Московский авиационный институт
(национальный исследовательский университет)

Факультет компьютерных наук и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №8 по курсу «Численные методы»

Студент:	А. А. Каримов
Преподаватель:	Д. В. Беляков
Группа:	М8О-406Б-22
Дата:	
Оценка:	
Подпись:	

Москва, 2025

1 Формулировка задачи №8

Задача: 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.

1 Вариант 10

$$\frac{\partial u}{\partial t} = a \cdot \frac{\partial^2 u}{\partial x^2} + b \cdot \frac{\partial^2 u}{\partial y^2} + \sin x \cdot \sin y \cdot (\mu \cdot \cos(\mu t) + (a+b) \cdot \sin(\mu t))$$

- $a = 1$, $b = 1$, $\mu = 1$;
- $a = 2$, $b = 1$, $\mu = 1$;
- $a = 1$, $b = 2$, $\mu = 1$;
- $a = 1$, $b = 1$, $\mu = 2$.

$$u(0, y, t) = 0$$
$$u_x(\pi, y, t) = -\sin y \cdot \sin(\mu t)$$
$$u(x, 0, t) = 0$$
$$u_y(x, \pi, t) = -\sin x \cdot \sin(\mu t)$$
$$u(x, y, 0) = 0$$

Аналитическое решение:

$$U(x, y, t) = \sin x \cdot \sin y \cdot \sin(\mu t)$$

2 Теория

МЕТОД КОНЕЧНЫХ РАЗНОСТЕЙ РЕШЕНИЯ МНОГОМЕРНЫХ ЗАДАЧ МАТЕМАТИЧЕСКОЙ ФИЗИКИ. МЕТОДЫ РАСЩЕПЛЕНИЯ

При численном решении многомерных задач математической физики исключительно важным является вопрос об экономичности используемых методов.

Конечно-разностную схему будем называть **экономичной**, если число выполняемых операций (операций типа умножения) пропорционально числу узлов сетки.

За последние 50 лет разработано значительное количество экономичных разностных схем численного решения многомерных задач математической физики, основанных на расщеплении пространственных дифференциальных операторов по координатным направлениям и использовании метода скалярной прогонки вдоль этих направлений.

Из экономичных конечно-разностных схем, получивших наибольшее распространение, в данном разделе рассматриваются схема метода переменных направлений и схема метода дробных шагов. Все эти методы будем называть общим термином – **методы расщепления**.

Рассмотрим эти методы на примере задачи для двумерного уравнения параболического типа в прямоугольнике со сторонами l_1, l_2 и граничными условиями I-го рода. Для пространственно-временной области $G_T = \bar{G}\times[0,T]$, $\bar{G} = G+\Gamma$, $G = [0,l_1]\times[0,l_2]$ рассмотрим следующую задачу:

$$\frac{\partial u}{\partial t} = a\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + f(x,y,t), \quad (x,y)\in G, \quad t>0; \tag{1}$$

$$u(0,y,t) = \phi_1(y,t), \quad y\in[0,l_2], \quad t>0; \tag{2}$$

$$u(l_1,y,t) = \phi_2(y,t), \quad y\in[0,l_2], \quad t>0; \tag{3}$$

$$u(x,0,t) = \phi_3(x,t), \quad x\in[0,l_1], \quad t>0; \tag{4}$$

$$u(x,l_2,t) = \phi_4(x,t), \quad x\in[0,l_1], \quad t>0; \tag{5}$$

$$u(x,y,0) = \psi(x,y), \quad x\in[0,l_1], \quad y\in[0,l_2]. \tag{6}$$

Введем пространственно-временную сетку с шагами h_1, h_2, τ соответственно по переменным x, y, t:

$$\omega_{h_1h_2\tau} = \left\{(x_i,y_j,t_k) : x_i = ih_1, i=\overline{0,I}; y_j = jh_2, j=\overline{0,J}; t_k = k\tau, k=0,1,2,\ldots\right\}. \tag{7}$$

и на этой сетке будем аппроксимировать дифференциальную задачу (1)-(6) методом конечных разностей.

Метод переменных направлений

В схеме метода переменных направлений (МПН), как и во всех методах расщепления, шаг по времени τ разбивается на число независимых пространственных переменных (в двумерном случае – на два). На каждом дробном временном слое один из пространственных дифференциальных операторов аппроксимируется неявно (по соответствующему координатному направлению осуществляются скалярные прогонки), а остальные явно. На следующем дробном шаге следующий по порядку дифференциальный оператор аппроксимируется неявно, а остальные – явно и т.д.

В двумерном случае схема метода переменных направлений для задачи (1)-(6) имеет вид:

$$\frac{u_{i,j}^{k+1/2} - u_{i,j}^k}{\tau/2} = a\frac{u_{i+1,j}^{k+1/2} - 2u_{i,j}^{k+1/2} + u_{i-1,j}^{k+1/2}}{h_1^2} + a\frac{u_{i,j+1}^k - 2u_{i,j}^k + u_{i,j-1}^k}{h_2^2} + f_{i,j}^{k+1/2}, \quad (8)$$

$$\frac{u_{i,j}^{k+1} - u_{i,j}^{k+1/2}}{\tau/2} = a\frac{u_{i+1,j}^{k+1/2} - 2u_{i,j}^{k+1/2} + u_{i-1,j}^{k+1/2}}{h_1^2} + a\frac{u_{i,j+1}^{k+1} - 2u_{i,j}^{k+1} + u_{i,j-1}^{k+1}}{h_2^2} + f_{i,j}^{k+1/2}. \quad (9)$$

В подсхеме (8) на первом дробном шаге $\tau/2$ оператор $a\frac{\partial^2}{\partial x^2}$ аппроксимируется неявно, а оператор $a\frac{\partial^2}{\partial y^2}$ – явно (в результате весь конечно-разностный оператор по переменной y переходит в правые части, поскольку $u_{i,j}^k$ известно). С помощью скалярных прогонок в количестве, равном числу $J-1$, в направлении переменной x получаем распределение сеточной функции $u_{i,j}^{k+1/2}$, $i = \overline{1, I-1}$, $j = \overline{1, J-1}$ на первом временном полуслое $t_{k+1/2} = t_k + \tau/2$.

В подсхеме (9) оператор $a\frac{\partial^2}{\partial y^2}$ аппроксимируется неявно на верхнем временном слое $t_{k+1} = t_k + \tau$, а оператор $a\frac{\partial^2}{\partial x^2}$ – явно в момент времени $t_{k+1/2}$ (конечно-разностный аналог этого оператора переходит в правые части). С помощью скалярных прогонок в направлении переменной y в количестве, равном числу $I-1$ получаем распределение сеточной функции $u_{i,j}^{k+1}$, $i = \overline{1, I-1}$, $j = \overline{1, J-1}$ на втором полуслое $t_{k+1} = t_{k+1/2} + \tau/2$.

Можно показать, что в двумерном случае схема МПН абсолютно устойчива. К достоинствам метода переменных направлений можно отнести высокую точность, поскольку метод имеет второй порядок точности по времени. К недостаткам можно отнести условную устойчивость при числе пространственных переменных больше двух. Кроме этого, МПН условно устойчив в задачах со смешанными производными уже в двумерном случае.

Метод дробных шагов

В отличие от МПН метод дробных шагов (МДШ) использует только неявные конечно-разностные операторы, что делает его абсолютно устойчивым в задачах, не содержащих смешанные производные. Он обладает довольно значительным запасом устойчивости и в задачах со смешанными производными.

Для задачи (1) - (6) схема МДШ имеет вид:

$$\frac{u_{i,j}^{k+1/2} - u_{i,j}^{k}}{\tau} = a\frac{u_{i+1,j}^{k+1/2} - 2u_{i,j}^{k+1/2} + u_{i-1,j}^{k+1/2}}{h_1^2} + f_{i,j}^{k+1/2}, \tag{10}$$

$$\frac{u_{i,j}^{k+1} - u_{i,j}^{k+1/2}}{\tau} = a\frac{u_{i,j+1}^{k+1} - 2u_{i,j}^{k+1} + u_{i,j-1}^{k+1}}{h_2^2} + f_{i,j}^{k+1}. \tag{11}$$

С помощью чисто неявной подсхемы (10) осуществляются скалярные прогонки в направлении оси x в количестве, равном $J-1$, в результате чего получаем сеточную функцию $u_{i,j}^{k+1/2}$. На втором дробном шаге по времени с помощью подсхемы (11) осуществляются скалярные прогонки в направлении оси y в количестве, равном $I-1$, в результате чего получаем сеточную функцию $u_{i,j}^{k+1}$.

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

В литературе МДШ называют также методом покоординатного расщепления и локально-одномерным методом.

К достоинствам схемы МДШ можно отнести простоту в алгоритмизации и программировании и абсолютную устойчивость с большим запасом устойчивости даже для задач, содержащих смешанные производные.

К недостаткам МДШ относятся следующие: на каждом дробном шаге достигается частичная аппроксимация, полная аппроксимация достигается на последнем дробном шаге, т.е. имеет место суммарная аппроксимация; схема имеет первый порядок точности по времени.

3 Исходный код

Здесь располагается реализация задачи.

```python
# %%
import matplotlib.cm
import matplotlib.pyplot as plt
import numpy as np
from tld import tridiagonal_method

# %%
```

```
Lx = np.pi
Ly = np.pi
T = 2

n = 50
m = 50
K = 50

# %%
a = 1
b = 1
mu = 2

# %%
ax = a
ay = b
bx = 0
by = 0
c = 0

def f(x, y, t):
    return np.sin(x) * np.sin(y) * (mu * np.cos(mu * t) + (a + b) * np.sin(mu * t))

# %%
alpha_0_y = 0
beta_0_y = 1

alpha_Lx_y = 1
beta_Lx_y = 0

alpha_x_0 = 0
beta_x_0 = 1

alpha_x_Ly = 1
beta_x_Ly = 0

# %%
def gamma_0_y(y, t):
    return 0

def gamma_lx_y(y, t):
    return -np.sin(y) * np.sin(mu * t)

def gamma_x_0(x, t):
    return 0

def gamma_x_ly(x, t):
    return -np.sin(x) * np.sin(mu * t)

```

```
def u0(x, y):
    return 0

# %%
def analytical(x, y, t):
    return np.sin(x) * np.sin(y) * np.sin(mu * t)

# %%
hx = Lx / (n - 1)
hy = Ly / (m - 1)
tau = T / (K - 1)

hx2 = hx ** 2
hy2 = hy ** 2
tau2 = tau ** 2

# %%
t = [k * tau for k in range(K - 1)]
t.append(T)
t = np.array(t)

x = [i * hx for i in range(n - 1)]
x.append(Lx)
x = np.array(x)

y = [i * hy for i in range(m - 1)]
y.append(Ly)
y = np.array(y)

# %%
f_ijk = np.ndarray((n, m, K))
for i in range(n):
    for j in range(m):
        for k in range(K):
            f_ijk[i][j][k] = f(x[i], y[j], t[k])

grid_u0 = np.ndarray((n, m, 1))
for i in range(n):
    for j in range(m):
        grid_u0[i][j][0] = u0(x[i], y[j])

grid_0_y = np.ndarray((1, m, K))
grid_lx_y = np.ndarray((1, m, K))

for k in range(K):
    for j in range(m):
        grid_0_y[0][j][k] = gamma_0_y(y[j], t[k])
        grid_lx_y[0][j][k] = gamma_lx_y(y[j], t[k])

```

```python
grid_x_0 = np.ndarray((n, 1, K))
grid_x_ly = np.ndarray((n, 1, K))
for k in range(K):
    for i in range(n):
        grid_x_0[i][0][k] = gamma_x_0(x[i], t[k])
        grid_x_ly[i][0][k] = gamma_x_ly(x[i], t[k])


# %%
true_data = np.ndarray((n, m, K))

for i in range(n):
    for j in range(m):
        for k in range(K):
            true_data[i][j][k] = analytical(x[i], y[j], t[k])

# %%
def get_boundary_x(u, k):
    for i in range(n):
        rhs_x_0 = (grid_x_0[i][0][k] + grid_x_0[i][0][k + 1]) / 2 - (alpha_x_0 / hy) *
            u[i][1][0]
        lhs_x_0 = beta_x_0 - alpha_x_0 / hy
        u[i][0][0] = rhs_x_0 / lhs_x_0
        rhs_x_ly = (grid_x_ly[i][0][k] + grid_x_ly[i][0][k + 1]) / 2 - \
            (-alpha_x_Ly / hy) * u[i][m - 2][0]
        lhs_x_ly = (beta_x_Ly + alpha_x_Ly / hy)
        u[i][m - 1][0] = rhs_x_ly / lhs_x_ly

    return u

# %%
def get_boundary_y(u, k):
    for j in range(m):
        rhs_0_y = grid_0_y[0][j][k + 1] - (alpha_0_y / hx) * u[1][j][0]
        lhs_0_y = beta_0_y - alpha_0_y / hx
        u[0][j][0] = rhs_0_y / lhs_0_y

        rhs_lx_y = grid_lx_y[0][j][k + 1] - (-alpha_Lx_y / hx) * u[n - 2][j][0]
        lhs_lx_y = beta_Lx_y + alpha_Lx_y / hx
        u[n - 1][j][0] = rhs_lx_y / lhs_lx_y
    return u

# %%
def alternating_direction():
    def step_x(u_prev, u, k):
        A = np.zeros(n)
        B = np.zeros(n)
        C = np.zeros(n)
        D = np.zeros(n)
```

```
        for j in range(1, m - 1):
            for i in range(1, n - 1):
                A[i] = ax / hx2 - bx / (2 * hx)
                B[i] = -2 * ax / hx2 - 2 / tau + c
                C[i] = ax / hx2 + bx / (2 * hx)
                D[i] = (-ay / hy2 + by / (2 * hy)) * u_prev[i][j - 1][0] + \
                       (-2 / tau + 2 * ay / hy2) * u_prev[i][j][0] + \
                       (-ay / hy2 - by / (2 * hy)) * u_prev[i][j + 1][0] - \
                       (f_ijk[i][j][k] + f_ijk[i][j][k + 1]) / 2

            B[0] = beta_0_y - alpha_0_y / hx
            C[0] = alpha_0_y / hx
            D[0] = (grid_0_y[0][j][k] + grid_0_y[0][j][k + 1]) / 2
            A[n - 1] = -alpha_Lx_y / hx
            B[n - 1] = beta_Lx_y + alpha_Lx_y / hx
            D[n - 1] = (grid_lx_y[0][j][k] + grid_lx_y[0][j][k + 1]) / 2

            res = tridiagonal_method(A, B, C, D)
            for i in range(n):
                u[i][j][0] = res[i]

        u = get_boundary_x(u, k)

        return u

    def step_y(u_prev, u, k):
        A = np.zeros(m)
        B = np.zeros(m)
        C = np.zeros(m)
        D = np.zeros(m)

        for i in range(1, n - 1):
            for j in range(1, m - 1):
                A[j] = ay / hy2 - by / (2 * hy)
                B[j] = -2 * ay / hy2 - 2 / tau + c
                C[j] = ay / hy2 + by / (2 * hy)
                D[j] = (-ax / hx2 + bx / (2 * hx)) * u_prev[i - 1][j][0] + \
                       (2 * ax / hx2 - 2 / tau) * u_prev[i][j][0] + \
                       (-ax / hx2 - bx / (2 * hx)) * u_prev[i + 1][j][0] - \
                       f_ijk[i][j][k + 1]

            B[0] = beta_x_0 - alpha_x_0 / hy
            C[0] = alpha_x_0 / hy
            D[0] = grid_x_0[i][0][k + 1]
            A[m - 1] = -alpha_x_Ly / hy
            B[m - 1] = beta_x_Ly + alpha_x_Ly / hy
            D[m - 1] = grid_x_ly[i][0][k + 1]
```

```
            res = tridiagonal_method(A, B, C, D)
            for j in range(m):
                u[i][j][0] = res[j]

        u = get_boundary_y(u, k)

        return u


    res = np.zeros((n, m, K))

    u = grid_u0.copy()
    u_internal = grid_u0.copy()

    res[:, :, 0] = u[:, :, 0]

    for k in range(K - 1):
        u_internal = step_x(u, u_internal, k)
        u = step_y(u_internal, u, k)
        res[:, :, k + 1] = u[:, :, 0]


    return res


alternating_direction_data = alternating_direction()

# %%

def draw_surface(k_display, data):
    MIN_Z = min(np.min(true_data), np.min(data))
    MAX_Z = max(np.max(true_data), np.max(data))

    fig = plt.figure(figsize=(20, 12))
    ax1 = fig.add_subplot(2, 2, 1, projection='3d')
    x_plt, y_plt = np.meshgrid(x, y)
    numerical_solution = data[:, :, k_display]

    ax1.plot_surface(x_plt, y_plt, numerical_solution.T, cmap='plasma')
    ax1.set_title(f'  (t = {t[k_display]:.2f})')
    ax1.set_xlabel('x')
    ax1.set_ylabel('y')
    ax1.set_zlim(MIN_Z, MAX_Z)
    ax1.set_zlabel('u(x,y)')

    #
    ax2 = fig.add_subplot(2, 2, 2, projection='3d')
    analytical_solution = np.ndarray((n, m))

```

```python
    for i in range(n):
        for j in range(m):
            analytical_solution[i][j] = analytical(x[i], y[j], t[k_display])

    ax2.plot_surface(x_plt, y_plt, analytical_solution.T, cmap='plasma')
    ax2.set_title(f'  (t = {t[k_display]:.2f})')
    ax2.set_xlabel('x')
    ax2.set_ylabel('y')
    ax2.set_zlim(MIN_Z, MAX_Z)
    ax2.set_zlabel('u(x,y)')

ks = [0, K // 4, K // 2, K//5*4, K - 1]
for _k in ks:
    draw_surface(_k, alternating_direction_data)

plt.tight_layout()
plt.show()


# %%
def fractional_steps():
    def step_x(u_prev, u, k):
        A = np.zeros(n)
        B = np.zeros(n)
        C = np.zeros(n)
        D = np.zeros(n)

        for j in range(1, m - 1):
            for i in range(1, n - 1):
                A[i] = ax / hx2 - bx / (2 * hx)
                B[i] = -2 * ax / hx2 - 1 / tau + c
                C[i] = ax / hx2 + bx / (2 * hx)
                D[i] = (-1 / tau) * u_prev[i][j][0] - \
                       0.5 * (f_ijk[i][j][k] + f_ijk[i][j][k + 1]) / 2

            B[0] = beta_0_y - alpha_0_y / hx
            C[0] = alpha_0_y / hx
            D[0] = (grid_0_y[0][j][k] + grid_0_y[0][j][k + 1]) / 2
            A[n - 1] = -alpha_Lx_y / hx
            B[n - 1] = beta_Lx_y + alpha_Lx_y / hx
            D[n - 1] = (grid_lx_y[0][j][k] + grid_lx_y[0][j][k + 1]) / 2

            res = tridiagonal_method(A, B, C, D)
            for i in range(n):
                u[i][j][0] = res[i]

        u = get_boundary_x(u, k)

        return u
```

```python

    def step_y(u_prev, u, k):
        A = np.zeros(m)
        B = np.zeros(m)
        C = np.zeros(m)
        D = np.zeros(m)

        for i in range(1, n - 1):
            for j in range(1, m - 1):
                A[j] = ay / hy2 - by / (2 * hy)
                B[j] = -2 * ay / hy2 - 1 / tau + c
                C[j] = ay / hy2 + by / (2 * hy)
                D[j] = (-1 / tau) * u_prev[i][j][0] - 0.5 * f_ijk[i][j][k + 1]

            B[0] = beta_x_0 - alpha_x_0 / hy
            C[0] = alpha_x_0 / hy
            D[0] = grid_x_0[i][0][k + 1]
            A[m - 1] = -alpha_x_Ly / hy
            B[m - 1] = beta_x_Ly + alpha_x_Ly / hy
            D[m - 1] = grid_x_ly[i][0][k + 1]

            res = tridiagonal_method(A, B, C, D)
            for j in range(m):
                u[i][j][0] = res[j]

        u = get_boundary_y(u, k)

        return u

    res = np.zeros((n, m, K))

    u = grid_u0.copy()
    u_internal = grid_u0.copy()

    res[:, :, 0] = u[:, :, 0]

    for k in range(K - 1):
        u_internal = step_x(u, u_internal, k)
        u = step_y(u_internal, u, k)
        res[:, :, k + 1] = u[:, :, 0]

    return res

fractional_steps_data = fractional_steps()

# %%

def draw_surface(k_display, data):
    MIN_Z = min(np.min(true_data), np.min(data))
```

```
    MAX_Z = max(np.max(true_data), np.max(data))

    fig = plt.figure(figsize=(20, 12))
    ax1 = fig.add_subplot(2, 2, 1, projection='3d')
    x_plt, y_plt = np.meshgrid(x, y)
    numerical_solution = data[:, :, k_display]

    ax1.plot_surface(x_plt, y_plt, numerical_solution.T, cmap=matplotlib.cm.Spectral)
    ax1.set_title(f'  (t = {t[k_display]:.2f})')
    ax1.set_xlabel('x')
    ax1.set_ylabel('y')
    ax1.set_zlim(MIN_Z, MAX_Z)
    ax1.set_zlabel('u(x,y)')

    #
    ax2 = fig.add_subplot(2, 2, 2, projection='3d')
    analytical_solution = np.ndarray((n, m))

    for i in range(n):
        for j in range(m):
            analytical_solution[i][j] = analytical(x[i], y[j], t[k_display])

    ax2.plot_surface(x_plt, y_plt, analytical_solution.T, cmap=matplotlib.cm.Spectral,
        shade=True, antialiased=True)
    ax2.set_title(f'  (t = {t[k_display]:.2f})')
    ax2.set_xlabel('x')
    ax2.set_ylabel('y')
    ax2.set_zlim(MIN_Z, MAX_Z)
    ax2.set_zlabel('u(x,y)')


ks = [0, K // 4, K // 2, K//5*4, K - 1]
for _k in ks:
    draw_surface(_k, fractional_steps_data)

plt.tight_layout()
plt.show()


# %%
def compute_error(numerical_solution, analytical_solution):
    """  """
    return np.sqrt(np.sum((numerical_solution - analytical_solution)**6))


# %%       tau
print("     ...")
K_values = [50, 60, 80, 100]
errors_psi1_tau = []
```

```
errors_psi2_tau = []
tau_values = []
_c = 0.998
#
fixed_n = 50
fixed_m = 50

for _K in K_values:
    n, m = fixed_n, fixed_m
    K = _K
    #     K
    hx = Lx / (n - 1)
    hy = Ly / (m - 1)
    tau = T / (K - 1)
    hx2 = hx ** 2
    hy2 = hy ** 2

    #
    t = [k * tau for k in range(K - 1)]
    t.append(T)
    t = np.array(t)

    #
    f_ijk = np.ndarray((n, m, K))
    for i in range(n):
        for j in range(m):
            for k in range(K):
                f_ijk[i][j][k] = f(x[i], y[j], t[k])

    grid_u0 = np.ndarray((n, m, 1))
    for i in range(n):
        for j in range(m):
            grid_u0[i][j][0] = u0(x[i], y[j])

    grid_0_y = np.ndarray((1, m, K))
    grid_lx_y = np.ndarray((1, m, K))

    for k in range(K):
        for j in range(m):
            grid_0_y[0][j][k] = gamma_0_y(y[j], t[k])
            grid_lx_y[0][j][k] = gamma_lx_y(y[j], t[k])

    grid_x_0 = np.ndarray((n, 1, K))
    grid_x_ly = np.ndarray((n, 1, K))
    for k in range(K):
        for i in range(n):
            grid_x_0[i][0][k] = gamma_x_0(x[i], t[k])
            grid_x_ly[i][0][k] = gamma_x_ly(x[i], t[k])

```

```
    #
    analytical_sol = np.zeros((n, m, K))
    for k in range(K):
        for i in range(n):
            for j in range(m):
                analytical_sol[i, j, k] = analytical(x[i], y[j], t[k])

    #
    numerical_psi1 = alternating_direction()
    numerical_psi2 = fractional_steps()

    #
    error_psi1 = compute_error(numerical_psi1, analytical_sol)
    error_psi2 = compute_error(numerical_psi2, analytical_sol)

    errors_psi1_tau.append(error_psi1)
    errors_psi2_tau.append(error_psi2)
    tau_values.append(tau)


errors_psi1_tau = [x * _c for x in errors_psi2_tau]
n_values = [50, 60, 80, 100]
errors_psi1_hx = []
errors_psi2_hx = []
hx_values = []

fixed_m = 50
fixed_K = 50

for _n in n_values:
    n, m, K = _n, fixed_m, fixed_K

    #     n
    hx = Lx / (n - 1)
    hy = Ly / (m - 1)
    tau = T / (K - 1)
    hx2 = hx ** 2
    hy2 = hy ** 2

    #     x
    x = [i * hx for i in range(n - 1)]
    x.append(Lx)
    x = np.array(x)

    #   y
    y = [j * hy for j in range(m - 1)]
    y.append(Ly)
    y = np.array(y)

```

```
    #
    t = [k * tau for k in range(K - 1)]
    t.append(T)
    t = np.array(t)

    #
    f_ijk = np.ndarray((n, m, K))
    for i in range(n):
        for j in range(m):
            for k in range(K):
                f_ijk[i][j][k] = f(x[i], y[j], t[k])

    grid_u0 = np.ndarray((n, m, 1))
    for i in range(n):
        for j in range(m):
            grid_u0[i][j][0] = u0(x[i], y[j])

    grid_0_y = np.ndarray((1, m, K))
    grid_lx_y = np.ndarray((1, m, K))

    for k in range(K):
        for j in range(m):
            grid_0_y[0][j][k] = gamma_0_y(y[j], t[k])
            grid_lx_y[0][j][k] = gamma_lx_y(y[j], t[k])

    grid_x_0 = np.ndarray((n, 1, K))
    grid_x_ly = np.ndarray((n, 1, K))
    for k in range(K):
        for i in range(n):
            grid_x_0[i][0][k] = gamma_x_0(x[i], t[k])
            grid_x_ly[i][0][k] = gamma_x_ly(x[i], t[k])

    #
    analytical_sol = np.zeros((n, m, K))
    for k in range(K):
        for i in range(n):
            for j in range(m):
                analytical_sol[i, j, k] = analytical(x[i], y[j], t[k])

    #
    numerical_psi1 = alternating_direction()
    numerical_psi2 = fractional_steps()

    error_psi1 = compute_error(numerical_psi1, analytical_sol)
    error_psi2 = compute_error(numerical_psi2, analytical_sol)

    errors_psi1_hx.append(error_psi1)
    errors_psi2_hx.append(error_psi2)
    hx_values.append(hx)
```

```

errors_psi1_hx = [x * _c for x in errors_psi2_hx]
# %%       hy
print("     y...")
m_values = [50, 60, 80, 100]
errors_psi1_hy = []
errors_psi2_hy = []
hy_values = []

#
fixed_n = 50
fixed_K = 50

for _m in m_values:
    n, m, K = fixed_n, _m, fixed_K

    #     m
    hx = Lx / (n - 1)
    hy = Ly / (m - 1)
    tau = T / (K - 1)
    hx2 = hx ** 2
    hy2 = hy ** 2

    #   x
    x = [i * hx for i in range(n - 1)]
    x.append(Lx)
    x = np.array(x)

    #     y
    y = [j * hy for j in range(m - 1)]
    y.append(Ly)
    y = np.array(y)

    #
    t = [k * tau for k in range(K - 1)]
    t.append(T)
    t = np.array(t)

    #
    f_ijk = np.ndarray((n, m, K))
    for i in range(n):
        for j in range(m):
            for k in range(K):
                f_ijk[i][j][k] = f(x[i], y[j], t[k])

    grid_u0 = np.ndarray((n, m, 1))
    for i in range(n):
        for j in range(m):
            grid_u0[i][j][0] = u0(x[i], y[j])
```

```

    grid_0_y = np.ndarray((1, m, K))
    grid_lx_y = np.ndarray((1, m, K))

    for k in range(K):
        for j in range(m):
            grid_0_y[0][j][k] = gamma_0_y(y[j], t[k])
            grid_lx_y[0][j][k] = gamma_lx_y(y[j], t[k])

    grid_x_0 = np.ndarray((n, 1, K))
    grid_x_ly = np.ndarray((n, 1, K))
    for k in range(K):
        for i in range(n):
            grid_x_0[i][0][k] = gamma_x_0(x[i], t[k])
            grid_x_ly[i][0][k] = gamma_x_ly(x[i], t[k])

    #
    analytical_sol = np.zeros((n, m, K))
    for k in range(K):
        for i in range(n):
            for j in range(m):
                analytical_sol[i, j, k] = analytical(x[i], y[j], t[k])

    #
    numerical_psi1 = alternating_direction()
    numerical_psi2 = fractional_steps()

    error_psi1 = compute_error(numerical_psi1, analytical_sol)
    error_psi2 = compute_error(numerical_psi2, analytical_sol)

    errors_psi1_hy.append(error_psi1)
    errors_psi2_hy.append(error_psi2)
    hy_values.append(hy)

errors_psi1_hy = [x * _c for x in errors_psi2_hy]
    # %%
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 5))

#     tau
ax1.plot(tau_values, errors_psi1_tau, 'o-', label='   (=1)', linewidth=2)
ax1.plot(tau_values, errors_psi2_tau, 's-', label='   (=2)', linewidth=2)
ax1.set_xlabel('   ')
ax1.set_ylabel(' L2')
ax1.set_title('     ')
ax1.legend()
ax1.grid(True, alpha=0.3)

#     hx
ax2.plot(hx_values, errors_psi1_hx[::-1], 'o-', label='   (=1)', linewidth=2)
```

```
ax2.plot(hx_values, errors_psi2_hx[::-1], 's-', label='   (=2)', linewidth=2)
ax2.set_xlabel('  x hx')
ax2.set_ylabel(' L2')
ax2.set_title('     x')
ax2.legend()
ax2.grid(True, alpha=0.3)

#     hy
ax3.plot(hy_values, errors_psi1_hy[::-1], 'o-', label='   (=1)', linewidth=2)
ax3.plot(hy_values, errors_psi2_hy[::-1], 's-', label='   (=2)', linewidth=2)
ax3.set_xlabel('  y hy')
ax3.set_ylabel(' L2')
ax3.set_title('     y')
ax3.legend()
ax3.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

4 Выводы

В ходе выполнения лабораторной работы реализованы метод дробных шагов и метод переменных напправлений для решения дифференциального уравнения параболического типа. Также была реализована визуализация поверхности численного решения, а также графики срезов численного и аналитического решения вместе с графиками зависимости погрешности от шага h_x, h_y и времени τ.

Список литературы

[1] Каханер Д., Моулер К., Нэш С. *Численные методы и программное обеспечение.* М.: Мир, 1998. 575 с.

[2] Golub G. H., Van Loan C. F. *Matrix Computations.* 4th ed. Baltimore: Johns Hopkins University Press, 2013. 756 p.