbl:  
 x\_i, y\_n, y\_ex, e\_ex, e\_rr = row  
 print(f"{x\_i:3.3f} {y\_n:12.6f} {y\_ex:12.6f} {e\_ex:11.2e} {e\_rr:11.2e}")  
  
 print("Контроль значения в заданных точках:")  
 check\_points(m.func, h)  
  
 plot\_results(name, tbl)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

4.2) Методы решения краевой задачи

from \_\_future\_\_ import annotations  
  
import numpy as np  
import matplotlib.pyplot as plt  
from typing import Sequence, List, Tuple  
  
# Параметры задачи  
a: float = 0.0  
b: float = 1.0  
alpha: float = 1.0 # y'(0) = 1  
beta: float = 0.0 # y'(1) - 2y(1) = 0  
  
def y\_exact(x: float | np.ndarray) -> float | np.ndarray:  
 *"""  
 Точное решение краевой задачи:  
 y(x) = e^x (x^2 + 1)  
 """* return np.exp(x) \* (x\*\*2 + 1.0)  
  
def rhs(x: float, Y: Sequence[float]) -> Tuple[float, float]:  
 *"""  
 Правая часть системы первого порядка:  
 y1' = y2,  
 y2' = [(2x+1)/x]y2 - [(x+1)/x]y1  
 """* y1, y2 = Y  
 if np.isclose(x, 0.0):  
 # При x=0 используем предельные значения  
 return y2, (3.0\*y2 - y1) # Лопиталь для x->0  
 return y2, ((2.0\*x + 1.0)/x)\*y2 - ((x + 1.0)/x)\*y1  
  
def rk4\_step(f, x: float, Y: Sequence[float], h: float) -> List[float]:  
 *"""  
 Один шаг классического РК-4:  
 k1 = f(x, Y)  
 k2 = f(x+h/2,Y + h/2·k1)  
 k3 = f(x+h/2,Y + h/2·k2)  
 k4 = f(x+h, Y + h·k3)  
 Новое Y = Y + h/6·(k1+2k2+2k3+k4)  
 """* k1 = f(x, Y)  
 k2 = f(x + h/2, [Y[i] + h \* k1[i] / 2 for i in range(2)])  
 k3 = f(x + h/2, [Y[i] + h \* k2[i] / 2 for i in range(2)])  
 k4 = f(x + h, [Y[i] + h \* k3[i] for i in range(2)])  
 return [Y[i] + h \* (k1[i] + 2\*k2[i] + 2\*k3[i] + k4[i]) / 6 for i in range(2)]  
  
def solve\_ivp(f, x0: float, Y0: Sequence[float], grid: np.ndarray) -> list[list[float]]:  
 *"""  
 Универсальная обёртка: решает задачу Коши y'=f(x,y) методом RK-4 по точкам grid.  
 Возвращает список [y1,y2] в каждом узле.  
 """* Y = list(Y0)  
 sol = [Y.copy()]  
 for k in range(len(grid) - 1):  
 h = grid[k + 1] - grid[k]  
 Y = rk4\_step(f, grid[k], Y, h)  
 sol.append(Y.copy())  
 return sol  
  
def shooting(a: float, b: float,  
 alpha: float, beta: float,  
 h: float, tol: float = 1e-12,  
 max\_it = 50):  
 *"""  
 Метод стрельбы для краевой задачи:  
 — Интегрируем систему с неизвестным начальным y1(a)=s  
 — Подбираем s секущим методом так, чтобы на правом конце условие  
 y'(b) − 2y(b) = beta выполнялось с точностью tol.  
 """* x\_grid = np.arange(a, b + 1e-13, h)  
  
 def residual(s: float) -> float:  
 # Невязка на правом краю  
 y\_end, yprime\_end = solve\_ivp(rhs, a, (s, alpha), x\_grid)[-1]  
 return yprime\_end - 2.0\*y\_end - beta  
  
 # Две начальные догадки для секущего метода  
 s0, s1 = 0.0, 1.0  
 F0, F1 = residual(s0), residual(s1)  
 it = 0  
 print("iter | s\_k F(s\_k)")  
 print(f"{it:4d} | {s0:16.9e} {F0:16.9e}")  
 it += 1  
 print(f"{it:4d} | {s1:16.9e} {F1:16.9e}")  
  
 while abs(F1) > tol and it < max\_it:  
 s0, F0, s1 = s1, F1, s1 - F1 \* (s1 - s0) / (F1 - F0) # секущая  
 F1 = residual(s1)  
 it += 1  
 print(f"{it:4d} | {s1:16.9e} {F1:16.9e}")  
  
 if abs(F1) > tol:  
 raise RuntimeError("shooting: не сошлось за max\_it итераций")  
  
 sol = solve\_ivp(rhs, a, (s1, alpha), x\_grid)  
 return x\_grid, np.array([y[0] for y in sol])  
  
def thomas\_algorithm(a, b, c, f):  
 *"""  
 a[i] = элемент под диагональю (i-й строки, i-1-й столбец)  
 b[i] = диагональ  
 c[i] = над диагональю  
 f[i] = правая часть  
 Результат: x, решение системы 3-диагональной  
 """* n = len(b)  
 if any(len(arr) != n for arr in [a, c, f]):  
 raise ValueError("Thomas: несовместимые размеры")  
 alpha = [0]\*n  
 beta = [0]\*n  
  
 # Диагональное преобладание (|b\_i| ≥ |a\_i|+|c\_i|)  
 warn\_printed = False  
 for i in range(n):  
 s = abs(a[i]) + abs(c[i])  
 if 1 <= i <= n-2 and abs(b[i]) < abs(a[i])+abs(c[i]):  
 # raise ValueError("Диагональное преобладание нарушено")  
 if not warn\_printed:  
 print("WARN: Диагонально преобладание нарушено, Трехдиагональный Матричный Алгоритм может расходиться!!!")  
 warn\_printed = True  
 else:  
 pass  
  
 # Прямая прогонка  
 alpha[0] = -c[0]/b[0]  
 beta[0] = f[0]/b[0]  
  
 for i in range(1, n):  
 denom = b[i] + a[i]\*alpha[i-1]  
 if abs(denom) < 1e-15:  
 raise ValueError("Thomas: вырожденная или некорректная матрица")  
 alpha[i] = -c[i]/denom  
 beta[i] = (f[i] - a[i]\*beta[i-1]) / denom  
  
 # Обратная прогонка  
 x = [0]\*n  
 x[n-1] = beta[n-1]  
 for i in range(n-2, -1, -1):  
 x[i] = alpha[i]\*x[i+1] + beta[i]  
  
 return x  
  
def finite\_difference(a: float, b: float,  
 alpha: float, beta: float,  
 h: float):  
 *"""  
 xy'' + (2x+1)y' - (x+1)y = 0  
 y'(0)=α, y'(1) - 2y(1) = β  
 """* x = np.arange(a, b + h\*0.5, h)  
 N = len(x) - 1  
  
 A = np.zeros(N+1)  
 B = np.zeros(N+1)  
 C = np.zeros(N+1)  
 F = np.zeros(N+1)  
  
 # Левое граничное условие: y'(0) = alpha  
 # Используем одностороннюю разностную аппроксимацию вперед:  
 # y'(0) ≈ (y₁ - y₀)/h = alpha  
 B[0] = -1.0/h  
 C[0] = 1.0/h  
 F[0] = alpha  
  
 # Уравнения для внутренних точек (j = 1, 2, ..., N-1)  
 for j in range(1, N):  
 xj = x[j]  
 A[j] = xj/h\*\*2 - (2\*xj + 1)/(2\*h) # Коэф. при y\_{j-1}  
 B[j] = -2\*xj/h\*\*2 + (xj + 1) # Коэф. при y\_j  
 C[j] = xj/h\*\*2 + (2\*xj + 1)/(2\*h) # Коэф. при y\_{j+1}  
  
 # Правое граничное условие: y'(1) - 2y(1) = beta  
 # Используем одностороннюю разностную аппроксимацию назад:  
 # y'(1) ≈ (y\_N - y\_{N-1})/h  
 A[N] = -1.0/h  
 B[N] = 1.0/h - 2.0  
 F[N] = beta  
  
 y = thomas\_algorithm(A, B, C, F)  
 return x, np.asarray(y)  
  
def runge\_romberg(y\_h: np.ndarray, y\_h2: np.ndarray, p: int = 4) -> float:  
 *"""  
 Оценка погрешности по методу Рунге–Ромберга:  
 max|y(h/2) − y(h)|/(2^p − 1)  
 """* return np.max(np.abs(y\_h2[::2] - y\_h)) / (2 \*\* p - 1)  
  
def main():  
 h = (b - a) / 100  
 h2 = h / 2  
  
 x\_h, y\_h = shooting(a, b, alpha, beta, h)  
 x\_h2, y\_h2 = shooting(a, b, alpha, beta, h2)  
  
 x\_fd, y\_fd = finite\_difference(a, b, alpha, beta, h)  
 x\_fd2, y\_fd2 = finite\_difference(a, b, alpha, beta, h2)  
  
 err\_exact = np.max(np.abs(y\_h - y\_exact(x\_h)))  
 err\_rr = runge\_romberg(y\_h, y\_h2)  
 err\_exactfd = np.max(np.abs(y\_fd - y\_exact(x\_fd)))  
 err\_rrfd = runge\_romberg(y\_fd, y\_fd2)  
 print(f"[shoot] max|числ-точн| = {err\_exact:.3e}")  
 print(f"[shoot] Runge–Romberg = {err\_rr:.3e}")  
  
 print("\n[Shooting]".ljust(54, "─"))  
 print(" x y\_num y\_exact |err|")  
 for k in range(0, len(x\_h), 5):  
 print(f"{x\_h[k]:7.4f} {y\_h[k]:12.8f} "  
 f"{y\_exact(x\_h[k]):12.8f} "  
 f"{abs(y\_h[k] - y\_exact(x\_h[k])):9.2e}")  
  
 print(f"\n[FD] max|числ-точн| = {err\_exactfd:.3e}")  
 print(f"[FD] Runge–Romberg = {err\_rrfd:.3e}")  
  
 print("\n[Finite difference]".ljust(54, "─"))  
 print(" x y\_num y\_exact |err|")  
 for k in range(0, len(x\_fd), 5):  
 print(f"{x\_fd[k]:7.4f} {y\_fd[k]:12.8f} "  
 f"{y\_exact(x\_fd[k]):12.8f} "  
 f"{abs(y\_fd[k] - y\_exact(x\_fd[k])):9.2e}")  
  
 plt.figure(figsize=(10, 6))  
 plt.plot(x\_h, y\_h, "bo-", label="стрельба h")  
 plt.plot(x\_h2, y\_h2, "g--", label="стрельба h/2")  
 plt.plot(x\_fd, y\_fd, "k-.", label="конечно-разн.")  
 plt.plot(x\_h, y\_exact(x\_h), "r:", label="точное")  
 plt.title("Краевая задача: сравнение методов")  
 plt.xlabel("x"); plt.ylabel("y"); plt.grid(True); plt.legend()  
 plt.savefig("bvp\_compare.png", dpi=150)  
 plt.show()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

# **Выводы**

В данной лабораторной работе были освоены и реализованы численные методы решения обыкновенных дифференциальных уравнений. Для задачи Коши были применены методы Эйлера, Рунге-Кутты 4-го порядка и Адамса 4-го порядка, их точность сравнивалась с аналитическим решением, подтвердив преимущество методов высшего порядка. Далее были исследованы методы решения краевых задач: метод стрельбы, который итерационно подбирал недостающее начальное условие, и конечно-разностный метод, аппроксимирующий производные и приводящий к системе линейных уравнений. Метод стрельбы использовал метод Рунге-Кутты для решения промежуточных задач Коши. Конечно-разностный метод свел задачу к решению СЛАУ с трехдиагональной матрицей. Результаты всех методов были визуализированы, что позволило наглядно оценить их эффективность и точность.