ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

МОСКОВСКИЙ ИНСТИТУТ ЭЛЕКТРОНИКИ И МАТЕМАТИКИ

им. А. Н. ТИХОНОВА

Щербинин Дмитрий Игоревич
БИВ172

ЛАБОРАТОРНАЯ РАБОТА №1

По курсу «Математический компьютерный практикум»
по направлению 09.03.01 Информатика и вычислительная техника
студента образовательной программы бакалавриата
«Информатика и вычислительная техника»

Руководитель:
Круглик Станислав Александрович

Москва 2019 г.

Оглавление

1. Исходные данные варианта

Найдите решение дифференцального уравнения, удовлетворяющее произвольно заданным начальным условиям на промежутке [0, 10], и постройте график полученной функции.

$$y'' - 6y' + 9y = x^2 - x + 3$$

2. Решение задачи

2.1. Аналитическое решение

Используя WolphramAlpha, получаем:

Input:

$$y''(x) - 6\,y'(x) + 9\,y(x) = x^2 - x + 3$$

ODE classification:

second-order linear ordinary differential equation

Alternate forms:

$$x^2 + 6\,y'(x) + 3 = y''(x) + 9\,y(x) + x$$

$$y''(x) = x^2 + 6\,y'(x) - 9\,y(x) - x + 3$$

Differential equation solution:

$$y(x) = c_2\,e^{3x}\,x + c_1\,e^{3x} + \frac{x^2}{9} + \frac{x}{27} + \frac{1}{3}$$

Таким образом аналитическое решение уравнения:

$$y(x) = C_1 e^{3x} + C_2 x e^{3x} + \frac{x^2}{9} + \frac{x}{27} + \frac{1}{3} (1)$$

Допустим нам заданы следующие начальные условия:

$$\begin{cases} y(x_0) = y_0 \\ y'(x_0) = y_0' \end{cases} (*)$$

Найдем производную от аналитического решения:

$$y'(x) = 3C_1 e^{3x} + C_2(3x + 1)e^{3x} + \frac{2x}{9} + \frac{1}{27} (2)$$

Перепишем систему (*) с учетом уравнений (1) и (2):

$$\begin{cases} a_{11}C_1 + a_{12}C_2 = b_1 \\ a_{21}C_1 + a_{22}C_2 = b_2 \end{cases}, \text{где}$$

$$a_{11} = e^{3x_0}, a_{12} = x_0 e^{3x_0}, b_1 = y_0 - \frac{x_0^2}{9} - \frac{x_0}{27} - \frac{1}{3}$$

$$a_{21} = 3e^{3x_0}, a_{22} = (3x_0 + 1)e^{3x_0}, b_2 = y_0' - \frac{2x_0}{9} - \frac{1}{27}$$

Решая полученную систему уравнений находим коэффициенты C_1 и C_2, а следовательно получаем из нашего семейства функций искомую функцию.

2.2. Численное решение

Для получения численного решения был применен метод Хьюна, в котором каждое следующее значение функции выражается через предыдущие уже известные значения. Сначала вычисляют первое приближение $\tilde{y}_{j+1}$ по формуле Эйлера:

$$\tilde{y}_{j+1} = y_j + hf(x_j, y_j),$$

а затем находят значение y_{j+1}:

$$y_{j+1} = y_j + \frac{h}{2}\left[f(x_j, y_j) + f(x_{j+1}, \tilde{y}_{j+1})\right].$$

2.3. Листинг решения

Численное решение, а также построение всех необходмых графиков было релазиовано с помощью Python и сторонних библиотек NumPy и Matplotlib.

Листинг фнукций, реализующих аналитическое решение и метод Хьюна (файл Lab.py):

```
import math
import numpy as np


def solve_analytically(x0: float, y0: float, dy0: float):
    """
    Принимает на вход условия для задачи Коши и возврщает функцию, из семейства функций,
    удовлетворяющую условиям задачи Коши
    Уравнение: y'' - 6*y' + 9*y = x^2 - x + 3
    Общее решение: y = C1*e^3x + C2*xe^3x + 1/9*x^2 + 1/27*x + 1/3

    :param x0: точка, для которой известны начальные условия задачи Коши
    :param y0: функция в точке x0
    :param dy0: производная функции в точке x0
    :return: функция, удовлетворяющая заданным условиям
```

```
    """
    A1 = [math.exp(3 * x0), x0 * math.exp(3 * x0)]
    B1 = y0 - (1 / 9 * x0 ** 2 + 1 / 27 * x0 + 1 / 3)
    A2 = [3 * math.exp(3 * x0), (3 * x0 + 1) * math.exp(3 * x0)]
    B2 = dy0 - (2 / 9 * x0 + 1 / 27)
    A = np.array([A1,
                  A2])
    B = np.array([[B1],
                  [B2]])
    X = np.linalg.solve(A, B)
    [C1], [C2] = X.tolist()

    return lambda x: C1 * math.exp(3 * x) + C2 * x * math.exp(3 * x) + 1 / 9 * x ** 2 + 1 / 27
* x + 1 / 3



def prepare_data_for_plotting_func(func, borders: tuple, step=0.01):
    """
    Подготавливает данные для построения графика функции

    :param func: сама фукнция
    :param borders: границы
    :param step: шаг дискретизации
    :return: (x, y)
    """
    start, finish = borders
    x = np.arange(start, finish + step, step)
    y = np.array([func(el) for el in x])
    return x, y



def solve_euler_numerically(x0: float, y0: float, dy0: float, borders: tuple, h=0.01):
    """
    Принимает на вход условия для задачи Коши и границы отрезка, на котором нужно решить ДУ,
    решает уравнение методом Эйлера с помощью построения сетки на необходимом отрезке
    и возврщает решение в узлах сетки.
    Уравнение: y'' - 6*y' + 9*y = x^2 - x + 3
```

```
    :param x0: точка, для которой известны начальные условия задачи Коши
    :param y0: функция в точке x0
    :param dy0: производная функции в точке x0
    :param borders: границы
    :param h: шаг сетки
    :return: двумерный массив, первая строка - X, вторая - Y
    """
    left, right = borders
    if x0 > left:
        raise ArithmeticError

    xj, dyj, yj = x0, dy0, y0

    if x0 < left:
        for xj in np.arange(x0, left + h, h):
            dyj_next = dyj + h * (6 * dyj - 9 * yj + xj ** 2 - xj + 3)
            yj_next = yj + h * dyj
            dyj = dyj_next
            yj = yj_next

    # steps_num = int(math.floor((right - left)) / h + 1)
    steps_num = np.arange(left, right + h, h).size
    shape = (3, steps_num)
    grid = np.empty(shape, np.float)
    grid[:, 0] = [xj,
                  dyj,
                  yj]
    grid[0] = np.arange(left, right + h, h)
    for j in np.arange(steps_num - 1):
        xj, dyj, yj = grid[:, j]
        grid[1, j + 1] = dyj + h * (6 * dyj - 9 * yj + xj ** 2 - xj + 3)
        grid[2, j + 1] = yj + h * dyj

    return grid[(0, 2), :]


def solve_hyung_numerically(x0: float, y0: float, dy0: float, borders: tuple, h=0.01):
    """
```

```
Принимает на вход условия для задачи Коши и границы отрезка, на котором нужно решить ДУ,
решает уравнение методом Хьюна с помощью построения сетки на необходимом отрезке
и возвращает решение в узлах сетки.
Уравнение: y'' - 6*y' + 9*y = x^2 - x + 3

:param x0: точка, для которой известны начальные условия задачи Коши
:param y0: функция в точке x0
:param dy0: производная функции в точке x0
:param borders: границы
:param h: шаг сетки
:return: двумерный массив, первая строка - X, вторая - Y
"""
left, right = borders
if x0 > left:
    raise ArithmeticError

xj, dyj, yj = x0, dy0, y0

if x0 < left:
    for xj in np.arange(x0, left + h, h):
        xj_temp = xj + h
        dyj_temp = dyj + h * (6 * dyj - 9 * yj + xj ** 2 - xj + 3)
        yj_temp = yj + h * dyj
        dyj_next = dyj + h/2*(6 * dyj - 9 * yj + xj ** 2 - xj + 3 +
                              6 * dyj_temp - 9 * yj_temp + xj_temp ** 2 - xj_temp + 3)
        yj_next = yj + h/2*(dyj + dyj_temp)
        dyj = dyj_next
        yj = yj_next

# steps_num = int(math.floor((right - left)) / h + 1)
steps_num = np.arange(left, right + h, h).size
shape = (3, steps_num)
grid = np.empty(shape, np.float)
grid[:, 0] = [xj,
              dyj,
              yj]
grid[0] = np.arange(left, right + h, h)
for j in np.arange(steps_num - 1):
```

```
        xj, dyj, yj = grid[:, j]

        xj_temp = xj + h
        dyj_temp = dyj + h * (6 * dyj - 9 * yj + xj ** 2 - xj + 3)
        yj_temp = yj + h * dyj
        dyj_next = dyj + h / 2 * (6 * dyj - 9 * yj + xj ** 2 - xj + 3 +
                                  6 * dyj_temp - 9 * yj_temp + xj_temp ** 2 - xj_temp + 3)
        yj_next = yj + h / 2 * (dyj + dyj_temp)

        grid[1, j + 1] = dyj_next
        grid[2, j + 1] = yj_next

    return grid[(0, 2), :]


def make_approximation_function(grid: np.ndarray):
    """
    Принимает на вход сетку, полученную в результате численного решения ДУ,
    и возвращает функцию аппроксимации искомой функции на промежутке, на котором задана сетка.

    :param grid: сетка, полученная в результате численного решения ДУ
    :return: функция аппроксимации
    """
    def linear_approximation(x: float):
        """
        Линейная аппроксимация.

        :param x: произвольная точка из области определения функции
        :return: значение функции в точке x
        """
        h = grid[0, 1] - grid[0, 0]
        x0 = grid[0, 0]
        j = int(math.floor((x - x0) / h))
        x1, y1, x2, y2 = grid[0, j], grid[1, j], grid[0, j + 1], grid[1, j + 1]
        return (y2 - y1) / (x2 - x1) * (x - x1) + y1
    return linear_approximation
```

```
def error_comparison(start_h, stop_h, num_h, x0: float, y0: float, dy0: float, borders: tuple, solve_method):
    """
    Рассчитывает ошибки для построения графика зависимости ошибки от шага сетки.
    За ошибку берется максимальное значение невязки (в процентах) на необходимом отрезке

    :param start_h: минимальное значение шага сетки
    :param stop_h: максимальное значение шага сетки
    :param num_h: количество отсчетов между start_h и num_h
    :param x0: точка, для которой известны начальные условия задачи Коши
    :param y0: функция в точке x0
    :param dy0: производная функции в точке x0
    :param borders: границы
    :param solve_method: метод численного решения ДУ
    :return: двумерный массив, первая строка - шаг сетки, вторая - ошибка
    """
    H = np.linspace(start_h, stop_h, num_h)
    error = np.empty(num_h)
    func = solve_analytically(x0, y0, dy0)

    for i, h in enumerate(H):
        ya = prepare_data_for_plotting_func(func, borders, h)[1]
        yc = solve_method(x0, y0, dy0, borders, h)[1]
        error[i] = np.max(np.abs((yc - ya) / ya) * 100)

    return np.array((H, error))
```

Листинг скрипта, запускащего расчет аналитического и численного решений, а также построение необходимых графиков (файл LabScript.py):

```
import Lab
import matplotlib.pyplot as plt
import numpy as np

x0 = -10
y0 = 4
dy0 = 10
X = (0, 10)
h = 10 ** -3
```

```
func = Lab.solve_analytically(x0, y0, dy0)
xa, ya = Lab.prepare_data_for_plotting_func(func, X, h)
xc, yc = Lab.solve_hyung_numerically(x0, y0, dy0, X, h)
xer, yer = xa, np.abs((yc - ya) / ya) * 100

plt.plot(xa, ya, color='green', label=u'Аналитическое решение')
plt.plot(xc, yc, color='blue', label=u'Численное решение')
plt.title(u'Метод Хьюна')
plt.xlabel(u'X')
plt.ylabel(u'Y')
plt.legend()
plt.grid(True)
plt.show()

plt.plot(xer, yer, color='red')
plt.title(u'Невязка')
plt.xlabel(u'X')
plt.ylabel(u'Невязка, %')
plt.grid(True)
plt.show()

func = Lab.make_approximation_function(np.array((xc, yc)))
xap = np.arange(X[0], X[1], 10 ** -5)
yap = np.array([func(x) for x in xap])
plt.plot(xap, yap, color='orange')
plt.title(u'Аппроксимация')
plt.xlabel(u'X')
plt.ylabel(u'Y')
plt.grid(True)
plt.show()

hc, erc = Lab.error_comparison(10 ** -4, 1, 200, x0, y0, dy0, X, Lab.solve_hyung_numerically)
plt.plot(hc, erc, color='purple', marker='.')
plt.title(u'Зависимость ошибки от шага сетки')
plt.xlabel(u'Шаг сетки')
plt.ylabel(u'Ошибка, %')
plt.grid(True)
plt.show()
```

3. Полученные результаты

На выходе было получено 3 графика:

- Аналитическое и численное решения.
- Ошибка (в процентах).
- Зависимость ошибки (в процентах) от шага сетки (в качестве ошибки для каждого шага сетки берется максимальная ошибка при данном шаге сетки).

Начальные условия: $x_0 = -10;\ y_0 = 4;\ y_0' = 10.$

Шаг сетки: $\delta = 0.001$

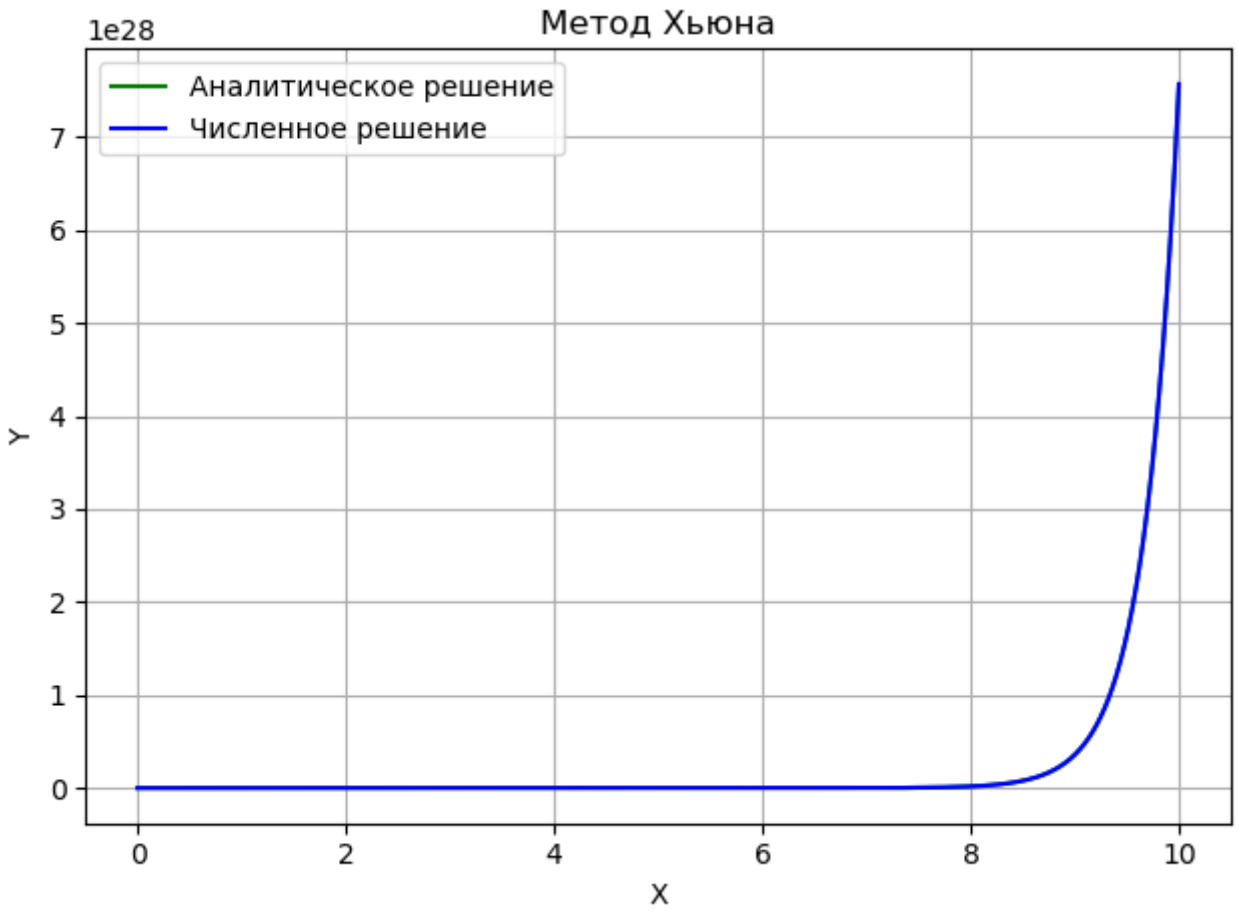

Рис. 1. Аналитическое и численное решения

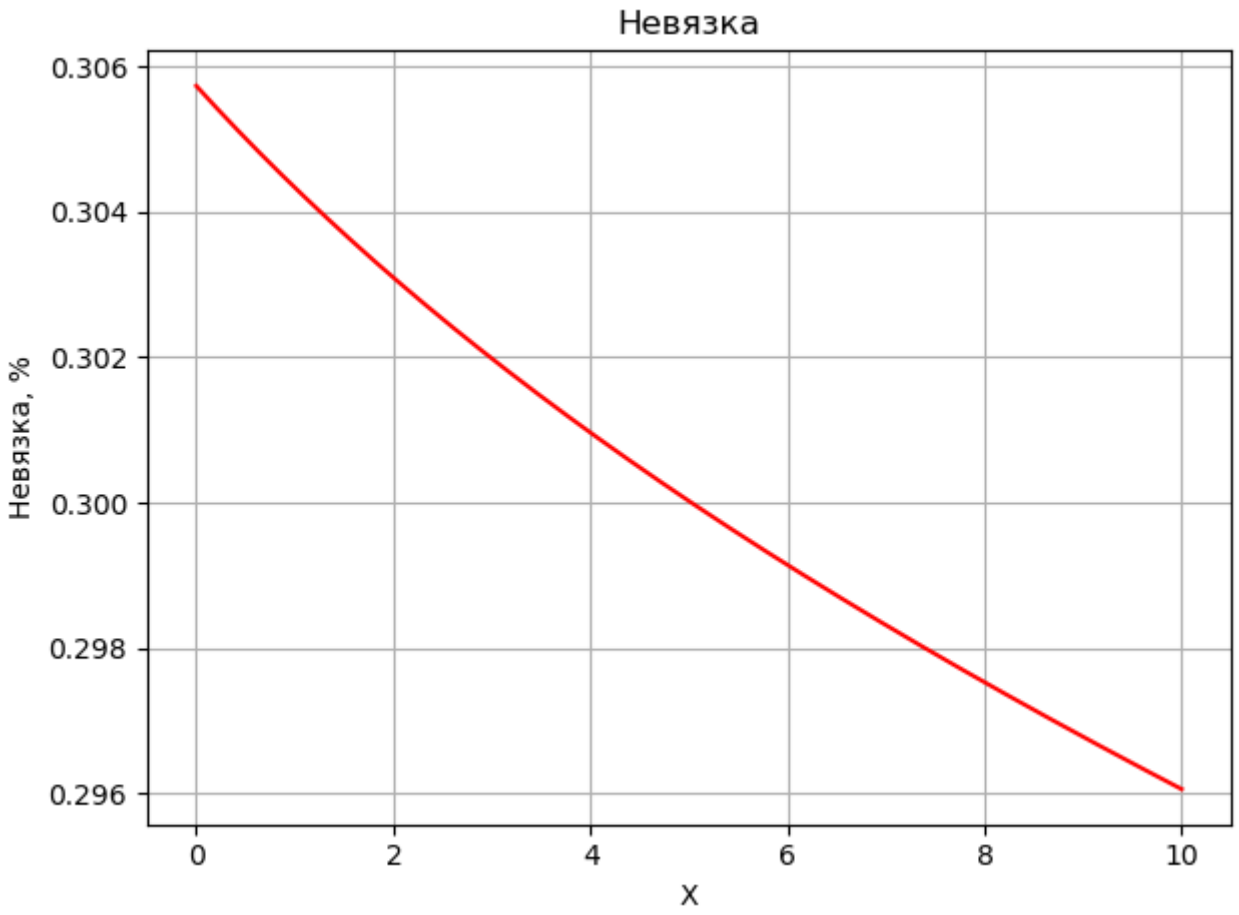

Рис. 2. Ошибка (в процентах)

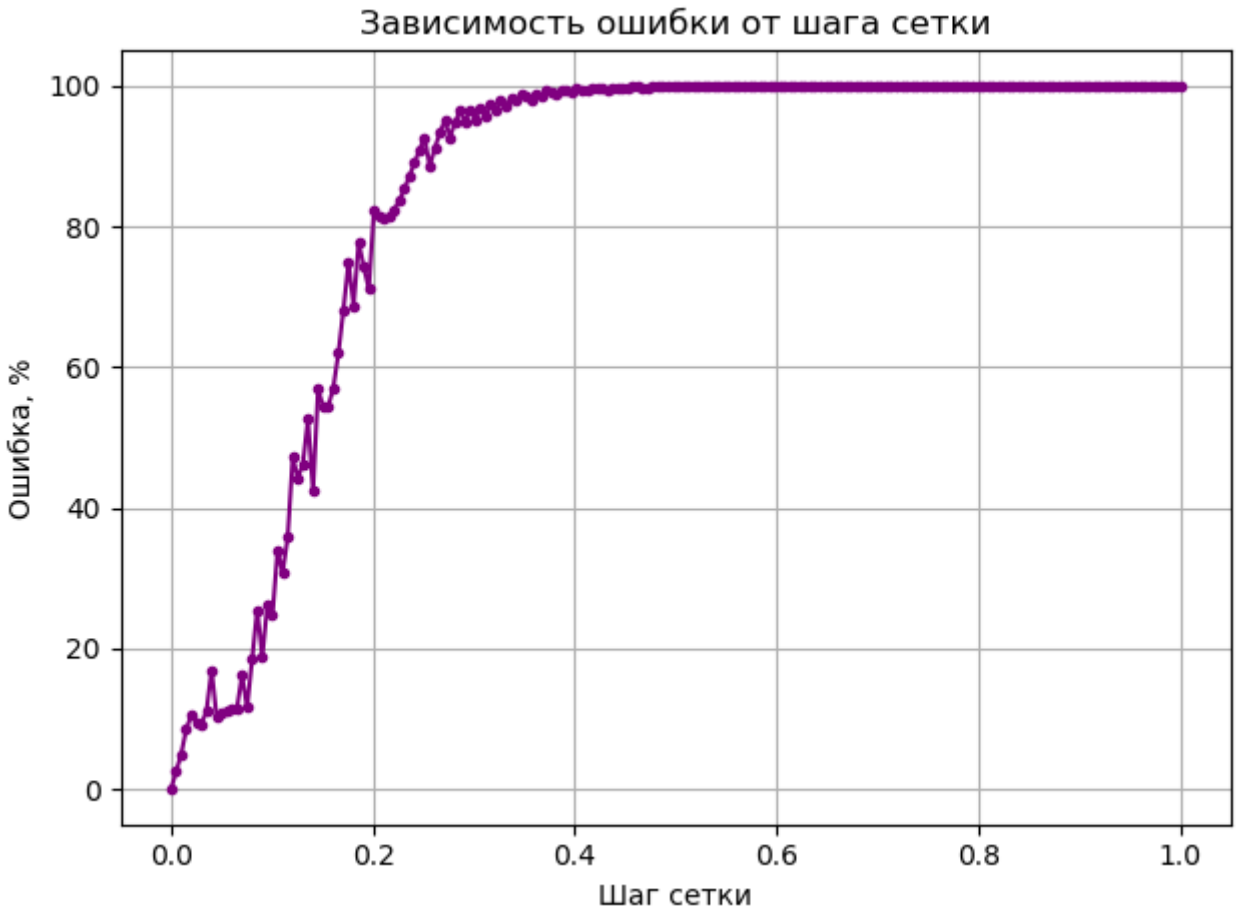

Рис. 3. Зависимость ошибки (в процентах) от шага сетки