Курсовая работа по дисциплине Численные методы

Абдурахманов Руслан, Чирков Михаил

22 декабря 2024 г.

Содержание

1. Введение

В данной работе будет рассмотрено решение начально-краевой задачи о колебании двумерной мембраны.

Решение задачи будет получено с помощью численных методов на дискретной сетке.

Помимо полученного решения также будут разработаны способы его визуализации.

Цель работы — изучить поведение колебаний мембраны при различных начальных и краевых условиях, получить численное решение при этих условиях и визуализировать его.

2. Постановка задачи

Рассматривается начально-краевая задача:

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + f(x, y, t) \tag{2.1}$$

$$u(x, y, t) = g(x, y, t), \quad \forall (x, y) \in \partial G = [0, 1] \times [0, 1], \ \forall t \in [a; b] \tag{2.2}$$

$$u(x, y, a) = v_1(x, y), \quad \frac{\partial u}{\partial t}(x, y, a) = v_2(x, y), \quad \forall (x, y) \tag{2.3}$$

Функция u играет роль отклонения мембраны от состояния покоя. Уравнение (2.1) описывает колебательный процесс. Краевые условия (2.3) показывают положение краев мембраны во времени. Если функция g не зависит от времени, то края мембраны жестко зафиксированы. Начальные условия (2.2) задают соответственно положение и скорость мембраны в начальный момент времени.

Необходимо решить задачу (2.1)-(2.3) с использованием явной и неявной схем метода конченых разностей [[1],с.373]. Для указанных методов разработать программу поиска численного решения исходной задачи. Также разработать программу, результатом которой будет являться анимация, демонстрирующая колебание мембраны.

Модельные задачи:

1. $g = 0, \quad v_1 = 0, \quad v_2 = 0, \quad f = e^{-10((x-0.5)^2+(y-0.5)^2)-t}$
2. $g = 0, \quad v_1 = x^{10}(1-x)y(1-y), \quad v_2 = 0, \quad f = 0$
3. $g = 0, \quad v_1 = 0, \quad v_2 = x^{10}(1-x)y(1-y), \quad f = 0$
4. $g(0, y) = y(1-y), \ g(1, y) = 0, \ g(x, 0) = 0, \ g(x, 1) = 0,$
   $v_1 = (1-x)y(1-y)\cos(5\pi x), \quad v_2 = 0, \quad f = 0$
5. $g = \dfrac{5\sin(t)}{t+1}, \quad v_1 = 0, \quad v_2 = 0, \quad f = 0$

3. Построение схем

В [1] полностью описано построение и анализ устойчивости разностных схем для уравнения колебаний с одним пространственным измерением. Будем использовать аналогичный принцип построения схем, который разберем ниже.

Однако сначала для удобства приведем постановку задачи для одного пространственного измерения:

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} + f(x, t) \tag{3.1}$$

$$u(x, t) = g(x, t), \quad \forall x \in \partial G = [0, 1], \ \forall t \in [a; b] \tag{3.2}$$

$$u(x, a) = v_1(x), \quad \frac{\partial u}{\partial t}(x, a) = v_2(x), \quad \forall x \tag{3.3}$$

3.1. Дискретизация

Сначала нам необходимо свести задачу к дискретной, для этого в каждом измерении введем следующие узлы:

1. $t_k = t(a + k\frac{b-a}{N}), \quad k = 0, \ldots, N-1$
2. $x_i = t(i\frac{1}{M}), \quad i = 0, \ldots, M-1$
3. $y_j = t(j\frac{1}{L}), \quad j = 0, \ldots, L-1$

Также введем следующие сеточные функции:

1. $f_{i,j}^k = f(x_i, y_j, t_k)$
2. $u_{i,j}^k$ - аппроксимация $u(x_i, y_j, t_k)$
3. $g_{i,j}^k = g(x_i, y_j, t_k)$
4. $v_{1i,j} = v_1(x_i, y_j)$
5. $v_{2i,j} = v_2(x_i, y_j)$

3.2. Явная схема

В [1] предлагается строить явную схему на основе центрально-разностной аппроксимации частных производных второго порядка.

Построим интересующие нас аппроксимации с использованием обозначений для сеточных функций:

$$\frac{\partial^2 u}{\partial x^2}\Big|_{x=x_i, y=y_j, t=t_k} \approx \frac{u_{i+1,j}^k - 2u_{i,j}^k + u_{i-1,j}^k}{h_x^2} \tag{3.4}$$

$$\frac{\partial^2 u}{\partial y^2}\Big|_{x=x_i, y=y_j, t=t_k} \approx \frac{u_{i,j+1}^k - 2u_{i,j}^k + u_{i,j-1}^k}{h_y^2} \tag{3.5}$$

$$\frac{\partial^2 u}{\partial t^2}\Big|_{x=x_i, y=y_j, t=t_k} \approx \frac{u_{i,j}^{k+1} - 2u_{i,j}^k + u_{i,j}^{k-1}}{h_t^2} \tag{3.6}$$

С использованием данных аппроксимаций и указанных сеточных функций можно перейти к следующей разностной схеме:

$$\frac{u_{i,j}^{k+1} - 2u_{i,j}^k + u_{i,j}^{k-1}}{h_t^2} = \frac{u_{i+1,j}^k - 2u_{i,j}^k + u_{i-1,j}^k}{h_x^2} + \frac{u_{i,j+1}^k - 2u_{i,j}^k + u_{i,j-1}^k}{h_y^2} + f_{i,j}^k \tag{3.7}$$

$$u_{0,j}^k = g_{0,j}^k,\ u_{1,j}^k = g_{1,j}^k,\ u_{i,0}^k = g_{i,0}^k,\ u_{i,1}^k = g_{i,1}^k, \quad i = 0, \ldots, M,\ j = 0, \ldots, L, k = 0, \ldots, N \tag{3.8}$$

$$u_{i,j}^0 = v_{1i,j}, \quad \frac{\partial u}{\partial t}(x_i, y_j, a) = v_{2i,j}, \quad i = 1, \ldots, M-1, \quad j = 1, \ldots, L-1 \tag{3.9}$$

Таким образом осуществлен переход от (2.1)-(2.3) к (3.7)-(3.9).

В источнике [1] аналог подобной схемы назывался схемой "крест", что логично при двух измерениях. В наших трех измерениях схему крестом можно назвать только с натяжкой.

Также в [1] указано, что данная схема имеет второй порядок точности.

Теперь выразим значение сеточной функции на $(k+1)$-ом временном слое через значения на предыдущих слоях

$$u_{i,j}^{k+1} = h_t^2 \Big[\frac{u_{i+1,j}^k - 2u_{i,j}^k + u_{i-1,j}^k}{h_x^2} + \frac{u_{i,j+1}^k - 2u_{i,j}^k + u_{i,j-1}^k}{h_y^2} + f_{i,j}^k \Big] + 2u_{i,j}^k - u_{i,j}^{k+1} \tag{3.10}$$

Отсюда видно, что для переход на следующий временной слой необходимо использовать значения двух последних слоев. Поэтому начального условия v_1 нам не хватит. Для применения схемы необходимо, чтобы значения $u_{i,j}^k$ были известны на первых двух слоях. Значение на втором слое можно аппроксимировать со вторым порядком точности с помощью формулы Тейлора:

$$u(x, y, t + h_t) = u(x, y, t) + h_t \frac{\partial u}{\partial t} + \frac{h_t^2}{2} \frac{\partial^2 u}{\partial t^2} + o(h_t^2)$$

Из этого равенства можно получить следующую аппроксимацию(вторая производная берется из ДУ):

$$u_{i,j}^1 = v_{1i,j} + h_t v_{2i,j} + \frac{h_t^2}{2} \Big[\frac{u_{i+1,j}^k - 2u_{i,j}^k + u_{i-1,j}^k}{h_x^2} + \frac{u_{i,j+1}^k - 2u_{i,j}^k + u_{i,j-1}^k}{h_y^2} + f_{i,j}^k \Big] \tag{3.11}$$

Так как известны значения для первых двух временных слоев, то можно осуществлять переход к последующим слоям с использованием (3.10).На этом построение явной схемы можно считать законченным, поэтому перейдем к построению неявной схемы.

3.3. Неявная схема

Логика аппроксимации производных в неявной схеме немного иная: производная по t аппроксимируется как и раньше, а вот производные по x и y аппроксимируются средним арифметическим центрально-разностной аппроксимации на предшествующем и последующем временных слоях.

$$\frac{\partial^2 u}{\partial x^2}\Big|_{x=x_i, y=y_j, z=z_k} \approx \frac{u_{i+1,j}^{k+1} - 2u_{i,j}^{k+1} + u_{i-1,j}^{k+1}}{2h_x^2} + \frac{u_{i+1,j}^{k-1} - 2u_{i,j}^{k-1} + u_{i-1,j}^{k-1}}{2h_x^2} \tag{3.12}$$

$$\frac{\partial^2 u}{\partial y^2}\Big|_{x=x_i, y=y_j, z=z_k} \approx \frac{u_{i,j+1}^{k+1} - 2u_{i,j}^{k+1} + u_{i,j-1}^{k+1}}{2h_y^2} + \frac{u_{i,j+1}^{k-1} - 2u_{i,j}^{k-1} + u_{i,j-1}^{k-1}}{2h_y^2} \tag{3.13}$$

Теперь можно перейти к дискретной задаче:

$$\frac{u_{i,j}^{k+1}-2u_{i,j}^{k}+u_{i,j}^{k-1}}{h_t^2}=\frac{u_{i+1,j}^{k+1}-2u_{i,j}^{k+1}+u_{i-1,j}^{k+1}}{2h_x^2}+\frac{u_{i+1,j}^{k-1}-2u_{i,j}^{k-1}+u_{i-1,j}^{k-1}}{2h_x^2}$$
$$+\frac{u_{i,j+1}^{k+1}-2u_{i,j}^{k+1}+u_{i,j-1}^{k+1}}{2h_y^2}+\frac{u_{i,j+1}^{k-1}-2u_{i,j}^{k-1}+u_{i,j-1}^{k-1}}{2h_y^2}+f_{i,j}^k \tag{3.14}$$

$$u_{0,j}^k=g_{0,j}^k,\ u_{1,j}^k=g_{1,j}^k,\ u_{i,0}^k=g_{i,0}^k,$$
$$u_{i,1}^k=g_{i,1}^k,\quad i=0,\dots,M,\ j=0,\dots,L,k=0,\dots,N \tag{3.15}$$

$$u_{i,j}^0=v_{1i,j},\quad \frac{\partial u}{\partial t}(x_i,y_j,a)=v_{2i,j},\quad i=1,\dots,M-1,\quad j=1,\dots,L-1 \tag{3.16}$$

Таким образом получили еще один переход от непрерывной (2.1)-(2.3) к дискретной (3.14)-(3.16)

Однако это еще не все, теперь нужно понять, как делать переход между слоями. Для этого выразим значения функции на $(n+1)$-ом слое через значения на предыдущих слоях

$$u_{i,j-1}^{k+1}\left[-\frac{h_t^2}{2h_y^2}\right]+u_{i-1,j}^{k+1}\left[-\frac{h_t^2}{2h_x^2}\right]+u_{i,j}^{k+1}\left[1+\frac{h_t^2}{h_x^2}+\frac{h_t^2}{h_y^2}\right]+u_{i+1,j}^{k+1}\left[-\frac{h_t^2}{2h_x^2}\right]+u_{i,j+1}^{k+1}\left[-\frac{h_t^2}{2h_y^2}\right]=$$
$$h_t^2\left[\frac{u_{i+1,j}^{k-1}-2u_{i,j}^{k-1}+u_{i-1,j}^{k-1}}{2h_x^2}+\frac{u_{i,j+1}^{k-1}-2u_{i,j}^{k-1}+u_{i,j-1}^{k-1}}{2h_y^2}+f_{i,j}^k\right]+2u_{i,j}^k-u_{i,j}^{k-1} \tag{3.17}$$

В данном случае нет явной связи между значением на текущем и предыдущем слоях. Однако можно составить систему из $(M-1)(L-1)$ уравнения и разрешить ее для получения значения функции на целом слое. Такое кол-во уравненй обусловлено тем, что значения на краях искать не надо.

Для составления системы уравнений нужно определиться с системой обхода сетки, от этого зависит как будут расположены уравнения в системе. Предлагается пройти сначала все значения $u_{i,1}^{k+1}$ с возрастанием по $i=1,\dots,M-1$, потом все значения $u_{i,2}^{k+1}$ с возрастанием по $i=1,\dots,M-1$ и так далее до $u_{i,L-1}^{k+1}$ с возрастанием по $i=1,\dots,M-1$. Так мы совершаем некоторый *построчный* обход сетки.

Однако важно отметить некоторые тонкости: когда мы идем по крайним строкам или заходим в крайний столбец, то нужно производить *коррекцию* коэффициентов, так как в уравнения для этих узлов будут входить граничные узлы, значения на которых уже известны из (3.15). Поэтому там уравнения будут запсиывать следующим образом:

$$u_{i,j}^{k+1}\left[1+\frac{h_t^2}{h_x^2}+\frac{h_t^2}{h_y^2}\right]+u_{i+1,j}^{k+1}\left[-\frac{h_t^2}{2h_x^2}\right]+u_{i,j+1}^{k+1}\left[-\frac{h_t^2}{2h_y^2}\right]=$$
$$h_t^2\left[\frac{u_{i+1,j}^{k-1}-2u_{i,j}^{k-1}+u_{i-1,j}^{k-1}}{2h_x^2}+\frac{u_{i,j+1}^{k-1}-2u_{i,j}^{k-1}+u_{i,j-1}^{k-1}}{2h_y^2}+f_{i,j}^k\right]+ \tag{3.18}$$
$$2u_{i,j}^k-u_{i,j}^{k-1}+u_{i,j-1}^{k+1}\left[\frac{h_t^2}{2h_y^2}\right]+u_{i-1,j}^{k+1}\left[\frac{h_t^2}{2h_x^2}\right]$$

$$u^{k+1}_{i-1,j}\left[-\frac{h_t^2}{2h_x^2}\right]+u^{k+1}_{i,j}\left[1+\frac{h_t^2}{h_x^2}+\frac{h_t^2}{h_y^2}\right]+u^{k+1}_{i,j+1}\left[-\frac{h_t^2}{2h_y^2}\right]=$$
$$h_t^2\left[\frac{u^{k-1}_{i+1,j}-2u^{k-1}_{i,j}+u^{k-1}_{i-1,j}}{2h_x^2}+\frac{u^{k-1}_{i,j+1}-2u^{k-1}_{i,j}+u^{k-1}_{i,j-1}}{2h_y^2}+f^k_{i,j}\right]+ \tag{3.19}$$
$$2u^k_{i,j}-u^{k-1}_{i,j}+u^{k+1}_{i,j-1}\left[\frac{h_t^2}{2h_y^2}\right]+u^{k+1}_{i+1,j}\left[\frac{h_t^2}{2h_x^2}\right]$$

$$u^{k+1}_{i,j-1}\left[-\frac{h_t^2}{2h_y^2}\right]+u^{k+1}_{i-1,j}\left[-\frac{h_t^2}{2h_x^2}\right]+u^{k+1}_{i,j}\left[1+\frac{h_t^2}{h_x^2}+\frac{h_t^2}{h_y^2}\right]=$$
$$h_t^2\left[\frac{u^{k-1}_{i+1,j}-2u^{k-1}_{i,j}+u^{k-1}_{i-1,j}}{2h_x^2}+\frac{u^{k-1}_{i,j+1}-2u^{k-1}_{i,j}+u^{k-1}_{i,j-1}}{2h_y^2}+f^k_{i,j}\right]+ \tag{3.20}$$
$$2u^k_{i,j}-u^{k-1}_{i,j}+u^{k+1}_{i,j+1}\left[\frac{h_t^2}{2h_y^2}\right]+u^{k+1}_{i+1,j}\left[\frac{h_t^2}{2h_x^2}\right]$$

$$u^{k+1}_{i,j-1}\left[-\frac{h_t^2}{2h_y^2}\right]+u^{k+1}_{i,j}\left[1+\frac{h_t^2}{h_x^2}+\frac{h_t^2}{h_y^2}\right]+u^{k+1}_{i+1,j}\left[-\frac{h_t^2}{2h_x^2}\right]=$$
$$h_t^2\left[\frac{u^{k-1}_{i+1,j}-2u^{k-1}_{i,j}+u^{k-1}_{i-1,j}}{2h_x^2}+\frac{u^{k-1}_{i,j+1}-2u^{k-1}_{i,j}+u^{k-1}_{i,j-1}}{2h_y^2}+f^k_{i,j}\right]+ \tag{3.21}$$
$$2u^k_{i,j}-u^{k-1}_{i,j}+u^{k+1}_{i,j+1}\left[\frac{h_t^2}{2h_y^2}\right]+u^{k+1}_{i-1,j}\left[\frac{h_t^2}{2h_x^2}\right]$$

$$u^{k+1}_{i-1,j}\left[-\frac{h_t^2}{2h_x^2}\right]+u^{k+1}_{i,j}\left[1+\frac{h_t^2}{h_x^2}+\frac{h_t^2}{h_y^2}\right]+u^{k+1}_{i+1,j}\left[-\frac{h_t^2}{2h_x^2}\right]+u^{k+1}_{i,j+1}\left[-\frac{h_t^2}{2h_y^2}\right]=$$
$$h_t^2\left[\frac{u^{k-1}_{i+1,j}-2u^{k-1}_{i,j}+u^{k-1}_{i-1,j}}{2h_x^2}+\frac{u^{k-1}_{i,j+1}-2u^{k-1}_{i,j}+u^{k-1}_{i,j-1}}{2h_y^2}+f^k_{i,j}\right]+2u^k_{i,j}-u^{k-1}_{i,j}+u^{k+1}_{i,j-1}\left[\frac{h_t^2}{2h_y^2}\right] \tag{3.22}$$

$$u^{k+1}_{i,j-1}\left[-\frac{h_t^2}{2h_y^2}\right]+u^{k+1}_{i-1,j}\left[-\frac{h_t^2}{2h_x^2}\right]+u^{k+1}_{i,j}\left[1+\frac{h_t^2}{h_x^2}+\frac{h_t^2}{h_y^2}\right]+u^{k+1}_{i,j+1}\left[-\frac{h_t^2}{2h_y^2}\right]=$$
$$h_t^2\left[\frac{u^{k-1}_{i+1,j}-2u^{k-1}_{i,j}+u^{k-1}_{i-1,j}}{2h_x^2}+\frac{u^{k-1}_{i,j+1}-2u^{k-1}_{i,j}+u^{k-1}_{i,j-1}}{2h_y^2}+f^k_{i,j}\right]+2u^k_{i,j}-u^{k-1}_{i,j}+u^{k+1}_{i+1,j}\left[\frac{h_t^2}{2h_x^2}\right] \tag{3.23}$$

$$u^{k+1}_{i,j-1}\left[-\frac{h_t^2}{2h_y^2}\right]+u^{k+1}_{i-1,j}\left[-\frac{h_t^2}{2h_x^2}\right]+u^{k+1}_{i,j}\left[1+\frac{h_t^2}{h_x^2}+\frac{h_t^2}{h_y^2}\right]+u^{k+1}_{i+1,j}\left[-\frac{h_t^2}{2h_x^2}\right]=$$
$$h_t^2\left[\frac{u^{k-1}_{i+1,j}-2u^{k-1}_{i,j}+u^{k-1}_{i-1,j}}{2h_x^2}+\frac{u^{k-1}_{i,j+1}-2u^{k-1}_{i,j}+u^{k-1}_{i,j-1}}{2h_y^2}+f^k_{i,j}\right]+2u^k_{i,j}-u^{k-1}_{i,j}+u^{k+1}_{i,j+1}\left[\frac{h_t^2}{2h_y^2}\right] \tag{3.24}$$

$$u^{k+1}_{i,j-1}\left[-\frac{h_t^2}{2h_y^2}\right]+u^{k+1}_{i,j}\left[1+\frac{h_t^2}{h_x^2}+\frac{h_t^2}{h_y^2}\right]+u^{k+1}_{i+1,j}\left[-\frac{h_t^2}{2h_x^2}\right]+u^{k+1}_{i,j+1}\left[-\frac{h_t^2}{2h_y^2}\right]=$$
$$h_t^2\left[\frac{u^{k-1}_{i+1,j}-2u^{k-1}_{i,j}+u^{k-1}_{i-1,j}}{2h_x^2}+\frac{u^{k-1}_{i,j+1}-2u^{k-1}_{i,j}+u^{k-1}_{i,j-1}}{2h_y^2}+f^k_{i,j}\right]+2u^k_{i,j}-u^{k-1}_{i,j}+u^{k+1}_{i-1,j}\left[\frac{h_t^2}{2h_x^2}\right] \tag{3.25}$$

Визуально для узлов это выглядит так:

$$\begin{array}{ccc} (1,1) & \ldots & (1,L-1) \\ \vdots & \vdots & \vdots \\ (M-1,1) & \ldots & (M-1,L-1) \end{array}$$

При таком обходе узлов получим пятидиагональную матрицу(однако не будем ее решать усовершенствованным методом прогонки). В своем программе для решения системы используем метод бисопряженных градиентов из scipy.

В качестве значения на втором временном слое снова будем использовать (3.11)

4. Реализация методов

Для решения задачи реализовано два класса:

1. `PDE` - класс для хранения информации об уравнении и реализации решения с помощью явной и неявной схемы

2. `PDESolutionAnimator` - класс для создания анимаций полученных решений

Ниже приведены сигнатуры классов:

```
class PDESolutionAnimator:
        """
        Класс для анимации решения уравнений в частных производных (УЧП) в различных форматах,
        таких как GIF и MP4 видео. Анимации визуализируют данные решения через 2D и 3D графики,
        используя эффект ``hillshading'' для данных высот, поверхность и сравнение между неявными и явными методами.

        Атрибуты:
        ----------
        PDEname : str
        Название решаемого УЧП, используется для именования выходных файлов.
        ls : LightSource
        Экземпляр класса LightSource, используемый для наложения эффекта hillshading на данные решения.
        cmap : matplotlib.colors.Colormap
        Колор карта, используемая для визуализаций (по умолчанию: серый).
        ve : float
        Коэффициент вертикального увеличения для 3D визуализаций (по умолчанию: 0.05).

        Методы:
        --------
        get\_cmap\_gif\_v1(solution, out\_FPS, gif\_name='default.gif'):
        Создает GIF, показывающий эволюцию решения УЧП с использованием эффекта hillshading в серых тонах.

        get\_cmap\_gif\_v2(solution, out\_FPS, gif\_name='default.gif'):
        Создает GIF, показывающий эволюцию решения УЧП с наложением оттенков в серых тонах.

        get\_surface\_gif\_v1(solution, x, y, out\_FPS, gif\_name='default.gif'):
        Создает 3D GIF, визуализирующий решение УЧП по времени.

        comparation\_gif\_v1(implicit\_solution, explicit\_solution, x, y, t, out\_FPS, gif\_name='default.gif'):
        Создает GIF, сравнивающий неявное и явное решения с использованием 3D рассеяния и 2D изображений.

        comparation\_gif\_v2(implicit\_solution, explicit\_solution, x, y, t, out\_FPS, gif\_name='default.gif'):
        Создает GIF, сравнивающий неявное и явное решения с динамической настройкой диапазона и подписанными заголовками.

        comparation\_mp4(implicit\_solution, explicit\_solution, x, y, t, out\_FPS, mp4\_name='default.mp4'):
        Создает MP4 видео, сравнивающее неявное и явное решения с улучшенной интерактивностью и обновлением кадров.
        """
```

Listing 1: PDESolutionAnimator

```python
class PDE:
        """
        Класс для решения параболических дифференциальных уравнений с помощью явных и неявных численных схем.

        Этот класс предоставляет методы для реализации явной и неявной схемы для решения
        парциальных дифференциальных уравнений (ПДУ) с заданными краевыми условиями и источниками.

        Атрибуты:
        -----------
        f : callable
        Функция источника (или правой части) уравнения ПДУ.
        v1 : callable
        Функция начальных условий для решения.
        v2 : callable
        Функция для граничных условий в уравнении ПДУ.
        g0, g1, g2, g3 : callable
        Граничные условия для каждого из краев.
        x_bord, y_bord, t_bord : tuple
        Границы по осям x, y и времени (t) для сетки.
        explicit_right_part : lambda
        Лямбда-функция, которая вычисляет правую часть явной схемы.
        implicit_right_part : lambda
        Лямбда-функция, которая вычисляет правую часть неявной схемы.
        transition_to_next_layer : lambda
        Лямбда-функция, которая осуществляет переход к следующему слою для явной схемы.
        calc_h : lambda
        Лямбда-функция для вычисления шага по соответствующей оси.
        implicit_b_part : lambda
        Лямбда-функция для вычисления правой части матрицы в неявной схеме.

        Методы:
        --------
        get_mesh(x_cnt, y_cnt, t_cnt):
        Генерирует сетку для пространственно-временной области.
        get_ex_tenzor(x_cnt, y_cnt, t_cnt):
        Инициализирует тензор для явного решения.
        get_im_tenzor(x_cnt, y_cnt, t_cnt):
        Инициализирует тензор для неявного решения.
        get_explicit_solution(x_cnt, y_cnt, t_cnt):
        Вычисляет решение ПДУ с использованием явной схемы.
        create_matrix_DE(n, hx, hy, ht, f):
        Создает разреженную матрицу для неявной схемы.
        get_implicit_solution(x_cnt, y_cnt, t_cnt):
        Вычисляет решение ПДУ с использованием неявной схемы.
        get_dense_mesh(x_cnt, y_cnt):
        Генерирует более плотную сетку для пространственных измерений.
        start_dense_explicit():
        Запускает решение задачи с помощью явной схемы для плотной сетки.
        start_dense_implicit():
        Запускает решение задачи с помощью неявной схемы для плотной сетки.
        """
```

Listing 2: PDE

4.1. Программа для явной схемы

Реализация явной схемы основана на использовании многомерного массива, который будет изменяться цикле по кол-ву временных слоев с помощью перехода между слоями, описанного в (3.10).

Приведем используемые для формирования решения методы и их описания:

```python
    def __init__(self,f,v1,v2,g0,g1,g2,g3,x_bord,y_bord,t_bord):
        """
        Инициализация объекта с параметрами для решения задачи с частными производными.

        Параметры:
        f : callable
        Функция правой части уравнения, которая зависит от тензора, времени и шагов по пространству и времени.
        v1 : float
        Значение для граничного условия по одной из осей (например, скорости).
        v2 : float
        Значение для граничного условия по другой оси (например, скорости).
        g0 : float
        Граничное условие на первой границе в пространстве.
        g1 : float
        Граничное условие на второй границе в пространстве.
        g2 : float
        Граничное условие на третьей границе в пространстве.
        g3 : float
        Граничное условие на четвертой границе в пространстве.
        x_bord : tuple
        Границы по оси x, определяющие диапазон пространственного интервала.
        y_bord : tuple
        Границы по оси y, определяющие диапазон пространственного интервала.
        t_bord : tuple
        Границы по времени, определяющие диапазон временного интервала.

        Атрибуты:
        explicit_right_part : lambda
        Лямбда-функция, вычисляющая правую часть для явной схемы решения.
        implicit_right_part : lambda
        Лямбда-функция, вычисляющая правую часть для неявной схемы решения.
        transition_to_next_layer : lambda
        Лямбда-функция для перехода к следующему слою решения.
        calc_h : lambda
        Лямбда-функция для расчета шага сетки по указанной оси.
        implicit_b_part : lambda
        Лямбда-функция для вычисления правой части для неявной схемы, учитывающая сдвиг по времени.

        """
        self.f = f
        self.v1 = v1
        self.v2 = v2
        self.g0 = g0; self.g1 = g1;  self.g2 = g2; self.g3 = g3;
        self.x_bord = x_bord; self.y_bord = y_bord; self.t_bord = t_bord;
        self.explicit_right_part = lambda tenz,t,h_x,h_y,h_t,f:(tenz[t,2:,1:-1] - 2*tenz[t,1:-1,1:-1] + tenz[t,:-2,1:-1:])/h_x**2 + (tenz[t,1:-1,2:] - 2*tenz[t,1:-1,1:-1] + tenz[t,1:-1,:-2])/h_y**2 + f
        self.implicit_right_part = lambda tenz,t,h_x,h_y,h_t,f:(tenz[t+1,2:,1:-1] - 2*tenz[t+1,1:-1,1:-1] + tenz[t+1,:-2,1:-1])/2/h_x**2 \
        + (tenz[t-1,2:,1:-1] - 2*tenz[t-1,1:-1,1:-1] + tenz[t-1,:-2,1:-1])/2/h_x**2 \
        + (tenz[t+1,1:-1,2:] - 2*tenz[t+1,1:-1,1:-1] + tenz[t+1,1:-1,:-2])/2/h_y**2 \
        + (tenz[t-1,1:-1,2:] - 2*tenz[t-1,1:-1,1:-1] + tenz[t-1,1:-1,:-2])/2/h_y**2 + f
        self.transition_to_next_layer = lambda tenz,t,h_x,h_y,h_t,f: self.explicit_right_part(tenz,t,h_x,h_y,h_t,f)*h_t**2+2*tenz[t,1:-1,1:-1] - tenz[t-1,1:-1,1:-1]
        self.calc_h = lambda cnt, bord: (bord[1]-bord[0])/(cnt-1)
        self.implicit_b_part = lambda tenz,t,h_x,h_y,h_t,f: h_t**2*((tenz[t-1,2:,1:-1] - 2*tenz[t-1,1:-1,1:-1] + tenz[t-1,:-2, 1:-1])/(2*h_x**2) + (tenz[t-1,1:-1,2:] - 2*tenz[t-1,1:-1,1:-1] + tenz[t-1,1:-1,:-2])/(2*h_y**2) + f) + 2*tenz[t,1:-1:,1:-1] - tenz[t-1,1:-1:,1:-1]
```

Listing 3: Инициализация

```python
def get_mesh(self,x_cnt,y_cnt,t_cnt):
        """
        Генерирует трехмерную сетку для пространственно-временной области задачи.

        Параметры:
        x_cnt : int
        Количество узлов по оси x (по пространству).
        y_cnt : int
        Количество узлов по оси y (по пространству).
        t_cnt : int
        Количество узлов по времени.

        Возвращаемое значение:
        tuple
        Кортеж трех массивов, представляющих сетку по времени, пространству x и пространству y.
        Массивы создаются с использованием функции `np.meshgrid`, где оси индексируются как 'ij',
        что означает, что индексы для оси времени (t) идут первыми, затем для оси x, и наконец для оси y.

        Примечания:
        - Метод использует диапазоны, заданные в атрибутах `t_bord`, `x_bord` и `y_bord` для создания сетки.
        - Сетка возвращается с использованием линейных интервалов для каждой оси.

        """
        return np.meshgrid( np.linspace(self.t_bord[0],self.t_bord[1],t_cnt), \
        np.linspace(self.x_bord[0],self.x_bord[1],x_cnt), \
        np.linspace(self.y_bord[0],self.y_bord[1],y_cnt), indexing = 'ij')
```

Listing 4: Получение сетки

```python
def get_explicit_solution(self,x_cnt,y_cnt,t_cnt):
        """
        Вычисляет явное решение задачи в виде тензора значений на сетке.

        Параметры:
        x_cnt : int
        Количество узлов по оси x (по пространству).
        y_cnt : int
        Количество узлов по оси y (по пространству).
        t_cnt : int
        Количество узлов по времени.

        Возвращаемое значение:
        numpy.ndarray
        Тензор размерности (t_cnt, x_cnt, y_cnt), содержащий решение задачи на сетке.
        Значения обновляются на каждом шаге времени с использованием явного метода.

        Примечания:
        - Метод использует функцию `get_mesh` для получения сетки и функцию `calc_h` для вычисления шага по каждой из осей.
        - На первом шаге времени значения в тензоре и на границах задаются с использованием граничных условий (`g0`, `g1`, `g2`, `g3` и начального условия `v1`).
        - Для каждого последующего шага времени вычисляется новое значение на основе предыдущего шага с использованием явного метода и функции правой части уравнения (`explicit_right_part`).
        - Результирующий тензор содержит решение на каждом шаге времени для всех точек сетки.

        """
        self.get_ex_tenzor(x_cnt,y_cnt,t_cnt)
        t_,x_,y_ = self.get_mesh(x_cnt,y_cnt,t_cnt)
        h_x = self.calc_h(x_cnt,self.x_bord)
        h_y = self.calc_h(y_cnt,self.y_bord)
        h_t = self.calc_h(t_cnt,self.t_bord)

        self.ex_tenz [0,:,:] = self.v1(x_[0,:,:],y_[0,:,:])
        self.ex_tenz[:,0,:] = self.g0(x_[:,0,:],y_[:,0,:],t_[:,0,:])
        self.ex_tenz[:,-1,:] = self.g1(x_[:,-1,:],y_[:,-1,:],t_[:,-1,:])
        self.ex_tenz[:,:,0] = self.g2(x_[:,:,0],y_[:,:,0],t_[:,:,0])
        self.ex_tenz[:,:,-1] = self.g3(x_[:,:,-1],y_[:,:,-1],t_[:,:,-1])

        self.ex_tenz[1,1:-1,1:-1] = self.v1(x_[0,1:-1,1:-1],y_[0,1:-1,1:-1]) + h_t*self.v2(x_[0,1:-1,1:-1],y_[0,1:-1,1:-1]) + (h_t**2)/2*self.explicit_right_part(self.ex_tenz,0,h_x,h_y,h_t,f(x_[0,1:-1,1:-1],y_[0,1:-1,1:-1],t_[0,1:-1,1:-1]))

        for i in range(2,t_cnt):
                self.ex_tenz[i,1:-1,1:-1] = self.transition_to_next_layer(self.ex_tenz,i-1,h_x,h_y,h_t,self.f(x_[i-1,1:-1,1:-1],y_[i-1,1:-1,1:-1],t_[i-1,1:-1,1:-1]))

        return self.ex_tenz
```

Listing 5: Получение явного решения

4.2. Программа для неявного решения

Часть функций является для методов общей, например получение сетки. Все отличие неявного метода заключается в составлении системы и разрешении этой системы на каждом шаге цикла.

```python
def create_matrix_DE(self,n,hx,hy,ht,f):
        """
        Создает матрицу системы линейных уравнений (СЛАУ) и правую часть для численного решения уравнений в частных
        ↪  производных.

        Параметры:
        n : int
        Номер временного слоя, для которого вычисляется матрица и правая часть.
        hx : float
        Шаг сетки по оси x.
        hy : float
        Шаг сетки по оси y.
        ht : float
        Шаг сетки по времени.
        f : callable
        Функция правой части уравнения, принимающая координаты x, y и время t.

        Возвращаемое значение:
        tuple
        A_ : scipy.sparse.csc_matrix
        Разреженная матрица коэффициентов СЛАУ (размерностью N x N, где N = m * k).
        b : numpy.ndarray
        Вектор правой части системы (размерностью N).

        Описание:
        - Метод формирует разреженную матрицу A_ для неявного численного метода решения задачи с использованием схемы с
        ↪  центральными разностями.
        - Матрица A_ включает главную диагональ и дополнительные диагонали, соответствующие численной аппроксимации
        ↪  второго порядка для пространственных производных.
        - Вектор b рассчитывается с учетом граничных условий и правой части уравнения. Он представляет собой
        ↪  преобразованное значение функции `implicit_b_part`.

        Примечания:
        - Корректировка элементов матрицы A_ и вектора b осуществляется для учета граничных условий задачи.
        - Возвращаемый вектор b преобразуется в одномерный массив для последующего решения системы.

        """
        m = self.im_tenz.shape[1] - 2
        k = self.im_tenz.shape[2] - 2

        N=m*k
        diagonals = [
        (1 + ht**2/hx**2 + ht**2/hy**2) * np.ones(N),      # главная диагональ (4)
        (-1)*ht**2/(2*hx**2) * np.ones(N - 1), # нижняя диагональ (-1)
        (-1)*ht**2/(2*hx**2) * np.ones(N - 1), # верхняя диагональ (-1)
        (-1)*ht**2/(2*hy**2) * np.ones(N - m), # нижняя диагональ (-m)
        (-1)*ht**2/(2*hy**2)* np.ones(N - m), # верхняя диагональ (+m)
        ]

        diagonals[1][m-1::m] = 0
        diagonals[2][m-1::m] = 0
        diagonals[3][:m+1] = 0
        diagonals[4][-m] = 0

        # Создание разреженной матрицы A
        A_ = csc_matrix(diags(diagonals, offsets=[0, -1, 1,-m,m], shape=(N, N)))




        b = self.implicit_b_part(self.im_tenz,n-1,hx,hy,ht,f)
        #     print('b: ',(b>0).any())
        b[0,0] = b[0,0] + self.im_tenz[n,0,1] * (ht**2)/(2*hx**2) + self.im_tenz[n,1,0] * (ht**2)/(2*hy**2)
        b[0,-1] = b[0,-1] + self.im_tenz[n,0,-2] * (ht**2)/(2*hx**2) + self.im_tenz[n,1,-1] * (ht**2)/(2*hy**2)
        b[-1,0] = b[-1,0] + self.im_tenz[n,-1,1] * (ht**2)/(2*hx**2) + self.im_tenz[n,-2,0] * (ht**2)/(2*hy**2)
        b[-1,-1] = b[-1,-1] + self.im_tenz[n,-1,-2] * (ht**2)/(2*hx**2) + self.im_tenz[n,-2,-1] * (ht**2)/(2*hy**2)
        b[0,1:-1] = b[0,1:-1] + self.im_tenz[n, 0, 2:-2] *  (ht**2)/(2*hx**2)
        b[-1,1:-1] = b[-1,1:-1] + self.im_tenz[n, -1, 2:-2] *  (ht**2)/(2*hx**2)
        b[1:-1,0] = b[1:-1,0] + self.im_tenz[n,2:-2, 0] *  (ht**2)/(2*hy**2)
        b[1:-1,-1] = b[1:-1,-1] + self.im_tenz[n,2:-2, -1] *  (ht**2)/(2*hy**2)

        return A_,b.reshape(m*k)
```

Listing 6: Составление матрицы системы для неявной схемы

```python
def get_implicit_solution(self,x_cnt,y_cnt,t_cnt):
        """
        Вычисляет численное решение задачи с использованием неявного метода.

        Параметры:
        x_cnt : int
        Количество узлов сетки по оси x.
        y_cnt : int
        Количество узлов сетки по оси y.
        t_cnt : int
        Количество временных слоев.

        Возвращаемое значение:
        numpy.ndarray
        Трехмерный тензор `im_tenz` размерностью (t_cnt, x_cnt, y_cnt), содержащий значения решения в узлах сетки.

        Описание:
        - Метод решает задачу с использованием неявной схемы для численного решения уравнений в частных производных.
        - На первом временном слое используется явная схема для вычисления начальных значений.
        - На каждом последующем временном слое формируется система линейных алгебраических уравнений (СЛАУ) с помощью
        ↪  метода `create_matrix_DE`.
        - Решение СЛАУ выполняется с использованием метода бисопряженных градиентов (bicgstab).

        Алгоритм:
        1. Заполняются граничные и начальные условия для тензора `im_tenz`.
        2. Для каждого временного слоя создается матрица системы A и вектор правой части b.
        3. Решается система Ax = b для вычисления значений решения на текущем временном слое.
        4. Проверяется сходимость решения методом `bicgstab`. Если метод не сходится, выводится сообщение с кодом
        ↪  завершения.

        Примечания:
        - Метод выводит промежуточные сообщения каждые 10 итераций для контроля прогресса.
        - Решение возвращается в виде трехмерного тензора, где каждая ось соответствует времени, x и y соответственно.

        """
        self.get_im_tenzor(x_cnt,y_cnt,t_cnt)
        t_,x_,y_ = self.get_mesh(x_cnt,y_cnt,t_cnt)
        h_x = self.calc_h(x_cnt,self.x_bord)
        h_y = self.calc_h(y_cnt,self.y_bord)
        h_t = self.calc_h(t_cnt,self.t_bord)

        self.im_tenz [0,:,:] = self.v1(x_[0,:,:],y_[0,:,:])
        self.im_tenz[:,0,:] = self.g0(x_[:,0,:],y_[:,0,:],t_[:,0,:])
        self.im_tenz[:,-1,:] = self.g1(x_[:,-1,:],y_[:,-1,:],t_[:,-1,:])
        self.im_tenz[:,:,0] = self.g2(x_[:,:,0],y_[:,:,0],t_[:,:,0])
        self.im_tenz[:,:,-1] = self.g3(x_[:,:,-1],y_[:,:,-1],t_[:,:,-1])

        self.im_tenz[1,1:-1,1:-1] = self.v1(x_[0,1:-1,1:-1],y_[0,1:-1,1:-1]) +
        ↪  h_t*self.v2(x_[0,1:-1,1:-1],y_[0,1:-1,1:-1]) +
        ↪  (h_t**2)/2*self.explicit_right_part(self.im_tenz,0,h_x,h_y,h_t,self.f(x_[0,1:-1,1:-1],y_[0,1:-1,1:-1],t_[0,1:-1,1:-1]))

        for i in range(2,t_cnt):
                A,b = self.create_matrix_DE(i,h_x,h_y,h_t,self.f(x_[i,1:-1,1:-1],y_[i,1:-1,1:-1],t_[i,1:-1,1:-1]))
                #      b_s.append(b)
                #      x, info = cg(A, b,tol=1e-12)

                # Используем метод бисопряженных градиентов
                x, info = bicgstab(A, b)
                self.im_tenz[i,1:-1,1:-1] = np.copy(x.reshape((x_cnt-2,y_cnt-2)))
                if info != 0:
                        print("Метод не сошелся. Код завершения:", info)
                if(i%10==0):
                        print('Ура! Уже ', i, ' итераций!')

        return self.im_tenz
```

Listing 7: Получение решения с помощью неявной схемы

4.3. Программа для анимации

```python
def get_cmap_gif_v1(self, solution, out_FPS, gif_name='default.gif'):
    """
    Создает GIF-анимацию на основе заданного 3D-решения задачи.

    Параметры:
    solution : numpy.ndarray
        Тензор решения задачи, где первая ось соответствует времени, а остальные две — пространственным координатам.
    out_FPS : int
        Частота кадров для выходного GIF-файла.
    gif_name : str, optional
        Имя выходного GIF-файла. По умолчанию 'default.gif'.

    Атрибуты:
    PDEname : str
        Название решаемого дифференциального уравнения, используется в имени выходного файла.
    ls : matplotlib.colors.LightSource
        Источник света, используемый для визуализации рельефа решений.
    cmap : matplotlib.colors.Colormap
        Цветовая карта для визуализации.
    ve : float
        Коэффициент вертикального масштабирования для рельефа.

    Описание:
    Создает GIF-анимацию, где каждый кадр представляет hillshade-визуализацию одного слоя тензора решения `solution`.
    Анимация сохраняется с заданной частотой кадров `out_FPS` в файл с именем, включающим `PDEname` и `gif_name`.
    """
    fig = plt.figure()
    ax = fig.add_subplot()
    frames = []
    for p in range(solution.shape[0]):
        img = ax.imshow(self.ls.hillshade(solution[p], vert_exag=self.ve), cmap=self.cmap)
        frames.append([img])
    animation = ArtistAnimation(
        fig,                # фигура, где отображается анимация
        frames,             # кадры
        interval=1000 / out_FPS,  # задержка между кадрами в мс
        blit=True,          # использовать ли двойную буферизацию
        repeat=True)        # зацикливать ли анимацию

    animation.save(self.PDEname + gif_name, writer=ImageMagickWriter(fps=out_FPS))
```

Listing 8: Получение gif анимации для цветовой карты(1)

```python
def get_cmap_gif_v2(self, solution, out_FPS, gif_name='default.gif'):
    """
    Создает GIF-анимацию с обновлением изображения на основе заданного 3D-решения задачи.

    Параметры:
    solution : numpy.ndarray
        Тензор решения задачи, где первая ось соответствует времени, а остальные две — пространственным координатам.
    out_FPS : int
        Частота кадров для выходного GIF-файла.
    gif_name : str, optional
        Имя выходного GIF-файла. По умолчанию 'default.gif'.

    Атрибуты:
    PDEname : str
        Название решаемого дифференциального уравнения, используется в имени выходного файла.
    ls : matplotlib.colors.LightSource
        Источник света, используемый для визуализации рельефа решений.
    cmap : matplotlib.colors.Colormap
        Цветовая карта для визуализации.
    ve : float
        Коэффициент вертикального масштабирования для рельефа.

    Описание:
    Создает GIF-анимацию, где каждый кадр представляет hillshade-визуализацию одного слоя тензора решения `solution`.
    Анимация реализована с помощью `FuncAnimation`, которая обновляет существующий объект `imshow` для повышения
    ↪  производительности.
    Анимация сохраняется с заданной частотой кадров `out_FPS` в файл с именем, включающим `PDEname` и `gif_name`.
    """
    # Создаем hillshade-визуализацию для первого слоя
    Z = self.ls.shade(solution[0], cmap=self.cmap, vert_exag=self.ve, blend_mode='overlay')
    fig, ax = plt.subplots()
    im = ax.imshow(Z)

    # Определяем функцию обновления для каждого кадра
    def update(frame):
        Z = self.ls.shade(solution[frame], cmap=self.cmap, vert_exag=self.ve, blend_mode='overlay')
        im.set_array(Z)  # обновляем значения массива
        return [im]

    # Создаем анимацию
    animation = FuncAnimation(
        fig,
        update,
        frames=solution.shape[0],
        interval=1000 / out_FPS,
        blit=False
    )

    # Сохраняем анимацию
    animation.save(self.PDEname + gif_name, writer=ImageMagickWriter(fps=out_FPS))

```

Listing 9: Получение gif анимации для цветовой карты(2)

```python
def get_surface_gif_v1(self, solution, x, y, out_FPS, gif_name='default.gif'):
    """
    Создает GIF-анимацию с поверхностной 3D-визуализацией решения задачи.

    Параметры:
    solution : numpy.ndarray
        Тензор решения задачи, где первая ось соответствует времени, а оставшиеся — пространственным координатам.
    x : numpy.ndarray
        Координаты по оси X для визуализации.
    y : numpy.ndarray
        Координаты по оси Y для визуализации.
    out_FPS : int
        Частота кадров для выходного GIF-файла.
    gif_name : str, optional
        Имя выходного GIF-файла. По умолчанию 'default.gif'.

    Атрибуты:
    PDEname : str
        Название решаемого дифференциального уравнения, используется в имени выходного файла.

    Описание:
    Создает GIF-анимацию, где каждый кадр представляет 3D-визуализацию решения задачи в виде цветного рассеяния точек.
    Анимация сохраняется с заданной частотой кадров `out_FPS` в файл с именем, включающим `PDEname` и `gif_name`.
    """
    # Создаем фигуру и 3D-ось
    fig = plt.figure()
    ax_3d = fig.add_subplot(projection='3d')
    frames = []

    # Создаем кадры для анимации
    for p in range(solution.shape[0]):
        surface = ax_3d.scatter(
            x, y, solution[p], cmap='inferno', c=solution[p]
        )  # Создаем точечную 3D визуализацию
        frames.append([surface])

    # Создаем анимацию
    animation = ArtistAnimation(
        fig,                # фигура для отображения
        frames,             # кадры
        interval=1000 / out_FPS,  # задержка между кадрами в мс
        blit=True,          # использовать ли двойную буферизацию
        repeat=True         # зацикливать ли анимацию
    )

    # Сохраняем анимацию
    animation.save(self.PDEname + gif_name, writer=ImageMagickWriter(fps=out_FPS))
```

Listing 10: Построение gif анимации колебания мембраны

```python
def comparation_gif_v1(self, implicit_solution, explicit_solution, x, y, t, out_FPS, gif_name='default.gif'):
    """
    Создает GIF-анимацию для сравнения двух решений задачи (явного и неявного) на различных графиках.

    Параметры:
    implicit_solution : numpy.ndarray
        Решение задачи с использованием неявного метода.
    explicit_solution : numpy.ndarray
        Решение задачи с использованием явного метода.
    x : numpy.ndarray
        Координаты по оси X для визуализации.
    y : numpy.ndarray
        Координаты по оси Y для визуализации.
    t : numpy.ndarray
        Время (или другие параметры), используемые для отображения в заголовке.
    out_FPS : int
        Частота кадров для выходного GIF-файла.
    gif_name : str, optional
        Имя выходного GIF-файла. По умолчанию 'default.gif'.

    Атрибуты:
    PDEname : str
        Название решаемого дифференциального уравнения, используется в имени выходного файла.

    Описание:
    Создает GIF-анимацию, на которой сравниваются два метода решения (явный и неявный) на 4 графиках:
    1. 3D-график для неявного решения.
    2. 2D-график для неявного решения.
    3. 3D-график для явного решения.
    4. 2D-график для явного решения.
    Время отображается в заголовке графиков.
    Анимация сохраняется с заданной частотой кадров `out_FPS` в файл с именем, включающим `PDEname` и `gif_name`.
    """
    # Создаем фигуру с несколькими подграфиками
    fig = plt.figure(figsize=(10, 10))
    fig.suptitle(self.PDEname)

    # Создаем подграфики (2D и 3D)
    ax1 = fig.add_subplot(221, projection='3d')
    ax2 = fig.add_subplot(222)
    ax3 = fig.add_subplot(223, projection='3d')
    ax4 = fig.add_subplot(224)

    # Список для кадров анимации
    frames = []

    # Тексты заголовков, отображающие время
    title1 = ax1.text(0.5, 1.05, 0, s=f'Время: {t[0, 0, 0]:.4f}', transform=ax1.transAxes, ha='center', va='bottom')
    title2 = ax3.text(0.5, 1.05, 0, s=f'Время: {t[0, 0, 0]:.4f}', transform=ax3.transAxes, ha='center', va='bottom')

    # Перебор всех временных шагов для отображения анимации
    for p in range(implicit_solution.shape[0]):
        # Создание теневых изображений для решений
        Z_im = self.ls.shade(implicit_solution[p], cmap=self.cmap, vert_exag=self.ve, blend_mode='overlay')
        Z_ex = self.ls.shade(explicit_solution[p], cmap=self.cmap, vert_exag=self.ve, blend_mode='overlay')

        # Отображение 2D изображений
        line2 = ax2.imshow(Z_im, animated=True)
        line4 = ax4.imshow(Z_ex, animated=True)

        # Отображение 3D точек для решений
        line1 = ax1.scatter(x, y, implicit_solution[p], cmap='inferno', c=implicit_solution[p], animated=True)
        line3 = ax3.scatter(x, y, explicit_solution[p], cmap='inferno', c=explicit_solution[p], animated=True)

        # Обновление времени в заголовке
        title1.set_text(f'Время: {t[p, 0, 0]:.4f}')
        title2.set_text(f'Время: {t[p, 0, 0]:.4f}')

        # Добавление всех объектов на текущий кадр
        frames.append([line1, line2, line3, line4, title1, title2])

    # Создание анимации
    animation = ArtistAnimation(
        fig,                # фигура, где отображается анимация
        frames,             # кадры
        interval=1000 / 12, # задержка между кадрами в мс
        blit=False,         # использовать ли двойную буферизацию
        repeat=True         # зацикливать ли анимацию
    )

    # Сохранение анимации в файл
    animation.save(self.PDEname + gif_name, writer=ImageMagickWriter(fps=out_FPS))
```

Listing 11: gif-анимация сравнения методов(1)

```python
def comparation_gif_v2(self, implicit_solution, explicit_solution, x, y, t, out_FPS, gif_name='default.gif'):
    """
    Создает GIF-анимацию для сравнения двух решений задачи (явного и неявного) на различных графиках.
    В отличие от v1, эта версия обновляет значения изображений и графиков с использованием `FuncAnimation`.
    Параметры:
    implicit_solution : numpy.ndarray
        Решение задачи с использованием неявного метода.
    explicit_solution : numpy.ndarray
        Решение задачи с использованием явного метода.
    x : numpy.ndarray
        Координаты по оси X для визуализации.
    y : numpy.ndarray
        Координаты по оси Y для визуализации.
    t : numpy.ndarray
        Время (или другие параметры), используемые для отображения в заголовке.
    out_FPS : int
        Частота кадров для выходного GIF-файла.
    gif_name : str, optional
        Имя выходного GIF-файла. По умолчанию 'default.gif'.
    Атрибуты:
    PDEname : str
        Название решаемого дифференциального уравнения, используется в имени выходного файла.
    Описание:
    Создает GIF-анимацию, на которой сравниваются два метода решения (явный и неявный) на 4 графиках:
    1. 3D-график для неявного решения.
    2. 2D-график для неявного решения.
    3. 3D-график для явного решения.
    4. 2D-график для явного решения.
    Время отображается в заголовке графиков.
    Анимация сохраняется с заданной частотой кадров `out_FPS` в файл с именем, включающим `PDEname` и `gif_name`.
    """
    # Создаем фигуру с несколькими подграфиками
    fig = plt.figure(figsize=(10, 10))
    fig.suptitle(self.PDEname)
    # Создаем подграфики (2D и 3D)
    ax1 = fig.add_subplot(221, projection='3d')
    ax2 = fig.add_subplot(222)
    ax3 = fig.add_subplot(223, projection='3d')
    ax4 = fig.add_subplot(224)
    # Инициализация значений для первого кадра
    Z_im = self.ls.shade(implicit_solution[0], cmap=self.cmap, vert_exag=self.ve, blend_mode='overlay')
    Z_ex = self.ls.shade(explicit_solution[0], cmap=self.cmap, vert_exag=self.ve, blend_mode='overlay')
    # Установка предельных значений для цветовых шкал
    zim_min = implicit_solution.min()
    zim_max = implicit_solution.max()
    zex_min = explicit_solution.min()
    zex_max = explicit_solution.max()
    # Установка пределов для оси Z
    z1_max = np.abs(implicit_solution).max()
    z2_max = np.abs(explicit_solution).max()
    ax1.set_zlim([-2 * z1_max, 2 * z1_max])
    ax3.set_zlim([-2 * z2_max, 2 * z2_max])
    # Отображение начальных значений
    line1 = ax1.scatter(x, y, implicit_solution[0], cmap='inferno', c=implicit_solution[0], vmin=zim_min, vmax=zim_max)
    line2 = ax2.imshow(Z_im, cmap=self.cmap)
    line3 = ax3.scatter(x, y, explicit_solution[0], cmap='inferno', c=explicit_solution[0], vmin=zex_min, vmax=zex_max)
    line4 = ax4.imshow(Z_ex, cmap=self.cmap)
    # Добавление цветовых шкал
    fig.colorbar(line1, ax=ax1, fraction=0.05, pad=0.05)
    fig.colorbar(line3, ax=ax3, fraction=0.05, pad=0.05)
    # Функция обновления кадров
    def update(frame):
        # Удаление старых коллекций
        for collection in ax1.collections:
            collection.remove()
        for collection in ax3.collections:
            collection.remove()
        for collection in ax2.collections:
            collection.remove()
        # Получение новых значений для кадра
        Z_im = self.ls.shade(implicit_solution[frame], cmap=self.cmap, vert_exag=self.ve, blend_mode='overlay')
        Z_ex = self.ls.shade(explicit_solution[frame], cmap=self.cmap, vert_exag=self.ve, blend_mode='overlay')
        # Обновление изображений
        line2.set_array(Z_im)
        line4.set_array(Z_ex)
        line2.set_clim(Z_im.min(), Z_im.max())
        line4.set_clim(Z_ex.min(), Z_ex.max())
        # Обновление точечных графиков
        line1 = ax1.scatter(x, y, implicit_solution[frame], cmap='inferno', c=implicit_solution[frame], vmin=zim_min,
        ↪ vmax=zim_max)
        line3 = ax3.scatter(x, y, explicit_solution[frame], cmap='inferno', c=explicit_solution[frame], vmin=zex_min,
        ↪ vmax=zex_max)
        # Обновление заголовков
        ax2.set_title(f'Неявный метод \n Время: {t[frame, 0, 0]:.4f}')
        ax4.set_title(f'Явный метод \n Время: {t[frame, 0, 0]:.4f}')
        ax1.set_title(f'Неявный метод \n Время: {t[frame, 0, 0]:.4f}')
        ax3.set_title(f'Явный метод \n Время: {t[frame, 0, 0]:.4f}')
        return line1, line2, line3, line4
    # Настройка расстояний между подграфиками
    plt.subplots_adjust(hspace=0.5, wspace=0.75)
    # Создание анимации
    animation = FuncAnimation(fig, update, frames=implicit_solution.shape[0], interval=1000 / 12, blit=True)
    # Сохранение анимации в GIF
    animation.save(self.PDEname + gif_name, writer=ImageMagickWriter(fps=out_FPS))
```

Listing 12: gif-анимация сравнения методов(2)

```python
def comparation_mp4(self, implicit_solution, explicit_solution, x, y, t, out_FPS, mp4_name='default.mp4'):
    """
    Создает видеофайл в формате MP4 для сравнения двух решений задачи (явного и неявного) на различных графиках.
    Параметры:
    implicit_solution : numpy.ndarray
        Решение задачи с использованием неявного метода.
    explicit_solution : numpy.ndarray
        Решение задачи с использованием явного метода.
    x : numpy.ndarray
        Координаты по оси X для визуализации.
    y : numpy.ndarray
        Координаты по оси Y для визуализации.
    t : numpy.ndarray
        Время (или другие параметры), используемые для отображения в заголовке.
    out_FPS : int
        Частота кадров для выходного видеофайла.
    mp4_name : str, optional
        Имя выходного MP4 файла. По умолчанию 'default.mp4'.
    Атрибуты:
    PDEname : str
        Название решаемого дифференциального уравнения, используется в имени выходного файла.
    Описание:
    Создает видеофайл в формате MP4, на котором сравниваются два метода решения (явный и неявный) на 4 графиках:
    1. 3D-график для неявного решения.
    2. 2D-график для неявного решения.
    3. 3D-график для явного решения.
    4. 2D-график для явного решения.
    Время отображается в заголовке графиков.
    Видео сохраняется в файл с заданной частотой кадров `out_FPS` в формате MP4 с именем, включающим `PDEname` и
    ↪  `mp4_name`.
    """
    # Создаем фигуру с несколькими подграфиками
    fig = plt.figure(figsize=(10, 10))
    fig.suptitle(self.PDEname)
    # Создаем подграфики (2D и 3D)
    ax1 = fig.add_subplot(221, projection='3d')
    ax2 = fig.add_subplot(222)
    ax3 = fig.add_subplot(223, projection='3d')
    ax4 = fig.add_subplot(224)
    # Инициализация значений для первого кадра
    Z_im = self.ls.shade(implicit_solution[0], cmap=self.cmap, vert_exag=self.ve, blend_mode='overlay')
    Z_ex = self.ls.shade(explicit_solution[0], cmap=self.cmap, vert_exag=self.ve, blend_mode='overlay')
    # Установка предельных значений для цветовых шкал
    zim_min = implicit_solution.min()
    zim_max = implicit_solution.max()
    zex_min = explicit_solution.min()
    zex_max = explicit_solution.max()
    # Установка пределов для оси Z
    z1_max = np.abs(implicit_solution).max()
    z2_max = np.abs(explicit_solution).max()
    ax1.set_zlim([-2 * z1_max, 2 * z1_max])
    ax3.set_zlim([-2 * z2_max, 2 * z2_max])
    # Отображение начальных значений
    line1 = ax1.scatter(x, y, implicit_solution[0], cmap='inferno', c=implicit_solution[0], vmin=zim_min, vmax=zim_max)
    line2 = ax2.imshow(Z_im, cmap=self.cmap)
    line3 = ax3.scatter(x, y, explicit_solution[0], cmap='inferno', c=explicit_solution[0], vmin=zex_min, vmax=zex_max)
    line4 = ax4.imshow(Z_ex, cmap=self.cmap)
    # Добавление цветовых шкал
    fig.colorbar(line1, ax=ax1, fraction=0.05, pad=0.05)
    fig.colorbar(line3, ax=ax3, fraction=0.05, pad=0.05)
    # Функция обновления кадров
    def update(frame):
        # Удаление старых коллекций
        for collection in ax1.collections:
            collection.remove()
        for collection in ax3.collections:
            collection.remove()
        for collection in ax2.collections:
            collection.remove()
        # Получение новых значений для кадра
        Z_im = self.ls.shade(implicit_solution[frame], cmap=self.cmap, vert_exag=self.ve, blend_mode='overlay')
        Z_ex = self.ls.shade(explicit_solution[frame], cmap=self.cmap, vert_exag=self.ve, blend_mode='overlay')
        # Обновление изображений
        line2.set_array(Z_im)
        line4.set_array(Z_ex)
        line2.set_clim(Z_im.min(), Z_im.max())
        line4.set_clim(Z_ex.min(), Z_ex.max())
        # Обновление точечных графиков
        line1 = ax1.scatter(x, y, implicit_solution[frame], cmap='inferno', c=implicit_solution[frame], vmin=zim_min,
        ↪  vmax=zim_max)
        line3 = ax3.scatter(x, y, explicit_solution[frame], cmap='inferno', c=explicit_solution[frame], vmin=zex_min,
        ↪  vmax=zex_max)
        ax2.set_title(f'Неявный метод \n Время: {t[frame, 0, 0]:.4f}')
        ax4.set_title(f'Явный метод \n Время: {t[frame, 0, 0]:.4f}')
        ax1.set_title(f'Неявный метод \n Время: {t[frame, 0, 0]:.4f}')
        ax3.set_title(f'Явный метод \n Время: {t[frame, 0, 0]:.4f}')
        return line1, line2, line3, line4
    # Настройка расстояний между подграфиками
    plt.subplots_adjust(hspace=0.5, wspace=0.75)
    animation = FuncAnimation(fig, update, frames=implicit_solution.shape[0], interval=150, blit=True)
    FFWriter = FFMpegWriter(fps=out_FPS)
    animation.save(self.PDEname + mp4_name, writer=FFWriter, dpi=200)

```

Listing 13: mp4 видео сравнения методов(2)

5. Решение модельных задач

Будем решать каждую из модельных задач на временных отрезках $[0; 1]$ и $[0; 3]$ для демонстрации различия в работе явного и неявного методов.

5.1. Модель 1

1. $f = e^{(-10((x-0.5)^2+(y-0.5^2)))-t}$
2. $g = 0$
3. $v_1 = 0$
4. $v_2 = 0$

5.2. Модель 2

1. $f = 0$
2. $g = 0$
3. $v_1 = x^1 0(1-x)y(1-y)$
4. $v_2 = 0$

5.3. Модель 3

1. $f = 0$
2. $g = 0$
3. $v_1 = 0$
4. $v_2 = x^1 0(1-x)y(1-y)$

5.4. Модель 4

1. $f = 0$
2. $g(0, y) = y(1{-}y), g(1, y) = 0, g(x, 0) = 0, g(x, 1) = 0$
3. $v_1 = (1{-}x)y(1{-}y)\cos{(5\pi x)}$
4. $v_2 = 0$

5.5. Модель 5

1. $f = 0$
2. $g = \dfrac{5\sin t}{t+1}$
3. $v_1 = 0$
4. $v_2 = 0$

Шаблон кода для решения задачи и получения различных анимаций:

```
# Определение функции f, которая используется в задаче (например, для PDE)
f = lambda x,y,t: np.exp(-10*((x-0.5)**2 + (y-0.5)**2) - t )

# Начальные условия для граничных условий
g0 = lambda x,y,t: np.zeros(x.shape)  # Граничные условия для g0
g1 = lambda x,y,t: np.zeros(x.shape)  # Граничные условия для g1
g2 = lambda x,y,t: np.zeros(x.shape)  # Граничные условия для g2
g3 = lambda x,y,t: np.zeros(x.shape)  # Граничные условия для g3

# Начальные скорости
v_1 = lambda x,y: np.zeros(x.shape)  # Начальная скорость для v_1
v_2 = lambda x,y: np.zeros(x.shape)  # Начальная скорость для v_2

# Задание граничных значений по координатам x, y и времени t
x_bord = np.array([0,1])  # Границы по x
y_bord = np.array([0,1])  # Границы по y
t1_bord = np.array([0,1]) # Границы по времени t1
t2_bord = np.array([0,1]) # Границы по времени t2

# Количество узлов в сетке для x, y и t
x_cnt = 100  # Количество точек по оси x
y_cnt = 100  # Количество точек по оси y
t_cnt = 300  # Количество точек по времени

# Название модели
model_name = 'model_1'

# Описание модели для отображения в титуле (включает формулу для f)
model_title = r'$\begin{array}& g = 0, & v_1 = 0, & v_2 = 0, & f = e^{-10((x-0.5)^2 + (y-0.5)^2)-t}\end{array}$'

# Частота кадров для анимаций
outFPS = 12

# Создание двух объектов класса PDE для решения задачи на двух различных временных интервалах
pde1_1 = PDE(f, v_1, v_2, g0, g1, g2, g3, x_bord, y_bord, t1_bord)
pde1_2 = PDE(f, v_1, v_2, g0, g1, g2, g3, x_bord, y_bord, t2_bord)

# Получение решений для явного и неявного методов для обоих объектов PDE
explicit_solution1_1 = pde1_1.get_explicit_solution(x_cnt, y_cnt, t_cnt)
implicit_solution1_1 = pde1_1.get_implicit_solution(x_cnt, y_cnt, t_cnt)

explicit_solution1_2 = pde1_2.get_explicit_solution(x_cnt, y_cnt, t_cnt)
implicit_solution1_2 = pde1_2.get_implicit_solution(x_cnt, y_cnt, t_cnt)

# Создание объектов аниматора решений для каждой модели
solutionAnimator1_1 = PDESolutionAnimator(model_name+'1', model_title)
solutionAnimator1_2 = PDESolutionAnimator(model_name+'2', model_title)

# Получение сетки координат (x, y, t) для обоих интервалов
t1, x1, y1 = pde1_1.get_mesh(x_cnt, y_cnt, t_cnt)
t2, x2, y2, t2 = pde1_2.get_mesh(x_cnt, y_cnt, t_cnt)

# Генерация GIF для явных решений для обеих моделей с использованием цветовой карты
solutionAnimator1_1.get_cmap_gif_v2(explicit_solution1_1, outFPS, 'cmap1_1_ex.gif')
solutionAnimator1_1.get_cmap_gif_v2(implicit_solution1_1, outFPS, 'cmap1_1_im.gif')
solutionAnimator1_2.get_cmap_gif_v2(explicit_solution1_2, outFPS, 'cmap1_2_ex.gif')
solutionAnimator1_2.get_cmap_gif_v2(implicit_solution1_2, outFPS, 'cmap1_2_im.gif')

# Генерация GIF для поверхностных графиков для обеих моделей
solutionAnimator1_1.get_surface_gif_v1(explicit_solution1_1, x1[0], y1[0], outFPS, 'surface1_1_ex.gif')
solutionAnimator1_1.get_surface_gif_v1(implicit_solution1_1, x1[0], y1[0], outFPS, 'surface1_1_im.gif')
solutionAnimator1_2.get_surface_gif_v1(explicit_solution1_2, x2[0], y2[0], outFPS, 'surface1_2_ex.gif')
solutionAnimator1_2.get_surface_gif_v1(implicit_solution1_2, x2[0], y2[0], outFPS, 'surface1_2_im.gif')

# Генерация MP4 для сравнения решений для обеих моделей
solutionAnimator1_1.comparation_mp4(implicit_solution1_1, explicit_solution1_1, x1[0], y1[0], t1, outFPS, 'comp1_1.mp4')
solutionAnimator1_2.comparation_mp4(implicit_solution1_2, explicit_solution1_2, x2[0], y2[0], t2, outFPS, 'comp1_2.mp4')
```

Listing 14: Шаблон получения решения и модели

6. Заключение

В ходе работы был построен и программно реализован метод решения начально-краевой задачи (2.1)-(2.3).

Также были получены результаты моделирования для некоторых задач, с результатами моделирования в виде `gif`-анимаций и видео, вместе с интерактивными ноутбуками `.ipynb` можно ознакомиться в следующем репозитории: GitHUB

По "ноутбукам" предполагается следующая история версий:

1. `проба.ipynb`
2. `like-clean.ipynb`
3. `ООП.ipynb`

Все вышеизложенное программное решение находится в последнем из них, однако можно обратить внимание и на второй, так как в нем есть пример решения задачи при повышении плотности сетки.

Список литературы

[1] Марчук Г.И. *Методы вычислительной математики.* Издательство Наука, 1977.