МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)»

Лабораторные работы по дисциплине

“Численные методы”

Вариант 7

Студент: Ткачев Г.А.

Группа: М8О-402Б-20

Преподаватель: Пивоваров Д.Е.

Москва, 2024

Задание №5.

Используя явную и неявную конечно-разностные схемы, а также схему Кранка - Николсона, решить начально-краевую задачу для дифференциального уравнения параболического типа. Осуществить реализацию трех вариантов аппроксимации граничных условий, содержащих производные: двухточечная аппроксимация с первым порядком, трехточечная аппроксимация со вторым порядком, двухточечная аппроксимация со вторым порядком. В различные моменты времени вычислить погрешность численного решения путем сравнения результатов с приведенным в задании аналитическим решением $U(x,t)$. Исследовать зависимость погрешности от сеточных параметров τ, h.

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + 0{,}5\exp(-0{,}5t)\cos x,$$

$$u_x(0,t) = \exp(-0{,}5t),$$

$$u_x(\pi,t) = -\exp(-0{,}5t),$$

$$u(x,0) = \sin x,$$

Аналитическое решение: $U(x,t) = \exp(-0{,}5t)\sin x$.

Метод решения

Чтобы выполнить данную лабораторную работу, мне пришлось реализовать 4 метода: явный, неявный, аналитический, Кранка-Николсона. Впоследствии были построены графики зависимости U(x) и график зависимости ошибки от времени для наглядности.

Код программы

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

from typing import Dict

def tma(a, b, c, d):
    size = len(a)
    p, q = [], []
    p.append(-c[0] / b[0])
    q.append(d[0] / b[0])
    for i in range(1, size):
        p_tmp = -c[i] / (b[i] + a[i] * p[i - 1])
        q_tmp = (d[i] - a[i] * q[i - 1]) / (b[i] + a[i] * p[i - 1])
        p.append(p_tmp)
        q.append(q_tmp)
    x = [0 for _ in range(size)]
    x[size - 1] = q[size - 1]
    for i in range(size - 2, -1, -1):
        x[i] = p[i] * x[i + 1] + q[i]
    return x

def norm_inf(A):
    n = len(A)
    norm = 0
    for i in range(n):
        sum_ = 0
```

```python
        for j in range(n):
            sum_ += abs(A[i][j])
        norm = sum_ if norm < sum_ else norm
    return norm

def norm_inf_vec(A):
    n = len(A)
    norm = 0
    for i in range(n):
        if abs(A[i]) > norm:
            norm = abs(A[i])
    return norm

def get_zeros(N, K):
    lst = [np.zeros(N) for _ in range(0, 4)]
    lst.append(np.zeros((K, N)))
    return lst

class Data:
    def __init__(self, params):
        self.l = params['l']
        self.f = params['f']
        self.psi = params['psi']
        self.phi0 = params['phi0']
        self.phi1 = params['phi1']
        self.bound_type = params['bound_type']
        self.solve = params['solution']

class ParabolicSolver:
    def __init__(self, params, equation_):
        self.alpha = 1
        self.beta = 0
        self.gamma = 1
        self.delta = 0
        self.h = 0
        self.tau = 0
        self.sigma = 0
        self.data = Data(params)
        self.a = 1
        self.b = 0
        self.c = 0
        try:
            self.solve_aux = getattr(self, f'{equation_}_solver')
        except:
            raise Exception("This type does not exist")

    def solve(self, N, K, T):
        self.h = self.data.l / N
        self.tau = T / K
        self.sigma = self.tau / (self.h ** 2)
        return self.solve_aux(N, K, T)

    def analyticSolve(self, N, K, T):
        self.h = self.data.l / N
        self.tau = T / K
        u = np.zeros((K, N))
        for i in range(K):
            for j in range(N):
                u[i][j] = self.data.solve(j * self.h, i * self.tau)
        return u
```

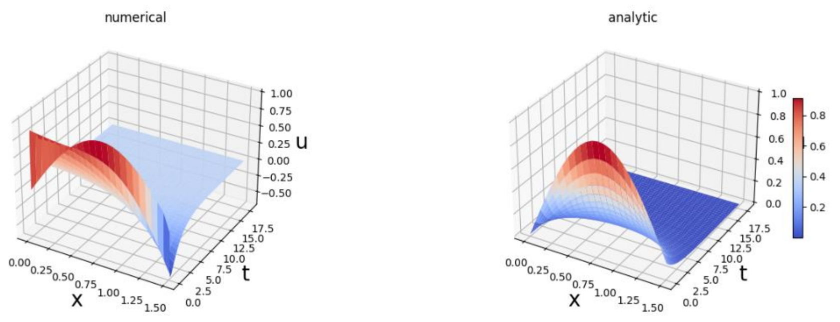
```
    def calculate(self, a, b, c, d, u, k, N, T, K):
        t = np.arange(0, T, T / K)
        for j in range(1, N):
            a[j] = self.sigma
            b[j] = -(1 + 2 * self.sigma)
            c[j] = self.sigma
            d[j] = -u[k - 1][j]
        if self.data.bound_type == 'a1p1':
            a[0] = 0
            b[0] = -(self.alpha / self.h) + self.beta
            c[0] = self.alpha / self.h
            d[0] = self.data.phi0(t[k])
            a[-1] = self.gamma / self.h
            b[-1] = self.gamma / self.h + self.delta
            c[-1] = 0
            d[-1] = self.data.phi1(t[k])
        elif self.data.bound_type == 'a1p2':
            a[0] = 0
            b[0] = -(1 + 2 * self.sigma)
            c[0] = self.sigma
            d[0] = -(u[k - 1][0] + self.sigma * self.data.phi0(k *
self.tau)) - \
                   self.tau * self.data.f(0, k * self.tau)
            a[-1] = self.sigma
            b[-1] = -(1 + 2 * self.sigma)
            c[-1] = 0
            d[-1] = -(u[k - 1][-1] + self.sigma * self.data.phi1(k *
self.tau)) - \
                    self.tau * self.data.f((N - 1) * self.h, k * self.tau)
        elif self.data.bound_type == 'a1p3':
            a[0] = 0
            b[0] = -(1 + 2 * self.sigma)
            c[0] = self.sigma
            d[0] = -((1 - self.sigma) * u[k - 1][1] + self.sigma / 2 * u[k
- 1][0]) - self.tau \
                   * self.data.f(0, k * self.tau) - self.sigma *
self.data.phi0(
                k * self.tau)
            a[-1] = self.sigma
            b[-1] = -(1 + 2 * self.sigma)
            c[-1] = 0
            d[-1] = self.data.phi1(k * self.tau) + self.data.f((N - 1) *
self.h, k * self.tau) \
                    * self.h / (2 * self.tau) * u[k - 1][-1]

    def implicit_solver(self, N, K, T):
        lst = get_zeros(N, K)
        a = lst[0]
        b = lst[1]
        c = lst[2]
        d = lst[3]
        u = lst[4]
        for i in range(1, N - 1):
            u[0][i] = self.data.psi(i * self.h)
        u[0][-1] = 0
        for k in range(1, K):
            self.calculate(a, b, c, d, u, k, N, T, K)
            u[k] = tma(a, b, c, d)
        return u

    def explicit_solver(self, N, K, T):
```

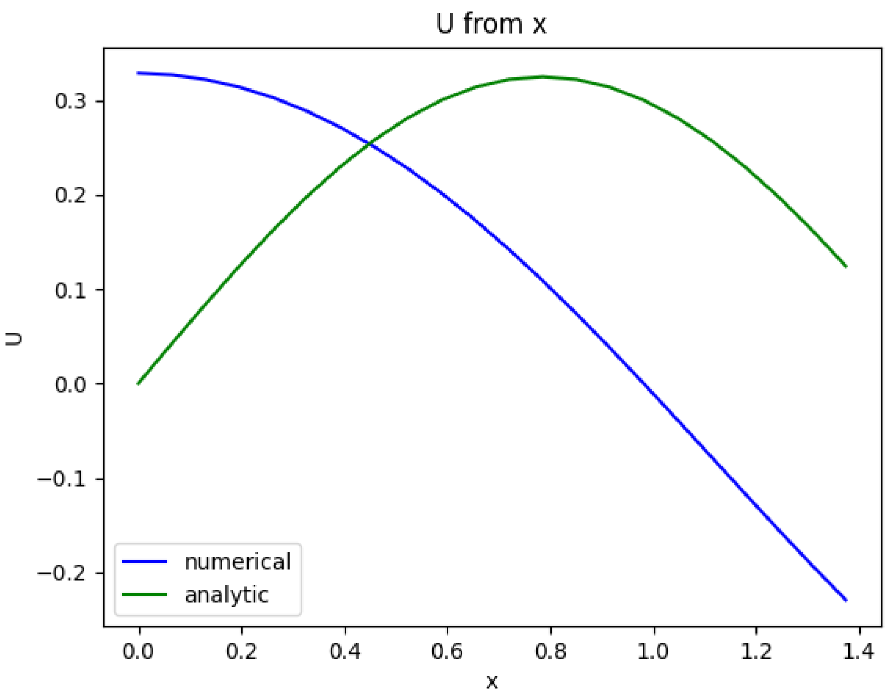
```
        u = np.zeros((K, N))
        t = np.arange(0, T, T / K)
        x = np.arange(0, np.pi / 2, np.pi / 2 / N)
        for j in range(1, N - 1):
            u[0][j] = self.data.psi(j * self.h)
        for k in range(1, K):
            for j in range(1, N - 1):
                u[k][j] = (u[k - 1][j + 1] * (self.a * self.tau / self.h **
2.0 + self.b * self.tau / 2.0 / self.h)
                           + u[k - 1][j] * (-2 * self.a * self.tau / self.h
** 2.0 + self.c * self.tau + 1)
                           + u[k - 1][j - 1] * (self.a * self.tau / self.h
** 2.0 - self.b * self.tau / 2.0 / self.h)
                           + self.tau * self.data.f(x[j], t[k]))
            if self.data.bound_type == 'a1p1':
                u[k][0] = (self.data.phi0(t[k]) - self.alpha / self.h *
u[k][1]) / (self.beta - self.alpha / self.h)
                u[k][-1] = (self.data.phi1(t[k]) + self.gamma / self.h *
u[k][-2]) / (self.delta + self.gamma / self.h)
            elif self.data.bound_type == 'a1p2':
                u[k][0] = (((2.0 * self.alpha * self.a / self.h / (2.0 *
self.a - self.h * self.b)) * u[k][1] +
                            (self.alpha * self.h / self.tau / (2.0 * self.a
- self.h * self.b)) * u[k - 1][0] +
                            (self.alpha * self.h / (2.0 * self.a - self.h *
self.b)) * self.data.f(0, t[k]) -
                            self.data.phi0(t[k]) /
                            ((2.0 * self.alpha * self.a / self.h / (2.0 *
self.a - self.h * self.b)) + (
                                    self.alpha * self.h / self.tau / (2.0 *
self.a - self.h * self.b)) -
                             (self.alpha * self.h / (2.0 * self.a - self.h
* self.b)) * self.c - self.beta))
                u[k][-1] = (((2.0 * self.gamma * self.a / self.h / (2.0 *
self.a + self.h * self.b)) * u[k][-2] +
                             (self.gamma * self.h / self.tau / (2.0 *
self.a + self.h * self.b)) * u[k - 1][-1] +
                             (self.gamma * self.h * self.c / (2.0 * self.a
+ self.h * self.b)) * self.data.f(
                            self.data.l, t[k]) + self.data.phi1(t[k])) / (
                                    (2.0 * self.gamma * self.a / self.h /
(2.0 * self.a + self.h * self.b)) + (
                                    self.gamma * self.h / self.tau / (2.0 *
self.a + self.h * self.b)) - (
                                            self.gamma * self.h * self.c /
(
                                            2.0 * self.a + self.h *
self.b)) * self.c + self.delta))
            elif self.data.bound_type == 'a1p3':
                u[k][-1] = (self.data.phi1(k * self.tau) + u[k][-2] /
self.h + 2 * self.tau * u[k - 1][-1] / self.h) / \
                           (1 / self.h + 2 * self.tau / self.h)
        return u

    def crank_nicolson_solver(self, N, K, T):
        lst = get_zeros(N, K)
        a = lst[0]
        b = lst[1]
        c = lst[2]
        d = lst[3]
        u = lst[4]
```

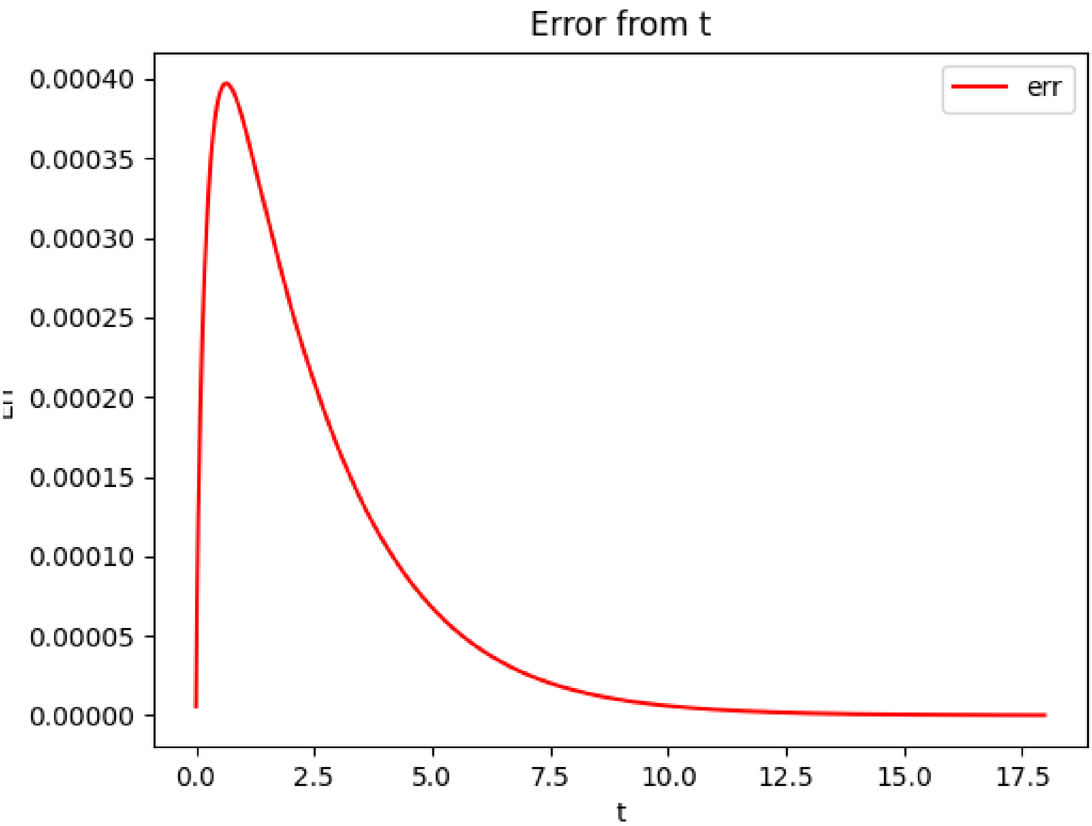
```
        theta = 0.5
        for i in range(1, N - 1):
            u[0][i] = self.data.psi(i * self.h)
        for k in range(1, K):
            self.calculate(a, b, c, d, u, k, N, T, K)
            tmp_imp = tma(a, b, c, d)
            tmp_exp = np.zeros(N)
            tmp_exp[0] = self.data.phi0(self.tau)
            for j in range(1, N - 1):
                tmp_exp[j] = self.sigma * u[k - 1][j + 1] + (1 - 2 *
self.sigma) * u[k - 1][j] + \
                             self.sigma * u[k - 1][j - 1] + self.tau *
self.data.f(j * self.h, k * self.tau)
            tmp_exp[-1] = self.data.phi1(self.tau)
            for j in range(N):
                u[k][j] = theta * tmp_imp[j] + (1 - theta) * tmp_exp[j]
        return u

def draw(my_dict: Dict, N, K, T, save_file="plot.png"):
    fig = plt.figure(figsize=plt.figaspect(0.7))
    z1 = np.array(my_dict['numerical'])
    z2 = np.array(my_dict['analytic'])
    x = np.arange(0, np.pi / 2, np.pi / 2 / N)
    t = np.arange(0, T, T / K)
    x, t = np.meshgrid(x, t)
    ax = fig.add_subplot(1, 2, 1, projection='3d')
    plt.title('numerical')
    ax.set_xlabel('x', fontsize=20)
    ax.set_ylabel('t', fontsize=20)
    ax.set_zlabel('u', fontsize=20)
    ax.plot_surface(x, t, z1, cmap=cm.coolwarm,
                    linewidth=0, antialiased=True)
    ax = fig.add_subplot(1, 2, 2, projection='3d')
    ax.set_xlabel('x', fontsize=20)
    ax.set_ylabel('t', fontsize=20)
    ax.set_zlabel('u', fontsize=20)
    plt.title('analytic')
    surf = ax.plot_surface(x, t, z2, cmap=cm.coolwarm,
                           linewidth=0, antialiased=True)
    fig.colorbar(surf, shrink=0.5, aspect=15)
    plt.savefig(save_file)
    plt.show()

def compare_error(my_dict: Dict):
    error = [[abs(i - j) for i, j in zip(x, y)] for x, y in
zip(my_dict['numerical'], my_dict['analytic'])]
    return error

data = {'equation_type': 'implicit', 'N': 25, 'K': 100, 'T': 1}

if __name__ == '__main__':
    equation_type = data['equation_type']
    N, K, T = int(data['N']), int(data['K']), int(data['T'])
    params = {
        'l': np.pi,
        'psi': lambda x: np.sin(x),
        'f': lambda x, t: 0.5 * np.exp(-0.5 * t) * np.cos(x),
        'phi0': lambda t: np.exp(-0.5 * t),
        'phi1': lambda t: -np.exp(-0.5 * t),
        'solution': lambda x, t: np.exp(-0.5 * t) * np.sin(x),
        'bound_type': 'a1p1',
```

```
    }
    var7 = ParabolicSolver(params, equation_type)
    ans = {
        'numerical': var7.solve(N, K, T).tolist(),
        'analytic': var7.analyticSolve(N, K, T).tolist()
    }
    draw(ans, N, K, T)
    error = compare_error(ans)
    avg_err = 0.0
    for i in error:
        for j in i:
            avg_err += j
        avg_err /= N

    print(error[0])
    print(error[int(K / 2)])
    print(error[-1])
    print(f'Средняя ошибка в каждом N: {avg_err}')
    print(f'Средняя погрешность\t\t    : {avg_err / K}')
```

Результаты работы







Вывод по лабораторной работе

В ходе этой лабораторной работы я углубил свои познания в численных методах для решения параболических дифференциальных уравнений. Исследовались разные подходы к решению начально-краевых задач для уравнений такого типа, в том числе метод Кранка-Николсона, а также неявные и явные конечно-разностные методы, дополнительно применялось аналитическое решение. Проведенные эксперименты дали возможность оценить точность и эффективность каждого из рассмотренных методов.

Задание № 6

Используя явную схему крест и неявную схему, решить начально-краевую задачу для дифференциального уравнения гиперболического типа. Аппроксимацию второго начального условия произвести с первым и со вторым порядком. Осуществить реализацию трех вариантов аппроксимации граничных условий, содержащих производные: двухточечная аппроксимация с первым порядком, трехточечная аппроксимация со вторым порядком, двухточечная аппроксимация со вторым порядком. В различные моменты времени вычислить погрешность численного решения путем сравнения результатов с приведенным в задании аналитическим решением $U(x,t)$. Исследовать зависимость погрешности от сеточных параметров τ, h .

$$\frac{\partial^2 u}{\partial t^2} + 2\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + 2\frac{\partial u}{\partial x} - 3u,$$

$$u(0,t) = \exp(-t)\cos(2t),$$

$$u(\frac{\pi}{2},t) = 0,$$

$$u(x,0) = \exp(-x)\cos x,$$

$$u_t(x,0) = -\exp(-x)\cos x.$$

Аналитическое решение: $U(x,t) = \exp(-t-x)\cos x\cos(2t)$

Метод решения

Чтобы выполнить данную лабораторную работу, мне пришлось решить ДУ гиперболического типа с тремя вариантами аппроксимации граничных условий, используя явную схему крест и неявную схему.

Код программы

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

args = {
    'aConst': 1,
    'bConst': 2,
    'cConst': -3,
    'dConst': 2,
    'l': np.pi / 2,
    'Function': lambda: 0,
    'alpha': 1,
    'beta': 0,
    'gamma': 1,
    'delta': 0,
    'Psi1': lambda x: np.exp(-x) * np.cos(x),
    'Psi2': lambda x: -np.exp(-x) * np.cos(x),
    'Psi11': lambda x: -np.exp(-x) * np.sin(x) - np.exp(-x) * np.cos(x),
    'Psi12': lambda x: 2 * np.exp(-x) * np.sin(x),
    'Phi0': lambda t: np.exp(-t) * np.cos(2 * t),
    'PhiL': lambda t: 0,
    'type': '1-2',
    'accuracyLevl': 1,
    'AnaliticalSolution': lambda x, t: np.exp(-t - x) * np.cos(x) *
np.cos(2 * t),
    'algorithm': 'Implicit'
}

class HyperbolicSolver:
```

```
    def __init__(self, args):
        for name, value in args.items():
            setattr(self, name, value)
        first = args['algorithm'][0].upper()
        word = args['algorithm'][1:].lower()
        functionName = first + word
        try:
            self.FunctionName = getattr(self, functionName)
        except:
            raise Exception("Данный тип не поддерживается, выберите
Implicit или Explicit")

    def Solve(self, N, K, T):
        self.h = self.l / N;
        self.tau = T / K;
        self.sigma = (self.tau ** 2) / (self.h ** 2)
        return self.FunctionName(N, K, T)

    def AnaliticalSolutionMatrix(self, N, K, T):
        self.h = self.l / N;
        self.tau = T / K;
        self.sigma = (self.tau ** 2) / (self.h ** 2)
        self.u = np.zeros((K, N))
        for k in range(K):
            for j in range(N):
                self.u[k][j] = self.AnaliticalSolution(j * self.h, k *
self.tau)
        return self.u

    def RunThroughMethod(self):
        size = len(self.a)
        p = np.zeros(size)
        q = np.zeros(size)
        p[0] = (-self.c[0] / self.b[0])
        q[0] = (self.d[0] / self.b[0])
        for i in range(1, size):
            p[i] = -self.c[i] / (self.b[i] + self.a[i] * p[i - 1])
            q[i] = (self.d[i] - self.a[i] * q[i - 1]) / (self.b[i] +
self.a[i] * p[i - 1])
        x = np.zeros(size)
        x[-1] = q[-1]
        for i in range(size - 2, -1, -1):
            x[i] = p[i] * x[i + 1] + q[i]
        return x

    def Implicit(self, N, K, T):
        self.u = np.zeros((K, N))
        for j in range(N):
            xCurrent = j * self.h
            self.u[0][j] = self.Psi1(xCurrent)
            if self.accuracyLevl == 1:
                self.u[1][j] = self.Psi1(xCurrent) + self.Psi2(xCurrent) *
self.tau + self.Psi12(
                    xCurrent) * self.tau ** 2 / 2
            elif self.accuracyLevl == 2:
                k = self.tau ** 2 / 2
                self.u[1][j] = (1 + self.cConst * k) * self.Psi2(xCurrent)
+ self.aConst * k * self.Psi12(
                    xCurrent) + self.bConst * k * self.Psi11(xCurrent) +
(self.tau - self.dConst * k) * self.Psi1(
                    xCurrent) + k * self.Function()
```

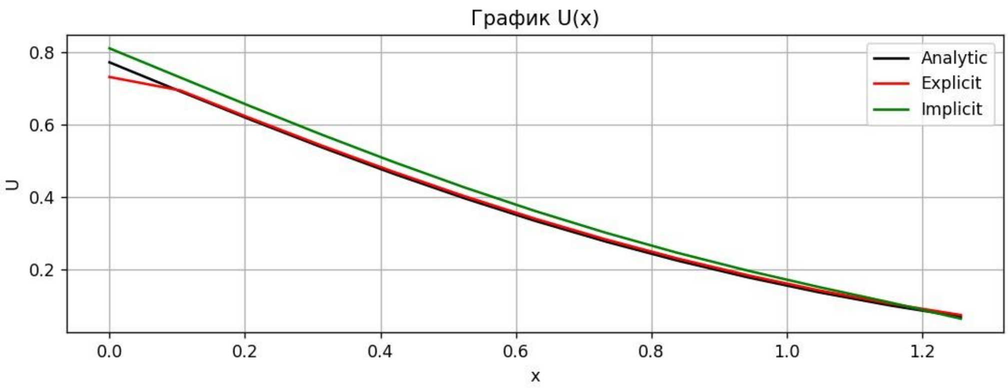
```
        self.a = np.zeros(N)
        self.b = np.zeros(N)
        self.c = np.zeros(N)
        self.d = np.zeros(N)
        for k in range(2, K):
            for j in range(1, N - 1):
                self.a[j] = self.sigma
                self.b[j] = -(1 + 2 * self.sigma)
                self.c[j] = self.sigma
                self.d[j] = -2 * self.u[k - 1][j] + self.u[k - 2][j]
            if self.type == '1-2':
                self.b[0] = self.alpha / self.h / (self.beta - self.alpha /
self.h)
                self.c[0] = 1
                self.d[0] = 1 / (self.beta - self.alpha / self.h) *
self.Phi0(k * self.tau)
                self.a[-1] = -self.gamma / self.h / (self.delta +
self.gamma / self.h)
                self.d[-1] = 1 / (self.delta + self.gamma / self.h) *
self.PhiL(k * self.tau)
            elif self.type == '2-2':
                self.b[0] = 2 * self.aConst / self.h
                self.c[
                    0] = -2 * self.aConst / self.h + self.h / self.tau ** 2
- self.cConst * self.h + -self.dConst * self.h / (
                            2 * self.tau) + self.beta / self.alpha * (2 *
self.aConst + self.bConst * self.h)
                self.d[0] = self.h / self.tau ** 2 * (self.u[k - 2][0] - 2
* self.u[k - 1][
                    0]) - self.h * self.Function() + -self.dConst * self.h
/ (2 * self.tau) * self.u[k - 2][0] + (
                                        2 * self.aConst - self.bConst *
self.h) / self.alpha * self.Phi0(k * self.tau)
                self.a[-1] = -self.b[0]
                self.d[-1] = self.h / self.tau ** 2 * (-self.u[k - 2][0] +
2 * self.u[k - 1][
                    0]) + self.h * self.Function() + self.dConst * self.h /
(2 * self.tau) * self.u[k - 2][0] + (
                                         2 * self.aConst + self.bConst *
self.h) / self.alpha * self.PhiL(k * self.tau)
            elif self.type == '2-3':
                k1 = 2 * self.h * self.beta - 3 * self.alpha
                k2 = 2 * self.h * self.delt + 3 * self.gamma
                omega = self.tau ** 2 * self.bConst / (2 * self.h)
                xi = self.dConst * self.tau / 2
                self.b[0] = 4 * self.alpha - self.alpha / (self.sigma +
omega) * (
                            1 + xi + 2 * self.sigma - self.cConst *
self.tau ** 2)
                self.c[0] = k1 - self.alpha * (omega - self.sigma) / (omega
+ self.sigma)
                self.d[0] = 2 * self.h * self.Phi0(k * self.tau) +
self.alpha * self.d[1] / (-self.sigma - omega)
                self.a[-1] = -self.gamma / (omega - self.sigma) * (
                            1 + xi + 2 * self.sigma - self.cConst *
self.tau ** 2) - 4 * self.gamma
                self.d[-1] = 2 * self.h * self.PhiL(k * self.tau) -
self.gamma * self.d[-2] / (omega - self.sigma)
            self.u[k] = self.RunThroughMethod()
        return self.u
```

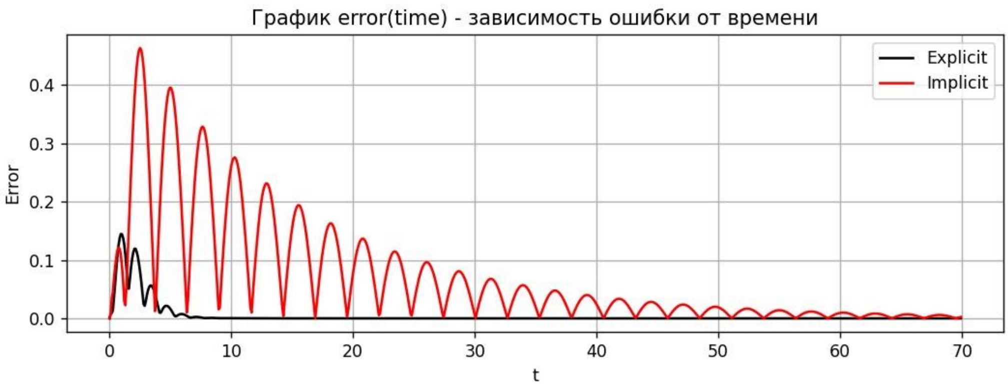
```
    def Explicit(self, N, K, T):
        self.u = np.zeros((K, N))
        for j in range(N):
            xCurrent = j * self.h
            self.u[0][j] = self.Psi1(xCurrent)
            if self.accuracyLevl == 1:
                self.u[1][j] = self.Psi1(xCurrent) + self.Psi2(xCurrent) *
self.tau + self.Psi12(
                    xCurrent) * self.tau ** 2 / 2
            elif self.accuracyLevl == 2:
                k = self.tau ** 2 / 2
                self.u[1][j] = (1 + self.cConst * k) * self.Psi2(xCurrent)
+ self.aConst * k * self.Psi12(
                    xCurrent) + self.bConst * k * self.Psi11(xCurrent) +
(self.tau - self.dConst * k) * self.Psi1(
                    xCurrent) + k * self.Function()
        if self.type == '1-2':
            LBound = self.LBound12
            RBound = self.RBound12
        elif self.type == '2-2':
            LBound = self.LBound22
            RBound = self.RBound22
        elif self.type == '2-3':
            LBound = self.LBound23
            RBound = self.RBound23
        for k in range(2, K):
            t = k * self.tau
            for j in range(1, N - 1):
                self.u[k][j] = self.u[k - 1][j + 1] * (self.sigma +
self.bConst * self.tau ** 2 / (2 * self.h)) + \
                               self.u[k - 1][j] * (-2 * self.sigma + 2 +
self.cConst * self.tau ** 2) + self.u[k - 1][
                                   j - 1] * (self.sigma - self.bConst *
self.tau ** 2 / (2 * self.h)) - self.u[k - 2][
                                   j] + self.tau ** 2 * self.Function()
            self.u[k][0] = LBound(k, t)
            self.u[k][-1] = RBound(k, t)
        return self.u

    def LBound12(self, k, t):
        return -(self.alpha / self.h) / (self.beta - self.alpha / self.h) *
self.u[k - 1][1] + self.Phi0(t) / (
                    self.beta - self.alpha / self.h)

    def RBound12(self, k, t):
        return (self.gamma / self.h) / (self.delta + self.gamma / self.h) *
self.u[k - 1][-2] + self.PhiL(t) / (
                    self.delta + self.gamma / self.h)

    def LBound22(self, k, t):
        n = self.cConst * self.h - 2 * self.aConst / self.h - self.h /
self.tau ** 2 - self.dConst * self.h / (
                    2 * self.tau) + self.beta / self.alpha * (2 *
self.aConst - self.bConst * self.h)
        return 1 / n * (- 2 * self.aConst / self.h * self.u[k][1] + self.h
/ self.tau ** 2 * (
                    self.u[k - 2][0] - 2 * self.u[k - 1][0]) + -self.dConst
* self.h / (2 * self.tau) * self.u[k - 2][
                            0] + -self.h * self.Function() + (
                                    2 * self.aConst - self.bConst * self.h)
/ self.alpha * self.Phi0(t))
```

```
    def RBound22(self, k, t):
        n = -self.cConst * self.h + 2 * self.aConst / self.h + self.h /
self.tau ** 2 + self.dConst * self.h / (
                    2 * self.tau) + self.delta / self.gamma * (2 *
self.aConst + self.bConst * self.h)
        return 1 / n * (2 * self.aConst / self.h * self.u[k][-2] + self.h /
self.tau ** 2 * (
                    2 * self.u[k - 1][-1] - self.u[k - 2][-1]) +
self.dConst * self.h / (2 * self.tau) * self.u[k - 2][
                            -1] + self.h * self.Function() + (
                                    2 * self.aConst + self.bConst * self.h)
/ self.gamma * self.PhiL(t))

    def LBound23(self, k, t):
        n = 2 * self.h * self.beta - 3 * self.alpha
        return self.alpha / n * self.u[k - 1][2] - 4 * self.alpha / n *
self.u[k - 1][1] + 2 * self.h / n * self.Phi0(t)

    def RBound23(self, k, t):
        n = 2 * self.h * self.delta + 3 * self.gamma
        return 4 * self.gamma / n * self.u[k - 1][-2] - self.gamma / n *
self.u[k - 1][-3] + 2 * self.h / n * self.PhiL(
            t)

algorithms = ('Explicit', 'Implicit')
T, K, N = 70, 750, 15
answers = dict()
solver = HyperbolicSolver(args)
analytic = solver.AnaliticalSolutionMatrix(N, K, T)
answers['Analytic'] = analytic
for algorithm in algorithms:
    args['algorithm'] = algorithm
    solver = HyperbolicSolver(args)
    numeric = solver.Solve(N, K, T)
    answers[algorithm] = numeric

def calculate_error(numeric_data, analytic_data):
    error_list = []
    error_values = [[abs(i - j) for i, j in zip(x, y)] for x, y in
zip(numeric_data, analytic_data)]
    for i in range(len(error_values)):
        tmp = 0
        for j in error_values[i]:
            tmp += j
        error_list.append(tmp / len(error_values[i]))
    return error_list

def make_graphics(data_dict, N, K, T, time=2):
    fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 8))
    fig.subplots_adjust(hspace=0.5)
    x_values = np.arange(0, np.pi / 2, np.pi / 2 / N)
    time_values = np.arange(0, T, T / K)
    analytic_values = np.array(data_dict['Analytic'])
    explicit_values = np.array(data_dict['Explicit'])
    implicit_values = np.array(data_dict['Implicit'])
    colors = ['black', 'red', 'green']
    ax1.set_title('График U(x)')
    ax1.plot(x_values[0:-2], analytic_values[time][0:-2], color=colors[0],
label='Analytic')
    ax1.plot(x_values[0:-2], explicit_values[time][0:-2], color=colors[1],
```

```
label='Explicit')
    ax1.plot(x_values[0:-2], implicit_values[time][0:-2], color=colors[2],
label='Implicit')
    ax1.legend(loc='best')
    ax1.set_ylabel('U')
    ax1.set_xlabel('x')
    ax1.grid(True)
    ax2.set_title('График error(time) - зависимость ошибки от времени')
    for method, color in zip(algorithms, colors):
        ax2.plot(time_values, calculate_error(data_dict[method],
data_dict['Analytic']), label=method, color=color)
    ax2.legend(loc='best')
    ax2.set_ylabel('Error')
    ax2.set_xlabel('t')
    ax2.grid(True)
    plt.show()

make_graphics(answers, N, K, T)
```

Результат работы





Вывод

Благодаря данной лабораторной работе, я приобрел знания в области численных методов для решения дифференциальных уравнений гиперболического типа: были исследованы различные методы решения начально-краевой задачи для дифференциального уравнения гиперболического типа, а также была оценена точность и эффективность каждого метода, построен график зависимости ошибки от времени и график U(x).

Задание №7.

Решить краевую задачу для дифференциального уравнения эллиптического типа. Аппроксимацию уравнения произвести с использованием центрально-разностной схемы. Для решения дискретного аналога применить следующие методы: метод простых итераций (метод Либмана), метод Зейделя, метод простых итераций с верхней релаксацией. Вычислить погрешность численного решения путем сравнения результатов с приведенным в задании аналитическим решением $U(x, y)$. Исследовать зависимость погрешности от сеточных параметров h_x, h_y.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -2u$$

$$u(0, y) = \cos y,$$

$$u(\frac{\pi}{2}, y) = 0,$$

$$u(x, 0) = \cos x$$

$$u(x, \frac{\pi}{2}) = 0$$

Аналитическое решение: $U(x, y) = \cos x \cos y$.

Метод решения

Для выполнения данной работы я решил ДУ эллиптического типа, реализовав четыре метода: метод простых итераций, метод простых итераций с верхней релаксацией, а также метод Зейделя.

Код программы

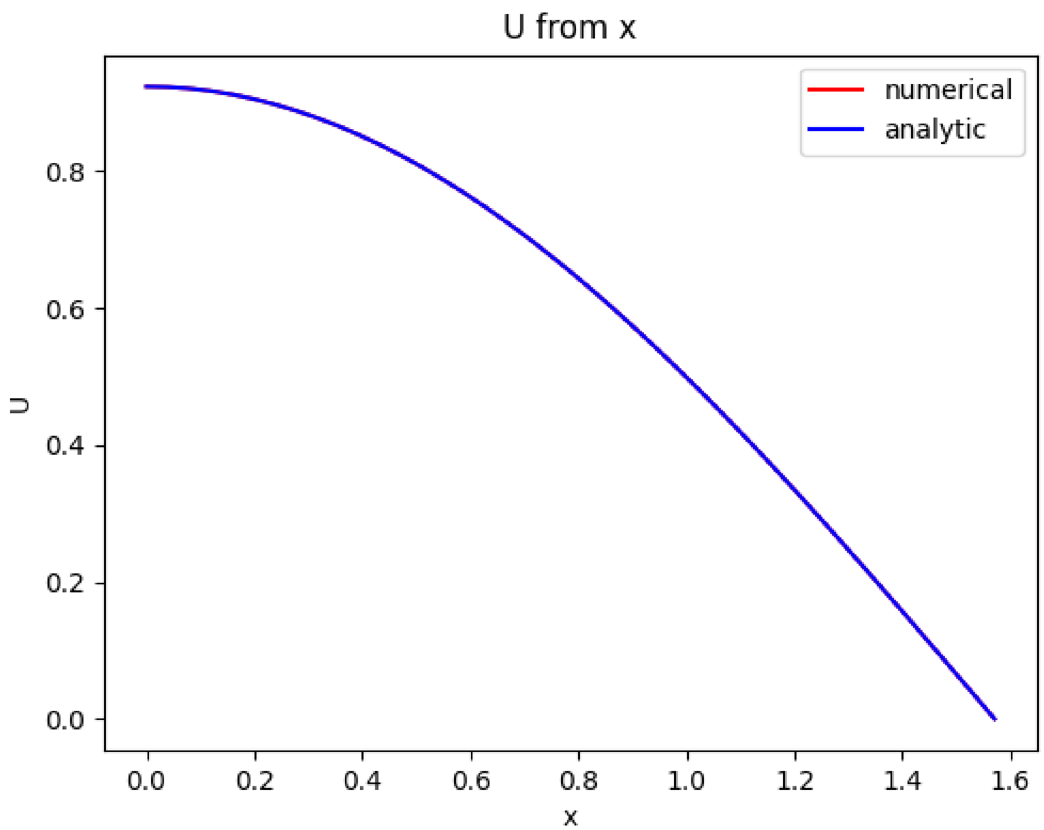
```
import copy
import numpy as np

def diff(L, u, nx, ny):
    mx = 0
    for i in range(nx):
        for j in range(ny):
            mx = max(mx, abs(u[i][j] - L[i][j]))
    return mx

class EquationParameters:
    def __init__(self, parameters):
        for key, value in parameters.items():
            setattr(self, key, value)

class ElepticalSolver:
    def __init__(self, params, equation_type):
        self.data = EquationParameters(params)
        self.hx = 0
        self.hy = 0
```

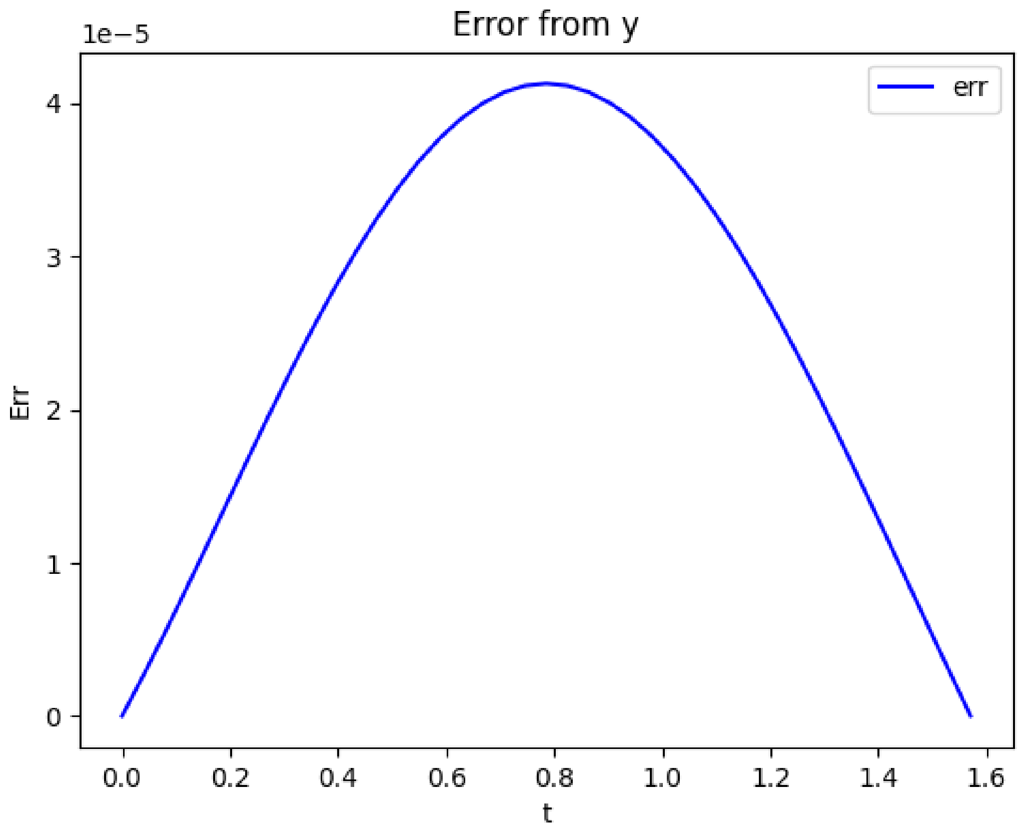
```
        self.iteration = 0
        self.eps = 1e-6
        try:
            self.solve_func = getattr(self, f'{equation_type}_solver')
        except:
            raise Exception("This type does not exist")

    def initalizeU(self, x, y):
        u = np.zeros((len(x), len(y)))
        for i in range(len(x)):
            u[i][0] = self.data.phi3(x[i]) / self.data.gamma2
            u[i][-1] = self.data.phi4(x[i]) / self.data.delta2
        for j in range(len(y)):
            u[0][j] = self.data.phi1(y[j]) / self.data.alpha2
            u[-1][j] = self.data.phi2(y[j]) / self.data.beta2
        return u

    def solve(self, nx, ny):
        self.hx = self.data.lx / nx
        self.hy = self.data.ly / ny
        x = np.arange(0, self.data.lx + self.hx, self.hx)
        y = np.arange(0, self.data.ly + self.hy, self.hy)
        u = self.initalizeU(x, y)
        for i in range(1, nx):
            for j in range(1, ny):
                u[i][j] = u[0][j] + (x[i] - x[0]) * (u[-1][j] - u[0][j]) /
(x[-1] - x[0])
        return self.solve_func(nx, ny, x, y, u)

    def analyticSolve(self, nx, ny):
        self.hx = self.data.lx / nx
        self.hy = self.data.ly / ny
        x = np.arange(0, self.data.lx + self.hx, self.hx)
        y = np.arange(0, self.data.ly + self.hy, self.hy)
        u = []
        for yi in y:
            u.append([self.data.solution(xi, yi) for xi in x])
        return u

    def simpleIterationMethod_solver(self, nx, ny, x, y, u):
        cur_eps = 1e9
        while self.iteration < 10000:
            L = copy.deepcopy(u)
            u = self.initalizeU(x, y)
            for j in range(1, len(y) - 1):
                for i in range(1, len(x) - 1):
                    u[i][j] = (self.hx * self.hx * self.data.f(x[i], y[j])
-
                               (L[i + 1][j] + L[i - 1][j]) - self.data.d *
self.hx * self.hx *
                               (L[i][j + 1] + L[i][j - 1]) /
                               (self.hy * self.hy) - self.data.a * self.hx
* 0.5 *
                               (L[i + 1][j] - L[i - 1][j]) - self.data.b *
self.hx * self.hx *
                               (L[i][j + 1] - L[i][j - 1]) /
                               (2 * self.hy)) / (self.data.c * self.hx *
self.hx - 2 *
                                                  (self.hy * self.hy +
self.data.d * self.hx * self.hx) /
                                                  (self.hy * self.hy))
```

```
            last_eps = cur_eps
            cur_eps = diff(L, u, nx, ny)
            if diff(L, u, nx, ny) <= self.eps or last_eps < cur_eps:
                break
            self.iteration += 1
        return u, self.iteration

    def seidelMethod_solver(self, nx, ny, x, y, u):
        cur_eps = 1e9
        while self.iteration < 10000:
            L = copy.deepcopy(u)
            u = self.initalizeU(x, y)
            for j in range(1, len(y) - 1):
                for i in range(1, len(x) - 1):
                    u[i][j] = ((self.hx ** 2) * self.data.f(x[i], y[j]) -
                               (L[i + 1][j] + u[i - 1][j]) - self.data.d *
(self.hx ** 2) *
                               (L[i][j + 1] + u[i][j - 1]) / (self.hy ** 2)
- self.data.a * self.hx * 0.5 *
                               (L[i + 1][j] - u[i - 1][j]) - self.data.b *
(self.hx ** 2) *
                               (L[i][j + 1] - u[i][j - 1]) /
                               (2 * self.hy)) / \
                              (self.data.c * (self.hx ** 2) - 2 * (self.hy
** 2 + self.data.d * (self.hx ** 2)) /
                               (self.hy ** 2))
            last_eps = cur_eps
            cur_eps = diff(L, u, nx, ny)
            if cur_eps <= self.eps or last_eps < cur_eps:
                break
            self.iteration += 1
        return u, self.iteration

    def simpleIterationMethodRelaxed_solver(self, nx, ny, x, y, u):
        cur_eps = 1e9
        while self.iteration < 10000:
            L = copy.deepcopy(u)
            u = self.initalizeU(x, y)
            for j in range(1, len(y) - 1):
                for i in range(1, len(x) - 1):
                    u[i][j] = (((self.hx ** 2) * self.data.f(x[i], y[j]) -
                                (L[i + 1][j] + u[i - 1][j]) - self.data.d *
(self.hx ** 2) *
                                (L[i][j + 1] + u[i][j - 1]) / (self.hy **
2) - self.data.a * self.hx * 0.5 *
                                (L[i + 1][j] - u[i - 1][j]) - self.data.b *
(self.hx ** 2) *
                                (L[i][j + 1] - u[i][j - 1]) /
                                (2 * self.hy)) / (self.data.c * (self.hx **
2) - 2 *
                                                  (self.hy ** 2 +
self.data.d * (self.hx ** 2)) /
                                                  (self.hy ** 2))) *
self.data.w + (1 - self.data.w) * L[i][j]
            last_eps = cur_eps
            cur_eps = diff(L, u, nx, ny)
            if diff(L, u, nx, ny) <= self.eps or last_eps < cur_eps:
                break
            self.iteration += 1
        return u, self.iteration
```

```
def compareError(a, b):
    err = 0
    lst = [abs(i - j) for i, j in zip(a, b)]
    for each in lst:
        err = max(err, each)
    return err

data = {'equation_type': 'simpleIterationMethodRelaxed', 'nx': 40, 'ny':
40}
```

Результаты работы





Вывод по лабораторной работе

Эта лабораторная работа обогатила мои познания в области численных методов для эллиптических дифференциальных уравнений. В ходе работы была использована центрально-разностная схема, успешно реализованы три ключевых метода, предусмотренных заданием. Также проведена оценка точности и эффективности каждого метода, с построением графиков, отображающих зависимость ошибки от времени и функции U(x).

Задание №8.

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

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} - xy\sin t,$$

$$u(0, y, t) = 0,$$

$$u(1, y, t) = y\cos t,$$

$$u(x, 0, t) = 0,$$

$$u(x, 1, t) = x\cos t,$$

$$u(x, y, 0) = xy.$$

Аналитическое решение: $U(x, y, t) = xy\cos t$.

Метод решения

Чтобы выполнить данную лабораторную работу, мне пришлось решить двумерную начально-краевую задачу для ДУ параболического типа, а также вычислять погрешность, сравнивая с аналитическим решением результаты реализованных численных решений.

Код программы

```
import ipywidgets as widgets
from ipywidgets import interact
from IPython.display import display
from ipywidgets import interact, interactive, fixed, interact_manual
from tqdm import tqdm
import random
import matplotlib.pyplot as plt
from matplotlib import cm
import math
import sys
import warnings
import numpy as np
import glob
import moviepy.editor as mpy
from functools import reduce
from mpl_toolkits.mplot3d import Axes3D
import plotly.offline as offline
from plotly.graph_objs import *


def RunThroughMethod(a, b, c, d):
    size = len(a)
```

```
    p, q = [], []
    p.append(-c[0] / b[0])
    q.append(d[0] / b[0])
    for i in range(1, size):
        pTmp = -c[i] / (b[i] + a[i] * p[i - 1])
        qTmp = (d[i] - a[i] * q[i - 1]) / (b[i] + a[i] * p[i - 1])
        p.append(pTmp)
        q.append(qTmp)
    x = [0 for _ in range(size)]
    x[size - 1] = q[size - 1]
    for i in range(size - 2, -1, -1):
        x[i] = p[i] * x[i + 1] + q[i]
    return x

class ParabolicSolver:
    def __init__(self, args):
        for name, value in args.items():
            setattr(self, name, value)
        self.hx = 0
        self.hy = 0
        self.tau = 0
        try:
            self.solve_func = getattr(self, args['algorithm'])
        except:
            raise Exception("This type does not exist")

    def Solve(self, nx, ny, T, K):
        self.hx = self.dx / nx
        self.hy = self.dy / ny
        self.tau = T / K
        x = np.arange(0, self.dx + self.hx, self.hx)
        y = np.arange(0, self.dy + self.hy, self.hy)
        t = np.arange(0, T + self.tau, self.tau)
        uu = np.zeros((len(x), len(y), len(t)))
        for i in range(len(x)):
            for j in range(len(y)):
                uu[i][j][0] = self.Psi(x[i], y[j])
        return self.solve_func(x, y, t, uu)

    def AnalyticSolve(self, nx, ny, T, K):
        self.hx = self.dx / nx
        self.hy = self.dy / ny
        self.tau = T / K
        x = np.arange(0, self.dx + self.hx, self.hx)
        y = np.arange(0, self.dy + self.hy, self.hy)
        t = np.arange(0, T + self.tau, self.tau)
        uu = np.zeros((len(x), len(y), len(t)))
        for i in range(len(x)):
            for j in range(len(y)):
                for k in range(len(t)):
                    uu[i][j][k] = self.AnaliticalSolution(x[i], y[j], t[k])
        return uu

    def VariableDirections(self, x, y, t, uu):
        for k in range(1, len(t)):
            u1 = np.zeros((len(x), len(y)))
            t2 = t[k] - self.tau / 2
            for j in range(len(y) - 1):
                aa = np.zeros(len(x))
                bb = np.zeros(len(x))
                cc = np.zeros(len(x))
```

```
                dd = np.zeros(len(x))
                bb[0] = self.hx * self.alpha2 - self.alpha1
                bb[-1] = self.hx * self.beta2 + self.beta1
                cc[0] = self.alpha1
                aa[-1] = -self.beta1
                dd[0] = self.Phi11(y[j], t2) * self.hx
                dd[-1] = self.Phi12(y[j], t2) * self.hx
                for i in range(len(x) - 1):
                    aa[i] = self.a - self.hx * self.c / 2
                    bb[i] = self.hx ** 2 - 2 * (self.hx ** 2) / self.tau -
2 * self.a
                    cc[i] = self.a + self.hx * self.c / 2
                    dd[i] = -2 * (self.hx ** 2) * uu[i][j][k - 1] /
self.tau
                    - self.b * (self.hx ** 2) * (uu[i][j + 1][k - 1]
                                                 - 2 * uu[i][j][k - 1] +
uu[i][j - 1][k - 1]) / (self.hy ** 2)
                    - self.d * (self.hx ** 2) * (uu[i][j + 1][k - 1] -
uu[i][j - 1][k - 1]) / (2 * self.hy ** 2)
                    - (self.hx ** 2) * self.Function(x[i], y[j], t[k])
                xx = RunThroughMethod(aa, bb, cc, dd)
                for i in range(len(x)):
                    u1[i][j] = xx[i]
                    u1[i][0] = (self.Phi21(x[i], t2) - self.gamma1 *
u1[i][1] / self.hy) / (
                            self.gamma2 - self.gamma1 / self.hy)
                    u1[i][-1] = (self.Phi22(x[i], t2) + self.delta1 *
u1[i][-2] / self.hy) / (
                            self.delta2 + self.delta1 / self.hy)
            for j in range(len(y)):
                u1[0][j] = (self.Phi11(y[j], t2) - self.alpha1 * u1[1][j] /
self.hx) / (
                        self.alpha2 - self.alpha1 / self.hx)
                u1[-1][j] = (self.Phi12(y[j], t2) + self.beta1 * u1[-2][j]
/ self.hx) / (
                        self.beta2 + self.beta1 / self.hx)
            u2 = np.zeros((len(x), len(y)))
            for i in range(len(x) - 1):
                aa = np.zeros(len(x))
                bb = np.zeros(len(x))
                cc = np.zeros(len(x))
                dd = np.zeros(len(x))
                bb[0] = self.hy * self.gamma2 - self.gamma1
                bb[-1] = self.hy * self.delta2 + self.delta1
                cc[0] = self.gamma1
                aa[-1] = -self.delta1
                dd[0] = self.Phi21(x[i], t[k]) * self.hy
                dd[-1] = self.Phi22(x[i], t[k]) * self.hy
                for j in range(len(y) - 1):
                    aa[j] = self.b - self.hy * self.d / 2
                    bb[j] = self.hy ** 2 - 2 * (self.hy ** 2) / self.tau -
2 * self.b
                    cc[j] = self.b + self.hy * self.d / 2
                    dd[j] = -2 * (self.hy ** 2) * u1[i][j] / self.tau
                    - self.a * (self.hy ** 2) * (u1[i + 1][j]
                                                 - 2 * u1[i][j] + u1[i -
1][j]) / (self.hx ** 2)
                    - self.c * (self.hy ** 2) * (u1[i + 1][j] - u1[i -
1][j]) / (2 * self.hx ** 2)
                    - (self.hy ** 2) * self.Function(x[i], y[j], t[k])
                xx = RunThroughMethod(aa, bb, cc, dd)
```

```
                for j in range(len(y)):
                    u2[i][j] = xx[j]
                    u2[0][j] = (self.Phi11(y[j], t[k]) - self.alpha1 *
u2[1][j] / self.hx) / (
                            self.alpha2 - self.alpha1 / self.hx)
                    u2[-1][j] = (self.Phi12(y[j], t[k]) + self.beta1 * u2[-
2][j] / self.hx) / (
                            self.beta2 + self.beta1 / self.hx)
            for i in range(len(x)):
                u2[i][0] = (self.Phi21(x[i], t[k]) - self.gamma1 * u2[i][1]
/ self.hy) / (
                        self.gamma2 - self.gamma1 / self.hy)
                u2[i][-1] = (self.Phi22(x[i], t[k]) + self.delta1 * u2[i][-
2] / self.hy) / (
                        self.delta2 + self.delta1 / self.hy)
            for i in range(len(x)):
                for j in range(len(y)):
                    uu[i][j][k] = u2[i][j]
        return uu

    def FractionalSteps(self, x, y, t, uu):
        for k in range(len(t)):
            u1 = np.zeros((len(x), len(y)))
            t2 = t[k] - self.tau / 2
            for j in range(len(y) - 1):
                aa = np.zeros(len(x))
                bb = np.zeros(len(x))
                cc = np.zeros(len(x))
                dd = np.zeros(len(x))
                bb[0] = self.hx * self.alpha2 - self.alpha1
                bb[-1] = self.hx * self.beta2 + self.beta1
                cc[0] = self.alpha1
                aa[-1] = -self.beta1
                dd[0] = self.Phi11(y[j], t2) * self.hx
                dd[-1] = self.Phi12(y[j], t2) * self.hx
                for i in range(len(x) - 1):
                    aa[i] = self.a
                    bb[i] = -(self.hx ** 2) / self.tau - 2 * self.a
                    cc[i] = self.a
                    dd[i] = -(self.hx ** 2) * uu[i][j][k - 1] / self.tau -
(self.hx ** 2) * self.Function(x[i], y[j],

t2) / 2
                xx = RunThroughMethod(aa, bb, cc, dd)
                for i in range(len(x)):
                    u1[i][j] = xx[i]
                    u1[i][0] = (self.Phi21(x[i], t2) - self.gamma1 *
u1[i][1] / self.hy) / (
                            self.gamma2 - self.gamma1 / self.hy)
                    u1[i][-1] = (self.Phi22(x[i], t2) + self.delta1 *
u1[i][-2] / self.hy) / (
                            self.delta2 + self.delta1 / self.hy)
            for j in range(len(y)):
                u1[0][j] = (self.Phi11(y[j], t2) - self.alpha1 * u1[1][j] /
self.hx) / (
                        self.alpha2 - self.alpha1 / self.hx)
                u1[-1][j] = (self.Phi12(y[j], t2) + self.beta1 * u1[-2][j]
/ self.hx) / (
                        self.beta2 + self.beta1 / self.hx)
            u2 = np.zeros((len(x), len(y)))
            for i in range(len(x) - 1):
```

```
                aa = np.zeros(len(x))
                bb = np.zeros(len(x))
                cc = np.zeros(len(x))
                dd = np.zeros(len(x))

                bb[0] = self.hy * self.gamma2 - self.gamma1
                bb[-1] = self.hy * self.delta2 + self.delta1
                cc[0] = self.gamma1
                aa[-1] = -self.delta1
                dd[0] = self.Phi21(x[i], t[k]) * self.hy
                dd[-1] = self.Phi22(x[i], t[k]) * self.hy

                for j in range(len(y) - 1):
                    aa[j] = self.b
                    bb[j] = -(self.hy ** 2) / self.tau - 2 * self.b
                    cc[j] = self.b
                    dd[j] = -(self.hy ** 2) * u1[i][j] / self.tau -
(self.hy ** 2) * self.Function(x[i], y[j], t[k]) / 2
                xx = RunThroughMethod(aa, bb, cc, dd)
                for j in range(len(y)):
                    u2[i][j] = xx[j]
                    u2[0][j] = (self.Phi11(y[j], t[k]) - self.alpha1 *
u2[1][j] / self.hx) / (
                            self.alpha2 - self.alpha1 / self.hx)
                    u2[-1][j] = (self.Phi12(y[j], t[k]) + self.beta1 * u2[-
2][j] / self.hx) / (
                            self.beta2 + self.beta1 / self.hx)
            for i in range(len(x)):
                u2[i][0] = (self.Phi21(x[i], t[k]) - self.gamma1 * u2[i][1]
/ self.hy) / (
                        self.gamma2 - self.gamma1 / self.hy)
                u2[i][-1] = (self.Phi22(x[i], t[k]) + self.delta1 * u2[i][-
2] / self.hy) / (
                        self.delta2 + self.delta1 / self.hy)
            for i in range(len(x)):
                for j in range(len(y)):
                    uu[i][j][k] = u2[i][j]
        return uu

def Approximation(x_list, y, t, numericalAnswer):
    res = []
    for i in range(len(x_list)):
        res.append(numericalAnswer[0][i][y][t])
    return res

def PrepareNumerical(x, y, time, numericalAnswer):
    return [Approximation(x, y, time, numericalAnswer) for _ in
range(len(x))]

def PrepareAnalitic(x_list, y, t, analyticAnswer):
    res = []
    for xi in x_list:
        res.append(xi * y * np.cos(t))
    return res

def psi_0(x, t):
    return 0.0

def psi_1(x, t):
    return x * math.cos(t)
```

```python
def phi_0(y, t):
    return 0.0

def phi_1(y, t):
    return y * math.cos(t)

def u0(x, y):
    return x * y

def u(x, y, t):
    return x * y * math.cos(t)

class Schema:
    def __init__(self, rho=u0, psi0=psi_0, psi1=psi_1, phi0=phi_0,
phi1=phi_1,
                 lx0=0, lx1=1.0, ly0=0, ly1=1.0, T=3, order2nd=True):
        self.psi0 = psi0
        self.psi1 = psi1
        self.phi0 = phi0
        self.phi1 = phi1
        self.rho0 = rho
        self.T = T
        self.lx0 = lx0
        self.lx1 = lx1
        self.ly0 = ly0
        self.ly1 = ly1
        self.tau = None
        self.hx = None
        self.hy = None
        self.order = order2nd
        self.Nx = None
        self.Ny = None
        self.K = None
        self.cx = None
        self.bx = None
        self.cy = None
        self.by = None
        self.hx2 = None
        self.hy2 = None

    def set_l0_l1(self, lx0, lx1, ly0, ly1):
        self.lx0 = lx0
        self.lx1 = lx1
        self.ly0 = ly0
        self.ly1 = ly1

    def set_T(self, T):
        self.T = T

    def CalculateH(self):
        self.hx = (self.lx1 - self.lx0) / (self.Nx - 1)
        self.hy = (self.ly1 - self.ly0) / (self.Ny - 1)
        self.hx2 = self.hx * self.hx
        self.hy2 = self.hy * self.hy

    def CalculateTau(self):
        self.tau = self.T / (self.K - 1)

    def race_method(A, b):
        P = [-item[2] for item in A]
        Q = [item for item in b]
```

```
        P[0] /= A[0][1]
        Q[0] /= A[0][1]
        for i in range(1, len(b)):
            z = (A[i][1] + A[i][0] * P[i - 1])
            P[i] /= z
            Q[i] -= A[i][0] * Q[i - 1]
            Q[i] /= z
        for i in range(len(Q) - 2, -1, -1):
            Q[i] += P[i] * Q[i + 1]
        return Q

    def nparange(start, end, step=1):
        now = start
        e = 0.00000000001
        while now - e <= end:
            yield now
            now += step

    def CalculateLeftEdge(self, X, Y, t, square):
        for i in range(self.Ny):
            square[i][0] = self.phi0(Y[i][0], t)

    def CalculateRightEdge(self, X, Y, t, square):
        for i in range(self.Ny):
            square[i][-1] = self.phi1(Y[i][-1], t)

    def CalculateBottomEdge(self, X, Y, t, square):
        for j in range(1, self.Nx - 1):
            square[0][j] = self.psi0(X[0][j], t)

    def CalculateTopEdge(self, X, Y, t, square):
        for j in range(1, self.Nx - 1):
            square[-1][j] = self.psi1(X[-1][j], t)

    def CalculateLineFirstStep(self, i, X, Y, t, last_square, now_square):
        hy2 = self.hy2
        hx2 = self.hx2
        b = self.bx
        c = self.cx
        A = [(0, b, c)]
        w = [
            -self.cy * self.order * last_square[i - 1][1] -
            ((self.order + 1) * hx2 * hy2 - 2 * self.cy * self.order) *
last_square[i][1] -
            self.cy * self.order * last_square[i + 1][1] +
            self.tau * hy2 * hx2 * X[i][1] * Y[i][1] * math.sin(t) -
            c * now_square[i][0]
        ]
        A.extend([(c, b, c) for _ in range(2, self.Nx - 2)])
        w.extend([
            -self.cy * self.order * last_square[i - 1][j] -
            ((self.order + 1) * hx2 * hy2 - 2 * self.cy * self.order) *
last_square[i][j] -
            self.cy * self.order * last_square[i + 1][j] +
            self.tau * hy2 * hx2 * X[i][j] * Y[i][j] * math.sin(t)
            for j in range(2, self.Nx - 2)
        ])
        A.append((c, b, 0))
        w.append(
            -self.cy * self.order * last_square[i - 1][-2] -
            ((self.order + 1) * hx2 * hy2 - 2 * self.cy * self.order) *
```

```
last_square[i][-2] -
            self.cy * self.order * last_square[i + 1][-2] +
            self.tau * hy2 * hx2 * X[i][-2] * Y[i][-2] * math.sin(t) -
            c * now_square[i][-1]
        )
        line = self.race_method(A, w)
        for j in range(1, self.Nx - 1):
            now_square[i][j] = line[j - 1]

    def CalculateLineSecondStep(self, j, X, Y, t, last_square, now_square):
        hx2 = self.hx2
        hy2 = self.hy2
        c = self.cy
        b = self.by
        A = [(0, b, c)]
        w = [
            -self.cx * self.order * last_square[1][j - 1] -
            ((self.order + 1) * hx2 * hy2 - 2 * self.cx * self.order) *
last_square[1][j] -
            self.cx * self.order * last_square[1][j + 1] +
            self.tau * hy2 * hx2 * X[1][j] * Y[1][j] * math.sin(t) -
            c * now_square[0][j]
        ]
        A.extend([(c, b, c) for _ in range(2, self.Ny - 2)])
        w.extend([
            -self.cx * self.order * last_square[i][j - 1] -
            ((self.order + 1) * hx2 * hy2 - 2 * self.cx * self.order) *
last_square[i][j] -
            self.cx * self.order * last_square[i][j + 1] +
            self.tau * hy2 * hx2 * X[i][j] * Y[i][j] * math.sin(t)
            for i in range(2, self.Ny - 2)
        ])
        A.append((c, b, 0))
        w.append(
            -self.cx * self.order * last_square[-2][j - 1] -
            ((self.order + 1) * hx2 * hy2 - 2 * self.cx * self.order) *
last_square[-2][j] -
            self.cx * self.order * last_square[-2][j + 1] +
            self.tau * hy2 * hx2 * X[-2][j] * Y[-2][j] * math.sin(t) -
            c * now_square[-1][j]
        )
        line = self.race_method(A, w)
        for i in range(1, self.Ny - 1):
            now_square[i][j] = line[i - 1]

    def CalculateSquare(self, X, Y, t, last_square):
        square = [[0.0 for _ in range(self.Nx)] for _ in range(self.Ny)]
        self.CalculateLeftEdge(X, Y, t - 0.5 * self.tau, square)
        self.CalculateRightEdge(X, Y, t - 0.5 * self.tau, square)
        self.CalculateBottomEdge(X, Y, t - 0.5 * self.tau, square)
        self.CalculateTopEdge(X, Y, t - 0.5 * self.tau, square)
        for i in range(1, self.Ny - 1):
            self.CalculateLineFirstStep(i, X, Y, t - 0.5 * self.tau,
last_square, square)
        last_square = square
        square = [[0.0 for _ in range(self.Nx)] for _ in range(self.Ny)]
        self.CalculateLeftEdge(X, Y, t, square)
        self.CalculateRightEdge(X, Y, t, square)
        self.CalculateBottomEdge(X, Y, t, square)
        self.CalculateTopEdge(X, Y, t, square)
        for j in range(1, self.Nx - 1):
```

```python
            self.CalculateLineSecondStep(j, X, Y, t, last_square, square)
        return square

    def init_t0(self, X, Y):
        first = [[0.0 for _ in range(self.Nx)] for _ in range(self.Ny)]
        for i in range(self.Ny):
            for j in range(self.Nx):
                first[i][j] = self.rho0(X[i][j], Y[i][j])
        return first

    def __call__(self, Nx=20, Ny=20, K=20):
        self.Nx, self.Ny, self.K = Nx, Ny, K
        self.CalculateTau()
        self.CalculateH()
        self.bx = -2 * self.tau * self.hy2
        self.bx -= (1 + self.order) * self.hx2 * self.hy2
        self.cx = self.tau * self.hy2
        self.cy = self.tau * self.hx2
        self.by = -2 * self.tau * self.hx2
        self.by -= (1 + self.order) * self.hx2 * self.hy2
        x = list(self.nparange(self.lx0, self.lx1, self.hx))
        y = list(self.nparange(self.ly0, self.ly1, self.hy))
        X = [x for _ in range(self.Ny)]
        Y = [[y[i] for _ in x] for i in range(self.Ny)]
        taus = [0.0]
        ans = [self.init_t0(X, Y)]
        for t in self.nparange(self.tau, self.T, self.tau):
            ans.append(self.CalculateSquare(X, Y, t, ans[-1]))
            taus.append(t)
        return X, Y, taus, ans

def RealZByTime(lx0, lx1, ly0, ly1, t, f):
    x = np.arange(lx0, lx1 + 0.002, 0.002)
    y = np.arange(ly0, ly1 + 0.002, 0.002)
    X = np.ones((y.shape[0], x.shape[0]))
    Y = np.ones((x.shape[0], y.shape[0]))
    Z = np.ones((y.shape[0], x.shape[0]))
    for i in range(Y.shape[0]):
        Y[i] = y
    Y = Y.T
    for i in range(X.shape[0]):
        X[i] = x
    for i in range(Z.shape[0]):
        for j in range(Z.shape[1]):
            Z[i, j] = f(X[i, j], Y[i, j], t)
    return X, Y, Z

def Error(X, Y, t, z, ut=u):
    ans = 0.0
    for i in range(len(z)):
        for j in range(len(z[i])):
            ans = max(abs(ut(X[i][j], Y[i][j], t) - z[i][j]), ans)
    return (ans / len(z) / len(z[0]))

def StepSlice(lst, step):
    return lst[step]

def AnimateList(lst, play=False, interval=200):
    slider = widgets.IntSlider(min=0, max=len(lst) - 1, step=1, value=0)
    if play:
        play_widjet = widgets.Play(interval=interval)
```

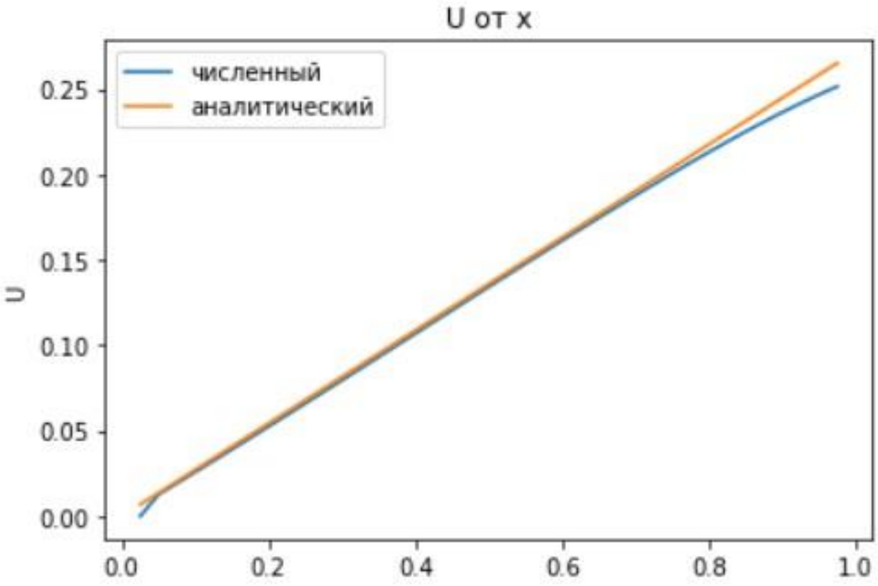
```python
        widgets.jslink((play_widjet, 'value'), (slider, 'value'))
        display(play_widjet)
    return interact(StepSlice,
                    lst=fixed(lst),
                    step=slider)

def PlotByTime(X, Y, T, Z, j, extrems, plot_true=True):
    t = T[j]
    z = Z[j]
    fig = plt.figure(num=1, figsize=(20, 13), clear=True)
    ax = fig.add_subplot(1, 1, 1, projection='3d')
    ax.plot_surface(np.array(X), np.array(Y), np.array(z))
    if plot_true:
        ax.plot_wireframe(*RealZByTime(0, 1, 0, 1, t, u), color="green")
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    ax.set_title(
        't = ' + str(round(t, 8)) + " error = " + str(round(Error(X, Y, t,
z), 11)),
        loc="right", fontsize=25
    )
    ax.set_zlim(extrems[0], extrems[1])
    fig.tight_layout()
    plt.close(fig)
    return fig

def SquareMinMax(z):
    minimum, maximum = z[0][0], z[0][0]
    for i in range(len(z)):
        for j in range(len(z[i])):
            minimum = z[i][j] if z[i][j] < minimum else minimum
            maximum = z[i][j] if z[i][j] > maximum else maximum
    return minimum, maximum

def SearchMinMax(zz):
    minimum, maximum = 0.0, 0.0
    for z in zz:
        minmax = SquareMinMax(z)
        minimum = minmax[0] if minmax[0] < minimum else minimum
        maximum = minmax[1] if minmax[1] > maximum else maximum
    return minimum, maximum

def PlotAnimate(nx=15, ny=15, k=50, t=5, plot_true=False):
    schema = Schema(T=t, order2nd=True)
    xx, yy, tt, zz = schema(Nx=nx, Ny=ny, K=k)
    extrems = SearchMinMax(zz)
    plots = []
    for j in range(len(tt)):
        plots.append(PlotByTime(xx, yy, tt, zz, j, extrems, plot_true))
    AnimateList(plots, play=True, interval=2000)

first = Schema(T=2 * math.pi, order2nd=False)
second = Schema(T=2 * math.pi, order2nd=True)

def GetGraphicH(solver, time=0, tsteps=40):
    h, e = [], []
    for N in range(4, 20, 1):
        x, y, t, z = solver(Nx=N, Ny=N, K=tsteps)
        h.append(solver.hx)
        e.append(Error(x, y, t[time], z[time]))
```

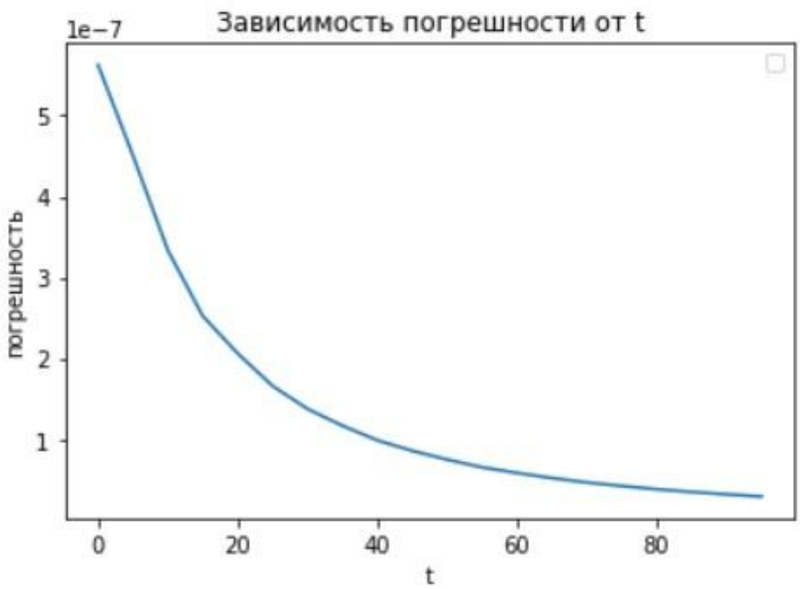
```python
    return h, e

TSTEPS = 100
time = random.randint(0, TSTEPS - 1)
h1, e1 = GetGraphicH(first, time, TSTEPS)
h2, e2 = GetGraphicH(second, time, TSTEPS)

def GetGraphicTau(solver):
    tau = []
    e = []
    for K in range(15, 100, 2):
        x, y, t, z = solver(Nx=10, Ny=10, K=K)
        tau.append(solver.tau)
        time = K // 2
        e.append(Error(x, y, t[time], z[time]))
    return tau, e

tau1, e1 = GetGraphicTau(first)
tau2, e2 = GetGraphicTau(second)

def FullError(X, Y, T, Z):
    ans = 0.0
    for k in range(len(T)):
        for i in range(len(X)):
            for j in range(len(X[i])):
                ans = max(abs(u(X[i][j], Y[i][j], T[k]) - Z[k][i][j]), ans)
    return (ans / len(T) / len(X) / len(X[0]))

TimeList = [random.randint(0, 40 - 1) for i in range(4)]

def plotDependenceError(Nx=4, Ny=4, K=40, Method=0):
    NxFix = Nx
    NyFix = Ny
    if Method != 0:
        method = True
        stri = 'Погрешность метода переменных направлений'
    else:
        method = False
        stri = 'Погрешность метода дробных шагов'
    plt.figure(figsize=(6, 12))
    for i in range(1, 4):
        plt.subplot(5, 1, i)
        Time = TimeList[i - 1]  # random.randint(0, K - 1)
        schema = Schema(T=Time, order2nd=method)
        h, eps = [], []
        for j in range(10):
            h.append([])
            eps.append([])
            X, Y, T, Z = schema(Nx, Ny, K)
            Nx += 1
            Ny += 1
            h[-1].append(schema.hx)
            eps[-1].append(Error(X, Y, T[Time], Z[Time]))
        Nx = NxFix
        Ny = NyFix
        plt.plot(np.array(h), np.array(eps), label="const T = " +
str(round(T[Time])))
        plt.xlabel('$h$')
        plt.ylabel('$\epsilon$')
        plt.title(stri)
        plt.legend()
```

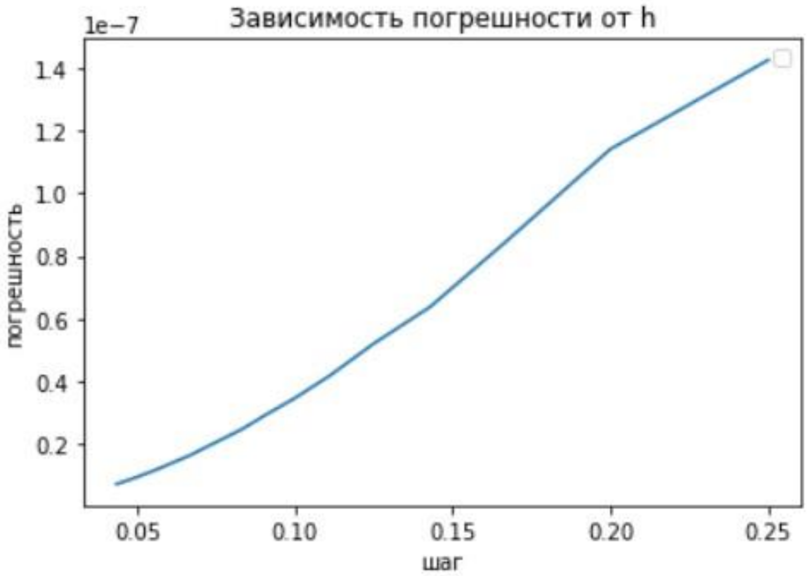
```
        plt.grid()
        plt.show()

firstMethod = True
nx, ny, T, K = 40, 40, 5, 100
plottingTime = 2
args = {
    'a': 1,
    'b': 1,
    'c': 0,
    'd': 0,
    'dx': 1,
    'dy': 1,
    'Function': lambda x, y, t: - x * y * np.sin(t),
    'alpha1': 0,
    'alpha2': 1,
    'beta1': 0,
    'beta2': 1,
    'gamma1': 0,
    'gamma2': 1,
    'delta1': 0,
    'delta2': 1,
    'Phi11': lambda y, t: 0,
    'Phi12': lambda y, t: y * np.cos(t),
    'Phi21': lambda x, t: 0,
    'Phi22': lambda x, t: x * np.cos(t),
    'Psi': lambda x, y: x * y,
    'AnaliticalSolution': lambda x, y, t: x * y * np.cos(t),
    'algorithm': 'VariableDirections' if firstMethod else 'FractionalSteps'
}

def DrawCharts(numericalAnswer, analyticAnswer, nx, ny, T, K, time):
    fig = plt.figure()
    hx = 1 / nx
    hy = 1 / ny
    tau = T / K
    x = np.arange(hx, 1, hx)
    y = np.arange(hy, 1, hy)
    t = np.arange(tau, T, tau)
    z1 = PrepareNumerical(x, 10, time, numericalAnswer)
    z2 = PrepareAnalitic(x, y[10], t[time], analyticAnswer)
    plt.title('U от x')
    plt.plot(x, z1[time], label='численный')
    plt.plot(x, z2, label='аналитический')
    plt.legend(loc='best')
    plt.ylabel('U')
    plt.xlabel('x')
    plt.show()
    solv = Schema(T=T, order2nd=firstMethod)
    h = []
    tau = []
    eps = []
    for i in tqdm(range(20)):
        h.append([])
        tau.append([])
        eps.append([])
        for j in range(40):
            N = i + 5
            K = j + 40
            X, Y, T, Z = solv(N, N, K)
            h[-1].append(solv.hx)
```

```
            tau[-1].append(solv.tau)
            eps[-1].append(FullError(X, Y, T, Z))
    t = range(0, 100, 5)
    err = [max(m) for m in eps]
    fig = plt.figure()
    plt.title('Зависимость погрешности от t')
    plt.plot(t, err)
    plt.legend(loc='best')
    plt.ylabel('погрешность')
    plt.xlabel('t')
    plt.show()
    fig = plt.figure()
    plt.title('Зависимость погрешности от t логарифмическая')
    plt.plot(np.log(t), np.log(err))
    plt.legend(loc='best')
    plt.ylabel('погрешность')
    plt.xlabel('t')
    plt.show()
    fig = plt.figure()
    plt.title('Зависимость погрешности от шага по t')
    plt.plot(tau[-1], eps[-1])
    plt.legend(loc='best')
    plt.ylabel('погрешность')
    plt.xlabel('шаг')
    plt.show()
    ee = [eps[i][-1] for i in range(len(eps))]
    hh = [h[i][-1] for i in range(len(h))]
    fig = plt.figure()
    plt.title('Зависимость погрешности от h')
    plt.plot(hh, ee)
    plt.legend(loc='best')
    plt.ylabel('погрешность')
    plt.xlabel('шаг')
    plt.show()
    fig = plt.figure()
    plt.title('Зависимость погрешности от h логарифмическая')
    plt.plot(np.log(hh), np.log(ee))
    plt.legend(loc='best')
    plt.ylabel('погрешность')
    plt.xlabel('шаг')
    plt.show()
    fig = plt.figure(num=1, figsize=(19, 12), clear=True)
    ax = fig.add_subplot(1, 1, 1, projection='3d')
    ax.plot_surface(np.array(h), np.array(tau), np.array(eps))
    ax.set(xlabel='$h$', ylabel='$t$', zlabel='погрешность',
title='Погрешность от шага и времени')
    fig.tight_layout()
    return (np.array(h), np.array(tau), np.array(eps))

h, tau, eps = DrawCharts(numericalAnswer, analyticAnswer, nx, ny, T, K,
plottingTime)
firstMethod = False
args['algorithm'] = 'FractionalSteps'
solver = ParabolicSolver(args)
numericalAnswer = solver.Solve(nx, ny, T, K),
analyticAnswer = solver.AnalyticSolve(nx, ny, T, K)

h1, tau1, eps1 = DrawCharts(numericalAnswer, analyticAnswer, nx, ny, T, K,
plottingTime)
```

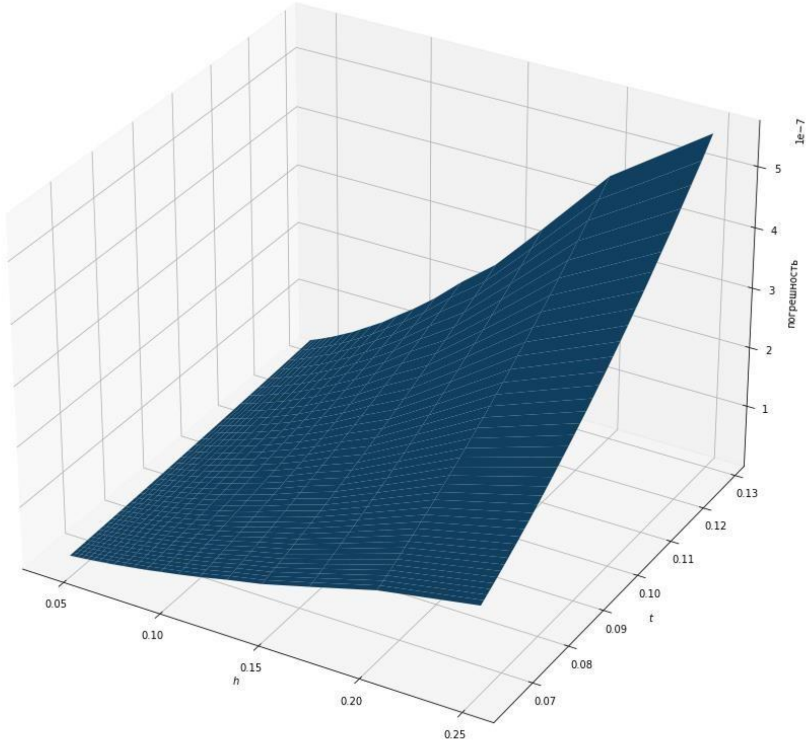
Результаты работы

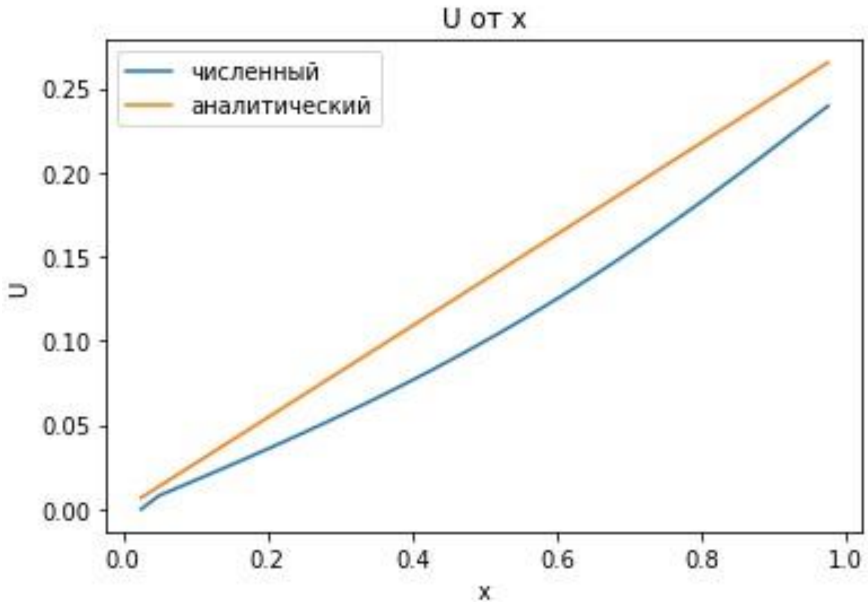


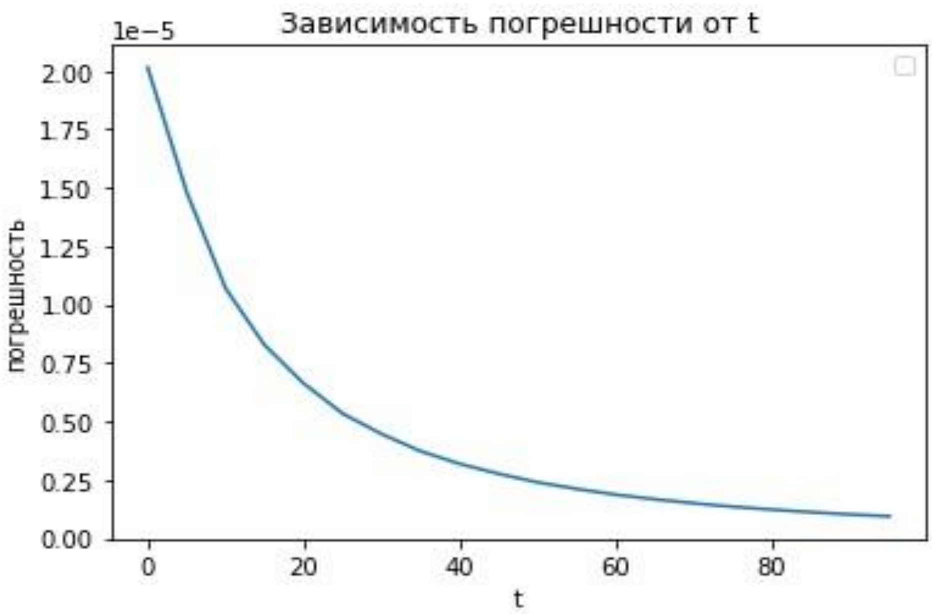




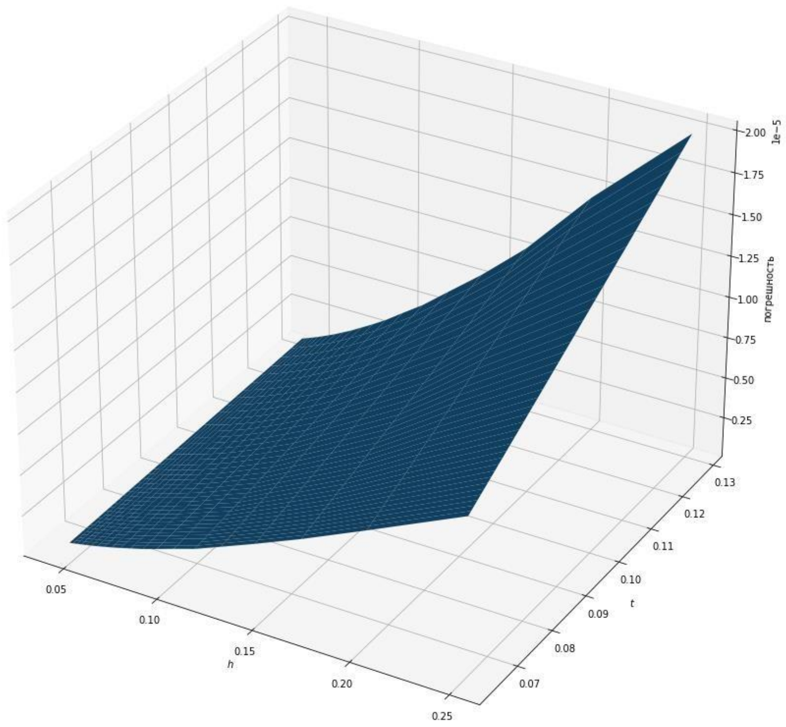
Погрешность от шага и времени:







Погрешность от шага и времени



Вывод по лабораторной работе

Благодаря данной лабораторной работе, я приобрел знания в области численных методов для решения дифференциальных уравнений параболического типа: были реализованы необходимые численные методы, были измерены погрешности от шага и времени, построены графики зависимостей погрешностей по заданию, а также графики U от x.