Санкт-Петербургский национальный исследовательский университет
информационных технологий, механики и оптики

Расчетно графическая работа №1

Выполнили:

Левахин Лев

Останин Андрей

Дощенников Никита

Группы: К3221, К3240

Проверил:

Владимир Владимирович Беспалов

Санкт-Петербург
2025

Цель

Изучение и практическое применение численных методов решения обыкновенных дифференциальных уравнений, сравнение их точности и устойчивости на различных типах задач.

Задачи

1. Реализовать следующие численные методы решения ОДУ:
   - Явный метод Эйлера (1-го порядка)
   - Явный метод трапеций (улучшенный Эйлер, 2-го порядка)
   - Классический метод Рунге-Кутты 4-го порядка (RK4)
   - Неявный метод Эйлера (1-го порядка)
   - Неявный метод трапеций (2-го порядка)
2. Решить две задачи:
   - Нежесткое нелинейное уравнение типа Риккати (`1.4`): исследование точности и сходимости явных методов
   - Жесткое линейное уравнение (`2.4`): исследование устойчивости неявных методов
3. Провести анализ погрешностей и построить графики решений
4. Сформулировать выводы о применимости различных методов

Теоретические основы

Постановка задачи Коши

Рассматривается начальная задача для обыкновенного дифференциального уравнения первого порядка:

$$\frac{dy}{dx} = f(t, y), \quad y(t_0) = y_0$$

где

- t - независимая переменная (время)
- $y(t)$ - искомая функция
- $f(t, y)$ - правая часть уравнения
- y_0 - начальное условие

Классификация численных методов

Явные методы - значение решения на следующем шаге вычисляется непосредственно из известных значений:

- Просты в реализации
- Требуют малого шага для жестких систем
- Примеры: явный Эйлер, RK4

Неявные методы - значение решения на следующем шаге находится из неявного уравнения:

- Требуют решения нелинейных уравнений на каждом шаге
- Обладают лучшей устойчивостью для жестких систем
- Примеры: неявный Эйлер, неявная трапеция

Описание реализованных методов

Явный метод Эйлера

$$y_{n+1} = y_n + h \cdot f(t_n, y_n)$$

Имеет первый порядок точности $O(h)$.

```
def euler_method(self, f, t_span, y0, h):
    t0, t_end = t_span
    N = int((t_end - t0) / h)
    t = np.linspace(t0, t_end, N + 1)
    y = np.zeros(N + 1)
    y[0] = y0

    for i in range(N):
        y[i + 1] = y[i] + h * f(t[i], y[i])

    return t, y
```

Метод использует линейную аппроксимацию решения на основе производной в начале интервала. Это самый простой, но наименее точный метод.

Явный метод трапеций

$$k_1 = f(t_n, y_n)$$
$$k_2 = f(t_n + h, y_n + h \cdot k_1)$$
$$y_{n+1} = y_n + \frac{h}{2} \cdot (k_1 + k_2)$$

Имеет второй порядок точности $O(h^2)$.

```
def explicit_trapezoid(self, f, t_span, y0, h):
    t0, t_end = t_span
    N = int((t_end - t0) / h)
    t = np.linspace(t0, t_end, N + 1)
    y = np.zeros(N + 1)
    y[0] = y0

    for i in range(N):
        k1 = f(t[i], y[i])
        k2 = f(t[i] + h, y[i] + h * k1)
        y[i + 1] = y[i] + (h / 2) * (k1 + k2)

    return t, y
```

Метод сначала делает предсказание решения в конце интервала (явным Эйлером), затем усредняет производные в начале и конце интервала.

Метод Рунге-Кутты 4-го порядка (RK4)

$$k_1 = f(t_n, y_n)$$
$$k_2 = f\left(t_n + \frac{h}{2}, y_n + h \cdot \frac{k_1}{2}\right)$$
$$k_3 = f\left(t_n + \frac{h}{2}, y_n + h \cdot \frac{k_2}{2}\right)$$
$$k_4 = f(t_n + h, y_n + h \cdot k_3)$$
$$y_{n+1} = y_n + \frac{h}{6} \cdot (k_1 + 2k_2 + 2k_3 + k_4)$$

Имеет четвертый порядок точности $O(h^4)$

```
def rk4_method(self, f, t_span, y0, h):
    t0, t_end = t_span
    N = int((t_end - t0) / h)
    t = np.linspace(t0, t_end, N + 1)
    y = np.zeros(N + 1)
    y[0] = y0

    for i in range(N):
        k1 = f(t[i], y[i])
        k2 = f(t[i] + h/2, y[i] + h/2 * k1)
        k3 = f(t[i] + h/2, y[i] + h/2 * k2)
        k4 = f(t[i] + h, y[i] + h * k3)
        y[i + 1] = y[i] + h/6 * (k1 + 2*k2 + 2*k3 + k4)

    return t, y
```

Метод вычисляет четыре промежуточных значения производной внутри интервала и формирует взвешенное среднее. Это обеспечивает высокую точность.

Неявный метод Эйлера

$$y_{n+1} = y_n + h \cdot f(t_{n+1}, y_{n+1})$$

Имеет первый порядок точности $O(h)$

```
def backward_euler(self, f, t_span, y0, h):
    t0, t_end = t_span
    N = int((t_end - t0) / h)
    t = np.linspace(t0, t_end, N + 1)
    y = np.zeros(N + 1)
    y[0] = y0

    for i in range(N):
        t_next = t[i + 1]

        def equation(z):
            return z - h * f(t_next, z) - y[i]

        y[i + 1] = fsolve(equation, y[i])[0]

    return t, y
```

Метод использует производную в конце интервала, что требует решения нелинейного уравнения на каждом шаге с помощью `fsolve`. Это обеспечивает безусловную устойчивость для линейных задач.

Неявный метод трапеций

$$y_{n+1} = y_n + \frac{h}{2} \cdot [f(t_n, y_n) + f(t_{n+1}, t_{n+1}]$$

Имеет второй порядок точности $O(h^2)$

```
def implicit_trapezoid(self, f, t_span, y0, h):
    t0, t_end = t_span
    N = int((t_end - t0) / h)
    t = np.linspace(t0, t_end, N + 1)
    y = np.zeros(N + 1)
    y[0] = y0

    for i in range(N):
        t_n = t[i]
        t_np1 = t[i + 1]
        y_n = y[i]

        def equation(z):
            return z - y_n - (h/2) * (f(t_n, y_n) + f(t_np1,
z))

        y[i + 1] = fsolve(equation, y_n)[0]

    return t, y
```

Метод усредняет производные в начале и конце интервала (как явная трапеция), но использует неизвестное значение y_{n+1}, что требует решения нелинейного уравнения. Метод A-устойчив.

Задачи

1.4

$$y' = y^2 - t, \quad y(0) = 1, \quad t \in [0, 1.5], \quad h = 0.25$$

- Нелинейное уравнение типа Риккати
- Не имеет аналитического решения
- Нежесткая система - подходит для явных методов

- Используется для анализа точности и сходимости

```
def problem_1_4():
    def f(t, y):
        return y**2 - t

    t_span = [0, 1.5]
    y0 = 1
    h = 0.25

    solver = ODESolver()

    t_euler, y_euler = solver.euler_method(f, t_span, y0, h)
    t_trap, y_trap = solver.explicit_trapezoid(f, t_span, y0,
h)
    t_rk4, y_rk4 = solver.rk4_method(f, t_span, y0, h)

    sol_ref = solve_ivp(f, t_span, [y0], method='RK45',
                        rtol=1e-10, atol=1e-12,
dense_output=True)
```

2.4

$$y' = -2000y + t, \quad y(0) = 0, \quad t \in [0, 1], \quad h = 1$$

- Линейное уравнение с большим отрицательным коэффициентом $\lambda = -2000$
- Жесткая система
- Явный метод Эйлера неустойчив при больших шагах
- Требует неявных методов для устойчивости

```
def problem_2_4():
    def f(t, y):
        return -2000 * y + t

    t_span = [0, 1]
    y0 = 0
    h = 1

    solver = ODESolver()

    t_euler, y_euler = solver.euler_method(f, t_span, y0, h)
```

```
    t_beuler, y_beuler = solver.backward_euler(f, t_span, y0,
h)
    t_itrap, y_itrap = solver.implicit_trapezoid(f, t_span,
y0, h)

    sol_ref = solve_ivp(f, t_span, [y0], method='Radau',
                        rtol=1e-10, atol=1e-12,
dense_output=True)
```

Результаты

1.4

Мы уменьшили интервал с $[0, 1.5]$ до $[0, 1]$ для того, чтобы избежать “взрыва решения”, так как уравнение нелинейно.

Результаты программы:

```
=== ЗАДАЧА 1.4: y' = y² - t, y(0) = 1 ===
Ошибка метода Эйлера: 6.44e+00
Ошибка явного метода трапеций: 3.76e+00
Ошибка метода РК4: 6.45e-01
```

Ошибка получилась порядка 10^{-1}.

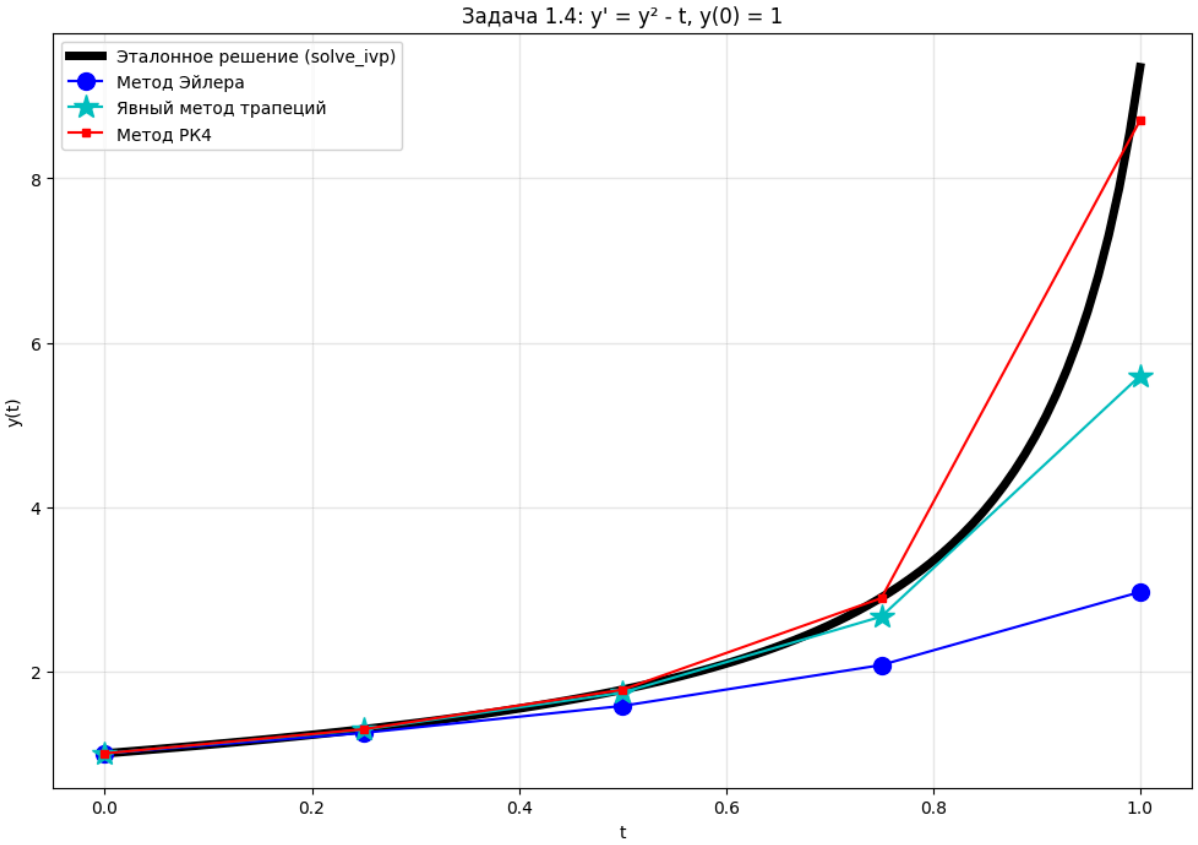

Рис. 1: График решения задачи 1.4

- RK4 самый точный. Ошибка $\sim 10^{-1}$ в разы меньше остальных. На графике траектория почти совпадает с эталоном.
- Явная трапеция имеет среднюю точность. Ошибка меньше, чем у Эйлера, но заметно хуже RK4.
- Метод эйлера самый неточный. Ошибка огромная, отклонение от эталона быстро растёт.

2.4

Результаты программы:

```
=== ЗАДАЧА 2.4: y' = -2000y + t, y(0) = 0 ===
Ошибка явного метода Эйлера: 5.00e-04
Ошибка неявного метода Эйлера: 1.25e-10
Ошибка неявного метода трапеций: 2.50e-07
```

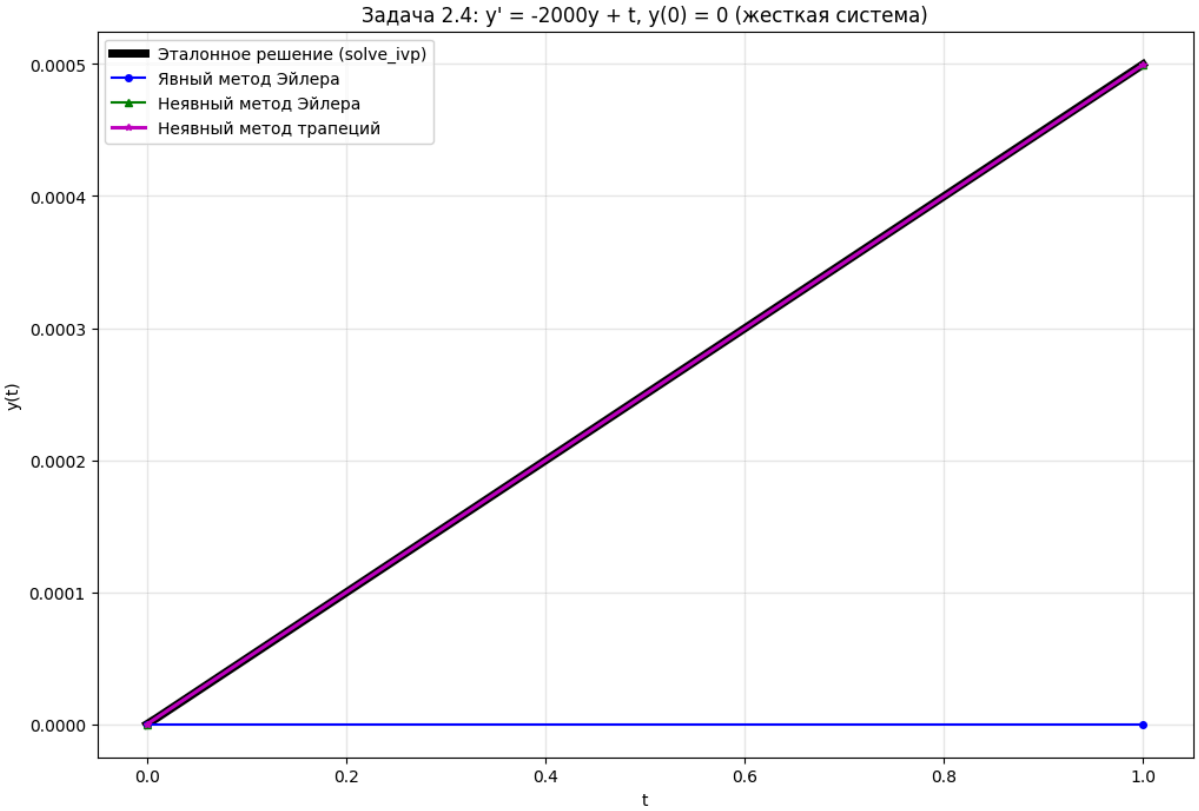

Рис. 2: График решения задачи 2.4

- Неявный Эйлер самый точный и устойчивый. Ошибка $\sim 10^{-10}$.
- Неявная трапеция тоже устойчива, но точность хуже. Ошибка $\sim 10^{-7}$.
- Явный Эйлер нестабилен, даёт неправильную форму решения.

Структура программы

Класс `ODESolver`

Класс инкапсулирует все численные методы:

```
class ODESolver:
    def euler_method(self, f, t_span, y0, h): ...
    def explicit_trapezoid(self, f, t_span, y0, h): ...
    def rk4_method(self, f, t_span, y0, h): ...
    def backward_euler(self, f, t_span, y0, h): ...
    def implicit_trapezoid(self, f, t_span, y0, h): ...
```

Функция решения задач

Каждая задача решается в отдельной функции:

- `problem_1_4()` - для нежесткой задачи

- `problem_2_4()` - для жесткой задачи

Функции выполняют:

1. Определение правой части ОДУ
2. Задание параметров интегрирования
3. Вызов численных методов
4. Получение эталонного решения
5. Вычисление погрешностей
6. Построение графиков

Вычисление эталонного решения

Используются высокоточные методы из библиотеки SciPy:

- `solve_ivp` с методом RK45 для нежестких систем
- `solve_ivp` с методом Radau для жестких систем
- Строгие допуски: `rtol=1e-10`, `atol=1e-12`

Анализ погрешности

Глобальная погрешность вычисляется как норма разности:

```
err = np.linalg.norm(y_numerical - y_reference)
```

где используется евклидова норма L_2.

Выводы

О порядке точности:

- Порядок метода существенно влияет на точность при одинаковом шаге
- Для достижения заданной точности метод более высокого порядка требует меньше вычислений
- RK4 является оптимальным выбором для нежестких задач

О явных и неявных методах:

- Явные методы просты в реализации, но имеют ограничения по устойчивости
- Неявные методы требуют решения нелинейных уравнений, но обеспечивают устойчивость
- Для жестких систем неявные методы необходимы

О жесткости:

- Жесткие системы характеризуются наличием сильно различающихся масштабов времени
- Явные методы требуют непрактично малых шагов для жестких систем
- A-устойчивые неявные методы позволяют использовать большие шаги

Приложение

Полный код программы:

```
import matplotlib.pyplot as plt
import math
import numpy as np
from scipy.integrate import solve_ivp
from scipy.optimize import fsolve


class ODESolver:
    def euler_method(self, f, t_span, y0, h):
        t0, t_end = t_span
        N = int((t_end - t0) / h)
        t = np.linspace(t0, t_end, N + 1)
        y = np.zeros(N + 1)
        y[0] = y0

        for i in range(N):
            y[i + 1] = y[i] + h * f(t[i], y[i])

        return t, y

    def explicit_trapezoid(self, f, t_span, y0, h):
        t0, t_end = t_span
        N = int((t_end - t0) / h)
        t = np.linspace(t0, t_end, N + 1)
        y = np.zeros(N + 1)
        y[0] = y0

        for i in range(N):
            k1 = f(t[i], y[i])
            k2 = f(t[i] + h, y[i] + h * k1)
            y[i + 1] = y[i] + (h / 2) * (k1 + k2)
```

```
        return t, y

    def rk4_method(self, f, t_span, y0, h):
        t0, t_end = t_span
        N = int((t_end - t0) / h)
        t = np.linspace(t0, t_end, N + 1)
        y = np.zeros(N + 1)
        y[0] = y0

        for i in range(N):
            k1 = f(t[i], y[i])
            k2 = f(t[i] + h/2, y[i] + h/2 * k1)
            k3 = f(t[i] + h/2, y[i] + h/2 * k2)
            k4 = f(t[i] + h, y[i] + h * k3)
            y[i + 1] = y[i] + h/6 * (k1 + 2*k2 + 2*k3 + k4)

        return t, y

    def backward_euler(self, f, t_span, y0, h):
        t0, t_end = t_span
        N = int((t_end - t0) / h)
        t = np.linspace(t0, t_end, N + 1)
        y = np.zeros(N + 1)
        y[0] = y0

        for i in range(N):
            t_next = t[i + 1]

            def equation(z):
                return z - h * f(t_next, z) - y[i]

            y[i + 1] = fsolve(equation, y[i])[0]

        return t, y

    def implicit_trapezoid(self, f, t_span, y0, h):
        t0, t_end = t_span
        N = int((t_end - t0) / h)
        t = np.linspace(t0, t_end, N + 1)
        y = np.zeros(N + 1)
        y[0] = y0
```

```
        for i in range(N):
            t_n = t[i]
            t_np1 = t[i + 1]
            y_n = y[i]

            def equation(z):
                return z - y_n - (h/2) * (f(t_n, y_n) +
f(t_np1, z))

            y[i + 1] = fsolve(equation, y_n)[0]

        return t, y

def problem_1_4():
    print("=== ЗАДАЧА 1.4: y' = y² - t, y(0) = 1 ===")

    def f(t, y):
        return y**2 - t

    t_span = [0, 1.0]
    y0 = 1
    h = 0.25

    solver = ODESolver()

    t_euler, y_euler = solver.euler_method(f, t_span, y0, h)
    t_trap, y_trap = solver.explicit_trapezoid(f, t_span, y0,
h)
    t_rk4, y_rk4 = solver.rk4_method(f, t_span, y0, h)

    sol_ref = solve_ivp(f, t_span, [y0], method='RK45',
                        rtol=1e-10, atol=1e-12,
dense_output=True)
    t_ref = np.linspace(t_span[0], t_span[1], 100)
    y_ref = sol_ref.sol(t_ref)[0]

    y_ref_euler = sol_ref.sol(t_euler)[0]
    y_ref_trap = sol_ref.sol(t_trap)[0]
    y_ref_rk4 = sol_ref.sol(t_rk4)[0]

    err_euler = np.linalg.norm(y_euler - y_ref_euler)
    err_trap = np.linalg.norm(y_trap - y_ref_trap)
```

```
    err_rk4 = np.linalg.norm(y_rk4 - y_ref_rk4)

    print(f"Ошибка метода Эйлера: {err_euler:.2e}")
    print(f"Ошибка явного метода трапеций: {err_trap:.2e}")
    print(f"Ошибка метода РК4: {err_rk4:.2e}")

    plt.figure(figsize=(12, 8))
    plt.plot(t_ref, y_ref, 'k-', linewidth=5, label='Эталонное
решение (solve_ivp)')
    plt.plot(t_euler, y_euler, 'bo-', markersize=10,
label='Метод Эйлера')
    plt.plot(t_trap, y_trap, 'c*-',  markersize=15,
label='Явный метод трапеций')
    plt.plot(t_rk4, y_rk4, 'rs-', markersize=5, label='Метод
РК4')

    plt.xlabel('t')
    plt.ylabel('y(t)')
    plt.title('Задача 1.4: y\' = y² - t, y(0) = 1')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

    return t_euler, y_euler, t_trap, y_trap, t_rk4, y_rk4,
t_ref, y_ref


if __name__ == "__main__":
    result1 = problem_1_4()

def problem_2_4():
    print("\n=== ЗАДАЧА 2.4: y' = -2000y + t, y(0) = 0 ===")

    def f(t, y):
        return -2000 * y + t

    t_span = [0, 1]
    y0 = 0
    h = 1

    solver = ODESolver()
```

```
    t_euler, y_euler = solver.euler_method(f, t_span, y0, h)
    t_beuler, y_beuler = solver.backward_euler(f, t_span, y0,
h)
    t_itrap, y_itrap = solver.implicit_trapezoid(f, t_span,
y0, h)

    sol_ref = solve_ivp(f, t_span, [y0], method='Radau',
                        rtol=1e-10, atol=1e-12,
dense_output=True)
    t_ref = np.linspace(t_span[0], t_span[1], 100)
    y_ref = sol_ref.sol(t_ref)[0]

    y_ref_euler = sol_ref.sol(t_euler)[0]
    y_ref_beuler = sol_ref.sol(t_beuler)[0]
    y_ref_itrap = sol_ref.sol(t_itrap)[0]

    err_euler = np.linalg.norm(y_euler - y_ref_euler)
    err_beuler = np.linalg.norm(y_beuler - y_ref_beuler)
    err_itrap = np.linalg.norm(y_itrap - y_ref_itrap)

    print(f"Ошибка явного метода Эйлера: {err_euler:.2e}")
    print(f"Ошибка неявного метода Эйлера: {err_beuler:.2e}")
    print(f"Ошибка неявного метода трапеций: {err_itrap:.2e}")

    plt.figure(figsize=(12, 8))
    plt.plot(t_ref, y_ref, 'k-', linewidth=5, label='Эталонное
решение (solve_ivp)')
    plt.plot(t_euler, y_euler, 'bo-', markersize=4,
label='Явный метод Эйлера')
    plt.plot(t_beuler, y_beuler, 'g^-', markersize=4,
label='Неявный метод Эйлера')
    plt.plot(t_itrap, y_itrap, 'm*-',linewidth=2.2,
markersize=4, label='Неявный метод трапеций')

    plt.xlabel('t')
    plt.ylabel('y(t)')
    plt.title('Задача 2.4: y\' = -2000y + t, y(0) = 0 (жесткая
система)')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()
```

```
    return t_euler, y_euler, t_beuler, y_beuler, t_itrap,
y_itrap, t_ref, y_ref


if __name__ == "__main__":
    result2 = problem_2_4()
```