Московский авиационный институт

(Национальный исследовательский университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Отчет по лабораторным работам

по курсу «Численные методы»

Вариант 2

Выполнил: Примаченко А.А.

Группа: М8О-408Б-20

Проверил: проф. Пивоваров Д.Е.

Дата:

Оценка:

Москва, 2023

ЛАБОРАТОРНАЯ РАБОТА №4. РЕШЕНИЕ ДВУМЕРНОЙ НАЧАЛЬНО-КРАЕВОЙ ЗАДАЧИ ДЛЯ ДИФФЕРЕНЦИАЛЬНОГО УРАВНЕНИЯ ПАРАБОЛИЧЕСКОГО ТИПА

Задание

Используя схемы переменных направлений и дробных шагов, решить двумерную начально-краевую задачу для дифференциального уравнения параболического типа. В различные моменты времени вычислить погрешность численного решения путем сравнения результатов с приведенным в задании аналитическим решением $U(x,t)$. Исследовать зависимость погрешности от сеточных параметров τ, h_x, h_y.

Вариант 2

$$\frac{\partial u}{\partial t} = a\frac{\partial^2 u}{\partial x^2} + a\frac{\partial^2 u}{\partial y^2}, a > 0,$$

$$u(0,y,t) = \cos(\mu_2 y) * \exp(-(\mu_1^2 + \mu_2^2)at),$$

$$u\left(\frac{\pi}{2}\mu_1, y, t\right) = 0,$$

$$u(x,0,t) = \cos(\mu_1 x) * \exp(-(\mu_1^2 + \mu_2^2)at),$$

$$u\left(x, \frac{\pi}{2}\mu_2, t\right) = 0,$$

$$u(x,y,0) = \cos(\mu_1 x) * \cos(\mu_2 y)$$

Аналитическое решение: $U(x,y,t) = \cos(\mu_1 x)\cos(\mu_2 y) * \exp(-(\mu_1^2 + \mu_2^2)at)$

$\mu_1 = 1, \mu_2 = 1$

Ход решения

Введем пространственно-временную сетку с шагами h_x, h_y, τ соответственно по переменным x, y, t.

$$\omega_{h_1 h_2}^{\tau} = \{x_i = ih_x, i = \overline{0,I}; x_j = jh_y, j = \overline{0,J}; t^k = k\tau, k = 0,1,2,\ldots\}$$

и на этой сетке будем аппроксимировать дифференциальную задачу методом конечных разностей.

Метод переменных направлений

В двумерном случае схема метода переменных направлений для поставленной задачи имеет вид

$$\frac{u_{ij}^{k+\frac{1}{2}} - u_{ij}^{k}}{\frac{\tau}{2}} = \frac{a}{h_x^2}\left(u_{i+1j}^{k+\frac{1}{2}} - 2u_{ij}^{k+\frac{1}{2}} + u_{i-1j}^{k+\frac{1}{2}}\right) + \frac{a}{h_y^2}\left(u_{ij+1}^{k} - 2u_{ij}^{k} + u_{ij-1}^{k}\right) + f_{ij}^{k+\frac{1}{2}},$$

$$\frac{u_{ij}^{k+1} - u_{ij}^{k+\frac{1}{2}}}{\frac{\tau}{2}} = \frac{a}{h_x^2}\left(u_{i+1j}^{k+\frac{1}{2}} - 2u_{ij}^{k+\frac{1}{2}} + u_{i-1j}^{k+\frac{1}{2}}\right) + \frac{a}{h_y^2}\left(u_{ij+1}^{k+1} - 2u_{ij}^{k+1} + u_{ij-1}^{k+1}\right) + f_{ij}^{k+\frac{1}{2}},$$

В первом соотношении на первом дробном шаге $\frac{\tau}{2}$ оператор $a\frac{\partial^2}{\partial x^2}$ аппроксимируется неявно, а оператор $a\frac{\partial^2}{\partial x^2}$ – явно (в результате весь конечно-разностный оператор по переменной y переходит в правые части, поскольку u_{ij}^{k} известно). помощью скалярных прогонок в количестве, равном числу $J-1$, в направлении переменной x получаем распределение сеточной функции $u_{ij}^{k+\frac{1}{2}}, i = \overline{1, I-1}, j = \overline{1, J-1}$ на первом временном полуслое $t^{k+\frac{1}{2}} = t^k + \frac{\tau}{2}$.

Во втором соотношении оператор $a\frac{\partial^2}{\partial x^2}$ аппроксимируется явно, а $a\frac{\partial^2}{\partial x^2}$ – неявно. В результате методом прогонки получаем значение функции $u_{ij}^{k}, i = \overline{1, I-1}, j = \overline{1, J-1}$.

Метод дробных шагов

Метод дробных шагов использует только неявные конечно-разностные операторы, что делает его абсолютно устойчивым в задачах, не содержащих смешанные производные.

Для поставленной задачи метод дробных шагов имеет вид

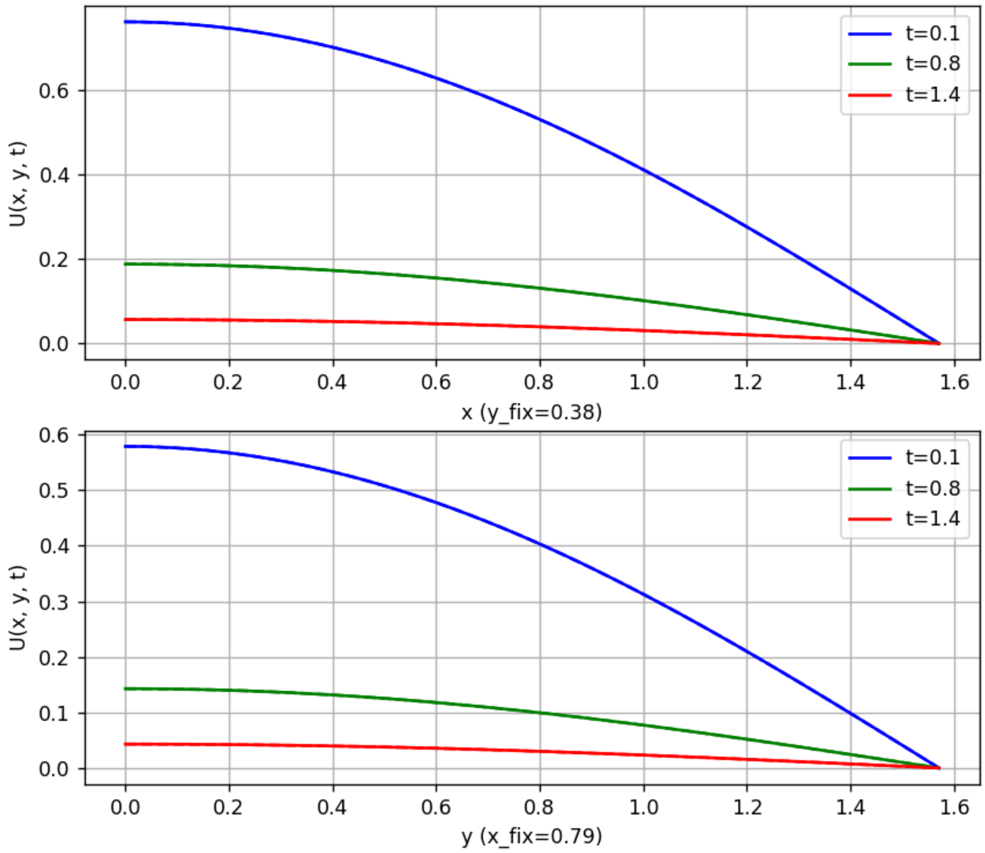
$$\frac{u_{ij}^{k+\frac{1}{2}} - u_{ij}^{k}}{\tau} = \frac{a}{h_x^2}\left(u_{i+1j}^{k+\frac{1}{2}} - 2u_{ij}^{k+\frac{1}{2}} + u_{i-1j}^{k+\frac{1}{2}}\right) + \frac{f_{ij}^{k}}{2},$$

$$\frac{u_{ij}^{k+1} - u_{ij}^{k+\frac{1}{2}}}{\tau} = \frac{a}{h_y^2}\left(u_{ij+1}^{k+1} - 2u_{ij}^{k+1} + u_{ij-1}^{k+1}\right) + \frac{f_{ij}^{k+1}}{2},$$

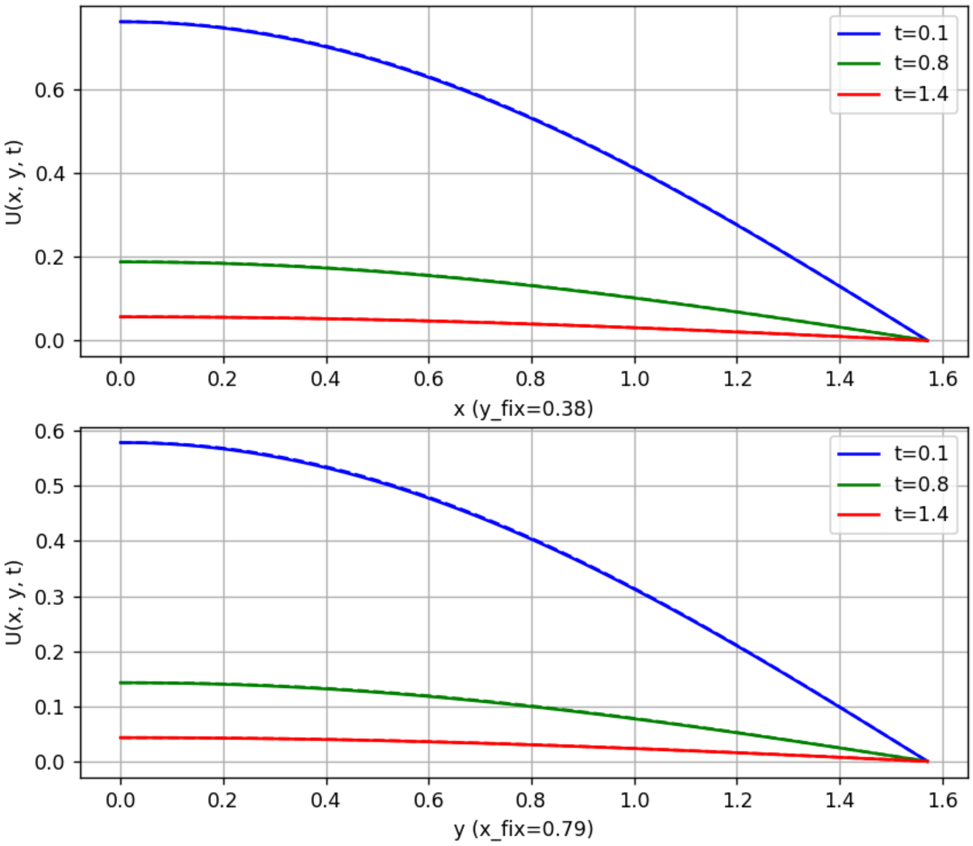
С помощью чисто неявной первой подсхемы осуществляются скалярные прогонки в направлении оси x в количестве, равном $J-1$, в результате чего получаем сеточную функцию $u_{ij}^{k+\frac{1}{2}}$. На втором дробном шаге по времени с помощью второй подсхемы осуществляются скалярные прогонки в направлении оси y в количестве, равном $I-1$, в результате чего получаем сеточную функцию u_{ij}^{k+1}.

Схема МДШ имеет порядок $O(\tau+|h|^2)$, т.е. первый порядок по времени и второй – по переменным x и y.

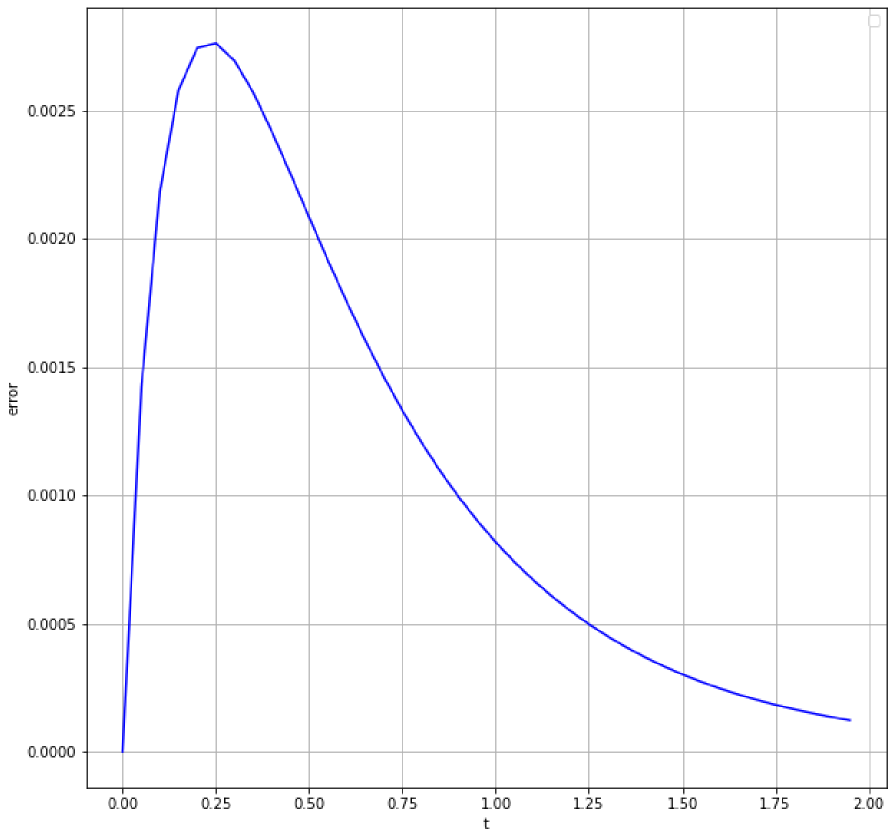
Результат работы программы



Метод переменных направлений при фиксированных $y = 0{,}38$ и $y = 0{,}79$



Метод дробных шагов при фиксированных $y = 0{,}38$ и $y = 0{,}79$



Зависимость погрешностей численных методов от выбора размера шага

Вывод

Из результатов выполненной работы можно заключить, что для поставленной задачи метод переменных направлений оказался точнее метода дробных шагов. Также было показано, что с увеличением шага дробления закономерно растет погрешность метода.

Код программы

```
import matplotlib.pyplot as plt
import numpy as np
from copy import deepcopy
plt.rcParams['figure.figsize'] = [8, 7]

mu1 = 1
mu2 = 1
a = 1

# Объявление начальных и краевых условий
def phi1(x, t):
    return np.cos(mu1*x)*np.exp(-(mu1*mu1+mu2*mu2)*a*t)

def phi2(x, t):
    return 0

def phi3(y, t):
    return np.cos(mu2*y)*np.exp(-(mu1*mu1+mu2*mu2)*a*t)

def phi4(y, t):
    return 0

def psi(x, y):
    return np.cos(mu1*x)*np.cos(mu2*y)

# Точное решение
def U(x, y, t):
    return np.cos(mu1*x)*np.cos(mu2*y)*np.exp(-(mu1*mu1+mu2*mu2)*a*t)

# Норма
def norm(v1, v2):
    return np.amax(np.abs(v1 - v2))

def Check(A):
```

```python
    if np.shape(A)[0] != np.shape(A)[1]:
        return False

    n = np.shape(A)[0]
    for i in range(n):
        sum = 0
        for j in range(n):
            if i != j:
                sum += abs(A[i][j])
        if abs(A[i][i]) < sum:
            return False
    return True

# Метод прогонки
def solve(a, b):
    if (Check(a)):
        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])
        i = len(a) - 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

# Функция для вычисления ошибок
def error(Nt, l, tau, U):
    N_array = [10, 20, 40]
    size = np.size(N_array)
    h_array = np.zeros(size)

    errors1x = np.zeros(size)
    errors2x = np.zeros(size)
    errors1y = np.zeros(size)
    errors2y = np.zeros(size)
```

```
    for i in range(0, size):
        h_array[i] = l/N_array[i]
        x_array = np.arange(0, l + h_array[i], h_array[i])
        y_array = np.arange(0, l + h_array[i], h_array[i])

        u1 = VariableDirectionMethod(Nt, N_array[i], N_array[i], tau, h_ar-
ray[i], h_array[i])
        u2 = FractionalStepsMethod(Nt, N_array[i], N_array[i], tau, h_array[i],
h_array[i])

        t = tau * Nt/2
        x = h_array[i] * N_array[i]/2
        y = h_array[i] * N_array[i]/2

        if (np.size(x_array)!=N_array[i]+1):
            x_array = x_array[N_array[i]+1]
            y_array = y_array[N_array[i]+1]

        ux_correct = np.array([U(x_i*h_array[i], y, t) for x_i in range(N_ar-
ray[i]+1)])
        uy_correct = np.array([U(x, y_i*h_array[i], t) for y_i in range(N_ar-
ray[i]+1)])
        u1x_calculated = u1[int(Nt / 2)][:][int(N_array[i]/2)]
        u2x_calculated = u2[int(Nt / 2)][:][int(N_array[i]/2)]
        u1y_calculated = u1[int(Nt / 2)][int(N_array[i]/2)][:]
        u2y_calculated = u2[int(Nt / 2)][int(N_array[i]/2)][:]

        errors1x[i] = np.amax(np.abs(ux_correct - u1x_calculated))
        errors2x[i] = np.amax(np.abs(ux_correct - u2x_calculated))
        errors1y[i] = np.amax(np.abs(uy_correct - u1y_calculated))
        errors2y[i] = np.amax(np.abs(uy_correct - u2y_calculated))

    return N_array, errors1x, errors2x, errors1y, errors2y

# Функция для построения графиков ошибок
def show_errors(Nt, l, tau, U):
    N_array, errors1x, errors2x, errors1y, errors2y = error(Nt, l, tau, U)

    colors = ['blue', 'red']

    delta = np.zeros(np.size(N_array))
    for i in range(np.size(N_array)):
```

```
        delta[i] = l / N_array[i]

    delta2 = np.zeros(np.size(N_array))
    for i in range(np.size(N_array)):
        delta2[i] = l / N_array[np.size(N_array) - i - 1]

    fig, ax = plt.subplots()
    plt.plot(delta,   errors1x,   color=colors[0],   label='Метод   переменных
направлений')
    plt.plot(delta2, errors2x, color=colors[1], label='Метод дробных шагов')
    ax.set_xlabel('delta X')
    ax.set_ylabel('Epsilon')
    plt.grid()
    ax.legend()
    plt.show()

    fig, ax = plt.subplots()
    plt.plot(delta,   errors1y,   color=colors[0],   label='Метод   переменных
направлений')
    plt.plot(delta2, errors2y, color=colors[1], label='Метод дробных шагов')
    ax.set_xlabel('delta Y')
    ax.set_ylabel('Epsilon')
    plt.grid()
    ax.legend()
    plt.show()

# Функция для построения графиков решения
def show_solution(Nx, Ny, Nt, hx, hy, tau, U, u):
    x_array = np.array([i * hx for i in range(Nx + 1)])
    y_array = np.array([j * hy for j in range(Ny + 1)])

    fig, ax = plt.subplots(2)

    t = [int(Nt * 0.05), int(Nt * 0.4), int(Nt * 0.7)]
    x_fix = int(Nx / 2)
    y_fix = int(Ny / 4)
    colors = ['blue', 'green', 'red']

    for i in range(len(t)):
        u_correct = np.zeros(Nx + 1)
        for x in range(Nx + 1):
            u_correct[x] = U(x * hx, y_fix * hy, t[i] * tau)
        u_calculated = u[t[i]][:][y_fix]
```

```
        ax[0].plot(y_array, u_correct, color=colors[i], label='t=%s' % round(t[i] * tau, 2))
        ax[0].plot(y_array, u_calculated, color=colors[i], linestyle='--')

    for i in range(len(t)):
        u_correct = np.zeros(Ny + 1)
        for y in range(Ny + 1):
            u_correct[y] = U(x_fix * hx, y * hy, t[i] * tau)
        u_calculated = u[t[i]][x_fix][:]

        ax[1].plot(y_array, u_correct, color=colors[i], label='t=%s' % round(t[i] * tau, 2))
        ax[1].plot(y_array, u_calculated, color=colors[i], linestyle='--')

    label1 = 'x (y_fix=%s)' % round(y_fix * hy, 2)
    label2 = 'y (x_fix=%s)' % round(x_fix * hx, 2)
    ax[0].set_xlabel(label1)
    ax[0].set_ylabel('U(x, y, t)')
    ax[0].grid()
    ax[0].legend()
    ax[1].set_xlabel(label2)
    ax[1].set_ylabel('U(x, y, t)')
    ax[1].grid()
    ax[1].legend()
    plt.show()

# Метод переменных направлений
def VariableDirectionMethod(Nt, Nx, Ny, tau, hx, hy):
    u = np.zeros((Nt + 1, Nx + 1, Ny + 1))

    # Заполняем краевые условия 1-го рода
    for t in range(Nt + 1):
        for x in range(Nx + 1):
            u[t][x][0] = phi1(x * hx, t * tau)
            u[t][x][Ny] = phi2(x * hx, t * tau)

    for t in range(Nt + 1):
        for y in range(Ny + 1):
            u[t][0][y] = phi3(y * hy, t * tau)
            u[t][Nx][y] = phi4(y * hy, t * tau)

    for x in range(Nx + 1):
```

```
        for y in range(Ny + 1):
            u[0][x][y] = psi(y * hy, x * hx)

    # Выполнение схемы метода переменных направлений
    for t in range(Nt):
        # Первый дробный шаг
        tmp = deepcopy(u[t])  # Временная переменная для хранения промежуточного
состояния на шаге tau+1/2
        for y in range(1, Ny):
            # Заполнение матрицы для метода прогонки на 1-ом дробном шаге
            matrix = np.zeros((Nx - 1, Nx - 1))
            d = np.zeros(Nx - 1)
            a_i = a * tau / (2 * hx * hx)
            b_i = -(a * tau / (hx * hx) + 1)
            c_i = a * tau / (2 * hx * hx)

            # Первая строка
            matrix[0][0] = b_i
            matrix[0][1] = c_i
            d[0] = -(u[t][1][y] + (a * tau / (2 * hy * hy)) * (
                        u[t][1][y - 1] - 2 * u[t][1][y] + u[t][1][y + 1]) + a
* tau / (2 * hx * hx) * phi3(
                y * hy, (t + 1 / 2) * tau))

            # Строки с первой по N-2
            for x in range(1, Nx - 2):
                matrix[x][x - 1] = a_i
                matrix[x][x] = b_i
                matrix[x][x + 1] = c_i
                d[x] = -(u[t][x + 1][y] + (a * tau / (2 * hy * hy)) * (
                            u[t][x + 1][y - 1] - 2 * u[t][x + 1][y] + u[t][x +
1][y + 1]))

            # Последняя строка
            matrix[Nx - 2][Nx - 3] = a_i
            matrix[Nx - 2][Nx - 2] = b_i
            d[Nx - 2] = -(u[t][Nx - 1][y] + (a * tau / (2 * hy * hy)) * (
                        u[t][Nx - 1][y - 1] - 2 * u[t][Nx - 1][y] + u[t][Nx -
1][y + 1]) + a * tau / (
                                    2 * hx * hx) * phi4(y * hy, (t + 1 / 2)
* tau))

            # Решем СЛАУ методом прогонки
```

```
            ans = np.linalg.solve(matrix, d)
            p = tmp[1:Nx, y]
            tmp[1:Nx, y] = ans

        # Меняем краевые условия во временном массиве на шаге tau+1/2
        tmp[0][:] = np.array([phi3(j * hy, (t + 1 / 2) * tau) for j in range(Ny
+ 1)])
        tmp[Nx][:] = np.array([phi4(j * hy, (t + 1 / 2) * tau) for j in range(Ny
+ 1)])
        tmp[:][0] = np.array([phi1(i * hx, (t + 1 / 2) * tau) for i in range(Nx
+ 1)])
        tmp[:][Ny] = np.array([phi2(i * hx, (t + 1 / 2) * tau) for i in range(Nx
+ 1)])

        # Второй дробный шаг
        for x in range(1, Nx):
            # Заполнение матрицы для метода прогонки на 2-ом дробном шаге
            matrix = np.zeros((Ny - 1, Ny - 1))
            d = np.zeros(Ny - 1)
            a_i = a * tau / (2 * hy * hy)
            b_i = -(a * tau / (hy * hy) + 1)
            c_i = a * tau / (2 * hy * hy)

            # Первая строка
            matrix[0][0] = b_i
            matrix[0][1] = c_i
            d[0] = -(tmp[x][1] + (a * tau / (2 * hx * hx)) * (
                        tmp[x - 1][1] - 2 * tmp[x][1] + tmp[x + 1][1]) + a *
tau / (2 * hy * hy) * phi1(x * hx,

(t + 1) * tau))

            # Строки с первой по N-2
            for y in range(1, Ny - 2):
                matrix[y][y - 1] = a_i
                matrix[y][y] = b_i
                matrix[y][y + 1] = c_i
                d[y] = -(tmp[x][y + 1] + (a * tau / (2 * hx * hx)) * (
                            tmp[x - 1][y + 1] - 2 * tmp[x][y + 1] + tmp[x +
1][y + 1]))

            # Последняя строка
            matrix[Ny - 2][Ny - 3] = a_i
```

```
            matrix[Ny - 2][Ny - 2] = b_i
            d[Ny - 2] = -(tmp[x][Ny - 1] + (a * tau / (2 * hx * hx)) * (
                        tmp[x - 1][Ny - 1] - 2 * tmp[x][Ny - 1] + tmp[x + 1][Ny
- 1]) + a * tau / (2 * hy * hy) * phi2(
                x * hx, (t + 1) * tau))

            # Решем СЛАУ методом прогонки
            ans = np.linalg.solve(matrix, d)
            u[t + 1, x, 1:Ny] = ans

    return u

def FractionalStepsMethod(Nt, Nx, Ny, tau, hx, hy):
    u = np.zeros((Nt + 1, Nx + 1, Ny + 1))

    # Заполняем краевые условия 1-го рода
    for t in range(Nt + 1):
        for x in range(Nx + 1):
            u[t][x][0] = phi1(x * hx, t * tau)
            u[t][x][Ny] = phi2(x * hx, t * tau)

    for t in range(Nt + 1):
        for y in range(Ny + 1):
            u[t][0][y] = phi3(y * hy, t * tau)
            u[t][Nx][y] = phi4(y * hy, t * tau)

    for x in range(Nx + 1):
        for y in range(Ny + 1):
            u[0][x][y] = psi(y * hy, x * hx)

    # Выполнение схемы метода переменных направлений
    for t in range(Nt):
        # Первый дробный шаг
        tmp = deepcopy(u[t])  # Временная переменная для хранения промежуточного
состояния на шаге tau+1/2
        for y in range(1, Ny):
            # Заполнение матрицы для метода прогонки на 1-ом дробном шаге
            matrix = np.zeros((Nx - 1, Nx - 1))
            d = np.zeros(Nx - 1)
            a_i = a * tau / (hx * hx)
            b_i = -(a * tau * 2 / (hx * hx) + 1)
            c_i = a * tau / (hx * hx)
```

```
            # Первая строка
            matrix[0][0] = b_i
            matrix[0][1] = c_i
            d[0] = -(u[t][1][y] + a * tau / (hx * hx) * phi3(y * hy, (t + 1 / 2) * tau))

            # Строки с первой по N-2
            for x in range(1, Nx - 2):
                matrix[x][x - 1] = a_i
                matrix[x][x] = b_i
                matrix[x][x + 1] = c_i
                d[x] = -u[t][x + 1][y]

            # Последняя строка
            matrix[Nx - 2][Nx - 3] = a_i
            matrix[Nx - 2][Nx - 2] = b_i
            d[Nx - 2] = -(u[t][Nx - 1][y] + a * tau / (hx * hx) * phi4(y * hy, (t + 1 / 2) * tau))

            # Решем СЛАУ методом прогонки
            ans = np.linalg.solve(matrix, d)
            p = tmp[1:Nx, y]
            tmp[1:Nx, y] = ans

        # Меняем краевые условия во временном массиве на шаге tau+1/2
        tmp[0][:] = np.array([phi3(j * hy, (t + 1 / 2) * tau) for j in range(Ny + 1)])
        tmp[Nx][:] = np.array([phi4(j * hy, (t + 1 / 2) * tau) for j in range(Ny + 1)])
        tmp[:][0] = np.array([phi1(i * hx, (t + 1 / 2) * tau) for i in range(Nx + 1)])
        tmp[:][Ny] = np.array([phi2(i * hx, (t + 1 / 2) * tau) for i in range(Nx + 1)])

        # Второй дробный шаг
        for x in range(1, Nx):
            # Заполнение матрицы для метода прогонки на 2-ом дробном шаге
            matrix = np.zeros((Ny - 1, Ny - 1))
            d = np.zeros(Ny - 1)
            a_i = a * tau / (hy * hy)
            b_i = -(a * tau * 2/ (hy * hy) + 1)
            c_i = a * tau / (hy * hy)
```

```
            # Первая строка
            matrix[0][0] = b_i
            matrix[0][1] = c_i
            d[0] = -(tmp[x][1] + a * tau / (hy * hy)*phi1(x*hx, (t + 1) * tau))

            # Строки с первой по N-2
            for y in range(1, Ny - 2):
                matrix[y][y - 1] = a_i
                matrix[y][y] = b_i
                matrix[y][y + 1] = c_i
                d[y] = -tmp[x][y + 1]

            # Последняя строка
            matrix[Ny - 2][Ny - 3] = a_i
            matrix[Ny - 2][Ny - 2] = b_i
            d[Ny - 2] = -(tmp[x][Ny - 1] + a * tau / (hy * hy) * phi2(x * hx,
(t + 1) * tau))

            # Решем СЛАУ методом прогонки
            ans = solve(matrix, d)
            u[t + 1, x, 1:Ny] = ans

    return u

def main():
    Nx = 50
    Ny = 50
    Nt = 40

    lx = mu1 * (np.pi / 2)
    ly = mu2 * (np.pi / 2)
    T = 2
    hx = lx / Nx
    hy = ly / Ny
    tau = T / Nt

    u1 = VariableDirectionMethod(Nt, Nx, Ny, tau, hx, hy)
    show_solution(Nx, Ny, Nt, hx, hy, tau, U, u1)
    u2 = FractionalStepsMethod(Nt, Nx, Ny, tau, hx, hy)
    show_solution(Nx, Ny, Nt, hx, hy, tau, U, u2)
    show_errors(Nt, lx, tau, U)

main()
```