Московский авиационный институт

(Национальный исследовательский университет)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра вычислительной математики и программирования

Лабораторные работы

по курсу «Численные методы»

Вариант 2

Выполнил: Баранов И.В.

Группа: М8О-402Б-20

Преподаватель: доц. Пивоваров Д. Е.

Дата:

Оценка:

Москва, 2023

Оглавление

Лабораторная работа №5 3
- **Задание** 3
- **Теоретическая часть** 3
- **Код программы** 4
- **Решение** 8
  - **Явный метод** 8
  - **Неявный метод** 9
  - **Метод Кранка-Николсона** 10

Лабораторная работа №6 11
- **Задание** 11
- **Теоретическая часть** 12
- **Код программы** 12
- **Решение** 16
  - **Явный метод** 16
  - **Неявный метод** 16

Лабораторная работа №7 17
- **Задание** 17
- **Теоретическая часть** 18
- **Код программы** 18
- **Решение** 24
  - **Метод Либмана** 24
  - **Метод Зейделя** 24
  - **Метод простых итераций с верхней релаксацией** 25
  - **Сравнение погрешностей методов** 25

Лабораторная работа №8 26
- **Задание** 26
- **Теоретическая часть** 26
- **Код программы** 27
- **Решение** 36
  - **Метод переменных направлений** 36
  - **Метод дробных шагов** 37

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.

Реализованный метод	Решаемая задача
• Явная конечно-разностная схема • Неявная конечно-разностная схема • Разностная схема Кранка-Николсона	Начально-краевая задача для дифференциального уравнения параболического типа

2.

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

$u(0,t) = 0,$

$u(1,t) = 1,$

$u(x,0) = x + \sin(\pi x)$.

Аналитическое решение: $U(x,t) = x + \exp(-\pi^2 at)\sin(\pi x)$

Теоретическая часть

Данное уравнение представляет собой уравнение теплопроводности с граничными условиями первого рода.

Явная конечно-разностная схема имеет вид:

$$\frac{u_j^{k+1} - u_j^k}{\tau} = a^2\,\frac{u_{j+1}^k - 2u_j^k + u_{j-1}^k}{h^2}$$

неявная:

$$\frac{u_j^{k+1} - u_j^k}{\tau} = a^2 \frac{u_{j+1}^{k+1} - 2u_j^{k+1} + u_{j-1}^{k+1}}{h^2}$$

схема Кранка-Николсона:

$$\frac{u_j^{k+1} - u_j^k}{\tau} = \theta a^2 \frac{u_{j+1}^{k+1} - 2u_j^{k+1} + u_{j-1}^{k+1}}{h^2} + (1-\theta) a^2 \frac{u_{j+1}^k - 2u_j^k + u_{j-1}^k}{h^2},$$

где θ – вес явной части конечно-разностной схемы. Для нахождения неявной части конечно-разностной схемы применялся метод прогонки трехдиагональной матрицы.

Погрешности вычислялись как модуль разности точного и вычисленного решения.

Код программы

```
import numpy as np
import matplotlib.pyplot as plt
def analyt_func(x, a, c, t):
 return x + np.exp(-np.pi * np.pi * a * t) * np.sin(np.pi * x)
def func_border1(a, c, t):
 return 0
def func_border2(a, c, t):
 return 1
def run_through(a, b, c, d, s):
 P = np.zeros(s + 1)
 Q = np.zeros(s + 1)
 P[0] = -c[0] / b[0]
 Q[0] = d[0] / b[0]
 k = s - 1
 for i in range(1, s):
 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])
 P[k] = 0
 Q[k] = (d[k] - a[k] * Q[k - 1]) / (b[k] + a[k] * P[k - 1])
 x = np.zeros(s)
```

```
 x[k] = Q[k]
 for i in range(s - 2, -1, -1):
 x[i] = P[i] * x[i + 1] + Q[i]
 return x
def explicit(K, t, tau, h, a, c, x, approx): # явный метод
 N = len(x)
 U = np.zeros((K, N))
 for j in range(N):
 U[0, j] = x[j] + np.sin(np.pi * x[j])
 for k in range(K - 1):
 t += tau
 for j in range(1, N - 1):
 U[k + 1, j] = tau * (a * (U[k, j - 1] - 2 * U[k, j] + U[k, j + 1]) / h**2 +
\
 + c * U[k, j]) + U[k, j]
 if approx == 1:
 U[k + 1, 0] = func_border1(a, c, t)
 U[k + 1, N - 1] = func_border2(a, c, t)
 elif approx == 2:
 U[k + 1, 0] = func_border1(a, c, t)
 U[k + 1, N - 1] = func_border2(a, c, t)
 elif approx == 3:
 U[k + 1, 0] = func_border1(a, c, t)
 U[k + 1, N - 1] = func_border2(a, c, t)
 return U
def implicit(K, t, tau, h, a1, c1, x, approx): # неявный метод
 N = len(x)
 U = np.zeros((K, N))
 for j in range(N):
 U[0, j] = x[j] + np.sin(np.pi * x[j])
 for k in range(0, K - 1):
 a = np.zeros(N)
 b = np.zeros(N)
 c = np.zeros(N)
 d = np.zeros(N)
 t += tau
 for j in range(1, N - 1):
 a[j] = tau * (a1 / h**2 - 0 / (2 * h))
```

```
 b[j] = tau * ((-2 * a1) / h**2 + c1) - 1
 c[j] = tau * (a1 / h**2 + 0 / (2 * h))
 d[j] = -U[k][j]
 b[0] = 1
 c[0] = 0
 d[0] = 0
 a[N - 1] = 0
 b[N - 1] = 1
 d[N - 1] = 1
 u_new = run_through(a, b, c, d, N)
 for i in range(N):
 U[k + 1, i] = u_new[i]
 return U
def Krank_Nikolson(K, t, tau, h, a1, c1, x, approx, theta):
 N = len(x)
 if theta == 0:
 U = explicit(K, t, tau, h, a1, c1, x, approx)
 elif theta == 1:
 U = implicit(K, t, tau, h, a1, c1, x, approx)
 else:
 U_ex = explicit(K, t, tau, h, a1, c1, x, approx)
 U_im = implicit(K, t, tau, h, a1, c1, x, approx)
 U = np.zeros((K, N))
 for i in range(K):
 for j in range(N):
 U[i, j] = theta * U_im[i][j] + (1 - theta) * U_ex[i][j]
 return U
N = 50
K = 30000
time = 3
h = (2 - 0) / N
tau = time / K
x = np.arange(0, 1 + h / 2 - 1e-4, h)
T = np.arange(0, time, tau)
a = 1
c = 0
t = 0
dt = float(input("Введите момент времени от 0 до 3: "))
```

```
dt = int(dt*2333)
while (1):
 print("Выберите метод:\n"
 "|0| – ВЫХОД                 |\n"
 "|1| - ЯВНАЯ СХЕМА           |\n"
 "|2| - НЕЯВНАЯ СХЕМА         |\n"
 "|3| - СХЕМА КРАНКА-НИКОЛСОНА |")
 method = int(input())
 if method == 0:
 break
 else:
 print("АПРОКСИМАЦИЯ:\n"
 "1 - двухточечная аппроксимация с 1-ым порядком\n"
 "2 - трехточечная аппроксимация со 2-ым порядком\n"
 "3 - двухточечная аппроксимация со 2-ым порядком")
 approx = int(input())
 if method == 1:
 if a * tau / h**2 <= 0.5:
 print("Условие Куррента выполнено:", a * tau / h**2, "<= 0.5\n")
 U = explicit(K, t, tau, h, a, c, x, approx)
 else:
 print("Условие Куррента не выполнено:", a * tau / h**2, "> 0.5")
 break
 elif method == 2:
 U = implicit(K, t, tau, h, a, c, x, approx)
 elif method == 3:
 theta = 0.5;
 U = Krank_Nikolson(K, t, tau, h, a, c, x, approx, theta)
 U_analytic = analyt_func(x, a, c, T[dt])
 error = abs(U_analytic - U[dt, :])
 plt.title("График точного и численного решения задачи")
 plt.xlabel("x")
 plt.ylabel("U")
 plt.text(0.2, -0.8, "Максимальная ошибка метода: " + str(max(error)))
 plt.axis([0, 3, -1.2, 2])
 plt.plot(x, U_analytic, label = "Точное решение", color = "red")
 plt.scatter(x, U[dt, :], label = "Численное решение")
 plt.grid()
```

```
plt.legend()
plt.show()
##############################################################################
###########
plt.title("График ошибки по шагам")
error_time = np.zeros(len(T))
for i in range(len(T)):
error_time[i] = max(abs(analyt_func(x, a, c, T[i]) - U[i, :]))
plt.plot(T, error_time, label = "По времени")
plt.plot(x, error, label = "По пространству")
plt.legend()
plt.grid()
plt.show()
```

Решение

Явный метод

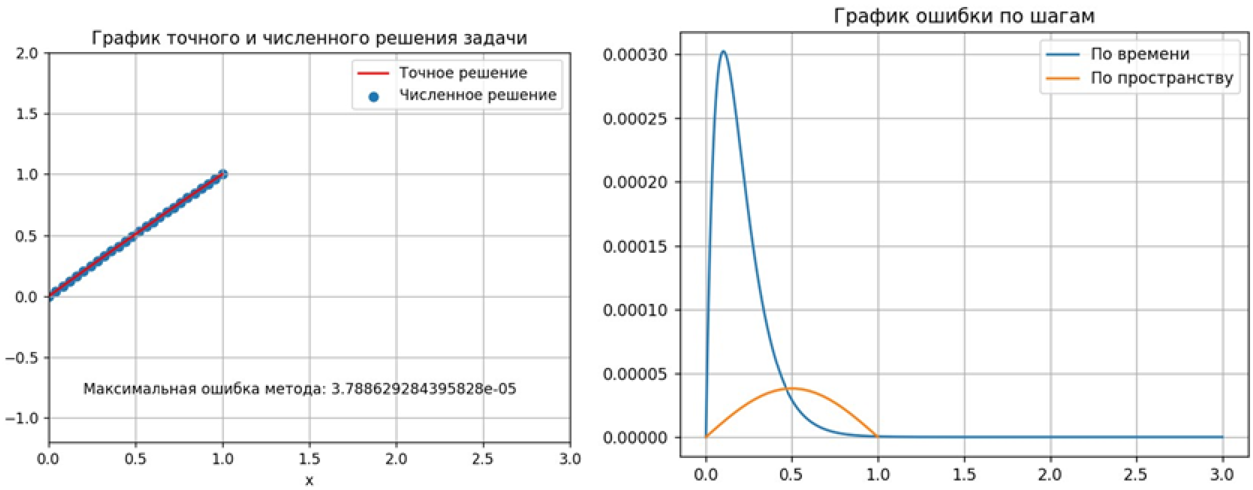

Рисунок 1.1. (Двухточечная аппроксимация с первым порядком точности)

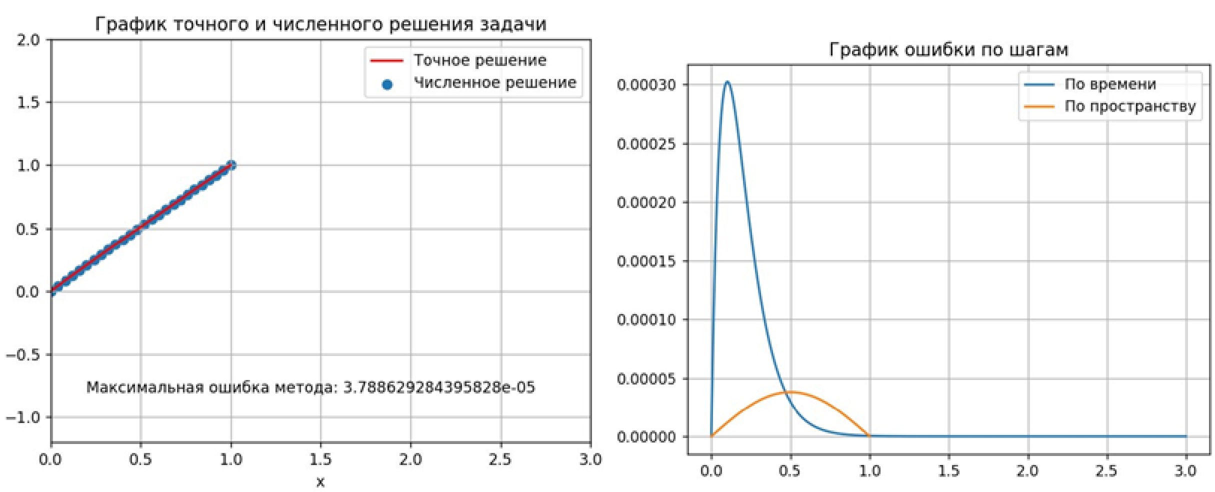

Рисунок 1.2. (Трёхточечная аппроксимация со вторым порядком точности)

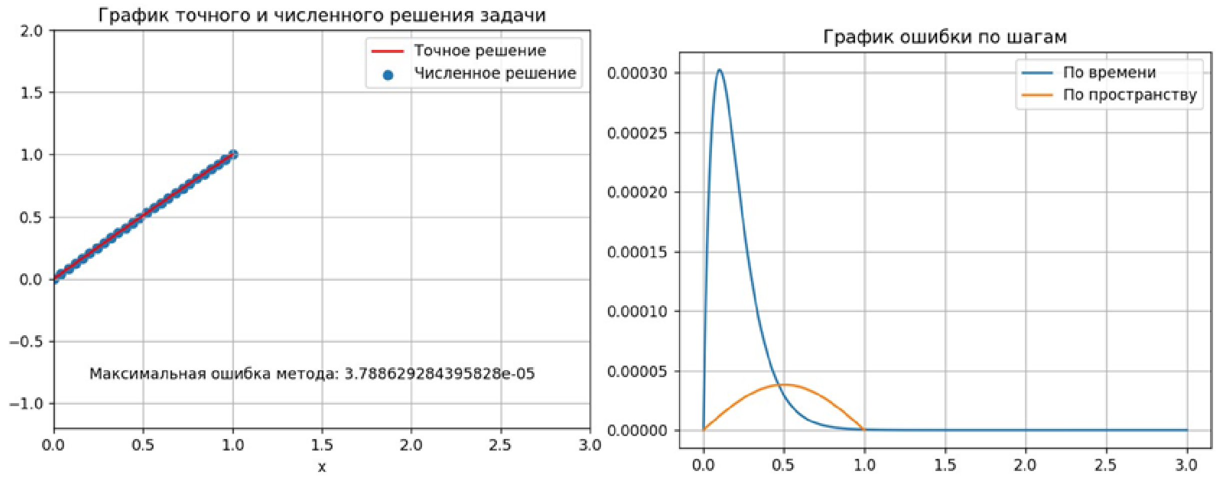

Рисунок 1.3. (Двухточечная аппроксимация со вторым порядком точности)

Неявный метод

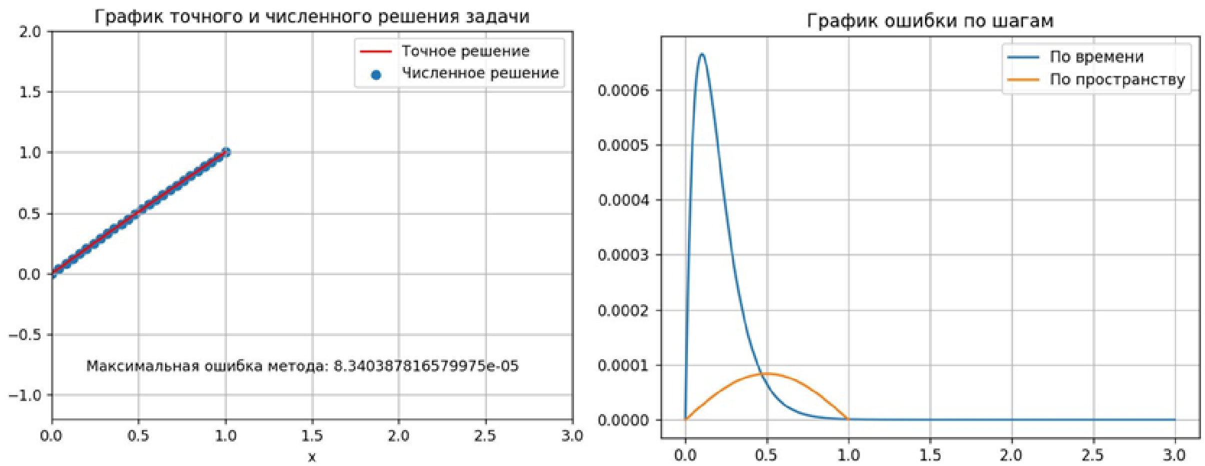

Рисунок 1.4. (Двухточечная аппроксимация с первым порядком точности)

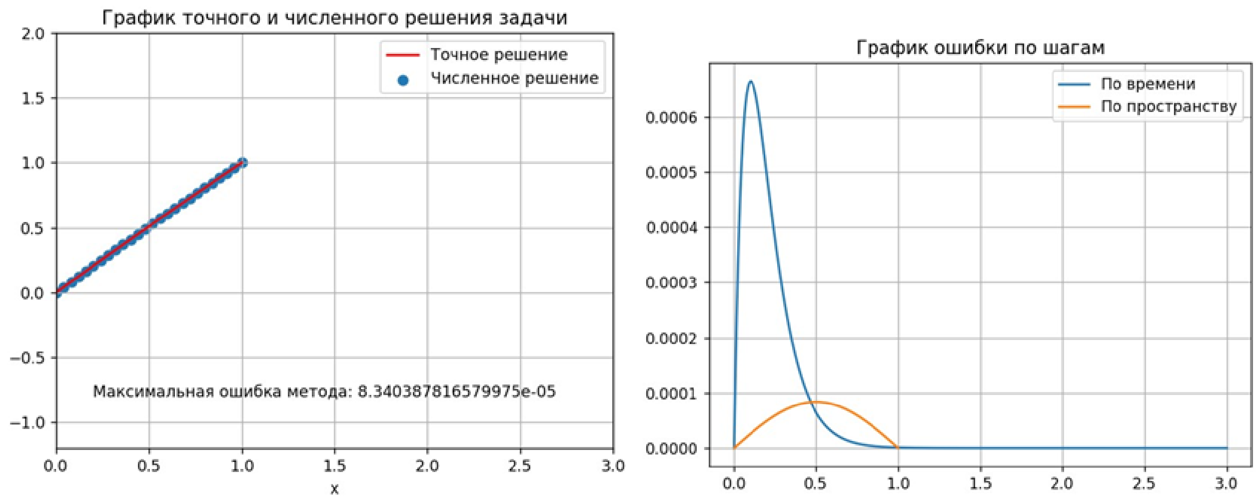

Рисунок 1.5. (Трёхточечная аппроксимация со вторым порядком точности)

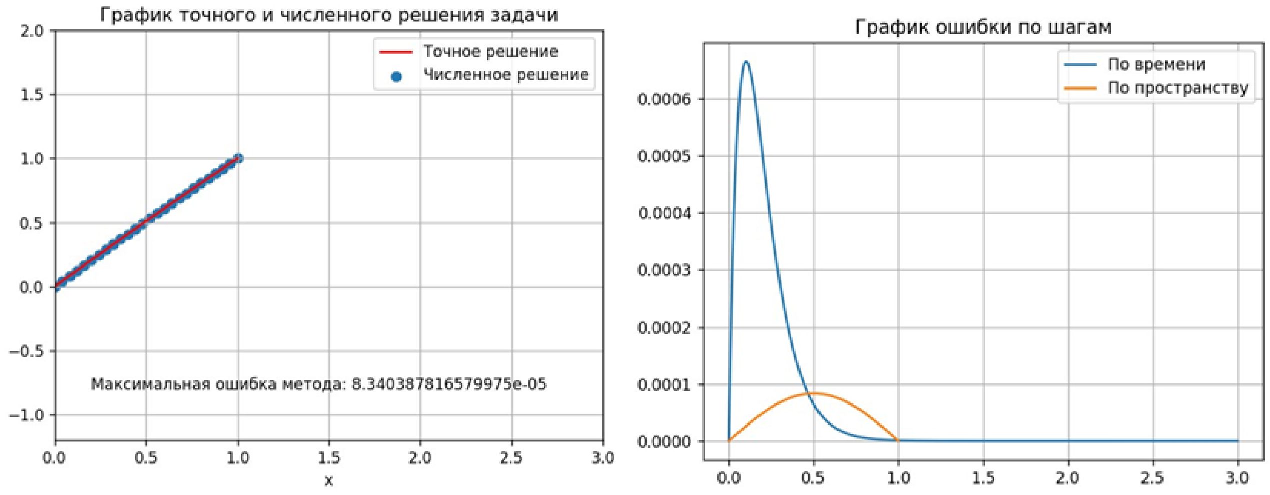

Рисунок 1.6. (Двухточечная аппроксимация со вторым порядком точности)

Метод Кранка-Николсона

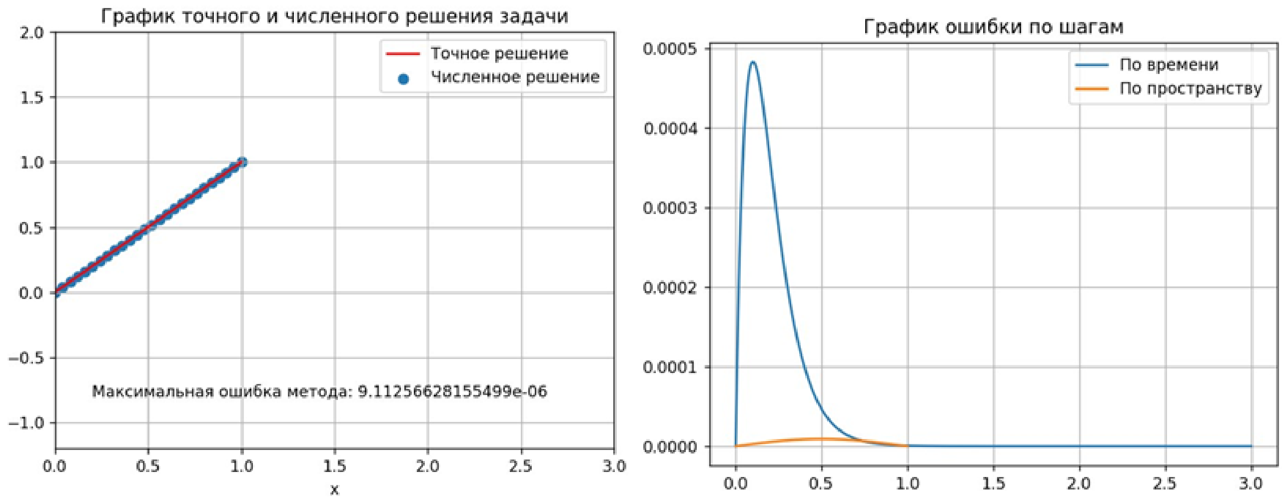

Рисунок 1.7. (Двухточечная аппроксимация с первым порядком точности)

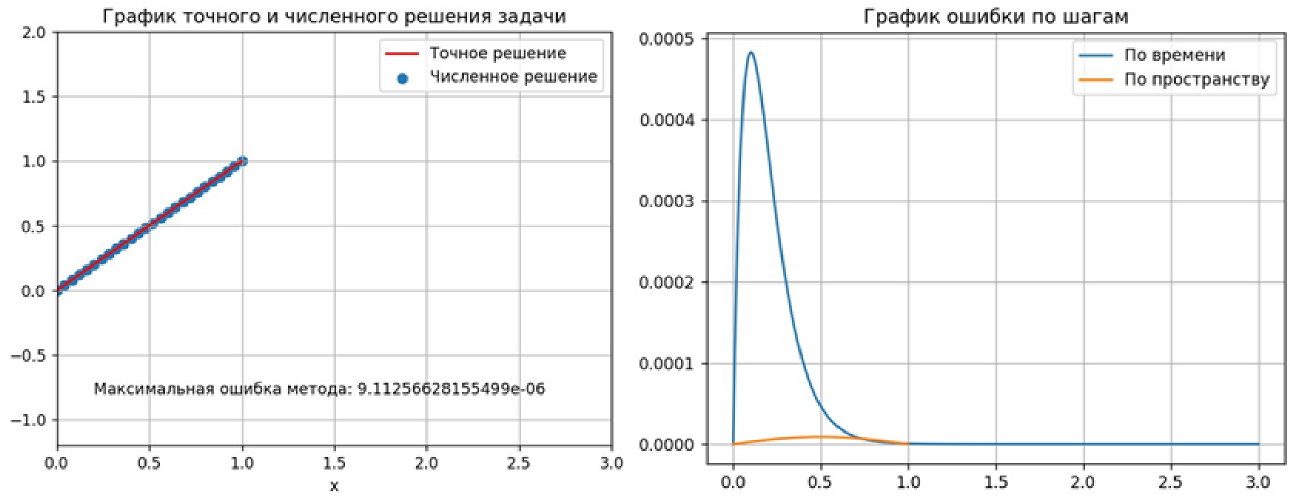

Рисунок 1.8. (Трёхточечная аппроксимация со вторым порядком точности)

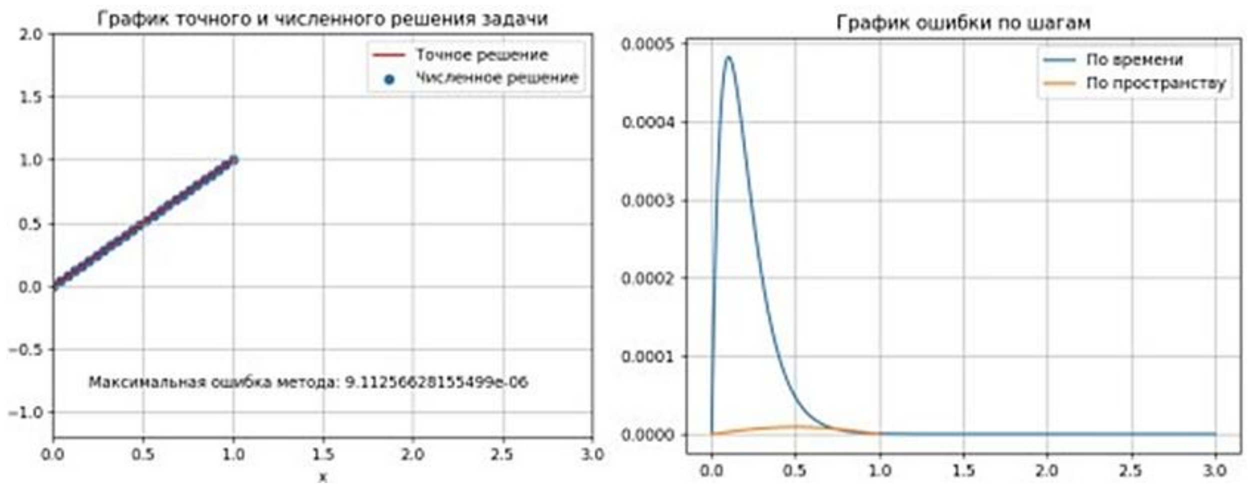

Рисунок 1.9. (Двухточечная аппроксимация со вторым порядком точности)

Лабораторная работа №6

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.

Реализованный метод	Решаемая задача
• Явная конечно-разностная схема («крест») • Неявная конечно-разностная схема	Начально-краевая задача для дифференциального уравнения гиперболического типа

2.

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

$u_x(0,t) - u(0,t) = 0,$

$u_x(\pi,t) - u(\pi,t) = 0,$

$u(x,0) = \sin x + \cos x,$

$u_t(x,0) = -a(\sin x + \cos x).$

Аналитическое решение: $U(x,t) = \sin(x - at) + \cos(x + at)$

Теоретическая часть

Явная конечно-разностная схема уравнения имеет вид:

$$\frac{u_j^{k+1} - 2u_j^k + u_j^{k-1}}{\tau^2} = a^2 \frac{u_{j+1}^k - 2u_j^k + u_{j-1}^k}{h^2}$$

неявная:

$$\frac{u_j^{k+1} - 2u_j^k + u_j^{k-1}}{\tau^2} = a^2 \frac{u_{j+1}^{k+1} - 2u_j^{k+1} + u_{j-1}^{k+1}}{h^2}$$

Для нахождения неявной части конечно-разностной схемы применялся метод прогонки трехдиагональной матрицы. Погрешности вычислялись как модуль разности точного и вычисленного решения.

Код программы

```
import numpy as np, matplotlib.pyplot as plt
def analyt_func(x, t):
 return np.sin(x-t)+np.cos(x+t)
def run_through(a, b, c, d, s):
 P = np.zeros(s + 1)
 Q = np.zeros(s + 1)
 P[0] = -c[0] / b[0]
 Q[0] = d[0] / b[0]
 k = s - 1
 for i in range(1, s):
 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])
 P[k] = 0
```

```
 Q[k] = (d[k] - a[k] * Q[k - 1]) / (b[k] + a[k] * P[k - 1])
 x = np.zeros(s)
 x[k] = Q[k]
 for i in range(s - 2, -1, -1):
 x[i] = P[i] * x[i + 1] + Q[i]
 return x
def explicit(K, t, tau, h, x, approx_st): # явный метод
 N = len(x)
 U = np.zeros((K, N))
 t += tau
 for j in range(N):
 U[0, j] = (np.sin(x[j])+np.cos(x[j]))
 if approx_st == 1:
 U[1][j] = (np.sin(x[j]) + np.cos(x[j])) - tau * (np.cos(x[j]) -
np.sin(x[j]))
 if approx_st == 2:
 U[1][j] = (np.sin(x[j]) + np.cos(x[j])) - tau * (np.cos(x[j]) +
np.sin(x[j])) - (tau ** 2) / 2 * (
 np.cos(x) - np.sin(x)) + (-np.sin(x[j]) - np.cos(x[j])) * (tau ** 2) / 2
 for k in range(1, K - 1):
 t += tau
 for j in range(1, N - 1):
 U[k + 1, j] = ((tau ** 2) * (U[k, j + 1] - 2 * U[k, j] + U[k, j - 1]) / (h
** 2)) + 2 * U[k, j] - U[k -
1, j]
 U[k, 0] = (h * (np.sin(tau * k)+np.cos(tau * k)) - U[k, 1]) / -1
 U[k, N - 1] = (-h * (np.sin(tau * k)+np.cos(tau * k)) - U[k, N - 2]) / -1
 return U
def implicit(K, t, tau, h, x, approx_st): # неявный метод
 N = len(x)
 U = np.zeros((K, N))
 t += tau
 for j in range(N):
 U[0, j] = (np.sin(x[j])+np.cos(x[j]))
 if approx_st == 1:
 U[1][j] = (np.sin(x[j])+np.cos(x[j])) - tau * (np.cos(x[j])-np.sin(x[j]))
 if approx_st == 2:
 U[1][j] = (np.sin(x[j])+np.cos(x[j])) - tau * (np.cos(x[j])+np.sin(x[j]))-
(tau ** 2) / 2 *
```

```
(np.cos(x)-np.sin(x)) + (-np.sin(x[j])-np.cos(x[j])) * (tau ** 2) / 2
 for k in range(K - 1):
 U[k + 1, 0] = U[k + 1, 1]
 U[k + 1, N - 1] = U[k + 1, N - 2]
 for k in range(1, K - 1):
 a = np.zeros(N)
 b = np.zeros(N)
 c = np.zeros(N)
 d = np.zeros(N)
 t += tau
 for j in range(1, N - 1):
 a[j] = (tau ** 2) * (1 / h ** 2 - 1 / (2 * h))
 b[j] = -2 * tau ** 2 / (h ** 2) - tau ** 2 - 1 - 3 * tau / 2
 c[j] = (tau ** 2) * (1 / h ** 2 + 1 / (2 * h))
 d[j] = -2 * U[k, j]+ U[k - 1, j]
 b[0] = 1
 c[0] = 0
 d[0] = 0
 a[N - 1] = 0
 b[N - 1] = 1
 d[N - 1] = 0
 u_new = run_through(a, b, c, d, N)
 for i in range(N):
 U[k + 1, i] = u_new[i]
 return U
N = 100
K = 7000
time = 3
h = (np.pi - 0) / N
tau = time / K
x = np.arange(0, np.pi + h / 2 - 1e-4, h)
T = np.arange(0, time, tau)
t = 0
dt = float(input("Введите момент времени от 0 до 3: "))
dt = int(dt*2333)
while (1):
 print("МЕТОД:\n"
 "0 - ВЫХОД\n"
```

```
"1 - ЯВНАЯ СХЕМА\n"
"2 - НЕЯВНАЯ СХЕМА")
method = int(input())
if method == 0:
break
else:
print("АПРОКСИМАЦИЯ НАЧАЛЬНЫХ УСЛОВИЙ:\n"
"1 - 1-ОГО ПОРЯДКА\n"
"2 - 2-ОГО ПОРЯДКА")
approx_st = int(input())
if method == 1:
if tau / h**2 <= 1:
print("Условие Куррента выполнено:", tau / h**2, "<= 1\n")
U = explicit(K, t, tau, h, x, approx_st)
else:
print("Условие Куррента не выполнено:", tau / h**2, "> 1")
break
elif method == 2:
U = implicit(K, t, tau, h, x, approx_st)
U_analytic = analyt_func(x, T[dt])
error = abs(U_analytic - U[dt, :])
plt.title("График точного и численного решения задачи")
plt.xlabel("x")
plt.ylabel("y")
plt.text(0.2, -0.8, "Максимальная ошибка метода: " + str(max(error)))
plt.axis([0, 3, -2.2, 2.2])
plt.scatter(x, U_analytic, label = "Точное решение", color = "green")
plt.plot(x, U[dt, :], label = "Численное решение", color = "red")
plt.grid()
plt.legend()
plt.show()
plt.title("График зависимости ошибок по времени и пространству")
error_time = np.zeros(len(T))
for i in range(len(T)):
error_time[i] = max(abs(analyt_func(x, T[i]) - U[i, :]))
plt.plot(T, error_time, label = "По времени")
plt.plot(x, error, label = "По пространству")
plt.legend()
```

```
plt.grid()
plt.show()
```

Решение

Явный метод

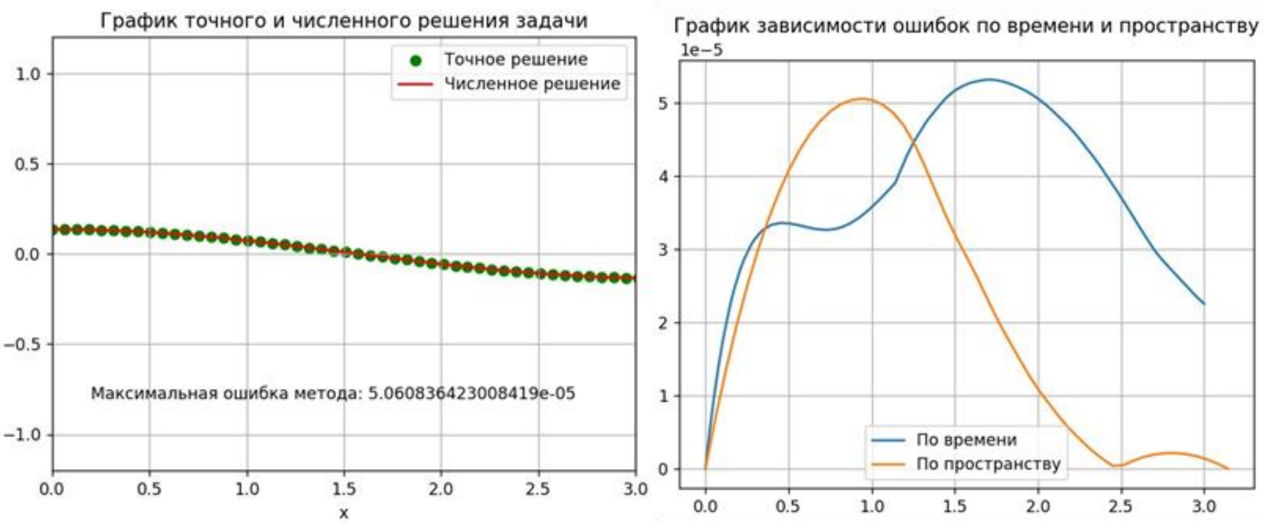

Рисунок 2.1. (Аппроксимация с первым порядком точности)

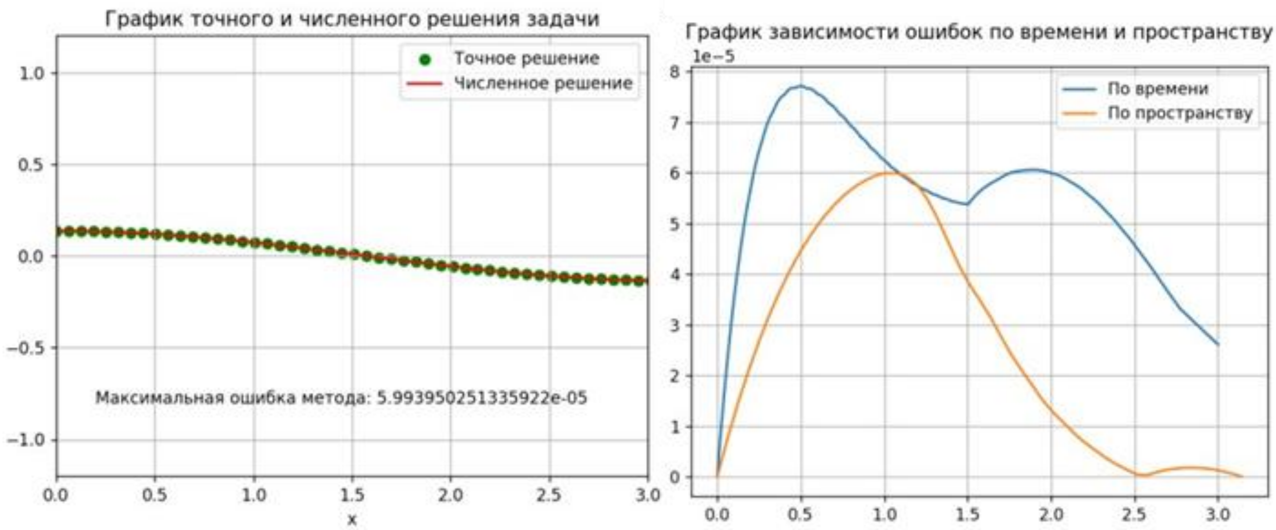

Рисунок 2.2. (Аппроксимация со вторым порядком точности)

Неявный метод

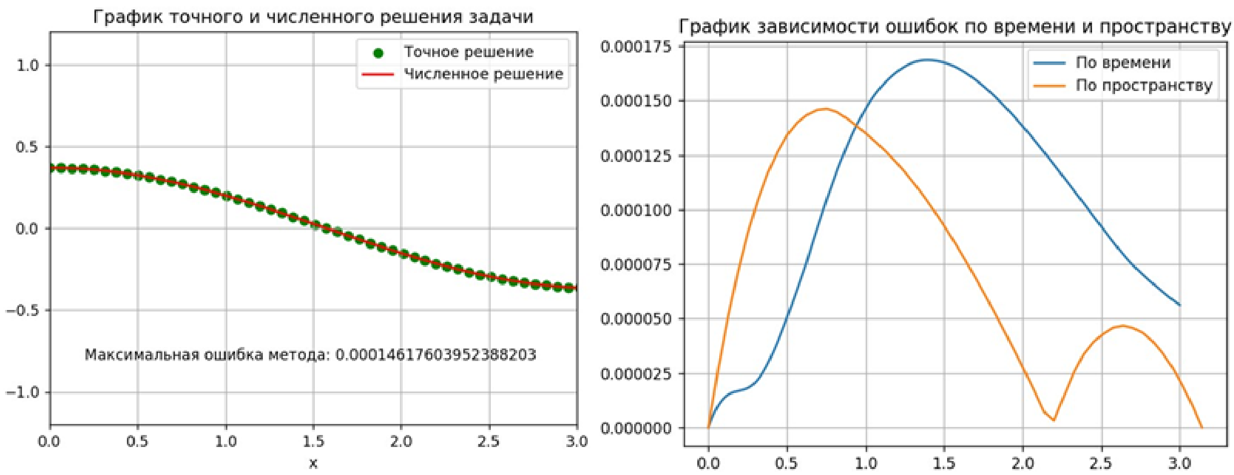

Рисунок 2.3. (Аппроксимация с первым порядком точности)

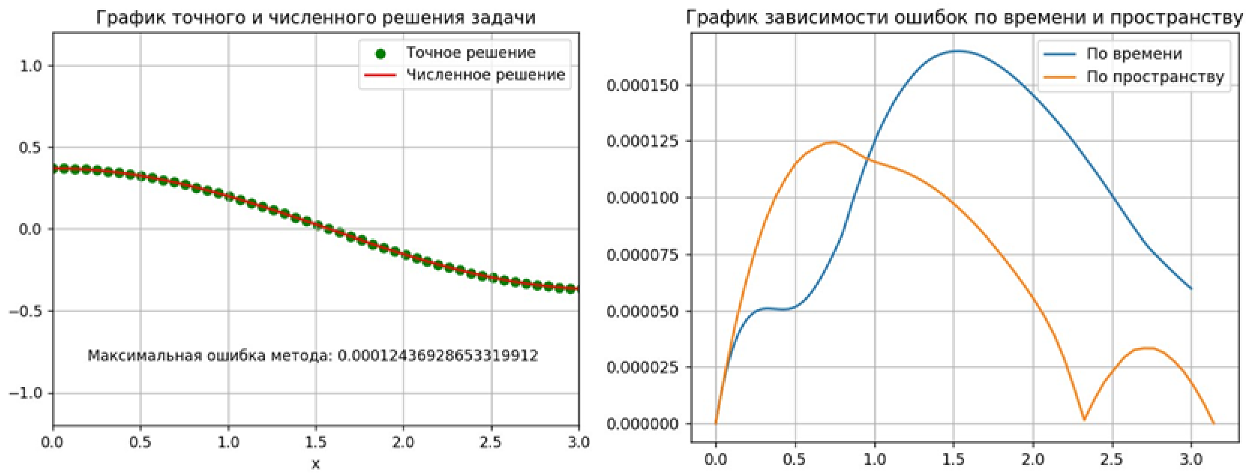

Рисунок 2.4. (Аппроксимация со вторым порядком точности)

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.

Реализованный метод	Решаемая задача

• Метод простых итераций (метод Либмана) • Метод Зейделя	Краевая задача для дифференциального уравнения эллиптического типа

2.

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

$u_x(0, y) = 0,$

$u(1, y) = 1 - y^2,$

$u_y(x,0) = 0,$

$u(x,1) = x^2 - 1.$

Аналитическое решение: $U(x, y) = x^2 - y^2$.

Теоретическая часть

Конечно-разностная схема уравнения имеет вид:

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h_1^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h_2^2}$$

в результате чего была получена пятидиагональная матрица СЛАУ. Для нахождения узлов применялись итерационные методы: метод Либмана (простых итераций), метод Зейделя и метод простых итераций с верхней релаксацией. Погрешности вычислялись как модуль разности точного и вычисленного решения.

Код программы

```
import matplotlib.pyplot as plt
import numpy as np
def psi_0(y):
 return 0
def psi_1(y):
 return -(1 - y ** 2)
def phi_0(x):
 return 0
def phi_1(x):
 return -(x ** 2 - 1)
def u(x, y):
 return x ** 2 - y ** 2
```

```
class Schema:
 def __init__(self, psi0=psi_0, psi1=psi_1, phi0=phi_0, phi1=phi_1, lx0=0,
 lx1=1, ly0=0, ly1=1, solver="zeidel", relax=1.5, epsilon=0.01):
 self.psi1 = psi1
 self.psi0 = psi0
 self.phi0 = phi0
 self.phi1 = phi1
 self.lx0 = lx0
 self.ly0 = ly0
 self.lx1 = lx1
 self.ly1 = ly1
 self.eps = epsilon
 self.method = None
 if solver == "zeidel":
 self.method = self.ZeidelStep
 elif solver == "simple":
 self.method = self.SimpleEulerStep
 elif solver == "relaxation":
 self.method = lambda x, y, m: self.RelaxationStep(x, y, m, relax)
 else:
 raise ValueError("Wrong solver name")
 def ZeidelStep(self, X, Y, M):
 return self.RelaxationStep(X, Y, M, w=1)
 def RelaxationStep(self, X, Y, M, w):
 norm = 0.0
 hx2 = self.hx * self.hx
 hy2 = self.hy * self.hy
 for i in range(1, self.Ny - 1):
 for j in range(1, self.Nx - 1):
 diff = hy2 * (M[i][j - 1] + M[i][j + 1])
 diff += hx2 * (M[i - 1][j] + M[i + 1][j])
 diff /= 2 * (hy2 + hx2 - hx2 * hy2)
 diff -= M[i][j]
 diff *= w
 M[i][j] += diff
 diff = abs(diff)
 norm = diff if diff > norm else norm
 return norm
```

```
def SimpleEulerStep(self, X, Y, M):
tmp = [[0.0 for _ in range(self.Nx)] for _ in range(self.Ny)]
norm = 0.0
hx2 = self.hx * self.hx
hy2 = self.hy * self.hy
for i in range(1, self.Ny - 1):
tmp[i][0] = M[i][0]
for j in range(1, self.Nx - 1):
tmp[i][j] = hy2 * (M[i][j - 1] + M[i][j + 1])
tmp[i][j] += hx2 * (M[i - 1][j] + M[i + 1][j])
tmp[i][j] /= 2 * (hy2 + hx2 - hx2 * hy2)
diff = abs(tmp[i][j] - M[i][j])
norm = diff if diff > norm else norm
tmp[i][-1] = M[i][-1]
for i in range(1, self.Ny - 1):
M[i] = tmp[i]
return norm
def set_l0_l1(self, lx0, lx1, ly0, ly1):
self.lx0 = lx0
self.lx1 = lx1
self.ly0 = ly0
self.ly1 = ly1
def _compute_h(self):
self.hx = (self.lx1 - self.lx0) / (self.Nx - 1)
self.hy = (self.ly1 - self.ly0) / (self.Ny - 1)
@staticmethod
def nparange(start, end, step=1):
now = start
e = 0.00000000001
while now - e <= end:
yield now
now += step
def init_values(self, X, Y):
ans = [[0.0 for _ in range(self.Nx)] for _ in range(self.Ny)]
for j in range(1, self.Nx - 1):
coeff = (self.psi1(X[-1][j]) - self.psi0(Y[0][j])) / (self.ly1 - self.ly0)
addition = self.psi0(X[0][j])
for i in range(self.Ny):
```

```
 ans[i][j] = coeff * (Y[i][j] - self.ly0) + addition
 for i in range(self.Ny):
 ans[i][0] = self.phi0(Y[i][0])
 ans[i][-1] = self.phi1(Y[i][-1])
 return ans
 def __call__(self, Nx=10, Ny=10):
 self.Nx, self.Ny = Nx, Ny
 self._compute_h()
 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)]
 ans = self.init_values(X, Y)
 self.itters = 0
 while (self.method(X, Y, ans) >= self.eps):
 self.itters += 1
 ans=np.array(ans)
 return X, Y, ans
def real_z(lx0, lx1, ly0, ly1, f):
 x = np.arange(lx0, lx1 + 0.001, 0.001)
 y = np.arange(ly0, ly1 + 0.001, 0.001)
 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])
 return X, Y, Z
def plot(Nx=5, Ny=5, eps=1, solv="simple"):
 schema = Schema(epsilon=eps, solver=solv, relax=1.6)
 x, y, z = schema(Nx, Ny)
 fig = plt.figure(num=1, figsize=(10, 7), clear=True)
 ax = fig.add_subplot(1, 1, 1, projection='3d')
```

```
 #ax.plot_wireframe(*real_z(0, 1, 0, 1, u), color="green")
 ax.plot_surface(np.array(x), np.array(y), np.array(z))
 ax.set(xlabel='x', ylabel='y', zlabel='u', title=f"График приближения
функции конечно разностным методом")
 fig.tight_layout()
 plt.show()
def error(x, y, z, f):
 ans = 0.0
 for i in range(len(z)):
 for j in range(len(z[i])):
 ans += (f(x[i][j], y[i][j]) - z[i][j])**2
 return ((ans)**0.5)/(len(z)*len(z[0]))
def get_step_error(solver, real_f): # вычисление погрешности
 h = []
 e = []
 N=10
 x, y, z = solver(N, N)
 h.append(solver.hx)
 e.append(error(x, y, z, real_f))
 N=20
 x, y, z = solver(N, N)
 h.append(solver.hx)
 e.append(error(x, y, z, real_f))
 N=40
 x, y, z = solver(N, N)
 h.append(solver.hx)
 e.append(error(x, y, z, real_f))
 return h, e
def error_step(eps, r): # построение графика погрешности
 simp = Schema(epsilon=eps,solver="simple")
 zeid = Schema(epsilon=eps,solver="zeidel")
 relax = Schema(epsilon=eps,solver="relaxation", relax=r)
 plt.figure(figsize = (10, 7))
 plt.title(f"Зависимость погрешности от длины шага eps={eps}, w={r},
(N=10,20,40)")
 h_s, e_s = get_step_error(simp, u)
 h_z, e_z = get_step_error(zeid, u)
 h_r, e_r = get_step_error(relax, u)
```

```
 plt.plot(h_s, e_s, label="Метод Простых Итераций", color = "red")

 plt.plot(h_z, e_z, label="Метод Зейделя", color="green")

 plt.plot(h_r, e_r, label="Метод Простых Итераций С Верхней Релаксацией",
color="blue")

 plt.xlabel("h")

 plt.ylabel("error")

 plt.legend()

 plt.grid()

 plt.show()

eps_0, eps_1, eps_2, eps_3, eps_4, eps_5, eps_6, eps_7 = 1.0, 0.1, 0.01,
0.001, 0.0001, 0.00001,

0.000001, 0.0000001

Nx = 10

Ny = 10

r = 1.6

eps = eps_2

plot(Nx, Ny, eps, solv="simple")

plot(Nx, Ny, eps, solv="zeidel")

plot(Nx, Ny, eps, solv="relaxation")

error_step(eps, r=r)

schema = Schema(epsilon=eps, solver="simple")

schema(Nx, Ny)

print("Количество итераций метода простых итераций:", schema.itters)

schema = Schema(epsilon=eps)

schema(Nx, Ny)

print("Количество итераций метода Зейделя:", schema.itters)

schema = Schema(epsilon=eps, solver="relaxation", relax=r)

schema(Nx, Ny)

print("Количество итераций метода верхних релаксаций:", schema.itters)
```

Решение

Метод Либмана

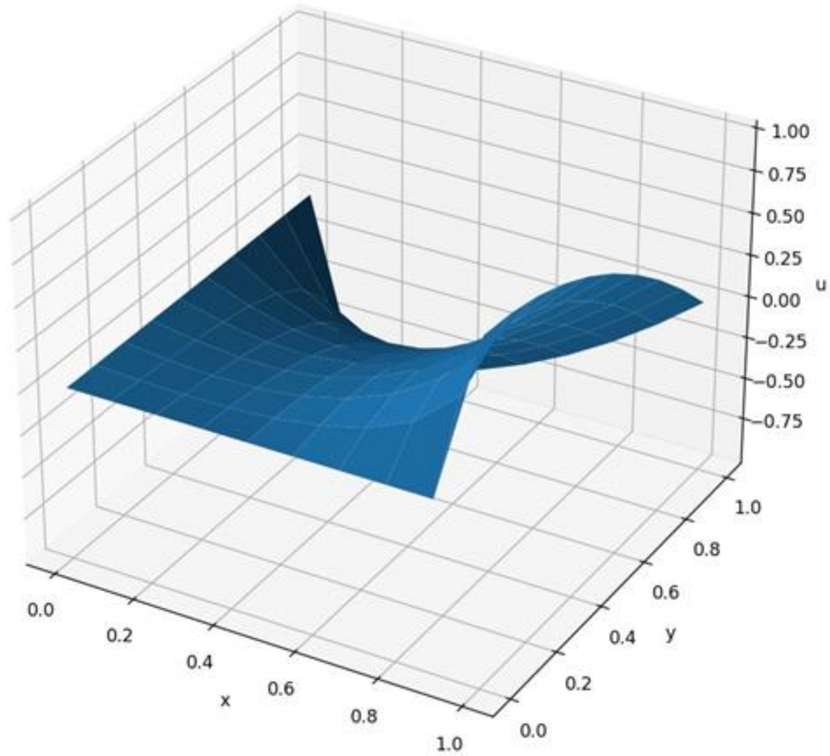

Рисунок 3.1. (Решение краевой задачи методом Либмана)

Метод Зейделя

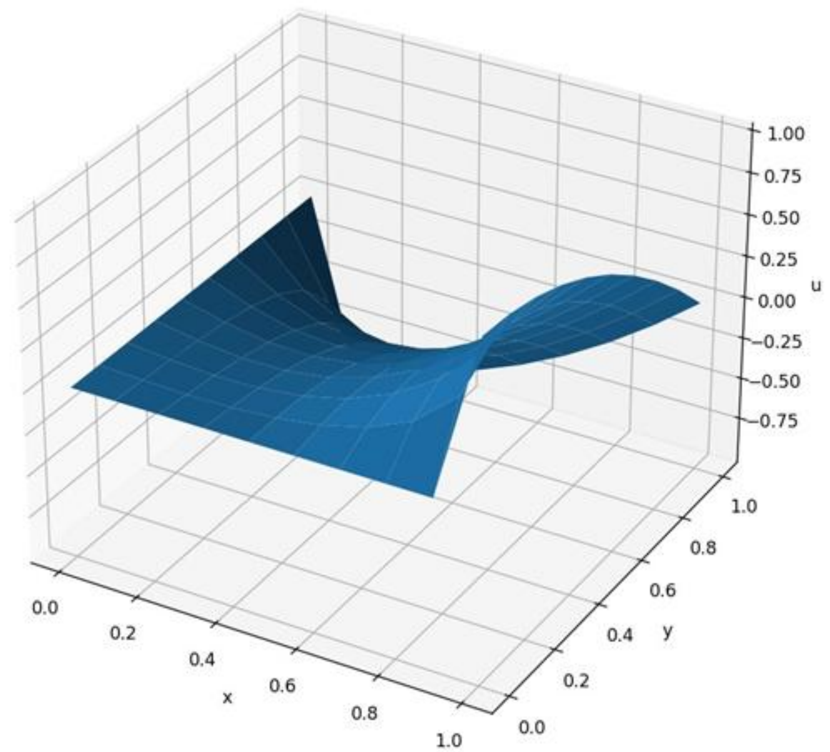

Рисунок 3.2. (Решение краевой задачи методом Зейделя)

Метод простых итераций с верхней релаксацией

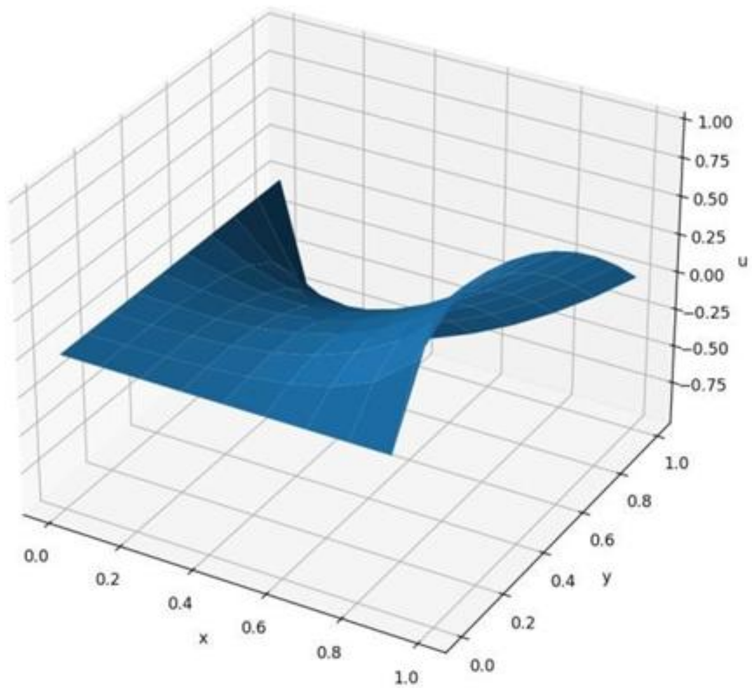

Рисунок 3.3. (Решение краевой задачи методом простых итераций с верхней релаксацией)

Сравнение погрешностей методов

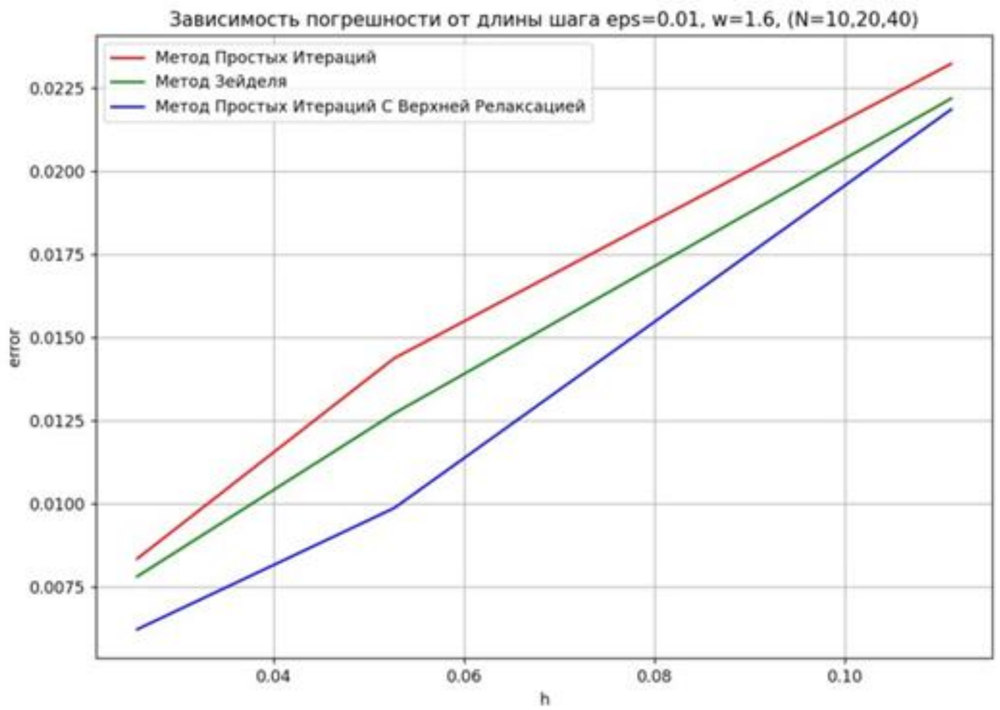

Рисунок 3.4. (График сравнения погрешностей методов)

Количество итераций метода простых итераций: 20

Количество итераций метода Зейделя: 17

Количество итераций метода верхних релаксаций: 12

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.

Реализованный метод	Решаемая задача
• Метод переменных направлений • Метод дробных шагов	Двумерная начально-краевая задача для дифференциального уравнения параболического типа

2.

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

$$u(0,y,t) = \cos(\mu_2 y)\exp(-(\mu_1^2 + \mu_2^2)at),$$

$$u(\frac{\pi}{2}\mu_1, y, t) = 0,$$

$$u(x,0,t) = \cos(\mu_1 x)\exp(-(\mu_1^2 + \mu_2^2)at),$$

$$u(x,\frac{\pi}{2}\mu_2, t) = 0,$$

$$u(x,y,0) = \cos(\mu_1 x)\cos(\mu_2 y).$$

Аналитическое решение: $U(x,y,t) = \cos(\mu_1 x)\cos(\mu_2 y)\exp(-(\mu_1^2 + \mu_2^2)at)$.

1). $\mu_1 = 1,\ \mu_2 = 1$.

2). $\mu_1 = 2,\ \mu_2 = 1$.

3). $\mu_1 = 1,\ \mu_2 = 2$.

Теоретическая часть

Для метода переменных направлений была составлена схема:

$$\frac{u_{ij}^{k+1/2} - u_{ij}^k}{\tau/2} = \frac{a}{h_1^2}\left(u_{i+1j}^{k+1/2} - 2u_{ij}^{k+1/2} + u_{i-1j}^{k+1/2}\right) + \frac{a}{h_2^2}\left(u_{ij+1}^k - 2u_{ij}^k + u_{ij-1}^k\right) + f_{ij}^{k+1/2},$$

$$\frac{u_{ij}^{k+1} - u_{ij}^{k+1/2}}{\tau/2} = \frac{a}{h_1^2}\left(u_{i+1j}^{k+1/2} - 2u_{ij}^{k+1/2} + u_{i-1j}^{k+1/2}\right) + \frac{a}{h_2^2}\left(u_{ij+1}^{k+1} - 2u_{ij}^{k+1} + u_{ij-1}^{k+1}\right) + f_{ij}^{k+1/2}.$$

В этих схемах последовательно аппроксимировались операторы $a\frac{\partial^2}{\partial x^2}$ и $a\frac{\partial^2}{\partial y^2}$.

Для метода дробных шагов была составлена схема:

$$\frac{u_{ij}^{k+1/2} - u_{ij}^{k}}{\tau} = \frac{a}{h_1^2}\left(u_{i+1j}^{k+1/2} - 2u_{ij}^{k+1/2} + u_{i-1j}^{k+1/2}\right)$$

$$\frac{u_{ij}^{k+1} - u_{ij}^{k+1/2}}{\tau} = \frac{a}{h_2^2}\left(u_{i\,j+1}^{k+1} - 2u_{ij}^{k+1} + u_{i\,j-1}^{k+1}\right)$$

Последовательно вычислялись прогонки в направлении осей x и y. Погрешности вычислялись как модуль разности точного и вычисленного решения.

Код программы

```
import random
import matplotlib.pyplot as plt
import math
import numpy as np
def psi_0(x, t, a=1):
 return math.cos(x)*math.exp(-2*a*t)
def psi_1(x, t, a=1):
 return 0
def phi_0(y, t, a=1):
 return math.cos(y)*math.exp(-2*a*t)
def phi_1(y, t, a=1):
 return 0
def u0(x, y):
 return math.cos(x)*math.cos(y)
def u(x, y, t, a=1):
 return math.cos(x)*math.cos(y)*math.exp(-2*a*t)
class Schema:
 def __init__(self, rho=u0, psi0=psi_0, psi1=psi_1, phi0=phi_0, phi1=phi_1, 
lx0=0, lx1=math.pi/2,
ly0=0, ly1=math.pi/2, 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 compute_h(self):
self.hx = (self.lx1 - self.lx0) / (self.Nx - 1)
self.hy = (self.ly1 - self.ly0) / (self.Ny - 1)
self.hx2 = self.hx * self.hx
self.hy2 = self.hy * self.hy
def compute_tau(self):
self.tau = self.T / (self.K - 1)
@staticmethod
def progonka(A, b):
P = [-item[2] for item in A]
Q = [item for item in b]
```

```
P[0] /= A[0][1]
Q[0] /= A[0][1]
for i in range(1, len(b)):
z = (A[i][1] + A[i][0] * P[i - 1])
P[i] /= z
Q[i] -= A[i][0] * Q[i - 1]
Q[i] /= z
for i in range(len(Q) - 2, -1, -1):
Q[i] += P[i] * Q[i + 1]
return Q
@staticmethod
def nparange(start, end, step=1):
now = start
e = 0.00000000001
while now - e <= end:
yield now
now += step
def compute_left_edge(self, X, Y, t, square):
for i in range(self.Ny):
square[i][0] = self.phi0(Y[i][0], t)
def compute_right_edge(self, X, Y, t, square):
for i in range(self.Ny):
square[i][-1] = self.phi1(Y[i][-1], t)
def compute_bottom_edge(self, X, Y, t, square):
for j in range(1, self.Nx - 1):
square[0][j] = self.psi0(X[0][j], t)
def compute_top_edge(self, X, Y, t, square):
for j in range(1, self.Nx - 1):
square[-1][j] = self.psi1(X[-1][j], t)
def compute_line_first_step(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] -
 c * now_square[i][0]
 ]
 A.extend([(c, b, c) for _ in range(2, self.Nx - 2)])
 w.extend([
 -self.cy * self.order * last_square[i - 1][j] -
 ((self.order + 1) * hx2 * hy2 - 2 * self.cy * self.order) *
last_square[i][j] -
 self.cy * self.order * last_square[i + 1][j]
 for j in range(2, self.Nx - 2)
 ])
 A.append((c, b, 0))
 w.append(
 -self.cy * self.order * last_square[i - 1][-2] -
 ((self.order + 1) * hx2 * hy2 - 2 * self.cy * self.order) * last_square[i][-
2] -
 self.cy * self.order * last_square[i + 1][-2] -
 c * now_square[i][-1]
 )
 line = self.progonka(A, w)
 for j in range(1, self.Nx - 1):
 now_square[i][j] = line[j - 1]
 def compute_line_second_step(self, j, X, Y, t, last_square, now_square):
 hx2 = self.hx2
 hy2 = self.hy2
 c = self.cy
 b = self.by
 A = [(0, b, c)]
 w = [
 -self.cx * self.order * last_square[1][j - 1] -
 ((self.order + 1) * hx2 * hy2 - 2 * self.cx * self.order) *
last_square[1][j] -
 self.cx * self.order * last_square[1][j + 1] -
 c * now_square[0][j]
 ]
 A.extend([(c, b, c) for _ in range(2, self.Ny - 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]
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] -
c * now_square[-1][j]
)
line = self.progonka(A, w)
for i in range(1, self.Ny - 1):
now_square[i][j] = line[i - 1]
def compute_square(self, X, Y, t, last_square):
square = [[0.0 for _ in range(self.Nx)] for _ in range(self.Ny)]
self.compute_left_edge(X, Y, t - 0.5 * self.tau, square)
self.compute_right_edge(X, Y, t - 0.5 * self.tau, square)
self.compute_bottom_edge(X, Y, t - 0.5 * self.tau, square)
self.compute_top_edge(X, Y, t - 0.5 * self.tau, square)
for i in range(1, self.Ny - 1):
self.compute_line_first_step(i, X, Y, 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.compute_left_edge(X, Y, t, square)
self.compute_right_edge(X, Y, t, square)
self.compute_bottom_edge(X, Y, t, square)
self.compute_top_edge(X, Y, t, square)
for j in range(1, self.Nx - 1):
self.compute_line_second_step(j, X, Y, t, last_square, square)
return square
def init_t0(self, X, Y):
first = [[0.0 for _ in range(self.Nx)] for _ in range(self.Ny)]
for i in range(self.Ny):
for j in range(self.Nx):
```

```
 first[i][j] = self.rho0(X[i][j], Y[i][j])
 return first
 def __call__(self, Nx=20, Ny=20, K=20):
 self.Nx, self.Ny, self.K = Nx, Ny, K
 self.compute_tau()
 self.compute_h()
 self.bx = -2 * self.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.compute_square(X, Y, t, ans[-1]))
 taus.append(t)
 return X, Y, taus, ans
def real_z_by_time(lx0, lx1, ly0, ly1, t, f):
 x = np.arange(lx0, lx1 + 0.002, 0.002)
 y = np.arange(ly0, ly1 + 0.002, 0.002)
 X = np.ones((y.shape[0], x.shape[0]))
 Y = np.ones((x.shape[0], y.shape[0]))
 Z = np.ones((y.shape[0], x.shape[0]))
 for i in range(Y.shape[0]):
 Y[i] = y
 Y = Y.T
 for i in range(X.shape[0]):
 X[i] = x
 for i in range(Z.shape[0]):
 for j in range(Z.shape[1]):
 Z[i, j] = f(X[i, j], Y[i, j], t)
 return X, Y, Z
def 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 plot_by_time(X, Y, T, Z, j, extrems):
 t = T[j]
 z = Z[j]
 fig = plt.figure(num=1, figsize=(10, 7), clear=True)
 ax = fig.add_subplot(1, 1, 1, projection='3d')
 ax.plot_surface(np.array(X), np.array(Y), np.array(z))
 ax.plot_wireframe(*real_z_by_time(0, math.pi/2, 0, math.pi/2, 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="center", fontsize=12)
 ax.set_zlim(extrems[0], extrems[1])
 fig.tight_layout()
 plt.show()
 return fig
def square_minmax(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 search_minmax(zz):
 minimum, maximum = 0.0, 0.0
 for z in zz:
 minmax = square_minmax(z)
 minimum = minmax[0] if minmax[0] < minimum else minimum
 maximum = minmax[1] if minmax[1] > maximum else maximum
 return minimum, maximum
def plot(nx, ny, k, t, order):
```

```
 schema = Schema(T=t, order2nd=order)
 xx, yy, tt, zz = schema(Nx=nx, Ny=ny, K=k)
 extrems = search_minmax(zz)
 plots = []
 for j in range(len(tt)):
 plots.append(plot_by_time(xx, yy, tt, zz, j, extrems))
def get_graphic_h(solver, time = 0, tsteps = 40):
 h, e = [], []
 for N in range(10, 100, 10):
 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
def get_graphic_tau(solver):
 tau = []
 e = []
 for K in range(4, 101, 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
def plot(nx, ny, k, t, order):
 schema = Schema(T=t, order2nd=order)
 xx, yy, tt, zz = schema(Nx=nx, Ny=ny, K=k)
 extrems = search_minmax(zz)
 plots = []
 for j in range(len(tt)):
 plots.append(plot_by_time(xx, yy, tt, zz, j, extrems))
def get_graphic_h(solver, time = 0, tsteps = 40):
 h, e = [], []
 for N in range(10, 100, 10):
 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
def get_graphic_tau(solver):
 tau = []
```

```
 e = []
 for K in range(4, 101, 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
def error_tau():
 plt.figure(figsize = (10, 7))
 plt.title("Зависимость погрешности от длины шага по времени")
 first = Schema(T=1, order2nd=False) # метод дробных шагов
 second = Schema(T=1, order2nd=True) # метод переменных направлений
 tau1, e1 = get_graphic_tau(first)
 tau2, e2 = get_graphic_tau(second)
 plt.plot(tau1, e1, label="Метод дробных шагов")
 plt.plot(tau2, e2, label="Метод переменных направлений")
 plt.xlabel("$tau$")
 plt.ylabel("$\epsilon$")
 plt.legend()
 plt.grid()
 plt.show()
def error_h():
 TSTEPS = 100
 time = 50
 plt.figure(figsize = (10, 7))
 plt.title("Зависимость погрешности от длины шага при t = " + str(time /
TSTEPS))
 first = Schema(T=1, order2nd=False) # метод дробных шагов
 second = Schema(T=1, order2nd=True) # метод переменных направлений
 h1, e1 = get_graphic_h(first, time, TSTEPS)
 h2, e2 = get_graphic_h(second, time, TSTEPS)
 plt.plot(h1, e1, label="Метод дробных шагов")
 plt.plot(h2, e2, label="Метод переменных направлений")
 plt.xlabel("$h_x$")
 plt.ylabel("$\epsilon$")
 plt.legend()
 plt.grid()
 plt.show()
```

```
plot(nx=40, ny=40, k=3, t=1, order=False)
error_h()
error_tau()
```

Решение

Метод переменных направлений

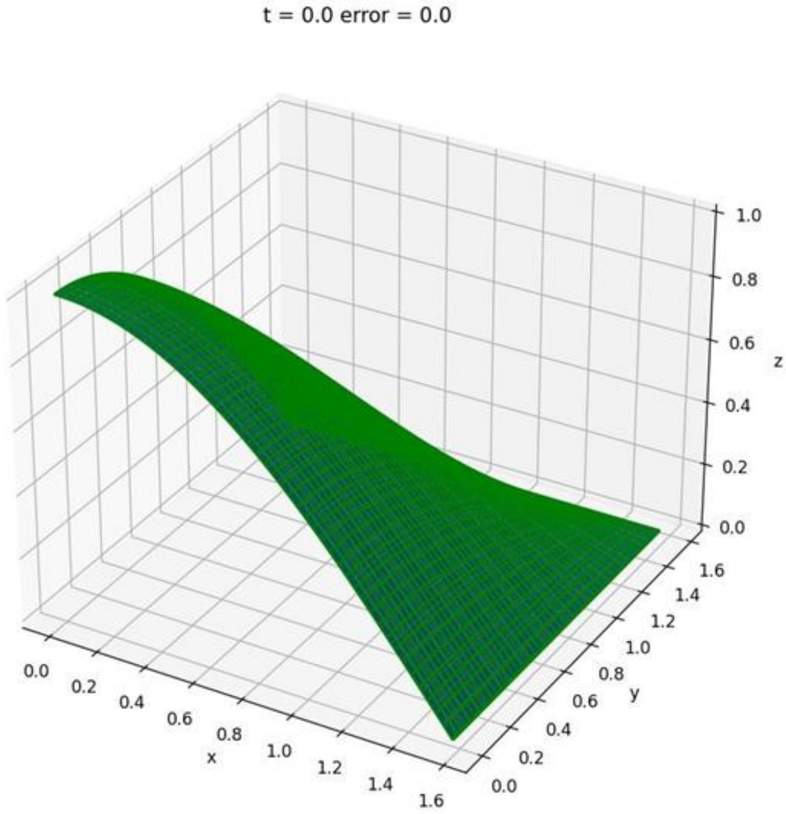

Рисунок 4.1. (Решение задачи методом переменных направлений)

Метод дробных шагов

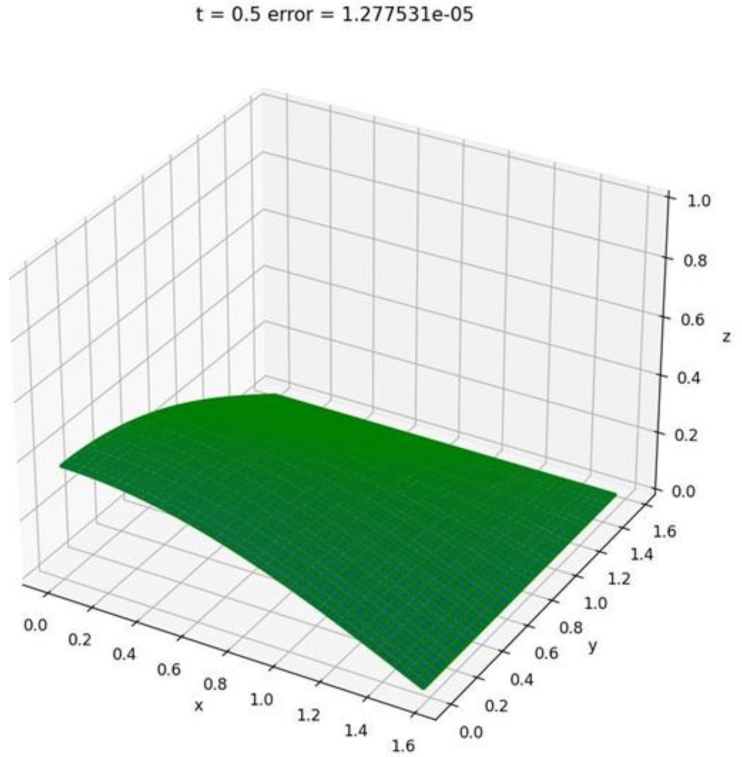

Рисунок 4.2. (Решение задачи методом дробных шагов, t = 0.5)

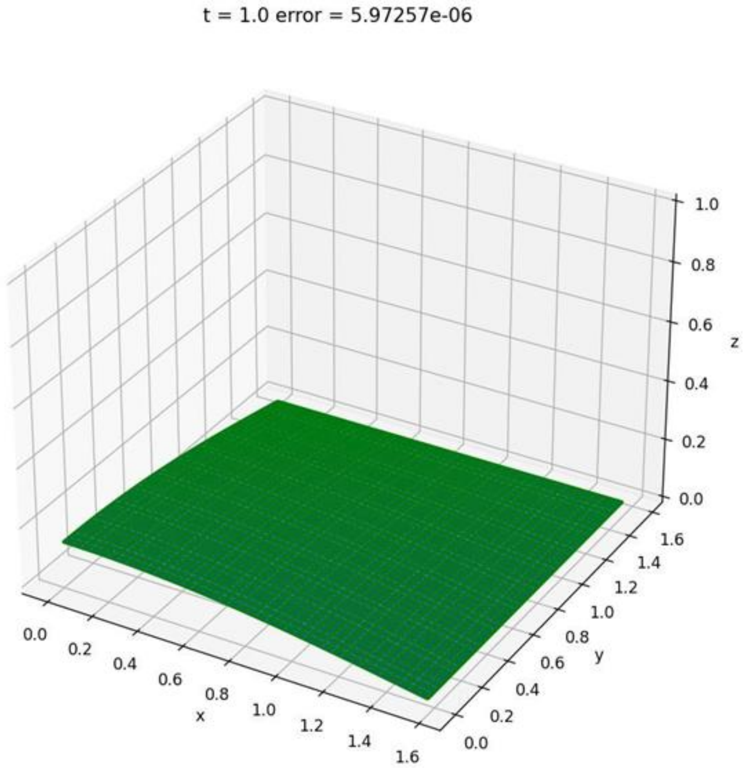

Рисунок 4.3. (Решение задачи методом дробных шагов, t = 1)

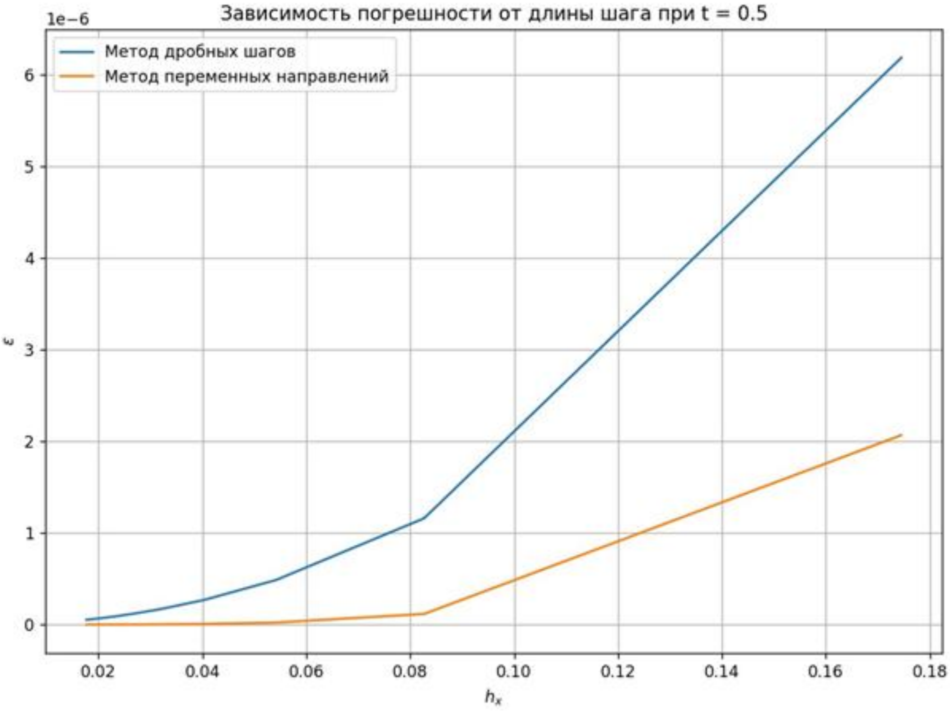

Рисунок 4.4. (График зависимости погрешности от длины шага при t = 0.5)

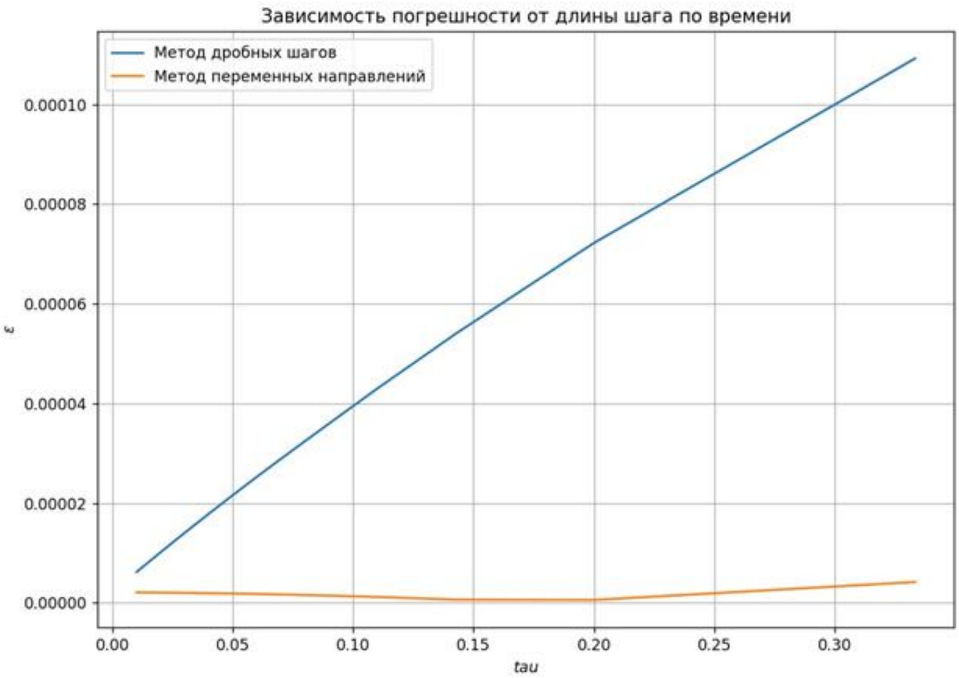

Рисунок 4.5. (График зависимости погрешности от длины шага при t = 1)