Московский авиационный институт
(Национальный исследовательский университет)

Институт: «Информационные технологии и прикладная математика»
Кафедра: 806 «Информационные технологии и прикладная математика»

Лабораторные работы по дисциплине «Численные методы»

Студент: Бухарин А.И.

Группа: М80-402Б-20

Преподаватель: Пивоваров Д.Е.

Москва, 2023

Лабораторная работа №1

1. Задание

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. Исследовать зависимость погрешности от сеточных параметров.

Вариант 4:

Уравнение:

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

где $g(x,t) = \frac{1}{2} e^{-\frac{1}{2}t} \cos x$

Граничные условия:

$$\begin{cases} u'_x(0,\ t) = \phi_0(t) = e^{-\frac{1}{2}t} \\ u'_x(\pi,\ t) = \phi_l(t) = -e^{-\frac{1}{2}t} \\ u(x,\ 0) = \sin x \end{cases}$$

Аналитическое решение:

$$u(x,t) = e^{-at} \sin x$$

2. Теория

<u>Конечно-разностная схема:</u>

Рассмотрим конечно-разностную схему решения краевой задачи на сетке с граничными параметрами *l*, *T*, и параметрами насыщенности сетки *N*,*K*. Отсюда размер шага по каждой из координат определяется по формулам:

$$h = \frac{l}{N},\ \tau = \frac{T}{K}$$

Определим значения функции на временном слое t^{k+1} путем разностной апроксимации производной:

$$\frac{\partial u}{\partial t}(x_j, t^k) = \frac{u_j^{k+1} - u_j^k}{\tau}$$

И одним из методов апроксимации второй производной по x:

$$\frac{\partial^2 u}{\partial x^2}(x_j, t^k)$$

<u>Явная конечно-разностная схема:</u>

Аппроксимируем вторую производную по значениям нижнего временного слоя t^{k+1}, а именно:

$$\frac{\partial^2 u}{\partial x^2}(x_j, t^k) = \frac{u_{j-1}^k - 2u_j^k + u_{j+1}^k}{h^2}$$

Тогда получим явную схему конечно-разностного метода во внутренних узлах сетки для $\forall j \in \{1, \dots, N-1\}, \forall k \in \{0, \dots, K-1\}$:

$$\frac{u_j^{k+1} - u_j^k}{\tau} = \frac{u_{j-1}^k - 2u_j^k + u_{j+1}^k}{h^2} + g(x, t^k)$$

Обозначим $\sigma = \frac{\tau}{h^2}$, тогда:

$$u_j^{k+1} = \sigma u_{j-1}^k + (1 - 2\sigma)u_j^k + \sigma u_{j+1}^k + \tau g(x_j, t^k)$$

Граничные же значения u_0^{k+1} и u_N^{k+1} определяются граничными условиями $u_x(0, t) = \phi_0(t)$ и $u_x(l, t) = \phi_l(t)$ при помощи аппроксимации производной.

<u>Неявная конечно-разностная схема:</u>

Аппроксимируем вторую производную по значениям верхнего временного слоя t^{k+1}, а именно:

$$\frac{\partial^2 u}{\partial x^2}(x_j, t^k) = \frac{u_{j-1}^{k+1} - 2u_j^{k+1} + u_{j+1}^{k+1}}{h^2}$$

Тогда получим явную схему конечно-разностного метода во внутренних узлах сетки для $\forall j \in \{1, \dots, N-1\}, \forall k \in \{0, \dots, K-1\}$:

$$\frac{u_j^{k+1} - u_j^k}{\tau} = \frac{u_{j-1}^{k+1} - 2u_j^{k+1} + u_{j+1}^{k+1}}{h^2} + g(x_j, t^{k+1})$$

Обозначим $\sigma = \frac{\tau}{h^2}$, $g_j^k = g(x_j, t^k)$

Тогда значения функции на верхнем временном слое можно найти из решения СЛАУ с трехдиагональной матрицей. Сделаем это с помощью метода прогонки.

Схема Кранка-Николсона:

Явно-неявная схема для $\forall j \in \{1, \ldots, N-1\}, \forall k \in \{0, \ldots, K-1\}$ будет выглядеть следующим образом:

$$\frac{u_j^{k+1} - u_j^k}{\tau} = \theta\left(\frac{u_{j-1}^{k+1} - 2u_j^{k+1} + u_{j+1}^{k+1}}{h^2} + g_j^{k+1}\right) + (1-\theta)\left(\frac{u_{j-1}^k - 2u_j^k + u_{j+1}^k}{h^2} + g_j^k\right)$$

θ - вес неявной части конечно-разностной схемы,

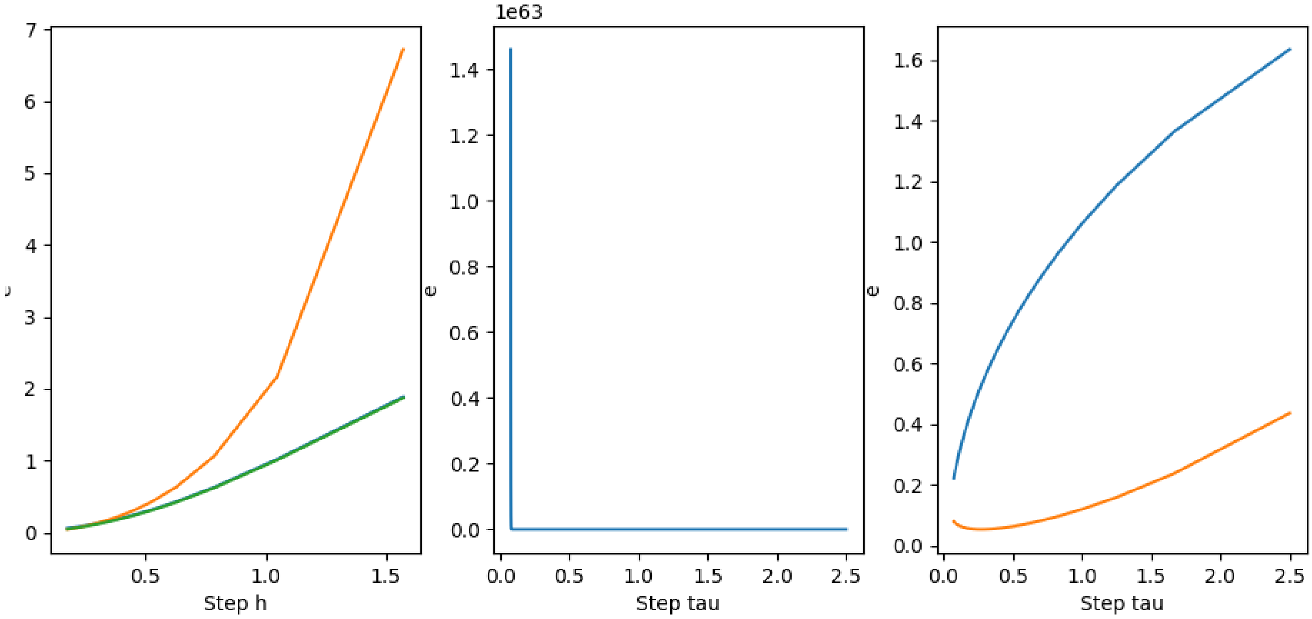
$(1-\theta)$ - вес для явной части.

При значении параметра $\theta = \frac{1}{2}$ мы имеем схему Кранка-Николсона.

Обозначим $\sigma = \frac{\tau}{h^2}$.

Тогда значения функции на слое можно найти эффективным образом с помощью методом прогонки.

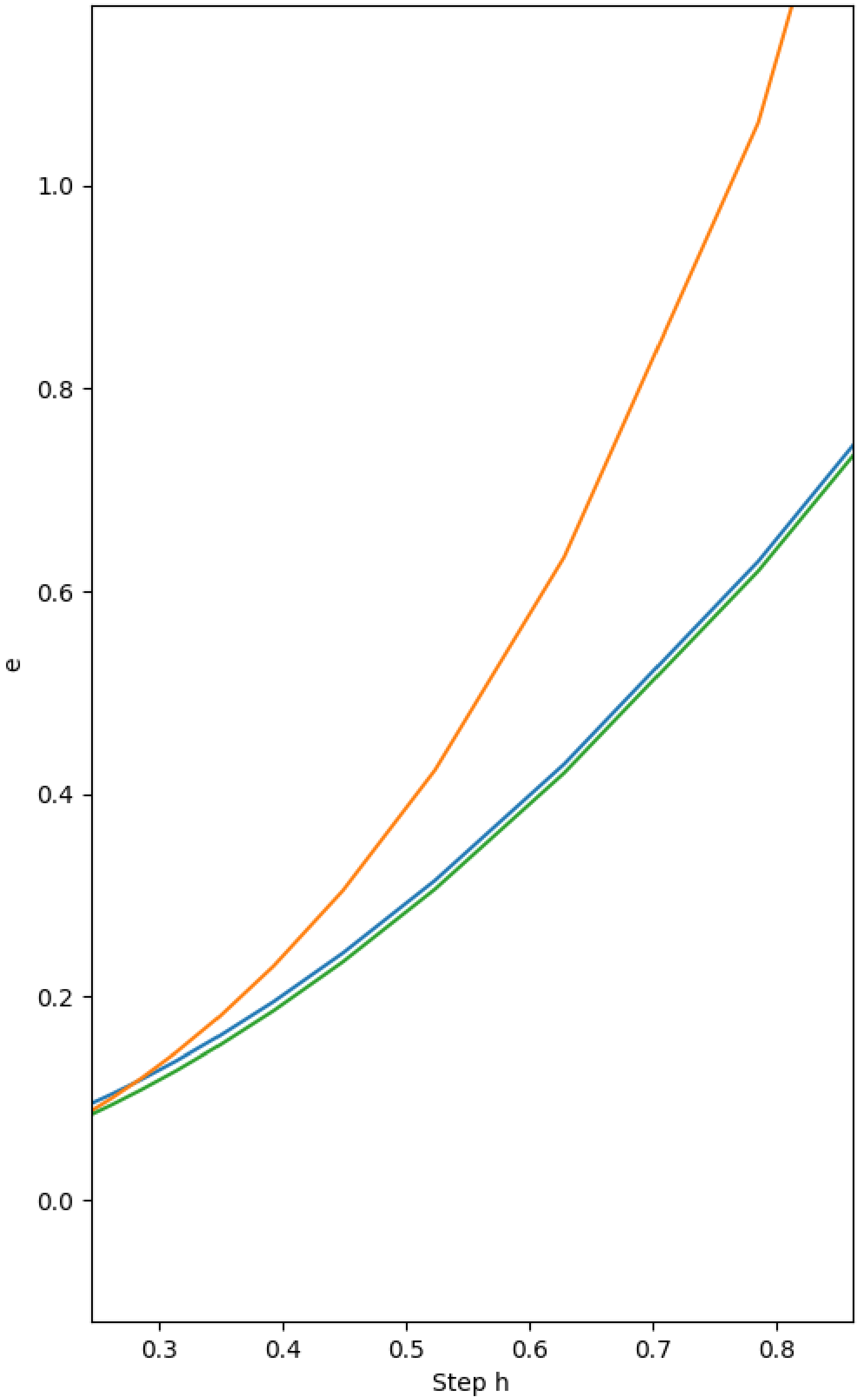
3. Результат



На первом графике (увеличенное изображение будет представлено ниже), синим желтым и зеленым представлены зависимости ошибок от выбранного размера шага явной, неявной и Кранка-Николсона конечных схем соответственно.

На втором графике представлена зависимость ошибки явной конечно-разностной схемы от шага по времени.

На третьем графике синим и желтым представлена зависимость ошибки от выбранного шага по времени неявной и Кранка-Николсона конечно-разностных схем соответственно.



4. Вывод

Выполнив данную лабораторную работу, я изучил явные и неявные конечно-разностные схемы, схему Кранка-Николсона для решения начально-краевой задачи для дифференциального уравнения параболического типа. Реализовал три варианта аппроксимации граничных условий, содержащих производные: двухточечная аппроксимация с первым порядком, трехточечная аппроксимация со вторым порядком, двухточечная аппроксимация со вторым порядком. Также исследовал зависимость погрешности от сеточных параметров τ и h.

5. Листинг кода

```
import matplotlib.pyplot as plt

import math

import numpy as np

def phi_0(t):

    return math.exp(-0.5*t)

def phi_l(t):

    return -math.exp(-0.5*t)

def u_0(x):

    return math.sin(x)

# Вместо cos нужно sin, так как ошибка в условии

def g(x, t):

    return 0.5 * math.exp(-0.5*t) * math.sin(x)

def u(x, t):

    return math.exp(-0.5*t)*math.sin(x)

class Schema:

    def __init__(self, f0=phi_0, fl=phi_l,

                 u0=u_0, g=g,

                 O=0.5, l0=0, l1=math.pi,

                 T=5, aprx_cls=None):

        self.fl = fl

        self.f0 = f0

        self.u0 = u0

        self.g = g

        self.T = T

        self.l0 = l0

        self.l1 = l1

        self.tau = None
```

```
    self.h = None
    self.O = O
    self.approx = None
    if aprx_cls is not None:
        self._init_approx(aprx_cls)
    self.sigma = None
def _init_approx(self, a_cls):
    self.approx = a_cls(self.f0, self.fl)

def SetApprox(self, aprx_cls):
    self._init_approx(self, aprx_cls)

def Set_l0_l1(self, l0, l1):
    self.l0 = l0
    self.l1 = l1

def SetT(self, T):
    self.T = T
def CalculateH(self, N):
    self.h = (self.l1 - self.l0) / N

def CalculateTau(self, K):
    self.tau = self.T / K
def CalculateSigma(self):
    self.sigma = self.tau / (self.h * self.h)
@staticmethod
def nparange(start, end, step=1):
    now = start
    e = 0.00000000001
    while now - e <= end:
        yield now
        now += step

def CalculateLine(self, t, x, lastLine):
    pass
```

```
    def __call__(self, N=50, K=70):
        N, K = N - 1, K - 1
        self.CalculateTau(K)
        self.CalculateH(N)
        self.CalculateSigma()
        ans = []
        x = list(np.arange(self.l0, self.l1 + 0.5 * self.h, self.h))
        lastLine = list(map(self.u0, x))
        ans.append(list(lastLine))
        X, Y = [], []
        X.append(x)
        Y.append([0.0 for _ in x])
        for t in np.arange(self.tau, self.T + 0.5 * self.tau, self.tau):
            ans.append(self.CalculateLine(t, x, lastLine))
            X.append(x)
            Y.append([t for _ in x])
            lastLine = ans[-1]
        return X, Y, ans
class ExplictSchema(Schema):
    def CalculateSigma(self):
        self.sigma = self.tau / (self.h * self.h)
        if self.sigma > 0.5:
            print("Sigma > 0.5")


    def CalculateLine(self, t, x, lastLine):
        line = [None for _ in lastLine]
        for i in range(1, len(x) - 1):
            line[i] = self.sigma * lastLine[i - 1]
            line[i] += (1 - 2 * self.sigma) * lastLine[i]
            line[i] += self.sigma * lastLine[i + 1]
            line[i] += self.tau * self.g(x[i], t - self.tau)
        line[0] = self.approx.explict_0(t, self.h, self.sigma,
                                        self.g, self.l0, lastLine,
                                        line, t - self.tau)
        line[-1] = self.approx.explict_l(t, self.h, self.sigma,
                                         self.g, self.l0, lastLine,
```

```
                                    line, t - self.tau)
        return line


class ImplictExplict(Schema):
    def SetO(self, O):
        self.O = O
    @staticmethod
    def SweepMethod(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
        x = [item for item in Q]
        for i in range(len(x) - 2, -1, -1):
            x[i] += P[i] * x[i + 1]
        return x


    def CalculateLine(self, t, x, lastLine):
        a = self.sigma * self.O
        b = -1 - 2 * self.sigma * self.O
        A = [(a, b, a) for _ in range(1, len(x) - 1)]
        w = [-(lastLine[i] + self.O * self.tau * self.g(x[i], t) + (1 - self.O) * self.sigma * (
                lastLine[i - 1] - 2 * lastLine[i] + lastLine[i + 1] + self.h * self.h * self.g(x[i], t - self.tau)))
             for i in range(1, len(x) - 1)]
        koeffs = self.approx.nikolson_0(t, self.h, self.sigma,
                                        self.g, self.l0, lastLine,
                                        self.O, t - self.tau)
        A.insert(0, koeffs[:-1])
        w.insert(0, koeffs[-1])
        koeffs = self.approx.nikolson_l(t, self.h, self.sigma,
                                        self.g, self.l1, lastLine,
```

```
                                        self.O, t - self.tau)
        A.append(koeffs[:-1])
        w.append(koeffs[-1])

        return self.SweepMethod(A, w)
class Approx:
    def __init__(self, f0, fl):
        self.f0 = f0
        self.fl = fl

    def explict_0(self, t, h, sigma, g, x0, l0, l1, t0):
        pass

    def explict_l(self, t, h, sigma, g, xN, l0, l1, t0):
        pass

    def nikolson_0(self, t, h, sigma, g, x0, l0, O, t0):
        pass

    def nikolson_l(self, t, h, sigma, g, xN, l0, O, t0):
        pass
class Approx2pointFirstOrder(Approx):
    def explict_0(self, t, h, sigma, g, x0, l0, l1, t0):
        return -h * self.f0(t) + l1[1]

    def explict_l(self, t, h, sigma, g, xN, l0, l1, t0):
        return h * self.fl(t) + l1[-2]

    def nikolson_0(self, t, h, sigma, g, x0, l0, O, t0):
        return 0, -1, 1, h * self.f0(t)

    def nikolson_l(self, t, h, sigma, g, xN, l0, O, t0):
        return -1, 1, 0, h * self.fl(t)
class Approx3pointSecondOrder(Approx):
    def explict_0(self, t, h, sigma, l0, l1, t0):
        return (-2 * h * self.f0(t) + 4 * l1[1] - l1[2]) / 3
```

```
    def explict_l(self, t, h, sigma, l0, l1, t0):
        return (2 * h * self.fl(t) + 4 * l1[-2] - l1[-3]) / 3

    def nikolson_0(self, t, h, sigma, g, x0, l0, O, t0):
        d = 2 * sigma * O * h * self.f0(t)
        d -= l0[1] + O * (t - t0) * g(x0 + h, t)
        d -= (1 - O) * sigma * (l0[0] - 2 * l0[1] + l0[2] + h * h * g(x0 + h, t0))
        return 0, -2 * sigma * O, 2 * sigma * O - 1, d

    def nikolson_l(self, t, h, sigma, g, xN, l0, O, t0):
        d = 2 * sigma * O * h * self.fl(t)
        d += l0[-2] + O * (t - t0) * g(xN - h, t)
        d += (1 - O) * sigma * (l0[-3] - 2 * l0[-2] + l0[-1] + h * h * g(xN - h, t0))
        return 1 - 2 * sigma * O, 2 * sigma * O, 0, d

class Approx2pointSecondOrder(Approx):
    def explict_0(self, t, h, sigma, g, x0, l0, l1, t0):
        return -2 * sigma * h * self.f0(t0) + 2 * sigma * l0[1] + \
               (1 - 2 * sigma) * l0[0] + (t - t0) * g(x0, t0)

    def explict_l(self, t, h, sigma, g, xN, l0, l1, t0):
        return 2 * sigma * h * self.fl(t0) + 2 * sigma * l0[-2] + \
               (1 - 2 * sigma) * l0[-1] + (t - t0) * g(xN, t0)

    def nikolson_0(self, t, h, sigma, g, x0, l0, O, t0):
        d = 2 * sigma * O * h * self.f0(t) - l0[0] - O * (t - t0) * g(x0, t)
        d -= 2 * (1 - O) * sigma * (l0[1] - l0[0] - h * self.f0(t0) + 0.5 * h * h * g(x0, t0))
        return 0, -(2 * sigma * O + 1), 2 * sigma * O, d

    def nikolson_l(self, t, h, sigma, g, xN, l0, O, t0):
        d = -2 * sigma * O * h * self.fl(t) - l0[-1] - O * (t - t0) * g(xN, t)
        d -= 2 * (1 - O) * sigma * (l0[-2] - l0[-1] + h * self.fl(t0) + 0.5 * h * h * g(xN, t0))
        return 2 * sigma * O, -(2 * sigma * O + 1), 0, d

def Error(x, y, z, f):
```

```
    ans = 0.0
    for i in range(len(z)):
        for j in range(len(z[i])):
            ans += (z[i][j] - f(x[i][j], y[i][j]))**2
    return ans **0.5

def GetStepHandError(solver, real_f):
    h = []
    e = []
    for N in range(3, 20):
        x, y, z = solver(N, 70)
        h.append(solver.h)
        e.append(Error(x, y, z, real_f))
    return h, e

explict = ExplictSchema(T = 1, aprx_cls=Approx2pointSecondOrder)
h, e = GetStepHandError(explict, u)
plt.subplot(1,3,1)
plt.plot(h,e)
plt.xlabel("Step h")
plt.ylabel("e")
# ImplictExplict с параметром O=1 это неявная схема
implict = ImplictExplict(T=1, aprx_cls=Approx2pointFirstOrder, O=1)
h, e = GetStepHandError(implict, u)
plt.plot(h,e)

# ImplictExplict с параметром O=0.5 это схема Кранка-Николсона
krank = ImplictExplict(T=1, aprx_cls=Approx2pointSecondOrder, O=0.5)
h, e = GetStepHandError(krank, u)
plt.plot(h,e)

def GetTandError(solver, real_f):
    tau, e = [], []
    for K in range(3, 70):
        x, y, z = solver(20, K)
        tau.append(solver.tau)
```

```
        e.append(Error(x, y, z, real_f))
    return tau, e


explict = ExplictSchema(T=5, aprx_cls=Approx2pointSecondOrder)
tau, e = GetTandError(explict, u)
plt.subplot(1,3,2)
plt.xlabel("Step tau")
plt.ylabel("e")
print(tau, e)
plt.plot(tau,e)
implict = ImplictExplict(T=5, aprx_cls=Approx2pointSecondOrder, O=1)
tau, e = GetTandError(implict, u)
plt.subplot(1,3,3)
plt.plot(tau,e)


krank = ImplictExplict(T=5, aprx_cls=Approx2pointSecondOrder)
tau, e = GetTandError(krank, u)
plt.plot(tau,e)
plt.xlabel("Step tau")
plt.ylabel("e")
fig = plt.gcf()
fig.canvas.manager.set_window_title('Зависимость ошибки от размера шага')
plt.show()
```

Лабораторная работа №6

1. Задача

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. Исследовать зависимость погрешности от сеточных параметров.

Вариант 4:

Уравнение:

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

Граничные условия:

$$\begin{cases} u(0,\ t) = e^{-t}\cos 2t \\ u(\frac{\pi}{2},\ t) = 0 \\ u(x,\ 0) = \psi_1(x) = e^{-x}\cos x \\ u_t(x,0) = \psi_2(x) = -e^{-x}\cos x \end{cases}$$

Аналитическое решение:

$$u(x,t) = e^{-x-t}\cos x \cos 2t$$

2. Теория

Конечно-разностная схема:

Так как значения функции $u_j^k = u(x_j, t^k)$ для всех координат $x_j = jh,\ \forall j \in \{0, \dots, N\}$ на предыдущих временных слоях известны, попробуем определить значения функции на следующем временном слое t^{k+1} с помощью разностной аппроксимации производной:

$$\frac{\partial^2 u}{\partial t^2}(x_j, t^k) = \frac{u_j^{k+1} - 2u_j^k + u_j^{k-1}}{\tau^2}$$

И одним из методов апроксимации второй производной по *x*:

$$\frac{\partial^2 u}{\partial x^2}(x_j, t^k)$$

Для расчета u_0^{k+1} и u_N^{k+1} можно использовать следующие формулы:

$$u_j^0 = \psi_1(x_j)$$

$$u_j^1 = \psi_1(x_j) + \tau\psi_2(x_j) + \frac{\tau^2}{2}\psi_1''(x_j) + O(\tau^2)$$

$$u_j^1 = \psi_1(x_j) + \tau\psi_2(x_j) + O(\tau^1)$$

Неявная конечно-разностная схема:

Аппроксимируем вторую производную по значениям верхнего временного слоя t^{k+1}:

$$\frac{\partial^2 u}{\partial x^2}(x_j, t^k) = \frac{u_{j-1}^{k+1} - 2u_j^{k+1} + u_{j+1}^{k+1}}{h^2}$$

Тогда у нас получается явная схема конечно-разностного метода во внутренних узлах сетки:

$$\frac{u_j^{k+1} - 2u_j^k + u_j^{k-1}}{\tau^2} + \frac{u^{k+1} - u^{k-1}}{\tau} = \frac{u_{j-1}^{k+1} - 2u_j^{k+1} + u_{j+1}^{k+1}}{h^2} + \frac{u_{j+1}^{k+1} - u_{j-1}^{k+1}}{h} - 3u_j^{k+1}$$

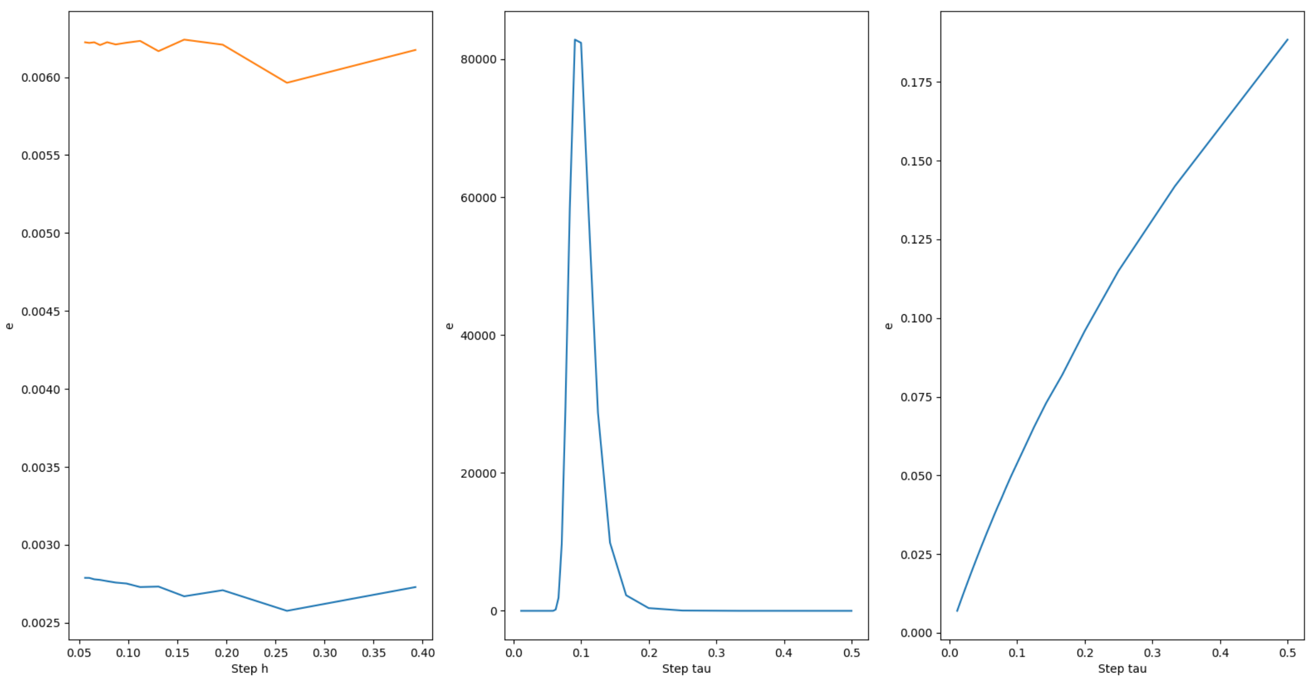
Введем обозначение $\sigma = \frac{\tau^2}{h^2}$. Следовательно значения функции на слое можно найти из решения СЛАУ с трехдиагональной матрицей.

3. Результат

На первом графике представлены в синем и рыжем цвете зависимости ошибки от шага для явной и неявной конечно-разностных схем соответственно.

На втором графике представлена зависимость ошибки от шага по времени для явной конечно-разностной схемы.

На третьем графике представлена зависимость ошибки от шага по времени для неявной конечно-разностной схемы.



4. Вывод

Выполнив данную лабораторную работу, изучил явную схему крест и неявную схему для решения начально-краевой задачи для дифференциального уравнения гиперболического типа. Выполнил три варианта аппроксимации граничных условий, содержащих производные: двухточечная аппроксимация с первым порядком, трехточечная аппроксимация со вторым порядком и двухточечная аппроксимация со вторым порядком. В различные моменты времени вычислил погрешность численного решения путем сравнения результатов с приведенным в задании аналитическим решением. Также исследовал зависимость погрешности от сеточных параметров τ и h.

5. Листинг кода

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



def psi_1(x):
    return math.exp(-x) * math.cos(x)

def psi_2(x):
    return -math.exp(-x) * math.cos(x)

def phi_0(t):
    return math.exp(-t) * math.cos(2 * t)

def phi_l(t):
    return 0

def dpsi2_dx2(x):
    return 2 * math.sin(x) * math.exp(-x)

def u(x, t):
    return math.exp(-x - t) * math.cos(2 * t) * math.cos(x)


class Schema:
    def __init__(self, psi1=psi_1, psi2=psi_2,
                 diffpsi2=dpsi2_dx2, f0=phi_0, fl=phi_l,
                 l0=0, l1=math.pi / 2, T=5, order2nd=True):
        self.psi1 = psi1
```

```
        self.diffpsi = diffpsi2
        self.psi2 = psi2
        self.T = T
        self.l0 = l0
        self.l1 = l1
        self.tau = None
        self.h = None
        self.order = order2nd
        self.sigma = None
        self.f0 = f0
        self.fl = fl

    def CalculateH(self, N):
        self.h = (self.l1 - self.l0) / N

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

    def CalculateSigma(self):
        self.sigma = self.tau * self.tau / (self.h * self.h)

    def Set_l0_l1(self, l0, l1):
        self.l0 = l0
        self.l1 = l1

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

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

    def CalculateLine(self, t, x, lastLine1, lastLine2):
        pass

    def __call__(self, N=30, K=70):
        N, K = N - 1, K - 1
        self.CalculateTau(K)
```

```
        self.CalculateH(N)
        self.CalculateSigma()
        ans = []
        x = list(self.nparange(self.l0, self.l1, self.h))
        lastLine = list(map(self.psi1, x))
        ans.append(list(lastLine))
        if self.order:
            lastLine = list(
                map(lambda a: self.psi1(a) + self.tau * self.psi2(a) + self.tau * self.tau * self.diffpsi(a) / 2, x))
        else:
            lastLine = list(map(lambda a: self.psi1(a) + self.tau * self.psi2(a), x))
        ans.append(list(lastLine))
        X = [x, x]
        Y = [[0.0 for _ in x]]
        Y.append([self.tau for _ in x])
        for t in self.nparange(self.tau + self.tau, self.T, self.tau):
            ans.append(self.CalculateLine(t, x, ans[-1], ans[-2]))
            X.append(x)
            Y.append([t for _ in x])
        return X, Y, ans


class ExplictSchema(Schema):
    def CalculateSigma(self):
        self.sigma = self.tau * self.tau / (self.h * self.h)
        if self.sigma > 1:
            warnings.warn("Sigma > 1")

    def CalculateLine(self, t, x, lastLine1, lastLine2):
        line = [None for _ in lastLine1]
        for i in range(1, len(x) - 1):
            line[i] = self.sigma * (lastLine1[i - 1] - 2 * lastLine1[i] + lastLine1[i + 1])
            line[i] -= 3 * self.tau * self.tau * lastLine1[i]
            line[i] += 2 * lastLine1[i]
            line[i] += (self.tau - 1) * lastLine2[i]
            line[i] += self.tau * self.tau * (lastLine1[i + 1] - lastLine1[i - 1]) / self.h
            line[i] /= (1 + self.tau)
        line[0] = self.f0(t)
        line[-1] = self.fl(t)
```

```
        return line


class ImplictSchema(Schema):
    @staticmethod
    def SweepMethod(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
        x = [item for item in Q]
        for i in range(len(x) - 2, -1, -1):
            x[i] += P[i] * x[i + 1]
        return x

    def CalculateLine(self, t, x, lastLine1, lastLine2):
        a = 1 - self.h
        c = 1 + self.h
        b = -2 - 3 * self.h * self.h - (1 + self.tau) / self.sigma
        A = [(a, b, c) for _ in range(2, len(x) - 2)]
        w = [((1 - self.tau) * lastLine2[i] - 2 * lastLine1[i]) / self.sigma for i in range(2, len(x) - 2)]
        koeffs = (0, b, c, (((1 - self.tau) * lastLine2[1] - 2 * lastLine1[1]) / self.sigma) - a * self.f0(t))
        A.insert(0, koeffs[:-1])
        w.insert(0, koeffs[-1])
        koeffs = (a, b, 0, (((1 - self.tau) * lastLine2[-2] - 2 * lastLine1[-2]) / self.sigma) - c * self.fl(t))
        A.append(koeffs[:-1])
        w.append(koeffs[-1])
        ans = self.SweepMethod(A, w)
        ans.insert(0, self.f0(t))
        ans.append(self.fl(t))
        return ans



def Error(x, y, z, f):
```

```
    ans = 0.0
    for i in range(len(z)):
        for j in range(len(z[i])):
            tmp = abs(z[i][j] - f(x[i][j], y[i][j]))
            ans = tmp if tmp > ans else ans
    return ans

def GetStepHandEror(solver, real_f):
    h = []
    e = []
    for N in range(5, 30, 2):
        x, y, z = solver(N, 100)
        h.append(solver.h)
        e.append(Error(x, y, z, real_f))
    return h, e

explict = ExplictSchema(T=1)
h, e = GetStepHandEror(explict, u)
plt.subplot(1,3,1)
plt.plot(h,e)
plt.xlabel("Step h")
plt.ylabel("e")

implict = ImplictSchema(T=1)
h, e = GetStepHandEror(implict, u)
plt.plot(h,e)


def GetTandError(methodToSolve, realF):
    tau, e = [], []
    for K in range(3, 90):
        x, y, z = methodToSolve(K=K)
        tau.append(methodToSolve.tau)
        e.append(Error(x, y, z, realF))
    return tau, e

explict = ExplictSchema(T=1)
tau, e = GetTandError(explict, u)
plt.subplot(1,3,2)
plt.plot(tau,e)
plt.xlabel("Step tau")
plt.ylabel("e")
```

```
implict = ImplictSchema(T=1)
tau, e = GetTandError(implict, u)
plt.subplot(1,3,3)
plt.plot(tau,e)
plt.xlabel("Step tau")
plt.ylabel("e")
fig = plt.gcf()
fig.canvas.manager.set_window_title('Зависимость ошибки от размера шага')
plt.show()
```

Лабораторная работа №7

1. Задача

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. В различные моменты времени вычислить погрешность численного решения путем сравнения результатов с приведенным в задании аналитическим решением. Исследовать зависимость погрешности от сеточных параметров.

Вариант 4:

Уравнение:

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

Граничные условия:

$$\begin{cases} u(0,\ y) = \phi_0(y) = \cos y \\ u(\frac{\pi}{2},\ y) = \phi_1(y) = 0 \\ u(x,\ 0) = \psi_0(x) = \cos x \\ u(x,\ \frac{\pi}{2}) = \psi_1(x) = 0 \end{cases}$$

Аналитическое решение:

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

2. Теория

<u>Конечно-разностная схема:</u>

Будем решать задачу на заданном промежутке от 0 до *lx* по координате *x* и на промежутке 0 от *ly* по координате *y*.

Рассмотрим конечно-разностную схему решения краевой задачи на сетке с граничными параметрами и параметрами насыщенности сетки *Nx Ny*.

Тогда размер шага по каждой из координат будет:

$$h_x = \frac{l_x}{N_x - 1},\ h_y = \frac{N_y}{N_y - 1}$$

Теперь давайте определим связь между дискретными значениями функции с помощью разностной аппроксимации производной:

$$\frac{\partial^2 u}{\partial x^2}(x_j, y_i) + \frac{\partial^2 u}{\partial y^2}(x_j, y_i) + 2u(x_j, y_i) = \frac{u_{j-1,i} - 2u_{j,i} + u_{j+1,i}}{h_x^2} + \frac{u_{j,i-1} - 2u_{j,i} + u_{j,i+1}}{h_y^2} + 2u_{j,i}$$

Теперь выразим $u_{i,j} = \frac{h_y^2(u_{j-1,i} + u_{j+1,i}) + h_x^2(u_{j,i-1} + u_{j,i+1})}{2(h_x^2 + h_y^2 - h_y^2 h_x^2)}$,

это будет основой для применения итерационных методов решения СЛАУ.

Для расчета $u_{j,0}$ и $u_{0,i}$ используем граничные условия.

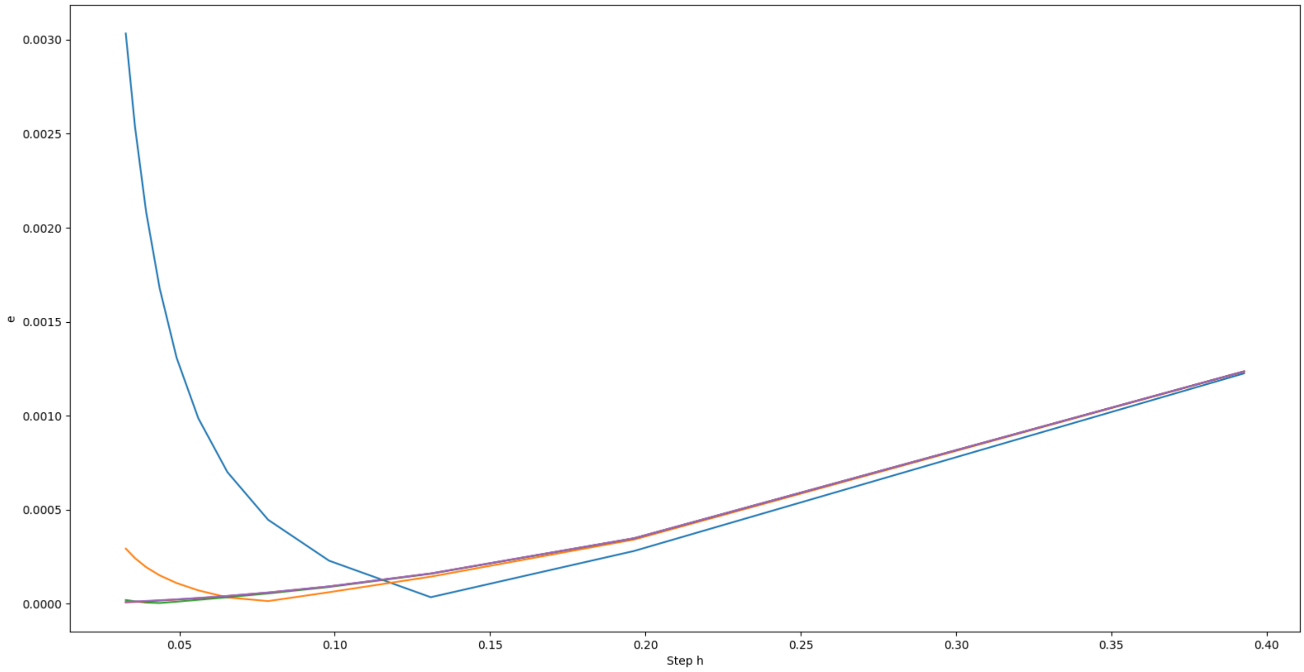
$$u_{j,i} = \frac{u(x_j, l_{y1}) - u(x_j, l_{y0})}{l_{y1} - l_{y0}} \cdot (y_i - l_{y0}) + u(x_j, l_{y0})$$

Для решения СЛАУ можно воспользоваться итерационными методами, такими как метод простых иттераций, метод Зейделя и метод верхних релаксаций. Первые два метода были изучены нами ранее, когда как последний является небольшой модификацией метода Зейделя с добавлением параметра w, который позволяет регулировать скорость сходимости метода. w = 1 соответствует методу Зейделя. Итерационный метод при w > 1 - метод верхней релаксации.

Вычисление погрешности приближенного решения по формуле:

$$MSE = \frac{\sum_{i=0}^{Ny} \sum_{j=0}^{Nx} (y_{ij} - \hat{y_{ij}})^2}{N_x \cdot N_y}$$

3. Результат



На графике представлены в синем, рыжем, зеленом, фиолетовом и красном цветах зависимости ошибок от размера шага h для epsilon равного 0.00001, 0.000001, 0.0000001, 0.00000001, 0.000000001 соответственно.

4. Вывод

Для выполнения данной лабораторной работы нужно было решить краевую задачу для дифференциального уравнения эллиптического типа. Также нужно было аппроксимировать уравнения с использованием центрально-разностной схемы. Для решения дискретного аналога пришлось вспомнить метод простых итераций (метод Либмана), метод Зейделя и применить новый для меня метод простых итераций с верхней релаксацией. Вычислил погрешности численного решения путем сравнения с приведенным в задании аналитическим решением.

5. Листинг кода

```
import matplotlib.pyplot as plt

import math

def psi_0(x):

    return math.cos(x)


def psi_1(x):

    return 0.0
```

```
def phi_0(y):
    return math.cos(y)
def phi_1(y):
    return 0.0
def u(x, y):
    return math.cos(y) * math.cos(x)
class Schema:
    def __init__(self, psi0=psi_0, psi1=psi_1, phi0=phi_0, phi1=phi_1,
                 lx0=0, lx1=math.pi / 2, ly0=0, ly1=math.pi / 2,
                 solver="zeidel", relax=0.1, epsilon=0.01):
        self.psi1 = psi1
        self.psi0 = psi0
        self.phi0 = phi0
        self.phi1 = phi1
        self.lx0 = lx0
        self.ly0 = ly0
        self.lx1 = lx1
        self.ly1 = ly1
        self.eps = epsilon
        self.method = None
        if solver == "zeidel":
            self.method = self.ZeidelStep
        elif solver == "simple":
            self.method = self.SimpleEulerStep
        elif solver == "relaxation":
            self.method = lambda x, y, m: self.RelaxationStep(x, y, m, relax)
        else:
            raise ValueError("Wrong solver name")
    def RelaxationStep(self, X, Y, M, w):
        norm = 0.0
        hx2 = self.hx * self.hx
        hy2 = self.hy * self.hy
        for i in range(1, self.Ny - 1):
            for j in range(1, self.Nx - 1):
```

```
            diff = hy2 * (M[i][j - 1] + M[i][j + 1])
            diff += hx2 * (M[i - 1][j] + M[i + 1][j])
            diff /= 2 * (hy2 + hx2 - hx2 * hy2)
            diff -= M[i][j]
            diff *= w
            M[i][j] += diff
            diff = abs(diff)
            norm = diff if diff > norm else norm
    return norm
def SimpleEulerStep(self, X, Y, M):
    tmp = [[0.0 for _ in range(self.Nx)] for _ in range(self.Ny)]
    norm = 0.0
    hx2 = self.hx * self.hx
    hy2 = self.hy * self.hy
    for i in range(1, self.Ny - 1):
        tmp[i][0] = M[i][0]
        for j in range(1, self.Nx - 1):
            tmp[i][j] = hy2 * (M[i][j - 1] + M[i][j + 1])
            tmp[i][j] += hx2 * (M[i - 1][j] + M[i + 1][j])
            tmp[i][j] /= 2 * (hy2 + hx2 - hx2 * hy2)
            diff = abs(tmp[i][j] - M[i][j])
            norm = diff if diff > norm else norm
        tmp[i][-1] = M[i][-1]
    for i in range(1, self.Ny - 1):
        M[i] = tmp[i]
    return norm
def ZeidelStep(self, X, Y, M):
    return self.RelaxationStep(X, Y, M, w=1)
def Set_l0_l1(self, lx0, lx1, ly0, ly1):
    self.lx0 = lx0
    self.lx1 = lx1
    self.ly0 = ly0
    self.ly1 = ly1
```

```
def CalculateH(self):
    self.hx = (self.lx1 - self.lx0) / (self.Nx - 1)
    self.hy = (self.ly1 - self.ly0) / (self.Ny - 1)
@staticmethod
def nparange(start, end, step=1):
    now = start
    e = 0.00000000001
    while now - e <= end:
        yield now
        now += step
def InitializeValues(self, X, Y):
    ans = [[0.0 for _ in range(self.Nx)] for _ in range(self.Ny)]
    for j in range(1, self.Nx - 1):
        coeff = (self.psi1(X[-1][j]) - self.psi0(X[0][j])) / (self.ly1 - self.ly0)
        addition = self.psi0(X[0][j])
        for i in range(self.Ny):
            ans[i][j] = coeff * (Y[i][j] - self.ly0) + addition
    for i in range(self.Ny):
        ans[i][0] = self.phi0(Y[i][0])
        ans[i][-1] = self.phi1(Y[i][-1])
    return ans
def __call__(self, Nx=10, Ny=10):
    self.Nx, self.Ny = Nx, Ny
    self.CalculateH()
    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)]
    ans = self.InitializeValues(X, Y)
    self.itters = 0
    while (self.method(X, Y, ans) >= self.eps):
        self.itters += 1
    return X, Y, ans
```

```
def Error(x, y, z, f):
    ans = 0.0
    for i in range(len(z)):
        for j in range(len(z[i])):
            ans += (z[i][j] - f(x[i][j], y[i][j]))**2
    return (ans / (len(z) * len(z[0])))**0.5
def GetStepHandError(solver, real_f):
    h , e = [], []
    for N in range(5, 50, 4):
        x, y, z = solver(N, N)
        h.append(solver.hx)
        e.append(Error(x, y, z, real_f))
    return h, e
schema = Schema(epsilon=0.001, solver="simple")
schema(10, 10)
print("Количество итераций метода простых итераций = {}".format(schema.itters))
schema = Schema(epsilon=0.001)
schema(10, 10)
print("Количество итераций метода Зейделя = {}".format(schema.itters))
schema = Schema(epsilon=0.001, solver="relaxation", relax=1.5)
schema(10, 10)
print("Количество итераций метода релаксаций = {}".format(schema.itters))
explict1 = Schema(epsilon=0.00001, solver="simple", relax=1.55)
h1, e1 = GetStepHandError(explict1, u)
explict2 = Schema(epsilon=0.000001, solver="simple", relax=1.55)
h2, e2 = GetStepHandError(explict2, u)
explict3 = Schema(epsilon=0.0000001, solver="simple", relax=1.55)
h3, e3 = GetStepHandError(explict3, u)
explict4 = Schema(epsilon=0.00000001, solver="simple", relax=1.55)
h4, e4 = GetStepHandError(explict4, u)
explict5 = Schema(epsilon=0.000000001, solver="simple", relax=1.55)
h5, e5 = GetStepHandError(explict5, u)
plt.plot(h1,e1)
plt.plot(h2,e2)
```

```
plt.plot(h3,e3)
plt.plot(h4,e4)
plt.plot(h5,e5)
plt.xlabel("Step h")
plt.ylabel("e")
fig = plt.gcf()
fig.canvas.manager.set_window_title('Зависимость ошибки от размера шага')
plt.show()
```

Лабораторная работа №8

1. Задача

Используя схемы переменных направлений и дробных шагов, решить двумерную начально-краевую задачу для дифференциального уравнения параболического типа. В различные моменты времени вычислить погрешность численного решения путем сравнения результатов с приведенным в задании аналитическим решением. Исследовать зависимость погрешности от сеточных параметров.

Вариант 4:

Уравнение:

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

Граничные условия:

$$\begin{cases} u(0, y,\ t) = \phi_0(y, t) = 0 \\ u(1, y,\ t) = \phi_1(y, t) = y\cos t \\ u(x, 0,\ t) = \psi_0(x, t) = 0 \\ u(x, 1,\ t) = \psi_1(x, t) = x\cos t \\ u(x, y, 0) = u_0(x, y) = xy \end{cases}$$

Аналитическое решение:

$$u(x, y, t) = xy\cos t$$

2. Теория

Конечно-разностная схема:

Рассмотрим конечно-разностную схему решения краевой задачи на сетке с граничными параметрами *lx*, *ly*, *T* и параметрами насыщенности сетки *Nx*, *Ny*, *K*.

Размер шага по каждой из координат определяется:

$$h_x = \frac{l_x}{N_x - 1},\ h_y = \frac{l_y}{N_y - 1},\ \tau = \frac{T}{K - 1}$$

Конечно-разностная схема решения параболического типа в сетке на временном слое t^{k+1} определяется с помощью 2-ух этапов, на каждом из которых решается трёхдиагональное уравнение с помощью метода прогонки:

Первый:

Считая, что значения функции $u_{i,j}^k = u(x_i, y_j, t^k)$а временном слое t^k известно, попробуем определить значения функции на временном слое $t^{k+\frac{1}{2}}$ путем разностной аппроксимации производной по времени:

$$\frac{\partial u}{\partial t}(x_i, y_j, t^k) = (1+\gamma)\frac{u_{i,j}^{k+\frac{1}{2}} - u_{i,j}^k}{\tau}$$

неявной аппроксимацией производной по *x*:

$$\frac{\partial^2 u}{\partial x^2}(x_i, y_j, t^k) = \frac{u_{i-1,j}^{k+\frac{1}{2}} - 2u_{i,j}^{k+\frac{1}{2}} + u_{i+1,j}^{k+\frac{1}{2}}}{h_x^2}$$

и явной аппроксимацией по *y*:

$$\frac{\partial^2 u}{\partial y^2}(x_i, y_j, t^k) = \frac{u_{i,j-1}^k - 2u_{i,j}^k + u_{i,j+1}^k}{h_y^2}$$

получаем уравнение:

$$(1+\gamma)\frac{u_{i,j}^{k+\frac{1}{2}} - u_{i,j}^k}{\tau} = a\frac{u_{i-1,j}^{k+\frac{1}{2}} - 2u_{i,j}^{k+\frac{1}{2}} + u_{i+1,j}^{k+\frac{1}{2}}}{h_x^2} + a\gamma\frac{u_{i,j-1}^k - 2u_{i,j}^k + u_{i,j+1}^k}{h_y^2}$$

$$-a\tau h_x^2\gamma u_{i,j-1}^k - ((1+\gamma)h_x^2h_y^2 - 2a\tau h_x^2\gamma)u_{i,j}^k - a\tau h_x^2\gamma u_{i,j+1}^k + x_i y_j \sin t^{k+\frac{1}{2}} =$$

$$a\tau h_y^2 u_{i-1,j}^{k+\frac{1}{2}} - (2a\tau h_y^2 + (1+\gamma)h_x^2h_y^2)u_{i,j}^{k+\frac{1}{2}} + a\tau h_y^2 u_{i+1,j}^{k+\frac{1}{2}}$$

Второй:

Известны значения функции $u_{i,j}^{k+\frac{1}{2}} = u(x_i, y_j, t^{k+\frac{1}{2}})$ на временном слое $t^{k+\frac{1}{2}}$ то, определим значения функции на временном слое t^{k+1} путем разностной аппроксимации производной по времени:

$$\frac{\partial u}{\partial t}(x_i, y_j, t^{k+\frac{1}{2}}) = (1+\gamma)\frac{u_{i,j}^{k+1} - u_{i,j}^{k+\frac{1}{2}}}{\tau}$$

явной аппроксимацией производной по *x*:

$$\frac{\partial^2 u}{\partial x^2}(x_i, y_j, t^{k+\frac{1}{2}}) = \frac{u_{i-1,j}^{k+\frac{1}{2}} - 2u_{i,j}^{k+\frac{1}{2}} + u_{i+1,j}^{k+\frac{1}{2}}}{h_x^2}$$

и неявной аппроксимацией по *y*:

$$\frac{\partial^2 u}{\partial y^2}(x_i, y_j, t^{k+\frac{1}{2}}) = \frac{u_{i,j-1}^{k+1} - 2u_{i,j}^{k+1} + u_{i,j+1}^{k+1}}{h_y^2}$$

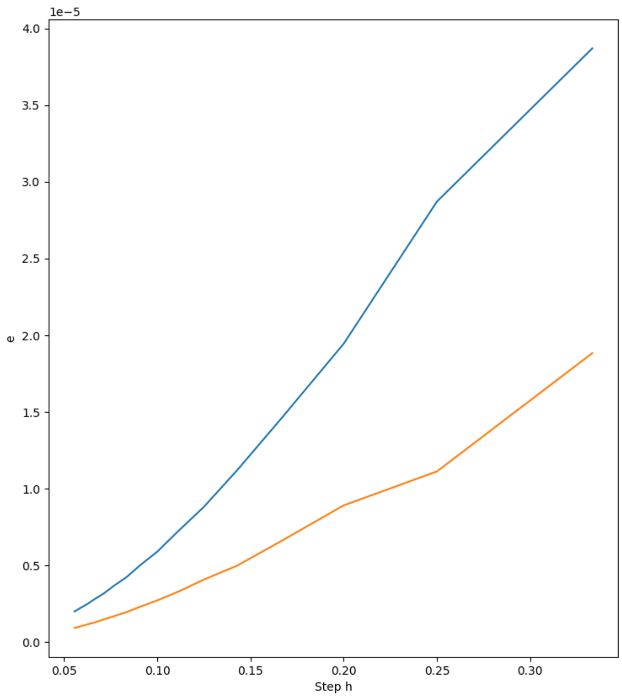
получим второе уравнение:

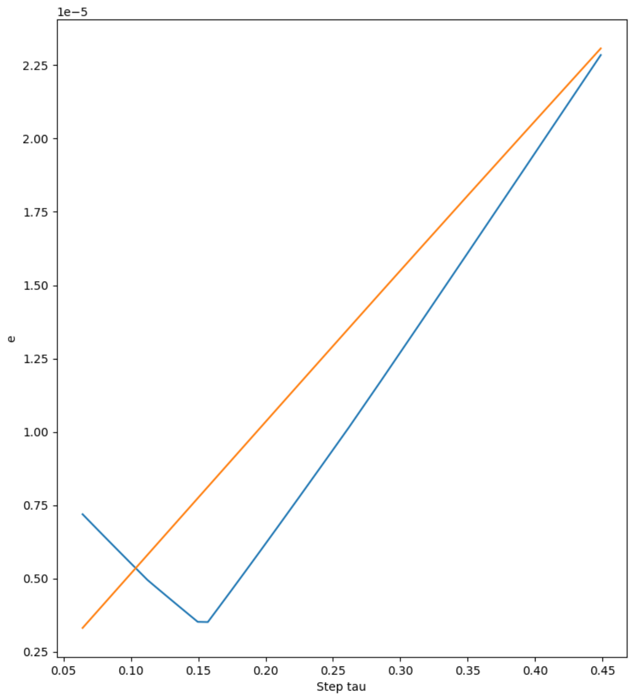
$$-a\tau h_y^2 \gamma u_{i-1,j}^{k+\frac{1}{2}} - ((1+\gamma)h_x^2 h_y^2 - 2a\tau h_y^2\gamma)u_{i,j}^{k+\frac{1}{2}} - a\tau h_y^2 \gamma u_{i+1,j}^{k+\frac{1}{2}} =$$
$$a\tau h_x^2 u_{i,j-1}^{k+1} - (2a\tau h_x^2 + (1+\gamma)h_x^2 h_y^2)u_{i,j}^{k+1} + a\tau h_x^2 u_{i,j+1}^{k+1}$$

$\gamma = 1$ - метод переменных направлений,

$\gamma = 0$ - метод дробных шагов.

3. Результат





На первом графике отображены синим и рыжим зависимости ошибки от шага для методов дробных шагов и переменных направлений соответственно.

На втором графике отображены синим и рыжим зависимости ошибки от шага по времени для методов дробных шагов и переменных направлений соответственно.

4. Вывод

Используя схемы переменных направлений и дробных шагов, научился решать двумерную начально-краевую задачу для дифференциального уравнения параболического типа. Вычислил погрешности в различные моменты времени и исследовал зависимость погрешность от различных параметров.

5. Листинг кода

```
import random

import matplotlib.pyplot as plt

import math

import numpy as np

def psi_0(x, t):

    return 0.0

# need to approx

def psi_1(x, t):

    return x * math.cos(t)
```

```
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
# analytic solve
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)
@staticmethod
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
@staticmethod
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):
        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 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
first = Schema(T = 2*math.pi, order2nd = False) #метод дробных шагов
second = Schema(T = 2*math.pi, order2nd = True) #метод переменных направлений
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]))
    return h, e
TSTEPS = 100
time = random.randint(0, TSTEPS - 1)
```

```
h1, e1 = GetGraphicH(first, time, TSTEPS)
h2, e2 = GetGraphicH(second, time, TSTEPS)
plt.subplot(1,2,1)
plt.plot(h1,e1)
plt.plot(h2,e2)
plt.xlabel("Step h")
plt.ylabel("e")
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)
plt.subplot(1,2,2)
plt.plot(tau1,e1)
plt.plot(tau2,e2)
plt.xlabel("Step tau")
plt.ylabel("e")
fig = plt.gcf()
fig.canvas.manager.set_window_title('Зависимость ошибки от размера шага')
plt.show()
```