Отчет по лабораторной работе №8
по курсу «Численные методы»

Студент группы 8О-406: Киреев А. К.
Работа выполнена: 20.11.2022
Преподаватель: Пивоваров Д.Е.
Отчет сдан:
Итоговая оценка:
Подпись преподавателя:

1. Тема работы

МЕТОД КОНЕЧНЫХ РАЗНОСТЕЙ РЕШЕНИЯ МНОГОМЕРНЫХ ЗАДАЧ МАТЕМАТИЧЕСКОЙ ФИЗИКИ. МЕТОДЫ РАСЩЕПЛЕНИЯ

2. Цель работы

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, y, t). Исследовать зависимость погрешности от сеточных параметров tau, hx, hy.

du/dt = a * (d^2u)/(dx^2) + a * (d^2u)/(dy^2), a > 0,

u(0,y,t) = cos(mu2, y) * exp(-(mu1^2 + mu2^2) * a * t),

u(pi/2,y,t) = 0,

u(x,0,t) = cos(mu1, x) * exp(-(mu1^2 + mu2^2) * a * t),

u(x,pi/2,t) = 0,

u(x, y, 0) = cos(mu1, x) * cos(mu2, y)

Аналитическое решение:
U(x, y, t) = cos(mu1, x) * cos(mu2, y) * exp(-(mu1^2 + mu2^2) * a * t)

Параметры:
1) mu1 = 1, mu2 = 1
2) mu1 = 2, mu2 = 1
3) mu1 = 1, mu2 = 2

3. Ход выполнения работы

В лабораторной реализованы метод переменных направлений и метод дробных шагов, также реализована функция вычисления ошибки. Графики выводятся при помощи библиотеки matplotlib.

Код на Python:

main.py:

```
import numpy as np
import math
import sys
import matplotlib.pyplot as plt

from matplotlib import cm
from typing import Callable, List
from functools import partial

from methods import fractional_steps_method, alternating_directions_methods

sys.path.append(".")


def analytical_solution(a: float,
                        x: float, y: float,
                        t: float,
                        mu1: float, mu2: float) -> float:
   return np.cos(mu1 * x) * np.cos(mu2 * y) * np.exp(-(mu1**2 + mu2**2) * a * t)


def analytical_grid(a: float, x: np.ndarray, y: np.ndarray, t: np.ndarray,
                    afunc: Callable) -> np.ndarray:
   grid: np.ndarray = np.zeros(shape=(len(t), len(y), len(x)))
   for i in range(len(t)):
       for j in range(len(y)):
           for k in range(len(x)):
               grid[i, j, k] = afunc(a, x[k], y[j], t[i])
   return grid


@np.vectorize
def u_yt_initial_0(y: float, t: float,
                   mu1: float, mu2: float) -> float:
   return np.cos(mu2 * y) * np.exp(-(mu1**2 + mu2**2) * a * t)


@np.vectorize
def u_yt_initial_1(y: float, t: float,
                   mu1: float, mu2: float) -> float:
   return 0.0


@np.vectorize
def u_xt_initial_0(x: float, t: float,
                   mu1: float, mu2: float) -> float:
   return np.cos(mu1 * x) * np.exp(-(mu1**2 + mu2**2) * a * t)


@np.vectorize
def u_xt_initial_1(x: float, t: float,
                   mu1: float, mu2: float) -> float:
   return 0.0


@np.vectorize
def u_xy_initial_0(x: float, y: float,
                   mu1: float, mu2: float) -> float:
   return np.cos(mu1 * x) * np.cos(mu2 * y)


def error(numeric: np.ndarray, analytical: np.ndarray) -> np.ndarray:
```

```
    return np.max(np.abs(numeric - analytical))


def draw(numerical1: np.ndarray, label1: str,
         numerical2: np.ndarray, label2: str,
         analytical: np.ndarray,
         x: np.ndarray, y: np.ndarray,
         t_points: List[int], t: np.ndarray):
    fig = plt.figure(figsize=plt.figaspect(0.7))
    xx, yy = np.meshgrid(x, y)

    for i, ti in enumerate(t_points):
        ax = fig.add_subplot(len(t_points), 3, len(t_points) * i + 1, projection='3d')
        plt.title(label1 + f', t = {t[ti]}')
        ax.set_xlabel('x', fontsize=10)
        ax.set_ylabel('y', fontsize=10)
        ax.set_zlabel(f'u[t={t[ti]}]', fontsize=10)
        ax.plot_surface(xx, yy, numerical1[ti], cmap=cm.coolwarm, linewidth=0, antialiased=True)

        ax = fig.add_subplot(len(t_points), 3, len(t_points) * i + 2, projection='3d')
        ax.set_xlabel('x', fontsize=10)
        ax.set_ylabel('y', fontsize=10)
        ax.set_zlabel(f'u[t={t[ti]}]', fontsize=10)
        plt.title(label2 + f', t = {t[ti]}')
        ax.plot_surface(xx, yy, numerical2[ti], cmap=cm.coolwarm, linewidth=0, antialiased=True)

        ax = fig.add_subplot(len(t_points), 3, len(t_points) * i + 3, projection='3d')
        ax.set_xlabel('x', fontsize=10)
        ax.set_ylabel('y', fontsize=10)
        ax.set_zlabel(f'u[t={t[ti]}]', fontsize=10)
        plt.title(f'analytic, t = {t[ti]}')
        ax.plot_surface(xx, yy, analytical[ti], cmap=cm.coolwarm, linewidth=0, antialiased=True)

    plt.show()


if __name__ == "__main__":
    a = float(input("Enter parameter 'a': "))
    hx = float(input("Enter step 'hx': "))
    hy = float(input("Enter step 'hy': "))
    tau = float(input("Enter step 'tau': "))
    t_bound = float(input("Enter time border: "))

    mu = [(1.0, 1.0), (2.0, 1.0), (1.0, 2.0)]

    for mu1, mu2 in mu:
        print(f"mu1 = {mu1}, mu2 = {mu2}\n")
        x: np.ndarray = np.arange(0, mu1 * np.pi / 2.0 + hx / 2.0, step=hx)
        y: np.ndarray = np.arange(0, mu2 * np.pi / 2.0 + hy / 2.0, step=hy)
        t: np.ndarray = np.arange(0, t_bound + tau / 2.0, step=tau)

        kwargs = {
            "u_yt_initial_0": partial(u_yt_initial_0, mu1=mu1, mu2=mu2),
            "u_yt_initial_1": partial(u_yt_initial_1, mu1=mu1, mu2=mu2),
            "u_xt_initial_0": partial(u_xt_initial_0, mu1=mu1, mu2=mu2),
            "u_xt_initial_1": partial(u_xt_initial_1, mu1=mu1, mu2=mu2),
            "u_xy_initial_0": partial(u_xy_initial_0, mu1=mu1, mu2=mu2),
            "a": a,
            "hx": hx,
            "hy": hy,
            "tau": tau,
            "lx": 0.0,
            "rx": mu1 * np.pi / 2.0,
            "ly": 0.0,
            "ry": mu2 * np.pi / 2.0,
            "t_bound": t_bound
        }

        analytical = analytical_grid(a, x, y, t, partial(analytical_solution,
                                                         mu1=mu1, mu2=mu2))

        print("---------------- FSM ----------------")
        sol1 = fractional_steps_method(**kwargs)
        print(np.round(sol1, 2))
        print("\nError: ", error(sol1, analytical))
        print("-------------------------------------\n")
```

```
        print("---------------- ADM ----------------")
        sol2 = alternating_directions_methods(**kwargs)
        print(np.round(sol2, 2))
        print("\nError: ", error(sol2, analytical))
        print("-------------------------------------\n")
        print("------------- ANALYTICAL ------------")
        print(np.round(analytical, 2))

        t_points = [0, len(t) // 2, len(t) - 1]
        draw(sol1, "FSM", sol2, "ADM", analytical, x, y, t_points, t)

        print("=====================================\n\n")
```

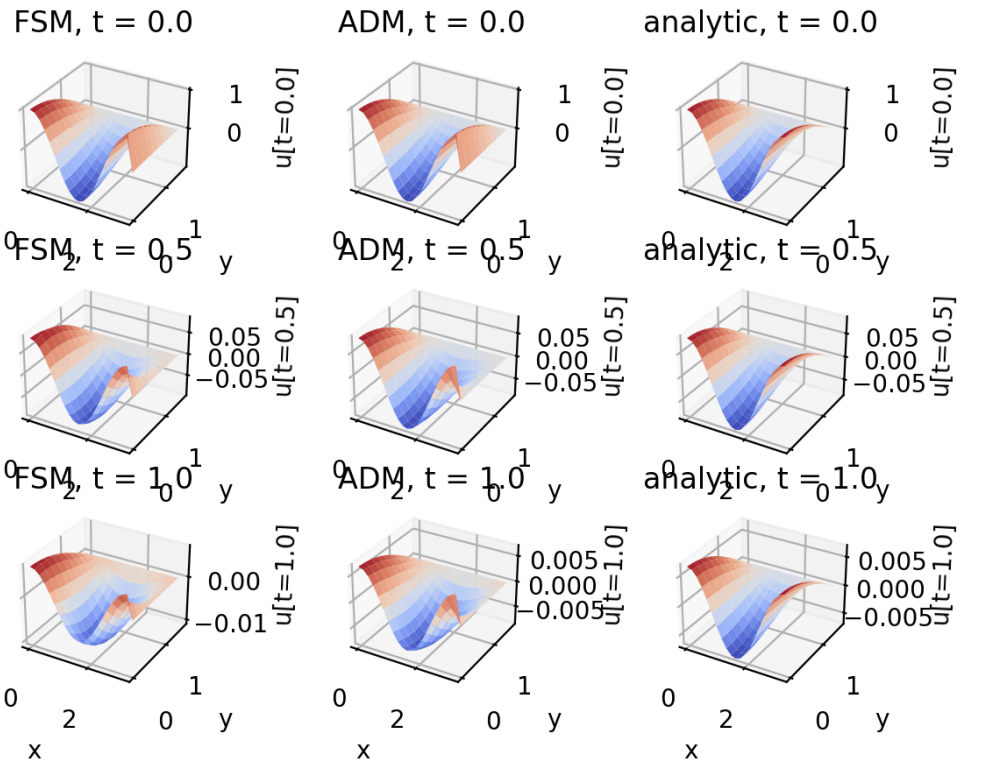
methods.py:

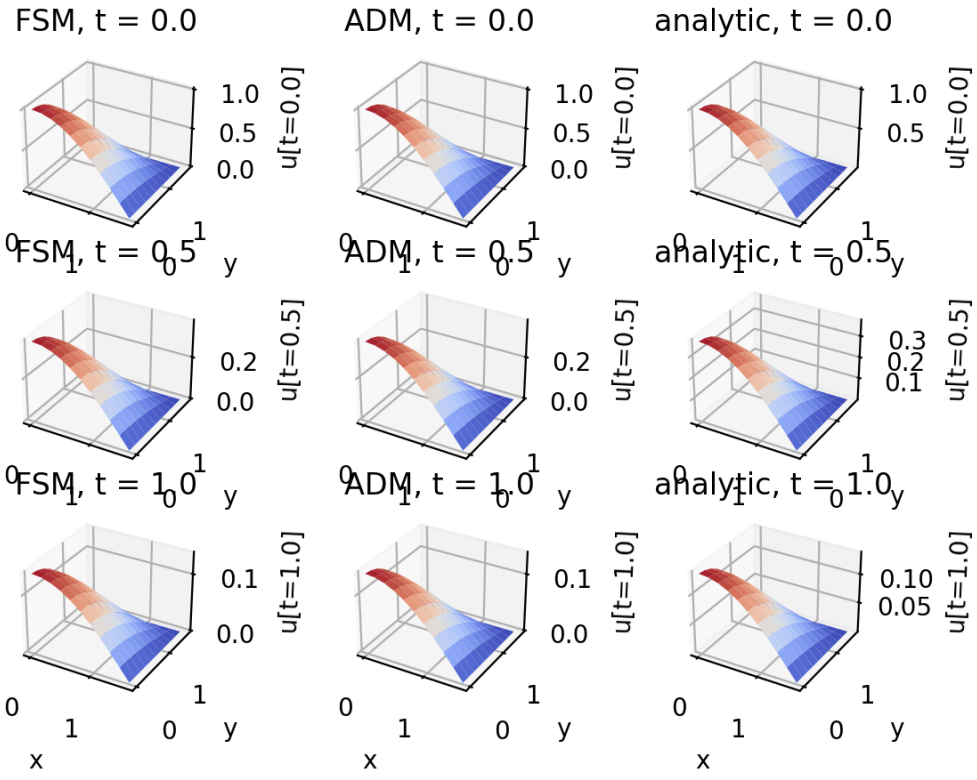
```
import numpy as np
from typing import List, Callable

from sweep import sweep_solve


def common_algo(u_yt_initial_0: Callable, u_yt_initial_1: Callable,
                u_xt_initial_0: Callable, u_xt_initial_1: Callable,
                u_xy_initial_0,
                a: float, hx: float, hy: float, tau: float,
                lx: float, rx: float,
                ly: float, ry: float,
                t_bound: float,
                coef: float) -> np.ndarray:
    x: np.ndarray = np.arange(lx, rx + hx / 2.0, step=hx)
    y: np.ndarray = np.arange(ly, ry + hy / 2.0, step=hy)
    t: np.ndarray = np.arange(0, t_bound + tau / 4.0, step=tau / 2.0)
    u: np.ndarray = np.zeros(shape=(len(t), len(y), len(x)))

    for i, yi in enumerate(y):
        for j, xj in enumerate(x):
            u[0, i, j] = u_xy_initial_0(xj, yi)
    for i, ti in enumerate(t):
        for j, yj in enumerate(y):
            u[i, j, 0] = u_yt_initial_0(yj, ti)
            u[i, j, -1] = u_yt_initial_1(yj, ti)
    for i, ti in enumerate(t):
        for j, xj in enumerate(x):
            u[i, 0, j] = u_xt_initial_0(xj, ti)
            u[i, -1, j] = u_xt_initial_1(xj, ti)

    for k in range(0, len(t) - 2, 2):
        for i in range(1, len(y) - 1):
            matrix: np.ndarray = np.zeros(shape=(len(x) - 2, len(x) - 2))
            matrix[0] += np.array(
                [
                    -(2.0 * a * tau * hy**2 + (1.0 + coef) * hx**2 * hy**2),
                    a * tau * hy**2
                ]
                + [0.0] * (len(matrix) - 2)
            )
            target: List[float] = [-a * tau * hx**2 * coef * u[k, i-1, 1] -
                                   ((1.0 + coef) * hx**2 * hy**2 - 2.0 * a * tau * hx**2 * coef) *
u[k, i, 1] -
                                   a * tau * hx**2 * coef * u[k, i+1, 1] -
                                   a * tau * hy**2 * u[k+1, i, 0]]

            for j in range(1, len(matrix) - 1):
                matrix[j] += np.array(
                    [0.0] * (j - 1)
                    + [
                        a * tau * hy**2,
                        -(2.0 * a * tau * hy**2 + (1.0 + coef) * hx**2 * hy**2),
                        a * tau * hy**2
                    ]
                    + [0.0] * (len(matrix) - j - 2)
                )
                target += [-a * tau * hx**2 * coef * u[k, i-1, j+1] -
                           ((1.0 + coef) * hx**2 * hy**2 - 2.0 * a * tau * hx**2 * coef) * u[k, i,
j+1] -
                           a * tau * hx**2 * coef * u[k, i+1, j+1]]
```

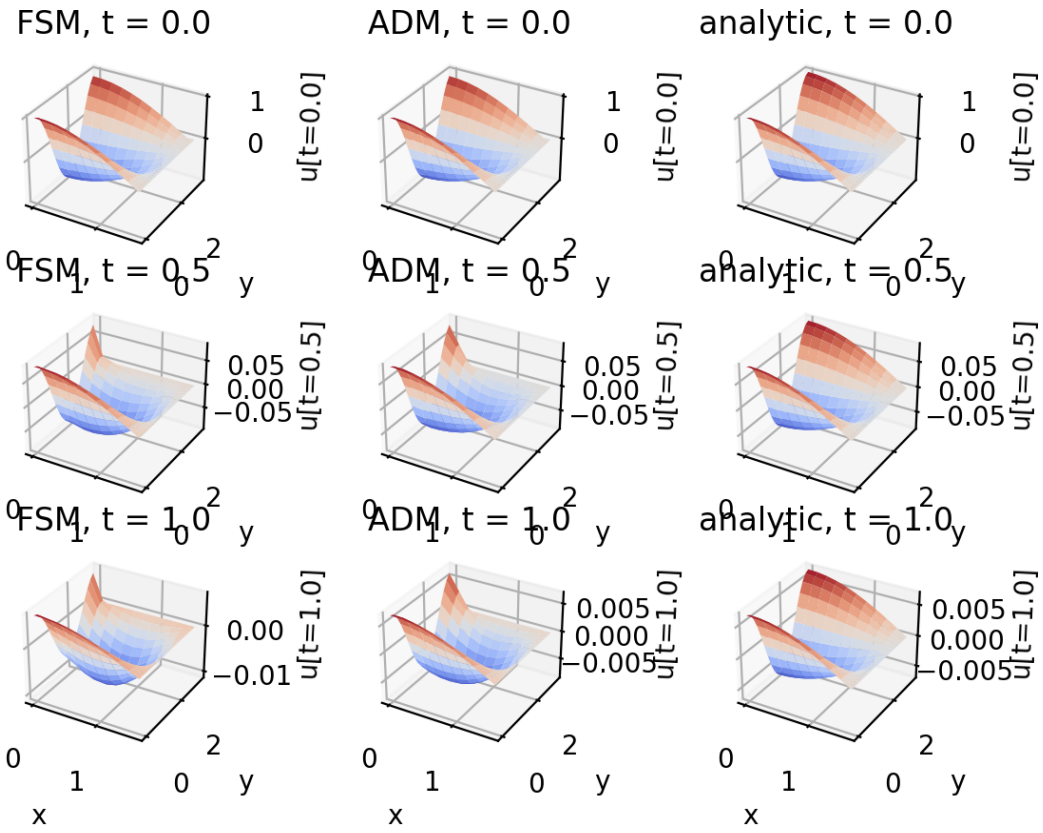
```
            matrix[-1] += np.array(
                [0.0] * (len(matrix) - 2)
                + [
                    a * tau * hy ** 2,
                    -(2.0 * a * tau * hy**2 + (1.0 + coef) * hx**2 * hy**2)
                ]
            )
            target += [-a * tau * hx**2 * coef * u[k, i-1, -2] -
                       ((1.0 + coef) * hx**2 * hy**2 - 2.0 * a * tau * hx**2 * coef) * u[k, i, -2]
-
                       a * tau * hx**2 * coef * u[k, i+1, -2] -
                       a * tau * hy**2 * u[k+1, i, -1]]

            u[k+1, i] += np.array([0.0] + sweep_solve(matrix, np.array(target)).tolist() + [0.0])

        for j in range(1, len(x) - 1):
            matrix: np.ndarray = np.zeros(shape=(len(y) - 2, len(y) - 2))
            matrix[0] += np.array(
                [
                    -(2.0 * a * tau * hx ** 2 + (1.0 + coef) * hx ** 2 * hy ** 2),
                    a * tau * hx ** 2
                ]
                + [0.0] * (len(matrix) - 2)
            )
            target: List[float] = [-a * tau * hy ** 2 * coef * u[k+1, 1, j-1] -
                                   ((1.0 + coef) * hx ** 2 * hy ** 2 - 2.0 * a * tau * hy ** 2 *
coef) * u[k+1, 1, j] -
                                   a * tau * hy ** 2 * coef * u[k+1, 1, j+1] -
                                   a * tau * hx ** 2 * u[k+2, 0, j]]

            for i in range(1, len(matrix) - 1):
                matrix[i] += np.array(
                    [0.0] * (i - 1)
                    + [
                        a * tau * hx ** 2,
                        -(2.0 * a * tau * hx**2 + (1.0 + coef) * hx**2 * hy**2),
                        a * tau * hx ** 2
                    ]
                    + [0.0] * (len(matrix) - i - 2)
                )
                target += [-a * tau * hy ** 2 * coef * u[k+1, i+1, j-1] -
                           ((1.0 + coef) * hx**2 * hy**2 - 2.0 * a * tau * hy**2 * coef) * u[k+1,
i+1, j] -
                           a * tau * hy**2 * coef * u[k+1, i+1, j+1]]

            matrix[-1] += np.array(
                [0.0] * (len(matrix) - 2)
                + [
                    a * tau * hx ** 2,
                    -(2.0 * a * tau * hx**2 + (1.0 + coef) * hx ** 2 * hy ** 2)
                ]
            )
            target += [-a * tau * hy ** 2 * coef * u[k+1, -2, j-1] -
                       ((1.0 + coef) * hx ** 2 * hy ** 2 - 2.0 * a * tau * hy ** 2 * coef) *
u[k+1, -2, j] -
                       a * tau * hy ** 2 * coef * u[k+1, -2, j+1] -
                       a * tau * hx ** 2 * u[k+2, -1, j]]

            u[k+2, :, j] += np.array([0.0] + sweep_solve(matrix, np.array(target)).tolist() +
[0.0])

    return u[::2]


def fractional_steps_method(u_yt_initial_0: Callable, u_yt_initial_1: Callable,
                            u_xt_initial_0: Callable, u_xt_initial_1: Callable,
                            u_xy_initial_0,
                            a: float, hx: float, hy: float, tau: float,
                            lx: float, rx: float,
                            ly: float, ry: float,
                            t_bound: float) -> np.ndarray:
    return common_algo(coef=0.0, **locals())


def alternating_directions_methods(u_yt_initial_0: Callable, u_yt_initial_1: Callable,
```

```
                                u_xt_initial_0: Callable, u_xt_initial_1: Callable,
                                u_xy_initial_0,
                                a: float, hx: float, hy: float, tau: float,
                                lx: float, rx: float,
                                ly: float, ry: float,
                                t_bound: float) -> np.ndarray:
    return common_algo(coef=1.0, **locals())
```

4.Результаты







5. Выводы

В данной лабораторной работе я научился применять методы конечных разностей для решения многомерных задач матфизики.