**Московский авиационный институт**

**(национальный исследовательский университет)**

**Институт информационных технологий и прикладной математики**

**Кафедра вычислительной математики и программирования**

**Лабораторная работа №6**

«Численное решение уравнений с частными производными гиперболического типа»

Вариант №3

Студент: Гордионок Е.А.

Группа: М8О-409Б-19

Руководитель: Пивоваров Д.Е.

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: 09.12.2022

**Москва 2022**

**Лабораторная работа №6**

Метод конечных разностей для решения задачи гиперболического типа

**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 .

**Описание метода**

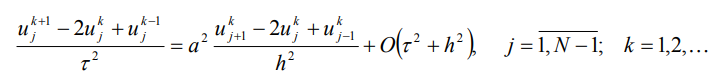
Рассмотрим общий вид уравнения гиперболического типа для третьей начально-краевой задачи:

Изображение выглядит как текст

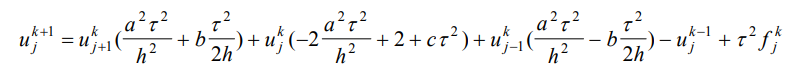
Автоматически созданное описание

Для решения такой задачи применяют метод конечных разностей.

Если аппроксимировать вторую производную по пространству на нижнем временном слое, то получим **явную** конечно-разностную схему:

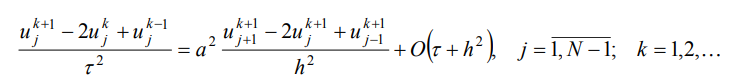


где для каждого j-го уравнения неизвестна только одна величина :



Для устойчивости данной схемы накладывается ограничение на

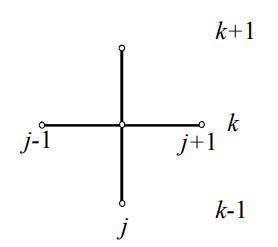
Если аппроксимировать производную по пространству на верхнем временном слое, то получим **неявную** конечно-разностную схему:



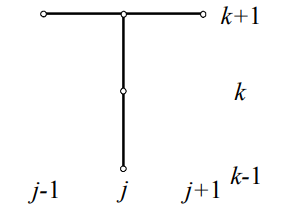
где для нахождения сеточной функции на верхнем временном слое необходимо решать СЛАУ с трёхдиагональной матрицей с помощью метода прогонки.

Шаблоны данных двух схем:

1) Явной схемы



2) Неявной схемы



**Аппроксимация начальных и граничных условий**

**Начальные условия**

В обеих схемах необходимо знать значения на нижних временных слоях. Для k = 1 имеем:

Изображение выглядит как текст

Автоматически созданное описание

Для определения можем воспользоваться аппроксимацией второго начального условия. Тогда получим:

Изображение выглядит как часы

Автоматически созданное описание

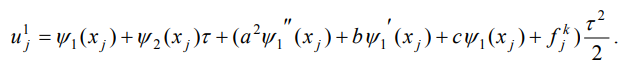


Чтобы повысить порядок аппроксимации второго начального условия раскладывают в ряд Тейлора в окрестности t = 0:

Изображение выглядит как текст, часы

Автоматически созданное описание

Согласно исходному уравнению, получают следующую аппроксимацию:



**Граничные условия**

Рассмотрим 3 способа аппроксимации граничных условий 2Т1П, 3Т2П, 2Т2П.

Двухточечная, с первым порядком:



Трёхточечная, со вторым порядком:

Изображение выглядит как текст

Автоматически созданное описание

При использовании неявной и явно-неявной схемы СЛАУ теряет трёхдиагональность, поэтому сначала необходимо привести СЛАУ к трёхдиагональному виду линейной комбинацией первой строки со второй, предпоследней с последней.

Двухточечная, со вторым порядком:

Для того, чтобы получить эти формулы для начала раскладывают в ряд Тейлора в окрестности x = 0 и в окрестности x = l:

Изображение выглядит как текст

Автоматически созданное описание

Далее берут информацию из исходного уравнения, выражая оттуда вторую производную и подставляя это выражение и получают выражение для первой производной.

**Вариант**

**Text

Description automatically generated**

**Результаты работы программы**

Chart

Description automatically generated

**Выводы**

В данной работе реализована явная и неявная конечно-разностные схемы для решения начально-краевой задачи для дифференциального уравнения гиперболического типа. Осуществлена реализацию трех вариантов аппроксимации граничных условий, содержащих производные: двухточечная аппроксимация с первым порядком, трехточечная аппроксимация со вторым порядком, двухточечная аппроксимация со вторым порядком; и двух вариантов начальных условий: аппроксимация с первым и со вторым порядком второго начального условия.

**Приложение. Листинг программы.**

import numpy as np

import matplotlib.pyplot as plt

import math

class Data:

def \_\_init\_\_(self, params):

self.a = params['a']

self.b = params['b']

self.c = params['c']

self.d = params['d']

self.l = params['l']

self.f = params['f']

self.alpha = params['alpha']

self.beta = params['beta']

self.gamma = params['gamma']

self.delta = params['delta']

self.psi1 = params['psi1']

self.psi2 = params['psi2']

self.psi1\_dir1 = params['psi1\_dir1']

self.psi1\_dir2 = params['psi1\_dir2']

self.phi0 = params['phi0']

self.phi1 = params['phi1']

self.bound\_type = params['bound\_type']

self.approximation = params['approximation']

self.solution = params['solution']

class HyperbolicSolver:

def \_\_init\_\_(self, params, N, T, K):

self.data = Data(params)

self.N = N

self.T = T

self.K = K

self.h = self.data.l / N

self.tau = T / K

self.sigma = (self.tau \*\* 2) / (self.h \*\* 2)

def analyticSolve(self):

N = self.N

T = self.T

K = self.K

u = np.zeros((K, N))

for k in range(K):

for j in range(N):

u[k][j] = self.data.solution(j \* self.h, k \* self.tau)

return u

def calculate(self, N, K):

u = np.zeros((K, N))

for j in range(0, N - 1):

x = j \* self.h

u[0][j] = self.data.psi1(x)

if self.data.approximation == 'p1':

u[1][j] = self.data.psi1(x) + self.data.psi2(x) \* self.tau + self.data.psi1\_dir2(x) \* \

(self.tau \*\* 2 / 2)

elif self.data.approximation == 'p2':

u[1][j] = self.data.psi1(x) + self.data.psi2(x) \* self.tau + \

(self.data.psi1\_dir2(x) + self.data.b \* self.data.psi1\_dir1(x) +

self.data.c \* self.data.psi1(x) + self.data.f()) \* (self.tau \*\* 2 / 2)

return u

def implicit\_solver(self):

N = self.N

T = self.T

K = self.K

u = self.calculate(N, K)

a = np.zeros(N)

b = np.zeros(N)

c = np.zeros(N)

d = np.zeros(N)

for k in range(2, K):

for j in range(1, N):

a[j] = self.sigma

b[j] = -(1 + 2 \* self.sigma)

c[j] = self.sigma

d[j] = -2 \* u[k - 1][j] + u[k - 2][j]

if self.data.bound\_type == 'a1p2':

b[0] = -1 / self.h

c[0] = 1 / self.h

d[0] = 1 / (self.data.beta - self.data.alpha / self.h) \* self.data.phi0(k \* self.tau)

a[-1] = -self.data.gamma / self.h / (self.data.delta + self.data.gamma / self.h)

b[-1] = 1

d[-1] = 1 / (self.data.delta + self.data.gamma / self.h) \* self.data.phi1(k \* self.tau)

elif self.data.bound\_type == 'a2p3':

k1 = 2 \* self.h \* self.data.beta - 3 \* self.data.alpha

omega = self.tau \*\* 2 \* self.data.b / (2 \* self.h)

xi = self.data.d \* self.tau / 2

b[0] = 4 \* self.data.alpha - self.data.alpha / (self.sigma + omega) \* \

(1 + xi + 2 \* self.sigma - self.data.c \* self.tau \*\* 2)

c[0] = k1 - self.data.alpha \* (omega - self.sigma) / (omega + self.sigma)

d[0] = 2 \* self.h \* self.data.phi0(k \* self.tau) + self.data.alpha \* d[1] / (-self.sigma - omega)

a[-1] = -self.data.gamma / (omega - self.sigma) \* \

(1 + xi + 2 \* self.sigma - self.data.c \* self.tau \*\* 2) - 4 \* self.data.gamma

d[-1] = 2 \* self.h \* self.data.phi1(k \* self.tau) - self.data.gamma \* d[-2] / (omega - self.sigma)

elif self.data.bound\_type == 'a2p2':

b[0] = 2 \* self.data.a / self.h

c[0] = -2 \* self.data.a / self.h + self.h / self.tau \*\* 2 - self.data.c \* self.h + \

-self.data.d \* self.h / (2 \* self.tau) + \

self.data.beta / self.data.alpha \* (2 \* self.data.a + self.data.b \* self.h)

d[0] = self.h / self.tau \*\* 2 \* (u[k - 2][0] - 2 \* u[k - 1][0]) - self.h \* self.data.f() + \

-self.data.d \* self.h / (2 \* self.tau) \* u[k - 2][0] + \

(2 \* self.data.a - self.data.b \* self.h) / self.data.alpha \* self.data.phi0(k \* self.tau)

a[-1] = -b[0]

d[-1] = self.h / self.tau \*\* 2 \* (-u[k - 2][0] + 2 \* u[k - 1][0]) + self.h \* self.data.f() + \

self.data.d \* self.h / (2 \* self.tau) \* u[k - 2][0] + \

(2 \* self.data.a + self.data.b \* self.h) / self.data.alpha \* self.data.phi1(k \* self.tau)

u[k] = self.progonka(a, b, c, d)

return u

def \_left\_bound\_a1p2(self, u, k, t):

coeff = self.data.alpha / self.h

return (-coeff \* u[k - 1][1] + self.data.phi0(t)) / (self.data.beta - coeff)

def \_right\_bound\_a1p2(self, u, k, t):

coeff = self.data.gamma / self.h

return (coeff \* u[k - 1][-2] + self.data.phi1(t)) / (self.data.delta + coeff)

def \_left\_bound\_a2p2(self, u, k, t):

n = self.data.c \* self.h - 2 \* self.data.a / self.h - self.h / self.tau \*\* 2 - self.data.d \* self.h / \

(2 \* self.tau) + self.data.beta / self.data.alpha \* (2 \* self.data.a - self.data.b \* self.h)

return 1 / n \* (- 2 \* self.data.a / self.h \* u[k][1] +

self.h / self.tau \*\* 2 \* (u[k - 2][0] - 2 \* u[k - 1][0]) +

-self.data.d \* self.h / (2 \* self.tau) \* u[k - 2][0] + -self.h \* self.data.f() +

(2 \* self.data.a - self.data.b \* self.h) / self.data.alpha \* self.data.phi0(t))

def \_right\_bound\_a2p2(self, u, k, t):

n = -self.data.c \* self.h + 2 \* self.data.a / self.h + self.h / self.tau \*\* 2 + self.data.d \* self.h / \

(2 \* self.tau) + self.data.delta / self.data.gamma \* (2 \* self.data.a + self.data.b \* self.h)

return 1 / n \* (2 \* self.data.a / self.h \* u[k][-2] +

self.h / self.tau \*\* 2 \* (2 \* u[k - 1][-1] - u[k - 2][-1]) +

self.data.d \* self.h / (2 \* self.tau) \* u[k - 2][-1] + self.h \* self.data.f() +

(2 \* self.data.a + self.data.b \* self.h) / self.data.gamma \* self.data.phi1(t))

def \_left\_bound\_a2p3(self, u, k, t):

denom = 2 \* self.h \* self.data.beta - 3 \* self.data.alpha

return self.data.alpha / denom \* u[k - 1][2] - 4 \* self.data.alpha / denom \* u[k - 1][1] + \

2 \* self.h / denom \* self.data.phi0(t)

def \_right\_bound\_a2p3(self, u, k, t):

denom = 2 \* self.h \* self.data.delta + 3 \* self.data.gamma

return 4 \* self.data.gamma / denom \* u[k - 1][-2] - self.data.gamma / denom \* u[k - 1][-3] + \

2 \* self.h / denom \* self.data.phi1(t)

def explicit\_solver(self):

global left\_bound, right\_bound

N = self.N

T = self.T

K = self.K

u = self.calculate(N, K)

# for j in range(1, N - 1):

# u[1][j] = self.data.ps1()

if self.data.bound\_type == 'a1p2':

left\_bound = self.\_left\_bound\_a1p2

right\_bound = self.\_right\_bound\_a1p2

elif self.data.bound\_type == 'a2p2':

left\_bound = self.\_left\_bound\_a2p2

right\_bound = self.\_right\_bound\_a2p2

elif self.data.bound\_type == 'a2p3':

left\_bound = self.\_left\_bound\_a2p3

right\_bound = self.\_right\_bound\_a2p3

for k in range(2, K):

t = k \* self.tau

for j in range(1, N - 1):

# u[k][j] = self.sigma \* u[k - 1][j + 1] + (2 - 2 \* self.sigma) \* u[k - 1][j] + \

# self.sigma \* u[k - 1][j - 1] - u[k - 2][j]

quadr = self.tau \*\* 2

tmp1 = self.sigma + self.data.b \* quadr / (2 \* self.h)

tmp2 = self.sigma - self.data.b \* quadr / (2 \* self.h)

u[k][j] = u[k - 1][j + 1] \* tmp1 + \

u[k - 1][j] \* (-2 \* self.sigma + 2 + self.data.c \* quadr) + \

u[k - 1][j - 1] \* tmp2 - u[k - 2][j] + quadr \* self.data.f()

u[k][0] = left\_bound(u, k, t)

u[k][-1] = right\_bound(u, k, t)

return u

def progonka(self, a, b, c, d):

# print('a', a, 'b', b, 'c', c, 'd', d, sep='\n')

n = len(a)

for i in range(1, n):

if math.fabs(b[i]) < math.fabs(a[i]) + math.fabs(c[i]):

raise Exception(f"{math.fabs(b[i])} < {math.fabs(a[i]) + math.fabs(c[i])}, a[{i}]={a[i]}, b[{i}]={b[i]}, c[{i}]={c[i]}")

# Формирование массивов P, Q (Расчет значений) ((Прямой ход))

P, Q = [-c[0] / b[0]], [d[0] / b[0]]

for i in range(1, n):

P.append(-c[i] / (b[i] + a[i] \* P[i - 1]))

Q.append((d[i] - a[i] \* Q[i - 1]) / (b[i] + a[i] \* P[i - 1]))

# Вычисление решения системы (Обратный ход)

x = [Q[n - 1]]

for i in range(1, n):

x.append(P[n - 1 - i] \* x[i - 1] + Q[n - 1 - i])

# print('result', np.array([i for i in reversed(x)]))

return np.array([i for i in reversed(x)])

params = {

'a': 1,

'b': 0,

'c': -3,

'd': 0,

'l': np.pi,

'f': lambda: 0,

'alpha': 0,

'beta': 1,

'gamma': 0,

'delta': 1,

'psi1': lambda x: 0,

'psi2': lambda x: 2 \* np.cos(x),

'psi1\_dir1': lambda x: 0,

'psi1\_dir2': lambda x: -2 \* np.sin(x),

'phi0': lambda t: np.sin(2 \* t),

'phi1': lambda t: -np.sin(2 \* t),

'bound\_type': 'a1p2',

'approximation': 'p2',

'solution': lambda x, t: np.cos(x) \* np.sin(2 \* t),

}

#%%

N = 50

K = 100

T = 1

#%%

solver = HyperbolicSolver(params, N, T, K)

#%%

X = np.arange(0, np.pi / 2, np.pi / 2 / N)

T = np.arange(0, T, T / K)

#%%

fig = plt.figure(figsize=(20, 12))

ax = fig.add\_subplot(1, 4, 1, projection='3d')

ax.set\_title('Точное решение')

U = solver.analyticSolve()

W, Q= np.meshgrid(X, T)

ax.plot\_surface(W, Q, np.array(U))

ax.set\_xlabel('x Label')

ax.set\_ylabel('t Label')

ax.set\_zlabel('u Label')

ax = fig.add\_subplot(1, 4, 2, projection='3d')

ax.set\_title('Явная схема')

# \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

u = solver.explicit\_solver()

W, Q = np.meshgrid(X, T)

ax.plot\_surface(W, Q, np.array(u))

ax.set\_xlabel('x Label')

ax.set\_ylabel('t Label')

ax.set\_zlabel('u Label')

print('Явная схема: средкв ошибка:',

math.sqrt(sum([sum([(U[i][j] - u[i][j])\*\*2 for j in range(len(u[0]))]) for i in range(len(u))])))

# \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

ax = fig.add\_subplot(1, 4, 3, projection='3d')

ax.set\_title('Неявная схема')

u = solver.implicit\_solver()

W, Q = np.meshgrid(X, T)

ax.plot\_surface(W, Q, np.array(u))

ax.set\_xlabel('x Label')

ax.set\_ylabel('t Label')

ax.set\_zlabel('u Label')

print('Неявная схема: средкв ошибка:',

math.sqrt(sum([sum([(U[i][j] - u[i][j])\*\*2 for j in range(len(u[0]))]) for i in range(len(u))])))