Лабораторная работа 5

Начально-краевые задачи для дифференциального уравнения параболического типа

Студентка: Айрапетова Е. А.

Группа: М8О-406Б-19

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).

Вариант 3.

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

$a > 0$

$u(0, t) = e^{-at}$

$u(\pi, t) = -e^{-at}$

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

Аналитическое решение: $U(x, t) = e^{-at}cos(x)$

In [ ]:

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

In [ ]:

```
a = 1

x_begin = 0
x_end = math.pi

t_begin = 0
t_end = 5

h = 0.01
sigma = 0.45
```

Начальные условия:

In [ ]:

```
# boundary conditions
def phi_0(t, a=a):
    return math.exp(-a * t)

def phi_1(t, a=a):
    return -math.exp(-a * t)

# initial condition
def psi(x):
    return math.cos(x)

def solution(x, t, a=a):
    return math.exp(-a * t) * math.cos(x)
```

Аналитическое решение

Подготовим ответ, полученный аналитическим способом. С ним будем сравнивать численные методы

In [ ]:

```
def get_analytical_solution(
    x_range, # (x_begin, x_end)
    t_range, # (t_begin, t_end)
    h=h, # len of cell by x
    sigma=sigma, # coefficient sigma
    a=a, # coefficient a
):
    """
    Get analytical solution of parabolic DE
    Returns matrix U with values of function
    """
    tau = sigma * h**2 / a # len of cell by t
    x = np.arange(*x_range, h)
    t = np.arange(*t_range, tau)

    res = np.zeros((len(t), len(x)))
    for idx in range(len(x)):
        for idt in range(len(t)):
            res[idt][idx] = solution(x[idx], t[idt], a)

    return res
```

In [ ]:

```
analytical_solution = get_analytical_solution(
    x_range=(x_begin, x_end),
    t_range=(t_begin, t_end),
    h=h,
    sigma=sigma,
    a=a,
)
```

Будем складывать все решения в словарь, чтобы потом удобнее было строить графики

In [ ]:

```
solutions = dict()
solutions["analytical solution"] = analytical_solution
```

Функция для вычисления погрешности - максимального модуля ошибки

In [ ]:

```
def max_abs_error(A, B):
    """
    Calculate max absolute error of elements of matrices A and B
    """
    assert A.shape == B.shape
    return abs(A - B).max()
```

И среднего модуля ошибки:

In [ ]:

```
def mean_abs_error(A, B):
    """
    Calculate mean absolute error of elements of matrices A and B
    """
    assert A.shape == B.shape
    return abs(A - B).mean()
```

Функция для построения результата - функций U(x), полученных разыми методами, при заданном времени t.

In [ ]:

```
def plot_results(
    solutions, # dict: solutions[method name] = solution
    time, # moment of time
    x_range, # (x_begin, x_end)
    t_range, # (t_bein, t_end)
    h, # len of cell by x
    sigma, # coefficient sigma
):
    tau = sigma * h**2 / a # len of cell by t
    x = np.arange(*x_range, h)
    times = np.arange(*t_range, tau)
    cur_t_id = abs(times - time).argmin()

    plt.figure(figsize=(15, 9))
    for method_name, solution in solutions.items():
        plt.plot(x, solution[cur_t_id], label=method_name)

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

Зависимость погрешности от времени

In [ ]:

```
def plot_errors_from_time(
    solutions, # dict: solutions[method name] = solution
    analytical_solution_name, # for comparing
    t_range, # (t_bein, t_end)
    h, # len of cell by x
    sigma, # coefficient sigma
):
    """
    Plot max_abs_error = f(time)
    """
    tau = sigma * h**2 / a # len of cell by t
    t = np.arange(*t_range, tau)

    plt.figure(figsize=(15, 9))
    for method_name, solution in solutions.items():
        if method_name == analytical_solution_name:
            continue
        max_abs_errors = np.array([
            max_abs_error(solution[i], solutions[analytical_solution_name][i])
            for i in range(len(t))
        ])
        plt.plot(t, max_abs_errors, label=method_name)

    plt.xlabel('time')
    plt.ylabel('Max abs error')

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

Явная конечно-разностная схема

В исходном уравнении перейдем от производных к их численным приближениям. Вторую производную будем аппроксимировать по значениям нижнего временного слоя.

Получим рекуррентное соотношение:

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

где $\sigma = \frac{a\tau}{h^2}$

Значения u в нижнем временном ряду нам известны из начальных условий. Далее можем в цикле проходиться по сетке и рекурретно считать значения в ней.

In [ ]:

```
def explicit_finite_difference_method(
    x_range, # (x_begin, x_end)
    t_range, # (t_begin, t_end)
    h, # len of cell by x
    sigma, # coefficient sigma
    a=a, # coefficient a
    phi_0=phi_0, # boundary condition 0
    phi_1=phi_1, # boundary condition 1
    psi=psi, # initial condition,
):
    """
    Solves parabolic DE using explicit schema of finite difference method.
    Returns matrix U with values of function
    """
    tau = sigma * h**2 / a # len of cell by t
    x = np.arange(*x_range, h)
    t = np.arange(*t_range, tau)

    res = np.zeros((len(t), len(x)))
    # row 0 -> use initial condition
    for col_id in range(len(x)):
        res[0][col_id] = psi(x[col_id])

    for row_id in range(1, len(t)):
        # col 0 -> use boundary condition 0
        res[row_id][0] = phi_0(t[row_id], a)
        # cols 1..n-1 -> use explicit schema
        for col_id in range(1, len(x)-1):
            res[row_id][col_id] = (
                sigma * res[row_id-1][col_id-1]
                + (1 - 2*sigma) * res[row_id-1][col_id]
                + sigma * res[row_id-1][col_id+1]
            )
        # col n -> use boundary condition 1
        res[row_id][-1] = phi_1(t[row_id], a)
    return res
```

In [ ]:

```
explicit_solution = explicit_finite_difference_method(
    x_range=(x_begin, x_end),
    t_range=(t_begin, t_end),
    h=h,
    sigma=sigma,
)
```

In [ ]:

```
solutions["explicit schema"] = explicit_solution
```

Погрешность в сравнении с аналитическим решением:

In [ ]:

```
print(f'max abs error = {max_abs_error(explicit_solution, analytical_solution)}')
print(f'mean abs error = {mean_abs_error(explicit_solution, analytical_solution)}')
```

```
max abs error = 1.894904644861306e-06
mean abs error = 3.4062064884757373e-07
```

У этого метода есть особенность - он сходится только при $\sigma < \frac{1}{2}$, иначе погрешность будет большой

Неявная конечно-разностная схема

В исходном уравнении перейдем от производных к их численным приближениям. Вторую производную будем аппроксимировать по значениям верхнего временного слоя.

Чтобы получить значения u в одном временном ряду, необходимо решить систему уравнений:

$$\begin{cases} b_1 u_1^{k+1} + c_1 u_2^{k+1} = d_1, \quad j = 1, \\ a_j u_{j-1}^{k+1} + b_j u_j^{k+1} + c_j u_{j+1}^{k+1} = d_j, \quad j = 2 \dots N-2, \\ a_{N-1} u_{N-2}^{k+1} + b_{N-1} u_{N-1}^{k+1} = d_{N-1}, \quad j = N-1. \end{cases}$$

$a_j = c_j = \sigma$

$b_j = -(1 + 2\sigma)$

$d_j = -u_j^k, \quad j = 2 \dots N-2$

$d_1 = -(u_1^k + \sigma\phi_0(t^{k+1}))$

$d_{N-1} = -(u_{N-1}^k + \sigma\phi_1(t^{k+1}))$

Это трехдиагональная СЛАУ, которую можно решить методом прогонки.

In [ ]:

```
# stolen from lab 1-2
def tridiagonal_solve(A, b):
    """
    Solves Ax=b, where A - tridiagonal matrix
    Returns x
    """
    n = len(A)
    # Step 1. Forward
    v = [0 for _ in range(n)]
    u = [0 for _ in range(n)]
    v[0] = A[0][1] / -A[0][0]
    u[0] = b[0] / A[0][0]
    for i in range(1, n-1):
        v[i] = A[i][i+1] / (-A[i][i] - A[i][i-1] * v[i-1])
        u[i] = (A[i][i-1] * u[i-1] - b[i]) / (-A[i][i] - A[i][i-1] * v[i-1])
    v[n-1] = 0
    u[n-1] = (A[n-1][n-2] * u[n-2] - b[n-1]) / (-A[n-1][n-1] - A[n-1][n-2] * v[n-2])

    # Step 2. Backward
    x = [0 for _ in range(n)]
    x[n-1] = u[n-1]
    for i in range(n-1, 0, -1):
        x[i-1] = v[i-1] * x[i] + u[i-1]
    return np.array(x)
```

In [ ]:

```
def implicit_finite_difference_method(
    x_range, # (x_begin, x_end)
    t_range, # (t_begin, t_end)
    h, # len of cell by x
    sigma, # coefficient sigma
    a=a, # coefficient a
    phi_0=phi_0, # boundary condition 0
    phi_1=phi_1, # boundary condition 1
    psi=psi, # initial condition,
):
    """
    Solves parabolic DE using implicit schema of finite difference method.
    Returns matrix U with values of function
    """
    tau = sigma * h**2 / a # len of cell by t
    x = np.arange(*x_range, h)
    t = np.arange(*t_range, tau)
    res = np.zeros((len(t), len(x)))

    # row 0 -> use initial condition
    for col_id in range(len(x)):
        res[0][col_id] = psi(x[col_id])

    for row_id in range(1, len(t)):
        A = np.zeros((len(x)-2, len(x)-2)) # first and last elements will be counted with boundary conditions

        # create system of equations for implicit schema
        A[0][0] = -(1 + 2*sigma)
        A[0][1] = sigma
        for i in range(1, len(A) - 1):
            A[i][i-1] = sigma
            A[i][i] = -(1 + 2*sigma)
            A[i][i+1] = sigma
        A[-1][-2] = sigma
        A[-1][-1] = -(1 + 2*sigma)

        # vector b is previous line except first and last elements
        b = -res[row_id-1][1:-1]
        # apply boundary conditions
        b[0] -= sigma * phi_0(t[row_id])
        b[-1] -= sigma * phi_1(t[row_id])

        res[row_id][0] = phi_0(t[row_id])
        res[row_id][-1] = phi_1(t[row_id])
        res[row_id][1:-1] = tridiagonal_solve(A, b)

    return res
```

In [ ]:

```
implicit_solution = implicit_finite_difference_method(
    x_range=(x_begin, x_end),
    t_range=(t_begin, t_end),
    h=h,
    sigma=sigma,
)
```

In [ ]:

```
solutions["implicit schema"] = implicit_solution
```

Погрешность в сравнении с аналитическим решением:

In [ ]:

```
print(f'max abs error = {max_abs_error(implicit_solution, analytical_solution)}')
print(f'mean abs error = {mean_abs_error(implicit_solution, analytical_solution)}')
```

```
max abs error = 4.562762782045482e-06
mean abs error = 8.974405002195539e-07
```

Неявная схема абсолютна устойчива, поэтому при ее использовании всегда получается адекватный результат независимо от параметров сетки. Но с другой стороны этот метод вычислительно более сложный

Схема Кранка-Николсона

Если визуализировать решения, полученные явной и неявной схемой, а также аналитическое решение, то можно заметить, что аналитическое решение часто лежит между двумя численными. Поэтому стоит попробовать использовать выпуклую комбинацию аппроксимаций второй производной (по верхнему и нижнему временному ряду).

Выпуклая комбинация - линейная комбинация, при которой коэффициенты неотрицательные и в сумме дают единицу. Будем использовать коэффициенты θ и $1-\theta$.

При $\theta = \frac{1}{2}$ имеем схему Кранка-Николсона.

Немного скорректируем систему уравнений из прошлой схемы

$$\begin{cases} b_1 u_1^{k+1} + c_1 u_2^{k+1} = d_1, \quad j = 1, \\ a_j u_{j-1}^{k+1} + b_j u_j^{k+1} + c_j u_{j+1}^{k+1} = d_j, \quad j = 2 \dots N-2, \\ a_{N-1} u_{N-2}^{k+1} + b_{N-1} u_{N-1}^{k+1} = d_{N-1}, \quad j = N-1. \end{cases}$$

$a_j = c_j = \sigma\theta$

$b_j = -(1 + 2\sigma\theta)$

$d_j = -(u_j^k + (1-\theta)\sigma(u_{j-1}^k - 2u_j^k + u_{j+1}^k)), \quad j = 2 \dots N-2$

$d_1 = -(u_1^k + \sigma\phi_0(t^{k+1}))$

$d_{N-1} = -(u_{N-1}^k + \sigma\phi_1(t^{k+1}))$

Это трехдиагональная СЛАУ, которую можно решить методом прогонки.

In [ ]:

```
def crank_nicolson_method(
    x_range, # (x_begin, x_end)
    t_range, # (t_begin, t_end)
    h, # len of cell by x
    sigma, # coefficient sigma
    a=a, # coefficient a
    phi_0=phi_0, # boundary condition 0
    phi_1=phi_1, # boundary condition 1
    psi=psi, # initial condition,
    theta=0.5, # coefficient theta for combination
):
    """
    Solves parabolic DE using Crank-Nicolson schema.
    Returns matrix U with values of function
    """
    tau = sigma * h**2 / a # len of cell by t
    x = np.arange(*x_range, h)
    t = np.arange(*t_range, tau)
    res = np.zeros((len(t), len(x)))

    # row 0 -> use initial condition
    for col_id in range(len(x)):
        res[0][col_id] = psi(x[col_id])

    for row_id in range(1, len(t)):
        A = np.zeros((len(x)-2, len(x)-2)) # first and last elements will be counted with boundary conditions

        # create system of equations for implicit schema
        A[0][0] = -(1 + 2*sigma*theta)
        A[0][1] = sigma * theta
        for i in range(1, len(A) - 1):
            A[i][i-1] = sigma * theta
            A[i][i] = -(1 + 2*sigma*theta)
            A[i][i+1] = sigma * theta
        A[-1][-2] = sigma * theta
        A[-1][-1] = -(1 + 2*sigma*theta)

        # vector b is previous line except first and last elements
        b = np.array([-(
            res[row_id-1][i] +
            (1-theta) * sigma *
                (res[row_id-1][i-1] - 2*res[row_id-1][i] + res[row_id-1][i+1])
        ) for i in range(1, len(res[row_id-1])-1)])
        # apply boundary conditions
        b[0] -= sigma * theta * phi_0(t[row_id])
        b[-1] -= sigma * theta * phi_1(t[row_id])

        res[row_id][0] = phi_0(t[row_id])
        res[row_id][-1] = phi_1(t[row_id])
        res[row_id][1:-1] = tridiagonal_solve(A, b)

    return res
```

In [ ]:

```
crank_nicolson_solution = crank_nicolson_method(
    x_range=(x_begin, x_end),
    t_range=(t_begin, t_end),
    h=h,
    sigma=sigma,
)
```

In [ ]:

```
solutions["crank-nicolson schema"] = crank_nicolson_solution
```

Погрешность в сравнении с аналитическим решением:

In [ ]:

```
print(f'max abs error = {max_abs_error(crank_nicolson_solution, analytical_solution)}')
print(f'mean abs error = {mean_abs_error(crank_nicolson_solution, analytical_solution)}')
```

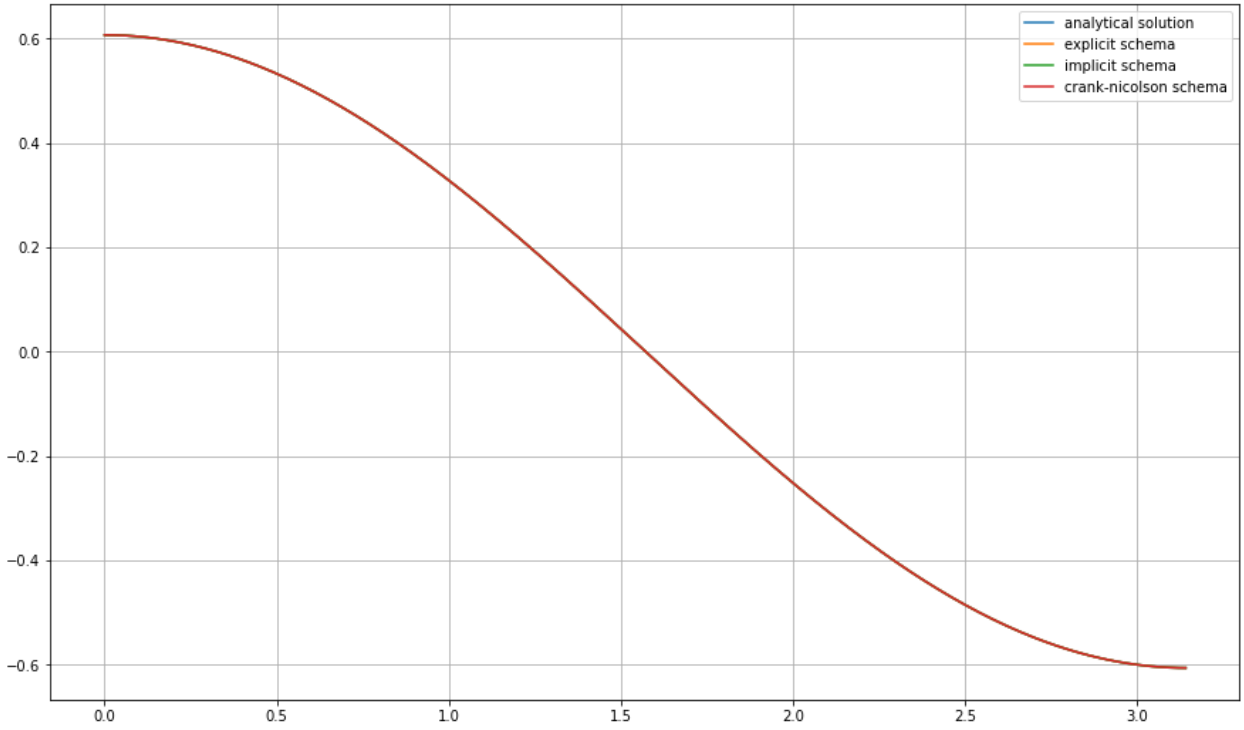
```
max abs error = 1.6059529972523023e-06
mean abs error = 2.9911393897420475e-07
```

Визуализация

Посмотрим на полученные функции при некотором фиксированном моменте времени

In [ ]:

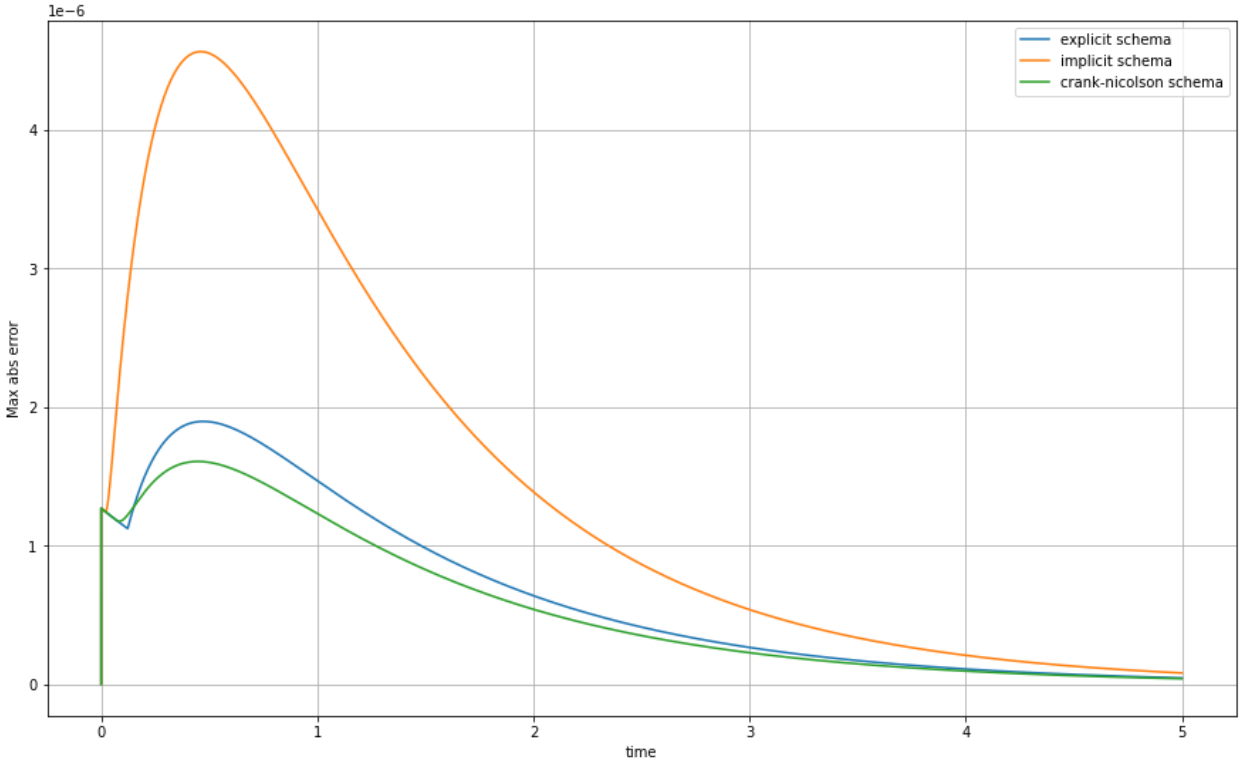
```
plot_results(
    solutions=solutions,
    time=0.5,
    x_range=(x_begin, x_end),
    t_range=(t_begin, t_end),
    h=h,
    sigma=sigma,
)
```



Посмотрим, как меняется зависимость погрешности с течением времени

In [ ]:

```
plot_errors_from_time(
    solutions=solutions,
    analytical_solution_name="analytical solution",
    t_range=(t_begin, t_end),
    h=h,
    sigma=sigma,
)
```



Вывод

В данной работе я научилась решать начально-краевые задачи для ДУ параболического типа тремя способами:

- с помощью явной конечно-разностной схемы
- с помощью неявной конечно-разностной схемы
- с помощью схемы Кранка-Николсона

С помощью каждого метода получилось решить заданное ДУ с приемлемой точностью.

В ходе работы я выявила плюсы и минусы изученных алгоритмов.

Явная конечно-разностная схема легко считается, но она не всегда устойчива и, соответственно, не всегда гарантирует адекватный результат.

Неявная схема абсолютно устойчива, но она требует больших вычислительных затрат - приходится решать много СЛАУ.

Схема Кранка-Николсона "комбинирует" предыдущие схемы, поэтому имеет наименьшую погрешность. Но при этом она по-прежнему использует сложные вычисления.

In [ ]: