Московский авиационный институт
(Национальный исследовательский университет)

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

Лабораторные работы по дисциплине «Численные методы»

Студент: Суров В.О.
Группа: М8О-402Б-20
Преподователь: Пивоваров Д.Е.

Москва, 2024

Лабораторная работа №5

1. Задание

Используя явную и неявную конечно-разностные схемы, а также схему Кранка - Николсона, решить начально-краевую задачу для дифференциального уравнения параболического типа. Осуществить реализацию трех вариантов аппроксимации граничных условий, содержащих производные: двухточечная аппроксимация с первым порядком, трехточечная аппроксимация со вторым порядком, двухточечная аппроксимация со вторым порядком. В различные моменты времени вычислить погрешность численного решения путем сравнения результатов с приведенным в задании аналитическим решением *U(x, t)*. Исследовать зависимость погрешности от сеточных параметров τ, *h*.

Вариант:

6.

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

$$u(0,t) = \sin t,$$

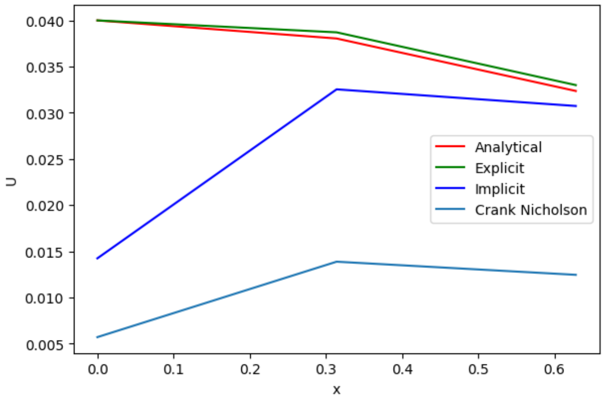
$$u_x(\frac{\pi}{2},t) = -\sin t,$$

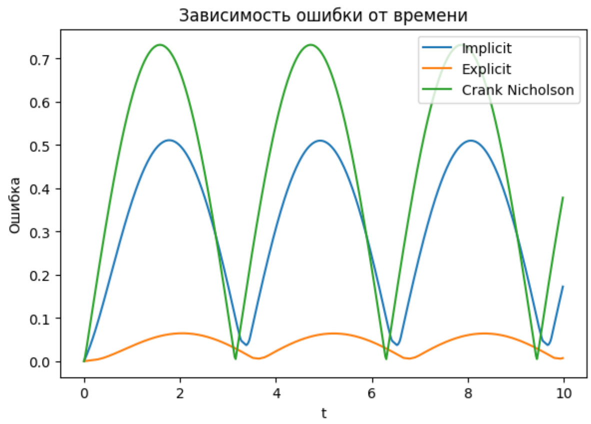
$$u(x,0) = 0,$$

Аналитическое решение: $U(x,t) = \sin t \cos x$.

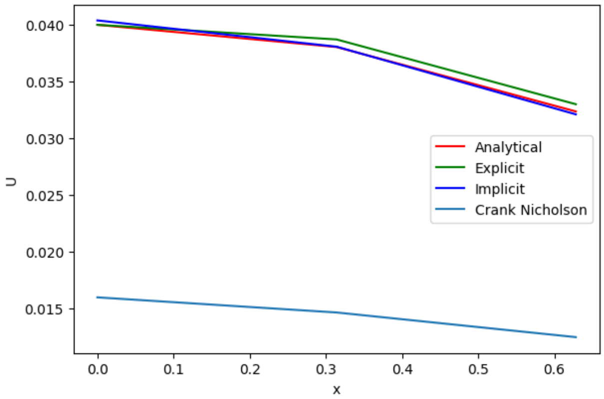
2. Результат

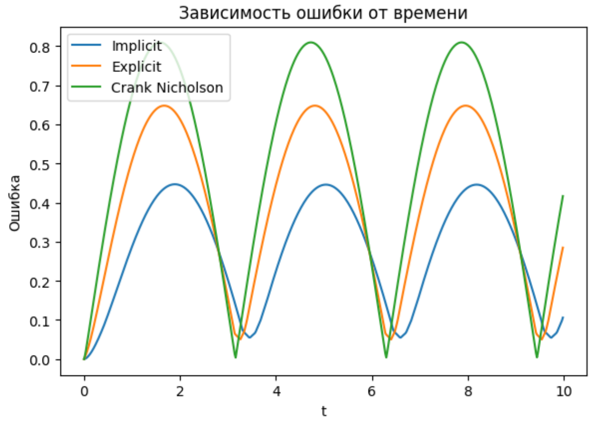
1) Двухточеченая аппроксимация с первым порядком



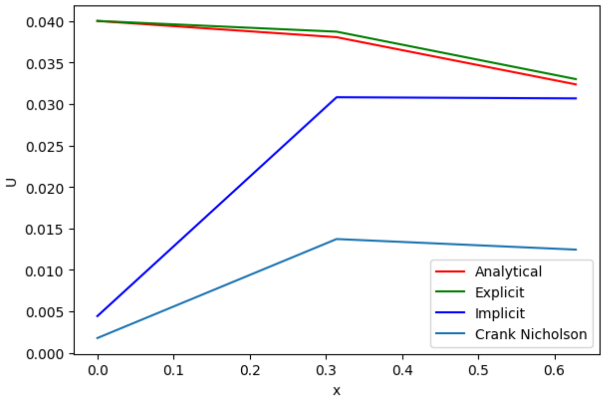


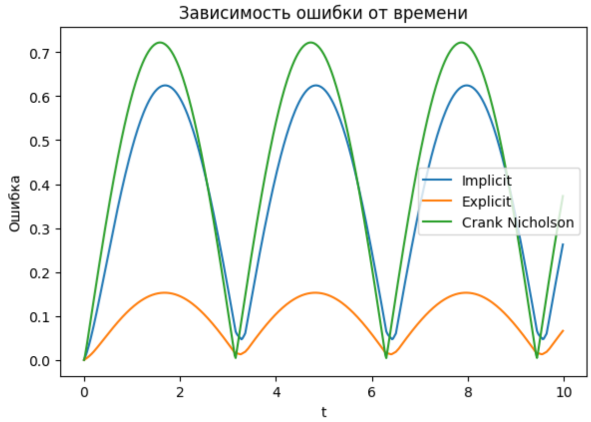
2) Трёхточечная аппроксимация со вторым порядком





3) Двухточечная аппроксимация со вторым порядком





3. Вывод

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

4. Код

```
5. import numpy as np

   args = {
       'l': (np.pi)/2,
       'Psi': lambda x: np.sin(0),
       'Function': lambda x, t: np.cos(x)*(np.sin(t)+np.cos(t)),
       'Phi0': lambda t: np.sin(t),
       'PhiL': lambda t: -np.sin(t),
       'AnaliticalSolution': lambda x, t: np.sin(t)*np.cos(x),
       'type': '1-1',
       'algorithm': 'Implicit'
   }

   class Parabolic:
       def __init__(self, args):
           functionName = ''
           for name, value in args.items():
               setattr(self, name, value)
           for word in args['algorithm'].split():
               first = word[0].upper()
               word = word[1:].lower()
```

```
            functionName += first + word
        try:
            self.SolveFunction = getattr(self, functionName)
        except:
            raise Exception("Данный тип не поддерживается, выберите Implicit, Explicit или Crank Nicholson")

    def SolveThis(self, N, K, T):
        self.h = self.l / N
        self.tau = T / K
        self.sigma = self.tau / (self.h ** 2)
        return self.SolveFunction(N, K, T)

    def AnaliticalSolution(self, N, K, T):
        self.h = self.l / N;
        self.tau = T / K;
        self.u = np.zeros((K, N))
        for k in range(K):
            for j in range(N):
                self.u[k][j] = self.AnaliticalSolution(j * self.h, k * self.tau)
        return self.u

    def RunThrough(self):
        size = len(self.a)
        p = np.zeros(size)
        q = np.zeros(size)
        p[0] = (-self.c[0] / self.b[0])
        q[0] = (self.d[0] / self.b[0])

        for i in range(1, size):
            p[i] = -self.c[i] / (self.b[i] + self.a[i] * p[i - 1])
            q[i] = (self.d[i] - self.a[i] * q[i - 1]) / (self.b[i] + self.a[i] * p[i - 1])

        x = np.zeros(size)
        x[-1] = q[-1]

        for i in range(size - 2, -1, -1):
            x[i] = p[i] * x[i + 1] + q[i]

        return x

    def Implicit(self, N, K, T):
        self.a = np.zeros(N)
        self.b = np.zeros(N)
        self.c = np.zeros(N)
        self.d = np.zeros(N)
        self.u = np.zeros((K, N))

        for i in range(1, N - 1):
            self.u[0][i] = self.Psi(i * self.h)
        self.u[0][-1] = 0

        for k in range(1, K):
            for j in range(1, N - 1):
                self.a[j] = self.sigma
                self.b[j] = -(1 + 2 * self.sigma)
                self.c[j] = self.sigma
                self.d[j] = -self.u[k - 1][j] - self.tau * self.Function(j * self.h, k * self.tau)

            self.a[0] = 0
            self.b[0] = -(1 + 2 * self.sigma)
```

```
            self.c[0] = self.sigma
            self.a[-1] = self.sigma
            self.b[-1] = -(1 + 2 * self.sigma)
            self.c[-1] = 0

            if self.type == '1-1':
                self.d[0] = -(self.u[k - 1][0] + self.sigma * self.Phi0(k *
self.tau))
                self.d[-1] = -(self.u[k - 1][-1] + self.sigma * self.PhiL(k *
self.tau))

            elif self.type == '2-2':
                self.d[0] = -(self.u[k - 1][0] + self.sigma * self.Phi0(k *
self.tau)) - self.tau * self.Function(0, k * self.tau)
                self.d[-1] = -(self.u[k - 1][-1] + self.sigma * self.PhiL(k *
self.tau)) - self.tau * self.Function((N - 1) * self.h, k * self.tau)

            elif self.type == '2-3':
                self.d[0] = -((1 - self.sigma) * self.u[k - 1][1] +
self.sigma / 2 * self.u[k - 1][0]) - self.tau * Function(0, k * self.tau) -
self.sigma * self.Phi0(k * self.tau)
                self.d[-1] = self.PhiL(k * self.tau) + self.Function((N - 1)
* self.h, k * self.tau) * self.h / (2 * self.tau) * self.u[k - 1][-1]

            self.u[k] = self.RunThroughMethod()

        return self.u

    def Explicit(self, N, K, T):
        self.u = np.zeros((K, N))
        for j in range(1, N - 1):
            self.u[0][j] = self.Psi(j * self.h)

        for k in range(1, K):
            self.u[k][0] = self.Phi0(k * self.tau)
            for j in range(1, N - 1):
                self.u[k][j] = self.sigma * self.u[k - 1][j + 1] + (1 - 2 *
self.sigma) * self.u[k - 1][j] + self.sigma * self.u[k - 1][j - 1] + self.tau
* self.Function(j * self.h, k * self.tau)

            if self.type == '1-1':
                self.u[k][-1] = self.u[k][-2] + self.PhiL(k * self.tau) *
self.h
            elif self.type == '2-2':
                self.u[k][-1] = self.PhiL(k * self.tau)
            elif self.type == '2-3':
                self.u[k][-1] = (self.PhiL(k * self.tau) + self.u[k][-2] /
self.h + 2 * self.tau * self.u[k - 1][-1] / self.h) / (1 / self.h + 2 *
self.tau / self.h)

        return self.u

    def Crank_Nicholson(self, N, K, T):
        theta = 0.5
        self.a = np.zeros(N)
        self.b = np.zeros(N)
        self.c = np.zeros(N)
        self.d = np.zeros(N)
        uImplisit = np.zeros(N)
        self.u = np.zeros((K, N))
        for j in range(1, N - 1):
            self.u[0][j] = self.Psi(j * self.h)

        for k in range(1, K):
```

```
            for j in range(1, N - 1):
                self.a[j] = self.sigma
                self.b[j] = -(1 + 2 * self.sigma)
                self.c[j] = self.sigma
                self.d[j] = -self.u[k - 1][j] - self.tau * self.Function(j *
self.h, k * self.tau)

            self.a[0] = 0
            self.b[0] = -(1 + 2 * self.sigma)
            self.c[0] = self.sigma
            self.a[-1] = self.sigma
            self.b[-1] = -(1 + 2 * self.sigma)
            self.c[-1] = 0

            if self.type == '1-1':
                self.d[0] = -(self.u[k - 1][0] + self.sigma * self.Phi0(k *
self.tau))
                self.d[-1] = -(self.u[k - 1][-1] + self.sigma * self.PhiL(k *
self.tau))

            elif self.type == '2-2':
                self.d[0] = -(self.u[k - 1][0] + self.sigma * self.Phi0(k *
self.tau)) - self.tau * self.Function(0, k * self.tau)
                self.d[-1] = -(self.u[k - 1][-1] + self.sigma * self.PhiL(k *
self.tau)) - self.tau * self.Function((N - 1) * self.h, k * self.tau)

            elif self.type == '2-3':
                self.d[0] = -((1 - self.sigma) * self.u[k - 1][1] +
self.sigma / 2 * self.u[k - 1][0]) - self.tau * Function(0, k * self.tau) -
self.sigma * self.Phi0(k * self.tau)
                self.d[-1] = self.PhiL(k * self.tau) + self.Function((N - 1)
* self.h, k * self.tau) * self.h / (2 * self.tau) * self.u[k - 1][-1]

            uImplisit = self.RunThroughMethod()

            uExplisit = np.zeros(N)
            for j in range(1, N - 1):
                uImplisit[j] = self.sigma * self.u[k - 1][j + 1] + (1 - 2 *
self.sigma) * self.u[k - 1][j] + self.sigma * self.u[k - 1][j - 1] + self.tau
* self.Function(j * self.h, k * self.tau)

            if self.type == '1-1':
                uImplisit[-1] = self.u[k][-2] + self.PhiL(k * self.tau) *
self.h
            elif self.type == '2-2':
                uImplisit[-1] = self.PhiL(k * self.tau)
            elif self.type == '2-3':
                uImplisit[-1] = (self.PhiL(k * self.tau) + self.u[k][-2] /
self.h + 2 * self.tau * self.u[k - 1][-1] / self.h) / (1 / self.h + 2 *
self.tau / self.h)

            for j in range(N):
                self.u[k][j] = theta * uImplisit[j] + (1 - theta) *
uExplisit[j]

        return self.u

algorithms = ('Implicit', 'Explicit', 'Crank Nicholson')
T, K, N = 10, 500, 5

answers = dict()
solver = ParabolicSolver(args)
analytic = solver.AnaliticalSolutionMatrix(N, K, T)
answers['Analytic'] = analytic
```

```python
for algorithm in algorithms:
    args['algorithm'] = algorithm
    solver = ParabolicSolver(args)
    numeric = solver.Solve(N, K, T)
    answers[algorithm] = numeric

import matplotlib.pyplot as plt


def GetError(numeric, analytic):
    err = []
    error = [[abs(i - j) for i, j in zip(x, y)] for x, y in zip(numeric,
analytic)]
    for i in range(len(error)):
        tmp = 0
        for j in error[i]:
            tmp += j
        err.append(tmp / len(error[i]))
    return err

def Draw2DCharts(answers, N, K, T, time=2):
    x = np.arange(0, np.pi / 2, np.pi / 2 / N)
    t = np.arange(0, T, T / K)
    z1 = np.array(answers['Analytic'])
    z2 = np.array(answers['Explicit'])
    z3 = np.array(answers['Implicit'])
    z4 = np.array(answers['Crank Nicholson'])

    figure = plt.figure(figsize=(15, 10))
    axes = figure.add_subplot(221)
    plt.plot(x[0:-2], z1[time][0:-2], color='r', label='Analytical')
    plt.plot(x[0:-2], z2[time][0:-2], color='g', label='Explicit')
    plt.plot(x[0:-2], z3[time][0:-2], color='b', label='Implicit')
    plt.plot(x[0:-2], z4[time][0:-2], label='Crank Nicholson')
    plt.legend(loc='best')
    plt.ylabel('U')
    plt.xlabel('x')
    axes = figure.add_subplot(222)
    plt.title('Зависимость ошибки от времени')
    for method in algorithms:
        plt.plot(t, GetError(answers[method], answers['Analytic']),
label=method)
    plt.legend(loc='best')
    plt.ylabel('Ошибка')
    plt.xlabel('t')
    plt.show()

Draw2DCharts(answers, N, K, T)
```

Лабораторная работа №6

1. 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 U(x, t). Исследовать зависимость погрешности от сеточных параметров τ, h.

Вариант:

6.

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

$$u(0,t) = \cos(2t),$$

$$u(\frac{\pi}{2}, t) = 0,$$

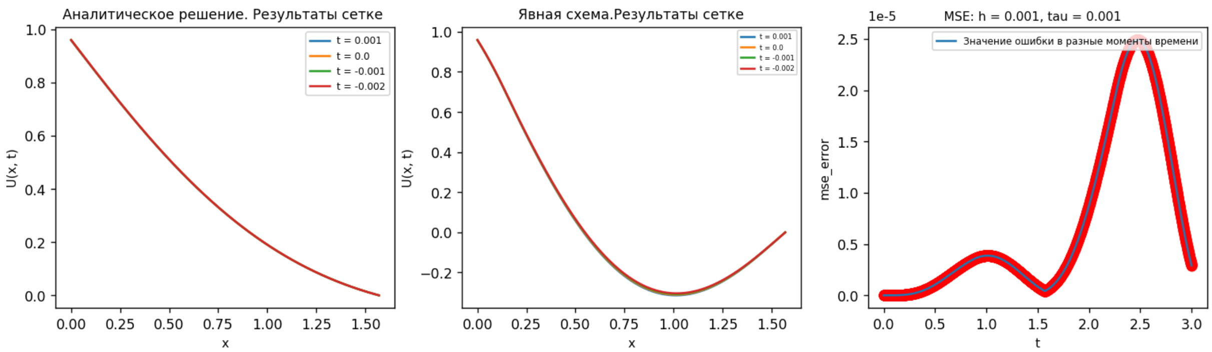
$$u(x,0) = \exp(-x)\cos x,$$

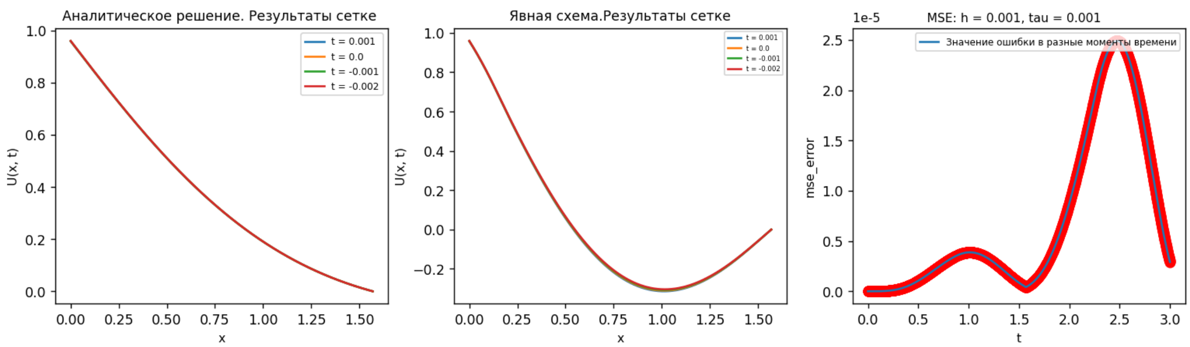
$$u_t(x,0) = 0.$$

Аналитическое решение: $U(x,t) = \exp(-x)\cos x\cos(2t)$

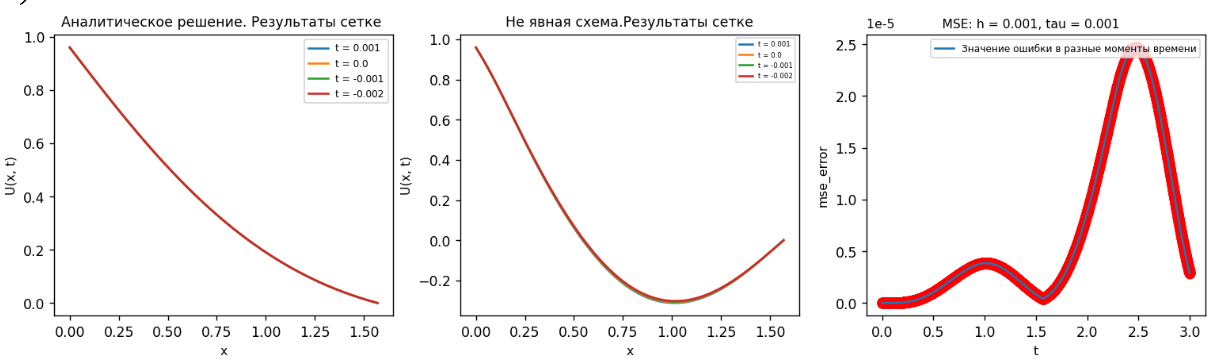
2. Результат

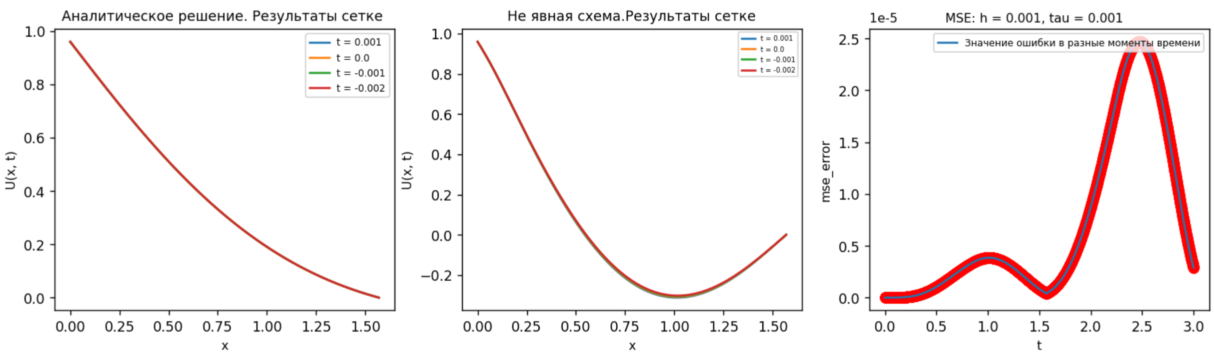
1) Явная схема





2) Неявная схема





3. Вывод

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

4. Код

```
5. import numpy as np
   import matplotlib
   import matplotlib.pyplot as plt

   matplotlib.rcParams['figure.figsize'] = (15, 12)

   def AnaliticalSolution(x, t):
       return np.exp(-x) * np.cos(x) * np.cos(2 * t)

   def u_x_0(x):
       return np.exp(-x) * np.cos(x)

   def u_x_0t(x):
       return 0

   def u_r(t):
       return 0

   def u_l(t):
       return np.cos(2 * t)

   def Compute_Analytical(time_p, l, h_steps):
       node = int(np.pi / (2 * h_steps)) + 1
       grid = np.zeros((l, node))
       for i in range(l):
           for j in range(node):
               x_val = j * h_steps
               t_val = i * time_p
               grid[i, j] = analitic(x_val, t_val)
       return grid

   def Explicit(time_p, l, h_steps, start_aprox):
       node = int(np.pi / (2 * h_steps)) + 1
       grid = np.zeros((l, node))
       for i in range(l):
           grid_ = np.zeros(node)
```

```
        if i == 0:
            for j in range(node):
                grid_[j] = u_x_0(j * h_steps)
        elif i == 1:
            for j in range(node):
                if start_aprox == 1:
                    grid_[j] = grid[i - 1][j] + u_x_0t(j * h_steps) * time_p
                elif start_aprox == 2:
                    term = ((-u_x_0(j * h_steps)) * (time_p ** 2)) / 2
                    grid_[j] = grid[i - 1][j] + u_x_0t(j * h_steps) * time_p
+ term
        else:
            grid_[0] = u_l(i * time_p)
            for j in range(1, node - 1):
                term = (time_p ** 2) / (h_steps ** 2) * (grid[i - 1][j + 1] -
2 * grid[i - 1][j] + grid[i - 1][j - 1])
                grid_[j] = term + 2 * grid[i - 1][j] - grid[i - 2][j]
            grid_[-1] = u_r(i * time_p)
        grid[i] = grid_
    return grid

def Implicit(time_p, l, h_steps, start_aprox):
    node = int(np.pi / (2 * h_steps)) + 1
    grid = np.zeros((l, node))
    alpha = np.zeros(node - 1)
    beta = np.zeros(node - 1)
    a = -(time_p ** 2) / (h_steps ** 2)
    b = 1 + (2 * (time_p ** 2) / (h_steps ** 2))
    c = a
    for i in range(l):
        grid_ = np.zeros(node)
        if i == 0:
            for j in range(node):
                grid_[j] = u_x_0(j * h_steps)
        elif i == 1:
            for j in range(node):
                if start_aprox == 1:
                    grid_[j] = grid[i - 1][j] + u_x_0t(j * h_steps) * time_p
                elif start_aprox == 2:
                    term = ((-u_x_0(j * h_steps)) * (time_p ** 2)) / 2
                    grid_[j] = grid[i - 1][j] + u_x_0t(j * h_steps) * time_p
+ term
        else:
            beta[0] = u_l(i * time_p)
            for j in range(1, node - 1):
                alpha[j] = -a / (b + c * alpha[j - 1])
                beta[j] = (2 * grid[i - 1][j] - grid[i - 2][j] - c * beta[j -
1]) / (b + c * alpha[j - 1])
            grid_[node - 1] = u_r(i * time_p)
            for j in range(node - 2, -1, -1):
                grid_[j] = grid_[j + 1] * alpha[j] + beta[j]
        grid[i] = grid_
    return grid

def Error(res, analitic):
    er = max(abs(res - analitic) ** 2)
    return er

def UpdatePlot(analitic, time_p, l, start_aprox, h_steps, cur_display_mode):
    if cur_display_mode == 1:
        name = 'Явная схема.'
        res = explicit(time_p, l, h_steps, start_aprox)
    elif cur_display_mode == 2:
        name = 'Не явная схема.'
```

```python
        res = implicit(time_p, l, h_steps, start_aprox)

    plot_function(name, analitic, res, time_p, l, h_steps)
    plt.draw()

def plot_function(name, analitic, res, time_p, l, h_steps,
num_last_layers=4):
    node = int(np.pi / (2 * h_steps)) + 1
    X = np.array([i * h_steps for i in range(node)])

    for layer in analitic[-num_last_layers:]:
        fst.plot(X, layer)

    fst.legend(['t = {}'.format(time_p - i * time_p) for i in
range(num_last_layers)], fontsize=7, loc='upper right')

    scd.set_title(name + 'Результаты сетке', fontsize=10)
    scd.set_xlabel('x', fontsize=9)
    scd.set_ylabel('U(x, t)', fontsize=9)

    for layer in res[-num_last_layers:]:
        scd.plot(X, layer)
    scd.legend(['t = {}'.format(time_p - i * time_p) for i in
range(num_last_layers)], fontsize=5, loc='upper right')

    T = np.array([i * time_p for i in range(l)])
    MSE_error = (np.array([error(i, j) for i, j in zip(res, analitic)]))

    thd.set_title('1e-5            MSE: h = {}, tau = {}'.format(h_steps,
time_p), fontsize=9, pad=5, loc='left')
    thd.set_xlabel('t', fontsize=9)
    thd.set_ylabel('mse_error', fontsize=9)
    thd.plot(T, MSE_error)
    thd.scatter(T, MSE_error, marker='o', c='r', s=50)
    thd.legend(['Значение ошибки в разные моменты времени'], fontsize=7,
loc='upper right')


time_p = 0.001
h_steps = 0.001
l = 3000

cur_display_mode = int(input("Выберите схему (1 - Явная схема, 2 - Не явная
схема): "))
start_aprox = int(input("Введите порядок аппроксимации для начальных условий
(1, 2): "))

analitic = compute_analytical_on_grid(time_p, l, h_steps)
fig = plt.figure(figsize=(15, 12))
fst = fig.add_subplot(2, 3, 1)
scd = fig.add_subplot(2, 3, 2)
thd = fig.add_subplot(2, 3, 3)

fst.set_title('Аналитическое решение. Результаты сетке ', fontsize=10)
fst.set_xlabel('x', fontsize=9)
fst.set_ylabel('U(x, t)', fontsize=9)

display_modes = [1, 2]
display_index = 0

update_plot(analitic, time_p, l, start_aprox, h_steps, cur_display_mode)
plt.show()
```

Лабораторная работа №7

1. 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 U(x,y). Исследовать зависимость погрешности от сеточных параметров *h_x*, *h_y*.

Вариант:

6.

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

$$u(0, y) = 0,$$

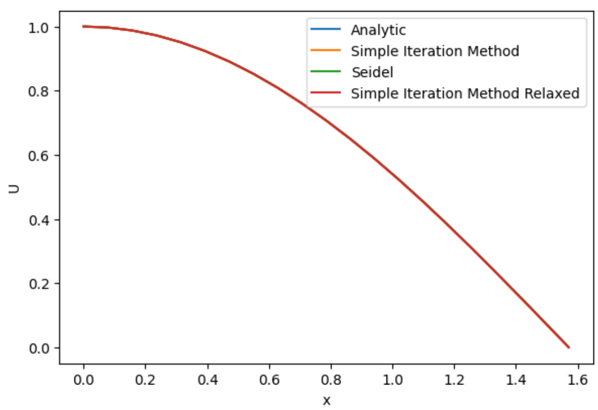
$$u(\frac{\pi}{2}, y) = y,$$

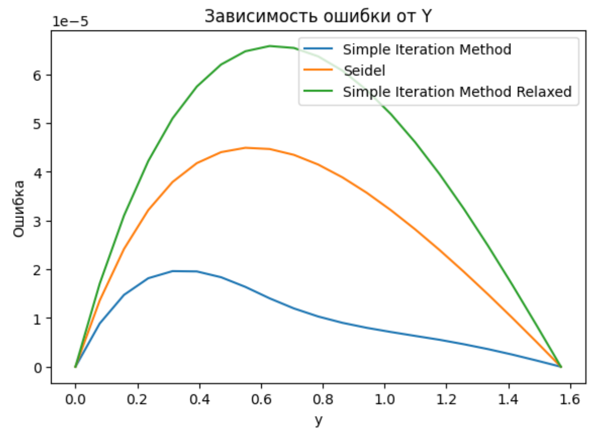
$$u_y(x,0) = \sin x,$$

$$u_y(x,1) - u(x,1) = 0.$$

Аналитическое решение: $U(x, y) = y \sin x$.

2. Результат





3. Вывод

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

4. Код

```
5. import numpy as np
   import copy
```

```python
args = {
    'a': 0,
    'b': 0,
    'c': 2,
    'd': 1,
    'lx': np.pi / 2,
    'ly': np.pi / 2,
    'w': 1.5,
    'Function': lambda x, y: 0,
    'alpha1': 0,
    'alpha2': 1,
    'beta1': 0,
    'beta2': 1,
    'gamma1': 0,
    'gamma2': 1,
    'delta1': 0,
    'delta2': 1,
    'Phi1': lambda y: np.cos(y),
    'Phi2': lambda y: 0,
    'Phi3': lambda x: np.cos(x),
    'Phi4': lambda x: 0,
    'AnaliticalSolution': lambda x, y: np.cos(x) * np.cos(y),
    'algorithm': 'SimpleIterationMethodRelaxed',
    'nx': 10,
    'ny': 10,
    'maxIterNum': 10000,
    'eps': 1e-6
}

class Eleptical:
    def __init__(self, args):
        for name, value in args.items():
            setattr(self, name, value)
        functionName = args['algorithm']
        self.hx = 0
        self.hy = 0
        self.iteration = 0
        try:
            self.FunctionName = getattr(self, functionName)
        except:
            raise Exception("Данный тип не поддерживается, выберите Simple
Iteration Method, Seidel или Simple Iteration Method Relaxed")

    def Accurancy(self):
        res = 0
        for i in range(self.nx):
            for j in range(self.ny):
                res = max(res, abs(self.u[i][j] - self.L[i][j]))
        return res

    def Prepare(self):
        self.u = np.zeros((len(self.x), len(self.y)))
        for i in range(len(self.y)):
            self.u[0][i] = self.Phi1(self.y[i]) / self.alpha2
            self.u[-1][i] = self.Phi2(self.y[i]) / self.beta2
        for i in range(len(self.x)):
            self.u[i][0] = self.Phi3(self.x[i]) / self.gamma2
            self.u[i][-1] = self.Phi4(self.x[i]) / self.delta2

    def Solve(self):
        self.hx = self.lx / self.nx
        self.hy = self.ly / self.ny
        self.x = np.arange(0, self.lx + self.hx, self.hx)
        self.y = np.arange(0, self.ly + self.hy, self.hy)
```

```
        self.UPrepare()
        for i in range(1, self.nx):
            for j in range(1, self.ny):
                self.u[i][j] = self.u[0][j] + (self.x[i] - self.x[0]) *
(self.u[-1][j] - self.u[0][j]) / (self.x[-1] - self.x[0])
        return self.FunctionName()

    def AnaliticalSolutionMatrix(self):
        self.hx = self.lx / self.nx
        self.hy = self.ly / self.ny
        self.x = np.arange(0, self.lx + self.hx, self.hx)
        self.y = np.arange(0, self.ly + self.hy, self.hy)
        self.u = []
        for yi in self.y:
            self.u.append([self.AnaliticalSolution(xi, yi) for xi in self.x])
        return self.u

    def SimpleIterationMethod(self):
        currentAccurancy = 1e9
        while self.iteration < self.maxIterNum:
            self.L = copy.deepcopy(self.u)
            self.UPrepare()
            for j in range(1, len(self.y) - 1):
                for i in range(1, len(self.x) - 1):
                    self.u[i][j] = (self.hx * self.hx *
self.Function(self.x[i], self.y[j]) - (self.L[i + 1][j] + self.L[i - 1][j]) -
self.d * self.hx * self.hx * (self.L[i][j + 1] + self.L[i][j - 1]) / (self.hy
* self.hy) - self.a * self.hx * 0.5 * (self.L[i + 1][j] - self.L[i - 1][j]) -
self.b * self.hx * self.hx * (self.L[i][j + 1] - self.L[i][j - 1]) / (2 *
self.hy)) / (self.c * self.hx * self.hx - 2 * (self.hy * self.hy + self.d *
self.hx * self.hx) / (self.hy * self.hy))
            previousAccurancy = currentAccurancy
            currentAccurancy = self.GetCurrentAccurancy()
            if self.GetCurrentAccurancy() <= self.eps:# or previousAccurancy
< currentAccurancy:
                break
            self.iteration += 1
        return self.u, self.iteration

    def Seidel(self):
        currentAccurancy = 1e9
        while self.iteration < self.maxIterNum:
            self.L = copy.deepcopy(self.u)
            self.UPrepare()
            for j in range(1, len(self.y) - 1):
                for i in range(1, len(self.x) - 1):
                    self.u[i][j] = ((self.hx ** 2) * self.Function(self.x[i],
self.y[j]) - (self.L[i + 1][j] + self.u[i - 1][j]) - self.d * (self.hx ** 2)
* (self.L[i][j + 1] + self.u[i][j - 1]) / (self.hy ** 2) - self.a * self.hx *
0.5 * (self.L[i + 1][j] - self.u[i - 1][j]) - self.b * (self.hx ** 2) *
(self.L[i][j + 1] - self.u[i][j - 1]) / (2 * self.hy)) / (self.c * (self.hx
** 2) - 2 * (self.hy ** 2 + self.d * (self.hx ** 2)) / (self.hy ** 2))
            previousAccurancy = currentAccurancy
            currentAccurancy = self.GetCurrentAccurancy()
            if currentAccurancy <= self.eps:# or previousAccurancy <
currentAccurancy:
                break
            self.iteration += 1
        return self.u, self.iteration

    def SimpleIterationMethodRelaxed(self):
        currentAccurancy = 1e9
        while self.iteration < self.maxIterNum:
            self.L = copy.deepcopy(self.u)
```

```
                self.UPrepare()
                for j in range(1, len(self.y) - 1):
                    for i in range(1, len(self.x) - 1):
                        self.u[i][j] = (((self.hx ** 2) *
self.Function(self.x[i], self.y[j]) - (self.L[i + 1][j] + self.u[i - 1][j]) -
self.d * (self.hx ** 2) * (self.L[i][j + 1] + self.u[i][j - 1]) / (self.hy **
2) - self.a * self.hx * 0.5 * (self.L[i + 1][j] - self.u[i - 1][j]) - self.b
* (self.hx ** 2) * (self.L[i][j + 1] - self.u[i][j - 1]) / (2 * self.hy)) /
(self.c * (self.hx ** 2) - 2 * (self.hy ** 2 + self.d * (self.hx ** 2)) /
(self.hy ** 2))) * self.w + (1 - self.w) * self.L[i][j]
                previousAccurancy = currentAccurancy
                currentAccurancy = self.GetCurrentAccurancy()
                if self.GetCurrentAccurancy() <= self.eps:# or previousAccurancy
< currentAccurancy:
                    break
                self.iteration += 1
            return self.u, self.iteration

algorithms = ('SimpleIterationMethod', 'Seidel',
'SimpleIterationMethodRelaxed')
args['nx'] = 20
args['ny'] = 20

answers = dict()
solver = ElepticalSolver(args)
analytic = solver.AnaliticalSolutionMatrix()
answers['Analytic'] = analytic
for algorithm in algorithms:
    args['algorithm'] = algorithm
    solver = ElepticalSolver(args)
    numeric = solver.Solve()
    answers[algorithm] = numeric

for algorithm in algorithms:
    print(f'{algorithm}: {answers[algorithm][1]} итераций.')

import matplotlib.pyplot as plt

def GetError(numeric, analytic):
    err = []
    error = [[abs(i - j) for i, j in zip(x, y)] for x, y in zip(numeric,
analytic)]
    for i in range(len(error)):
        tmp = 0
        for j in error[i]:
            tmp += j
        err.append(tmp / len(error[i]))
    return err

def Draw2DCharts(answers, args, time=0):
```

Лабораторная работа №8

1. 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 U(x, t). Исследовать зависимость погрешности от сеточных параметров τ, h_x, h_y.

Вариант:

6.

$$\frac{\partial u}{\partial t} = a\frac{\partial^2 u}{\partial x^2} + a\frac{\partial^2 u}{\partial y^2},\ a > 0,$$

$$u(0, y, t) = \sinh(y)\exp(-3at),$$

$$u_x(\frac{\pi}{4}, y, t) = -2\sinh(y)\exp(-3at),$$

$$u_y(x, 0, t) = \cos(2x)\exp(-3at),$$

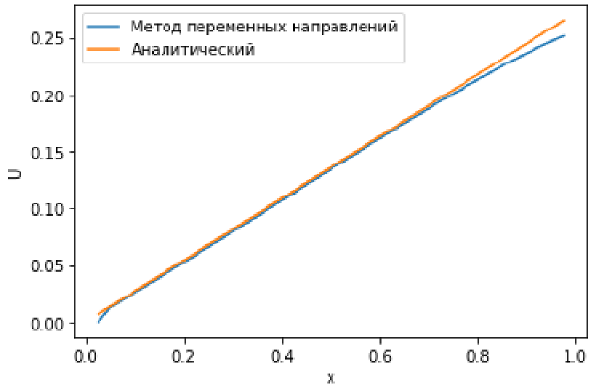
$$u(x, \ln 2, t) = \frac{3}{4}\cos(2x)\exp(-3at),$$

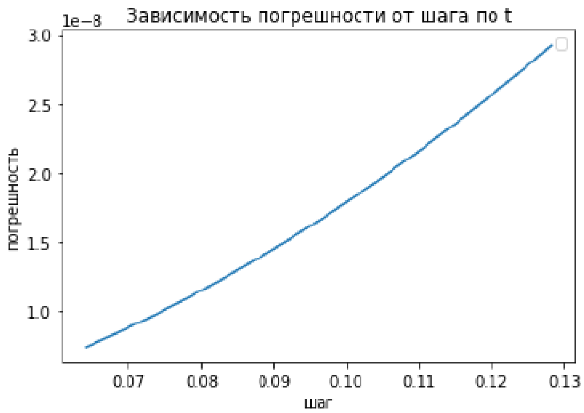
$$u(x, y, 0) = \cos(2x)\sinh(y).$$

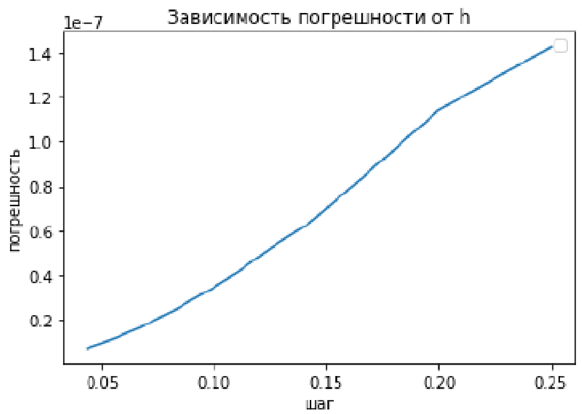
Аналитическое решение: $U(x, y, t) = \cos(2x)\sinh(y)\exp(-3at)$.

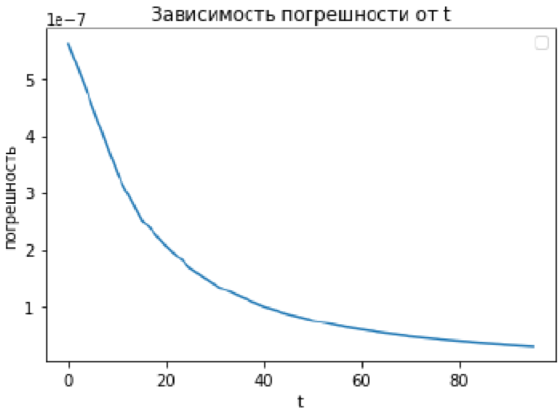
2. Результат

1) Метод переменных направлений

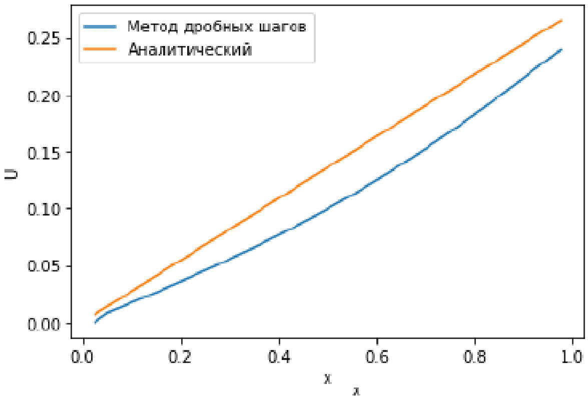


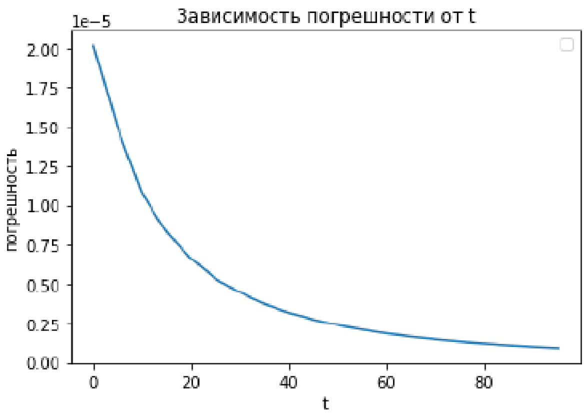


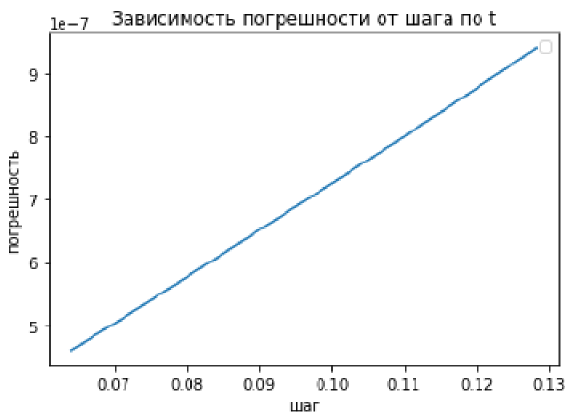


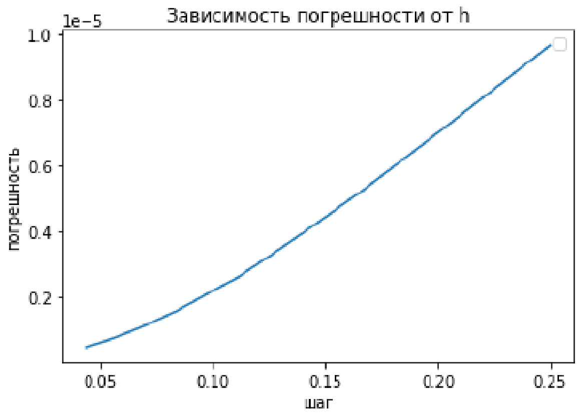


2) Метод дробных шагов









3. Вывод

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

4. Код

```
5. import ipywidgets as widgets
   from ipywidgets import interact
   from IPython.display import display
   from ipywidgets import interact, interactive, fixed, interact_manual
   from tqdm import tqdm
   import random
   import matplotlib.pyplot as plt
   from matplotlib import cm
   import math
   import sys
   import warnings
   import numpy as np
   import glob
   from functools import reduce
   from mpl_toolkits.mplot3d import Axes3D
   import plotly.offline as offline
   from plotly.graph_objs import *

   def RunThrough(a, b, c, d):
       size = len(a)
       p, q = [], []
       p.append(-c[0] / b[0])
       q.append(d[0] / b[0])

       for i in range(1, size):
           pTmp = -c[i] / (b[i] + a[i] * p[i - 1])
           qTmp = (d[i] - a[i] * q[i - 1]) / (b[i] + a[i] * p[i - 1])
           p.append(pTmp)
           q.append(qTmp)

       x = [0 for _ in range(size)]
       x[size - 1] = q[size - 1]

       for i in range(size - 2, -1, -1):
           x[i] = p[i] * x[i + 1] + q[i]

       return x
```

```python
class ParabolicSolver:
    def __init__(self, args):
        for name, value in args.items():
            setattr(self, name, value)
        self.hx = 0
        self.hy = 0
        self.tau = 0
        try:
            self.solve_func = getattr(self, args['algorithm'])
        except:
            raise Exception("This type does not exist")

    def Solve(self, nx, ny, T, K):
        self.hx = self.dx / nx
        self.hy = self.dy / ny
        self.tau = T / K

        x = np.arange(0, self.dx + self.hx, self.hx)
        y = np.arange(0, self.dy + self.hy, self.hy)
        t = np.arange(0, T + self.tau, self.tau)

        uu = np.zeros((len(x), len(y), len(t)))
        for i in range(len(x)):
            for j in range(len(y)):
                uu[i][j][0] = self.Psi(x[i], y[j])

        return self.solve_func(x, y, t, uu)

    def AnalyticSolve(self, nx, ny, T, K):
        self.hx = self.dx / nx
        self.hy = self.dy / ny
        self.tau = T / K
        x = np.arange(0, self.dx + self.hx, self.hx)
        y = np.arange(0, self.dy + self.hy, self.hy)
        t = np.arange(0, T + self.tau, self.tau)

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

        for i in range(len(x)):
            for j in range(len(y)):
                for k in range(len(t)):
                    uu[i][j][k] = self.AnaliticalSolution(x[i], y[j], t[k])

        return uu

    def VariableDirections(self, x, y, t, uu):
        for k in range(1, len(t)):
            u1 = np.zeros((len(x), len(y)))
            t2 = t[k] - self.tau / 2
            for j in range(len(y) - 1):
                aa = np.zeros(len(x))
                bb = np.zeros(len(x))
                cc = np.zeros(len(x))
                dd = np.zeros(len(x))

                bb[0] = self.hx * self.alpha2 - self.alpha1
                bb[-1] = self.hx * self.beta2 + self.beta1
                cc[0] = self.alpha1
                aa[-1] = -self.beta1
                dd[0] = self.Phi11(y[j], t2) * self.hx
                dd[-1] = self.Phi12(y[j], t2) * self.hx
                for i in range(len(x) - 1):
                    aa[i] = self.a - self.hx * self.c / 2
                    bb[i] = self.hx ** 2 - 2 * (self.hx ** 2) / self.tau - 2 *
```

```
self.a
                cc[i] = self.a + self.hx * self.c / 2
                dd[i] = -2 * (self.hx ** 2) * uu[i][j][k - 1] / self.tau
                - self.b * (self.hx ** 2) * (uu[i][j + 1][k - 1]
                                          - 2 * uu[i][j][k - 1] + uu[i][j -
1][k - 1]) / (self.hy ** 2)
                - self.d * (self.hx ** 2) * (uu[i][j + 1][k - 1] - uu[i][j -
1][k - 1]) / (2 * self.hy ** 2)
                - (self.hx ** 2) * self.Function(x[i], y[j], t[k])

             xx = RunThroughMethod(aa, bb, cc, dd)
             for i in range(len(x)):
                u1[i][j] = xx[i]
                u1[i][0] = (self.Phi21(x[i], t2) - self.gamma1 * u1[i][1] /
self.hy) / (
                      self.gamma2 - self.gamma1 / self.hy)
                u1[i][-1] = (self.Phi22(x[i], t2) + self.delta1 * u1[i][-2] /
self.hy) / (
                      self.delta2 + self.delta1 / self.hy)
          for j in range(len(y)):
             u1[0][j] = (self.Phi11(y[j], t2) - self.alpha1 * u1[1][j] /
self.hx) / (
                      self.alpha2 - self.alpha1 / self.hx)
             u1[-1][j] = (self.Phi12(y[j], t2) + self.beta1 * u1[-2][j] /
self.hx) / (
                      self.beta2 + self.beta1 / self.hx)
          ####
          u2 = np.zeros((len(x), len(y)))
          for i in range(len(x) - 1):
             aa = np.zeros(len(x))
             bb = np.zeros(len(x))
             cc = np.zeros(len(x))
             dd = np.zeros(len(x))

             bb[0] = self.hy * self.gamma2 - self.gamma1
             bb[-1] = self.hy * self.delta2 + self.delta1
             cc[0] = self.gamma1
             aa[-1] = -self.delta1
             dd[0] = self.Phi21(x[i], t[k]) * self.hy
             dd[-1] = self.Phi22(x[i], t[k]) * self.hy

             for j in range(len(y) - 1):
                aa[j] = self.b - self.hy * self.d / 2
                bb[j] = self.hy ** 2 - 2 * (self.hy ** 2) / self.tau - 2 *
self.b
                cc[j] = self.b + self.hy * self.d / 2
                dd[j] = -2 * (self.hy ** 2) * u1[i][j] / self.tau
                - self.a * (self.hy ** 2) * (u1[i + 1][j]
                                          - 2 * u1[i][j] + u1[i - 1][j]) /
(self.hx ** 2)
                - self.c * (self.hy ** 2) * (u1[i + 1][j] - u1[i - 1][j]) /
(2 * self.hx ** 2)
                - (self.hy ** 2) * self.Function(x[i], y[j], t[k])
             xx = RunThroughMethod(aa, bb, cc, dd)
             for j in range(len(y)):
                u2[i][j] = xx[j]
                u2[0][j] = (self.Phi11(y[j], t[k]) - self.alpha1 * u2[1][j] /
self.hx) / (
                         self.alpha2 - self.alpha1 / self.hx)
                u2[-1][j] = (self.Phi12(y[j], t[k]) + self.beta1 * u2[-2][j]
/ self.hx) / (
                         self.beta2 + self.beta1 / self.hx)
          for i in range(len(x)):
             u2[i][0] = (self.Phi21(x[i], t[k]) - self.gamma1 * u2[i][1] /
```

```
self.hy) / (
                      self.gamma2 - self.gamma1 / self.hy)
             u2[i][-1] = (self.Phi22(x[i], t[k]) + self.delta1 * u2[i][-2] /
self.hy) / (
                      self.delta2 + self.delta1 / self.hy)
          for i in range(len(x)):
             for j in range(len(y)):
                uu[i][j][k] = u2[i][j]
       return uu

    def FractionalSteps(self, x, y, t, uu):
       for k in range(len(t)):
          u1 = np.zeros((len(x), len(y)))
          t2 = t[k] - self.tau / 2
          for j in range(len(y) - 1):
             aa = np.zeros(len(x))
             bb = np.zeros(len(x))
             cc = np.zeros(len(x))
             dd = np.zeros(len(x))

             bb[0] = self.hx * self.alpha2 - self.alpha1
             bb[-1] = self.hx * self.beta2 + self.beta1
             cc[0] = self.alpha1
             aa[-1] = -self.beta1
             dd[0] = self.Phi11(y[j], t2) * self.hx
             dd[-1] = self.Phi12(y[j], t2) * self.hx
             for i in range(len(x) - 1):
                aa[i] = self.a
                bb[i] = -(self.hx ** 2) / self.tau - 2 * self.a
                cc[i] = self.a
                dd[i] = -(self.hx ** 2) * uu[i][j][k - 1] / self.tau -
(self.hx ** 2) * self.Function(x[i], y[j],

t2) / 2
             xx = RunThroughMethod(aa, bb, cc, dd)
             for i in range(len(x)):
                u1[i][j] = xx[i]
                u1[i][0] = (self.Phi21(x[i], t2) - self.gamma1 * u1[i][1] /
self.hy) / (
                      self.gamma2 - self.gamma1 / self.hy)
                u1[i][-1] = (self.Phi22(x[i], t2) + self.delta1 * u1[i][-2] /
self.hy) / (
                      self.delta2 + self.delta1 / self.hy)
          for j in range(len(y)):
             u1[0][j] = (self.Phi11(y[j], t2) - self.alpha1 * u1[1][j] /
self.hx) / (
                   self.alpha2 - self.alpha1 / self.hx)
             u1[-1][j] = (self.Phi12(y[j], t2) + self.beta1 * u1[-2][j] /
self.hx) / (
                   self.beta2 + self.beta1 / self.hx)
          #####
          u2 = np.zeros((len(x), len(y)))
          for i in range(len(x) - 1):
             aa = np.zeros(len(x))
             bb = np.zeros(len(x))
             cc = np.zeros(len(x))
             dd = np.zeros(len(x))

             bb[0] = self.hy * self.gamma2 - self.gamma1
             bb[-1] = self.hy * self.delta2 + self.delta1
             cc[0] = self.gamma1
             aa[-1] = -self.delta1
             dd[0] = self.Phi21(x[i], t[k]) * self.hy
             dd[-1] = self.Phi22(x[i], t[k]) * self.hy
```

```
                for j in range(len(y) - 1):
                    aa[j] = self.b
                    bb[j] = -(self.hy ** 2) / self.tau - 2 * self.b
                    cc[j] = self.b
                    dd[j] = -(self.hy ** 2) * u1[i][j] / self.tau - (self.hy **
2) * self.Function(x[i], y[j], t[k]) / 2
                xx = RunThroughMethod(aa, bb, cc, dd)
                for j in range(len(y)):
                    u2[i][j] = xx[j]
                    u2[0][j] = (self.Phi11(y[j], t[k]) - self.alpha1 * u2[1][j] /
self.hx) / (
                            self.alpha2 - self.alpha1 / self.hx)
                    u2[-1][j] = (self.Phi12(y[j], t[k]) + self.beta1 * u2[-2][j]
/ self.hx) / (
                            self.beta2 + self.beta1 / self.hx)
            for i in range(len(x)):
                u2[i][0] = (self.Phi21(x[i], t[k]) - self.gamma1 * u2[i][1] /
self.hy) / (
                        self.gamma2 - self.gamma1 / self.hy)
                u2[i][-1] = (self.Phi22(x[i], t[k]) + self.delta1 * u2[i][-2] /
self.hy) / (
                        self.delta2 + self.delta1 / self.hy)
            for i in range(len(x)):
                for j in range(len(y)):
                    uu[i][j][k] = u2[i][j]
        return uu

def Approximation(x_list, y, t, numericalAnswer):
    res = []
    for i in range(len(x_list)):
        res.append(numericalAnswer[0][i][y][t])
    return res

def PrepareNumerical(x, y, time, numericalAnswer):
    return [Approximation(x, y, time, numericalAnswer) for _ in
range(len(x))]

def PrepareAnalitic(x_list, y, t, analyticAnswer):
    res = []
    for xi in x_list:
        res.append(xi * y * np.cos(t))
    return res

def psi_0(x, t):
    return 0.0

def psi_1(x, t):
    return x * math.cos(t)

def phi_0(y, t):
    return 0.0

def phi_1(y, t):
    return y * math.cos(t)

def u0(x, y):
    return x*y

def u(x, y, t):
    return x*y * math.cos(t)

class Schema:
    def __init__(self, rho=u0, psi0=psi_0, psi1=psi_1, phi0=phi_0,
```

```python
    phi1=phi_1,
                  lx0=0, lx1=1.0, ly0=0, ly1=1.0, T=3, order2nd=True):
        self.psi0 = psi0
        self.psi1 = psi1
        self.phi0 = phi0
        self.phi1 = phi1
        self.rho0 = rho
        self.T = T
        self.lx0 = lx0
        self.lx1 = lx1
        self.ly0 = ly0
        self.ly1 = ly1
        self.tau = None
        self.hx = None
        self.hy = None
        self.order = order2nd
        self.Nx = None
        self.Ny = None
        self.K = None
        self.cx = None
        self.bx = None
        self.cy = None
        self.by = None
        self.hx2 = None
        self.hy2 = None

    def set_l0_l1(self, lx0, lx1, ly0, ly1):
        self.lx0 = lx0
        self.lx1 = lx1
        self.ly0 = ly0
        self.ly1 = ly1

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

    def CalculateH(self):
        self.hx = (self.lx1 - self.lx0) / (self.Nx - 1)
        self.hy = (self.ly1 - self.ly0) / (self.Ny - 1)
        self.hx2 = self.hx * self.hx
        self.hy2 = self.hy * self.hy

    def CalculateTau(self):
        self.tau = self.T / (self.K - 1)

    @staticmethod
    def race_method(A, b):
        P = [-item[2] for item in A]
        Q = [item for item in b]
        P[0] /= A[0][1]
        Q[0] /= A[0][1]
        for i in range(1, len(b)):
            z = (A[i][1] + A[i][0] * P[i - 1])
            P[i] /= z
            Q[i] -= A[i][0] * Q[i - 1]
            Q[i] /= z
        for i in range(len(Q) - 2, -1, -1):
            Q[i] += P[i] * Q[i + 1]
        return Q

    @staticmethod
    def nparange(start, end, step=1):
        now = start
        e = 0.00000000001
        while now - e <= end:
```

```
            yield now
            now += step

    def CalculateLeftEdge(self, X, Y, t, square):
        for i in range(self.Ny):
            square[i][0] = self.phi0(Y[i][0], t)

    def CalculateRightEdge(self, X, Y, t, square):
        for i in range(self.Ny):
            square[i][-1] = self.phi1(Y[i][-1], t)

    def CalculateBottomEdge(self, X, Y, t, square):
        for j in range(1, self.Nx - 1):
            square[0][j] = self.psi0(X[0][j], t)

    def CalculateTopEdge(self, X, Y, t, square):
        for j in range(1, self.Nx - 1):
            square[-1][j] = self.psi1(X[-1][j], t)

    def CalculateLineFirstStep(self, i, X, Y, t, last_square, now_square):
        hy2 = self.hy2
        hx2 = self.hx2
        b = self.bx
        c = self.cx
        A = [(0, b, c)]
        w = [
            -self.cy * self.order * last_square[i - 1][1] -
            ((self.order + 1) * hx2 * hy2 - 2 * self.cy * self.order) *
last_square[i][1] -
            self.cy * self.order * last_square[i + 1][1] +
            self.tau * hy2 * hx2 * X[i][1] * Y[i][1] * math.sin(t) -
            c * now_square[i][0]
        ]
        A.extend([(c, b, c) for _ in range(2, self.Nx - 2)])
        w.extend([
            -self.cy * self.order * last_square[i - 1][j] -
            ((self.order + 1) * hx2 * hy2 - 2 * self.cy * self.order) *
last_square[i][j] -
            self.cy * self.order * last_square[i + 1][j] +
            self.tau * hy2 * hx2 * X[i][j] * Y[i][j] * math.sin(t)
            for j in range(2, self.Nx - 2)
        ])
        A.append((c, b, 0))
        w.append(
            -self.cy * self.order * last_square[i - 1][-2] -
            ((self.order + 1) * hx2 * hy2 - 2 * self.cy * self.order) *
last_square[i][-2] -
            self.cy * self.order * last_square[i + 1][-2] +
            self.tau * hy2 * hx2 * X[i][-2] * Y[i][-2] * math.sin(t) -
            c * now_square[i][-1]
        )
        line = self.race_method(A, w)
        for j in range(1, self.Nx - 1):
            now_square[i][j] = line[j - 1]

    def CalculateLineSecondStep(self, j, X, Y, t, last_square, now_square):
        hx2 = self.hx2
        hy2 = self.hy2
        c = self.cy
        b = self.by
        A = [(0, b, c)]
        w = [
            -self.cx * self.order * last_square[1][j - 1] -
            ((self.order + 1) * hx2 * hy2 - 2 * self.cx * self.order) *
```

```
last_square[1][j] -
            self.cx * self.order * last_square[1][j + 1] +
            self.tau * hy2 * hx2 * X[1][j] * Y[1][j] * math.sin(t) -
            c * now_square[0][j]
        ]
        A.extend([(c, b, c) for _ in range(2, self.Ny - 2)])
        w.extend([
            -self.cx * self.order * last_square[i][j - 1] -
            ((self.order + 1) * hx2 * hy2 - 2 * self.cx * self.order) *
last_square[i][j] -
            self.cx * self.order * last_square[i][j + 1] +
            self.tau * hy2 * hx2 * X[i][j] * Y[i][j] * math.sin(t)
            for i in range(2, self.Ny - 2)
        ])
        A.append((c, b, 0))
        w.append(
            -self.cx * self.order * last_square[-2][j - 1] -
            ((self.order + 1) * hx2 * hy2 - 2 * self.cx * self.order) *
last_square[-2][j] -
            self.cx * self.order * last_square[-2][j + 1] +
            self.tau * hy2 * hx2 * X[-2][j] * Y[-2][j] * math.sin(t) -
            c * now_square[-1][j]
        )
        line = self.race_method(A, w)
        for i in range(1, self.Ny - 1):
            now_square[i][j] = line[i - 1]

    def CalculateSquare(self, X, Y, t, last_square):
        square = [[0.0 for _ in range(self.Nx)] for _ in range(self.Ny)]
        self.CalculateLeftEdge(X, Y, t - 0.5 * self.tau, square)
        self.CalculateRightEdge(X, Y, t - 0.5 * self.tau, square)
        self.CalculateBottomEdge(X, Y, t - 0.5 * self.tau, square)
        self.CalculateTopEdge(X, Y, t - 0.5 * self.tau, square)
        for i in range(1, self.Ny - 1):
            self.CalculateLineFirstStep(i, X, Y, t - 0.5 * self.tau,
last_square, square)
        last_square = square
        square = [[0.0 for _ in range(self.Nx)] for _ in range(self.Ny)]
        self.CalculateLeftEdge(X, Y, t, square)
        self.CalculateRightEdge(X, Y, t, square)
        self.CalculateBottomEdge(X, Y, t, square)
        self.CalculateTopEdge(X, Y, t, square)
        for j in range(1, self.Nx - 1):
            self.CalculateLineSecondStep(j, X, Y, t, last_square, square)
        return square

    def init_t0(self, X, Y):
        first = [[0.0 for _ in range(self.Nx)] for _ in range(self.Ny)]
        for i in range(self.Ny):
            for j in range(self.Nx):
                first[i][j] = self.rho0(X[i][j], Y[i][j])
        return first

    def __call__(self, Nx=20, Ny=20, K=20):
        self.Nx, self.Ny, self.K = Nx, Ny, K
        self.CalculateTau()
        self.CalculateH()

        self.bx = -2 * self.tau * self.hy2
        self.bx -= (1 + self.order) * self.hx2 * self.hy2
        self.cx = self.tau * self.hy2

        self.cy = self.tau * self.hx2
        self.by = -2 * self.tau * self.hx2
```

```
        self.by -= (1 + self.order) * self.hx2 * self.hy2
        x = list(self.nparange(self.lx0, self.lx1, self.hx))
        y = list(self.nparange(self.ly0, self.ly1, self.hy))
        X = [x for _ in range(self.Ny)]
        Y = [[y[i] for _ in x] for i in range(self.Ny)]

        taus = [0.0]
        ans = [self.init_t0(X, Y)]
        for t in self.nparange(self.tau, self.T, self.tau):
           ans.append(self.CalculateSquare(X, Y, t, ans[-1]))
           taus.append(t)
        return X, Y, taus, ans


def RealZByTime(lx0, lx1, ly0, ly1, t, f):
    x = np.arange(lx0, lx1 + 0.002, 0.002)
    y = np.arange(ly0, ly1 + 0.002, 0.002)
    X = np.ones((y.shape[0], x.shape[0]))
    Y = np.ones((x.shape[0], y.shape[0]))
    Z = np.ones((y.shape[0], x.shape[0]))
    for i in range(Y.shape[0]):
        Y[i] = y
    Y = Y.T
    for i in range(X.shape[0]):
        X[i] = x
    for i in range(Z.shape[0]):
        for j in range(Z.shape[1]):
            Z[i, j] = f(X[i, j], Y[i, j], t)
    return X, Y, Z

def Error(X, Y, t, z, ut = u):
    ans = 0.0
    for i in range(len(z)):
        for j in  range(len(z[i])):
            ans = max(abs(ut(X[i][j], Y[i][j], t) - z[i][j]), ans)
    return (ans / len(z) / len(z[0]))

def StepSlice(lst, step):
    return lst[step]

def AnimateList(lst, play=False, interval=200):
    slider = widgets.IntSlider(min=0, max=len(lst) - 1, step=1, value=0)
    if play:
        play_widjet = widgets.Play(interval=interval)
        widgets.jslink((play_widjet, 'value'), (slider, 'value'))
        display(play_widjet)
    return interact(StepSlice,
                    lst=fixed(lst),
                    step=slider)


def PlotByTime(X, Y, T, Z, j, extrems, plot_true=True):
    t = T[j]
    z = Z[j]
    fig = plt.figure(num=1, figsize=(20, 13), clear=True)
    ax = fig.add_subplot(1, 1, 1, projection='3d')
    ax.plot_surface(np.array(X), np.array(Y), np.array(z))
    if plot_true:
       ax.plot_wireframe(*RealZByTime(0, 1, 0, 1, t, u), color="green")
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    ax.set_title(
       't = ' + str(round(t, 8)) + " error = " + str(round(Error(X, Y, t, z),
```

```
11)),
        loc="right", fontsize=25
    )
    ax.set_zlim(extrems[0], extrems[1])
    fig.tight_layout()
    plt.close(fig)
    return fig


def SquareMinMax(z):
    minimum, maximum = z[0][0], z[0][0]
    for i in range(len(z)):
       for j in range(len(z[i])):
          minimum = z[i][j] if z[i][j] < minimum else minimum
          maximum = z[i][j] if z[i][j] > maximum else maximum
    return minimum, maximum


def SearchMinMax(zz):
    minimum, maximum = 0.0, 0.0
    for z in zz:
       minmax = SquareMinMax(z)
       minimum = minmax[0] if minmax[0] < minimum else minimum
       maximum = minmax[1] if minmax[1] > maximum else maximum
    return minimum, maximum


def PlotAnimate(nx=15, ny=15, k=50, t=5, plot_true=False):
    schema = Schema(T=t, order2nd=True)
    xx, yy, tt, zz = schema(Nx=nx, Ny=ny, K=k)
    extrems = SearchMinMax(zz)
    plots = []
    for j in range(len(tt)):
       plots.append(PlotByTime(xx, yy, tt, zz, j, extrems, plot_true))
    AnimateList(plots, play=True, interval=2000)

first = Schema(T = 2*math.pi, order2nd = False) #метод дробных шагов
second = Schema(T = 2*math.pi, order2nd = True) #метод переменных направлений

def GetGraphicH(solver, time = 0, tsteps = 40):
    h, e = [], []
    for N in range(4, 20, 1):
        x, y, t, z = solver(Nx = N, Ny = N, K = tsteps)
        h.append(solver.hx)
        e.append(Error(x, y, t[time], z[time]))
    return h, e

TSTEPS = 100
time = random.randint(0, TSTEPS - 1)
h1, e1 = GetGraphicH(first, time, TSTEPS)
h2, e2 = GetGraphicH(second, time, TSTEPS)

def GetGraphicTau(solver):
    tau = []
    e = []
    for K in range(15, 100, 2):
        x, y, t, z = solver(Nx = 10, Ny = 10, K = K)
        tau.append(solver.tau)
        time = K // 2
        e.append(Error(x, y, t[time], z[time]))
    return tau, e

tau1, e1 = GetGraphicTau(first)
tau2, e2 = GetGraphicTau(second)
```

```python
def FullError(X, Y, T, Z):
    ans = 0.0
    for k in range(len(T)):
        for i in range(len(X)):
            for j in range(len(X[i])):
                ans = max(abs(u(X[i][j], Y[i][j], T[k]) - Z[k][i][j]), ans)
    return (ans / len(T) / len(X) / len(X[0]))

TimeList = [random.randint(0, 40 - 1) for i in range(4)]
def plotDependenceError(Nx=4, Ny=4, K=40, Method=0):
    NxFix = Nx
    NyFix = Ny
    if Method != 0:
        method = True
        stri = 'Погрешность метода переменных направлений'
    else:
        method = False
        stri = 'Погрешность метода дробных шагов'
    plt.figure(figsize=(6, 12))
    for i in range(1, 4):
        plt.subplot(5, 1, i)
        Time = TimeList[i-1]#random.randint(0, K - 1)
        schema = Schema(T=Time, order2nd=method)
        h, eps = [], []
        for j in range(10):
            h.append([])
            eps.append([])
            X, Y, T, Z = schema(Nx, Ny, K)
            Nx += 1
            Ny += 1
            h[-1].append(schema.hx)
            eps[-1].append(Error(X, Y, T[Time], Z[Time]))
        Nx = NxFix
        Ny = NyFix
        plt.plot(np.array(h), np.array(eps), label="const T = " +
str(round(T[Time])))
        plt.xlabel('$h$')
        plt.ylabel('$\epsilon$')
        plt.title(stri)
        plt.legend()
        plt.grid()
        plt.show()

firstMethod = True
nx, ny, T, K = 40, 40, 5, 100
plottingTime = 2

args = {
    'a': 1,
    'b': 1,
    'c': 0,
    'd': 0,
    'dx': 1,
    'dy': 1,
    'Function': lambda x, y, t: - x * y * np.sin(t),
    'alpha1': 0,
    'alpha2': 1,
    'beta1': 0,
    'beta2': 1,
    'gamma1': 0,
    'gamma2': 1,
    'delta1': 0,
    'delta2': 1,
```

```
        'Phi11': lambda y, t: 0,
        'Phi12': lambda y, t: y * np.cos(t),
        'Phi21': lambda x, t: 0,
        'Phi22': lambda x, t: x * np.cos(t),
        'Psi': lambda x, y: x * y,
        'AnaliticalSolution': lambda x, y, t: x * y * np.cos(t),
        'algorithm' : 'VariableDirections' if firstMethod else 'FractionalSteps'
    }
    solver = ParabolicSolver(args)
    numericalAnswer = solver.Solve(nx, ny, T, K),
    analyticAnswer = solver.AnalyticSolve(nx, ny, T, K)

    def DrawCharts(numericalAnswer, analyticAnswer, nx, ny, T, K, time, check =
    0):
        fig = plt.figure()
        hx = 1 / nx
        hy = 1 / ny
        tau = T / K

        x = np.arange(hx, 1, hx)
        y = np.arange(hy, 1, hy)
        t = np.arange(tau, T, tau)


        z1 = PrepareNumerical(x, 10, time, numericalAnswer)
        z2 = PrepareAnalitic(x, y[10], t[time], analyticAnswer)

        # plt.title('U от x')
        if check == 0:
          plt.plot(x, z1[time], label='Метод переменных направлений')
        else:
          plt.plot(x, z1[time], label='Метод дробных шагов')

        plt.plot(x, z2, label='Аналитический')
        plt.legend(loc='best')
        plt.ylabel('U')
        plt.xlabel('x')
        plt.show()

        solv = Schema(T=T, order2nd=firstMethod)
        h = []
        tau = []
        eps = []
        for i in tqdm(range(20)):
            h.append([])
            tau.append([])
            eps.append([])
            for j in range(40):
                N = i + 5
                K = j + 40
                X, Y, T, Z = solv(N, N, K)
                h[-1].append(solv.hx)
                tau[-1].append(solv.tau)
                eps[-1].append(FullError(X, Y, T, Z))

        t = range(0, 100, 5)
        err = [max(m) for m in eps]
        fig = plt.figure()
        plt.title('Зависимость погрешности от t')
        plt.plot(t, err)
        plt.legend(loc='best')
        plt.ylabel('погрешность')
        plt.xlabel('t')
        plt.show()
```

```
    fig = plt.figure()
    plt.title('Зависимость погрешности от шага по t')
    plt.plot(tau[-1], eps[-1])
    plt.legend(loc='best')
    plt.ylabel('погрешность')
    plt.xlabel('шаг')
    plt.show()

    ee = [eps[i][-1] for i in range(len(eps))]
    hh = [h[i][-1] for i in range(len(h))]

    fig = plt.figure()
    plt.title('Зависимость погрешности от h')
    plt.plot(hh, ee)
    plt.legend(loc='best')
    plt.ylabel('погрешность')
    plt.xlabel('шаг')
    plt.show()

    fig = plt.figure(num=1, figsize=(19, 12), clear=True)
    ax = fig.add_subplot(1, 1, 1, projection='3d')
    ax.plot_surface(np.array(h), np.array(tau), np.array(eps))
    ax.set(xlabel='$h$', ylabel='$t$', zlabel='погрешность',
title='Погрешность от шага и времени')
    fig.tight_layout()
    return (np.array(h), np.array(tau), np.array(eps))

h, tau, eps = DrawCharts(numericalAnswer, analyticAnswer, nx, ny, T, K,
plottingTime)

firstMethod = False
args['algorithm'] = 'FractionalSteps'
solver = ParabolicSolver(args)
numericalAnswer = solver.Solve(nx, ny, T, K),
analyticAnswer = solver.AnalyticSolve(nx, ny, T, K)

h1, tau1, eps1 = DrawCharts(numericalAnswer, analyticAnswer, nx, ny, T, K,
plottingTime, 1)
```