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

Студент группы 8О-406: Полюбин А.И. Работа выполнена: 26.12.2022

Преподаватель: Пивоваров Д.Е. Отчет сдан: 4.01.2023

Итоговая оценка:

Подпись преподавателя:

Москва 2023

**1. Тема работы**

МЕТОД КОНЕЧНЫХ РАЗНОСТЕЙ РЕШЕНИЯ МНОГОМЕРНЫХ ЗАДАЧ МАТЕМАТИЧЕСКОЙ ФИЗИКИ. МЕТОДЫ РАСЩЕПЛЕНИЯ

**2. Цель работы**

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

Вариант №4:

, ,

.

Аналитическое решение: .

**3. Ход выполнения работы**

В данной лабораторной реализованы метод переменных направлений и метод дробных шагов. Реализованы функции вычисления погрешности. Графики выводятся при помощи библиотеки matplotlib.

**4. Листинг**

import random

import matplotlib.pyplot as plt

import sys

import warnings

import numpy as np

from functools import reduce

from mpl\_toolkits.mplot3d import Axes3D

#Граничные условия

def psi\_0(x, t, a = 1):

return np.cos(2\*x) \* np.exp(-3\*a\*t)

def psi\_1(x, t, a = 1):

return 3 \* np.cos(2\*x) \* np.exp(-3\*a\*t) / 4

def phi\_0(y, t, a = 1):

return np.cosh(y) \* np.exp(-3\*a\*t)

def phi\_1(y, t, a = 1):

return 0

def u0(x, y, a = 1):

return np.cos(2\*x) \* np.cosh(y)

#Истинное решение

def u(x, y, t, a = 1):

return np.cos(2\*x) \* np.cosh(y) \* np.exp(-3\*a\*t)

class Schema:

def \_\_init\_\_(self, a = 1, rho = u0, psi0 = psi\_0, psi1 = psi\_1, phi0 = phi\_0, phi1 = phi\_1,

lx0 = 0, lx1 = np.pi/4, ly0 = 0, ly1 = np.log(2), T = 5, 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.a = a

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 compute\_h(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 compute\_tau(self):

self.tau = self.T / (self.K - 1)

@staticmethod

def three\_diagonal(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 compute\_left\_edge(self, X, Y, t, square):

for i in range(self.Ny):

square[i][0] = self.phi0(Y[i][0], t, self.a)

def compute\_right\_edge(self, X, Y, t, square):

for i in range(self.Ny):

square[i][-1] = self.phi1(Y[i][-1], t, self.a)

def compute\_bottom\_edge(self, X, Y, t, square):

for j in range(1, self.Nx - 1):

square[0][j] = self.psi0(X[0][j], t, self.a)

def compute\_line\_first\_step(self, i, X, Y, 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] - 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]

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] - c\*now\_square[i][-1]

)

line = self.three\_diagonal(A, w)

for j in range(1, self.Nx - 1):

now\_square[i][j] = line[j - 1]

def compute\_line\_second\_step(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] - c\*now\_square[0][j]

]

A.extend([(c, b, c) for \_ in range(2, self.Ny - 1)])

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]

for i in range(2, self.Ny - 1)

])

koeffs = self.implict\_top\_approx(j, X, Y, t, now\_square, last\_square)

A.append(koeffs[:-1])

w.append(koeffs[-1])

line = self.three\_diagonal(A, w)

for i in range(1, self.Ny):

now\_square[i][j] = line[i - 1]

def explict\_top\_approx(self, X, Y, t, square):

for j in range(1, self.Nx - 1):

square[-1][j] = 2\*self.hy\*self.psi1(X[-1][j], t, self.a)

square[-1][j] += 4\*square[-2][j] - square[-3][j]

square[-1][j] /= 3

def implict\_top\_approx(self, j, X, Y, t, square, last\_square):

hx2 = self.hx2

hy2 = self.hy2

c = 2 \* self.a \* self.tau \* hx2

b = -(c + (1 + self.order)\*hx2\*hy2)

w = -self.cx\*self.order\*last\_square[-1][j - 1]

w -= ((self.order + 1)\*hx2\*hy2 - 2\*self.cx\*self.order)\*last\_square[-1][j]

w -= self.cx\*self.order\*last\_square[-1][j + 1]

w -= c\*self.hy\*self.psi1(X[-1][j], t, self.a)

return (c, b, 0, w)

def explict\_top\_approx\_0(self, X, Y, t, square):

for j in range(1, self.Nx - 1):

square[-1][j] = self.hy\*self.psi1(X[-1][j], t, self.a)

square[-1][j] += square[-2][j]

def implict\_top\_approx\_0(self, j, X, Y, t, square, last\_square):

return (-1, 1, 0, self.hy\*self.psi1(X[-1][j], t, self.a))

def compute\_square(self, X, Y, t, last\_square):

square = [[0.0 for \_ in range(self.Nx)] for \_ in range(self.Ny)]

self.compute\_left\_edge(X, Y, t - 0.5\*self.tau, square)

self.compute\_right\_edge(X, Y, t - 0.5\*self.tau, square)

self.compute\_bottom\_edge(X, Y, t - 0.5\*self.tau, square)

for i in range(1, self.Ny - 1):

self.compute\_line\_first\_step(i, X, Y, last\_square, square)

self.explict\_top\_approx(X, Y, t - 0.5\*self.tau, square)

last\_square = square

square = [[0.0 for \_ in range(self.Nx)] for \_ in range(self.Ny)]

self.compute\_left\_edge(X, Y, t, square)

self.compute\_right\_edge(X, Y, t, square)

self.compute\_bottom\_edge(X, Y, t, square)

for j in range(1, self.Nx - 1):

self.compute\_line\_second\_step(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], self.a)

return first

def \_\_call\_\_(self, Nx=20, Ny=20, K=20):

self.Nx, self.Ny, self.K = Nx, Ny, K

self.compute\_tau()

self.compute\_h()

self.bx = -2\*self.a\*self.tau\*self.hy2

self.bx -= (1 + self.order)\*self.hx2\*self.hy2

self.cx = self.a \* self.tau \* self.hy2

self.cy = self.a \* self.tau \* self.hx2

self.by = -2\*self.a\*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.compute\_square(X, Y, t, ans[-1]))

taus.append(t)

return X, Y, taus, ans

#Истинное значение функции в данный момент времени

def real\_z\_by\_time(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 epsilon(X, Y, t, z, ut = u, a = 1):

ans = 0.0

for i in range(len(z)):

for j in range(len(z[i])):

ans += (ut(X[i][j], Y[i][j], t, a) - z[i][j])\*\*2

return (ans / len(z) / len(z[0]))\*\*0.5

def plot\_by\_time( nx=15, ny=15, k=50, t=1, a=1):

schema = Schema(T = t, a = a, order2nd = False)

xx, yy, tt, zz = schema(Nx = nx, Ny = ny, K = k)

z = zz[t]

extrems = search\_minmax(zz)

fig = plt.figure(num=1, figsize=(20, 13), clear=True)

ax = fig.add\_subplot(1, 1, 1, projection='3d')

ax.plot\_surface(np.array(xx), np.array(yy), np.array(z))

ax.plot\_wireframe(\*real\_z\_by\_time(0, np.pi/4, 0, np.log(2), t, u), color="red")

ax.set\_xlabel('x')

ax.set\_ylabel('y')

ax.set\_zlabel('z')

ax.set\_zlim(extrems[0], extrems[1])

fig.tight\_layout()

plt.show()

def search\_minmax(zz):

z = zz[0]

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

plot\_by\_time()

first = Schema(T = 1, order2nd = False) #метод дробных шагов

second = Schema(T = 1, order2nd = True) #метод переменных направлений

def graphic\_h(solver, time = 0, tsteps = 400):

h = []

e = []

for N in range(4, 35, 1):

x, y, t, z = solver(Nx = N, Ny = N, K = tsteps)

h.append(solver.hx)

e.append(epsilon(x, y, t[time], z[time]))

return h, e

TSTEPS = 100

time = random.randint(0, TSTEPS - 1)

plt.figure(figsize = (16, 10))

plt.title("Зависимость погрешности от длины шага при t = " + str(time / TSTEPS))

plt.subplot(2, 1, 1)

h1, e1 = graphic\_h(first, time, TSTEPS)

h2, e2 = graphic\_h(second, time, TSTEPS)

plt.plot(h1, e1, label="Метод дробных шагов")

plt.plot(h2, e2, label="Метод переменных направлений")

plt.xlabel("$h\_x$")

plt.ylabel("$\epsilon$")

plt.legend()

plt.grid()

plt.subplot(2, 1, 2)

plt.plot(list(map(np.log, h1)), list(map(np.log, e1)), label="Метод дробных шагов")

plt.plot(list(map(np.log, h2)), list(map(np.log, e2)), label="Метод переменных направлений")

plt.xlabel("$\log{h\_x}$")

plt.ylabel("$\log{\epsilon}$")

plt.legend()

plt.grid()

def graphic\_tau(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(epsilon(x, y, t[time], z[time]))

return tau, e

plt.figure(figsize = (16, 10))

plt.title("Зависимость погрешности от длины шага по времени")

plt.subplot(2, 1, 1)

tau1, e1 = graphic\_tau(first)

tau2, e2 = graphic\_tau(second)

plt.plot(tau1, e1, label="Метод дробных шагов")

plt.plot(tau2, e2, label="Метод переменных направлений")

plt.xlabel("$\tau$")

plt.ylabel("$\epsilon$")

plt.legend()

plt.grid()

plt.subplot(2, 1, 2)

plt.plot(list(map(np.log, tau1)), list(map(np.log, e1)), label="Метод дробных шагов")

plt.plot(list(map(np.log, tau2)), list(map(np.log, e2)), label="Метод переменных направлений")

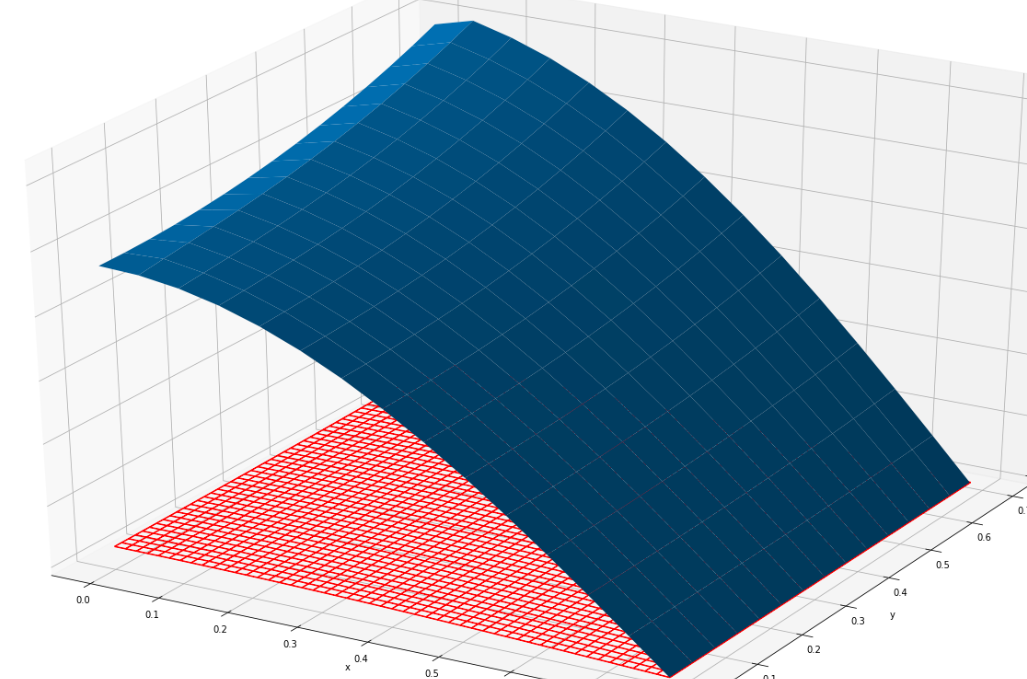
plt.xlabel("$\log{\tau}$")

plt.ylabel("$\log{\epsilon}$")

plt.legend()

plt.grid()

**5. Результаты**

****

**6. Выводы**

При выполнении данной лабораторной работы я научился применять методы конечных разностей для решения многомерных задач математической физики.