

Московский авиационный институт  
(Национальный исследовательский университет)  
Факультет прикладной математики и физики  
Кафедра вычислительной математики и программирования

**Отчет по лабораторным работам**  
по курсу «Численные методы»  
Вариант 2

Выполнил: Примаченко А.А.

Группа: М8О-408Б-20

Проверил: проф. Пивоваров Д.Е.

Дата:

Оценка:

Москва, 2023

# ЛАБОРАТОРНАЯ РАБОТА №2. РЕШЕНИЕ ДИФФЕРЕНЦИАЛЬНЫХ УРАВНЕНИЙ ГИПЕРБОЛИЧЕСКОГО ТИПА

## 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  $\tau, h$ .

## Вариант 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:  $U(x, t) = \sin(x - at) + \cos(x + at)$

## Ход решения

Действуем способом, аналогичным тому, что применялся в предыдущей лабораторной работе: Задаем пространственно-временную сетку и аппроксимируем производные в уравнении. Получаем явную конечно-разностную схему:

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

и неявную конечно-разностную схему:

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

Явная схема условно устойчива с условием  $\sigma = \frac{a^2 \tau^2}{h^2} < 1$ .

Для обоих методов для вычисления  $u_j^{k+1}$  необходимо знать  $u_j^{k-1}$  и  $u_j^k$ .  $u_j^0 = \sin x_j + \cos x_j$ . Для нахождения  $u_j^1$  воспользуемся аппроксимацией второго порядка начального условия:

$$\frac{u_j^1 - u_j^0}{\tau} = -a(\sin x_j + \cos x_j),$$

откуда  $u_j^1 = \sin x_j + \cos x_j - a\tau(\sin x_j + \cos x_j)$ . Разложим полученный результат в ряд Тейлора. В итоге получаем:

$$u_j^1 = \left(1 - a\tau - \frac{a^2 \tau^2}{2}\right)(\sin x_j + \cos x_j).$$

В результате для явной схемы получаем формулу:

$$u_j^{k+1} = \sigma(u_{j+1}^k - 2u_j^k + u_{j-1}^k) + 2u_j^k - u_j^{k-1}$$

Для неявной схемы имеем СЛАУ, которая опять же решается прогонкой, так как полученная матрица является трёхдиагональной. Для этой СЛАУ:  $a_j = \sigma, b_j = -(2\sigma + 1), c_j = \sigma, d_j = -2u_j^k + u_j^{k-1}$ . Причем с учетом аппроксимации первым порядком граничных условий:

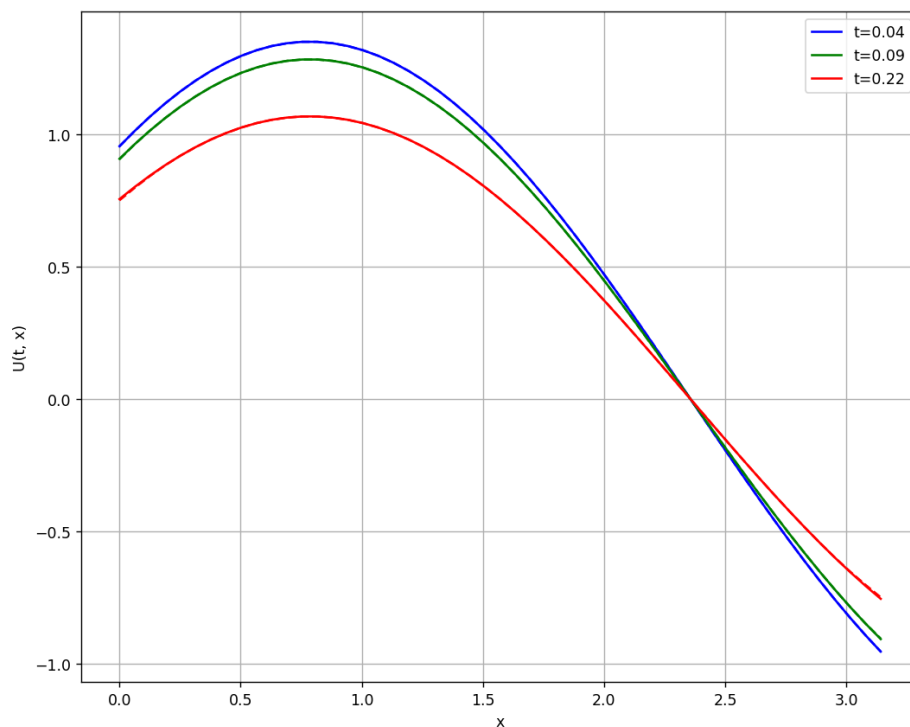
$$\begin{cases} \frac{u_1^{k+1} - u_0^{k+1}}{h} - u_0^{k+1} = 0 \\ \frac{u_N^{k+1} - u_{N-1}^{k+1}}{h} - u_N^{k+1} = 0 \end{cases} \Rightarrow \begin{cases} u_0^{k+1} = \frac{u_1^{k+1}}{h+1} \\ u_N^{k+1} = \frac{u_{N-1}^{k+1}}{1-h} \end{cases}$$

для  $j = 1$  и  $j = N - 1$  имеем  $b_1 = -(2\sigma + 1) + \frac{\sigma}{h+1}$  и  $b_{N-1} = -(2\sigma + 1) + \frac{\sigma}{1-h}$ .

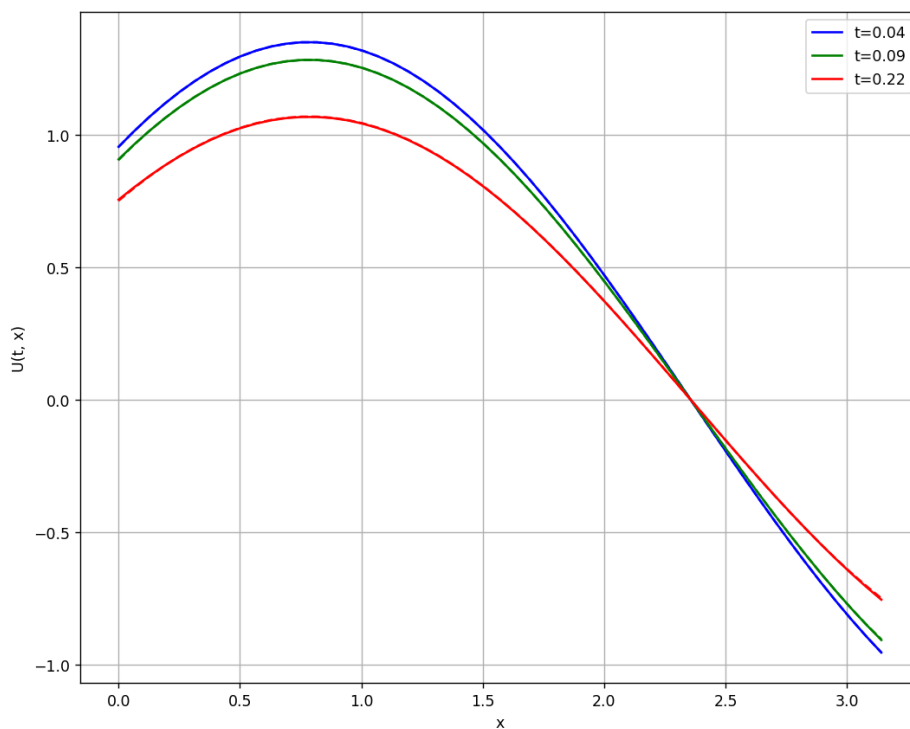
## Результат работы программы

Программа считает задачу с одним заранее заданным  $\sigma = \frac{1}{2}$ . На основании этого значения она рассчитывает шаги по времени и координате таким образом, чтобы выполнялось условие устойчивости для явного метода. На графики она выводит значение функции при трёх различных фиксированных  $t$ .

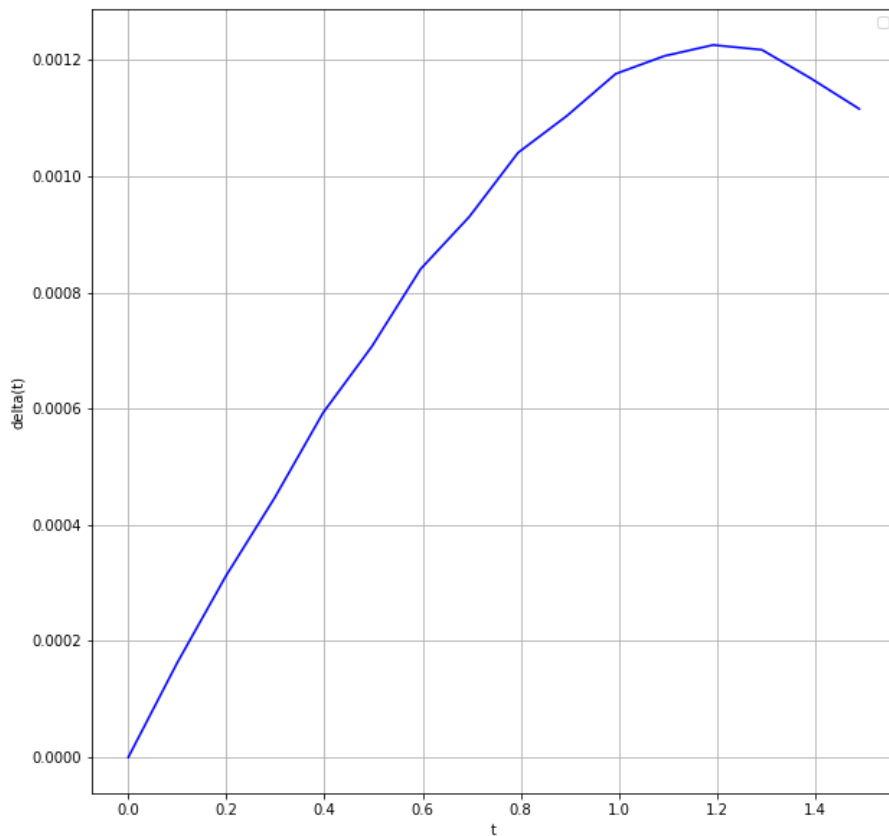
Сплошной линией на графике изображено точное решение, пунктирной – рассчитанное численно. Для построения графика погрешностей методов перебирается 3 различных разбиений сетки по координате  $x$ .



Явная схема



Неявная схема



## Погрешности

### Вывод

В результате выполнения данной лабораторной работы, было выяснено, что явная, как и неявная схемы практически в равной степени дают достаточно точный результат. При увеличении шага по координате  $x$  закономерно росла и погрешность. Однако стоит отметить, что неявная схема сильнее реагирует на изменение шага – если он увеличивается, погрешность этой схемы увеличится заметно больше, чем погрешность явной схемы.

### Код программы

```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [10, 10]

def U(x, t, a):
    return np.sin(x-a*t) + np.cos(x+a*t)

def psi(x):
    return np.sin(x) + np.cos(x)

def d_psi(x, a, tau):
```

```

    return (1 - a*tau - a**2*tau**2/2)*(np.sin(x) + np.cos(x))

u_0=0; u_l=0

def Check(A):
    if np.shape(A)[0] != np.shape(A)[1]:
        return False
    n = np.shape(A)[0]
    for i in range(n):
        sum = 0
        for j in range(n):
            if i != j:
                sum += abs(A[i][j])
        if abs(A[i][i]) < sum:
            return False
    return True

def solve(a, b):
    if (Check(a)):
        p = np.zeros(len(b))
        q = np.zeros(len(b))
        p[0] = -a[0][1] / a[0][0]
        q[0] = b[0] / a[0][0]
        for i in range(1, len(p) - 1):
            p[i] = -a[i][i + 1] / (a[i][i] + a[i][i - 1] * p[i - 1])
            q[i] = (b[i] - a[i][i - 1] * q[i - 1]) / (a[i][i] + a[i][i - 1] *
p[i - 1])
            i = len(a) - 1
            p[-1] = 0
            q[-1] = (b[-1] - a[-1][-2] * q[-2]) / (a[-1][-1] + a[-1][-2] * p[-2])
            x = np.zeros(len(b))
            x[-1] = q[-1]
            for i in reversed(range(len(b) - 1)):
                x[i] = p[i] * x[i + 1] + q[i]
            return x

def error(sigma, l, a, T, U):
    N_array = [10, 20, 40]
    size = np.size(N_array)
    h_array = np.zeros(size)
    tau_array = np.zeros(size)
    K_array = np.zeros(size)
    errors1 = np.zeros(size)

```

```

errors2 = np.zeros(size)
for i in range(0, size):
    h_array[i] = l/N_array[i]
    tau_array[i] = np.sqrt(sigma * h_array[i]**2 / a)
    K_array[i] = int(round(T/tau_array[i]))
    x_array = np.arange(0, 1+h_array[i], h_array[i])
    u1 = Explicit_Method(N_array[i], int(K_array[i]), sigma, tau_array[i],
a, h_array[i])
    u2 = Implicit_Method(N_array[i], int(K_array[i]), sigma, tau_array[i],
a, h_array[i])
    t = tau_array[i] * K_array[i]/2
    if (np.size(x_array)!=N_array[i]+1):
        x_array = x_array[:N_array[i]+1]
    u_correct = U(x_array, t, a)
    u1_calculated = u1[int(K_array[i]/2)]
    u2_calculated = u2[int(K_array[i]/2)]
    errors1[i] = np.amax(np.abs(u_correct - u1_calculated))
    errors2[i] = np.amax(np.abs(u_correct - u2_calculated))
return N_array, errors1, errors2

def show_solution(h, tau, K, l, u, U, a):
    x_array = np.arange(0, 1 + h, h)
    fig, ax = plt.subplots()
    t = [int(K * 0.05), int(K * 0.1), int(K * 0.25)]
    colors = ['blue', 'green', 'red']
    for i in range(len(t)):
        u_correct = U(x_array, t[i] * tau, a)
        u_calculated = u[t[i]]
        plt.plot(x_array, u_correct, color=colors[i], label='t=%s' % round(t[i]
* tau, 2))
        plt.plot(x_array, u_calculated, color=colors[i], linestyle='--')

    ax.set_xlabel('x')
    ax.set_ylabel('U(t, x)')
    plt.grid()
    ax.legend()
    plt.show()

def show_errors(sigma, l, a, T, U):
    N_array, errors1, errors2 = error(sigma, l, a, T, U)
    colors = ['blue', 'green']
    deltaX = np.zeros(np.size(N_array))
    for i in range(np.size(N_array)):

```

```

        deltaX[i] = 1/N_array[i]
    fig, ax = plt.subplots()
    plt.plot(deltaX, errors1, color=colors[0], label='Явный метод')
    plt.plot(deltaX, errors2, color=colors[1], label='Неявный метод')
    ax.set_xlabel('delta X')
    ax.set_ylabel('Epsilon')
    plt.grid()
    ax.legend()
    plt.show()

def Explicit_Method(N, K, sigma, tau, a, h):
    # Проверка на устойчивость
    if (sigma > 1):
        raise Exception("Измените параметры сетки")

    u = np.zeros((K + 1, N + 1))
    for i in range(0, N + 1):
        u[0][i] = psi(i * h)
        u[1][i] = d_psi(i * h, a, tau)

    for k in range(2, K+1):
        for j in range(1, N):
            u[k][j] = sigma*u[k-1][j-1] + 2*(1-sigma)*u[k-1][j] + sigma*u[k-1][j+1] - u[k-2][j]
        u[k][0] = u[k][1] / (h+1)
        u[k][N] = u[k][N-1] / (1-h)
    return u

def Implicit_Method(N, K, sigma, tau, a, h):
    a_j = sigma
    b_j = -(1 + 2*sigma)
    c_j = sigma
    u = np.zeros((K + 1, N + 1))
    for i in range(0, N + 1):
        u[0][i] = psi(i * h)
        u[1][i] = d_psi(i * h, a, tau)

    # Заполняем верхние слои по неявной конечно-разностной схеме
    for k in range(2, K + 1):
        # Создаем матрицу и столбец для решения СЛАУ
        matrix = np.zeros((N - 1, N - 1))
        d = np.zeros(N - 1)

```



```

# Первая строка
matrix[0][0] = b_j + sigma/(h+1)
matrix[0][1] = c_j
d[0] = -2*u[k-1][1] + u[k-2][1]

# Строки с первой по N-2
for j in range(1, N-2):
    matrix[j][j-1] = a_j
    matrix[j][j] = b_j
    matrix[j][j+1] = c_j
    d[j] = -2*u[k-1][j+1] + u[k-2][j+1]

# Последняя строка
matrix[N-2][N-3] = a_j
matrix[N-2][N-2] = b_j + sigma/(1-h)
d[N-2] = -2*u[k-1][N-1] + u[k-2][N-1]
# Решем СЛАУ методом прогонки
ans = solve(matrix, d)
u[k][1:N] = ans
u[k][0] = u[k][1] / (h+1)
u[k][N] = u[k][N-1] / (1-h)
return u

def main():
    a = 1;
    T = 1;
    N = 50;
    sigma = 0.5;
    l = np.pi
    h = l / N
    tau = np.sqrt(sigma * h ** 2 / a)
    K = int(round(T / tau))
    u0 = np.zeros((K+1, N+1))
    for k in range(0, K+1):
        for j in range(0, N+1):
            u0[k][j] = U(j*h, k*tau, a)
    u1 = Explicit_Method(N, K, sigma, tau, a, h)
    show_solution(h, tau, K, l, u1, U, a)
    u2 = Implicit_Method(N, K, sigma, tau, a, h)
    show_solution(h, tau, K, l, u2, U, a)
    show_errors(sigma, l, a, T, U)

main()

```