ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В.ЛОМОНОСОВА»

ФИЗИЧЕСКИЙ ФАКУЛЬТЕТ

КАФЕДРА ОБЩЕЙ ФИЗИКИ И МОЛЕКУЛЯРНОЙ ЭЛЕКТРОНИКИ

КУРСОВАЯ РАБОТА

«ЧИСЛЕННОЕ РЕШЕНИЕ ДВУМЕРНОГО УРАВНЕНИЯ ТЕПЛОПРОВОДНОСТИ ПРИ ПОМОЩИ МЕТОДА ПЕРЕМЕННЫХ НАПРАВЛЕНИЙ»

Выполнил студент:
406 группа
Гафни Д.

____
подпись студента

Научный руководитель:
Приклонский В.И.

____
подпись научного руководителя

Москва

2020

Содержание

1 Постановка задачи

Используя метод переменных направлений, решить краевую задачу:

$$\begin{cases} \frac{\partial u}{\partial t} = \Delta u + (yt)^2, \quad 0 < x < 2, \quad 0 < y < 1, \quad t > 0 \\ \frac{\partial u}{\partial x}\big|_{x=0} = u\big|_{x=2} = 0 \\ u\big|_{y=0} = u\big|_{y=1} = 0 \\ u\big|_{t=0} = \cos(\pi x/4) \cdot y(1-y) \end{cases} \tag{1}$$

2 Численное решение

2.1 Сетка

Введем в расчетной области сетку, используя фиктивные узлы в окрестности границ, чтобы получить второй порядок аппроксимации для условий Неймана:

$$\begin{cases} x_0 = 0; \quad x_n = x_0 + nh_x, \quad n = 0, 1, ..., N; \quad x_N = 2 \longrightarrow h_x = \frac{2}{N-1} \\ y_0 = 0; \quad y_m = y_0 + mh_y, \quad m = 0, 1, ..., M; \quad y_M = 1 \longrightarrow h_y = \frac{1}{M-1} \\ t_j = j\tau, \quad j = 0, 1, ..., J; \quad t_J = T \longrightarrow \tau = \frac{T}{J} \end{cases} \tag{2}$$

На данной сетке будем рассматривать сеточную функцию $w^j_{n,m} = u(x_n, y_m, t_j)$.

2.2 Аппроксимация

2.2.1 Оператор Лапласа

Аппроксимируем оператор Лапласа $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ разностным оператором $\Lambda w = \Lambda_x w + \Lambda_y w$, где

$$\begin{aligned} \Lambda_x w &= \frac{w_{n-1,m} - 2w_{n,m} + w_{n+1,m}}{h_x^2}, \\ \Lambda_y w &= \frac{w_{n,m-1} - 2w_{n,m} + w_{n,m+1}}{h_y^2}. \end{aligned} \tag{3}$$

Данное приближение имеет второй порядок аппроксимации. Здесь и далее в соответствующих ситуациях для краткости верхний индекс j, соответствующий времени, может быть негласно опущен, как и другие.

2.2.2 Неоднородность

$$f(y,t) = (yt)^2 \longrightarrow f^j_{n,m} = (mjh_yh_t)^2, \qquad m = 0, 1, ..., M, \quad j = 0, 1, ..., J. \tag{4}$$

Неоднородность аппроксимируется точно.

2.2.3 Начальное условие

$$u\big|_{t=0} = \cos(\pi x/4)\cdot y(1-y) \longrightarrow w^0_{n,m} = \cos(\pi n h_x/4)\cdot m h_y(1-m h_y) \qquad (5)$$

Начальное условие аппроксимируется точно.

2.2.4 Граничное условие

- По x : $\begin{cases} w_{0,m} = w_{1,m} \\ w_{N,m} = 0 \end{cases} \qquad m = 0, 1, ..., M$

- По y : $\begin{cases} w_{n,0} = 0 \\ w_{n,M} = 0 \end{cases} \qquad n = 0, 1, ..., N$

Условие при $x = 0$ имеет первый порядок аппроксимации; остальные аппроксимируются точно.

2.3 Метод переменных направлений

В данном методе переход со слоя j на слой $j+1$ осуществляется в два этапа, с помощью вспомогательного промежуточного слоя $j+1/2$. Схема переменных направлений безусловно устойчива при любых шагах h_x, h_y, τ. При условии, что для начальных и граничных условий порядки аппроксимации будут не ниже первого, и с учетом вышеописанной аппроксимации дифференциальных операторов, которая имеет первый порядок, метод переменных направлений будет давать первый порядок аппроксимации в данном случае. Рассмотрим подробно переход со слоя j на промежуточный слой $j+1/2$ и дальнейший переход с промежуточного слоя $j+1/2$ на слой $j+1$.

2.3.1 Переход к промежуточному слою

Пусть значения на слое j уже известны (на самом первом шаге значения $w\,^0_{n,m}$ известны из начального условия). Перейдем на вспомогательный промежуточный слой $j+1/2$, используя **неявную схему по переменной x и явную - по переменной y**:

- Заменим выражение $\frac{\partial^2}{\partial x^2}$ разностным аналогом, взятым на слое $j{+}1/2$: $\Lambda_x w^{\,j+1/2}$.

- А выражение $\frac{\partial^2}{\partial y^2}$ разностным аналогом, взятым на слое j : $\Lambda_y w^{\,j}$.

При этом неоднороднось $f(x, y, t)$ в правой части уравнения аппроксимируем на промежуточным слое $j+1/2$.

В результате придем к разностному уравнению:

$$\frac{w^{\,j+1/2} - w^{\,j}}{0.5\tau} = \Lambda_x w^{\,j+1/2} + \Lambda_y w^{\,j} + f^{j+1/2} \qquad (6)$$

Перейдем к конкретной задаче и добавим соответствующее граничное условие:

$$\begin{cases} w_{n,m}^{j+1/2} - w_{n,m}^{j} = \left(\frac{\tau}{2{h_x}^2} w_{n+1,\ m}^{j+1/2} - \frac{\tau}{{h_x}^2} w_{n,\ m}^{j+1/2} + \frac{\tau}{2{h_x}^2} w_{n-1,\ m}^{j+1/2} \right) + \\ +\left(\frac{\tau}{2{h_y}^2} w_{n,\ m+1}^{j} - \frac{\tau}{{h_y}^2} w_{n,\ m}^{j} + \frac{\tau}{2{h_y}^2} w_{n,\ m-1}^{j} \right) + \frac{\tau}{2}(mjh_yh_t)^2 \\ w_{0,m}^{j+1/2} = w_{1,m}^{j+1/2}, \quad w_{N,m}^{j+1/2} = w_{N-1,m}^{j+1/2} \end{cases} \tag{7}$$

$$\text{где} \ \ n = 1, 2, ..., N-1, \quad m = 1, 2, ..., M-1$$

При каждом фиксированным $n = 0, 1, ..., N-1$ можно переписать:

$$\begin{cases} \frac{\tau}{2{h_y}^2} w_{n,m-1}^{j+1} - \left(1 + \frac{\tau}{{h_y}^2}\right) w_{n,m}^{j+1} + \frac{\tau}{2{h_y}^2} w_{n,\ m+1}^{j+1} = \\ -\left[w_{n,\ m}^{j+1/2} + \frac{\tau}{2{h_x}^2} \left(w_{n+1,\ m}^{j+1/2} - 2w_{n,\ m}^{j+1/2} + w_{n,\ m}^{j+1/2} \right) + \frac{\tau}{2}(mjh_yh_t)^2 \right] \\ \\ w_{0,m}^{j+1} = w_{1,m}^{j+1}, \quad w_{N,m}^{j+1} = 0 \end{cases} \tag{8}$$

$$\text{где} \quad m = 1, 2, ..., M-1$$

Введем обозначения:
$\chi_n = w_{n,m}^{j+1/2}, \quad \chi_{n-1} = 0, \quad \chi_{n+1} = w_{n+1,m}^{j+1/2},$
$A^x = B^x = \frac{\tau}{2{h_x}^2}, \quad C^x = \left(1 + \frac{\tau}{2{h_x}^2}\right),$
$F_n^x = w_{n,\ m}^{j} + \frac{\tau}{2{h_y}^2} \left(w_{n,\ m+1}^{j} - 2w_{n,\ m}^{j} + w_{n,\ m-1}^{j} \right) + \frac{\tau}{2}(mjh_yh_t)^2$.

Получим простую систему, состоящую из уравнения, в котором неизвестные связаны рекуррентным соотношением, и граничных условий:

$$\begin{cases} A^x\chi_{n-1} - C^x\chi_n + B^x\chi_{n+1} = -F_n^x, \\ \chi_0 = \chi_1, \quad \chi_N = \chi_{N-1}. \end{cases} \qquad n = 1, ..., N-1 \tag{9}$$

Данную систему можно решить методом прогонки.

И снова получим простую систему уже для перехода $j + 1/2 \longrightarrow j + 1$, состоящую из уравнения, в котором неизвестные связаны рекуррентным соотношением, и граничных условий:

$$\begin{cases} A^y\gamma_{n-1} - C^y\gamma_n + B^y\gamma_{n+1} = -F_m^y, \\ \gamma_0 = \gamma_1, \quad \gamma_n = \gamma_{n-1}. \end{cases} \qquad m = 1, ..., M-1 \tag{10}$$

Данная система аналогично решается методом прогонки.

2.3.2 Метод прогонки

Рассмотрим систему для перехода $j \longrightarrow j+1/2$:

$$\begin{cases} A^x \chi_{n-1} - C^x \chi_n + B^x \chi_{n+1} = -F_n^x, \\ \chi_0 = \chi_1, \quad \chi_N = \chi_{N-1}. \end{cases} \quad n = 1, ..., N-1 \tag{11}$$

Система для перехода $j+1/2 \longrightarrow j+1$ будет решаться абсолютно аналогично.

2.3.3 Прямой ход прогонки

Идея заключается в первоначальном нахождении всех коэффициентов прогонки α_n и β_n через известные α_1 и β_1.
Рекуррентное соотношение: $\chi_n = \alpha_{n+1}\chi_{n+1} + \beta_{n+1}$
Тогда $\chi_{n-1}(\chi_n)$: $\chi_{n-1} = \alpha_n \chi_n + \beta_n = \alpha_n \alpha_{n+1} \chi_{n+1} + \alpha_n \beta_{n+1} + \beta_n$
В результате после подстановки в первое уравнение системы, получим:

$$A^x \left(\alpha_n \alpha_{n+1} \chi_{n+1} + \alpha_n \beta_{n+1} + \beta_n\right) - C^x \left(\alpha_{n+1} \chi_{n+1} + \beta_{n+1}\right) + B^x \chi_{n+1} = -F_n^x$$

Приравняв коэффициенты при одинаковых степенях χ_{n+1}:
χ_{n+1} : $A^x \alpha_n \alpha_{n+1} - C^x \alpha_{n+1} + B^x = 0$
χ_{n+1}^0 : $A^x \alpha_n \beta_{n+1} + A^x \beta_n - C^x \beta_{n+1} + F_n^x = 0$
Выразим $\alpha_{n+1}(\alpha_n)$ и $\beta_{n+1}(\beta_n)$:

$$\alpha_{n+1} = \frac{B^x}{C^x - A^x \alpha_n}, \quad \beta_{n+1} = \frac{A^x \beta_n + F_n^x}{C^x - A^x \alpha_n}, \; n = 1, 2, 3, ..., N-1$$

Из первых граничных условий:

$$\chi_0 = k_1 \chi_1 + \mu_1 = \chi_1 \Rightarrow \alpha_1 = k_1 = 1, \beta_1 = \mu_1 = 0$$

В итоге получим формулы для прямой прогонки:

$$\begin{cases} \alpha_{n+1} = \frac{B^x}{C^x - A^x \alpha_n}, \quad \beta_{n+1} = \frac{A^x \beta_n + F_n^x}{C^x - A^x \alpha_n}, \; n = 1, 2, 3, ..., N-1 \\ \alpha_1 = 1, \quad \beta_1 = 0 \end{cases} \tag{12}$$

2.3.4 Обратный ход прогонки

По известному χ_N и найденымм ранее коэффициентам α_n, β_n вычисляем значения χ_n.

$$\chi_n = \alpha_{n+1} \chi_{n+1} + \beta_{n+1}$$

Из вторых граничных условий:

$$\chi_N = k_2\chi_{N-1} + \mu_2 = \chi_{N-1} \Rightarrow \quad k_2 = 1, \quad \mu_2 = 0$$

Откуда получим:

$$\chi_N = \frac{k_2\beta_N + \mu_2}{1 - \alpha_N k_2}$$

Используем, что $k_2 = 1, \mu_2 = 0$, и получим итоговые формулы для обратной прогонки:

$$\begin{cases} \chi_n = \alpha_{n+1}\chi_{n+1} + \beta_{n+1} \\ \chi_N = \frac{\beta_N}{1-\alpha_N} \end{cases} \tag{13}$$

2.3.5 Сложность

Как видим, здесь для прямой прогонки необходимо $0(N)$ действий для одной системы. Поскольку систем таких $M-1$, суммарная сложность будет $O(NM)$. Аналогично для обратной прогонки: сложность $0(M)$ для одной системы, а систем $N-1$. Таким образом, для обратной прогонки сложность будет $O(MN)$. Суммарная сложность перехода $j+1 \longrightarrow j+1/2$ будет $O(NM)$. Такая же сложность будет и для перехода $j+1/2 \longrightarrow j+1$. В итоге, для перехода $j \longrightarrow j+1$ сложность будет все так же $O(NM)$, а сложность всей задачи $O(NMJ)$. Именно поэтому метод переменных направлений относится к так называемым экономичным схемам. Экономичные схемы сочетают в себе достоинства явных и неявных схем (требуют при переходе со слоя на слой числа арифметических операций, пропорционального числу узлов сетки, и являются безусловно устойчивыми, соответственно).

3 Реализация

Код выполнен на языке Python 3.7

```
import os
from tqdm.notebook import tqdm
import numpy as np
from numba import njit
import plotly
import plotly.graph_objs as go


# Plotly settings
layout = go.Layout(
    scene=dict(
        aspectmode="cube",
        camera=dict(eye=dict(x=-2, y=1.5, z=1)),
        xaxis=dict(
            title="x",
            gridcolor="rgb(255, 255, 255)",
            zerolinecolor="rgb(255, 255, 255)",
            showbackground=True,
            backgroundcolor="rgb(200, 200, 230)",
```

```python
        ),
        yaxis=dict(
            title="y",
            gridcolor="rgb(255, 255, 255)",
            zerolinecolor="rgb(255, 255, 255)",
            showbackground=True,
            backgroundcolor="rgb(230, 200,230)",
        ),
        zaxis=dict(
            title="u(x, y, t)",
            gridcolor="rgb(255, 255, 255)",
            zerolinecolor="rgb(255, 255, 255)",
            showbackground=True,
            backgroundcolor="rgb(230, 230,200)",
        ),
    ),
    autosize=False,
    width=800,
    height=600,
    margin=dict(r=20, b=10, l=10, t=10),
)
camera = dict(eye=dict(x=-2, y=2, z=1))


@njit
def TDMA(coeffs, F):
    """
    Tridiagonal matrix algorithm.
    :param coeffs: matrix coefficients
    :param F: right side of the equations array
    :return: solution
    """
    N = F.size
    x = np.empty(N)
    A = np.diag(coeffs, -1)
    B = np.diag(coeffs, 1)
    C = np.diag(coeffs, 0)
    alpha = np.empty(N - 1)
    alpha[0] = -B[0] / C[0]
    beta = np.empty(N - 1)
    beta[0] = F[0] / C[0]

    # Straight run
    for i in range(1, N - 1):
        alpha[i] = -B[i] / (A[i - 1] * alpha[i - 1] + C[i])
        beta[i] = (F[i] - A[i - 1] * beta[i - 1]) / (A[i - 1] * alpha[i - 1] + C[i])
    x[-1] = (F[-1] - A[-1] * beta[-1]) / (C[-1] + A[-1] * alpha[-1])

    # Return run
    for i in range(N - 2, -1, -1):
        x[i] = alpha[i] * x[i + 1] + beta[i]

    return x


class HeatEquationSolver2D:
    """
    2D heat equation solver
    """

    def __init__(
        self, X_START=0, X_END=2, Y_START=0, Y_END=1, T_START=0, T_END=20, N=5, M
```

```
    =5, J=5
    ):

        self.X_START = X_START
        self.X_END = X_END
        self.Y_START = Y_START
        self.Y_END = Y_END
        self.T_START = T_START
        self.T_END = T_END
        self.N = N
        self.M = M
        self.J = J

        self.x = np.linspace(X_START, X_END, N)
        self.y = np.linspace(Y_START, Y_END, M)
        self.t = np.linspace(T_START, T_END, J)

        self.dx = self.x[1] - self.x[0]
        self.dy = self.y[1] - self.y[0]
        self.dt = self.t[1] - self.t[0]

    def initialize(
        self,
        a=1,
        f=lambda x, y, t: 0,
        fi=lambda x, y: 0,
        alpha1x=0,
        alpha2x=0,
        beta2x=0,
        beta1x=0,
        mu1x=lambda y, t: 0,
        mu2x=lambda y, t: 0,
        alpha1y=0,
        alpha2y=0,
        beta2y=0,
        beta1y=0,
        mu1y=lambda x, t: 0,
        mu2y=lambda x, t: 0,
    ):
        """
        Problem parameters initialization
        """

        self.a = a  # Laplace operator coefficient
        self.f = f  # Heterogeneity - "heat source"
        self.fi = fi  # Initial condition

        self.alpha1x = alpha1x  # Coefficient for left Neumann condition for x
        self.alpha2x = alpha2x  # Coefficient for right Neumann condition for x
        self.beta1x = beta1x  # Coefficient for left Dirichlet condition for x
        self.beta2x = beta2x  # Coefficient for right Dirichlet condition for x
        self.mu1x = mu1x  # Right side of the left boundary condition for x
        self.mu2x = mu2x  # Right side of the right boundary condition for x

        self.alpha1y = alpha1y  # Coefficient for left Neumann condition for y
        self.alpha2y = alpha2y  # Coefficient for right Neumann condition for y
        self.beta1y = beta1y  # Coefficient for left Dirichlet condition for y
        self.beta2y = beta2y  # Coefficient for right Dirichlet condition for y
        self.mu1y = mu1y  # Right side of the left boundary condition for y
        self.mu2y = mu2y  # Right side of the right boundary condition for y

        self.u = np.empty((self.J, self.N, self.M))  # u(x, y, t) initialisation

```

```
        self.u[0] = [
            [self.fi(x, y) for y in self.y] for x in self.x
        ]  # Apply initial condition

    def calculate_layer(self, j):
        """
        Step from j-1 to j layer
        """

        step_x = self.dx
        step_y = self.dt
        step_t = self.dt / 2  # Divide dt by 2 for transition to the intermediate layer

        # Transition to the intermediate layer, TDMA coefficients definition
        middle_layer = [np.array([0] * self.N)] * (self.M)
        for m in range(self.M):
            coeffs = np.empty((self.N, self.N))
            # Boundary conditions
            coeffs[0][0] = self.beta1x - self.alpha1x / step_x
            coeffs[0][1] = self.alpha1x / step_x
            coeffs[-1][-2] = -self.alpha2x / step_x
            coeffs[-1][-1] = self.alpha2x / step_x + self.beta2x

            for i in range(1, self.N - 1):
                coeffs[i][i - 1] = -self.a ** 2 * step_t / step_x ** 2
                coeffs[i][i] = 2 * self.a ** 2 * step_t / step_x ** 2 + 1
                coeffs[i][i + 1] = -self.a ** 2 * step_t / step_x ** 2

            F = np.empty(self.N)
            F[0], F[-1] = (
                self.mu1x(self.y[m], self.t[j] + step_t),
                self.mu2x(self.y[m], self.t[j] + step_t),
            )
            for k in range(1, self.N - 1):
                F[k] = (
                    self.f(self.x[k], self.y[m], self.t[j] + step_t) * step_t
                    + self.u[j - 1][k][m]
                    + self.a ** 2
                    * step_t
                    / step_y ** 2
                    * (
                        self.u[j - 1][k - 1][m]
                        - 2 * self.u[j - 1][k][m]
                        + self.u[j - 1][k + 1][m]
                    )
                )

            middle_layer[m] = TDMA(coeffs, F)  # Apply TDMA

        middle_layer = np.array(middle_layer).T

        # Transition to the next layer, TDMA coefficients definition
        new_layer = [np.array([0] * self.M)] * (self.N)

        for n in range(self.N):
            coeffs = np.empty((self.M, self.M))
            # Boundary conditions
            coeffs[0][0] = self.beta1y - self.alpha1y / step_y
            coeffs[0][1] = self.alpha1y / step_y
            coeffs[-1][-2] = self.alpha2y / step_y
            coeffs[-1][-1] = self.alpha2y / step_y + self.beta2y

```

```python
            for i in range(1, self.M - 1):
                coeffs[i][i - 1] = -self.a ** 2 * step_t / step_y ** 2
                coeffs[i][i] = 2 * self.a ** 2 * step_t / step_y ** 2 + 1
                coeffs[i][i + 1] = -self.a ** 2 * step_t / step_y ** 2

            F = np.empty(self.M)
            F[0], F[-1] = (
                self.mu1y(self.x[n], self.t[j] + 2 * step_t),
                self.mu2y(self.x[n], self.t[j] + 2 * step_t),
            )
            for k in range(1, self.M - 1):
                F[k] = (
                    self.f(self.x[n], self.y[k], self.t[j] + 2 * step_t) * step_t
                    + middle_layer[n][k]
                    + self.a ** 2
                    * step_t
                    / step_x ** 2
                    * (
                        middle_layer[n][k - 1]
                        - 2 * middle_layer[n][k]
                        + middle_layer[n][k + 1]
                    )
                )

            new_layer[n] = TDMA(coeffs, F)  # Apply TDMA

        new_layer = np.array(new_layer)

        return new_layer

    def solve(self):
        print("Calculating...")
        for j in tqdm(range(1, self.J)):
            new_layer = self.calculate_layer(j)
            self.u[j] = new_layer

    def plot_state(self, n=0, save=False, filename=None):
        """
        Plot the solution state at moment n% from maximum time.
        :param n: percent of maximum time when to plot the solution
        :param save: if to save on disk
        :param filename: file name to save to
        :return: plotly.graph_objs.Figure with the state of the solution
        """
        if filename is None:
            filename = f"state-{n}%"
        if not filename.endswith(".html"):
            filename += ".html"
        num = int(round(n / 100 * self.J, 0))
        if num > self.J - 1:
            num = self.J - 1

        data = [go.Surface(x=self.x, y=self.y, z=self.u[num].T)]
        fig = go.Figure(data=data, layout=layout)
        if save:
            if not os.path.exists("results"):
                os.makedirs("results")
            plotly.offline.plot(fig, filename=f"results//{filename}")
        return fig

    def plot_initial_state(self, save=False, filename=None):
        """
        Plot the initial state.
```

```
        :param save: if to save on disk
        :param filename: file name to save to
        :return: plotly.graph_objs.Figure with the initial state of the solution
        """
        if filename is None:
            filename = "initial_state"
        return self.plot_state(n=0, save=save, filename=filename)

    def show_evolution(self, save=False, filename=None):
        """
        Animate evolution of the solution.
        :param save: if to save on disk
        :param filename: file name to save to
        :return: plotly.graph_objs.Figure with the evolution of the solution
        """
        if filename is None:
            filename = "evolution"
        if not filename.endswith(".html"):
            filename += ".html"
        fig = go.Figure(
            data=go.Surface(x=self.x, y=self.y, z=self.u[0].T), layout=layout
        )

        # Scale fix
        fig.layout.scene.zaxis.range = [np.min(self.u), np.max(self.u)]
        fig.layout.scene.yaxis.range = [self.Y_START, self.Y_END]
        fig.layout.scene.xaxis.range = [self.X_START, self.X_END]
        fig.layout.coloraxis.cmin = np.min(self.u)
        fig.layout.coloraxis.cmax = np.max(self.u)
        fig.layout.scene.xaxis.autorange = False
        fig.layout.scene.yaxis.autorange = False
        fig.layout.scene.zaxis.autorange = False

        frames = []
        for j in range(self.J):
            frames.append(
                go.Frame(data=[go.Surface(x=self.x, y=self.y, z=self.u[j].T)])
            )

        fig.frames = frames

        fig.layout.updatemenus = [
            {
                "buttons": [
                    {
                        "args": [
                            None,
                            {
                                "frame": {"duration": 100, "redraw": True},
                                "fromcurrent": True,
                            },
                        ],
                        "label": "Play",
                        "method": "animate",
                    },
                    {
                        "args": [
                            [None],
                            {
                                "frame": {"duration": 0, "redraw": True},
                                "mode": "immediate",
                            },
                        ],
```

```
                        "label": "Pause",
                        "method": "animate",
                    },
                ],
                "direction": "left",
                "pad": {"r": 10, "t": 87},
                "showactive": False,
                "type": "buttons",
                "x": 0.1,
                "xanchor": "right",
                "y": 0,
                "yanchor": "top",
            }
        ]
        if save:
            if not os.path.exists("results"):
                os.makedirs("results")
            plotly.offline.plot(fig, filename=f"results//{filename}")
        return fig
```

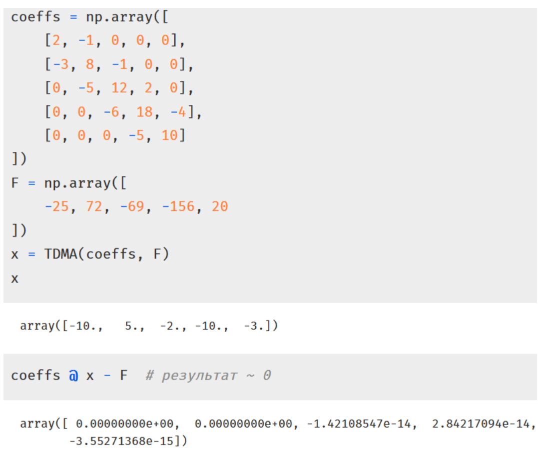
Реализация численного решения

4 Результаты

4.1 Метод прогонки

Решим систему линейных уравнений с трехдиагональной матрицей коэффициентов:

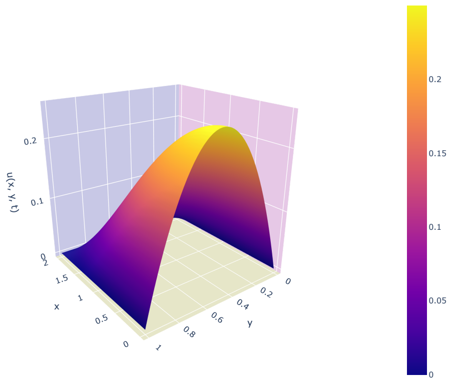
$$\begin{pmatrix} 2 & -1 & 0 & 0 & 0 \\ -3 & 8 & -1 & 0 & 0 \\ 0 & -5 & 12 & 2 & 0 \\ 0 & 0 & -6 & 18 & -4 \\ 0 & 0 & 0 & -5 & 10 \end{pmatrix} \cdot \left(X\right) = \begin{pmatrix} -25 \\ 72 \\ -69 \\ -156 \\ 20 \end{pmatrix} \tag{14}$$



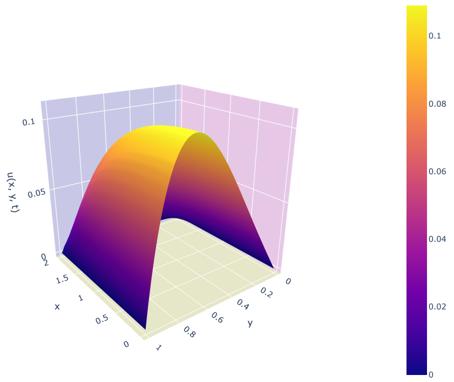
```
coeffs = np.array([
    [2, -1, 0, 0, 0],
    [-3, 8, -1, 0, 0],
    [0, -5, 12, 2, 0],
    [0, 0, -6, 18, -4],
    [0, 0, 0, -5, 10]
])
F = np.array([
    -25, 72, -69, -156, 20
])
x = TDMA(coeffs, F)
x

array([-10.,   5.,  -2., -10.,  -3.])

coeffs @ x - F  # результат ~ 0

array([ 0.00000000e+00,  0.00000000e+00, -1.42108547e-14,  2.84217094e-14,
       -3.55271368e-15])
```

Рис. 1: Демонстрация работы метода прогонки в среде Jupyter Notebook. Видно, что результат разности левой и правой частей уравнения практически равен 0.

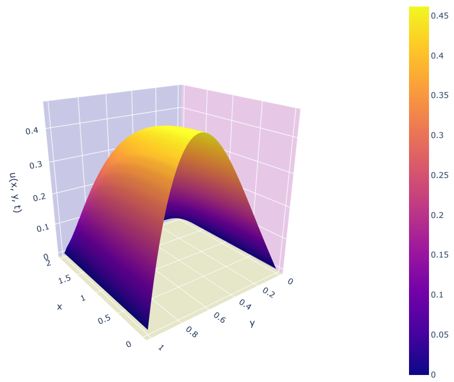
4.2 Результат численного решения



(a) Начальное состояние



(b) Середина времени



(c) Конечное состояние

Рис. 2: Визуализаций эволюции численного решения во времени

Анимация эволюции доступна в приложенном .ipynb файле.

5 Проверка сложности метода переменных направлений

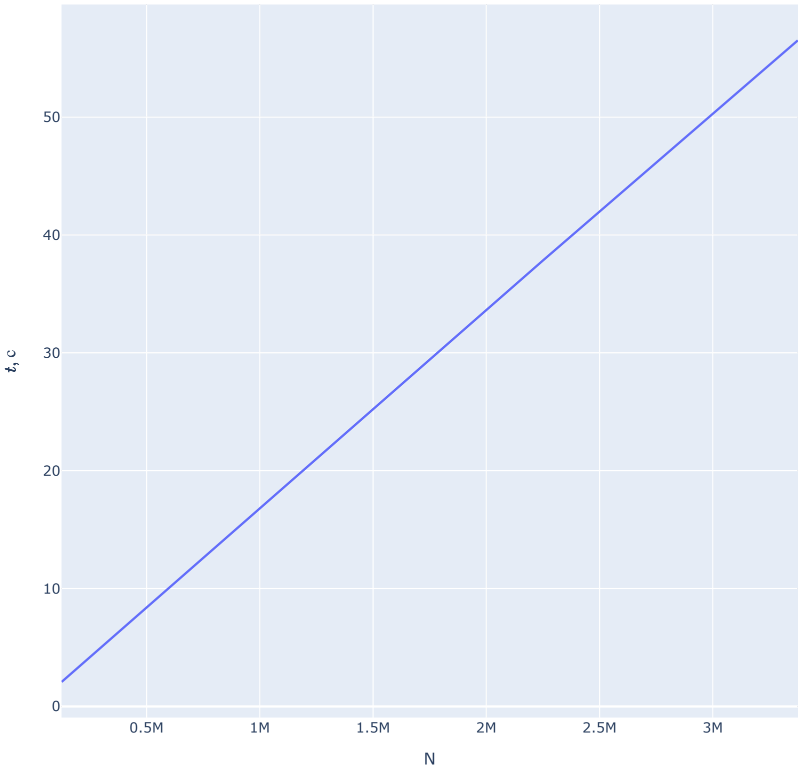


Рис. 3: График зависимости времени вычисления решения от количества узлов сетки. Видно, что зависимость носит линейный характер.

6 Заключение

Было получено численное решение двумерного уравнения тепропроводности. Устойчивость метода подтверждается видом решения. Время вычисления решения пропорционально числу узлов сетки. Таким образом, экспериментально подтверждены теоретические свойства метода переменных направлений.