



МИНИСТЕРСТВО НАУКИ
И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Новосибирский государственный технический университет»



Новосибирский
государственный
технический университет
НЭТИ

Кафедра прикладной математики

Практическая работа №2
по дисциплине «Уравнения математической физики»

РЕШЕНИЕ НЕЛИНЕЙНЫХ НАЧАЛЬНО-КРАЕВЫХ ЗАДАЧ



Факультет: ПМИ
Группа: ПМ-71
Студенты: Востриков Вячеслав,
Аникина Полина
Вариант: 11
Преподаватели: Задорожный Александр Геннадьевич,
Патрушев Илья Игоревич

Новосибирск
2020

1. Цель работы

Разработать программу решения нелинейной одномерной краевой задачи методом конечных элементов. Провести сравнение метода простой итерации и метода Ньютона для решения данной задачи.

2. Вариант задания

Уравнение: $-\operatorname{div}(\lambda \operatorname{grad} u) + \sigma \left(\frac{\delta u}{\delta x} \right) \frac{\delta u}{\delta t} = f$. Базисные функции – линейные.

3. Теория

Будем рассматривать параболическое одномерное уравнение с начальными условиями:

$$\begin{aligned} -\frac{\partial}{\partial x} \left(\lambda(x, t) \frac{\partial u}{\partial x} \right) + \sigma \left(\frac{\partial u}{\partial x} \right) \frac{\partial u}{\partial t} &= f(x, t), x \in [a, b] \\ u|_{t=0} &= u_0(x) \end{aligned}$$

и краевыми условиями всех типов.

Данное уравнение имеет зависимость по времени t , что означает, что в данном варианте необходимо реализовать не только конечноэлементную аппроксимацию, но и аппроксимацию по времени:

1. Аппроксимация по времени:

Будем использовать следующую неявную схему:

$$\frac{\partial u}{\partial t} = \frac{u^{n+1} - u^n}{\tau_n}.$$

Исходное уравнение можно записать в виде:

$$-\frac{\partial}{\partial x} \left(\lambda(x, t_{n+1}) \frac{\partial u^{n+1}}{\partial x} \right) + \frac{1}{\tau_n} \sigma \left(\frac{\partial u^{n+1}}{\partial x} \right) u^{n+1} = f(x, t_{n+1}) + \frac{1}{\tau_n} \sigma \left(\frac{\partial u^{n+1}}{\partial x} \right) u^n$$

Введем обозначения следующие обозначения:

$$\begin{aligned} \lambda_n(x) &= \lambda(x, t_{n+1}), \\ \gamma_n \left(\frac{\partial u}{\partial x} \right) &= \frac{1}{\tau_n} \sigma \left(\frac{\partial u^{n+1}}{\partial x} \right), \\ \tilde{f}_n(x) &= f(x, t_{n+1}) + \frac{1}{\tau_n} \sigma \left(\frac{\partial u^{n+1}}{\partial x} \right) u^n. \end{aligned}$$

Таким образом, получаем эллиптическое уравнение из исходного параболического:

$$-\frac{\partial}{\partial x} \left(\lambda_n(x) \frac{\partial u^{n+1}}{\partial x} \right) + \gamma_n \left(\frac{\partial u^{n+1}}{\partial x} \right) u^{n+1} = \tilde{f}_n(x)$$

2. Конечноэлементная аппроксимация

Рассмотрим гильбертово пространство $H = L_2[a, b]$ со скалярным произведением

$$(f, g) = \int_a^b f(x)g(x)dx$$

$$\text{и нормой } \|f\| = \sqrt{(f, f)}.$$

Умножим скалярно правую и левую часть уравнения из предыдущего пункта на функцию $v \in H_0$, где H_0 - множество функций, удовлетворяющих однородным первым краевым условиям:

$$\int_a^b -\frac{\partial}{\partial x} \left(\lambda_n(x) \frac{\partial u^{n+1}}{\partial x} \right) v(x) dx + \int_a^b \gamma_n \left(\frac{\partial u^{n+1}}{\partial x} \right) u^{n+1} v(x) dx = \int_a^b \tilde{f}_n(x) v(x) dx \quad \forall v \in H_0$$

.

И, таким образом, получим вариационную постановку Галёркина для краевых условий $u(a, t) = u_g$, $\lambda(b) \frac{\partial u}{\partial x}(b) + \beta(u(b) - u_\beta) = 0$:

$$\begin{aligned} & \int_a^b \lambda_n(x) \frac{\partial u^{n+1}}{\partial x} \frac{\partial v}{\partial x} dx + \int_a^b \gamma_n \left(\frac{\partial u^{n+1}}{\partial x} \right) u^{n+1} v(x) dx + \beta u(b) v(b) \\ &= \int_a^b \tilde{f}_n(x) v(x) dx + \beta u_\beta v(b), \quad \forall v \in H_0 \end{aligned}$$

При этом, функция $u^{n+1}(x) = \sum_{i=1}^m q_i \cdot \psi_i(x)$, базисные функции – линейные.

3. Метод простой итерации

В результате конечноэлементной аппроксимации нелинейной краевой задачи получается система нелинейных уравнений $A(q)q = b(q)$, у которой компоненты матрицы $A(q)$ и вектора правой части $b(q)$ определяются следующим образом:

$$A_{ij} = \int_a^b \lambda_n(x) \frac{\partial \psi_i}{\partial x} \frac{\partial \psi_j}{\partial x} dx + \int_a^b \gamma_n \left(\sum_k q_{s,k} \frac{\partial \psi_k}{\partial x} \right) \psi_i \psi_j dx.$$

К последнему элементу применяем краевое условие:

$$A_{mm} = \int_a^b \lambda_n(x) \frac{\partial \psi_m}{\partial x} \frac{\partial \psi_m}{\partial x} dx + \int_a^b \gamma_n \left(\sum_k q_{s,k} \frac{\partial \psi_k}{\partial x} \right) \psi_m \psi_m dx + \beta.$$

Теперь преобразуем выражения для получения локальных матриц:

$$\begin{aligned}\hat{A}_{ij}^k &= \int_{x_k}^{x_{k+1}} \lambda_n(x) \frac{\partial \varphi_i}{\partial x} \frac{\partial \varphi_j}{\partial x} dx + \int_{x_k}^{x_{k+1}} \gamma_n \left(\hat{q}_{s,1} \frac{\partial \varphi_1}{\partial x} + \hat{q}_{s,2} \frac{\partial \varphi_2}{\partial x} \right) \varphi_i \varphi_j dx = \\ &\frac{(-1)^{1-\delta_{ij}}}{h_k^2} \int_{x_k}^{x_{k+1}} \lambda_n(x) dx + \widetilde{\gamma_n^k} \int_{x_k}^{x_{k+1}} \varphi_i \varphi_j dx = \frac{(-1)^{1-\delta_{ij}}}{h_k^2} \int_{x_k}^{x_{k+1}} (\lambda^1_{n,k} \varphi_1 + \lambda^2_{n,k} \varphi_2) dx + \\ &\widetilde{\gamma_n^k} \int_{x_k}^{x_{k+1}} \varphi_i \varphi_j dx.\end{aligned}$$

Получаем следующие выражения для локальной матрицы:

$$\begin{aligned}\hat{A} &= \frac{\lambda_{n,k}^1 + \lambda_{n,k}^2}{2h_k} \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} + \frac{\widetilde{\gamma_n^k} \cdot h_k}{6} \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \\ \widetilde{\gamma_n^k} &= \frac{1}{\tau_n} \sigma_n \left(\frac{\widehat{q_{s,2}^n} - \widehat{q_{s,1}^n}}{h_k} \right) \\ \lambda_{n,k}^i &= \lambda_n(x_i^k)\end{aligned}$$

Теперь получим локальный вектор правой части:

$$\begin{aligned}\hat{b}_i &= \int_{x_k}^{x_{k+1}} \tilde{f}_n(x) \varphi_i dx = \int_{x_k}^{x_{k+1}} \left[f(x, t_{n+1}) + \frac{1}{\tau_n} \sigma \left(\frac{\partial u^{n,k}}{\partial x} \right) \cdot u^{n,k} \right] (x) \varphi_i dx \\ \hat{b} &= \frac{h_k}{6} \begin{pmatrix} 2f_{n,1}^k + f_{n,2}^k \\ f_{n,1}^k + 2f_{n,2}^k \end{pmatrix} + \frac{\widetilde{\gamma_n^k} \cdot h_k}{6} \begin{pmatrix} 2\widehat{q_1^n} + \widehat{q_2^n} \\ \widehat{q_1^n} + 2\widehat{q_2^n} \end{pmatrix}.\end{aligned}$$

Использование параметра релаксации при решении системы методом простой итерации.

Параметр релаксации вводится для ускорения сходимости итерационного процесса. Каждое последующее приближение решения строится как $q_k^\omega = \omega q^k + (1 - \omega)q^{k-1}$. И, таким образом, параметр релаксации необходимо подобрать так, чтобы значение функционала $R(\omega) = \|A(q_k^\omega)q_k^\omega - b(q_k^\omega)\|$ было минимально (используем алгоритм поиска интервала, содержащего минимум функции, а затем метод золотого сечения).

Тем самым сборка матрицы в программе для метода простой итерации выглядит таким образом:

```
void FEM::matrix_gen_si(double* q_n, double* q_t)
{
    // Локальная матрица и вектор правой части
    double A_loc[2][2], b_loc[2];

    // Генерация матрицы
    for(int i = 0; i < elements_n; i++){
        double x1 = nodes[elements[i].node1]; // левый узел для элемента
        double x2 = nodes[elements[i].node2]; // правый узел для элемента
```

```

        double h_k = x2 - x1; // шаг по сетки
        double x_midd = (x2+x1)/2; // центральная точка сетки

        double G_11_val, M_12_val; //значения элементов в матирце жескости и массы

        G_11_val = (lambda(x1, t_new) + lambda(x2, t_new))/(2*h_k);
        double gamma_aver = sigma((q_n[i+1] - q_n[i])/h_k, x_midd)/h_t;
        M_12_val = gamma_aver * h_k / 6;

        A_loc[0][0] = A_loc[1][1] = G_11_val + 2 * M_12_val;
        A_loc[1][0] = A_loc[0][1] = -G_11_val + M_12_val;

        double f1 = f(x1, t_new), f2 = f(x2, t_new); // значения правой части
        b_loc[0] = h_k*(2*f1 + f2)/6.0 + M_12_val * (2*q_last[i] + q_last[i+1]);
        b_loc[1] = h_k*(f1 + 2*f2)/6.0 + M_12_val * (q_last[i] + 2*q_last[i+1]);

        dia_0[i] += A_loc[0][0];
        dia_0[i+1] += A_loc[1][1];
        dia_1[i] += A_loc[0][1];
        dia_2[i] += A_loc[1][0];

        q_t[i] += b_loc[0];
        q_t[i+1] += b_loc[1];

    }

    // учёт краевых условий
    // Левых
    switch(bound_type_left){
        case 1:{
            dia_0[0] = 1;
            dia_1[0] = 0;
            q_t[0] = bounds_l[0](t_new);
            }break;

        case 2:{
            q_t[0] += bounds_l[0](t_new);
            }break;

        case 3:{
            dia_0[0] += bounds_l[0](t_now);
            q_t[0] += bounds_l[0](t_new) * bounds_l[1](t_new);
            }break;
    };

    // Правых
    switch(bound_type_right){
        case 1:{
            dia_0[nodes_n-1] = 1;
            dia_2[nodes_n-2] = 0;
            q_t[nodes_n-1] = bounds_r[0](t_new);
            }break;

        case 2:{
            q_t[nodes_n-1] += bounds_r[0](t_new);
            }break;

        case 3:{
            dia_0[nodes_n-1] += bounds_r[0](t_new);
            q_t[nodes_n-1] += bounds_r[0](t_new) * bounds_r[1](t_new);
            }
    };
}

```

```

    }break;
};

}

```

4. Метод Ньютона

Метод Ньютона основан на линеаризации нелинейных уравнений системы $A(q)q = b(q)$. Каждый нелинейный член уравнений системы представляется в виде разложения в ряд Тейлора в окрестности точки q^0 . Таким образом, слагаемые левой части уравнений системы представляются в виде:

$$\hat{A}_{ij}^L = \hat{A}_{ij}(q_s^{n+1}) + \sum_k \frac{\partial \hat{A}_{ik}(\hat{q}_{s+1}^{n+1})}{\partial \hat{q}_{s+1,j}^{n+1}} \hat{q}_{s,k}^{n+1} - \frac{\partial \hat{b}_i(\hat{q}_{s+1}^{n+1})}{\partial q_{s+1,j}^{n+1}}.$$

А компоненты вектора правой части – в виде:

$$\hat{b}_i^L = \hat{b}_i(\hat{q}_s^{n+1}) + \sum_j \left\{ \hat{q}_{s,j}^{n+1} \left[\sum_k \frac{\partial \hat{A}_{ij}(\hat{q}_{s+1}^{n+1})}{\partial \hat{q}_{s+1,k}^{n+1}} \cdot \hat{q}_{s,k}^{n+1} \right] \right\} - \sum_k \frac{\partial \hat{b}_i(\hat{q}_{s+1}^{n+1})}{\partial q_{s+1,k}^{n+1}} \hat{q}_{s,k}^{n+1}.$$

Вычислим производные, которые входят в формулы:

$$\begin{aligned} \frac{\partial \hat{A}_{ij}(\hat{q}_{s+1}^{n+1})}{\partial \hat{q}_{s,i}} &= \frac{\partial}{\partial \hat{q}_{s,i}} \int_{x_k}^{x_{k+1}} \gamma_n \left(\hat{q}_{s,1} \frac{\partial \varphi_1}{\partial x} + \hat{q}_{s,2} \frac{\partial \varphi_2}{\partial x} \right) \varphi_i \varphi_j dx = \frac{\partial}{\partial \hat{q}_{s,i}} \int_{x_k}^{x_{k+1}} \gamma_n \left(\frac{\widehat{q_{s,2}^{n+1}} - \widehat{q_{s,1}^{n+1}}}{h_k} \right) \varphi_i \varphi_j dx = \\ &= \int_{x_k}^{x_{k+1}} \frac{\partial}{\partial \hat{q}_{s,i}} \left\{ \gamma_n \left(\frac{\widehat{q_{s,2}^{n+1}} - \widehat{q_{s,1}^{n+1}}}{h_k} \right) \right\} \varphi_i \varphi_j dx = \frac{\pm 1}{h_k} \int_{x_k}^{x_{k+1}} \frac{\partial \gamma_n(p)}{\partial p} \varphi_i \varphi_j dx \end{aligned}$$

Пусть зависимость σ от $\frac{\partial u}{\partial x}$ задана следующим образом: $\sigma \left(\frac{\partial u}{\partial x} \right) = x \left[\left(\frac{\partial u}{\partial x} \right)^2 + c \right]$, введём функцию $\xi \left(\frac{\partial u}{\partial x} \right) = 2x \frac{\partial u}{\partial x}$, тогда выражение для производных можно переписать, как:

$$\begin{aligned} \frac{\partial \hat{A}_{ij}(\hat{q}_{s+1}^{n+1})}{\partial \hat{q}_{s,i}} &= \frac{\pm 1}{\tau_n h_k} \int_{x_k}^{x_{k+1}} \xi_n \cdot \varphi_i \varphi_j dx, \\ \frac{\partial \hat{b}_i(\hat{q}_{s+1}^{n+1})}{\partial q_{s+1,j}^{n+1}} &= \frac{1}{\tau_n} \int_{x_k}^{x_{k+1}} \frac{\partial \sigma}{\partial q_{s+1,j}^{n+1}} u^n \varphi_i dx = \frac{\pm 1}{\tau_n h_k} \int_{x_k}^{x_{k+1}} \xi \cdot (\widehat{q_1^n} \varphi_1 + \widehat{q_2^n} \varphi_2) \varphi_i dx. \end{aligned}$$

В матричном виде запись будет иметь следующий вид:

$$\hat{A}^{L1} = \frac{\tilde{\xi}_n}{6\tau_n} \begin{pmatrix} -2 & 2 \\ -1 & 1 \end{pmatrix} q_{s,1}^{n+1} + \frac{\tilde{\xi}_n}{6\tau_n} \begin{pmatrix} -1 & 1 \\ -2 & 2 \end{pmatrix} q_{s,2}^{n+1} + \frac{\tilde{\xi}_n}{6\tau_n} \begin{pmatrix} 2q_1^n + q_2^n & -(2q_1^n + q_2^n) \\ q_1^n + 2q_2^n & -(q_1^n + 2q_2^n) \end{pmatrix}$$

$$\hat{b}_i^{L1} = \frac{\partial A_{i1}}{\partial q_1} q_{s,1}^2 + \left(\frac{\partial A_{i1}}{\partial q_2} + \frac{\partial A_{i2}}{\partial q_1} \right) q_{s,1} q_{s,2} + \frac{\partial A_{i2}}{\partial q_2} q_{s,2}^2 - \frac{\partial b_i}{\partial q_1} q_{s,1} - \frac{\partial b_i}{\partial q_2} q_{s,2}$$

$$b_1^{L1} = -\frac{\tilde{\xi}_n}{3\tau_n} q_{s,1}^2 + \frac{\tilde{\xi}_n}{6\tau_n} q_{s,1} q_{s,2} + \frac{\tilde{\xi}_n}{6\tau_n} q_{s,2}^2 + \frac{\tilde{\xi}_n}{6\tau_n} (2q_1^n + q_2^n) q_{s,1} - \frac{\tilde{\xi}_n}{6\tau_n} (2q_1^n + q_2^n) q_{s,2}$$

$$b_2^{L1} = -\frac{\tilde{\xi}_n}{6\tau_n} q_{s,1}^2 - \frac{\tilde{\xi}_n}{6\tau_n} q_{s,1} q_{s,2} + \frac{\tilde{\xi}_n}{3\tau_n} q_{s,2}^2 + \frac{\tilde{\xi}_n}{6\tau_n} (q_1^n + 2q_2^n) q_{s,1} - \frac{\tilde{\xi}_n}{6\tau_n} (q_1^n + 2q_2^n) q_{s,2}$$

Использование коэффициента демпфирования при решении системы методом Ньютона

Коэффициент демпфирования вводится для того, чтобы избежать случаев, когда линеаризованная матрица A^L перестает быть положительно определенной. Ставится перед добавками с производными, выбирается вручную.

Тем самым сборка матрицы в программе для метода Ньютона выглядит таким образом:

```
void FEM::matrix_gen_Newton(double* q_n, double* q_t)
{
    double A_loc[2][2], b_loc[2];

    for(int i = 0; i < elements_n; i++){
        double x1 = nodes[elements[i].node1];
        double x2 = nodes[elements[i].node2];

        double h_k = x2 - x1;
        double x_midd = (x2+x1)/2;

        double G_11_val, M_12_val;

        G_11_val = (lambda(x1, t_new) + lambda(x2, t_new))/(2*h_k);
        double gamma_aver = sigma((q_n[i+1] - q_n[i])/h_k, x_midd)/h_t;
        M_12_val = gamma_aver * h_k / 6;

        double sigm_ux_aver = sigma_ux((q_n[i+1] - q_n[i])/h_k, x_midd);

        A_loc[0][0] = A_loc[1][1] = G_11_val + 2 * M_12_val;
        A_loc[1][0] = A_loc[0][1] = -G_11_val + M_12_val;
        double reg_par = 1.0;

        A_loc[0][0] += reg_par * sigm_ux_aver * (-q_n[i]/3.0 - q_n[i+1]/6.0 + (2*q_last[i] + q_last[i+1])/6.0) / h_t;
        A_loc[0][1] += reg_par * sigm_ux_aver * (q_n[i]/3.0 + q_n[i+1]/6.0 - (2*q_last[i] + q_last[i+1])/6.0) / h_t;
        A_loc[1][0] += reg_par * sigm_ux_aver * (-q_n[i]/6.0 - q_n[i+1]/3.0 + (q_last[i] + 2*q_last[i+1])/6.0) / h_t;
```

```

        A_loc[1][1] += reg_par * sigm_ux_aver * (q_n[i]/6.0 + q_n[i+1]/3.0 - (q_last[i] +
2*q_last[i+1])/6.0) / h_t;

    double f1 = f(x1, t_new), f2 = f(x2, t_new);

    b_loc[0] = h_k*(2*f1 + f2)/6.0 + M_12_val * (2*q_last[i] + q_last[i+1]);
    b_loc[1] = h_k*(f1 + 2*f2)/6.0 + M_12_val * (q_last[i] + 2*q_last[i+1]);

    b_loc[0] += reg_par * sigm_ux_aver * (-q_n[i]*q_n[i]/3.0 + q_n[i]*q_n[i+1]/6.0 +
q_n[i+1]*q_n[i+1]/6.0 + (2*q_last[i] + q_last[i+1])*(q_n[i]-q_n[i+1])/6.0) / h_t;
    b_loc[1] += reg_par * sigm_ux_aver * (-q_n[i]*q_n[i]/6.0 - q_n[i]*q_n[i+1]/6.0 +
q_n[i+1]*q_n[i+1]/3.0 + (q_last[i] + 2*q_last[i+1])*(q_n[i]-q_n[i+1])/6.0) / h_t;

    dia_0[i] += A_loc[0][0];
    dia_0[i+1] += A_loc[1][1];
    dia_1[i] += A_loc[0][1];
    dia_2[i] += A_loc[1][0];

    q_t[i] += b_loc[0];
    q_t[i+1] += b_loc[1];

}

switch(bound_type_left){
    case 1:{
        dia_0[0] = 1;
        dia_1[0] = 0;
        q_t[0] = bounds_l[0](t_new);
        }break;

    case 2:{
        q_t[0] += bounds_l[0](t_new);
        }break;

    case 3:{
        dia_0[0] += bounds_l[0](t_new);
        q_t[0] += bounds_l[0](t_new) * bounds_l[1](t_new);
        }break;
};

switch(bound_type_right){
    case 1:{
        dia_0[nodes_n-1] = 1;
        dia_2[nodes_n-2] = 0;
        q_t[nodes_n-1] = bounds_r[0](t_new);
        }break;

    case 2:{
        q_t[nodes_n-1] += bounds_r[0](t_new);
        }break;

    case 3:{
        dia_0[nodes_n-1] += bounds_r[0](t_new);
        q_t[nodes_n-1] += bounds_r[0](t_new) * bounds_r[1](t_new);
        }break;
};

}

```

4. Тестирование

Протестируем разработанные программы (в каждом тесте будем приводить только относительную погрешность решения $\frac{\|u^* - u\|}{\|u^*\|}$).

1. Линейная нестационарная задача с полиномиальным решением

$$u = t^2, \lambda(x, t) = 2, \sigma\left(\frac{\partial u}{\partial x}\right) = 3$$

$$\text{Уравнение: } -\frac{\partial}{\partial x} \left(2 \frac{\partial u}{\partial x} \right) + 3 \frac{\partial u}{\partial t} = 6t$$

Краевые условия первого рода.

Сетка: разбиение отрезка $[0,1]$ с шагом $h = 0.1$

Сетка по времени: $t_0 = 0, \tau = 0.01$

t	Метод простой итерации	Метод Ньютона
0.01	2.170e-11	2.170e-11
0.02	2.170e-11	2.170e-11
0.03	6.037e-12	6.037e-12
0.04	3.539e-12	3.539e-12
0.05	2.508e-12	2.508e-12
0.06	1.960e-12	1.960e-12
0.07	1.609e-12	1.609e-12

Для каждого значения t требуемая точность была достигнута за одну итерацию.

2. Линейная нестационарная задача

$$u = x, \lambda(x, t) = 2, \sigma\left(\frac{\partial u}{\partial x}\right) = 3$$

$$\text{Уравнение: } -\frac{\partial}{\partial x} \left(2 \frac{\partial u}{\partial x} \right) + 3 \frac{\partial u}{\partial t} = 0$$

Краевые условия первого рода

Сетка: разбиение отрезка $[0,1]$ с шагом $h = 0.1$

Сетка по времени: $t_0 = 0, \tau = 0.01$

t	Метод простой итерации	Метод Ньютона
0.01	3.547e-12	3.547e-12
0.02	6.123e-13	6.123e-13
0.03	4.874e-13	4.874e-13
0.04	4.350e-13	4.350e-13

0.05	3.992e-13	3.992e-13
0.06	3.739e-13	3.739e-13
0.07	3.565e-13	3.565e-13

Для каждого значения t требуемая точность была достигнута за одну итерацию.

3. Линейная нестационарная задача с полиномиальным решением

$$u = x^2, \lambda(x, t) = 2, \sigma\left(\frac{\partial u}{\partial x}\right) = 3$$

$$\text{Уравнение: } -\frac{\partial}{\partial x}\left(2 \frac{\partial u}{\partial x}\right) + 3 \frac{\partial u}{\partial t} = -4$$

$$\text{Краевые условия: } u(0) = u(1) = t$$

Сетка: разбиение отрезка $[0,1]$ с шагом $h = 0.1$

Сетка по времени: $t_0 = 0, \tau = 0.01$

t	Метод простой итерации	Метод Ньютона
0.01	2.331e-12	2.331e-12
0.02	1.359e-11	1.359e-11
0.03	5.912e-12	5.912e-12
0.04	4.551e-12	4.551e-12
0.05	4.027e-12	4.027e-12
0.06	3.816e-12	3.816e-12
0.07	3.765e-12	3.765e-12

Для каждого значения t требуемая точность была достигнута за одну итерацию.

4. Линейная нестационарная задача с полиномиальным решением

$$u = x^3, \lambda(x, t) = 2, \sigma\left(\frac{\partial u}{\partial x}\right) = 3$$

$$\text{Уравнение: } -\frac{\partial}{\partial x}\left(2 \frac{\partial u}{\partial x}\right) + 3 \frac{\partial u}{\partial t} = -12x$$

$$\text{Краевые условия: } u(0) = u(1) = t$$

Сетка: разбиение отрезка $[0,1]$ с шагом $h = 0.1$

Сетка по времени: $t_0 = 0, \tau = 0.01$

t	Метод простой итерации	Метод Ньютона

0.01	9.821e-13	9.821e-13
0.02	1.549e-11	1.549e-11
0.03	9.585e-12	9.585e-12
0.04	8.723e-12	8.723e-12
0.05	9.390e-12	9.390e-12
0.06	1.164e-11	1.164e-11
0.07	1.765e-11	1.765e-11

Для каждого значения t требуемая точность была достигнута за одну итерацию.

5. Решение зависит от x и t

$$u = 1 + x + t + xt, \lambda(x, t) = 2, \sigma\left(\frac{\partial u}{\partial x}\right) = 3$$

$$\text{Уравнение: } -\frac{\partial}{\partial x}\left(2\frac{\partial u}{\partial x}\right) + 3\frac{\partial u}{\partial t} = 3(1 + x)$$

$$\text{Краевые условия: } u(0) = u(1) = t$$

Сетка: разбиение отрезка $[0,1]$ с шагом $h = 0.1$

Сетка по времени: $t_0 = 0, \tau = 0.01$

t	Метод простой итерации	Метод Ньютона
0.01	6.541e-13	6.541e-13
0.02	4.646e-12	4.646e-12
0.03	2.129e-12	2.129e-12
0.04	1.429e-12	1.429e-12
0.05	1.090e-12	1.090e-12
0.06	8.814e-13	8.814e-13
0.07	7.340e-13	7.340e-13

Для каждого значения t требуемая точность была достигнута за одну итерацию.

6. Нелинейная задача

$$u = xt, \lambda(x, t) = 2, \sigma\left(\frac{\partial u}{\partial x}\right) = \frac{\partial u}{\partial x}$$

$$\text{Уравнение: } -\frac{\partial}{\partial x}\left(2\frac{\partial u}{\partial x}\right) + \frac{\partial u}{\partial x}\frac{\partial u}{\partial t} = xt$$

$$\text{Краевые условия: } u(0) = u(1) = t$$

Сетка: разбиение отрезка $[0,1]$ с шагом $h = 0.1$

Сетка по времени: $t_0 = 0, \tau = 0.01$

Метод простой итерации		Метод Ньютона		
t	Погрешность	Число итераций	Погрешность	Число итераций
0.01	1.435e-11	8	1.425e-11	2
0.02	7.691e-12	8	7.608e-12	2
0.03	5.198e-12	8	5.172e-12	2
0.04	4.354e-12	8	4.121e-12	2
0.05	3.379e-12	8	3.339e-12	2
0.06	3.438e-12	8	3.271e-12	2
0.07	2.779e-12	8	2.830e-12	2

7. Линейная нестационарная задача, решение зависит от времени

Тестовая функция: $u = t, \lambda(x, t) = 1, \sigma\left(\frac{\partial u}{\partial x}\right) = x\left(\frac{\partial u}{\partial x}\right)^2$

Уравнение: $-\frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x}\right) + x\left(\frac{\partial u}{\partial x}\right)^2 \frac{\partial u}{\partial t} = 0$

Краевые условия: $u(0) = u(1) = t$

Сетка: разбиение отрезка $[0,1]$ с шагом $h = 0.1$

Сетка по времени: $t_0 = 0, \tau = 0.01$

t	Метод простой итерации	Метод Ньютона
0.01	2.171e-11	2.171e-11
0.02	1.087e-11	1.087e-11
0.03	7.276e-12	7.276e-12
0.04	5.458e-12	5.458e-12
0.05	4.442e-12	4.442e-12
0.06	3.704e-12	3.704e-12
0.07	3.171e-12	3.171e-12

Для каждого значения t требуемая точность была достигнута за одну итерацию.

8. Нелинейная задача

Тестовая функция: $u = xt, \lambda(x, t) = x, \sigma\left(\frac{\partial u}{\partial x}\right) = x\left(\frac{\partial u}{\partial x}\right)^2$

Уравнение: $-\frac{\partial}{\partial x}\left(x \frac{\partial u}{\partial x}\right) + x\left(\frac{\partial u}{\partial x}\right)^2 \frac{\partial u}{\partial t} = x^2 t^2 - t$

Краевые условия: $u(0) = 0, x \frac{\partial u}{\partial x}\Big|_{x=1} = t$

Сетка: разбиение отрезка $[0,1]$ с шагом $h = 0.1$

Сетка по времени: $t_0 = 0, \tau = 0.01$

Метод простой итерации			Метод Ньютона	
t	Погрешность	Число итераций	Погрешность	Число итераций
0.01	1.24E-04	5	1.24E-04	4
0.02	2.39E-04	6	2.39E-04	7
0.03	3.46E-04	6	3.46E-04	3
0.04	4.45E-04	6	4.45E-04	4
0.05	5.39E-04	6	5.39E-04	4
0.06	6.27E-04	7	6.27E-04	4
0.07	7.10E-04	7	7.10E-04	4

9. Нелинейная задача с не полиномиальным решением

Тестовая функция: $u = e^{-t} \sin(x), \lambda(x, t) = xt, \sigma\left(\frac{\partial u}{\partial x}\right) = x\left(\frac{\partial u}{\partial x}\right)^2$

Уравнение: $-\frac{\partial}{\partial x}\left(xt \frac{\partial u}{\partial x}\right) + x\left(\frac{\partial u}{\partial x}\right)^2 \frac{\partial u}{\partial t} = e^{-t} t(x \sin(x) - \cos(x)) - xe^{-3t} \cos^2(x) \sin(x)$

Краевые условия: $u(0) = 0, xt \frac{\partial u}{\partial x}\Big|_{x=1} + u\Big|_{x=1} - e^{-t}(\sin(1) - t \cos(1)) = 0$

Сетка: разбиение отрезка $[0,1]$ с шагом $h = 0.1$

Сетка по времени: $t_0 = 0, \tau = 0.01$

Метод простой итерации			Метод Ньютона	
t	Погрешность	Число итераций	Погрешность	Число итераций
0.01	3.06E-03	99	3.06E-03	4
0.02	7.52E-03	67	7.52E-03	4
0.03	1.30E-02	52	1.30E-02	11
0.04	1.94E-02	44	1.94E-02	47
0.05	2.65E-02	56	2.65E-02	99

0.06	3.43E-02	99	3.43E-02	99
0.07	4.26E-02	99	4.26E-02	99

5. Выводы

Аникина Полина

В ходе данной работы, был сделан вывод, что метод Ньютона, работает лучше, так как в среднем сходится за меньшее число итераций. На простейших тестах с линейной задачей методы простой итерации и Ньютона достигают необходимой точности за 1 итерацию. На рассмотренных задачах точность методов примерно одинаковая. Несмотря на то, что метод Ньютона сложнее метода простой итерации в получении формул и построении матриц, он позволяет учитывать особенности нелинейности конкретной задачи.

Востриков Вячеслав

После проведенных исследований, видно, что метод Ньютона решает нелинейную задачу за меньшее число итераций, чем метод простой итерации. Так же в данном методе есть возможность учитывать особенности нелинейности (зависимость коэффициентов от параметров). Недостатком этого метода является сложность в построении матриц и получении формул. Метод простой итерации и Ньютона достигают нужной точности за одну итерацию при тестировании на простейших тестах с линейной задачей.

6. Текст программы

```
#include <windows.h>

#include "grid_gen.h"
#include "FEM.h"

double lam1(double x, double t){return x;}
double f1(double x, double t){return -1;}
double b_l1(double t){return 0;}
double b_r1(double t){return 1;}
double u1(double x, double t){return x;}

double lam2(double x, double t){return x;}
double f2(double x, double t){return x*x*t*t - t;}
double b_l2(double t){return 0;}
double b_r2(double t){return t;}
double u2(double x, double t){return x*t;}
```

```

double lam3(double x, double t){return 1;}
double f3(double x, double t){return 0;}
double b_l3(double t){return t;}
double b_r3(double t){return t;}
double u3(double x, double t){return t;}

double lam4(double x, double t){return x*t;}
double f4(double x, double t){return exp(-t)*t*(x*sin(x)-cos(x))-x*exp(-
3*t)*cos(x)*cos(x)*sin(x);}
double b_l4(double t){return 0;}
double b_r4(double t){return 1;}
double b_r42(double t){return exp(-t)*(-t*cos((double)1.0) + sin((double)1.0));}
double u4(double x, double t){return exp(-t)*sin(x);}

int main(){

    grid_generator::generate_reg_grid_ins(0, 1, 1E-1, "grid1.txt");
    vector<double> start_gen;
    grid_generator::generate_reg_grid_ins(0, 1, 1E-1, start_gen);

    FILE* out_f = fopen("u0.txt", "w");
    for(int i = 0; i < start_gen.size(); i++)
        fprintf(out_f, "% .15lf\n", u4(start_gen[i], 0));
    fclose(out_f);

    FEM our_sol_si, our_sol_Newton;

    our_sol_si.init("grid1.txt", "u0.txt", "pars.txt");
    our_sol_si.init_coefs(lam4, f4);
    our_sol_si.init_sigma((double)0.0);
    our_sol_si.init_bounds(1, 3, b_l4, NULL, b_r4, b_r42);
    our_sol_si.set_analytic(u4);

    our_sol_Newton.init("grid1.txt", "u0.txt", "pars.txt");
    our_sol_Newton.init_coefs(lam4, f4);
    our_sol_Newton.init_sigma((double)0.0);
    our_sol_Newton.init_bounds(1, 3, b_l4, NULL, b_r4, b_r42);
    our_sol_Newton.set_analytic(u4);

    for(int i = 0 ; i < 7; i++){
        our_sol_si.simple_iteration_step();
        our_sol_Newton.Newton_step();
        our_sol_si.out_q("out_si.txt");
        our_sol_Newton.out_q("out_Newton.txt");
    }
    system("pause");
    return 0;
}

// solver_LU.h
#pragma once
// Решение СЛАУ для трёхдиагональной матрицы
// Решение записывается на место правой части b
void solve_LU(double* dia_0, double* dia_1, double* dia_2, double* b, int n);

// grid_gen.h
#pragma once
#include <math.h>
#include <stdio.h>
#include <string>
#include <vector>

```

```

using namespace std;
typedef vector<double> dvector;

// Генератор одномерных сеток
class grid_generator
{
public:
    // Описание параметров:
    // a - начало отрезка
    // b - конец отрезка
    // h_min - минимальный шаг
    // k - коэффициент разрядки
    // n - число узлов сетки

    // Генерация неравномерной сетки
    static bool generate_unreg_grid(double a, double b, double h_min, double k, string file_name);
    static bool generate_unreg_grid(double a, double b, double h_min, double k, dvector &grid_vector);
    static bool generate_unreg_grid(double a, double b, double h_min, double k, int &n, double*& grid_mass);

    // Генерация равномерной сетки
    static bool generate_reg_grid(double a, double b, double h, string file_name);
    static bool generate_reg_grid(double a, double b, double h, dvector &grid_vector);
    static bool generate_reg_grid(double a, double b, double h, int &n, double*& grid_mass);

    // Генерация вложенных сеток
    // Для неравномерных
    static bool generate_unreg_grid_ins(double a, double b, double h_min, double k, string file_name);
    static bool generate_unreg_grid_ins(double a, double b, double h_min, double k, dvector &grid_vector);
    static bool generate_unreg_grid_ins(double a, double b, double h_min, double k, int &n, double*& grid_mass);

    // Для равномерных
    static bool generate_reg_grid_ins(double a, double b, double h, string file_name);
    static bool generate_reg_grid_ins(double a, double b, double h, dvector &grid_vector);
    static bool generate_reg_grid_ins(double a, double b, double h, int &n, double*& grid_mass);

private:
    static int generate_unreg_grid_pr(double a, double b, double h_min, double k, double *&grid_mass);
    static int generate_reg_grid_pr(double a, double b, double h, double *&grid_mass);
};

// FEM.h
#pragma once
#include <string>
#include <math.h>
#include <stdio.h>
#include "solver_LU.h"

using namespace std;

// Метод конечных элементов для одномерной нестационарной нелинейной задачи

```

```

typedef double(*func_t)(double, double); // указатель на функцию двух переменных (пространства
и времени)
typedef double(*func)(double); // указатель на функцию одной переменной (предположительно -
времени)

//Структура описывающая элемент - отрезок
struct element
{
    int node1, node2;
};

class FEM
{
public:
    void init(string node_file, string u0_file, string info_file);

    // Устанавливает коэффициенты уравнения
    void init_coefs(func_t set_lambda, func_t set_f);

    // Устанавливает коэффициент для сигма, который задаётся как: sigma(u_x, x, t) = ((u_x)^2
+ c) * x
    void init_sigma(double set_c);

    // Метод для инициализации краевых условий
    // set_b_l - тип левых краевых

    void init_bounds(int set_b_l, int set_b_r, func bound_l1, func bound_l2, func bound_r1,
func bound_r2);

    double get_current_time();

    void set_analytic(func_t set_u);

    void simple_iteration_step();

    void Newton_step();

    //Вывод q в файл
    void out_q(string file_out);

private:
    double* nodes; // массив координат узлов
    element* elements; // массив элементов
    int nodes_n; // количество узлов
    int elements_n; // количество элементов

    func_t lambda, f; // коэффициенты уравнения
    double sigma_c; // коэффициент для сигма
    double sigma(double u_x, double x); // функция для вычисления сигма
    double sigma_ux(double u_x, double x); // производная сигма по u_x

    func_t analytic; // Аналитическое решение

    double* q_last; // массив весов с предыдущего временного слоя
    double* q_new; // массив узлов, с нового временного слоя
    double* q_temp; // массив узлов, для промежуточных итераций
    double* q_min; // массив, используемый для минимизации невязки

```

```

        double* q_min_rp; // массив, используемый для минимизации невязки хранящий правую часть
матрицы
        double* q_min_R; // массив, используемый для минимизации - вектор невязки. Так же
используется для определения невязки, при выходе из процесса решения СНУ

    double *dia_0, *dia_1, *dia_2; // диагонали матрицы

    double t_now; // текущее время
    double t_new; // следующее время
    double h_t; // шаг по времени
    double k_t; // коэффициент разрядки по времени
    int iter_max; // максимальное число итераций на одном временном слое
    double eps; // точность для решения нелинейной системы

    int bound_type_left, bound_type_right; // тип краевых условий справа и слева, значения
соответственно могут быть 1, 2 или 3
    func* bounds_l; // функции для краевых условий слева
    func* bounds_r; // функции для краевых условий справа

    double norm_q_dif(double* q1, double* q2); // ||q1 - q2||
    double norm_q(double* q1); // ||q||

    void matrix_gen_si(double* q_n, double* q_t);

    // Метод генерации матрицы для метода Ньютона
    void matrix_gen_Newton(double* q_n, double* q_t);

    double w_min(); // Нахождение параметра релаксации
    double w_min_func(double w); // Функция для минимизации
    void w_min_find_area(double& a, double& b, double delta); // Нахождение отрезка,
содержащего минимум
    double w_min_golden(double a, double b, double eps_min); // Минимизация, методом
золотого сечения
};

// solver_LU.cpp
#include "solver_LU.h"

void solve_LU(double* dia_0, double* dia_1, double* dia_2, double* b, int n){

    for(int j = 0; j < n-1 ; j++){
        dia_1[j] /= dia_0[j];
        dia_0[j+1] -= dia_2[j]*dia_1[j];
    }

    b[0] /= dia_0[0];
    for(int i = 1; i < n; i++)
        b[i] = (b[i] - dia_2[i-1]*b[i-1])/dia_0[i];
    for(int i = n - 2; i >=0 ; i--)
        b[i] -= dia_1[i]*b[i+1];
}

// grid_gen.cpp
#include "grid_gen.h"

int grid_generator::generate_unreg_grid_pr(double a, double b, double h_min, double k, double
*&grid_mass){
    int n; // число узлов сетки
    double n_d = log(1 - (b-a)*(1-k)/h_min) / log(k);
    n = n_d;
    n += 2;
}

```

```

grid_mass = new double [n];
double h = h_min; // текущий шаг
grid_mass[0] = a;
for(int i = 1; i < n; i++){
    grid_mass[i] = grid_mass[i-1] + h;
    h *= k;
}
grid_mass[n-1] = b;

return n;
}

int grid_generator::generate_reg_grid_pr(double a, double b, double h, double *&grid_mass){
    int n; // число узлов сетки

    n = (b-a)/h;
    n++;

    grid_mass = new double [n];
    grid_mass[0] = a;
    for(int i = 1; i < n; i++){
        grid_mass[i] = grid_mass[i-1] + h;
    }
    grid_mass[n-1] = b;

    return n;
}

bool grid_generator::generate_unreg_grid(double a, double b, double h_min, double k, int &n,
double *&grid_mass){
    if(a > b || h_min <= 0 || k <= 1) return false; // проверка данных

    n = generate_unreg_grid_pr(a, b, h_min, k, grid_mass);

    return true;
}

bool grid_generator::generate_unreg_grid(double a, double b, double h_min, double k, dvector
&grid_vector){
    if(a > b || h_min <= 0 || k <= 1) return false; // проверка данных

    double *grid_mass;

    int n = generate_unreg_grid_pr(a, b, h_min, k, grid_mass);

    grid_vector.clear();
    for(int i = 0; i < n; i++)
        grid_vector.push_back(grid_mass[i]);

    delete[] grid_mass;

    return true;
}

bool grid_generator::generate_unreg_grid(double a, double b, double h_min, double k, std::string
file_name){
    if(a > b || h_min <= 0 || k <= 1) return false; // проверка данных

    double *grid_mass;
    int n = generate_unreg_grid_pr(a, b, h_min, k, grid_mass);

```

```

FILE *out_f = fopen(file_name.c_str(), "w");

fprintf(out_f, "%d\n", n);
for(int i = 0; i < n; i++){
    fprintf(out_f, "%.15lf\n", grid_mass[i]);
}

delete[] grid_mass;
fclose(out_f);

return true;
}

bool grid_generator::generate_reg_grid(double a, double b, double h, int &n, double *&grid_mass){
    if(a > b || h <= 0) return false; // проверка данных

    n = generate_reg_grid_pr(a, b, h, grid_mass);

    return true;
}

bool grid_generator::generate_reg_grid(double a, double b, double h, dvector &grid_vector){
    if(a > b || h <= 0) return false; // проверка данных

    double *grid_mass;

    int n = generate_reg_grid_pr(a, b, h, grid_mass);

    grid_vector.clear();
    for(int i = 0; i < n; i++)
        grid_vector.push_back(grid_mass[i]);

    delete[] grid_mass;

    return true;
}

bool grid_generator::generate_reg_grid(double a, double b, double h, std::string file_name){
    if(a > b || h <= 0) return false; // проверка данных

    double *grid_mass;
    int n = generate_reg_grid_pr(a, b, h, grid_mass);

    FILE *out_f = fopen(file_name.c_str(), "w");

    fprintf(out_f, "%d\n", n);
    for(int i = 0; i < n; i++){
        fprintf(out_f, "%.15lf\n", grid_mass[i]);
    }

    delete[] grid_mass;
    fclose(out_f);

    return true;
}

bool grid_generator::generate_unreg_grid_ins(double a, double b, double h_min, double k, string
file_name){
    double k1 = sqrt(k);
    double h_min1 = h_min/(1+k1);
    return generate_unreg_grid(a, b, h_min1, k1, file_name);
}

```

```

bool grid_generator::generate_unreg_grid_ins(double a, double b, double h_min, double k, dvector
&grid_vector){
    double k1 = sqrt(k);
    double h_min1 = h_min/(1+k1);
    return generate_unreg_grid(a, b, h_min1, k1, grid_vector);
}

bool grid_generator::generate_unreg_grid_ins(double a, double b, double h_min, double k, int &n,
double*& grid_mass){
    double k1 = sqrt(k);
    double h_min1 = h_min/(1+k1);
    return generate_unreg_grid(a, b, h_min1, k1, n, grid_mass);
}

bool grid_generator::generate_reg_grid_ins(double a, double b, double h, string file_name){
    return generate_reg_grid(a, b, h/2, file_name);
}

bool grid_generator::generate_reg_grid_ins(double a, double b, double h, dvector &grid_vector){
    return generate_reg_grid(a, b, h/2, grid_vector);
}

bool grid_generator::generate_reg_grid_ins(double a, double b, double h, int &n, double*&
grid_mass){
    return generate_reg_grid(a, b, h/2, n, grid_mass);
}

// FEM.cpp
#include "FEM.h"

void FEM::init(string node_file, string u0_file, string info_file){
    FILE* node_f = fopen(node_file.c_str(), "r");

    // Считывание узлов
    fscanf(node_f, "%d", &nodes_n);
    nodes = new double [nodes_n];

    for(int i = 0; i < nodes_n; i++)
        fscanf(node_f, "%lf", &nodes[i]);

    fclose(node_f);

    // Генерация элементов
    elements_n = nodes_n - 1;
    elements = new element [elements_n];

    for(int i = 0; i < elements_n; i++){
        elements[i].node1 = i;
        elements[i].node2 = i+1;
    }

    // Инициализация начальных условий
    q_last = new double [nodes_n];
    q_new = new double [nodes_n];
    q_temp = new double [nodes_n];
    q_min = new double [nodes_n];
    q_min_rp = new double [nodes_n];
    q_min_R = new double [nodes_n];

    FILE* u0_f = fopen(u0_file.c_str(), "r");
}

```

```

for(int i = 0; i < nodes_n; i++)
    fscanf(u0_f, "%lf", &q_last[i]);
fclose(u0_f);

// Инициализация информации о сетки по времени
FILE* info_f = fopen(info_file.c_str(), "r");
fscanf(info_f, "%lf %lf %lf %d %lf", &t_now, &h_t, &k_t, &iter_max, &eps);
fclose(info_f);

// Инициализация матрицы
dia_0 = new double [nodes_n];
dia_1 = new double [nodes_n - 1];
dia_2 = new double [nodes_n - 1];
}

void FEM::init_coefs(func_t set_lambda, func_t set_f){
    lambda = set_lambda;
    f = set_f;
}

void FEM::init_sigma(double set_c){
    sigma_c = set_c;
}

void FEM::set_analytic(func_t set_u){
    analytic = set_u;
}

void FEM::init_bounds(int set_b_l, int set_b_r, func bound_l1, func bound_l2, func bound_r1,
func bound_r2){
    bound_type_left = set_b_l;
    bound_type_right = set_b_r;

    // Левые краевые
    switch(bound_type_left){
        case 1: case 2: {
            bounds_l = new func [1];
            bounds_l[0] = bound_l1;
        }break;
        case 3: {
            bounds_l = new func [2];
            bounds_l[0] = bound_l1;
            bounds_l[1] = bound_l2;
        }break;
    };
}

// Правые краевые
switch(bound_type_right){
    case 1: case 2: {
        bounds_r = new func [1];
        bounds_r[0] = bound_r1;
    }break;
    case 3: {
        bounds_r = new func [2];
        bounds_r[0] = bound_r1;
        bounds_r[1] = bound_r2;
    }break;
};

double FEM::get_current_time(){
    return t_now;
}

```

```

}

double FEM::sigma(double u_x, double x){
    return (u_x*u_x + sigma_c)*x;
}

double FEM::sigma_ux(double u_x, double x){
    return 2*x*u_x;
}

void FEM::simple_iteration_step()
{
    // В качестве начального приближения берём решение на предыдущем слое
    for(int i = 0; i < nodes_n; i++)
        q_new[i] = q_last[i];

    t_new = t_now + h_t;
    bool flag = false;
    for(int iter = 0; iter < iter_max && !flag; iter++)
    {
        // Обнуление матрицы
        for(int i = 0; i < elements_n; i++)
        {
            dia_0[i] = 0;
            dia_1[i] = 0;
            dia_2[i] = 0;
            q_temp[i] = 0;
        }
        dia_0[nodes_n-1] = 0;
        q_temp[nodes_n-1] = 0;

        // Сборка глобальной матрицы
        matrix_gen_si(q_new, q_temp);

        // Решаем СЛАУ
        solve_LU(dia_0, dia_1, dia_2, q_temp, nodes_n);

        // Вычисление параметра релаксации
        double w = w_min();
        for(int i = 0; i < nodes_n; i++)
            q_temp[i] = w*q_temp[i] + (1-w)*q_new[i];

        // Проверка достижения необходимой точности
        // В q_min_R запишем вектор невязки, в q_new - вектор правой части
        for(int i = 0; i < nodes_n; i++)
            q_new[i] = 0;

        // Обнуление матрицы
        for(int i = 0; i < elements_n; i++)
        {
            dia_0[i] = 0;
            dia_1[i] = 0;
            dia_2[i] = 0;
        }
        dia_0[nodes_n-1] = 0;

        matrix_gen_si(q_temp, q_new);

        // Вычисление невязки: A(q)*q - b(q)
        q_min_R[0] = dia_0[0]*q_temp[0] + dia_1[0]*q_temp[1];
        q_min_R[elements_n] = dia_0[elements_n]*q_temp[elements_n] + dia_2[elements_n-1]*q_temp[elements_n-1];
    }
}

```

```

        for(int i = 1; i < elements_n; i++)
            q_min_R[i] = dia_0[i]*q_temp[i] + dia_1[i]*q_temp[i+1] + dia_2[i-
1]*q_temp[i-1];

        double norm1 = norm_q_dif(q_min_R, q_new);
        double norm2 = norm_q(q_new);
        double nev = norm1 / norm2;

        if(nev < eps)
            flag = true;

        double *ch_p;
        ch_p = q_new;
        q_new = q_temp;
        q_temp = ch_p;
        printf("SI\tTime:\t%.15lf\tIter:\t%d\tNev:\t%.3e\tRelax:\t%.3e\r", t_new, iter,
nev, w);
    }
    printf("\n");
    // Переход на новый слой
    double* ch_pt;
    ch_pt = q_new;
    q_new = q_last;
    q_last = ch_pt;
    t_now = t_new;
    h_t *= k_t;
}

void FEM::Newton_step()
{
    // В качестве начального приближения берём решение на предыдущем слое
    for(int i = 0; i < nodes_n; i++)
        q_new[i] = q_last[i];

    t_new = t_now + h_t;
    bool flag = false;
    for(int iter = 0; iter < iter_max && !flag; iter++)
    {
        // Обнуление матрицы
        for(int i = 0; i < elements_n; i++){
            dia_0[i] = 0;
            dia_1[i] = 0;
            dia_2[i] = 0;
            q_temp[i] = 0;
        }
        dia_0[nodes_n-1] = 0;
        q_temp[nodes_n-1] = 0;

        // Сборка глобальной матрицы
        matrix_gen_Newton(q_new, q_temp);

        // Решаем СЛАУ
        solve_LU(dia_0, dia_1, dia_2, q_temp, nodes_n);

        // Релаксация
        double w = w_min();
        for(int i = 0; i < nodes_n; i++)
            q_temp[i] = w*q_temp[i] + (1-w)*q_new[i];

        // Проверка - нужно ли выходит из цикла
        // В q_min_R запишем вектор невязки, в q_new - вектор правой части
    }
}

```

```

        for(int i = 0; i < nodes_n; i++)
            q_new[i] = 0;

        // Обнуление матрицы
        for(int i = 0; i < elements_n; i++){
            dia_0[i] = 0;
            dia_1[i] = 0;
            dia_2[i] = 0;
        }
        dia_0[nodes_n-1] = 0;

        matrix_gen_si(q_temp, q_new);

        // Вычисление невязки: A(q)*q - b(q)
        q_min_R[0] = dia_0[0]*q_temp[0] + dia_1[0]*q_temp[1];
        q_min_R[elements_n] = dia_0[elements_n]*q_temp[elements_n] + dia_2[elements_n-1]*q_temp[elements_n-1];

        for(int i = 1; i < elements_n; i++)
            q_min_R[i] = dia_0[i]*q_temp[i] + dia_1[i]*q_temp[i+1] + dia_2[i-1]*q_temp[i-1];

        double norm1 = norm_q_dif(q_min_R, q_new);
        double norm2 = norm_q(q_new);
        double nev = norm1/norm2;

        if(nev < eps)
            flag = true;

        // Переброс указателей
        double *ch_p;
        ch_p = q_new;
        q_new = q_temp;
        q_temp = ch_p;

        printf("Ne\tTime:\t%.15lf\tIter:\t%d\tNev:\t%.3e\tRelax:\t%.3e\r", t_new, iter,
nev, w);
    }

    printf("\n");

    // Переходим на новый слой
    double* ch_pt;
    ch_pt = q_new;
    q_new = q_last;
    q_last = ch_pt;

    t_now = t_new;
    h_t *= k_t;
}

void FEM::matrix_gen_si(double* q_n, double* q_t)
{
    // Локальная матрица и вектор правой части
    double A_loc[2][2], b_loc[2];

    // Генерация матрицы
    for(int i = 0; i < elements_n; i++){
        double x1 = nodes[elements[i].node1]; // левый узел для элемента
        double x2 = nodes[elements[i].node2]; // правый узел для элемента

        double h_k = x2 - x1; // шаг по сетки

```

```

        double x_midd = (x2+x1)/2; // центральная точка сетки

        double G_11_val, M_12_val; //значения элементов в матирце жескости и массы

        G_11_val = (lambda(x1, t_new) + lambda(x2, t_new))/(2*h_k);
        double gamma_aver = sigma((q_n[i+1] - q_n[i])/h_k, x_midd)/h_t;
        M_12_val = gamma_aver * h_k / 6;

        A_loc[0][0] = A_loc[1][1] = G_11_val + 2 * M_12_val;
        A_loc[1][0] = A_loc[0][1] = -G_11_val + M_12_val;

        double f1 = f(x1, t_new), f2 = f(x2, t_new); // значения правой части
        b_loc[0] = h_k*(2*f1 + f2)/6.0 + M_12_val * (2*q_last[i] + q_last[i+1]);
        b_loc[1] = h_k*(f1 + 2*f2)/6.0 + M_12_val * (q_last[i] + 2*q_last[i+1]);

        dia_0[i] += A_loc[0][0];
        dia_0[i+1] += A_loc[1][1];
        dia_1[i] += A_loc[0][1];
        dia_2[i] += A_loc[1][0];

        q_t[i] += b_loc[0];
        q_t[i+1] += b_loc[1];

    }

// учёт краевых условий
// Левых
switch(bound_type_left){
    case 1:{
        dia_0[0] = 1;
        dia_1[0] = 0;
        q_t[0] = bounds_l[0](t_new);
    }break;

    case 2:{
        q_t[0] += bounds_l[0](t_new);
    }break;

    case 3:{
        dia_0[0] += bounds_l[0](t_now);
        q_t[0] += bounds_l[0](t_new) * bounds_l[1](t_new);
    }break;
};

// Правых
switch(bound_type_right){
    case 1:{
        dia_0[nodes_n-1] = 1;
        dia_2[nodes_n-2] = 0;
        q_t[nodes_n-1] = bounds_r[0](t_new);
    }break;

    case 2:{
        q_t[nodes_n-1] += bounds_r[0](t_new);
    }break;

    case 3:{
        dia_0[nodes_n-1] += bounds_r[0](t_new);
        q_t[nodes_n-1] += bounds_r[0](t_new) * bounds_r[1](t_new);
    }break;
};

```

```

}

void FEM::matrix_gen_Newton(double* q_n, double* q_t)
{
    // Локальная матрица и вектор правой части
    double A_loc[2][2], b_loc[2];

    // Генерация матрицы
    for(int i = 0; i < elements_n; i++){
        double x1 = nodes[elements[i].node1]; // левый узел для элемента
        double x2 = nodes[elements[i].node2]; // правый узел для элемента

        double h_k = x2 - x1; // шаг по сетки
        double x_midd = (x2+x1)/2; // центральная точка сетки

        double G_11_val, M_12_val; // значения элементов в матирце жескости и массы

        G_11_val = (lambda(x1, t_new) + lambda(x2, t_new))/(2*h_k);
        double gamma_aver = sigma((q_n[i+1] - q_n[i])/h_k, x_midd)/h_t;
        M_12_val = gamma_aver * h_k / 6;

        // Среднее значение производной
        double sigm_ux_aver = sigma((q_n[i+1] - q_n[i])/h_k, x_midd);

        // Базовые значения
        A_loc[0][0] = A_loc[1][1] = G_11_val + 2 * M_12_val;
        A_loc[1][0] = A_loc[0][1] = -G_11_val + M_12_val;
        double reg_par = 1.0;

        // Прибавки от метода Ньютона
        A_loc[0][0] += reg_par * sigm_ux_aver * (-q_n[i]/3.0 - q_n[i+1]/6.0 + (2*q_last[i] + q_last[i+1])/6.0) / h_t;
        A_loc[0][1] += reg_par * sigm_ux_aver * (q_n[i]/3.0 + q_n[i+1]/6.0 - (2*q_last[i] + q_last[i+1])/6.0) / h_t;
        A_loc[1][0] += reg_par * sigm_ux_aver * (-q_n[i]/6.0 - q_n[i+1]/3.0 + (q_last[i] + 2*q_last[i+1])/6.0) / h_t;
        A_loc[1][1] += reg_par * sigm_ux_aver * (q_n[i]/6.0 + q_n[i+1]/3.0 - (q_last[i] + 2*q_last[i+1])/6.0) / h_t;

        double f1 = f(x1, t_new), f2 = f(x2, t_new); // значения правой части

        // Базовые значения
        b_loc[0] = h_k*(2*f1 + f2)/6.0 + M_12_val * (2*q_last[i] + q_last[i+1]);
        b_loc[1] = h_k*(f1 + 2*f2)/6.0 + M_12_val * (q_last[i] + 2*q_last[i+1]);

        //Прибавки от метода Ньютона

        b_loc[0] += reg_par * sigm_ux_aver * (-q_n[i]*q_n[i]/3.0 + q_n[i]*q_n[i+1]/6.0 + q_n[i+1]*q_n[i+1]/6.0 + (2*q_last[i] + q_last[i+1])*(q_n[i]-q_n[i+1])/6.0) / h_t;
        b_loc[1] += reg_par * sigm_ux_aver * (-q_n[i]*q_n[i]/6.0 - q_n[i]*q_n[i+1]/6.0 + q_n[i+1]*q_n[i+1]/3.0 + (q_last[i] + 2*q_last[i+1])*(q_n[i]-q_n[i+1])/6.0) / h_t;

        dia_0[i] += A_loc[0][0];
        dia_0[i+1] += A_loc[1][1];
        dia_1[i] += A_loc[0][1];
        dia_2[i] += A_loc[1][0];

        q_t[i] += b_loc[0];
        q_t[i+1] += b_loc[1];
    }
}

```

```

}

// Учёт краевых условий
// Левых
switch(bound_type_left){
    case 1:{  

        dia_0[0] = 1;  

        dia_1[0] = 0;  

        q_t[0] = bounds_l[0](t_new);  

    }break;  

  

    case 2:{  

        q_t[0] += bounds_l[0](t_new);  

    }break;  

  

    case 3:{  

        dia_0[0] += bounds_l[0](t_now);  

        q_t[0] += bounds_l[0](t_new) * bounds_l[1](t_new);  

    }break;  

};

// Правых
switch(bound_type_right){
    case 1:{  

        dia_0[nodes_n-1] = 1;  

        dia_2[nodes_n-2] = 0;  

        q_t[nodes_n-1] = bounds_r[0](t_new);  

    }break;  

  

    case 2:{  

        q_t[nodes_n-1] += bounds_r[0](t_new);  

    }break;  

  

    case 3:{  

        dia_0[nodes_n-1] += bounds_r[0](t_new);  

        q_t[nodes_n-1] += bounds_r[0](t_new) * bounds_r[1](t_new);  

    }break;  

};

double FEM::norm_q_dif(double *q1, double *q2){  

    double norm = 0;  

    for(int i = 0; i < nodes_n; i++)  

        norm += (q1[i] - q2[i])*(q1[i] - q2[i]);  

  

    norm = sqrt(norm);  

    return norm;
}

double FEM::norm_q(double *q1){  

    double norm = 0;  

    for(int i = 0; i < nodes_n; i++)  

        norm += q1[i]*q1[i];  

  

    norm = sqrt(norm);  

    return norm;
}

void FEM::out_q(string file_out){  

    FILE* out_f = fopen(file_out.c_str(), "a");
}

```

```

double diff = 0;
double u_norm = 0;

for(int i = 0; i < nodes_n; i++){
    double u_val = analytic(nodes[i], t_now);
    diff += (q_last[i]-u_val)*(q_last[i]-u_val);
    u_norm += u_val*u_val;
}

diff = sqrt(diff/u_norm);
fprintf(out_f, "Diff:\t%.3e\n", diff);
fclose(out_f);
}

double FEM::w_min_func(double w){
    // Обнуление матрицы и вычисление аргумента
    for(int i = 0; i < elements_n; i++){
        dia_0[i] = 0;
        dia_1[i] = 0;
        dia_2[i] = 0;
        q_min[i] = w*q_temp[i] + (1-w)*q_new[i];
        q_min_rp[i] = 0;
    }
    dia_0[nodes_n-1] = 0;
    q_min[nodes_n-1] = w*q_temp[nodes_n-1] + (1-w)*q_new[nodes_n-1];
    q_min_rp[nodes_n-1] = 0;

    matrix_gen_si(q_min, q_min_rp);

    // Вычисление невязки: A(q_min)*q_min - b(q_min)
    q_min_R[0] = dia_0[0]*q_min[0] + dia_1[0]*q_min[1] - q_min_rp[0];
    q_min_R[elements_n] = dia_0[elements_n]*q_min[elements_n] + dia_2[elements_n-1]*q_min[elements_n-1] - q_min_rp[elements_n];

    for(int i = 1; i < elements_n; i++)
        q_min_R[i] = dia_0[i]*q_min[i] + dia_1[i]*q_min[i+1]+dia_2[i-1]*q_min[i-1] - q_min_rp[i];
}

return norm_q(q_min_R);
}

double FEM::w_min()
{
    double a, b; // Отрезок минимизации
    w_min_find_area(a, b, 1E-3); // Ищем отрезок
    return w_min_golden(a, b, 1E-5);
}

void FEM::w_min_find_area(double &a, double &b, double delta)
{
    // Инициализация
    double x0 = 1;
    double f0 = w_min_func(x0);
    double x1 = x0-delta;
    double f1 = w_min_func(x1);
    double h = -delta;
    int k = 1;

    // Определяем сторону, куда идти
}

```

```

if(f0 < f1){
    x1 = x0 + delta;
    h *= -1;
}

bool end_cycle = false;

while(!end_cycle){
    h *= 2;
    x0 = x1 + h;
    f0 = w_min_func(x0);
    k++;

    if(f1 > f0){
        x1 = x0;
        f1 = f0;
    }
    else{
        end_cycle = true;
        x1 = x0;
        x0 -= h + h/2;
    }
}

if(x0 > x1){
    a = x1;
    b = x0;
}
else{
    a = x0;
    b = x1;
}
}

double FEM::w_min_golden(double a, double b, double eps_min)
{
    double x1, x2, f1, f2;
    const double mul = (3-sqrt((double)5.0))/2;

    int point_num;
    x1 = a + mul*(b-a);
    x2 = b - mul*(b-a);
    f1 = w_min_func(x1);
    f2 = w_min_func(x2);

    if(f1 < f2){
        b = x2;
        x2 = x1;
        f2 = f1;
        point_num = 1;
    }
    else{
        a = x1;
        x1 = x2;
        f1 = f2;
        point_num = 2;
    }

    while(fabs(a-b)>eps_min){
        switch(point_num){

```

```

        case 1:{  

            x1 = a + mul*(b-a);  

            f1 = w_min_func(x1);  

            }break;  

        case 2:{  

            x2 = b - mul*(b-a);  

            f2 = w_min_func(x2);  

            }break;  

    };  

  

    if(f1 < f2){  

        b = x2;  

        x2 = x1;  

        f2 = f1;  

        point_num = 1;  

    }  

    else{  

        a = x1;  

        x1 = x2;  

        f1 = f2;  

        point_num = 2;  

    }  

}  

return (a+b)/2;  

}

```