МИНИСТЕРСТВО НАУКИ
И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

НГТУ
НЭТИ | **Факультет прикладной математики и информатики**

Кафедра прикладной математики

Практическое задание № 2

по дисциплине «Уравнения математической физики»

РЕШЕНИЕ НЕЛИНЕЙНЫХ НАЧАЛЬНО-КРАЕВЫХ ЗАДАЧ

Группа ПМ-01 БУДНИК СВЕТЛАНА

САМСОНОВ СЕМЁН

Преподаватель ЗАДОРОЖНЫЙ АЛЕКСАНДР ГЕННАДЬЕВИЧ

Новосибирск, 2023

Цель работы:

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

Условие задания:

1. Выполнить конечноэлементную аппроксимацию исходного уравнения в соответствии с заданием. Получить формулы для вычисления компонент матрицы A и вектора правой части b для метода простой итерации.
2. Реализовать программу решения нелинейной задачи методом простой итерации с учетом следующих требований:
   - язык программирования C++ или Фортран;
   - предусмотреть возможность задания неравномерных сеток по пространству и по времени, разрывность параметров уравнения по подобластям, учет краевых условий;
   - матрицу хранить в ленточном формате, для решения СЛАУ использовать метод LU-разложения;
   - предусмотреть возможность использования параметра релаксации.
3. Выполнить линеаризацию нелинейной системы алгебраических уравнений с использованием метода Ньютона. Получить формулы для вычисления компонент линеаризованных матрицы A^L и вектора правой части b^L.
4. Реализовать программу решения нелинейной задачи методом Ньютона.
5. Протестировать разработанные программы.
6. Исследовать реализованные методы на различных зависимостях коэффициента от решения (или производной решения) в соответствии с заданием. На одних и тех же задачах сравнить по количеству итераций метод простой итерации и метод Ньютона. Исследовать скорость сходимости от параметра релаксации.

Вариант 5:

Эллиптическое нелинейное уравнение:

$$-div\left(\lambda(u,x)\ grad\big(u(x,t)\big)\right)+\sigma(x)\frac{\partial u}{\partial t}=f(x,t),$$

$$\lambda=\lambda(u,x),$$
$$\sigma=\sigma(x),$$
$$f=f(x,t),$$
$$u=u(x,t).$$

Вывод формул

Базисные функции

Базисные функции линейные:

$$\begin{aligned} \psi_1(x) &= \frac{x_{k+1} - x}{h_{x_k}} = \frac{x_k + h_{x_k} - x}{h_{x_k}}, \\ \psi_2(x) &= \frac{x - x_k}{h_{x_k}}, \\ h_{x_k} &= x_{k+1} - x_k. \end{aligned}$$

Производные этих функций будут выглядеть так:

$$\begin{aligned} \frac{d\psi_1(x)}{dx} &= -\frac{1}{h_{x^k}}, \\ \frac{d\psi_2(x)}{dx} &= \frac{1}{h_{x^k}}. \end{aligned}$$

Матрица жёсткости

Локальная матрица жёсткости считается по следующей формуле:

$$\hat{G}_{ij}^k = \int_{\Omega_k} \lambda^k\,(u(x,t),x)\ grad(\psi_i) grad\big(\psi_j\big)\ d\Omega_k$$

Разложим λ^k по базисным векторам:

$$\lambda^k(u(x,t),x) = \lambda^k(u(x^k,t),x^k)\cdot\psi_1 + \lambda^k(u(x^{k+1},t),x^{k+1})\cdot\psi_2$$

Подставим данное разложение в интеграл локальной матрицы:

$$\hat{G}_{ij}^k = \int_{\Omega_k} \lambda^k\,(u(x,t),x)\ grad(\psi_i) grad\big(\psi_j\big)\ d\Omega_k =$$

$$= \int_{x^k}^{x^{k+1}} (\lambda^k(u(x^k,t),x^k)\cdot\psi_1 + \lambda^k(u(x^{k+1},t),x^{k+1})\cdot\psi_2)\frac{d\psi_i}{dx}\frac{d\psi_j}{dx}\ dx$$

Учитывая производные и то, что $\int_{x^k}^{x^{k+1}} \psi_i\,(x)dx = \frac{h_{x^k}}{2}$, то итоговая формула локальной матрицы жёсткости будет выглядеть следующим образом:

$$\hat{G}^k = \frac{\lambda^k(u(x^k,t),x^k) + \lambda^k(u(x^{k+1},t),x^{k+1})}{2h_{x^k}}\cdot\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$$

Матрица массы

Общая формула локальной матрицы массы имеет следующий вид:

$$\hat{M}_{ij}^k = \int_{\Omega^k} \sigma^k\,(x)\psi_i\psi_j\ d\Omega^k$$

Коэффициент σ вынесем из интеграла как усреднённое значение на конечном элементе:

$$\hat{M}_{ij}^k = \bar{\sigma}^k \int_{\Omega^k} \psi_i\,\psi_j\ d\Omega^k$$

И тогда получим следующее выражение для нахождение локальной матрицы масс:

$$\hat{M}^k = \frac{\bar{\sigma} h_{x^k}}{6}\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$$

Вектор правой части

Локальный вектор правой части $\hat{b}$ будем находить как произведение матрицы массы $\hat{M}|_{\sigma=1} = \hat{C}$ на вектор значений функции $f(x,t)$:

$$\hat{b}^k = \hat{C} \cdot \{f(x^k,t), f(x^{k+1},t)\} = \frac{h_{x^k}}{6}\begin{pmatrix} 2f(x^k,t) + f(x^{k+1},t) \\ f(x^k,t) + 2f(x^{k+1},t) \end{pmatrix}$$

Подбор лямбды

Метод простых итераций

Алгоритм метода простых итераций:

1. Выбираем начальное приближение q^0
2. Подставляем его в λ везде, считаем матрицу жёсткости
3. Собираем глобальную матрицу A и вектор правой части b, решаем СЛАУ, находим q^1 (и q^k далее)
4. Повторяем с п.2, пока не получим $\frac{\|A(q^k)\cdot q^k - b\|}{\|b\|} < \varepsilon$

Метод релаксации

Алгоритм метода релаксации будет таким же, как и алгоритм метода простых итераций, за исключением шага 3. В методе релаксации, на шаге 3 после решения СЛАУ мы получим q^{k*}, а вектор q^k будем находить по следующей формуле:

$$q^k = \omega q^{k*} + (1-\omega) q^{k-1},$$

где $\omega \in (0;1]$.

Метод Ньютона

Метод Ньютона - тоже итерационный алгоритм, но при этом использующий при работе производные матрицы и вектора правой части. В нашем случае, вектор правой части не зависит от решения (линеен), поэтому трудности будут только с матрицей (причём, только с матрицей жёсткости).

В данном методе матрицу и вектор правой части представляют в линеаризованном виде как первые 2 члена разложения в ряд Тейлора относительно точки вектора на прошлой итерации:

$$A_{ij}(q)q_j \approx A_{ij}(q^0)q_j^0 + \sum_r \left.\frac{\partial\left(A_{ij}(q)q_j\right)}{\partial q_r}\right|_{q=q^0} (q_r - q_r^0)$$

Если немного раскрыть это уравнение и перейти на локальные матрицы, то получится следующее:

$$\hat{A}_{ij}(\hat{q})\hat{q}_j \approx \hat{A}_{ij}(\hat{q}^0)\hat{q}_j + \sum_r^{\hat{n}} \left.\frac{\partial \hat{A}_{ij}(\hat{q})}{\partial \hat{q}_r}\right|_{\hat{q}=\hat{q}^0} \hat{q}_j^0(\hat{q}_r - \hat{q}_r^0)$$

В нашем случае $\hat{n} = 2$. Поскольку в данном уравнении мы не можем просто взять и вычленить $\hat{q}_j$ компоненту, то придётся позаниматься небольшой перестановкой членов системы $A(q)q = b$, и получить в результате систему $A^L(q)q = b^L$. Для начала распишем первую строку системы $A(q)q = b$:

$$\hat{b}_1 = \hat{A}_{11}(\hat{q})\hat{q}_1 + \hat{A}_{12}(\hat{q})\hat{q}_2 = \sum_{j=1}^{2}\left(\hat{A}_{1j}(\hat{q})\hat{q}_j + \sum_{r=1}^{2}\left(\left.\frac{\partial \hat{A}_{1j}(\hat{q})}{\partial \hat{q}_r}\right|_{\hat{q}=\hat{q}^0} \hat{q}_j^0(\hat{q}_r - \hat{q}_r^0)\right)\right) =$$

$$= \hat{A}_{11}(\hat{q}^0)\hat{q}_1 + \hat{q}_1^0 \sum_{r=1}^{2} \left.\frac{\partial \hat{A}_{11}(\hat{q})}{\partial \hat{q}_r}\right|_{\hat{q}=\hat{q}^0} (\hat{q}_r - \hat{q}_r^0) +$$

$$+\hat{A}_{12}(\hat{q}^0)\hat{q}_2 + q_2^0 \sum_{r=1}^{2} \left.\frac{\partial \hat{A}_{12}(\hat{q})}{\partial \hat{q}_r}\right|_{\hat{q}=\hat{q}_0} (\hat{q}_r - \hat{q}_r^0) =$$

$$= \hat{A}_{11}(\hat{q}^0)\hat{q}_1 + \hat{q}_1^0 \sum_{r=1}^{2} \left.\frac{\partial \hat{A}_{11}(\hat{q})}{\partial \hat{q}_r}\right|_{\hat{q}=\hat{q}^0} \hat{q}_r - \hat{q}_1^0 \sum_{r=1}^{2} \left.\frac{\partial \hat{A}_{11}(\hat{q})}{\partial \hat{q}_r}\right|_{\hat{q}=\hat{q}^0} \hat{q}_r^0 +$$

$$+\hat{A}_{12}(\hat{q}^0)\hat{q}_2 + \hat{q}_2^0 \sum_{r=1}^{2} \left.\frac{\partial \hat{A}_{12}(\hat{q})}{\partial \hat{q}_r}\right|_{\hat{q}=\hat{q}^0} \hat{q}_r - \hat{q}_2^0 \sum_{r=1}^{2} \left.\frac{\partial \hat{A}_{12}(\hat{q})}{\partial \hat{q}_r}\right|_{\hat{q}=\hat{q}^0} \hat{q}_r^0 =$$

$$= \hat{A}_{11}(\hat{q}^0)\hat{q}_1 + \hat{q}_1^0 \left.\frac{\partial \hat{A}_{11}(\hat{q})}{\partial \hat{q}_1}\right|_{\hat{q}=\hat{q}^0} \hat{q}_1 + \hat{q}_1^0 \left.\frac{\partial \hat{A}_{11}(\hat{q})}{\partial \hat{q}_2}\right|_{\hat{q}=\hat{q}^0} \hat{q}_2 - \hat{q}_1^0 \sum_{r=1}^{2} \left.\frac{\partial \hat{A}_{11}(\hat{q})}{\partial \hat{q}_r}\right|_{\hat{q}=\hat{q}^0} \hat{q}_r^0 +$$

$$+\hat{A}_{12}(\hat{q}^0)\hat{q}_2 + \hat{q}_2^0 \left.\frac{\partial \hat{A}_{12}(\hat{q})}{\partial \hat{q}_1}\right|_{\hat{q}=\hat{q}^0} \hat{q}_1 + \hat{q}_2^0 \left.\frac{\partial \hat{A}_{12}(\hat{q})}{\partial \hat{q}_2}\right|_{\hat{q}=\hat{q}^0} \hat{q}_2 - \hat{q}_2^0 \sum_{r=1}^{2} \left.\frac{\partial \hat{A}_{12}(\hat{q})}{\partial \hat{q}_r}\right|_{\hat{q}=\hat{q}^0} \hat{q}_r^0 =$$

$$= \hat{q}_1\left(\hat{A}_{11}(\hat{q}^0) + \sum_{j=1}^{2} \left.\frac{\partial \hat{A}_{1j}(\hat{q})}{\partial \hat{q}_1}\right|_{\hat{q}=\hat{q}_0} \hat{q}_j^0\right) + \hat{q}_2\left(\hat{A}_{12}(\hat{q}^0) + \sum_{j=1}^{2} \left.\frac{\partial \hat{A}_{1j}(\hat{q})}{\partial \hat{q}_2}\right|_{\hat{q}=\hat{q}^0} \hat{q}_j^0\right) -$$

$$- \sum_{j=1}^{2} \hat{q}_j^0 \cdot \left(\sum_{r=1}^{2} \frac{\partial \hat{A}_{1j}(\hat{q})}{\partial \hat{q}_r} \hat{q}_r^0\right).$$

Таким образом, теперь мы можем выразить формулы для получения элементов первой строки матрицы $\hat{A}^L$ и вектора $\hat{b}^L$:

$$\hat{A}_{11}^L(\hat{q}^0) = \hat{A}_{11}(\hat{q}^0) + \sum_{j=1}^{2} \left.\frac{\partial \hat{A}_{1j}(\hat{q})}{\partial \hat{q}_1}\right|_{\hat{q}=\hat{q}^0} \hat{q}_j^0,$$

$$\hat{A}_{12}^L(\hat{q}^0) = \hat{A}_{12}(\hat{q}^0) + \sum_{j=1}^{2} \left.\frac{\partial \hat{A}_{1j}(\hat{q})}{\partial \hat{q}_2}\right|_{\hat{q}=\hat{q}^0} \hat{q}_j^0,$$

$$\hat{b}_1^L = \hat{b}_1 + \sum_{j=1}^{2} \hat{q}_j^0 \cdot \left(\sum_{r=1}^{2} \frac{\partial \hat{A}_{1j}(\hat{q})}{\partial \hat{q}_r} \hat{q}_r^0 \right).$$

Аналогично будет и для остальных (произвольных) строк:

$$\hat{A}_{ij}^L(\hat{q}^0) = \hat{A}_{ij}(\hat{q}^0) + \sum_{k=1}^{2} \left. \frac{\partial \hat{A}_{ik}(\hat{q})}{\partial \hat{q}_j} \right|_{\hat{q}=\hat{q}^0} \hat{q}_k^0,$$

$$\hat{b}_i^L = \hat{b}_i + \sum_{j=1}^{2} \hat{q}_j^0 \cdot \left(\sum_{r=1}^{2} \frac{\partial \hat{A}_{ij}(\hat{q})}{\partial \hat{q}_r} \hat{q}_r^0 \right) =$$

$$= \hat{b}_i + \hat{q}_1^0 \cdot \left(\sum_{r=1}^{2} \frac{\partial \hat{A}_{i1}(\hat{q})}{\partial \hat{q}_r} \hat{q}_r^0 \right) + \hat{q}_2^0 \cdot \left(\sum_{r=1}^{2} \frac{\partial \hat{A}_{i2}(\hat{q})}{\partial \hat{q}_r} \hat{q}_r^0 \right) =$$

$$= \hat{b}_i + \hat{q}_1^0 \cdot \left(\sum_{k=1}^{2} \frac{\partial \hat{A}_{ik}(\hat{q})}{\partial \hat{q}_1} \hat{q}_k^0 \right) + \hat{q}_2^0 \cdot \left(\sum_{k=1}^{2} \frac{\partial \hat{A}_{ik}(\hat{q})}{\partial \hat{q}_2} \hat{q}_k^0 \right).$$

Введём понятие добавочной матрицы $\hat{A}^*$ к линеаризованной матрице $\hat{A}^L$:

$$\hat{A}_{ij}^* = \sum_{k=1}^{2} \left. \frac{\partial \hat{A}_{ik}(\hat{q})}{\partial \hat{q}_j} \right|_{\hat{q}=\hat{q}^0} \hat{q}_k^0$$

Тогда матрица $\hat{A}_{ij}^L$ будет считаться следующим образом:

$$\hat{A}_{ij}^L = \hat{A}_{ij} + \hat{A}_{ij}^*,$$

а вектор правой части $\hat{b}_i^L$ можно будет найти следующим образом:

$$\hat{b}_i^L = \hat{b}_i + \hat{q}_1^0 \cdot \hat{A}_{i1}^* + \hat{q}_2^0 \cdot \hat{A}_{i2}^*$$

Итерационный процесс совпадает с методом простых итераций. В том числе и условие выхода из итерационного процесса.

Важное примечание по выходу из итерационного процесса метода Ньютона: условием выхода будет являться уравнение с системой $Aq = b$, а не $A^L q = b^L$!

Релаксация метода Ньютона

Метод Ньютона работает особенно хорошо, если подбирать параметр релаксации на каждой итерации. В целом, его можно подбирать при помощи одномерного поиска, минимизируя относительную невязку решения, но можно подбирать и следующим способом:

5. Получаем решение и относительную невязку этого решения при $\omega = 1$
6. Если относительная невязка полученного решения оказалась больше, чем на прошлой итерации, то
    1. Уменьшаем коэффициент релаксации на 0.1
    2. Пересчитываем решение и его относительную невязку
    3. Если относительная невязка всё ещё больше требуемого, то возвращаемся в п. 2.1.

Демпфирование метода Ньютона

Как сказано в учебнике на странице 837, матрица A^L может перестать быть положительно определённой из-за добавочной матрицы. Поэтому вводят так называемый *коэффициент демпфирования* $\nu \in (0; 1]$, который ставится перед добавками следующим образом:

$$A^L = A + \nu \cdot A^*,$$

$$b^L = b + \nu \cdot b^*.$$

На каждом шаге итерационного процесса Ньютона коэффициент ν берётся равным единице, затем проверяется: если после решения СЛАУ погрешность возросла (даже после перебора параметров релаксации вплоть до 0.1), то коэффициент уменьшают (например, можно уменьшать его в 1.5-2 раза) и пересчитывают матрицы и вектор. На следующей итерации коэффициент можно снова приравнять единице.

Вычисление производных для метода Ньютона

Для начала, выпишем формулу расчёта локальной матрицы $\hat{A}$, являющейся суммой матриц массы $\widehat{M}$ и жёсткости $\hat{G}$:

$$\hat{A}_{ij}(\hat{q}) = \hat{G}_{ij}(\hat{q}) + \widehat{M} =$$

$$= \int_{\Omega_k} \lambda^k\,(u(x,t),x)\ grad(\psi_i) grad\big(\psi_j\big)\ d\Omega_k + \int_{\Omega^k} \sigma^k\,(x)\psi_i\psi_j\ d\Omega^k$$

Отсюда видим, что от $\hat{q}$ зависит только матрица жёсткости $\hat{G}$, а точнее, коэффициент λ. Соответственно,

$$\frac{\partial \hat{A}_{ij}}{\partial q_r} = \frac{\partial \hat{G}_{ij}}{\partial q_r}$$

Теперь разложим λ по базисным векторам:

$$\lambda^k(u(x,t),x) = \lambda^k(q^k, x^k) \cdot \psi_1 + \lambda^k(u(q^{k+1},t), x^{k+1}) \cdot \psi_2$$

Подставим разложение в матрицу жёсткости, получим следующее выражение:

$$\hat{G}_{ij}(\hat{q}) = \lambda^k(q^k, x^k) \int_{\Omega^k} \psi_1\ grad(\psi_i) grad\big(\psi_j\big) +$$

$$+\lambda^k(q^{k+1}, x^{k+1}) \int_{\Omega^k} \psi_2\ grad(\psi_i) grad\big(\psi_j\big)$$

Теперь, если мы возьмём производную от матрицы жёсткости по $\hat{q}^k$, получим следующее выражение:

$$\left.\frac{\partial \hat{G}_{ij}(\hat{q})}{\partial \hat{q}_k}\right|_{\hat{q}=\hat{q}^0} = \left.\frac{\partial \lambda^k(\hat{q}^k, x^k)}{\partial \hat{q}_k}\right|_{\hat{q}=\hat{q}^0} \int_{\Omega^k} \psi_1\ \ grad(\psi_i) grad\big(\psi_j\big).$$

Аналогично будет выглядеть выражение и для $\hat{q}_{k+1}$. Как можно заметить, часть с интегралом не меняется, поэтому мы можем спокойно найти общую формулу локальной матрицы жёсткости на конечном элементе:

$$\left.\frac{\partial \hat{G}^k(\hat{q})}{\partial \hat{q}_k}\right|_{\hat{q}=\hat{q}^0} = \frac{1}{2h_{x^k}} \left.\frac{\partial \lambda^k(\hat{q}^k, x^k)}{\partial \hat{q}_k}\right|_{\hat{q}=\hat{q}^0} \cdot \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$$

Для $\hat{q}_{k+1}$ аналогично:

$$\left.\frac{\partial \hat{G}^k(\hat{q})}{\partial \hat{q}_{k+1}}\right|_{\hat{q}=\hat{q}^0} = \frac{1}{2h_{x^k}} \left.\frac{\partial \lambda^k(\hat{q}^{k+1}, x^{k+1})}{\partial \hat{q}_{k+1}}\right|_{\hat{q}=\hat{q}^0} \cdot \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$$

Вполне очевидно, что для других элементов вектора q, не относящихся к конечному элементу Ω_k, производные будут равны нулю, поскольку они не вносят вклад в локальную матрицу масс конечного элемента Ω_k.

Теперь мы можем переписать формулу для добавочной матрицы $\hat{A}^*$ следующим образом:

$$\hat{A}^*_{ij} = \sum_{k=1}^{2} \left.\frac{\partial \hat{A}_{ik}(\hat{q})}{\partial \hat{q}_j}\right|_{\hat{q}=\hat{q}^0} \hat{q}^0_k = \sum_{k=1}^{2} \left.\frac{\partial \hat{G}_{ik}(\hat{q})}{\partial \hat{q}_j}\right|_{\hat{q}=\hat{q}^0} \hat{q}^0_k =$$
$$= \left.\frac{\partial \hat{G}_{i1}(\hat{q})}{\partial \hat{q}_j}\right|_{\hat{q}=\hat{q}^0} \hat{q}^0_1 + \left.\frac{\partial \hat{G}_{i2}(\hat{q})}{\partial \hat{q}_j}\right|_{\hat{q}=\hat{q}^0} \hat{q}^0_2$$

Аппроксимация по времени

Аппроксимацию по времени для нашего параболического уравнения будем проводить по двухслойной неявной схеме:

$$\left.\frac{\partial u}{\partial t}\right|_{t=t_s} = \frac{u_s - u_{s-1}}{\Delta t_s}$$

Таким образом, решение нашей задачи (1) сводится к решению следующей задачи:

$$-div\left(\lambda(u(x,t_s),x)\ grad\big(u(x,t_s)\big)\right) + \sigma(x)\frac{u(x,t_s)}{\Delta t_s} = f(x,t_s) + \sigma(x)\frac{u(x,t_{s-1})}{\Delta t_s}$$

Это уравнение, после конечномерной аппроксимации, превратится в следующее:

$$A(q_s)q_s = d,$$

где

$$A(q_s) = G(q_s) + \frac{1}{\Delta t_s} M_\sigma,$$

$$d = b_s + \frac{1}{\Delta t} M_\sigma \cdot q_{s-1},$$

$$b_s = M_\sigma|_{\sigma=1} \cdot \bar{f}(x,t_s).$$

1 слой будет находиться за счёт начального условия, дальше второй слой:

1. собирается вектор b_s
2. собирается вектор d
3. собирается матрица массы M_σ
4. собирается матрица жёсткости с начальным приближением q_s^0
5. собирается вся матрица $A(q_s)$
6. Накладываются первые краевые условия
7. Решается система $Aq = d$, получается следующее приближение q_s^1
8. Повторяется по циклу в соответствии с выбранным методом

Тестирование

Основное уравнение:

$$-\frac{\partial}{\partial x}\left(\lambda(u,x)\frac{\partial u}{\partial x}\right)+\sigma(x)\frac{\partial u}{\partial t}=f(x,t).$$

Область тестирования: прямая с тремя узлами: (0), (1), (2).

Краевые 1 рода на узлах (0) и (2).

Все тесты делаются с измерением погрешности как на узлах сетки, так и на серединах конечных элементов, чтобы исключить особенность МКЭ в одномерном случае притягивать точные значения в узлах конечных элементов.

Тесты на работоспособность

Тест 1 (работоспособность по времени).

$u=t, \lambda=1, \sigma=1, f=1$

Результат:

Время	норма погрешности (узлы)	норма погрешности (середина)
0	0.000000000000000e+00	0.000000000000000e+00
1	3.845925372767128e-16	3.140184917367550e-16
2	0.000000000000000e+00	0.000000000000000e+00
3	0.000000000000000e+00	0.000000000000000e+00
4	0.000000000000000e+00	0.000000000000000e+00
5	8.881784197001252e-16	0.000000000000000e+00

Тест 2 (работоспособность по нелинейности).

$u=x, \lambda=u, \sigma=1, f=-1$

Результат:

Время	норма погрешности (узлы)	норма погрешности (середина)
0	0.000000000000000e+00	0.000000000000000e+00
1	0.000000000000000e+00	0.000000000000000e+00
2	0.000000000000000e+00	0.000000000000000e+00
3	0.000000000000000e+00	0.000000000000000e+00
4	0.000000000000000e+00	0.000000000000000e+00
5	0.000000000000000e+00	0.000000000000000e+00

Тест 3 (комплексный).

$u = t, \lambda = u, \sigma = 1, f = 1$

Результат:

Время	норма погрешности (узлы)	норма погрешности (середина)
0	0.000000000000000e+00	0.000000000000000e+00
1	3.845925372767128e-16	3.140184917367550e-16
2	4.440892098500626e-16	0.000000000000000e+00
3	4.440892098500626e-16	0.000000000000000e+00
4	0.000000000000000e+00	0.000000000000000e+00
5	8.881784197001252e-16	0.000000000000000e+00

Тест порядка аппроксимации по x

Тест 4.

$u = x + t, \lambda = 1, \sigma = 1, f = 1$

Результат:

Время	норма погрешности (узлы)	норма погрешности (середина)
0	0.000000000000000e+00	0.000000000000000e+00
1	0.000000000000000e+00	0.000000000000000e+00
2	0.000000000000000e+00	0.000000000000000e+00
3	0.000000000000000e+00	0.000000000000000e+00
4	0.000000000000000e+00	0.000000000000000e+00
5	8.881784197001252e-16	0.000000000000000e+00

Тест 5.

$u = x^2 + t, \lambda = 1, \sigma = 1, f = -1$

Время	норма погрешности (узлы)	норма погрешности (середина)
0	0.000000000000000e+00	0.000000000000000e+00
1	0.000000000000000e+00	2.500000000000000e-01
2	0.000000000000000e+00	2.500000000000000e-01
3	0.000000000000000e+00	2.500000000000000e-01
4	0.000000000000000e+00	2.500000000000000e-01
5	8.881784197001252e-16	2.500000000000000e-01

Тест порядка аппроксимации по t

Тест 6.

$u = t, \lambda = 1, \sigma = 1, f = 1$

Результат:

Время	норма погрешности (узлы)	норма погрешности (середина)
0	0.000000000000000e+00	0.000000000000000e+00
1	3.845925372767128e-16	3.140184917367550e-16
2	0.000000000000000e+00	0.000000000000000e+00
3	0.000000000000000e+00	0.000000000000000e+00
4	0.000000000000000e+00	0.000000000000000e+00
5	8.881784197001252e-16	0.000000000000000e+00

Тест 7.

$u = t^2, \lambda = 1, \sigma = 1, f = 2t$

Время	норма погрешности (узлы)	норма погрешности (середина)
0	0.000000000000000e+00	0.000000000000000e+00
1	3.750000000000000e-01	1.875000000000000e-01
2	4.687500000000000e-01	2.343750000000000e-01
3	4.921875000000000e-01	2.460937500000000e-01
4	4.980468750000000e-01	2.490234375000000e-01
5	4.995117187500036e-01	2.497558593750000e-01

Тест порядка сходимости по x

Порядок сходимости k_x можно определить по следующей формуле:

$$\frac{\| u^* - u^h \|}{\| u^* - u^{\frac{h}{2}} \|} \approx 2^{k_x}$$

Тест 8.

$u = x^2, t = 3, \lambda = 1, \sigma = 1, f = -1$

Шаг	норма погрешности	отношение норм
1	3.535533905932738e-01	-
1/2	1.250000000000038e-01	2.83
1/4	4.419417382417273e-02	2.82
1/8	1.562500000005995e-02	2.83
1/16	5.524271728361084e-03	2.83
1/32	1.953125001778966e-03	2.83

Тест порядка сходимости по t

Порядок сходимости k_t можно определить по следующей формуле:

$$\frac{\| u^* - u^h \|}{\| u^* - u^{\frac{h}{2}} \|} \approx 2^{k_t}$$

Тест 9.

$u = t^2, t = 3, \lambda = 1, \sigma = 1, f = 2t$

шаг	норма погрешности	отношение норм
1	2.460937500000000e-01	-
1/2	1.244879999999995e-01	1.98
1/4	6.242424288400628e-02	1.99
1/8	3.123501746621393e-02	1.99

Тестирование решения нелинейности методом простых итераций (число итераций)

Тест 10.

$u = x + t, \lambda = u, \sigma = 1, f = 0$

Результат:

Время	норма погрешности (узлы)	норма погрешности (середина)	число итераций
0	0.000000000000000e+00	0.000000000000000e+00	0
1	2.370787622225936e-15	1.884110950420530e-15	1
2	1.986027322597818e-15	2.220446049250313e-15	1
3	3.140184917367550e-15	3.323259344844179e-15	3
4	2.175583928816829e-15	1.986027322597818e-15	1

Тест 11.

$u = x^2 + t, \lambda = u, \sigma = 1, f = -6x^2 - 2t + 1$

Результат:

Время	норма погрешности (узлы)	норма погрешности (середина)	число итераций
0	0.000000000000000e+00	0.000000000000000e+00	0
1	2.970731181390800e-02	2.283296988681387e-02	24
2	2.316324456998595e-02	2.590767218531788e-02	19
3	1.696102680988593e-02	3.007054826696959e-02	17
4	1.324622633563345e-02	3.290178503323827e-02	16

Погрешность вышла на уровне теста 5, а значит, нелинейность не дала дополнительной погрешности в данном случае.

Тест 12.

$u = t, \lambda = u^2, \sigma = 1, f = 1$

Результат:

Время	норма погрешности (узлы)	норма погрешности (середина)	число итераций
0	0.000000000000000e+00	0.000000000000000e+00	0
1	1.093442869781314e-15	5.551115123125783e-16	1
2	4.282641246519316e-15	4.305604634972164e-15	2
3	1.776356839400250e-15	2.035072419451040e-15	22
4	1.008776768038948e-14	9.769962616701378e-15	500
5	2.512147933894040e-15	2.808666774861361e-15	500
6	2.512147933894040e-15	2.664535259100376e-15	500
7	8.881784197001252e-16	0.000000000000000e+00	500
8	2.682240007086545e-14	2.664535259100376e-14	500

Точность достигается, но после $t = 4$ метод начал зацикливаться и выходить по критическому числу итераций. Это происходит из-за стагнации нелинейного процесса.

Тестирование сходимости метода простых итераций от параметра релаксации

Тест 13.

$u = t + x, \lambda = u^2, \sigma = 1, f = -2t - 2x + 1$

Выберем результаты при времени $t = 2$:

ω	норма погрешности	число итераций
0.5	3.741962194848174e-15	53
0.6	1.332267629550188e-15	44
0.7	2.035072419451040e-15	500
0.8	2.264419546801470e-15	53
0.9	3.011959563148489e-15	23
1.0	1.719950113979703e-15	44
1.1	5.063396036227354e-15	58
1.2	6.137434987378955e-15	63

На коэффициенте релаксации, равном 0.7, метод начал стагнировать.

Минимальное число итераций было достигнуто при коэффициенте релаксации 0.9, но при этом минимальная погрешность была достигнута при коэффициенте релаксации 0.6.

Метод Ньютона

Тестирование на работоспособность

Тест 14.

$u = t, \lambda = u, \sigma = 1, f = 1$

Результат:

Время	норма погрешности (узлы)	норма погрешности (середина)	число итераций
0	0.000000000000000e+00	0.000000000000000e+00	0
1	9.450600277918405e-15	7.669383368012169e-15	1
2	5.599751867943366e-15	5.475099487534308e-15	1
3	8.296271088900922e-15	8.067281120720504e-15	1
4	3.352800008858181e-15	3.552713678800501e-15	1

Сравнение Ньютона с методом простых итераций

Тест 15.

$u = t + x, \lambda = u^2 + 1, \sigma = 1, f = -2t - 2x + 1$

Результаты:

Метод простых итераций

Время	норма погрешности (узлы)	норма погрешности (середина)	число итераций
0	0.000000000000000e+00	0.000000000000000e+00	0
1	2.151198569461397e-14	2.073592286714116e-14	21
2	8.826098480675078e-15	8.426000324584083e-15	19
3	2.773336226623967e-15	2.979040983896728e-15	17
4	7.483924389696347e-15	7.215595591812694e-15	15

Ньютон (начальное приближение - нулевое)

Время	норма погрешности (узлы)	норма погрешности (середина)	число итераций
0	0.000000000000000e+00	0.000000000000000e+00	0
1	2.349899218380883e-15	2.253506617997884e-15	7
2	3.076740298213702e-15	3.011959563148489e-15	7
3	1.776356839400250e-15	1.601186416994688e-15	7
4	2.349899218380883e-15	2.664535259100376e-15	8

Ньютон (начальное приближение - значение функции на прошлом временном слое)

Время	норма погрешности (узлы)	норма погрешности (середина)	число итераций
0	0.000000000000000e+00	0.000000000000000e+00	0
1	3.068717493689478e-15	2.987304636342383e-15	1
2	4.051933808678556e-15	4.021398830883446e-15	1
3	6.040266291023252e-15	5.178925639311150e-15	1

Время	норма погрешности (узлы)	норма погрешности (середина)	число итераций
4	4.440892098500626e-15	4.165926057296536e-15	1

Выводы:

1. Порядок аппроксимации по координатной сетке получился равным 1. Порядок сходимости так же вышел примерно равным 1. При этом если смотреть на значения погрешности в узлах сетки, то порядок аппроксимации получился гораздо выше, чем порядок аппроксимации в центрах конечных элементов (в центре между узлами сетки). Такое происходит потому, что в одномерном случае метод конечных элементов притягивает точное значение в узлах.

2. Порядок аппроксимации по времени получился равным 1, порядок сходимости так же вышел равным 1.

3. В случае линейности системы, метод простой итерации делает 1 итерацию в любом случае (то есть, программа работает так же, как работала бы без метода простой итерации). При добавлении нелинейности, количество итераций метода начинает расти в зависимости от сложности зависимости λ от u.

4. В некоторых моментах времени метод простой итерации зависает, выходя из цикла по лимиту итераций. Это происходит из-за стагнации получаемого решения. Чтобы избежать подобного, можно сделать дополнительное условие выхода по значению приращения решения между итерациями метода, но в таком случае может уменьшиться точность получаемого решения (метод будет завершаться раньше).

5. При использовании различных коэффициентов релаксации, количество итераций метода простой итерации меняется. Наиболее оптимальным по числу итераций значением коэффициента релаксации в исследуемом примере оказалось значение 0.9, но при этом по полученной точности решения оно оказалось не самым оптимальным (хотя в целом разница погрешности крайне невелика).

6. Метод Ньютона сходится значительно лучше метода простых итераций за счёт использования производных функции $\lambda(u)$. При этом нужно помнить о том, что

7. Сходимость метода Ньютона прямо зависит от начального приближения. В нашем тесте, при выборе начальным приближением нулевого вектора, метод сошёлся за 7 итераций в среднем. При выборе же начальным приближением значения функции с предыдущего слоя, метод сходился за 1 итерацию.

Текст программы:

Полный текст программы находится тут: https://github.com/SemafonKA/umf_lab_2_newtoon.

Файл `main.cpp`:

```
#include <fstream>
#include <iostream>
#include <vector>
#include <array>

#include "ThreeSteppers/Headers/IterSolvers.h"
#include "gaussian_quadrature.h"

// Файл, содержащий в себе пути до файлов, функции f, lambda и gamma
#include "Functions.h"

using namespace std;

#pragma region GLOBAL_OBJECTS
// Глобальная разреженная матрица системы
SparseMatrix global_mat;
// Массив финитных элементов
vector<FiniteElem> finiteElements;
// Массив узлов
vector<Node> nodes;
// Массив сопоставления узлов и первых краевых
vector<S1_node> s1_nodes;
// Массив сопоставления рёбер и вторых краевых
//vector<S23_edge> s2_edges;
//// Массив сопоставления рёбер и третьих краевых
//vector<S23_edge> s3_edges;
#pragma endregion GLOBAL_OBJECTS

void generatePortrait() {
   global_mat.di.resize(nodes.size());
   global_mat.ig.resize(nodes.size() + 1);

   for (auto& rect : finiteElements) {
      const int elems[2] = { rect.a, rect.b };
      for (int i = 0; i < 2; i++) {
         for (int k = 0; k < i; k++) {
            // Если элемент в верхнем прямоугольнике, то скипаем
            if (elems[k] > elems[i]) continue;

            bool isExist = false;
            // Пробегаем по всей строке для проверки, существует ли такой элемент
            for (auto it = global_mat.ig[elems[i]];
               it < global_mat.ig[elems[i] + 1ll]; it++) {
               if (global_mat.jg[it] == elems[k]) {
                  isExist = true;
                  break;
```

```
                }
            }
            if (!isExist) {
                // Ищем, куда вставить элемент портрета
                auto it = global_mat.ig[elems[i]];
                while (it < global_mat.ig[elems[i] + 1ll] &&
                    global_mat.jg[it] < elems[k])
                    it++;

                // Для вставки нужно взять итератор массива от начала, так что...
                global_mat.jg.insert(global_mat.jg.begin() + it, elems[k]);

                // Добавляем всем элементам ig с позиции elems[i]+1 один элемент
                for (auto j = elems[i] + 1; j < global_mat.ig.size(); j++)
                    global_mat.ig[j]++;
            }
        }
    }
    global_mat.ggl.resize(global_mat.jg.size());
    global_mat.ggu.resize(global_mat.jg.size());
}

Matrix getLocalG(const FiniteElem& finiteElem, const std::vector<double>& u, double
t) {
    Matrix g = {};

    double x_s = nodes[finiteElem.a].x;
    double hx = abs(nodes[finiteElem.b].x - nodes[finiteElem.a].x);

    double multiplier = lambda_value(finiteElem.regionNum, u[finiteElem.a],
nodes[finiteElem.a]);
    multiplier += lambda_value(finiteElem.regionNum, u[finiteElem.b],
nodes[finiteElem.b]);
    multiplier = multiplier / 2 / hx;

    constexpr Matrix constMat = {
        1, -1,
        -1, 1
    };
    for (int i = 0; i < finiteSize; i++) {
        for (int j = 0; j < finiteSize; j++) {
            g[i][j] = multiplier * constMat[i][j];
        }
    }

    return g;
}

Matrix getLocalAdditionalMat(const FiniteElem& finiteElem, const
std::vector<double>& u, double t) {
```

```
    Matrix add = {};
    Matrix dg_1 = {}, dg_2 = {};

    double x_s = nodes[finiteElem.a].x;
    double x_s_1 = nodes[finiteElem.b].x;
    double hx = abs(x_s_1 - x_s);
    double hx_1 = 1.0 / 2.0 / hx;
    double q1 = u[finiteElem.a];
    double q2 = u[finiteElem.b];
    constexpr Matrix constMat = {
         1, -1,
        -1,  1
    };

    // Считаем первую матрицу производных
    double multiplier = lambda_dif_value(finiteElem.regionNum, q1, x_s) / 2 / hx;

    for (int i = 0; i < finiteSize; i++) {
        for (int j = 0; j < finiteSize; j++) {
            dg_1[i][j] = multiplier * constMat[i][j];
        }
    }

    // Считаем вторую матрицу производных
    multiplier = lambda_dif_value(finiteElem.regionNum, q2, x_s_1) / 2 / hx;

    for (int i = 0; i < finiteSize; i++) {
        for (int j = 0; j < finiteSize; j++) {
            dg_2[i][j] = multiplier * constMat[i][j];
        }
    }

    // Считаем добавочную матрицу
    for (int i = 0; i < finiteSize; i++) {
        add[i][0] = dg_1[i][0] * q1 + dg_1[i][1] * q2;
        add[i][1] = dg_2[i][0] * q1 + dg_2[i][1] * q2;
    }

    return add;
}

Matrix getLocalM(const FiniteElem& finiteElem, bool getWithoutSigma = false) {
    Matrix m = {};
    std::function<double(int, double)> maybeSigma;
    if (getWithoutSigma == true) {
        maybeSigma = [](int reg, double x) {
            return 1.0;
        };
    } else {
        maybeSigma = [](int reg, double x) {
            return sigma_value(reg, x);
```

```
      };
   }

   double x_s = nodes[finiteElem.a].x;
   double hx = abs(nodes[finiteElem.b].x - nodes[finiteElem.a].x);
   double constCoef = (maybeSigma(finiteElem.regionNum, nodes[finiteElem.a].x) +
maybeSigma(finiteElem.regionNum, nodes[finiteElem.b].x)) / 2.0;
   constCoef = constCoef * hx / 6.0;

   static const Matrix constMat = {
      2, 1,
      1, 2
   };

   for (int i = 0; i < finiteSize; i++) {
      for (int j = 0; j < finiteSize; j++) {
         m[i][j] = constCoef * constMat[i][j];
      }
   }

   return m;
}

std::vector<double> getLocalB(const FiniteElem& finiteElem, double t) {
   std::vector<double> b(finiteSize);

   auto M = getLocalM(finiteElem, true);
   for (int i = 0; i < finiteSize; i++) {
      b[i] = 0;
      b[i] += M[i][0] * f_value(finiteElem.regionNum, nodes[finiteElem.a], t);
      b[i] += M[i][1] * f_value(finiteElem.regionNum, nodes[finiteElem.b], t);
   }

   return b;
}

void addLocalMatrixToGlobal(const FiniteElem& finiteElem, SparseMatrix& globalMat,
const Matrix& localMat) {
   const int elems[finiteSize] = { finiteElem.a, finiteElem.b };
   for (int i = 0; i < finiteSize; i++) {
      // добавляем все внедиагональные элементы на строке elems[i]
      for (int k = 0; k < i; k++) {
         // Если элемент в верхнем прямоугольнике, то скипаем
         if (elems[k] > elems[i]) {
            continue;
         }

         auto id = globalMat.ig[elems[i]];
         for (id; id < globalMat.ig[elems[i] + 1ll] && globalMat.jg[id] !=
elems[k]; id++);
```

```
            globalMat.ggl[id] += localMat[i][k];
            globalMat.ggu[id] += localMat[k][i];
        }
        // добавляем диагональные элементы
        globalMat.di[elems[i]] += localMat[i][i];
    }
}

void addLocalbToGlobal(const FiniteElem& finiteElem, std::vector<double>&
globalVec, const std::vector<double>& localVec) {
    const int elems[finiteSize] = { finiteElem.a, finiteElem.b };
    for (int i = 0; i < finiteSize; i++) {
        globalVec[elems[i]] += localVec[i];
    }
}


void include_s1(double t, SparseMatrix& global_mat, vector<double>& global_b) {
    for (const auto& node : s1_nodes) {
        double u = s1_u_value(node.funcNum, nodes[node.node], t);

        // ставим на диагональ значение 1
        global_mat.di[node.node] = 1;
        // ставим в соответствующую ячейку вектора b значение u
        global_b[node.node] = u;
        // зануляем строку в нижнем треугольнике
        for (auto j = global_mat.ig[node.node]; j < global_mat.ig[node.node + 1ll];
            j++) {
            global_mat.ggl[j] = 0;
        }
        // зануляем строку в верхнем треугольнике
        for (int i = node.node + 1; i < global_mat.Size(); i++) {
            for (auto j = global_mat.ig[i]; j < global_mat.ig[i + 1ll]; j++) {
                if (global_mat.jg[j] == node.node) {
                    global_mat.ggu[j] = 0;
                    break;
                }
            }
        }
    }
}

void getGlobals(
    const SparseMatrix& global_M,
    SparseMatrix& global_mat,
    std::vector<double>& global_d,
    const std::array<std::vector<double>, 2>& slices,
    double t,
    double dt
) {
    std::vector<double> global_b(global_mat.Size());
```

```
   // Считаем вектор правой части
   for (auto& elem : finiteElements) {
      auto local_b = getLocalB(elem, t);
      addLocalbToGlobal(elem, global_b, local_b);
   }

   global_d = global_b + (1 / dt) * global_M * slices[0];

   // Считаем матрицу жёсткости с учётом u с начального приближения
   auto global_G = SparseMatrix::copyShape(global_mat);
   for (auto& elem : finiteElements) {
      auto local_g = getLocalG(elem, slices[1], t);
      addLocalMatrixToGlobal(elem, global_G, local_g);
   }

   // Формируем полную глобальную матрицу
   global_mat = global_G + (1 / dt) * global_M;

   // Накладываем краевые
   include_s1(t, global_mat, global_d);
}

void getGlobalsNewtoon(
   const SparseMatrix& global_M,
   SparseMatrix& global_mat,
   std::vector<double>& global_d,
   std::array<std::vector<double>, 2> slices,
   double t,
   double dt,
   double nu
) {
   std::vector<double> global_b(global_mat.Size());

   // Считаем вектор правой части
   for (auto& elem : finiteElements) {
      auto local_b = getLocalB(elem, t);
      addLocalbToGlobal(elem, global_b, local_b);
   }

   global_d = global_b + (1 / dt) * global_M * slices[0];

   // Считаем матрицу жёсткости с учётом u с начального приближения
   auto global_G = SparseMatrix::copyShape(global_mat);
   for (auto& elem : finiteElements) {
      auto local_g = getLocalG(elem, slices[1], t);
      addLocalMatrixToGlobal(elem, global_G, local_g);
   }

   // Формируем полную глобальную матрицу
   global_mat = global_G + (1 / dt) * global_M;
```

```
    // Добавочные матрица и вектор Ньютона
    auto global_additional_mat = SparseMatrix::copyShape(global_mat);
    auto global_additional_vec = std::vector<double>(global_mat.Size());
    // Добавляем в глобальную матрицу часть с производными матрицы G
    for (auto& el : finiteElements) {
        auto local_additional_mat = getLocalAdditionalMat(el, slices[1], t);
        addLocalMatrixToGlobal(el, global_additional_mat, local_additional_mat);
    }
    global_additional_vec = global_additional_mat * slices[1];

    global_mat = global_mat + nu * global_additional_mat;
    global_d = global_d + nu * global_additional_vec;

    include_s1(t, global_mat, global_d);
}


int main() {
    setlocale(LC_ALL, "ru-RU");

    // Задаём параметры решателя нелинейного процесса
    constexpr double eps = 1e-13;
    constexpr double maxIter = 500;
    double relaxCoef = 1.0;
    constexpr bool newtoonMode = true;
    if (newtoonMode) {
        cout << "********************\n";
        cout << "** РЕЖИМ НЬЮТОНА  **\n";
        cout << "********************\n\n";
    }

    // Генерируем сетку по пространству
    const double g_begin = 0.0;
    const double g_end = 2.0;
    const double g_step = 1.0 / 4.0;
    nodes.resize(std::floor((g_end - g_begin) / g_step) + 1);
    for (size_t i = 0; i < nodes.size(); i++) {
        nodes[i].x = g_begin + i * g_step;
    }
    finiteElements.resize(nodes.size() - 1);
    for (size_t i = 0; i < nodes.size() - 1; i++) {
        finiteElements[i].a = i;
        finiteElements[i].b = i + 1;
    }
    s1_nodes.resize(2);
    s1_nodes[0].node = 0;
    s1_nodes[1].node = nodes.size() - 1;

    cout << "СЕТКА\n";
```

```
   cout << "Начало: " << g_begin << " , конец: " << g_end << ", шаг: " << g_step <<
endl;
   cout << "Число узлов: " << nodes.size() << endl;
   cout << "Координаты узлов: [" << endl;
   for (auto el : nodes) {
      cout << format("{:20.15e}\n", el.x);
   }
   cout << "]\n\n";

   // Генерируем сетку по времени
   const double t_begin = 0.0;
   const double t_end = 4.0;
   const double t_step = 1.0 / 1.0;
   std::vector<double> t(std::floor((t_end - t_begin) / t_step) + 1);
   for (size_t i = 0; i < t.size(); i++) {
      t[i] = t_begin + i * t_step;
   }

   generatePortrait();

   auto global_M = SparseMatrix::copyShape(global_mat);

   // Считаем глобальную матрицу массы (не будет меняться)
   for (auto& elem : finiteElements) {
      auto local_m = getLocalM(elem, false);
      addLocalMatrixToGlobal(elem, global_M, local_m);
   }

   // Создаём 2 слоя, 1 слой считаем через начальное условие
   std::array<std::vector<double>, 2> slices = {};
   slices[0].resize(global_mat.Size());
   slices[1].resize(global_mat.Size());
   for (size_t i = 0; i < slices[0].size(); i++) {
      slices[1][i] = slices[0][i] = s1_u_value(0, nodes[i], t_begin);
      //slices[1][i] = 0.0;
   }

   // Инициализируем решатель
   IterSolvers::minEps = 1e-15;
   IterSolvers::maxIter = 5000;
   IterSolvers::MSG_Assimetric::Init_LuPrecond(global_mat.Size(), global_mat);
   std::vector<double> errors = { 0.0 };           // Массив погрешностей
   std::vector<double> middle_errors = { 0.0 };    // Массив погрешностей по
середине конечных элементов
   std::vector<size_t> number_of_iter = { 0 };     // Массив числа итераций на
каждом слое

   for (size_t i = 1; i < t.size(); i++) {
      cout << "Текущее время: " << t[i] << endl;
      double dt = t[i] - t[i - 1];
```

```
      std::vector<double> global_d(global_mat.Size());

      // Цикл простой итерации
      double nev = INFINITY;
      size_t j;

      if (!newtoonMode) { // (без Ньютона)
         // Получаем глобальные элементы
         getGlobals(global_M, global_mat, global_d, slices, t[i], dt);
         for (j = 0; j < maxIter && nev > eps; j++) {
            // Решаем СЛАУ
            double ee = 0.0;
            auto newSlice = slices[1]; // Решение q* - временное, копируем его

            IterSolvers::MSG_Assimetric::LuPrecond(global_mat, global_d, newSlice,
ee);
            slices[1] = relaxCoef * newSlice + (1.0 - relaxCoef) * (slices[1]);

            // Считаем глобальные матрицы
            getGlobals(global_M, global_mat, global_d, slices, t[i], dt);

            // Находим норму погрешности
            nev = norm(global_mat * slices[1] - global_d) / norm(global_d);
         }
      } else { // С Ньютоном
         for (j = 0; j < maxIter && nev > eps; j++) {
            double nu = 2.0;  // коэффициент дэмпинга
            double newNev = nev;
            auto oldSlice = slices[1];

            // Дэмпинг матрицы на случай, если она перестаёт быть положительно
определённой
            while (newNev >= nev && nu > (1.0 / 256.0)) {
               static const vector<double> relaxCoeffs = { 0.1, 0.2, 0.3, 0.4, 0.5,
0.6, 0.7, 0.8, 0.9, 1.0 };

               nu /= 2.0;
               slices[1] = oldSlice;
               // Получаем глобальные элементы
               getGlobalsNewtoon(global_M, global_mat, global_d, slices, t[i], dt,
nu);

               // Решаем СЛАУ
               double ee = 0.0;
               auto newSlice = slices[1]; // Решение q* - временное, копируем его

               IterSolvers::MSG_Assimetric::LuPrecond(global_mat, global_d,
newSlice, ee);

               // Находим новое приближение функции
               relaxCoef = 1.0;
```

```
                slices[1] = relaxCoef * newSlice + (1.0 - relaxCoef) * oldSlice;

                // Считаем глобальные матрицы без добавочных матриц Ньютона
                getGlobals(global_M, global_mat, global_d, slices, t[i], dt);

                // Находим невязку
                newNev = norm(global_mat * slices[1] - global_d) / norm(global_d);

                // Если новая невязка вышла больше, чем была на прошлой итерации, то
                if (newNev > nev) {
                   // Уменьшаем значение коэф. релаксации с шагом 0.1
                   for (int k = relaxCoeffs.size() - 2; k >= 0 && newNev > nev; k--)
{
                      slices[1] = relaxCoeffs[k] * newSlice + (1.0 - relaxCoeffs[k])
* oldSlice;
                      // Считаем глобальные матрицы без добавочных матриц Ньютона
                      getGlobals(global_M, global_mat, global_d, slices, t[i], dt);
                      newNev = norm(global_mat * slices[1] - global_d) /
norm(global_d);
                   }
                }
             }

             nev = newNev;
          }
       }


       // Выводим чё мы нарешали
       //cout << "Выход из цикла за " << j << " итераций\n";
       number_of_iter.push_back(j);
       cout << "Полученное решение: \n";
       for (auto el : slices[1]) {
          cout << format("{:.15e}\n", el);
       }
       //cout << "Погрешность решения: \n";
       double err = 0.0;
       for (int j = 0; j < slices[1].size(); j++) {
          double tmperr = slices[1][j] - s1_u_value(0, nodes[j], t[i]);
          err += tmperr * tmperr;
          //cout << format("{:5} {:.15e}\n", nodes[j].x, tmperr);
       }
       errors.push_back(std::sqrt(err));

       //cout << "Погрешность решения между узлами: \n";
       err = 0.0;
       for (int j = 0; j < slices[1].size() - 1; j++) {
          double x1 = nodes[j].x;
          double x2 = nodes[j + 1].x;
          double dx = x2 - x1;
          double midx = (x1 + x2) / 2;
```

```
         double u = slices[1][j] * (x2 - midx) / dx + slices[1][j + 1] * (midx -
x1) / dx;
         double tmperr = u - s1_u_value(0, midx, t[i]);
         err += tmperr * tmperr;
         //cout << format("{:5} {:.15e}\n", midx, tmperr);
      }
      middle_errors.push_back(std::sqrt(err));

      // Сохраняем текущий слой на предыдущий
      slices[0] = slices[1];
      // Если не нужно использовать значение с прошлого слоя как начальное для
нового, то
      // раскомментить строчку ниже v
      //slices[1] = std::vector<double>(slices[1].size());
   }

   // Уничтожаем всё, что мы выделили в решателе
   IterSolvers::Destruct();

   cout << "Сводная таблица (по узлам): " << endl;
   cout << "| Время | норма погрешности (узлы) | норма погрешности (середина) |
число итераций | \n";
   cout << "| :---: |           :---:          |            :---:             |
:---:     |\n";
   for (int i = 0; i < t.size(); i++) {
      cout << format("| {:^5} | {:^24.15e} | {:^28.15e} | {:^14} |\n", t[i],
errors[i], middle_errors[i], number_of_iter[i]);
   }

   return 0;
}
```

Файл Functions.h:

```
/*
   Файл, содержащий в себе только вынесенные константы и константные функции для
main.cpp
   Ни в коем случае не добавлять его никуда, кроме main.cpp!
*/

#pragma once
#include <string>
#include <stdexcept>
#include <array>

namespace GlobalPaths {
   // Пути файлов:
   const std::string filesPath = "./iofiles/";
   const std::string nodesPath = filesPath + "nodes.txt";
   const std::string finiteElementsPath = filesPath + "finiteElements.txt";
   const std::string s1_nodesPath = filesPath + "s1_nodes.txt";
   //const std::string s2_edgesPath = filesPath + "s2_edges.txt";
   //const std::string s3_edgesPath = filesPath + "s3_edges.txt";
}

#pragma region TYPEDEFINES
constexpr size_t finiteSize = 2;
using Matrix = std::array<std::array<double, finiteSize>, finiteSize>;

/// <summary>
/// Структура финитного элемента, имеет 2 номера вершины: [a], [b], а также
/// номер области, в которой находится сам финитный элемент, [region]
/// </summary>
struct FiniteElem {
   int a = 0;
   int b = 0;
   int regionNum = 0;

   std::string toString() const {
      std::string out = "( ";
      out += "a: " + std::to_string(a);
      out += ", b: " + std::to_string(b);
      out += ", region: " + std::to_string(regionNum);
      out += " )";

      return out;
   }
};

/// <summary>
/// Структура описания узла сетки. Содержит координаты этого узла [x]
/// </summary>
struct Node {
   double x = 0.0;
```

```
};

struct S1_node {
   int node = 0;
   int funcNum = 0;
};

//struct S23_edge {
//   int node1 = 0;
//   int node2 = 0;
//   int funcNum = 0;
//};
#pragma endregion TYPEDEFINES

double f_value(int regionNum, double x, double t) {
   switch (regionNum)
   {
   case 0: {
      return -2*t - 2*x + 1;
   }

   default:
      throw std::runtime_error("Значения функции f для региона с номером " +
std::to_string(regionNum) + " не найдено.");
   }
}
double f_value(int regionNum, Node node, double t) {
   return f_value(regionNum, node.x, t);
}

double lambda_value(int regionNum, double u, double x) {
   switch (regionNum)
   {
   case 0: {
      return u*u + 1;
   }

   default:
      throw std::runtime_error("Значения функции lambda для региона с номером " +
std::to_string(regionNum) + " не найдено.");
   }

}
double lambda_value(int regionNum, double u, Node node) {
   return lambda_value(regionNum, u, node.x);
}

double lambda_dif_value(int regionNum, double u, double x) {
   switch (regionNum) {
   case 0: {
      return 2*u;
```

```
    }
    default:
        throw std::runtime_error("Значения производной функции lambda для региона с
номером " + std::to_string(regionNum) + " не найдено.");
    }
}
double lambda_dif_value(int regionNum, double u, Node node) {
    return lambda_dif_value(regionNum, u, node.x);
}

double sigma_value(int regionNum, double x) {
    switch (regionNum)
    {
    case 0: {
        return 1;
    }

    default:
        throw std::runtime_error("Значения функции gamma для региона с номером " +
std::to_string(regionNum) + " не найдено.");
    }
}
double sigma_value(int regionNum, Node node) {
    return sigma_value(regionNum, node.x);
}

double s1_u_value(int s1_funcNum, double x, double t) {
    switch (s1_funcNum) {
    case 0: {
        return x + t;
    }

    default:
        throw std::runtime_error("Значения функции u для s1-краевого с номером " +
std::to_string(s1_funcNum) + " не найдено.");
    }
}
double s1_u_value(int s1_funcNum, Node node, double t) {
    return s1_u_value(s1_funcNum, node.x, t);
}


std::vector<double> operator*(double coef, const std::vector<double> right) {
    auto result = right;
    for (std::size_t i = 0; i < right.size(); i++) {
        result[i] *= coef;
    }
    return result;
}
```

```
std::vector<double> operator+ (const std::vector<double> left, const
std::vector<double> right) {
   auto result = left;
   for (std::size_t i = 0; i < left.size(); i++) {
      result[i] += right[i];
   }

   return result;
}

std::vector<double> operator- (const std::vector<double> left, const
std::vector<double> right) {
   auto result = left;
   for (size_t i = 0; i < left.size(); i++) {
      result[i] -= right[i];
   }

   return result;
}

double norm(const std::vector<double> vec) {
   double res = 0.0;
   for (auto el : vec) {
      res += el * el;
   }
   return std::sqrt(res);
}
```