$U(x,t)$ τ, h

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.

Москва, 2025

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,
```

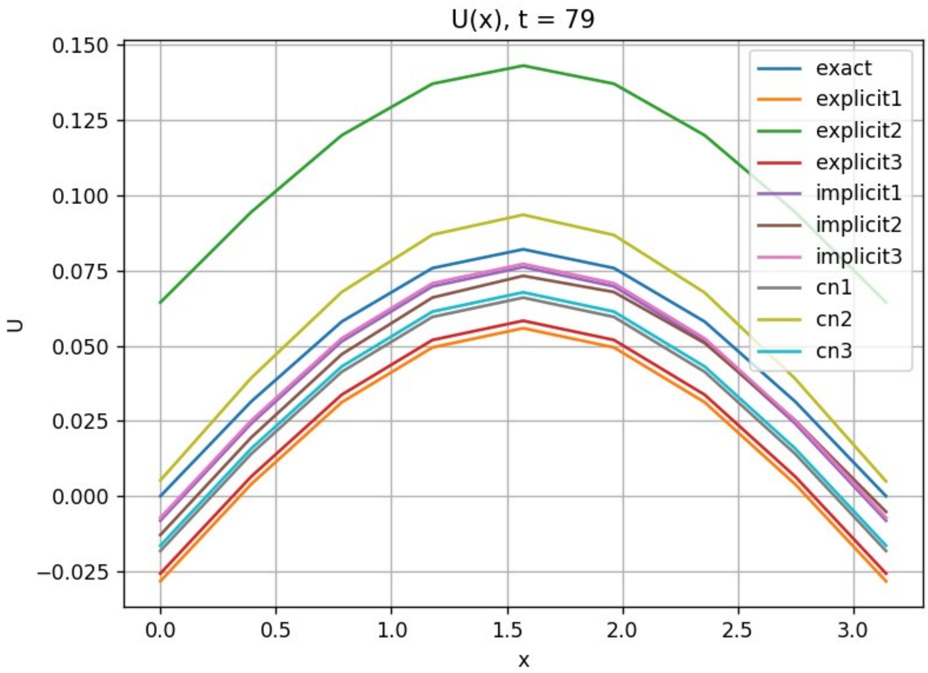
```
                    (self.sigma * self.h * self.D.phi_l(i * self.tau) + (curr[-2] + 0.5 * self.tau *
self.D.g((self.N-2) * self.h, i * self.tau)) +
                    0.5 * self.sigma * (curr[-3] - 2*curr[-2] + curr[-1] + self.h ** 2 *
self.D.g((self.N-2) * self.h, (i-1) * self.tau))
          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

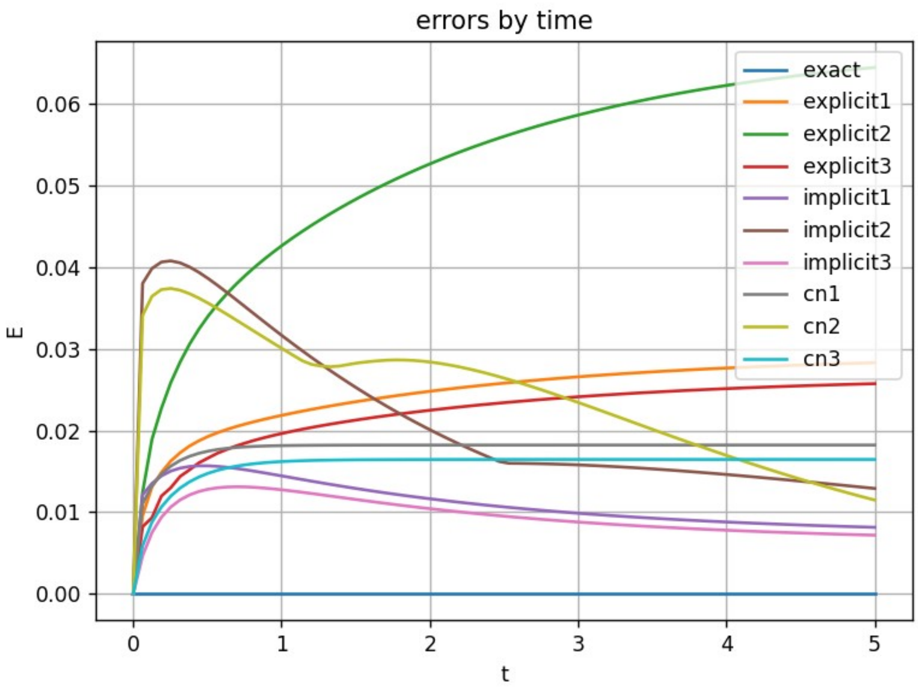
```python
def MAE(numeric, analytic):
  return np.abs(numeric - analytic).max(axis=1)


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}")
```

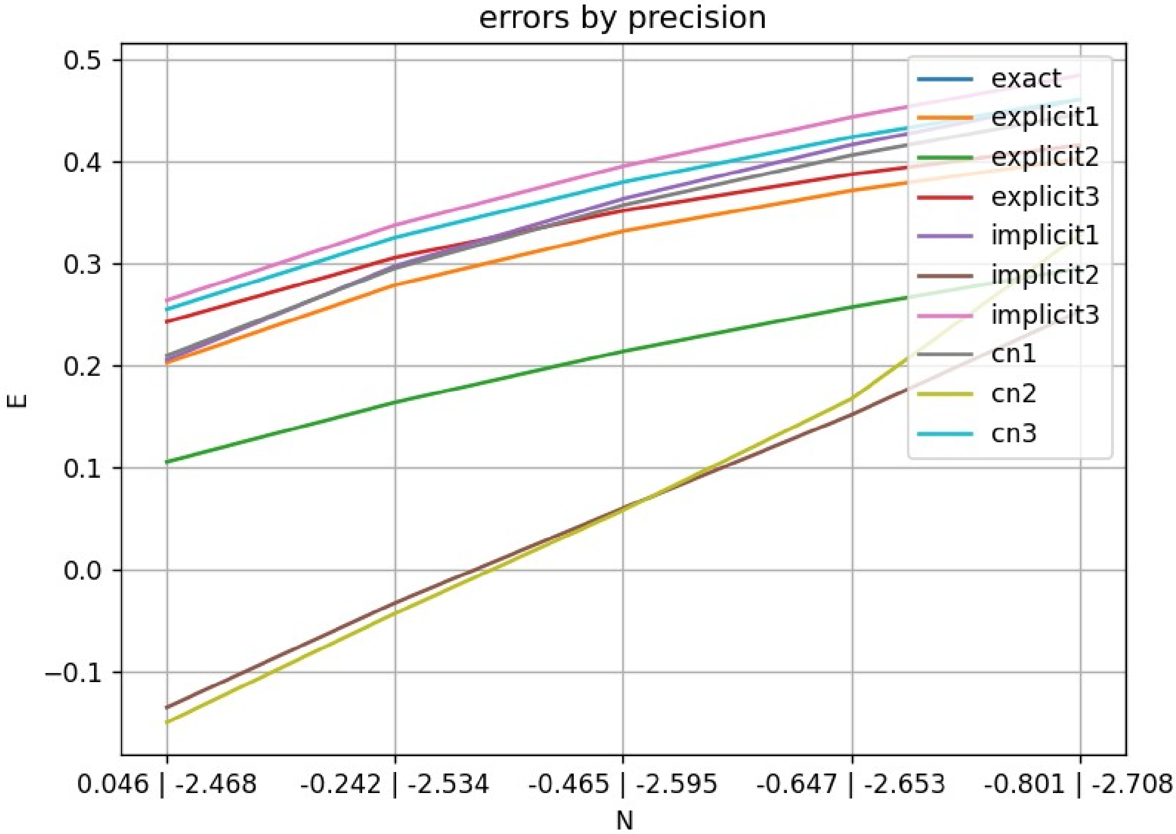
```
        x = [i * self.h for i in range(self.N - 1)]
        x.append(self.L)
        x = np.array(x)

        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:






errors by precision
0.5
0.4
0.3
0.2
0.1
0.0
−0.1
E
0.046 | -2.468
-0.242 | -2.534
-0.465 | -2.595
-0.647 | -2.653
-0.801 | -2.708
N
exact
explicit1
explicit2
explicit3
implicit1
implicit2
implicit3
cn1
cn2
cn3

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:

```
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
```

```python
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,
```

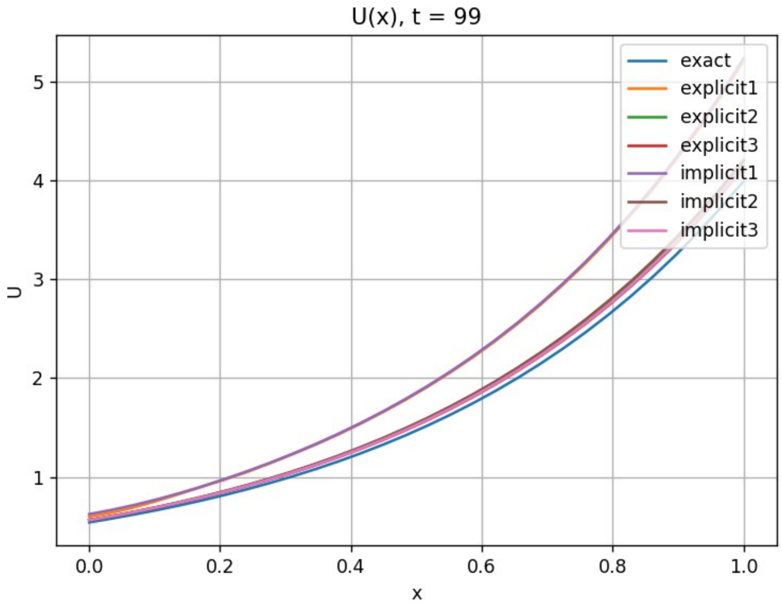
```
                    -(2 + 5 * self.h**2 + 4 * self.h + 1 / self.sigma),
                    2,
                    (-2 * u[k - 1][0] + u[k - 2][0]) / self.sigma
        )

    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
```

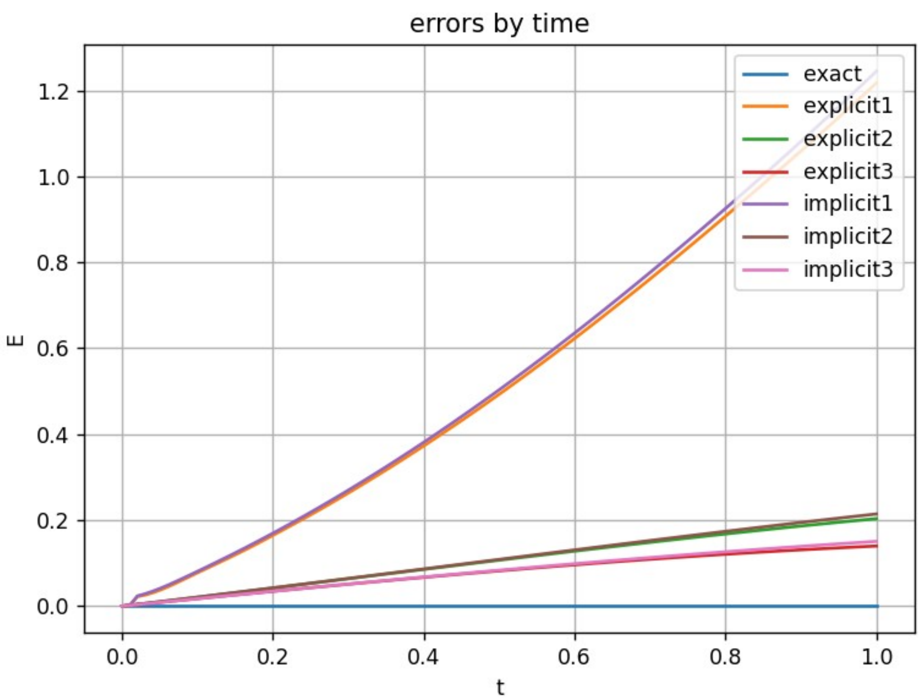
```python
                d[j] = (u[i - 2][j] - 2*u[i - 1][j]) / self.sigma
            a[0], b[0], c[0], d[0] = self.zero_aprox()(i, u)
            a[-1], b[-1], c[-1], d[-1] = self.l_aprox()(i, u)

            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))
```

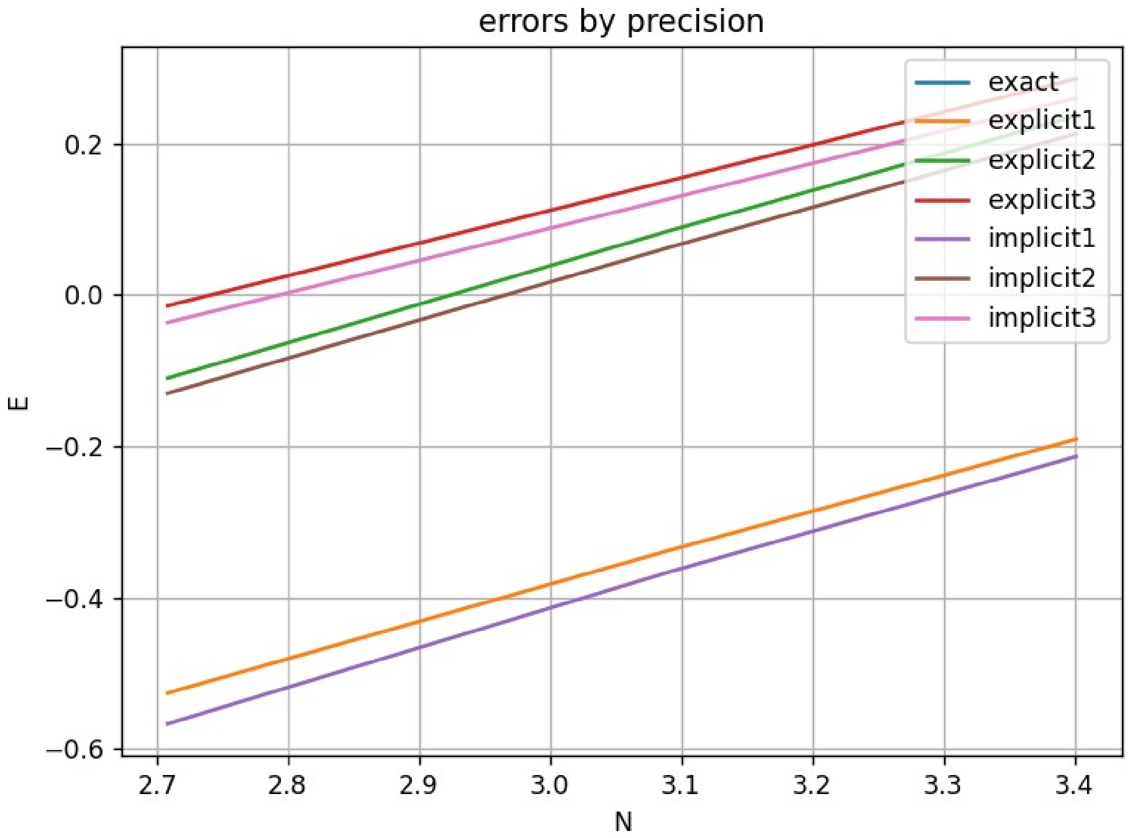
```
    def plot_solve(self, *args):
        fig, ax1 = plt.subplots(1, 1, figsize=(7, 5))
        ax1.set_title(f"U(x), t = {self.K - 1}")
        x = [i * self.h for i in range(self.N - 1)]
        x.append(self.L)
        x = np.array(x)

        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:






errors by precision
exact
explicit1
explicit2
explicit3
implicit1
implicit2
implicit3
0.2
0.0
−0.2
−0.4
−0.6
E
2.7
2.8
2.9
3.0
3.1
3.2
3.3
3.4
N

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:

```
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),
    ]  # OO - 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:
```

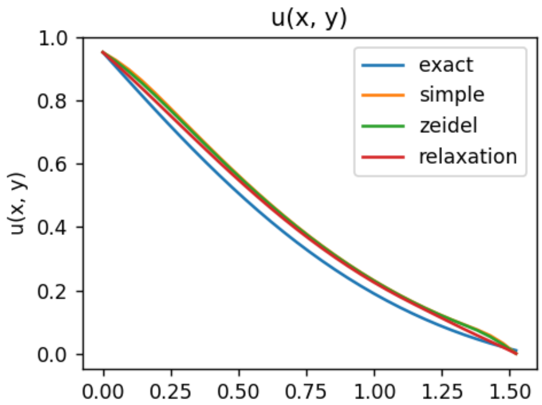
```
            ui0_coef = 2 * h_y * d.beta_3 - 3 * d.alpha_3
            res[j][0] = 1 / ui0_coef * (
                2 * h_y * d.phi_3(x[j]) - d.alpha_3 * (4 * res[j][1] - res[j][2]))
        if d.alpha_4 != 0:
            uim_coeff = 2 * h_y * d.beta_4 + 3 * d.alpha_4
            res[j][-1] = 1 / 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)}')
```

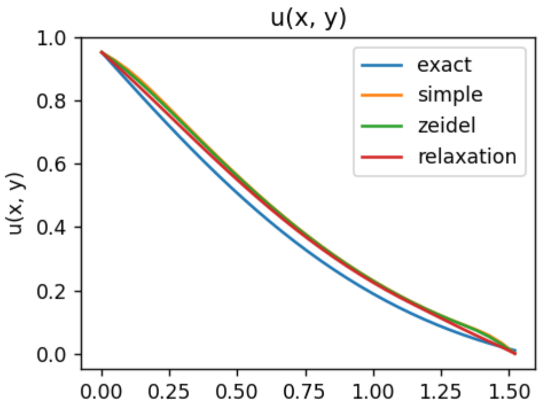
```
print()

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()
```

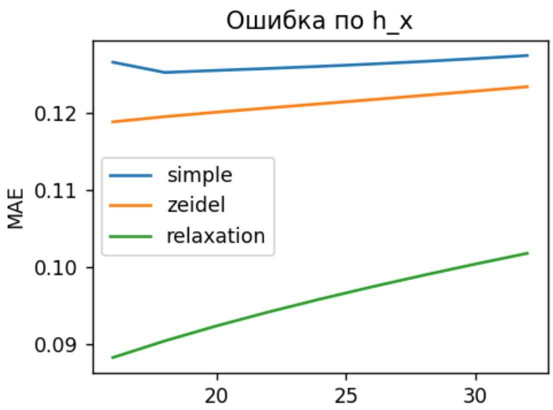
```
fig2, axs2 = plt.subplots(1, 2, figsize=(9, 3))
h_x_s = np.arange(x_steps // 2, x_steps, 2)
h_y_s = np.arange(y_steps // 2, y_steps, 2)

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')
```

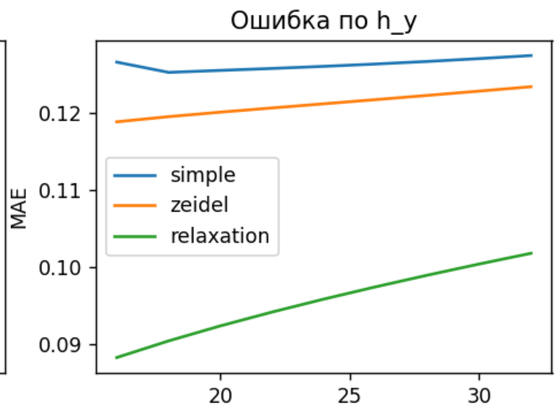
```
    axs2[0].set_xlabel('h_x')
    axs2[0].set_ylabel('MAE')
    axs2[0].legend()

    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]))
```

```python
        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,
```

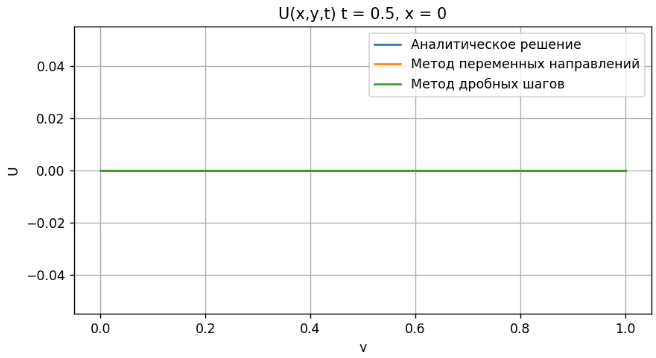
```
    alpha_4=alpha_4,
    beta_4=beta_4,
    N=N,
    graphics=graphics
)

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
    )
```

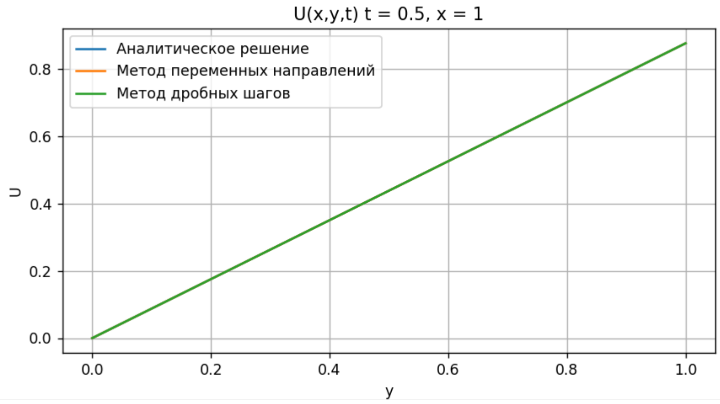
```
    errors_hx_max.append((max_error_MPN, max_error_MDSh))
    errors_hx_mean.append((mean_error_MPN, mean_error_MDSh))

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))
```

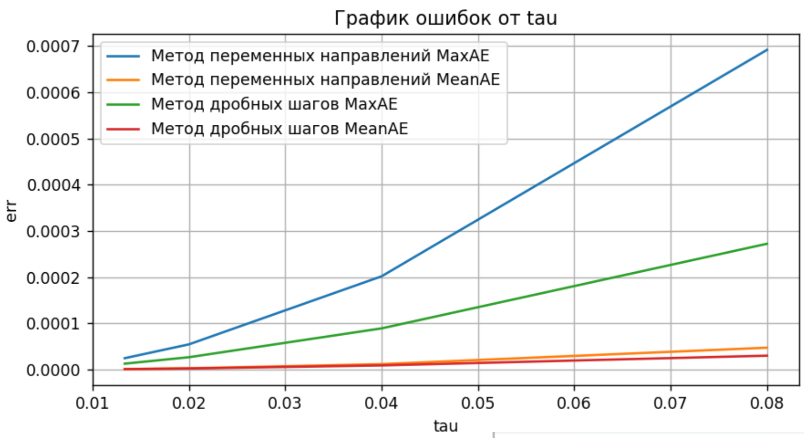
```
for K in K_values:
    tau = (T - 0) / K

    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()
```

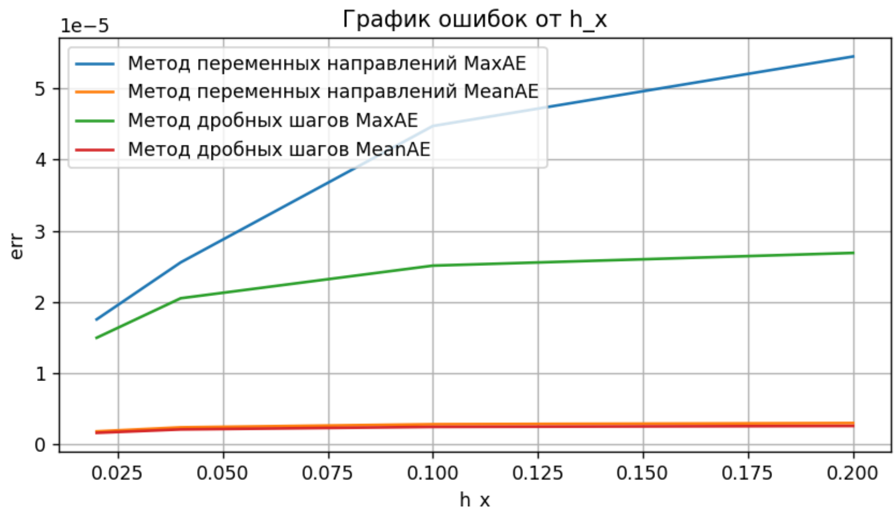
```
plot_errors(errors_hx, xlabel="h_x", title="График ошибок от h_x")
plot_errors(errors_hy, xlabel="h_y", title="График ошибок от h_y")
plot_errors(errors_tau, xlabel="tau", title="График ошибок от tau")
plt.show()
```

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.