Московский Авиационный Институт

(национальный исследовательский университет)

**Факультет Компьютерных наук и прикладной математики**

Кафедра Вычислительной математики и программирования

**Лабораторные работы**

**по дисциплине**

**«Численные методы»**

**IV курс, VII семестр**

Студент: Синюков А.С.

Группа: М8О-406Б-21

Руководитель: Ревизников Д. Л.

Оглавление

# Лабораторная работа №**5**

## **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.

## **Код**:

Программа реализована на языке программирования Python.

### **main**.py:

from abc import ABC, abstractmethod

import sys

import math

import numpy as np

import matplotlib.pyplot as plt

def tridiagonal\_solve(a, b, c, d) -> np.ndarray:

    n = len(d)

    p = np.ndarray(n, dtype=float)

    q = np.ndarray(n, dtype=float)

    x = np.ndarray(n, dtype=float)

    p[0] = -c[0] / b[0]

    q[0] = d[0] / b[0]

    for i in range(1, n):

        p[i] = -c[i] / (b[i] + a[i]\*p[i-1])

        q[i] = (d[i] - a[i]\*q[i-1]) / (b[i] + a[i]\*p[i-1])

    x[-1] = q[-1]

    for i in range(n-2, -1, -1):

        x[i] = p[i] \* x[i+1] + q[i]

    return x

class Diffur:

    @staticmethod

    def g(x, t): return 0.5 \* np.exp(-0.5 \* t) \* np.sin(x)

    @staticmethod

    def phi\_zero(t): return np.exp(-0.5 \* t)

    @staticmethod

    def phi\_l(t): return -1 \* np.exp(-0.5 \* t)

    @staticmethod

    def psi(x): return np.sin(x)

    def \_\_init\_\_(self):

        pass

class AbstractSolver(ABC):

    D: Diffur

    T: float

    L: float

    N: float

    K: float

    tau: float

    h: float

    sigma: float

    @staticmethod

    def calc\_tau(T: float, K: float) -> float:

        return T / (K-1)

    @staticmethod

    def calc\_h(L: float, N: float) -> float:

        return L / (N-1)

    @staticmethod

    def calc\_sigma(tau: float, h: float) -> float:

        return tau / h\*\*2

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

        self.D = Diffur()

        self.T = T

        self.L = L

        self.N = N

        self.K = K

        self.tau = self.calc\_tau(T, K)

        self.h = self.calc\_h(L, N)

        self.sigma = self.calc\_sigma(self.tau, self.h)

    @abstractmethod

    def solve(self): pass

    def change\_params(self, T, L, K, N):

        self.T = T

        self.L = L

        self.N = N

        self.K = K

        self.tau = self.calc\_tau(T, K)

        self.h = self.calc\_h(L, N)

        self.sigma = self.calc\_sigma(self.tau, self.h)

class SolveExact(AbstractSolver):

    @staticmethod

    def exact\_solve(x, t):

        return np.exp(-0.5 \* t) \* np.sin(x)

    def solve(self):

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

        for i in range(self.K):

            for j in range(self.N):

                u[i][j] = self.exact\_solve(j \* self.h, i \* self.tau)

        return u

class SolveExplicit(AbstractSolver):

    aprox = 0

    """

    тут и далее

    0 - двухточечная аппроксимация с первым порядком

    1 - трехточечная аппроксимация со вторым порядком

    2 - двухточечная аппроксимация со вторым порядком

    """

    def add\_aprox(self, num: int):

        self.aprox = num

        return self

    def zero\_aprox(self):

        if self.aprox == 0:

            return lambda i, curr, prev: curr[1] - self.h \* self.D.phi\_zero(i \* self.tau)

        elif self.aprox == 1:

            return lambda i, curr, prev: (-1 / 3) \* (2 \* self.h \* self.D.phi\_zero(i \* self.tau) + curr[2] - 4 \* curr[1])

        elif self.aprox == 2:

            return lambda i, curr, prev: -2 \* self.sigma \* self.h \* self.D.phi\_zero((i-1) \* self.tau) + 2 \* self.sigma \* prev[1] + (1 - 2 \* self.sigma) \* prev[0] + self.tau \* self.D.g(0, (i-1) \* self.tau)

    def l\_aprox(self):

        if self.aprox == 0:

            return lambda i, curr, prev: self.h \* self.D.phi\_l(i \* self.tau) + curr[-2]

        elif self.aprox == 1:

            return lambda i, curr, prev: (1 / 3) \* (2 \* self.h \* self.D.phi\_l(i \* self.tau) + 4 \* curr[-2] - curr[-3])

        elif self.aprox == 2:

            return lambda i, curr, prev: 2 \* self.sigma \* self.h \* self.D.phi\_l((i-1) \* self.tau) + 2 \* self.sigma \* prev[-2] + (1 - 2 \* self.sigma) \* prev[-1] + self.tau \* self.D.g((self.N-1) \* self.h, (i-1) \* self.tau)

    def solve(self):

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

        u\_zero = []

        for i in [i \* self.h for i in range(self.N)]:

            u\_zero.append(self.D.psi(i))

        u[0] = np.array(u\_zero)

        for i in range(1, self.K):

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

                u[i][j] = (self.sigma \* u[i-1][j-1] +

                           (1 - 2 \* self.sigma) \* u[i-1][j] +

                           self.sigma \* u[i-1][j+1]

                           + self.tau \* self.D.g(j \* self.h, (i-1) \* self.tau))

            u[i][0] = self.zero\_aprox()(i, u[i], u[i-1])

            u[i][-1] = self.l\_aprox()(i, u[i], u[i-1])

        return u

class SolveImplicit(AbstractSolver):

    aprox = 0

    def add\_aprox(self, num: int):

        self.aprox = num

        return self

    def zero\_aprox(self):

        if self.aprox == 0:

            return lambda i, curr: (0, -1, 1, self.h \* self.D.phi\_zero(i \* self.tau))

        elif self.aprox == 1:

            return lambda i, curr: (0,

                                    -2\*self.sigma - 1,

                                    2\*self.sigma,

                                    2 \* self.sigma \* self.h \* self.D.phi\_zero(i \* self.tau) - (curr[0] + self.tau \* self.D.g(0, i \* self.tau))

            )

        elif self.aprox == 2:

            return lambda i, curr: (0,

                                    -2\*self.sigma - 1,

                                    2\*self.sigma,

                                    2 \* self.sigma \* self.h \* self.D.phi\_zero(i \* self.tau) - (curr[0] + self.tau \* self.D.g(0, i \* self.tau))

            )

    def l\_aprox(self):

        if self.aprox == 0:

            return lambda i, curr: (-1, 1, 0, self.h \* self.D.phi\_l(i \* self.tau))

        elif self.aprox == 1:

            return lambda i, curr: (-4 + (1 + 2\*self.sigma) / self.sigma,

                                    2,

                                    0,

                                    2 \* self.sigma \* self.h \* self.D.phi\_l(i \* self.tau) + (curr[-2] + self.tau \* self.D.g((self.N-2) \* self.h, i \* self.tau))

            )

        elif self.aprox == 2:

            return lambda i, curr: (2\*self.sigma,

                                    -2 \* self.sigma - 1,

                                    0,

                                    -2 \* self.sigma \* self.h \* self.D.phi\_l(i \* self.tau) - (curr[-1] + self.tau \* self.D.g((self.N-1) \* self.h, i \* self.tau))

            )

    def solve(self):

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

        u\_zero = []

        for i in [i \* self.h for i in range(self.N)]:

            u\_zero.append(self.D.psi(i))

        u[0] = np.array(u\_zero)

        for i in range(1, self.K):

            a = np.zeros(self.N)

            b = np.zeros(self.N)

            c = np.zeros(self.N)

            d = np.zeros(self.N)

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

                a[j] = self.sigma

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

                c[j] = self.sigma

                d[j] = -self.tau \* self.D.g(j \* self.h, i \* self.tau) - u[i - 1][j]

            a[0], b[0], c[0], d[0] = self.zero\_aprox()(i, u[i-1])

            a[-1], b[-1], c[-1], d[-1] = self.l\_aprox()(i, u[i-1])

            u[i] = tridiagonal\_solve(a, b, c, d)

        return u

class SolveCN(AbstractSolver):

    aprox = 0

    def add\_aprox(self, num: int):

        self.aprox = num

        return self

    def zero\_aprox(self):

        if self.aprox == 0:

            return lambda i, curr: (0, -1, 1, self.h \* self.D.phi\_zero(i \* self.tau))

        elif self.aprox == 1:

            return lambda i, curr: (0,

                                    -2,

                                    4 + (-1 - 2 \* self.sigma) / self.sigma,

                                    (self.sigma \* self.h \* self.D.phi\_zero(i \* self.tau) - (curr[1] + self.tau \* self.D.g(self.h, i \* self.tau)) -

                                    0.5\*self.sigma \* (curr[0] - 2\*curr[1] + curr[2] + self.h\*\*2 \* self.D.g(self.h, (i-1) \* self.tau))

                                    )

            )

        elif self.aprox == 2:

            return lambda i, curr: (0,

                                    -self.sigma - 1,

                                    self.sigma,

                                    (self.sigma \* self.h \* self.D.phi\_zero(i \* self.tau) - (curr[0] + 0.5 \* self.tau \* self.D.g(0, i \* self.tau)) -

                                    self.sigma \* (curr[1] - curr[0] - self.h \* self.D.phi\_zero((i-1) \* self.tau) + 0.5 \* self.h \*\* 2 \* self.D.g(0, (i-1) \* self.tau))

                                    )

            )

    def l\_aprox(self):

        if self.aprox == 0:

            return lambda i, curr: (-1, 1, 0, self.h \* self.D.phi\_l(i \* self.tau))

        elif self.aprox == 1:

            return lambda i, curr: (-4 + (1 + 2 \* self.sigma) / self.sigma,

                                    2,

                                    0,

                                    (self.sigma \* self.h \* self.D.phi\_l(i \* self.tau) + (curr[-2] + 0.5 \* self.tau \* self.D.g((self.N-2) \* self.h, i \* self.tau)) +

                                    0.5 \* self.sigma \* (curr[-3] - 2\*curr[-2] + curr[-1] + self.h \*\* 2 \* self.D.g((self.N-2) \* self.h, (i-1) \* self.tau))

                                    )

            )

        elif self.aprox == 2:

            return lambda i, curr: (self.sigma,

                                    -self.sigma - 1,

                                    0,

                                    (-self.sigma \* self.h \* self.D.phi\_l(i \* self.tau) - (curr[-1] + 0.5 \* self.tau \* self.D.g((self.N-1) \* self.h, (i) \* self.tau)) -

                                    self.sigma \* (curr[-2] - curr[-1] + self.h \* self.D.phi\_l((i-1) \* self.tau) + 0.5 \* self.h \*\* 2 \* self.D.g((self.N-1) \* self.h, (i-1) \* self.tau))

                                    )

            )

    def solve(self):

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

        u\_zero = []

        for i in [i \* self.h for i in range(self.N)]:

            u\_zero.append(self.D.psi(i))

        u[0] = np.array(u\_zero)

        for i in range(1, self.K):

            a = np.zeros(self.N)

            b = np.zeros(self.N)

            c = np.zeros(self.N)

            d = np.zeros(self.N)

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

                a[j] = 0.5 \* self.sigma

                b[j] = -1 - self.sigma

                c[j] = 0.5 \* self.sigma

                d[j] = (-0.5 \* self.tau \* self.D.g(j \* self.h, i \* self.tau) - u[i - 1][j] -

                        0.5 \* self.sigma \* (u[i - 1][j - 1] - 2 \* u[i - 1][j] + u[i - 1][j + 1] + self.h \*\* 2 \* self.D.g(j \* self.h, (i-1) \* self.tau))

                )

            a[0], b[0], c[0], d[0] = self.zero\_aprox()(i, u[i-1])

            a[-1], b[-1], c[-1], d[-1] = self.l\_aprox()(i, u[i-1])

            u[i] = tridiagonal\_solve(a, b, c, d)

        return u

def MAE(numeric, analytic):

    return np.abs(numeric - analytic).max(axis=1)

class Plotter:

    solves = []

    solves\_lables = ["exact",

                     "explicit1", "explicit2", "explicit3",

                     "implicit1", "implicit2", "implicit3",

                     "cn1", "cn2", "cn3"]

    T: float

    L: float

    K: float

    N: float

    h: float

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

        self.T = T

        self.L = L

        self.K = K

        self.N = N

        self.h = SolveExact(T, L, K, N).h

        self.tau = SolveExact(T, L, K, N).tau

        self.solves.append(SolveExact(T, L, K, N))

        self.solves.append(SolveExplicit(T, L, K, N).add\_aprox(0))

        self.solves.append(SolveExplicit(T, L, K, N).add\_aprox(1))

        self.solves.append(SolveExplicit(T, L, K, N).add\_aprox(2))

        self.solves.append(SolveImplicit(T, L, K, N).add\_aprox(0))

        self.solves.append(SolveImplicit(T, L, K, N).add\_aprox(1))

        self.solves.append(SolveImplicit(T, L, K, N).add\_aprox(2))

        self.solves.append(SolveCN(T, L, K, N).add\_aprox(0))

        self.solves.append(SolveCN(T, L, K, N).add\_aprox(1))

        self.solves.append(SolveCN(T, L, K, N).add\_aprox(2))

    def plot\_solve(self, \*args):

        fig, ax1 = plt.subplots(1, 1, figsize=(7, 5))

        ax1.set\_title(f"U(x), t = {self.K - 1}")

        x = [i \* self.h for i in range(self.N - 1)]

        x.append(self.L)

        x = np.array(x)

        for i in range(len(args)):

            if args[i] == '1':

                ax1.plot(x, self.solves[i].solve()[self.K - 1], label = self.solves\_lables[i])

        ax1.grid()

        ax1.legend(loc="upper right")

        ax1.set\_ylabel("U")

        ax1.set\_xlabel("x")

    def plot\_errors\_time(self, \*args):

        fig, ax1 = plt.subplots(1, 1, figsize=(7, 5))

        ax1.set\_title(f"errors by time")

        t = [i \* self.tau for i in range(self.K - 1)]

        t.append(self.T)

        t = np.array(t)

        for i in range(len(args)):

            if args[i] == '1':

                ax1.plot(t, np.abs(self.solves[i].solve() - self.solves[0].solve()).max(axis=1), label = self.solves\_lables[i])

        ax1.grid()

        ax1.legend(loc="upper right")

        ax1.set\_ylabel("E")

        ax1.set\_xlabel("t")

    def plot\_errors\_by\_precision(self, \*args):

        fig, ax1 = plt.subplots(1, 1, figsize=(7, 5))

        ax1.set\_title(f"errors by precision")

        n\_step = (9 - 4) // 5

        k\_step = (80 - 60) // 5

        nn = [4 + n\_step\*i for i in range(5)]

        nn = np.array(nn)

        kk = [60 + k\_step\*i for i in range(5)]

        kk = np.array(kk)

        for i in range(len(args)):

            if args[i] == '1':

                ers = []

                h\_tau\_params = []

                for step in range(5):

                    n = nn[step]

                    k = kk[step]

                    h = self.L / (n - 1)

                    tau = self.T / (k - 1)

                    h\_tau\_params.append(f"{np.log(h):,.3f} | {np.log(tau):,.3f}")

                    tt = [i \* tau for i in range(k - 1)]

                    tt.append(self.T)

                    tt = np.array(tt)

                    x = [i \* h for i in range(n - 1)]

                    x.append(self.L)

                    x = np.array(x)

                    self.solves[i].\_\_init\_\_(self.T, self.L, k, n)

                    self.solves[0].\_\_init\_\_(self.T, self.L, k, n)

                    ers.append(max(MAE(self.solves[i].solve(), self.solves[0].solve())))

                ax1.plot(h\_tau\_params, np.log(ers), label=self.solves\_lables[i])

                print(self.solves\_lables[i], "tg:", (np.log10(ers[1]) - np.log10(ers[0])) / (np.log10(kk[1]) - np.log10(kk[0])))

        ax1.grid()

        ax1.legend(loc="upper right")

        ax1.set\_ylabel("E")

        ax1.set\_xlabel("N")

argv = sys.argv

print(sys.argv)

const\_sigma = 0.5

K = 80

L = np.pi

T = 5

N = int(1 + np.sqrt((const\_sigma\*L\*\*2 \* K) / (T)))

plotter = Plotter(T, L, K, N)

plotter.plot\_solve(\*sys.argv[1:])

plotter.plot\_errors\_time(\*sys.argv[1:])

plotter.plot\_errors\_by\_precision(\*sys.argv[1:])

plt.show()

## Пример работы:

## 

# Лабораторная работа №**2**

## **Задание:**

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.

## **Код**:

Программа реализована на языке программирования Python.

### **main.py:**

1. **from abc import ABC, abstractmethod**

import sys

import numpy as np

import matplotlib.pyplot as plt

def tridiagonal\_solve(a, b, c, d) -> np.ndarray:

    n = len(d)

    p = np.ndarray(n, dtype=float)

    q = np.ndarray(n, dtype=float)

    x = np.ndarray(n, dtype=float)

    p[0] = -c[0] / b[0]

    q[0] = d[0] / b[0]

    for i in range(1, n):

        p[i] = -c[i] / (b[i] + a[i]\*p[i-1])

        q[i] = (d[i] - a[i]\*q[i-1]) / (b[i] + a[i]\*p[i-1])

    x[-1] = q[-1]

    for i in range(n-2, -1, -1):

        x[i] = p[i] \* x[i+1] + q[i]

    return x

class Diffur:

    @staticmethod

    def psi\_1(x): return np.exp(2 \* x)

    @staticmethod

    def psi\_2(x): return 0

    @staticmethod

    def d2\_psi\_1(x): return 4 \* np.exp(2 \* x)

    def \_\_init\_\_(self):

        pass

class AbstractSolver(ABC):

    D: Diffur

    T: float

    L: float

    N: float

    K: float

    tau: float

    h: float

    sigma: float

    @staticmethod

    def calc\_tau(T: float, K: float) -> float:

        return T / (K-1)

    @staticmethod

    def calc\_h(L: float, N: float) -> float:

        return L / (N-1)

    @staticmethod

    def calc\_sigma(tau: float, h: float) -> float:

        return tau\*\*2 / h\*\*2

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

        self.D = Diffur()

        self.T = T

        self.L = L

        self.N = N

        self.K = K

        self.tau = self.calc\_tau(T, K)

        self.h = self.calc\_h(L, N)

        self.sigma = self.calc\_sigma(self.tau, self.h)

    @abstractmethod

    def solve(self): pass

    def change\_params(self, T, L, K, N):

        self.T = T

        self.L = L

        self.N = N

        self.K = K

        self.tau = self.calc\_tau(T, K)

        self.h = self.calc\_h(L, N)

        self.sigma = self.calc\_sigma(self.tau, self.h)

class SolveExact(AbstractSolver):

    @staticmethod

    def exact\_solve(x, t):

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

    def solve(self):

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

        for i in range(self.K):

            for j in range(self.N):

                u[i][j] = self.exact\_solve(j \* self.h, i \* self.tau)

        return u

class SolveExplicit(AbstractSolver):

    aprox = 0

    """

    тут и далее

    0 - двухточечная аппроксимация с первым порядком

    1 - трехточечная аппроксимация со вторым порядком

    2 - двухточечная аппроксимация со вторым порядком

    """

    def add\_aprox(self, num: int):

        self.aprox = num

        return self

    def zero\_aprox(self):

        if self.aprox == 0:

            return lambda k, u: u[k][1] / (1 + 2 \* self.h)

        elif self.aprox == 1:

            return lambda k, u: (4 \* u[k][1] - u[k][2]) / (3 + 4 \* self.h)

        elif self.aprox == 2:

            return lambda k, u: self.sigma \* (2 \* u[k - 1][1] - (2 + 4 \* self.h) \* u[k - 1][0]) + (2 - 5 \* self.tau\*\*2) \* u[k - 1][0] - u[k - 2][0]

    def l\_aprox(self):

        if self.aprox == 0:

            return lambda k, u: u[k][-2] / (1 - 2 \* self.h)

        elif self.aprox == 1:

            return lambda k, u: (4 \* u[k][-2] - u[k][-3]) / (3 - 4 \* self.h)

        elif self.aprox == 2:

            return lambda k, u: self.sigma \* (2 \* u[k - 1][-2] + (4 \* self.h - 2) \* u[k - 1][-1]) + (2 - 5 \* self.tau\*\*2) \* u[k - 1][-1] - u[k - 2][-1]

    def solve(self):

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

        u\_zero = []

        for i in [i \* self.h for i in range(self.N)]:

            u\_zero.append(self.D.psi\_1(i))

        u[0] = np.array(u\_zero)

        u\_one = []

        for i in [i \* self.h for i in range(self.N)]:

            u\_one.append(self.D.psi\_1(i) + self.tau \* self.D.psi\_2(i) + self.tau\*\*2 \* self.D.d2\_psi\_1(i) / 2)

        u[1] = np.array(u\_one)

        for i in range(2, self.K):

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

                u[i][j] = (self.sigma \*

                            (u[i - 1][j - 1] - 2 \* u[i - 1][j] + u[i - 1][j + 1]) -

                            (5 \* self.tau\*\*2 \* u[i - 1][j]) +

                            (2 \* u[i - 1][j]) - u[i - 2][j])

            u[i][0] = self.zero\_aprox()(i, u)

            u[i][-1] = self.l\_aprox()(i, u)

        return u

class SolveImplicit(AbstractSolver):

    aprox = 0

    def add\_aprox(self, num: int):

        self.aprox = num

        return self

    def zero\_aprox(self):

        if self.aprox == 0:

            return lambda k, u: (0, (1 + 2 \* self.h), -1, 0)

        elif self.aprox == 1:

            return lambda k, u: (0,

                                -(2 + 4 \* self.h),

                                -(5 \* self.h\*\*2 + 1 / self.sigma - 2),

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

            )

        elif self.aprox == 2:

            return lambda k, u: (0,

                                -(2 + 5 \* self.h\*\*2 + 4 \* self.h + 1 / self.sigma),

                                2,

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

            )

    def l\_aprox(self):

        if self.aprox == 0:

            return lambda k, u: (-1, (1 - 2 \* self.h), 0, 0)

        elif self.aprox == 1:

            return lambda k, u: (-(5 \* self.h\*\*2 + 1 / self.sigma - 2),

                                -(2 - 4 \* self.h),

                                0,

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

            )

        elif self.aprox == 2:

            return lambda k, u: (2,

                                -(2 + 5 \* self.h\*\*2 - 4 \* self.h + 1 / self.sigma),

                                0,

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

            )

    def solve(self):

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

        u\_zero = []

        for i in [i \* self.h for i in range(self.N)]:

            u\_zero.append(self.D.psi\_1(i))

        u[0] = np.array(u\_zero)

        u\_one = []

        for i in [i \* self.h for i in range(self.N)]:

            u\_one.append(self.D.psi\_1(i) + self.tau \* self.D.psi\_2(i) + self.tau\*\*2 \* self.D.d2\_psi\_1(i) / 2)

        u[1] = np.array(u\_one)

        for i in range(2, self.K):

            a = np.zeros(self.N)

            b = np.zeros(self.N)

            c = np.zeros(self.N)

            d = np.zeros(self.N)

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

                a[j] = 1

                b[j] = -(2 + 5 \* self.h\*\*2 + 1 / self.sigma)

                c[j] = 1

                d[j] = (u[i - 2][j] - 2\*u[i - 1][j]) / self.sigma

            a[0], b[0], c[0], d[0] = self.zero\_aprox()(i, u)

            a[-1], b[-1], c[-1], d[-1] = self.l\_aprox()(i, u)

            u[i] = tridiagonal\_solve(a, b, c, d)

        return u

def MAE(numeric, analytic):

    return np.abs(numeric - analytic).max(axis=1)

class Plotter:

    solves = []

    solves\_lables = ["exact",

                     "explicit1", "explicit2", "explicit3",

                     "implicit1", "implicit2", "implicit3"]

    T: float

    L: float

    K: float

    N: float

    h: float

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

        self.T = T

        self.L = L

        self.K = K

        self.N = N

        self.h = SolveExact(T, L, K, N).h

        self.tau = SolveExact(T, L, K, N).tau

        self.solves.append(SolveExact(T, L, K, N))

        self.solves.append(SolveExplicit(T, L, K, N).add\_aprox(0))

        self.solves.append(SolveExplicit(T, L, K, N).add\_aprox(1))

        self.solves.append(SolveExplicit(T, L, K, N).add\_aprox(2))

        self.solves.append(SolveImplicit(T, L, K, N).add\_aprox(0))

        self.solves.append(SolveImplicit(T, L, K, N).add\_aprox(1))

        self.solves.append(SolveImplicit(T, L, K, N).add\_aprox(2))

    def plot\_solve(self, \*args):

        fig, ax1 = plt.subplots(1, 1, figsize=(7, 5))

        ax1.set\_title(f"U(x), t = {self.K - 1}")

        x = [i \* self.h for i in range(self.N - 1)]

        x.append(self.L)

        x = np.array(x)

        for i in range(len(args)):

            if args[i] == '1':

                ax1.plot(x, self.solves[i].solve()[self.K - 1], label = self.solves\_lables[i])

        ax1.grid()

        ax1.legend(loc="upper right")

        ax1.set\_ylabel("U")

        ax1.set\_xlabel("x")

    def plot\_errors\_time(self, \*args):

        fig, ax1 = plt.subplots(1, 1, figsize=(7, 5))

        ax1.set\_title(f"errors by time")

        t = [i \* self.tau for i in range(self.K - 1)]

        t.append(self.T)

        t = np.array(t)

        for i in range(len(args)):

            if args[i] == '1':

                ax1.plot(t, np.abs(self.solves[i].solve() - self.solves[0].solve()).max(axis=1), label = self.solves\_lables[i])

        ax1.grid()

        ax1.legend(loc="upper right")

        ax1.set\_ylabel("E")

        ax1.set\_xlabel("t")

    def plot\_errors\_by\_precision(self, \*args):

        fig, ax1 = plt.subplots(1, 1, figsize=(7, 5))

        ax1.set\_title(f"errors by precision")

        count = 3 # self.N - 5

        const\_sigma = 1

        h = np.array(list(map(int, np.linspace(start=15, stop=self.N, num=count))))

        tau = []

        for i in h:

            tau.append(int(self.T \* i \* (const\_sigma \*\* 0.5) \* (1 / (self.L))))

        tau = np.array(tau)

        print(tau)

        print(h)

        for i in range(len(args)):

            if args[i] == '1':

                err = []

                for x in zip(tau, h):

                    self.solves[i].\_\_init\_\_(self.T, self.L, x[0], x[1])

                    self.solves[0].\_\_init\_\_(self.T, self.L, x[0], x[1])

                    err.append(max(MAE(self.solves[i].solve(), self.solves[0].solve())))

                ax1.plot(np.log(h), np.log10(err), label = self.solves\_lables[i])

                print(self.solves\_lables[i] + " tg =", (np.log10(err[-1]) - np.log10(err[0])) / (np.log10(h[-1]) - np.log10(h[0])))

        ax1.grid()

        ax1.legend(loc="upper right")

        ax1.set\_ylabel("E")

        ax1.set\_xlabel("N")

argv = sys.argv

print(sys.argv)

"""

const\_sigma = 0.5

K = 80

L = np.pi

T = 5

N = int(1 + np.sqrt((const\_sigma\*L\*\*2 \* K) / (T)))

"""

K = 100

L = 1

T = 1

N = 30

plotter = Plotter(T, L, K, N)

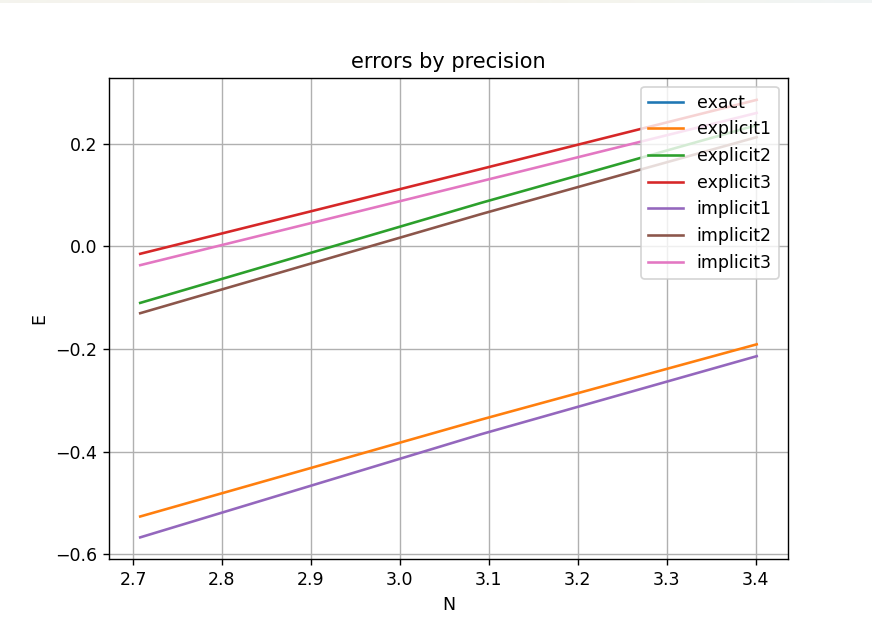
plotter.plot\_solve(\*sys.argv[1:])

plotter.plot\_errors\_time(\*sys.argv[1:])

plotter.plot\_errors\_by\_precision(\*sys.argv[1:])

plt.show()

## **Пример работы:**



# Лабораторная работа №**7**

## **Задание:**

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. Исследовать зависимость погрешности от сеточных параметров.

## **Код**:

Программа реализована на языке программирования Python.

### **main.py:**

1. from math import \*

from abc import ABC, abstractmethod

import numpy as np

import matplotlib.pyplot as plt

class Diffur:

    a = 2

    b = 2

    c = 4

    alpha\_1 = 0

    beta\_1 = 1

    alpha\_2 = 0

    beta\_2 = 1

    alpha\_3 = 0

    beta\_3 = 1

    alpha\_4 = 0

    beta\_4 = 1

    @staticmethod

    def f(x, y): return 0

    @staticmethod

    def phi\_1(y): return exp(-y) \* cos(y)

    @staticmethod

    def phi\_2(y): return 0

    @staticmethod

    def phi\_3(x): return exp(-x) \* cos(x)

    @staticmethod

    def phi\_4(x): return 0

    @staticmethod

    def exact(x, y): return exp(-x - y) \* cos(x) \* cos(y)

    def \_\_init\_\_(self):

        pass

def interpolate\_2d(n, m, a):

    up = a[0, 1:-1].any()

    down = a[-1, 1:-1].any()

    left = a[1:-1, 0].any()

    right = a[1:-1, -1].any()

    if up and left:

        for i in range(1, n):

            for j in range(1, m):

                a[i, j] = np.linspace(a[0, j], a[i, 0], i + 1 + j + 1 - 1)[i]

    elif up and right:

        for i in range(1, n):

            for j in range(0, m - 1):

                a[i, j] = np.linspace(a[0, j], a[i, -1], i + 1 + (m - j) - 1)[i]

    elif down and right:

        for i in range(0, n - 1):

            for j in range(0, m - 1):

                a[i, j] = np.linspace(a[-1, j], a[i, -1], (n - i) + (m - j) - 1)[n - i - 1]

    elif down and left:

        for i in range(0, n - 1):

            for j in range(1, m):

                a[i, j] = np.linspace(a[-1, j], a[i, 0], (n - i) + j + 1 - 1)[n - i - 1]

    elif up and down:

        for j in range(m):

            a[1:-1, j] = np.linspace(a[0, j], a[-1, j], n)[1:-1]

    elif left and right:

        for i in range(n):

            a[i, 1:-1] = np.linspace(a[i, 0], a[i, -1], m)[1:-1]

    else:

        print("no inerpolate =(")

    return a

def L2\_norm(A, B):

    diff = A - B

    return np.sqrt(np.sum(diff \*\* 2))

def solve(x\_range, y\_range, h\_x, h\_y, method, theta, eps):

    if method == "exact":

        exact, iterss = solve\_exact(x\_range, y\_range, h\_x, h\_y)

        return exact, iterss

    d = Diffur()

    obhod = "rd"

    x = np.arange(\*x\_range, h\_x)

    y = np.arange(\*y\_range, h\_y)

    n = len(x)

    m = len(y)

    res = np.zeros((n, m))

    for i in range(m):

        if d.alpha\_1 == 0:

            res[0][i] = 1 / d.beta\_1 \* d.phi\_1(y[i])

        if d.alpha\_2 == 0:

            res[-1][i] = 1 / d.beta\_2 \* d.phi\_2(y[i])

    for j in range(n):

        if d.alpha\_3 == 0:

            res[j][0] = 1 / d.beta\_3 \* d.phi\_3(x[j])

        if d.alpha\_4 == 0:

            res[j][-1] = 1 / d.beta\_4 \* d.phi\_4(x[j])

    n, m = res.shape

    interpolate\_2d(n, m, res)

    iters = 1

    while True:

        res\_prev = res

        res = np.zeros((n, m))

        for i in range(m):

            if d.alpha\_1 == 0:

                res[0][i] = 1 / d.beta\_1 \* d.phi\_1(y[i])

            if d.alpha\_2 == 0:

                res[-1][i] = 1 / d.beta\_2 \* d.phi\_2(y[i])

        for j in range(n):

            if d.alpha\_3 == 0:

                res[j][0] = 1 / d.beta\_3 \* d.phi\_3(x[j])

            if d.alpha\_4 == 0:

                res[j][-1] = 1 / d.beta\_4 \* d.phi\_4(x[j])

        obhods = ['rd', 'ld', 'lu', 'ru']

        obhods\_info = [

            (1, n - 2, 1, m - 2),

            (1, n - 2, m - 2, 1),

            (n - 2, 1, m - 2, 1),

            (n - 2, 1, 1, m - 2),

        ]  # ОО - обратный обход

        obhod\_idx = obhods.index(obhod)

        x\_start, x\_end, y\_start, y\_end = obhods\_info[obhod\_idx]

        x\_dir = int((x\_end - x\_start) > 0) - int((x\_end - x\_start) < 0)

        y\_dir = int((y\_end - y\_start) > 0) - int((y\_end - y\_start) < 0)

        uij\_coeff = (d.c - 2 / h\_x \*\* 2 - 2 / h\_y \*\* 2)

        for i in range(x\_start, x\_end + x\_dir, x\_dir):

            for j in range(y\_start, y\_end + y\_dir, y\_dir):

                if method == "simple":

                    part\_d2u\_dx2 = 1 / h\_x \*\* 2 \* (res\_prev[i + x\_dir][j] + res\_prev[i - x\_dir][j])

                    part\_d2u\_dy2 = 1 / h\_y \*\* 2 \* (res\_prev[i][j + y\_dir] + res\_prev[i][j - y\_dir])

                    du\_dx = d.a / (2 \* h\_x) \* (res\_prev[i + x\_dir][j] - res\_prev[i - x\_dir][j])

                    du\_dy = d.b / (2 \* h\_y) \* (res\_prev[i][j + y\_dir] - res\_prev[i][j - y\_dir])

                    res[i][j] = 1 / uij\_coeff \* (d.f(x[i], y[j]) - (part\_d2u\_dx2 + part\_d2u\_dy2 + du\_dx + du\_dy))

                elif method == "zeidel" or method == "relaxation":

                    part\_d2u\_dx2 = 1 / h\_x \*\* 2 \* (res\_prev[i + x\_dir][j] + res[i - x\_dir][j])

                    part\_d2u\_dy2 = 1 / h\_y \*\* 2 \* (res\_prev[i][j + y\_dir] + res[i][j - y\_dir])

                    du\_dx = d.a / (2 \* h\_x) \* (res\_prev[i + x\_dir][j] - res[i - x\_dir][j])

                    du\_dy = d.b / (2 \* h\_y) \* (res\_prev[i][j + y\_dir] - res[i][j - y\_dir])

                    res[i][j] = (

                            theta \* (1 / uij\_coeff \* (

                            d.f(x[i], y[j]) - (part\_d2u\_dx2 + part\_d2u\_dy2 + du\_dx + du\_dy))) +

                            (1 - theta) \* res\_prev[i][j]

                    )

        for i in range(1, m - 1):

            if d.alpha\_1 != 0:

                u0j\_coef = 2 \* h\_x \* d.beta\_1 - 3 \* d.alpha\_1

                res[0][i] = 1 / u0j\_coef \* (

                        2 \* h\_x \* d.phi\_1(y[i]) - d.alpha\_1 \* (4 \* res[1][i] - res[2][i]))

            if d.alpha\_2 != 0:

                unj\_coeff = 2 \* h\_x \* d.beta\_2 + 3 \* d.alpha\_2

                res[-1][i] = 1 / unj\_coeff \* (

                        2 \* h\_x \* d.phi\_2(y[i]) + d.alpha\_2 \* (4 \* res[-2][i] - res[-3][i]))

        for j in range(1, n - 1):

            if d.alpha\_3 != 0:

                ui0\_coef = 2 \* h\_y \* d.beta\_3 - 3 \* d.alpha\_3

                res[j][0] = 1 / ui0\_coef \* (

                        2 \* h\_y \* d.phi\_3(x[j]) - d.alpha\_3 \* (4 \* res[j][1] - res[j][2]))

            if d.alpha\_4 != 0:

                uim\_coeff = 2 \* h\_y \* d.beta\_4 + 3 \* d.alpha\_4

                res[j][-1] = 1 / uim\_coeff \* (

                        2 \* h\_y \* d.phi\_4(x[j]) + d.alpha\_4 \* (4 \* res[j][-2] - res[j][-3]))

        if L2\_norm(res, res\_prev) < eps:

            break

        iters += 1

    return res, iters

def solve\_exact(x\_range, y\_range, h\_x, h\_y):

    d = Diffur()

    x = np.arange(\*x\_range, h\_x)

    y = np.arange(\*y\_range, h\_y)

    res = np.zeros((len(x), len(y)))

    for idx in range(len(x)):

        for idy in range(len(y)):

            res[idx][idy] = d.exact(x[idx], y[idy])

    return res, 1

def MAE(A, B):

    return abs(A - B).mean()

def maxAE(A, B):

    return abs(A - B).max()

def plot\_results(x\_end, y\_end, x\_steps, y\_steps, theta, eps):

    x\_range = (0, x\_end)

    y\_range = (0, y\_end)

    h\_x = x\_end / x\_steps

    h\_y = y\_end / y\_steps

    exact, \_ = solve(x\_range, y\_range, h\_x, h\_y, "exact", theta, eps)

    simple, simple\_iters = solve(x\_range, y\_range, h\_x, h\_y, "simple", theta, eps)

    print("Метод простых итераций:")

    print(f'iters: {simple\_iters}')

    print(f'Mean Abs Err: {MAE(simple, exact)}')

    print()

    zeidel, zeidel\_iters = solve(x\_range, y\_range, h\_x, h\_y, "zeidel", 1, eps)

    print("Метод Зейделя:")

    print(f'iters: {zeidel\_iters}')

    print(f'Mean Abs Err: {MAE(zeidel, exact)}')

    print()

    relaxation, relaxation\_iters = solve(x\_range, y\_range, h\_x, h\_y, "relaxation", theta, eps)

    print("Метод верхней релаксации:")

    print(f'iters: {relaxation\_iters}')

    print(f'Mean Abs Err: {MAE(relaxation, exact)}')

    print()

    x = np.arange(\*x\_range, h\_x)

    y = np.arange(\*y\_range, h\_y)

    fig, axs = plt.subplots(1, 2, figsize=(9, 3))

    lines\_x = []

    lines\_y = []

    solutions = {

        "exact": exact,

        "simple": simple,

        "zeidel": zeidel,

        "relaxation": relaxation,

    }

    for method\_name, solution in solutions.items():

        line\_x, = axs[0].plot(y, solution[1, :], label=method\_name)

        lines\_x.append(line\_x)

        line\_y, = axs[1].plot(x, solution[:, 1], label=method\_name)

        lines\_y.append(line\_y)

    axs[0].set\_title('u(x, y)')

    axs[0].set\_xlabel('y')

    axs[0].set\_ylabel('u(x, y)')

    axs[0].legend()

    axs[1].set\_title('u(x, y)')

    axs[1].set\_xlabel('x')

    axs[1].set\_ylabel('u(x, y)')

    axs[1].legend()

    fig2, axs2 = plt.subplots(1, 2, figsize=(9, 3))

    h\_x\_s = np.arange(x\_steps // 2, x\_steps, 2)

    h\_y\_s = np.arange(y\_steps // 2, y\_steps, 2)

    MAES\_x\_simple = []

    MAES\_x\_zeidel = []

    MAES\_x\_relaxation = []

    for i in h\_x\_s:

        h\_x = x\_end / i

        h\_y = y\_end / y\_steps

        MAES\_x\_simple.append(MAE(solve(x\_range, y\_range, h\_x, h\_y, "simple", theta, eps)[0], solve(x\_range, y\_range, h\_x, h\_y, "exact", theta, eps)[0]))

        MAES\_x\_zeidel.append(MAE(solve(x\_range, y\_range, h\_x, h\_y, "zeidel", 1, eps)[0], solve(x\_range, y\_range, h\_x, h\_y, "exact", theta, eps)[0]))

        MAES\_x\_relaxation.append(MAE(solve(x\_range, y\_range, h\_x, h\_y, "relaxation", theta, eps)[0], solve(x\_range, y\_range, h\_x, h\_y, "exact", theta, eps)[0]))

    MAES\_y\_simple = []

    MAES\_y\_zeidel = []

    MAES\_y\_relaxation = []

    for j in h\_y\_s:

        h\_x = x\_end / x\_steps

        h\_y = y\_end / j

        MAES\_y\_simple.append(MAE(solve(x\_range, y\_range, h\_x, h\_y, "simple", theta, eps)[0], solve(x\_range, y\_range, h\_x, h\_y, "exact", theta, eps)[0]))

        MAES\_y\_zeidel.append(MAE(solve(x\_range, y\_range, h\_x, h\_y, "zeidel", 1, eps)[0], solve(x\_range, y\_range, h\_x, h\_y, "exact", theta, eps)[0]))

        MAES\_y\_relaxation.append(MAE(solve(x\_range, y\_range, h\_x, h\_y, "relaxation", theta, eps)[0], solve(x\_range, y\_range, h\_x, h\_y, "exact", theta, eps)[0]))

    print(f"интервалы h\_x: {h\_x\_s}")

    print(f"интервалы h\_y: {h\_y\_s}")

    solutions\_MAES = {

        "simple": (MAES\_x\_simple, MAES\_y\_simple),

        "zeidel": (MAES\_x\_zeidel, MAES\_y\_zeidel),

        "relaxation": (MAES\_x\_relaxation , MAES\_y\_relaxation),

    }

    for method\_name, solution in solutions\_MAES.items():

        line\_x, = axs2[0].plot(h\_x\_s, solution[0], label=method\_name)

        lines\_x.append(line\_x)

        line\_y, = axs2[1].plot(h\_y\_s, solution[1], label=method\_name)

        lines\_y.append(line\_y)

    axs2[0].set\_title('Ошибка по h\_x')

    axs2[0].set\_xlabel('h\_x')

    axs2[0].set\_ylabel('MAE')

    axs2[0].legend()

    axs2[1].set\_title('Ошибка по h\_y')

    axs2[1].set\_xlabel('h\_y')

    axs2[1].set\_ylabel('MAE')

    axs2[1].legend()

    plt.show()

plot\_results(pi / 2, pi / 2, 33, 33, 1.5, 1e-1)

## **Пример работы:**

# Лабораторная работа №**8**

## **Задание:**

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.

## **Код**:

Программа реализована на языке программирования Python.

**main.py**

from math import \*

import numpy as np

import matplotlib.pyplot as plt

def tridiagonal\_solve(A, b):

    n = len(A)

    v = [0 for \_ in range(n)]

    u = [0 for \_ in range(n)]

    v[0] = A[0][1] / -A[0][0]

    u[0] = b[0] / A[0][0]

    for i in range(1, n-1):

        v[i] = A[i][i+1] / (-A[i][i] - A[i][i-1] \* v[i-1])

        u[i] = (A[i][i-1] \* u[i-1] - b[i]) / (-A[i][i] - A[i][i-1] \* v[i-1])

    v[n-1] = 0

    u[n-1] = (A[n-1][n-2] \* u[n-2] - b[n-1]) / (-A[n-1][n-1] - A[n-1][n-2] \* v[n-2])

    x = [0 for \_ in range(n)]

    x[n-1] = u[n-1]

    for i in range(n-1, 0, -1):

        x[i-1] = v[i-1] \* x[i] + u[i-1]

    return np.array(x)

class Diffur:

    @staticmethod

    def f(x, y, t):

        return -x \* y \* sin(t)

    @staticmethod

    def phi\_1(y, t):

        return 0

    @staticmethod

    def phi\_2(y, t):

        return y \* cos(t)

    @staticmethod

    def phi\_3(x, t):

        return 0

    @staticmethod

    def phi\_4(x, t):

        return x \* cos(t)

    @staticmethod

    def psi(x, y):

        return x \* y

    @staticmethod

    def solution(x, y, t):

        return x \* y \* cos(t)

def max\_abs\_error(A, B):

    return abs(A - B).max()

def mean\_abs\_error(A, B):

    return abs(A - B).mean()

def get\_analytical\_solution(x\_range, y\_range, t\_range, h\_x, h\_y, tau):

    d = Diffur()

    x = np.arange(\*x\_range, h\_x)

    y = np.arange(\*y\_range, h\_y)

    t = np.arange(\*t\_range, tau)

    res = np.zeros((len(t), len(x), len(y)))

    for idt in range(len(t)):

      for idx in range(len(x)):

        for idy in range(len(y)):

            res[idt][idx][idy] = d.solution(x[idx], y[idy], t[idt])

    return res

def compute\_D\_1i(res, x, y, t, cur\_x\_id, cur\_y\_id, cur\_t\_id, tau, h\_y, gamma, n):

    d = Diffur()

    return (

        res[cur\_x\_id][cur\_y\_id] \* (1 + gamma) / tau + gamma / (n - 1) \*

        (res[cur\_x\_id][cur\_y\_id + 1] - 2 \* res[cur\_x\_id][cur\_y\_id] + res[cur\_x\_id][cur\_y\_id - 1]) / h\_y\*\*2 +

        gamma / (n - 1) \* d.f(x[cur\_x\_id], y[cur\_y\_id], t[cur\_t\_id] + tau / 2) +

        1 / n \* (1 - gamma / (n - 1)) \* d.f(x[cur\_x\_id], y[cur\_y\_id], t[cur\_t\_id])

    )

def compute\_D\_2j(res\_fraction, x, y, t, cur\_x\_id, cur\_y\_id, cur\_t\_id, tau, h\_x, gamma, n):

    d = Diffur()

    return (

        res\_fraction[cur\_x\_id][cur\_y\_id] \* (1 + gamma) / tau + gamma / (n - 1) \*

        (res\_fraction[cur\_x\_id + 1][cur\_y\_id] - 2 \* res\_fraction[cur\_x\_id][cur\_y\_id] + res\_fraction[cur\_x\_id - 1][cur\_y\_id]) / h\_x\*\*2 +

        gamma / (n - 1) \* d.f(x[cur\_x\_id], y[cur\_y\_id], t[cur\_t\_id] + tau / 2) +

        1 / n \* (1 - gamma / (n - 1)) \* d.f(x[cur\_x\_id], y[cur\_y\_id], t[cur\_t\_id + 1])

    )

def finite\_difference\_schema\_general\_view(x\_range, y\_range, t\_range, h\_x, h\_y, tau,

    alpha\_1, beta\_1, alpha\_2, beta\_2, alpha\_3, beta\_3, alpha\_4, beta\_4, N,

    gamma, # МПН: gamma = 1 МДШ: gamma = 0

):

    d = Diffur()

    x = np.arange(\*x\_range, h\_x)

    y = np.arange(\*y\_range, h\_y)

    t = np.arange(\*t\_range, tau)

    n = len(x)

    m = len(y)

    k = len(t)

    answer = []

    res = np.zeros((n, m))

    u0j\_coeff = 2 \* h\_x \* beta\_1 - 3 \* alpha\_1

    unj\_coeff = 2 \* h\_x \* beta\_2 + 3 \* alpha\_2

    ui0\_coeff = 2 \* h\_y \* beta\_3 - 3 \* alpha\_3

    uim\_coeff = 2 \* h\_y \* beta\_4 + 3 \* alpha\_4

    for cur\_x\_id in range(n):

      for cur\_y\_id in range(m):

        res[cur\_x\_id][cur\_y\_id] = d.psi(x[cur\_x\_id], y[cur\_y\_id])

    answer.append(res.copy())

    for cur\_t\_id in range(0, k - 1):

      res\_fraction = res.copy()

      for cur\_y\_id in range(m):

        res\_fraction[0][cur\_y\_id] = 1 / u0j\_coeff \* (2 \* h\_x \* d.phi\_1(y[cur\_y\_id], t[cur\_t\_id] + tau / 2) - alpha\_1 \* (4 \* res\_fraction[1][cur\_y\_id] - res\_fraction[2][cur\_y\_id]))

        res\_fraction[-1][cur\_y\_id] = 1 / unj\_coeff \* (2 \* h\_x \* d.phi\_2(y[cur\_y\_id], t[cur\_t\_id] + tau / 2) + alpha\_2 \* (4 \* res\_fraction[-2][cur\_y\_id] - res\_fraction[-3][cur\_y\_id]))

      for cur\_x\_id in range(n):

        res\_fraction[cur\_x\_id][0] = 1 / ui0\_coeff \* (2 \* h\_y \* d.phi\_3(x[cur\_x\_id], t[cur\_t\_id] + tau / 2) - alpha\_3 \* (4 \* res\_fraction[cur\_x\_id][1] - res\_fraction[cur\_x\_id][2]))

        res\_fraction[cur\_x\_id][-1] = 1 / uim\_coeff \* (2 \* h\_y \* d.phi\_4(x[cur\_x\_id], t[cur\_t\_id] + tau / 2) + alpha\_4 \* (4 \* res\_fraction[cur\_x\_id][-2] - res\_fraction[cur\_x\_id][-3]))

      A\_1i = -1 / h\_x\*\*2

      B\_1i = (1 + gamma) / tau + 2 / h\_x\*\*2

      C\_1i = -1 / h\_x\*\*2

      for cur\_y\_id in range(1, m - 1):

        A = np.zeros((n-2, n-2))

        A[0][0] = B\_1i + A\_1i / u0j\_coeff \* (-4 \* alpha\_1)

        A[0][1] = C\_1i + A\_1i / u0j\_coeff \* alpha\_1

        for i in range(1, len(A) - 1):

            A[i][i-1] = A\_1i

            A[i][i] = B\_1i

            A[i][i+1] = C\_1i

        A[-1][-2] = A\_1i + C\_1i / unj\_coeff \* (-alpha\_2)

        A[-1][-1] = B\_1i + C\_1i / unj\_coeff \* 4 \* alpha\_2

        B = np.zeros(n-2)

        for cur\_x\_id in range(1, n-1):

            B[cur\_x\_id - 1] = compute\_D\_1i(res, x, y, t, cur\_x\_id, cur\_y\_id, cur\_t\_id, tau, h\_y, gamma, N)

        B[0] -= A\_1i / u0j\_coeff \* 2 \* h\_x \* d.phi\_1(y[cur\_y\_id], t[cur\_t\_id] + tau / 2)

        B[-1] -= C\_1i / unj\_coeff \* 2 \* h\_x \* d.phi\_2(y[cur\_y\_id], t[cur\_t\_id] + tau / 2)

        res\_fraction[1:-1, cur\_y\_id] = tridiagonal\_solve(A, B)

      for cur\_y\_id in range(m):

        if alpha\_1 != 0:

          res\_fraction[0][cur\_y\_id] = 1 / u0j\_coeff \* (2 \* h\_x \* d.phi\_1(y[cur\_y\_id], t[cur\_t\_id] + tau / 2) - alpha\_1 \* (4 \* res\_fraction[1][cur\_y\_id] - res\_fraction[2][cur\_y\_id]))

        if alpha\_2 != 0:

          res\_fraction[-1][cur\_y\_id] = 1 / unj\_coeff \* (2 \* h\_x \* d.phi\_2(y[cur\_y\_id], t[cur\_t\_id] + tau / 2) + alpha\_2 \* (4 \* res\_fraction[-2][cur\_y\_id] - res\_fraction[-3][cur\_y\_id]))

      for cur\_x\_id in range(n):

        if alpha\_3 != 0:

          res\_fraction[cur\_x\_id][0] = 1 / ui0\_coeff \* (2 \* h\_y \* d.phi\_3(x[cur\_x\_id], t[cur\_t\_id] + tau / 2) - alpha\_3 \* (4 \* res\_fraction[cur\_x\_id][1] - res\_fraction[cur\_x\_id][2]))

        if alpha\_4 != 0:

          res\_fraction[cur\_x\_id][-1] = 1 / uim\_coeff \* (2 \* h\_y \* d.phi\_4(x[cur\_x\_id], t[cur\_t\_id] + tau / 2) + alpha\_4 \* (4 \* res\_fraction[cur\_x\_id][-2] - res\_fraction[cur\_x\_id][-3]))

      res = res\_fraction.copy()

      for cur\_y\_id in range(m):

        if alpha\_1 == 0:

          res[0][cur\_y\_id] = 1 / u0j\_coeff \* (2 \* h\_x \* d.phi\_1(y[cur\_y\_id], t[cur\_t\_id + 1]) - alpha\_1 \* (4 \* res[1][cur\_y\_id] - res[2][cur\_y\_id]))

        if alpha\_2 == 0:

          res[-1][cur\_y\_id] = 1 / unj\_coeff \* (2 \* h\_x \* d.phi\_2(y[cur\_y\_id], t[cur\_t\_id + 1]) + alpha\_2 \* (4 \* res[-2][cur\_y\_id] - res[-3][cur\_y\_id]))

      for cur\_x\_id in range(n):

        if alpha\_3 == 0:

          res[cur\_x\_id][0] = 1 / ui0\_coeff \* (2 \* h\_y \* d.phi\_3(x[cur\_x\_id], t[cur\_t\_id + 1]) - alpha\_3 \* (4 \* res[cur\_x\_id][1] - res[cur\_x\_id][2]))

        if alpha\_4 == 0:

          res[cur\_x\_id][-1] = 1 / uim\_coeff \* (2 \* h\_y \* d.phi\_4(x[cur\_x\_id], t[cur\_t\_id + 1]) + alpha\_4 \* (4 \* res[cur\_x\_id][-2] - res[cur\_x\_id][-3]))

      A\_2j = -1 / h\_y\*\*2

      B\_2j = (1 + gamma) / tau + 2 / h\_y\*\*2

      C\_2j = -1 / h\_y\*\*2

      for cur\_x\_id in range(1, n - 1):

        A = np.zeros((m-2, m-2))

        A[0][0] = B\_2j + A\_2j / ui0\_coeff \* (-4 \* alpha\_3)

        A[0][1] = C\_2j + A\_2j / ui0\_coeff \* alpha\_3

        for i in range(1, len(A) - 1):

            A[i][i-1] = A\_2j

            A[i][i] = B\_2j

            A[i][i+1] = C\_2j

        A[-1][-2] = A\_2j + C\_2j / uim\_coeff \* (-alpha\_4)

        A[-1][-1] = B\_2j + C\_2j / uim\_coeff \* 4 \* alpha\_4

        B = np.zeros(m-2)

        for cur\_y\_id in range(1, m-1):

          B[cur\_y\_id - 1] = compute\_D\_2j(res\_fraction, x, y, t, cur\_x\_id, cur\_y\_id, cur\_t\_id, tau, h\_x, gamma, N)

        B[0] -= A\_2j / ui0\_coeff \* 2 \* h\_y \* d.phi\_3(x[cur\_x\_id], t[cur\_t\_id + 1])

        B[-1] -= C\_2j / uim\_coeff \* 2 \* h\_y \* d.phi\_4(x[cur\_x\_id], t[cur\_t\_id + 1])

        res[cur\_x\_id, 1:-1] = tridiagonal\_solve(A, B)

      for cur\_y\_id in range(m):

        if alpha\_1 != 0:

          res[0][cur\_y\_id] = 1 / u0j\_coeff \* (2 \* h\_x \* d.phi\_1(y[cur\_y\_id], t[cur\_t\_id + 1]) - alpha\_1 \* (4 \* res[1][cur\_y\_id] - res[2][cur\_y\_id]))

        if alpha\_2 != 0:

          res[-1][cur\_y\_id] = 1 / unj\_coeff \* (2 \* h\_x \* d.phi\_2(y[cur\_y\_id], t[cur\_t\_id + 1]) + alpha\_2 \* (4 \* res[-2][cur\_y\_id] - res[-3][cur\_y\_id]))

      for cur\_x\_id in range(n):

        if alpha\_3 != 0:

          res[cur\_x\_id][0] = 1 / ui0\_coeff \* (2 \* h\_y \* d.phi\_3(x[cur\_x\_id], t[cur\_t\_id + 1]) - alpha\_3 \* (4 \* res[cur\_x\_id][1] - res[cur\_x\_id][2]))

        if alpha\_4 != 0:

          res[cur\_x\_id][-1] = 1 / uim\_coeff \* (2 \* h\_y \* d.phi\_4(x[cur\_x\_id], t[cur\_t\_id + 1]) + alpha\_4 \* (4 \* res[cur\_x\_id][-2] - res[cur\_x\_id][-3]))

      answer.append(res.copy())

    np\_answer = np.array(answer)

    return np\_answer

def plot\_results\_t\_and\_x(solutions, cur\_time, cur\_x, x\_range, y\_range, t\_range, h\_x, h\_y, tau):

    x = np.arange(\*x\_range, h\_x)

    y = np.arange(\*y\_range, h\_y)

    t = np.arange(\*t\_range, tau)

    cur\_t\_id = abs(t - cur\_time).argmin()

    cur\_x\_id = abs(x - cur\_x).argmin()

    plt.figure(figsize=(8, 4))

    for method\_name, solution in solutions.items():

        plt.plot(y, solution[cur\_t\_id][cur\_x\_id], label=method\_name)

    plt.xlabel('y')

    plt.ylabel('U')

    plt.title(f"U(x,y,t) t = {cur\_time}, x = {cur\_x}")

    plt.legend()

    plt.grid()

def plot\_results\_t\_and\_y(solutions, cur\_time, cur\_y, x\_range, y\_range, t\_range, h\_x, h\_y, tau):

    x = np.arange(\*x\_range, h\_x)

    y = np.arange(\*y\_range, h\_y)

    t = np.arange(\*t\_range, tau)

    cur\_t\_id = abs(t - cur\_time).argmin()

    cur\_y\_id = abs(y - cur\_y).argmin()

    plt.figure(figsize=(8, 4))

    for method\_name, solution in solutions.items():

        new\_solution = np.transpose(solution, axes=(0, 2, 1))

        plt.plot(y, new\_solution[cur\_t\_id][cur\_y\_id], label=method\_name)

    plt.xlabel('y')

    plt.ylabel('U')

    plt.title(f"U(x,y,t) t = {cur\_time}, y = {cur\_y}")

    plt.legend()

    plt.grid()

def performing\_a\_variant\_of\_laboratory\_work(l\_1, l\_2, T, N\_x, N\_y, K,

    alpha\_1, beta\_1, alpha\_2, beta\_2, alpha\_3, beta\_3, alpha\_4, beta\_4, N,

    graphics=True

):

    h\_x = (l\_1 - 0) / N\_x

    h\_y = (l\_2 - 0) / N\_y

    tau = (T - 0) / K

    x\_begin = 0

    x\_end = l\_1 + h\_x

    y\_begin = 0

    y\_end = l\_2 + h\_y

    t\_begin = 0

    t\_end = T + tau

    analytical\_solution = get\_analytical\_solution(

        x\_range=(x\_begin, x\_end),

        y\_range=(y\_begin, y\_end),

        t\_range=(t\_begin, t\_end),

        h\_x=h\_x,

        h\_y=h\_y,

        tau=tau

    )

    solutions\_4 = dict()

    solutions\_4["Аналитическое решение"] = analytical\_solution

    MPN = finite\_difference\_schema\_general\_view(

        x\_range=(x\_begin, x\_end),

        y\_range=(y\_begin, y\_end),

        t\_range=(t\_begin, t\_end),

        h\_x=h\_x,

        h\_y=h\_y,

        tau=tau,

        alpha\_1=alpha\_1,

        beta\_1=beta\_1,

        alpha\_2=alpha\_2,

        beta\_2=beta\_2,

        alpha\_3=alpha\_3,

        beta\_3=beta\_3,

        alpha\_4=alpha\_4,

        beta\_4=beta\_4,

        N=N,

        gamma=1

    )

    solutions\_4["Метод переменных направлений"] = MPN

    max\_error\_MPN = max\_abs\_error(MPN, analytical\_solution)

    mean\_error\_MPN = mean\_abs\_error(MPN, analytical\_solution)

    MDSh = finite\_difference\_schema\_general\_view(

        x\_range=(x\_begin, x\_end),

        y\_range=(y\_begin, y\_end),

        t\_range=(t\_begin, t\_end),

        h\_x=h\_x,

        h\_y=h\_y,

        tau=tau,

        alpha\_1=alpha\_1,

        beta\_1=beta\_1,

        alpha\_2=alpha\_2,

        beta\_2=beta\_2,

        alpha\_3=alpha\_3,

        beta\_3=beta\_3,

        alpha\_4=alpha\_4,

        beta\_4=beta\_4,

        N=N,

        gamma=0

    )

    solutions\_4["Метод дробных шагов"] = MDSh

    max\_error\_MDSh = max\_abs\_error(MDSh, analytical\_solution)

    mean\_error\_MDSh = mean\_abs\_error(MDSh, analytical\_solution)

    if graphics == True:

        plot\_results\_t\_and\_x(

            solutions=solutions\_4,

            cur\_x=0,

            cur\_time=0.5,

            x\_range=(x\_begin, x\_end),

            y\_range=(y\_begin, y\_end),

            t\_range=(t\_begin, t\_end),

            h\_x=h\_x,

            h\_y=h\_y,

            tau=tau,

        )

        plot\_results\_t\_and\_x(

            solutions=solutions\_4,

            cur\_x=1,

            cur\_time=0.5,

            x\_range=(x\_begin, x\_end),

            y\_range=(y\_begin, y\_end),

            t\_range=(t\_begin, t\_end),

            h\_x=h\_x,

            h\_y=h\_y,

            tau=tau,

        )

        plot\_results\_t\_and\_y(

            solutions=solutions\_4,

            cur\_y=0.5,

            cur\_time=0.5,

            x\_range=(x\_begin, x\_end),

            y\_range=(y\_begin, y\_end),

            t\_range=(t\_begin, t\_end),

            h\_x=h\_x,

            h\_y=h\_y,

            tau=tau,

        )

    return max\_error\_MPN, mean\_error\_MPN, max\_error\_MDSh, mean\_error\_MDSh

N = 2

l\_1 = 1

l\_2 = 1

T = 2

N\_x = 50

N\_y = 50

K = 100

alpha\_1 = 0

beta\_1 = 1

alpha\_2 = 0

beta\_2 = 1

alpha\_3 = 0

beta\_3 = 1

alpha\_4 = 0

beta\_4 = 1

graphics = True

max\_error\_MPN, mean\_error\_MPN, max\_error\_MDSh, mean\_error\_MDSh = performing\_a\_variant\_of\_laboratory\_work(

    l\_1=l\_1,

    l\_2=l\_2,

    T=T,

    N\_x=N\_x,

    N\_y=N\_y,

    K=K,

    alpha\_1=alpha\_1,

    beta\_1=beta\_1,

    alpha\_2=alpha\_2,

    beta\_2=beta\_2,

    alpha\_3=alpha\_3,

    beta\_3=beta\_3,

    alpha\_4=alpha\_4,

    beta\_4=beta\_4,

    N=N,

    graphics=graphics

)

print('Метод переменных направлений')

print(f'MaxAE = {max\_error\_MPN}')

print(f'MeanAE = {mean\_error\_MPN}')

print()

print('Метод дробных шагов')

print(f'MaxAE = {max\_error\_MDSh}')

print(f'MeanAE = {mean\_error\_MDSh}')

Nx\_values = [5, 10, 25, 50]

Ny\_values = [5, 10, 25, 50]

K\_values = [25, 50, 100, 150]

errors\_hx = {'max': [], 'mean': []}

errors\_hy = {'max': [], 'mean': []}

errors\_tau = {'max': [], 'mean': []}

errors\_hx\_max = []

errors\_hx\_mean = []

for N\_x in Nx\_values:

    h\_x = (l\_1 - 0) / N\_x

    max\_error\_MPN, mean\_error\_MPN, max\_error\_MDSh, mean\_error\_MDSh = performing\_a\_variant\_of\_laboratory\_work(

        l\_1=l\_1,

        l\_2=l\_2,

        T=T,

        N\_x=N\_x,

        N\_y=20,

        K=100,

        alpha\_1=alpha\_1,

        beta\_1=beta\_1,

        alpha\_2=alpha\_2,

        beta\_2=beta\_2,

        alpha\_3=alpha\_3,

        beta\_3=beta\_3,

        alpha\_4=alpha\_4,

        beta\_4=beta\_4,

        graphics=False,

        N=2

    )

    errors\_hx\_max.append((max\_error\_MPN, max\_error\_MDSh))

    errors\_hx\_mean.append((mean\_error\_MPN, mean\_error\_MDSh))

errors\_hx\_max = errors\_hx\_max

errors\_hx\_mean = errors\_hx\_mean

for i in range(len(Nx\_values)):

    curr\_N\_x = Nx\_values[i]

    h\_x = (l\_1 - 0) / curr\_N\_x

    errors\_hx['max'].append((h\_x, errors\_hx\_max[i][0], errors\_hx\_max[i][1]))

    errors\_hx['mean'].append((h\_x, errors\_hx\_mean[i][0], errors\_hx\_mean[i][1]))

errors\_hy\_max = []

errors\_hy\_mean = []

for N\_y in Ny\_values:

    h\_y = (l\_2 - 0) / N\_y

    max\_error\_MPN, mean\_error\_MPN, max\_error\_MDSh, mean\_error\_MDSh = performing\_a\_variant\_of\_laboratory\_work(

        l\_1=l\_1,

        l\_2=l\_2,

        T=T,

        N\_x=5,

        N\_y=N\_y,

        K=25,

        alpha\_1=alpha\_1,

        beta\_1=beta\_1,

        alpha\_2=alpha\_2,

        beta\_2=beta\_2,

        alpha\_3=alpha\_3,

        beta\_3=beta\_3,

        alpha\_4=alpha\_4,

        beta\_4=beta\_4,

        graphics=False,

        N=2

    )

    errors\_hy\_max.append((max\_error\_MPN, max\_error\_MDSh))

    errors\_hy\_mean.append((mean\_error\_MPN, mean\_error\_MDSh))

errors\_hy\_max = errors\_hy\_max

errors\_hy\_mean = errors\_hy\_mean

for i in range(len(Ny\_values)):

    curr\_N\_y = Ny\_values[i]

    h\_y = (l\_2 - 0) / curr\_N\_y

    errors\_hy['max'].append((h\_y, errors\_hy\_max[i][0], errors\_hy\_max[i][1]\*m))

    errors\_hy['mean'].append((h\_y, errors\_hy\_mean[i][0], errors\_hy\_mean[i][1]\*m))

for K in K\_values:

    tau = (T - 0) / K

    max\_error\_MPN, mean\_error\_MPN, max\_error\_MDSh, mean\_error\_MDSh = performing\_a\_variant\_of\_laboratory\_work(

        l\_1=l\_1,

        l\_2=l\_2,

        T=T,

        N\_x=50,

        N\_y=50,

        K=K,

        alpha\_1=alpha\_1,

        beta\_1=beta\_1,

        alpha\_2=alpha\_2,

        beta\_2=beta\_2,

        alpha\_3=alpha\_3,

        beta\_3=beta\_3,

        alpha\_4=alpha\_4,

        beta\_4=beta\_4,

        graphics=False,

        N=2

    )

    errors\_tau['max'].append((tau, max\_error\_MPN, max\_error\_MDSh))

    errors\_tau['mean'].append((tau, mean\_error\_MPN, mean\_error\_MDSh))

def plot\_errors(errors, xlabel, title):

    h\_values = [item[0] for item in errors['max']]

    max\_errors\_MPN = [item[1] for item in errors['max']]

    max\_errors\_MDSh = [item[2] for item in errors['max']]

    mean\_errors\_MPN = [item[1] for item in errors['mean']]

    mean\_errors\_MDSh = [item[2] for item in errors['mean']]

    plt.figure(figsize=(8, 4))

    plt.plot(h\_values, max\_errors\_MPN, label='Метод переменных направлений MaxAE')

    plt.plot(h\_values, mean\_errors\_MPN, label='Метод переменных направлений MeanAE')

    plt.plot(h\_values, max\_errors\_MDSh, label='Метод дробных шагов MaxAE')

    plt.plot(h\_values, mean\_errors\_MDSh, label='Метод дробных шагов MeanAE')

    plt.xlabel(xlabel)

    plt.ylabel("err")

    plt.title(title)

    plt.legend()

    plt.grid()

plot\_errors(errors\_hx, xlabel="h\_x", title="График ошибок от h\_x")

plot\_errors(errors\_hy, xlabel="h\_y", title="График ошибок от h\_y")

plot\_errors(errors\_tau, xlabel="tau", title="График ошибок от tau")

plt.show()

## **Пример работы:**

## Выводы:

При выполнении лабораторных работ были изучены многие численные методы, позволяющие решить различный спектр задач. Также во время выполнения был освоен язык программирования Python и пакеты matplotlib и numpy, работать с ними оказалось разительно легче, чем писать все с нуля на Go.