Московский авиационный институт

(Национальный исследовательский университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Курсовая работа по курсу

«Численные методы»

На тему: «Решение краевых задач для нелинейных дифференциальных уравнений методом конечных разностей»

Вариант 13

Выполнил: Искренкова А.В.

Группа: М8О-409Б-20

Проверил: Пивоваров Д.Е.

Дата:

Оценка:

Москва, 2023

Задача

Реализовать программу для решения краевых задач для нелинейных дифференциальных уравнений второго порядка методом конечных разностей, То есть найти решение ДУ виду на отрезке [a,b] с краевыми условиями.

$$\mathrm{F}(x, y, y', y'') = 0$$

$$a_1 y(a) + b_1 y'(a) = c_1$$
$$a_1 y(b) + b_1 y'(b) = c_2$$
$$|a_1| + |b_1| > 0 и |a_2| + |b_2| > 0.$$

Теоретические сведения

1. Основные идеи МКР:

Дискретизация пространства: Метод конечных разностей основан на идее разбиения рассматриваемой области на конечное число узлов или точек. Это позволяет аппроксимировать дифференциальные уравнения разностными аналогами.
Локальная аппроксимация: Дифференциальные операторы и производные заменяются разностными квадратурными формулами, что позволяет представить дифференциальные уравнения в алгебраической форме.

2. Дискретизация области:

Сетка: Рассматриваемая область разбивается на сетку узлов. Часто используются равномерные сетки для упрощения численных вычислений.
Шаги: Вводятся пространственные и временные шаги дискретизации для определения расстояний между узлами.

3. Аппроксимация уравнений:

Разностные схемы: Дифференциальные операторы и члены уравнения аппроксимируются с использованием разностных схем. Линейные и нелинейные члены уравнения выражаются в виде конечных разностей.
Производные: Производные заменяются разностными приближениями. Например, первая производная может быть аппроксимирована как отношение изменения функции к изменению координаты.

4. Система алгебраических уравнений:

Дискретизация уравнений: Дифференциальные уравнения преобразуются в систему алгебраических уравнений, представляющую собой систему линейных или нелинейных уравнений, в зависимости от характера задачи.
Матричная форма: Разностные аппроксимации и значения на сетке приводят к матричной форме системы уравнений.

5. Решение системы:

Итерационные методы: В случае нелинейных уравнений часто применяются итерационные методы, такие как метод Ньютона, для нахождения численного решения системы.
Метод прогонки: В случае линейных уравнений может быть использован метод прогонки для эффективного решения системы.

6. Проверка сходимости и стабильности:

Устойчивость: Усилия направлены на обеспечение устойчивости численного метода, чтобы избежать численных осцилляций и нефизичных результатов.
Сходимость: Тщательное изучение сходимости численного решения к реальному физическому решению.

7. Применение для нелинейных краевых задач:

Адаптация метода: Нелинейные краевые задачи требуют особого внимания к нелинейным членам и выбору метода для их решения.
Учет нелинейности: Применение итерационных методов для решения нелинейных систем уравнений и контроль за сходимостью.

8. Примеры применения:

Физика и инженерия: МКР широко используется в решении нелинейных краевых задач, возникающих в задачах теплопроводности, механики материалов, электродинамики и других областях.
Биология и медицина: Применение МКР для моделирования распространения волн в нейронных сетях, диффузии в биологических тканях и других биомедицинских задач.

Реализация методов на языке Python

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


def finite_difference_method(cond1, cond2, equation, borders, h=0.01, accuracy=2):
    x_grid = np.arange(borders[0], borders[1] + h, h)
    N = np.shape(x_grid)[0]

    A_matrix = np.zeros((N, N))
    b_vector = np.zeros(N)

    for i in range(1, N - 1):
        A_matrix[i][i - 1] = 1 / h ** 2 - equation['p'](x_grid[i]) / (2 * h)
        A_matrix[i][i] = -2 / h ** 2 + equation['q'](x_grid[i])
        A_matrix[i][i + 1] = 1 / h ** 2 + equation['p'](x_grid[i]) / (2 * h)
        b_vector[i] = equation['f'](x_grid[i])

    if accuracy == 1:
        A_matrix[0][0] = cond1['a'] - cond1['b'] / h
        A_matrix[0][1] = cond1['b'] / h
        b_vector[0] = cond1['c']

        A_matrix[N - 1][N - 2] = -cond2['b'] / h
```

```
        A_matrix[N - 1][N - 1] = cond2['a'] + cond2['b'] / h
        b_vector[N - 1] = cond2['c']

    elif accuracy == 2:
        p = equation['p']
        q = equation['q']
        a1 = cond1['a'];
        b1 = cond1['b'];
        c1 = cond1['c']
        a2 = cond2['a'];
        b2 = cond2['b'];
        c2 = cond2['c']

        A_matrix[0][0] = a1 - (3 * b1) / (2 * h) + ((2 - h * p(x_grid[1])) * b1) / ((2 
+ h * p(x_grid[1])) * 2 * h)
        A_matrix[0][1] = 2 * b1 / h + ((h * h * q(x_grid[1]) - 2) * b1) / ((2 + h * 
p(x_grid[1])) * h)
        A_matrix[0][2] = 0
        b_vector[0] = c1

        A_matrix[N - 1][N - 3] = 0
        A_matrix[N - 1][N - 2] = -2 * b2 / h - ((h * h * q(x_grid[N - 2]) - 2) * b2) / 
((2 - h * p(x_grid[N - 2])) * h)
        A_matrix[N - 1][N - 1] = a2 + 3 * b2 / (2 * h) - ((2 + h * p(x_grid[N - 2])) * 
b2) / (
                    (2 - h * p(x_grid[N - 2])) * 2 * h)
        b_vector[N - 1] = c2

    return solve_tridiagonal_system(A_matrix, b_vector)


def solve_tridiagonal_system(A, b):
    p = np.zeros(len(b))
    q = np.zeros(len(b))
    p[0] = -A[0][1] / A[0][0]
    q[0] = b[0] / A[0][0]

    for i in range(1, len(p) - 1):
        p[i] = -A[i][i + 1] / (A[i][i] + A[i][i - 1] * p[i - 1])
        q[i] = (b[i] - A[i][i - 1] * q[i - 1]) / (A[i][i] + A[i][i - 1] * p[i - 1])

    p[-1] = 0
    q[-1] = (b[-1] - A[-1][-2] * q[-2]) / (A[-1][-1] + A[-1][-2] * p[-2])
    x = np.zeros(len(b))
    x[-1] = q[-1]

    for i in reversed(range(len(b) - 1)):
        x[i] = p[i] * x[i + 1] + q[i]

    return x


def calculate_mean_square_error(y, y_correct):
    return np.sqrt(np.sum((y - y_correct) ** 2))


def true_function(x):
    return (1 + x) * np.exp(-x * x)


equation_parameters = {
    'p': lambda x: 4 * x,
    'q': lambda x: 4 * x ** 2 + 2,
    'f': lambda x: 0,
}

boundary_condition_1 = {
    'a': 0,
```

```
    'b': 1,
    'c': 1,
}

boundary_condition_2 = {
    'a': 4,
    'b': -1,
    'c': 23 * (np.e ** (-4)),
}

interval_bounds = (0, 2)
grid_steps = (0.1, 0.01, 0.001)

if __name__ == '__main__':
    x_values = [np.arange(interval_bounds[0], interval_bounds[1], step) for step in
grid_steps]
    y_correct_values = [true_function(xi) for xi in x_values]
    y_values = [
        finite_difference_method(boundary_condition_1, boundary_condition_2,
equation_parameters, interval_bounds,
                                 step)[:-1] for step in grid_steps]

    for i in range(len(grid_steps)):
        print(
            f'Mean square error with h = {grid_steps[i]}:
{calculate_mean_square_error(y_values[i], y_correct_values[i])}')

    plt.figure(figsize=(12, 7))
    plt.plot(x_values[2], y_correct_values[2], label='Correct')
    for i in range(len(x_values)):
        plt.plot(x_values[i], y_values[i], label='h = ' + str(grid_steps[i]))
    plt.legend()
    plt.show()
```

В данном коде решается линейное дифференциальное уравнение второго порядка методом конечных разностей с применением граничных условий Дирихле. Программа также оценивает точность численного метода для различных значений шага сетки.

Код приведён для решения следующей краевой задачи:

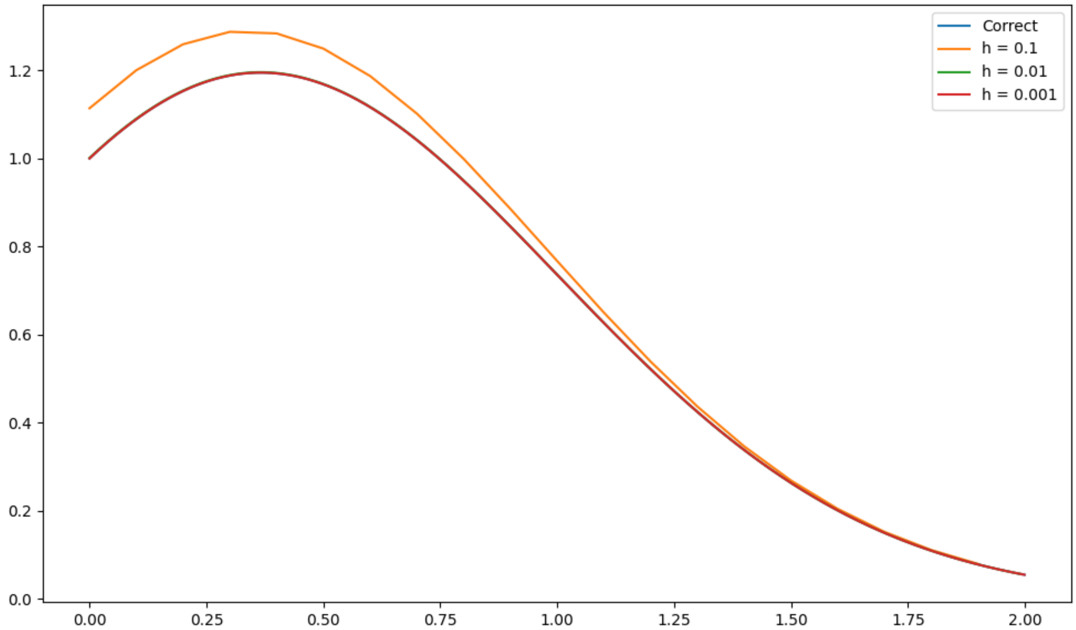
$$y'' + 4xy' + (4x\textasciicircum 2 + 2)y = 0$$
$$y'(0) = 1$$
$$4y(2) - y'(2) = 23e^{-4}$$

Аналитическое решение:

$$y(x) = (1 + x)e^{-x^{-2}}$$

Результаты тестов:

Mean square error with h = 0.1: 0.2757068134188619
Mean square error with h = 0.01: 0.008656104650093885
Mean square error with h = 0.001: 0.0002731450665959879



Вывод

Метод конечных разностей является широко применяемым численным методом для решения дифференциальных уравнений. В отличие от методов квадратур, которые используют аппроксимацию интегралов, метод конечных разностей основан на аппроксимации производных. Он является эффективным инструментом для численного моделирования и анализа различных физических и инженерных задач.