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

Кафедра прикладной математики

Курсовой проект по курсу

«ЧИСЛЕННЫЕ МЕТОДЫ»

Группа ПМ-24

Студент ГЕРАСИМЕНКО ВАДИМ

Новосибирск

2024

1. Условие задачи

МКЭ для двумерной краевой задачи для эллиптического уравнения в декартовой системе координат. Базисные функции билинейные на прямоугольниках. Краевые условия всех типов. Коэффициент диффузии λ разложить по биквадратичным базисным функциям. Матрицу СЛАУ генерировать в разреженном строчном формате. Для решения СЛАУ использовать МСГ или ЛОС с неполной факторизацией.

2. Постановка задачи

Решаемое уравнение в общем виде:

$$-div(\lambda\, grad\, u) + \gamma u = f\ ,$$

область интегрирования Ω с границей $S = S_1 \cup S_2 \cup S_3$ и краевыми условиями:

$$u|_{S_1} = u_g\ ,$$

$$\lambda \frac{\partial u}{\partial n}|_{S_2} = \theta\ ,$$

$$\lambda \frac{\partial u}{\partial n}|_{S_3} + \beta\left(u|_{S_3} - u_\beta\right) = 0\ .$$

Дифференциальное уравнение для двумерной эллиптической краевой задачи в декартовой системе координат:

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

3. Теоретическая часть

3.1 Вариационная постановка в форме уравнения Галеркина

Невязка $R(u)$ данного уравнения примет следующий вид:

$$R(u) = -\frac{\partial}{\partial x}\left(\lambda\, \frac{\partial u}{\partial x}\right) - \frac{\partial}{\partial y}\left(\lambda \frac{\partial u}{\partial y}\right) + \gamma u - f\ .$$

Потребуем, чтобы эта невязка была ортогональна (в смысле скалярного произведения пространства $L_2(\Omega) \equiv H^0$) некоторому пространству Ф функций v, которое будем называть *пространством пробных функций*:

$$\int_\Omega \left(-\frac{\partial}{\partial x}\left(\lambda\, \frac{\partial u}{\partial x}\right) - \frac{\partial}{\partial y}\left(\lambda \frac{\partial u}{\partial y}\right) + \gamma u - f\right) v\, d\,\Omega = 0\ , \quad \forall v \in \Phi\ .$$

Воспользуемся формулой Грина:

$$\int_\Omega \lambda\left(\frac{\partial u}{\partial x}\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\frac{\partial v}{\partial y}\right) d\Omega = -\int_\Omega \left(\frac{\partial}{\partial x}\left(\lambda\, \frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\lambda \frac{\partial u}{\partial y}\right)\right) v d\Omega + \int_S \lambda \frac{\partial u}{\partial n} v dS\ ,$$

где $S = S_1 \cup S_2 \cup S_3$ – граница Ω.

Преобразуем слагаемое $\int_\Omega -\left(\frac{\partial}{\partial x}\left(\lambda \frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\lambda \frac{\partial u}{\partial y}\right)\right) v d\Omega$:

$$\int_\Omega \lambda\left(\frac{\partial u}{\partial x}\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\frac{\partial v}{\partial y}\right) d\Omega - \int_S \lambda \frac{\partial u}{\partial n} v dS + \int_\Omega (\gamma u - f) v d\Omega = 0\,, \quad \forall v \in \Phi.$$

Преобразуем интегралы по границам S_2 и S_3, воспользовавшись краевыми условиями:

$$\int_\Omega \lambda\left(\frac{\partial u}{\partial x}\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\frac{\partial v}{\partial y}\right) d\Omega - \int_{S_1} \lambda \frac{\partial u}{\partial n} v dS - \int_{S_2} \theta v dS$$

$$- \int_{S_3} \beta(u - u_\beta)\, v dS + \int_\Omega (\gamma u - f) v d\Omega = 0, \qquad \forall v \in \Phi\,.$$

В качестве Ф выберем $H_0^1 -$ пространство пробных функций $v_0 \in H^1$, которые на границе S_1 удовлетворяют нулевым первым краевым условиям, и при этом будем считать, что $u \in H_g^1$, где H_g^1 – множество функций, имеющих с квадратом первым производные и удовлетворяющих только первым краевым условиям на границе S_1.

Учитывая, что $v_0|_{S_1} = 0$, получим итоговое уравнение:

$$\int_\Omega \lambda\left(\frac{\partial u}{\partial x}\frac{\partial v_0}{\partial x} + \frac{\partial u}{\partial y}\frac{\partial v_0}{\partial y}\right) d\Omega + \int_\Omega \gamma u v_0 d\Omega + \int_{S_3} \beta u v_0 dS =$$

$$= \int_\Omega f\, v_0 d\Omega + \int_{S_2} \theta v_0 dS + \int_{S_3} \beta u_\beta v_0 dS\,, \quad \forall v_0 \in H_0^1\,.$$

3.2 Конечноэлементная дискретизация и переход к локальным матрицам

При построении конечноэлементных аппроксимаций по методу Галеркина пространства H_g^1 и H_0^1 заменяются конечномерными пространстами V_g^h и V_0^h. При этом чаще всего в МКЭ функции из этих пространств являются элементами одного и того же конечномерного пространства V^h, которое мы будем определять как линейное пространство, натянутое на базисные функции $\psi_i\,,\ i = 1 \ldots n$.

Как правило, функции ψ_i являются финитными кусочно-полиномиальными функциями, а приближённое решение $u^h \in V_g^h$ является линейной комбинацией таких функций.

Для получения аппроксимации уравнения Галеркина на конечномерных пространствах V_g^h и V_0^h заменим в полученном уравнении функцию $u \in H_g^1$ аппроксимирующей её функцией $u^h \in V_g^h$, а функцию $v_0 \in H_0^1 -$ функцией $v_0^h \in V_0^h$:

$$\int_\Omega \lambda\left(\frac{\partial u^h}{\partial x}\frac{\partial v_0^h}{\partial x}+\frac{\partial u^h}{\partial y}\frac{\partial v_0^h}{\partial y}\right)d\Omega+\int_\Omega \gamma u^h v_0^h d\Omega+\int_{S_3}\beta u^h v_0^h dS=$$

$$=\int_\Omega f\ v_0^h d\Omega+\int_{S_2}\theta v_0^h dS+\int_{S_3}\beta u_\beta v_0^h dS\ ,\quad \forall v_0^h\in V_0^h\ .$$

Так как любая функция $v_0^h \in V_0^h$ может быть представлена в виде линейной комбинации:

$$v_0^h=\sum_{i\in N_0} q_i^v \psi_i\ ,$$

вариационное уравнение эквивалентно следующей системе уравнений:

$$\int_\Omega \lambda\left(\frac{\partial u^h}{\partial x}\frac{\partial \psi_i}{\partial x}+\frac{\partial u^h}{\partial y}\frac{\partial \psi_i}{\partial y}\right)d\Omega+\int_\Omega \gamma u^h \psi_i d\Omega+\int_{S_3}\beta u^h\ \psi_i dS=$$

$$=\int_\Omega f\ \psi_i d\Omega+\int_{S_2}\theta\psi_i dS+\int_{S_3}\beta u_\beta\psi_i dS\ ,\quad i\in N_0.$$

Таким образом, МКЭ-решение u^h удовлетворяет полученной системе уравнений. Поскольку $u^h \in V_g^h$, оно может быть представлено в виде линейной комбинации базисных функций пространства V^h:

$$u^h=\sum_{j=1}^n q_j\psi_j\ .$$

Подставляя данное выражение в ранее полученную систему уравнений, получаем СЛАУ для компонент q_j вектора весов $\boldsymbol{q}=(q_1,\dots,q_n)^T$ с индексами $j\in N_0$:

$$\sum_{j=1}^n\left(\int_\Omega \lambda\left(\frac{\partial \psi_j}{\partial x}\frac{\partial \psi_i}{\partial x}+\frac{\partial \psi_j}{\partial y}\frac{\partial \psi_i}{\partial y}\right)d\Omega+\int_\Omega \gamma\psi_j\psi_i d\Omega+\int_{S_3}\beta\psi_j\ \psi_i dS\right)q_j=$$

$$\int_\Omega f\ \psi_i d\Omega+\int_{S_2}\theta\psi_i dS+\int_{S_3}\beta u_\beta\psi_i dS,\quad i\in N_0.$$

При решении краевой задачи с использованием базисных функций, принимающих нулевые значения во всех узлах сетки, кроме одного, полученная конечноэлементная СЛАУ для вектора весов q может быть записана в матричном виде:

$$\boldsymbol{Aq}\ =\ \boldsymbol{b}\ ,$$

Где компоненты матрицы $\boldsymbol{A}$ и вектора $\boldsymbol{b}$ определяются соотношениями

$$A_{ij}=\begin{cases}\int_\Omega \lambda\left(\dfrac{\partial \psi_j}{\partial x}\dfrac{\partial \psi_i}{\partial x}+\dfrac{\partial \psi_j}{\partial y}\dfrac{\partial \psi_i}{\partial y}\right)d\Omega+\int_\Omega \gamma\psi_j\psi_i d\Omega+\int_{S_3}\beta\psi_j\ \psi_i dS\ , i\in N_0\ , \\ \delta_{ij}\ ,\ \ i\notin N_0\ ,\qquad j=1\dots n,\end{cases}$$

$$b_i = \begin{cases} \int_\Omega f\,\psi_i d\Omega + \int_{S_2} \theta\psi_i dS + \int_{S_3} \beta u_\beta \psi_i dS\,, & i \in N_0, \\ u_g(x_i), & i \notin N_0, \end{cases}$$

в которых δ_{ij} – символ Кронекера $\left(\delta_{ii} = 1 \text{ и } \delta_{ij} = 0 \text{ при } i \neq j\right)$.

Сборка глобальных матрицы и вектора правой части выполняется из локальных матриц и векторов конечных элементов. При этом локальная матрица представляет собой сумму двух матриц: матрицы жесткости и матрицы массы, где элементы матрицы жесткости определяются как

$$\hat{G}_{ij} = \int_{\Omega_l} \lambda \left(\frac{\partial \hat{\psi}_j}{\partial x}\frac{\partial \hat{\psi}_i}{\partial x} + \frac{\partial \hat{\psi}_j}{\partial y}\frac{\partial \hat{\psi}_i}{\partial y} \right) d\Omega\,,$$

элементы матрицы массы:

$$\hat{M}_{ij} = \int_{\Omega_l} \gamma \hat{\psi}_j \hat{\psi}_i d\Omega\,.$$

3.3 Базисные функции

Для реализации МКЭ для данной задачи необходимо разбить область Ω на прямоугольные конечные элементы $\Omega_{ps} = [\mathrm{x}_p,\ \mathrm{x}_{p+1}] \times [\mathrm{y}_s,\ \mathrm{y}_{s+1}]$ и определить построение билинейных базисных функций.

Зададим по две одномерные линейные функции на каждом из отрезков:

$$X_1(x) = \frac{x_{p+1} - x}{h_x}\,,\ X_2(x) = \frac{x - x_p}{h_x}\,, h_x = x_{p+1} - x_p\,,$$

$$Y_1(y) = \frac{y_{s+1} - y}{h_y}\,,\ Y_2(y) = \frac{y - y_s}{h_y}\,, h_y = y_{s+1} - y_s\,,$$

Локальные базисные функции на конечном элементе $\Omega_{ps} = \left[\mathrm{x}_p,\ \mathrm{x}_{p+1}\right] \times [\mathrm{y}_s,\ \mathrm{y}_{s+1}]$ представляются в виде произведения функций:

$$\hat{\psi}_1(x,y) = X_1(x)Y_1(y), \quad \hat{\psi}_2(x,y) = X_2(x)Y_1(y)\,,$$

$$\hat{\psi}_3(x,y) = X_1(x)Y_2(y), \quad \hat{\psi}_4(x,y) = X_2(x)Y_2(y)\,.$$

Если параметр λ на конечном элементе Ω_{ps} заменить его осредненным значением $\bar{\lambda}$ и учесть, что

$$\int_{x_p}^{x_p+h_x} \left(\frac{dX_1}{dx}\right)^2 dx = \frac{1}{h_x}\,,\ \int_{x_p}^{x_p+h_x} \frac{dX_1}{dx}\frac{dX_2}{dx}\,dx = -\frac{1}{h_x}\,,$$

$$\int_{x_p}^{x_p+h_x} \left(\frac{dX_2}{dx}\right)^2 dx = \frac{1}{h_x}, \int_{x_p}^{x_p+h_x} (X_1)^2 dx = \frac{h_x}{3},$$

$$\int_{x_p}^{x_p+h_x} X_1 X_2 dx = \frac{h_x}{6}, \int_{x_p}^{x_p+h_x} (X_2)^2 dx = \frac{h_x}{3}.$$

(аналогичный вид будут иметь и интегралы от произведений функций $Y_v(y)$ и их производных), а также заменить γ на $\bar{\gamma}$, то, подставляя в ранее полученные формулы компонент матриц жёсткости и массы, преобразовав, получим:

$$\hat{G} = \frac{\bar{\lambda}}{6}\frac{h_y}{h_x}\begin{pmatrix} 2 & -2 & 1 & -1 \\ -2 & 2 & -1 & 1 \\ 1 & -1 & 2 & -2 \\ -1 & 1 & -2 & 2 \end{pmatrix} + \frac{\bar{\lambda}}{6}\frac{h_x}{h_y}\begin{pmatrix} 2 & 1 & -2 & -1 \\ 1 & 2 & -1 & -2 \\ -2 & -1 & 2 & 1 \\ -1 & -2 & 1 & 2 \end{pmatrix},$$

$$\widehat{M} = \bar{\gamma}\hat{C} = \bar{\gamma}\frac{h_x h_y}{36}\begin{pmatrix} 4 & 2 & 2 & 1 \\ 2 & 4 & 1 & 2 \\ 2 & 1 & 4 & 2 \\ 1 & 2 & 2 & 4 \end{pmatrix}.$$

Если представить правую часть f решаемого уравнения в виде билинейного интерполянта $\sum_{v=1}^{4} \hat{f}_v \hat{\psi}_v$, то локальный вектор $\widehat{\boldsymbol{b}}$ в этом случае легко вычисляется через матрицу $\hat{C}$, фактически являющуюся матрицей массы при $\bar{\gamma} \equiv 1$:

$$\widehat{\boldsymbol{b}} = \widehat{\boldsymbol{C}} \cdot \hat{\boldsymbol{f}} = \frac{h_x h_y}{36}\begin{pmatrix} 4\hat{f}_1 + 2\hat{f}_2 + 2\hat{f}_3 + \hat{f}_4 \\ 2\hat{f}_1 + 4\hat{f}_2 + \hat{f}_3 + 2\hat{f}_4 \\ 2\hat{f}_1 + \hat{f}_2 + 4\hat{f}_3 + 2\hat{f}_4 \\ \hat{f}_1 + 2\hat{f}_2 + 2\hat{f}_3 + 4\hat{f}_4 \end{pmatrix}.$$

4. Занесение данных в программу

4.1 Способ задания расчётной области

Для задания расчётной области служит файл “AreaDescription.txt”, задающий расчетную область в виде подобластей.

В первой строке файла одно целое число n – количество вертикальных границ подобластей, во второй строке n чисел – координаты по оси X этих границ, так определяется массив xValues в классе Area. Следующие две строки аналогичны для горизонтальных границ и определяют массив yValues того же класса.

В пятой строке одно целое число k – количество подобластей. Следующие k строк описывают подобласти. Каждая представляется набором из пяти целых чисел. Первое обозначает индекс подобласти/номер формул, определяющих параметры дифференциального уравнения в рассматриваемой подобласти, второе и третье – индексы левой и правой вертикальных границ соответственно в массиве xValues, четвертое и пятое – индексы нижней и верхней горизонтальных границ соотвественно из массива yValues.

4.2 Способ задания краевых условий

Для задания краевых условий служит файл “BoundariesDescription.txt”. Каждый фрагмент границ S_1, S_2 и S_3 может быть описан шестью целыми числами:

- тип краевого условия (1 означает принадлежность границе S_1, 2 – границе S_2, 3 – границе S_3);
- индекс формулы, задающей параметр краевого условия на рассматриваемом фрагменте границы);
- индекс элемента массива Area.xValues, с которого рассматриваемый фрагмент начинается по оси x;
- индекс элемента массива Area.xValues, которым фрагмент рассматриваемый фрагмент заканчивается по оси x;
- индекс элемента массива Area.yValues, с которого рассматриваемый фрагмент начинается по оси y;
- индекс элемента массива Area.yValues, которым фрагмент рассматриваемый фрагмент заканчивается по оси y;

4.3 Способ задания сетки

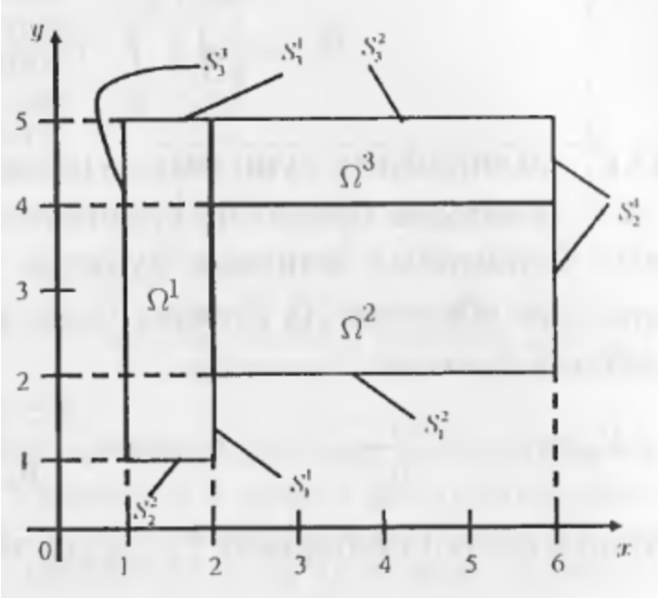
Двумерная регулярная прямоугольная сетка хранится в виде двух массивов xValues и yValues класса Grid. Эти массивы строятся на основе одноименных массивов из класса Area, разбиваемых на более мелкие отрезки на основе информации из файла “IntervalsDescription.txt”.

Первая строка файла “IntervalsDescription.txt” содержит n пар чисел, где n = количество интервалов в массиве Area.xValues. Первое число в каждой паре (целое) определяет количество интервалов, на которое должен быть разбит каждый интервал. Второе число в паре (вещественное) – коэффициент изменения длины интервалов при разбиении каждого подынтервала. Аналогично, вторая строка содержит m пар, где m = количество интервалов в массиве Area.yValues. Пары чисел во второй строке по смыслу соответствуют парам чисел из первой строки.

5. Тестирование программы

5.1 Проверка работоспособности программы

Расчетная область:



Рассмотрим тестовую краевую задачу с краевыми условиями всех типов, точное решение которой является функцией

$$u = \begin{cases} x, & \text{в } \Omega^1 \\ 1.8 + 0.1x, & \text{в } \Omega^2 \text{ и } \Omega^3 \end{cases},$$

принадлежащей конечномерному пространству кусочно-билинейных функций.

$$\lambda = \begin{cases} 1 & \text{в } \Omega^1 \\ 10 & \text{в } \Omega^2 \\ 10 & \text{в } \Omega^3 \end{cases}, \quad \gamma = \begin{cases} 2 & \text{в } \Omega^1 \\ 1 & \text{в } \Omega^2 \\ 0 & \text{в } \Omega^3 \end{cases}, \quad f = \begin{cases} 2x & \text{в } \Omega^1 \\ 1.8 + 0.1x & \text{в } \Omega^2 \\ 0 & \text{в } \Omega^3 \end{cases},$$

$$u|_{S_1^1} = 2\,,\ u|_{S_1^2} = 0.1x + 1.8\,,$$

$$\lambda \frac{\partial u}{\partial n}|_{S_2^1} = 1\,, \lambda \frac{\partial u}{\partial n}|_{S_2^2} = 0\,, \left(\lambda \frac{\partial u}{\partial n} + (u - x)\right)|_{S_3^1} = 0\,,$$

$$\left(\lambda \frac{\partial u}{\partial n} + 2(u - 1.8 - 0.1x)\right)|_{S_3^2} = 0\,,\ \left(\lambda \frac{\partial u}{\partial n} + 0.5(u + 1)\right)|_{S_3^3} = 0\,,$$

Результаты:

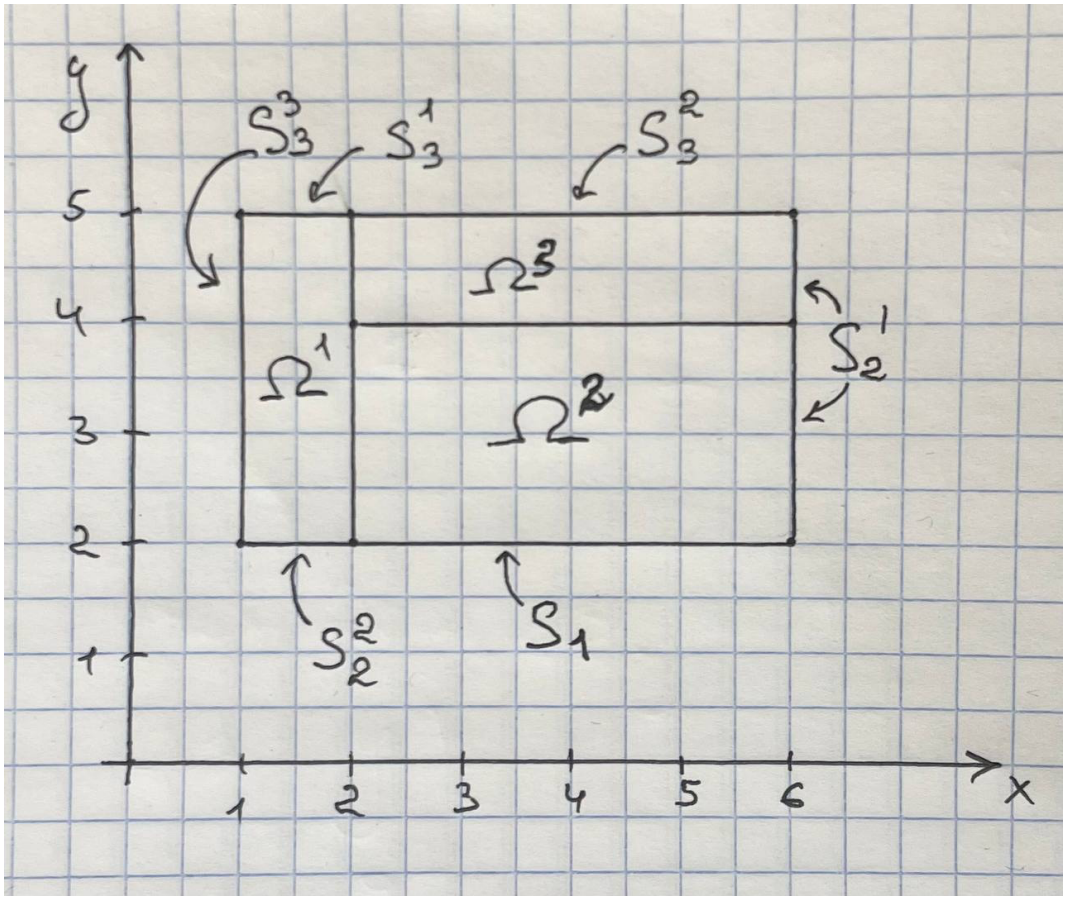
x	y	q_i^h	q_i^*	$\frac{\sqrt{\sum_{i=1}^{n}(q_i^h - q_i^*)^2}}{\sqrt{\sum_{i=1}^{n}(q_i^*)^2}}$
1	1	1,0000000000000000E+00	1,0000000000000000E+00	0,0000000000000000E+00
2	1	2,0000000000000000E+00	2,0000000000000000E+00	

1	2	1,0000000000000000E+00	1,0000000000000000E+00	
2	2	2,0000000000000000E+00	2,0000000000000000E+00	
6	2	2,4000000000000000E+00	2,4000000000000000E+00	
1	4	1,0000000000000000E+00	1,0000000000000000E+00	
2	4	2,0000000000000000E+00	2,0000000000000000E+00	
6	4	2,4000000000000000E+00	2,4000000000000000E+00	
1	5	1,0000000000000000E+00	1,0000000000000000E+00	
2	5	2,0000000000000000E+00	2,0000000000000000E+00	
6	5	2,4000000000000000E+00	2,4000000000000000E+00	

Ожидаемо, погрешность отсутствует, так как ненулевая погрешность возникает на полиномах, степень которых выше степени базисных функций.

5.2 Порядок аппроксимации

Дальнейшие исследования будем проводить на расчетной области, получаемой из предыдущей расчётной области вырезом квадрата [1, 2]×[1, 2], представленной на рисунке:



Первая исследуемая функция:

$$u = x\,,$$

$$\lambda = 1\,,\quad \gamma = 1\,,\quad f = x\,,$$

$$u|_{S_1} = x\,,\ \lambda\frac{\partial u}{\partial n}|_{S_2^1} = 1\,,\ \lambda\frac{\partial u}{\partial n}|_{S_2^2} = 0\,,\ \left(\lambda\frac{\partial u}{\partial n} + (u - x)\right)|_{S_3^1} = 0\,,$$

$$\left(\lambda\frac{\partial u}{\partial n} + 2(u - x)\right)|_{S_3^2} = 0\,,\ \left(\lambda\frac{\partial u}{\partial n} + 0.5(u + 1)\right)|_{S_3^3} = 0\,.$$

Результаты:

x	y	q_i^h	q_i^*	$\frac{\sqrt{\sum_{i=1}^{n}(q_i^h - q_i^*)^2}}{\sqrt{\sum_{i=1}^{n}(q_i^*)^2}}$
1	2	1,0000000000000000E+00	1,0000000000000000E+00	0,0000000000000000E+00
2	2	2,0000000000000000E+00	2,0000000000000000E+00	
6	2	6,0000000000000000E+00	6,0000000000000000E+00	
1	4	1,0000000000000000E+00	1,0000000000000000E+00	
2	4	2,0000000000000000E+00	2,0000000000000000E+00	
6	4	6,0000000000000000E+00	6,0000000000000000E+00	
1	5	1,0000000000000000E+00	1,0000000000000000E+00	
2	5	2,0000000000000000E+00	2,0000000000000000E+00	
6	5	6,0000000000000000E+00	6,0000000000000000E+00	

Ожидаемо, здесь погрешность также отсутствует, поскольку ненулевая погрешность возникает на полиномах, степень которых выше степени базисных функций.

Повысим степень полинома до двух и получим ещё одну тестовую задачу:

$$u = x^2\,,$$

$$\lambda = 1\,,\quad \gamma = 1\,,\quad f = x^2 - 2\ ,$$

$$u|_{S_1} = x^2\,,\ \lambda\frac{\partial u}{\partial n}|_{S_2^1} = 12\,,\ \lambda\frac{\partial u}{\partial n}|_{S_2^2} = 0\,,\ \left(\lambda\frac{\partial u}{\partial n} + (u - x^2)\right)|_{S_3^1} = 0\,,$$

$$\left(\lambda\frac{\partial u}{\partial n} + 2(u - x^2)\right)|_{S_3^2} = 0\,,\ \left(\lambda\frac{\partial u}{\partial n} + 0.5(u + 3)\right)|_{S_3^3} = 0\,.$$

Результаты:

x	y	q_i^h	q_i^*	$\frac{\sqrt{\sum_{i=1}^{n}(q_i^h - q_i^*)^2}}{\sqrt{\sum_{i=1}^{n}(q_i^*)^2}}$
1	2	1,2860931707132279E+00	1,0000000000000000E+00	1,3387307557450754E-02
2	2	3,9999999999999987E+00	4,0000000000000000E+00	
6	2	3,5768393817192398E+01	3,6000000000000000E+01	
1	4	1,2925180025564264E+00	1,0000000000000000E+00	
2	4	4,0757651212741742E+00	4,0000000000000000E+00	
6	4	3,5971835308986111E+01	3,6000000000000000E+01	
1	5	1,2187885057609292E+00	1,0000000000000000E+00	
2	5	4,0308315357848134E+00	4,0000000000000000E+00	
6	5	3,5988776572073182E+01	3,6000000000000000E+01	

Возникает погрешность, с точки зрения теории это связано с тем, что не существует точного представления квадратичной функции с помощью линейных базисных функций, на практике же имеет место ещё и вычислительная погрешность.

5.3 Порядок сходимости

Будем проводить дробление сетки для расчетной области, представленной в пункте 5.2. Для исследования возьмем не являющуюся полиномом функцию:

$$u = \sin(x)\,,$$

$$\lambda = 1\,,\quad \gamma = 1\,,\quad f = 2\sin(x)\,,$$

$$u|_{S_1} = \sin(x)\,,\ \lambda\frac{\partial u}{\partial n}|_{S_2^1} = \cos 6\,,\ \lambda\frac{\partial u}{\partial n}|_{S_2^2} = 0\,,\ \left(\lambda\frac{\partial u}{\partial n} + (u - \sin(x))\right)|_{S_3^1} = 0\,,$$

$$\left(\lambda\frac{\partial u}{\partial n} + 2(u - \sin(x))\right)|_{S_3^2} = 0\,,\ \left(\lambda\frac{\partial u}{\partial n} + 0.5(u + 2\cos 1 - \sin 1)\right)|_{S_3^3} = 0\,.$$

Результаты работы программы до дробления сетки:

x	y	q_i^h	q_i^*	$q_i^* - q_i^h$
1	2	8,5231600696964620E-01	8,4147098480789650E-01	1,0845022161749651E-02
2	2	9,0929742682568150E-01	9,0929742682568170E-01	-2,2204460492503130E-16
6	2	-2,7941549819892575E-01	-2,7941549819892586E-01	1,1102230246251565E-16
1	4	8,7505557900173760E-01	8,4147098480789650E-01	3,3584594193841080E-02
2	4	1,1043204526066719E+00	9,0929742682568170E-01	4,2197956368008980E-01
6	4	3,7288176012809110E-01	-2,7941549819892586E-01	6,5229725832701700E-01
1	5	8,6884306037337060E-01	8,4147098480789650E-01	2,7372075565474052E-02
2	5	1,0052349477211610E+00	9,0929742682568170E-01	9,5937520895479360E-02
6	5	1,9707942772735060E-02	-2,7941549819892586E-01	2,9912344097166094E-01

$$\frac{\sqrt{\sum_{i=1}^{n}(q_i^h - q_i^*)^2}}{\sqrt{\sum_{i=1}^{n}(q_i^*)^2}} = 3{,}4146228062369227\text{E-}01;$$

Результаты работы программы на тех же узлах после дробления сетки в два раза:

x	y	q_i^h	q_i^*	$q_i^* - q_i^h$
1	2	8,2823971449172170E-01	8,4147098480789650E-01	-1,3231270316174770E-02
2	2	9,0929742682568170E-01	9,0929742682568170E-01	0,0000000000000000E+00
6	2	-2,7941549819892580E-01	-2,7941549819892586E-01	5,5511151231257830E-17
1	4	8,2341227531128770E-01	8,4147098480789650E-01	-1,8058709496608816E-02
2	4	1,3312769905057715E+00	9,0929742682568170E-01	1,9502302578099018E-01
6	4	-9,1695470264034000E-02	-2,7941549819892586E-01	1,8772002793489184E-01
1	5	8,3013317694662240E-01	8,4147098480789650E-01	-1,1337807861274140E-02
2	5	8,9031187760996210E-01	9,0929742682568170E-01	-1,8985549215719600E-02
6	5	-1,8865819486337132E-01	-2,7941549819892586E-01	9,0757303335554540E-02

$\frac{\sqrt{\sum_{i=1}^{n}(q_i^h - q_i^*)^2}}{\sqrt{\sum_{i=1}^{n}(q_i^*)^2}}$ = 2,1444864838390684E-01; Отношение погрешностей ≈ 1,71.

Результаты работы программы на тех же узлах после дробления сетки в четыре раза:

x	y	q_i^h	q_i^*	$q_i^* - q_i^h$
1	2	8,3672210041960170E-01	8,4147098480789650E-01	-4,7488843882947940E-03
2	2	9,0929742682568230E-01	9,0929742682568170E-01	5,5511151231257830E-16
6	2	-2,7941549819892597E-01	-2,7941549819892586E-01	-1,1102230246251565E-16
1	4	8,3528185196886330E-01	8,4147098480789650E-01	-6,1891328390332180E-03
2	4	8,9707162698552810E-01	9,0929742682568170E-01	-1,2225799840153595E-02
6	4	-3,7647890221938785E-01	-2,7941549819892586E-01	-9,7063404020461990E-02
1	5	8,3763418021928040E-01	8,4147098480789650E-01	-3,8368045886161273E-03
2	5	9,0220312091233480E-01	9,0929742682568170E-01	-7,0943059133469210E-03
6	5	-3,0690538010552120E-01	-2,7941549819892586E-01	-2,7489881906595348E-02

$\frac{\sqrt{\sum_{i=1}^{n}(q_i^h - q_i^*)^2}}{\sqrt{\sum_{i=1}^{n}(q_i^*)^2}}$ = 4,6476559284704455E-02; Отношение погрешностей ≈ 4,11.

6. Текст программы

Программа написана на объектно-ориентированном языке программирования Java.

```
public class Main {
    public static void main(String[] args) {
        List<BiFunction<Double, Double, Double>> lambdaFunctions = List.of(
                (x, y) -> 1.,
                (x, y) -> 10.,
                (x, y) -> 10.
        );

        List<BiFunction<Double, Double, Double>> gammaFunctions = List.of(
                (x, y) -> 2.,
                (x, y) -> 1.,
                (x, y) -> 0.
        );

        List<BiFunction<Double, Double, Double>> rightFunctions = List.of(
                (x, y) -> 2. * x,
                (x, y) -> 1.8 + 0.1 * x,
                (x, y) -> 0.
        );

        List<BiFunction<Double, Double, Double>> firstKind = List.of(
                (x, y) -> 2.,
                (x, y) -> 0.1 * x + 1.8
        );
        List<BiFunction<Double, Double, Double>> secondKind = List.of(
                (x, y) -> 1.,
                (x, y) -> 0.
        );
        List<BiFunction<Double, Double, Double>> thirdKind = List.of(
                (x, y) -> x,
                (x, y) -> 0.1 * x + 1.8,
                (x, y) -> -1.
        );
        List<Double> beta = List.of(1., 2., 0.5);

        File areaDescription = new File(directoryPath + "AreaDescription.txt");
        File boundariesDescription = new File(directoryPath + "BoundariesDescription.txt");
        File intervalsDescription = new File(directoryPath + "IntervalsDescription.txt");

        Equation equation = new Equation(lambdaFunctions, gammaFunctions, rightFunctions);

        Area area = new Area(areaDescription, equation);
        Grid grid = new Grid(area, intervalsDescription);

        BoundaryConditions conditions = new BoundaryConditions(boundariesDescription,
                firstKind, secondKind, thirdKind, beta);

        SolutionArea solutionArea = new SolutionArea(area, grid, conditions);
        Slae slae = solutionArea.assembleSlae();

        Vector result = slae.solve();
    }
}
```

```
public class Equation {
    List<BiFunction<Double, Double, Double>> lambda;
    List<BiFunction<Double, Double, Double>> gamma;
    List<BiFunction<Double, Double, Double>> rightFunctions;

    public Equation(List<BiFunction<Double, Double, Double>> lambda,
                    List<BiFunction<Double, Double, Double>> gamma,
                    List<BiFunction<Double, Double, Double>> rightFunctions) {
        this.lambda = lambda;
```

```
        this.gamma = gamma;
        this.rightFunctions = rightFunctions;
    }
}


public class Area {
    Subarea[] subareas;
    double[] xValues;
    double[] yValues;

    Equation equation;

    public Area(File description, Equation equation) {
        try (Scanner scanner = new Scanner(description)) {
            int xValuesCount = scanner.nextInt();
            xValues = new double[xValuesCount];

            for (int i = 0; i < xValuesCount; ++i) {
                xValues[i] = scanner.nextDouble();
            }

            int yValuesCount = scanner.nextInt();
            yValues = new double[yValuesCount];

            for (int i = 0; i < yValuesCount; ++i) {
                yValues[i] = scanner.nextDouble();
            }

            int subareasCount = scanner.nextInt();
            subareas = new Subarea[subareasCount];

            for (int i = 0; i < subareasCount; ++i) {
                int subareaIndex = scanner.nextInt();
                subareas[i] = new Subarea(this, subareaIndex,
                        scanner.nextInt(),
                        scanner.nextInt(),
                        scanner.nextInt(),
                        scanner.nextInt()
                );
            }

            this.equation = equation;
        } catch (FileNotFoundException e) {
            System.out.println("Error reading file \""
                    + description.getName() + "\": " + e);
            System.exit(0);
        }
    }


class Subarea {
    final int index;
    final Area parentArea;

    final int xStartIndex;
    final int xEndIndex;

    final int yStartIndex;
    final int yEndIndex;

    Subarea(Area parentArea, int index,
                    int xStartIndex, int xEndIndex, int yStartIndex, int yEndIndex) {
        this.parentArea = parentArea;
        this.index = index;

        this.xEndIndex = xEndIndex;
        this.xStartIndex = xStartIndex;
        this.yStartIndex = yStartIndex;
        this.yEndIndex = yEndIndex;
    }
```

```
public class Grid {
    double[] xValues;
    double[] yValues;

    public Grid(Area area, File description) {
        try (Scanner scanner = new Scanner(description)) {
            xValues = getIntervalsBorders(scanner, area.xValues);
            yValues = getIntervalsBorders(scanner, area.yValues);
            System.out.println(Arrays.toString(xValues));
        } catch (FileNotFoundException e) {
            System.out.println("Error reading file \"" + description.getName() + "\"" + e);
            System.exit(0);
        }
    }

    public double[] getXValues() {
        return xValues;
    }

    public double[] getYValues() {
        return yValues;
    }

    private double[] getIntervalsBorders(Scanner scanner, double[] sourceValues) {
        int totalBordersQuantity = 0;
        int subareasCount = sourceValues.length - 1;

        int[] intervalsBordersQuantities = new int[subareasCount];
        double[] compressionCoefficients = new double[subareasCount];

        for (int i = 0; i < subareasCount; ++i) {
            int subareaIntervalsQuantity = scanner.nextInt();
            totalBordersQuantity += subareaIntervalsQuantity;

            intervalsBordersQuantities[i] = subareaIntervalsQuantity;
            compressionCoefficients[i] = scanner.nextDouble();
        }

        double[] intervalsBorders = new double[totalBordersQuantity + 1];

        for (int i = 0, index = 0; i < subareasCount; ++i) {
            int intervalBordersQuantity = intervalsBordersQuantities[i];
            double compressionCoefficient = compressionCoefficients[i];

            double startBorder = sourceValues[i];
            double endBorder = sourceValues[i + 1];

            if (compressionCoefficient == 1) {
                double step = (endBorder - startBorder) / intervalBordersQuantity;

                for (int j = 0; j < intervalBordersQuantity; ++j) {
                    intervalsBorders[index + j] = startBorder + step * j;
                }
            } else {
                double base = (endBorder - startBorder)
                        / (Math.pow(compressionCoefficient, intervalBordersQuantity) - 1);

                for (int j = 0; j < intervalBordersQuantity; ++j) {
                    intervalsBorders[index + j] = Math.round((startBorder
                            + base * (Math.pow(compressionCoefficient, j) - 1)) * 100.) /
100.;
                }
            }

            intervalsBorders[index + intervalBordersQuantity] = endBorder;
            index += intervalBordersQuantity;
        }
```

```
            return intervalsBorders;
        }
    }


public class BoundaryConditions {
    Map<Integer, List<List<Integer>>> edges;

    List<BiFunction<Double, Double, Double>> firstKind;
    List<BiFunction<Double, Double, Double>> secondKind;
    List<BiFunction<Double, Double, Double>> thirdKind;
    List<Double> beta;

    public static final int X_START_POSITION = 1;
    public static final int Y_START_POSITION = 3;

    public static final int FIRST_CONDITION_NUMBER = 1;
    public static final int SECOND_CONDITION_NUMBER = 2;
    public static final int THIRD_CONDITION_NUMBER = 3;

    public static final DenseMatrix LOCAL_MATRIX = new DenseMatrix(new double[][]{{2, 1}, {1,
2}});
    public static final double COMMON_LOCAL_MATRIX_DIVISOR = 6;

    public BoundaryConditions(File description,
                              List<BiFunction<Double, Double, Double>> firstKind,
                              List<BiFunction<Double, Double, Double>> secondKind,
                              List<BiFunction<Double, Double, Double>> thirdKind,
                              List<Double> beta) {
        final int conditionsMaxCount = 3;
        final int propertiesCount = 5;

        try (Scanner scanner = new Scanner(description)) {
            edges = new HashMap<>(conditionsMaxCount);

            for (int i = 1; i <= conditionsMaxCount; ++i) {
                edges.put(i, new ArrayList<>());
            }

            while (scanner.hasNext()) {
                int typeNumber = scanner.nextInt();
                List<List<Integer>> type = edges.get(typeNumber);
                List<Integer> edge = new ArrayList<>();

                for (int i = 0; i < propertiesCount; ++i) {
                    edge.add(scanner.nextInt());
                }

                type.add(edge);
                edges.put(typeNumber, type);
            }

            this.firstKind = firstKind;
            this.secondKind = secondKind;
            this.thirdKind = thirdKind;
            this.beta = beta;
        } catch (FileNotFoundException e) {
            System.out.println("Error reading file \"" + description.getName() + "\"" + e);
            System.exit(0);
        }
    }
}


class FiniteElement {
    SolutionArea solutionArea;
    int subAreaIndex;

    double lambdaAverage;
    double gammaAverage;
    Vector functionValues = new Vector(NODES_COUNT);
```

```
    int xStartIndex;
    int yStartIndex;

    static final int EDGE_NODES_COUNT = 2;
    static final int NODES_COUNT = 4;

    FiniteElement(SolutionArea solutionArea, int xStartIndex, int yStartIndex) {
        this.solutionArea = solutionArea;

        this.xStartIndex = xStartIndex;
        this.yStartIndex = yStartIndex;

        subAreaIndex = getSubareaIndex();

        if (isSignificant()) {
            lambdaAverage =
getNodesFunctionAverage(solutionArea.area.equation.lambda.get(subAreaIndex));
            gammaAverage =
getNodesFunctionAverage(solutionArea.area.equation.gamma.get(subAreaIndex));
            functionValues =
getFunctionValues(solutionArea.area.equation.rightFunctions.get(subAreaIndex));
        }
    }

    public boolean isSignificant() {
        return subAreaIndex >= 0;
    }

    private int getSubareaIndex() {
        double[] gridXValues = solutionArea.grid.xValues;
        double[] gridYValues = solutionArea.grid.yValues;

        double[] areaXValues = solutionArea.area.xValues;
        double[] areaYValues = solutionArea.area.yValues;

        for (int i = 0; i < solutionArea.area.subareas.length; ++i) {
            Subarea subarea = solutionArea.area.subareas[i];

            if (Double.compare(gridXValues[xStartIndex], areaXValues[subarea.xStartIndex]) >=
0
                    && Double.compare(gridXValues[xStartIndex + 1],
areaXValues[subarea.xEndIndex]) <= 0
                    && Double.compare(gridYValues[yStartIndex],
areaYValues[subarea.yStartIndex]) >= 0
                    && Double.compare(gridYValues[yStartIndex + 1],
areaYValues[subarea.yEndIndex]) <= 0) {
                return subarea.index;
            }
        }

        return -1;
    }

    private double getNodesFunctionAverage(BiFunction<Double, Double, Double> function) {
        double average = 0;

        for (int i = 0; i < EDGE_NODES_COUNT; ++i) {
            for (int j = 0; j < EDGE_NODES_COUNT; ++j) {
                average += function.apply(
                        solutionArea.grid.xValues[xStartIndex + j],
                        solutionArea.grid.yValues[yStartIndex + i]
                );
            }
        }

        return average / NODES_COUNT;
    }

    private Vector getFunctionValues(BiFunction<Double, Double, Double> function) {
        double[] functionValues = new double[NODES_COUNT];
```

```
        for (int i = 0, index = 0; i < EDGE_NODES_COUNT; ++i) {
            for (int j = 0; j < EDGE_NODES_COUNT; ++j) {
                functionValues[index] = function.apply(
                        solutionArea.grid.xValues[xStartIndex + j],
                        solutionArea.grid.yValues[yStartIndex + i]
                );
                ++index;
            }
        }

        return new Vector(functionValues);
    }

    Matrix getLocalMatrixCoefficients() {
        Matrix matrix = generateStiffnessMatrix();
        matrix.add(generateWeightMatrix(gammaAverage));
        return matrix;
    }

    Vector getLocalConstantTermsVector() {
        final double gamma = 1;
        return generateWeightMatrix(gamma).multiplyByVector(functionValues);
    }

    private Matrix generateStiffnessMatrix() {
        double heightX = solutionArea.grid.xValues[xStartIndex + 1] -
solutionArea.grid.xValues[xStartIndex];
        double heightY = solutionArea.grid.yValues[yStartIndex + 1] -
solutionArea.grid.yValues[yStartIndex];

        double squaredHeightX = heightX * heightX;
        double squaredHeightY = heightY * heightY;

        final double commonMultiplierConstant = 6;
        double commonMultiplier = lambdaAverage / (commonMultiplierConstant * heightX *
heightY);

        final double[][] leftStiffnessMatrixCoefficients = new double[][]{
                new double[]{2, -2, 1, -1},
                new double[]{-2, 2, -1, 1},
                new double[]{1, -1, 2, -2},
                new double[]{-1, 1, -2, 2}
        };

        final double[][] rightStiffnessMatrixCoefficients = new double[][]{
                new double[]{2, 1, -2, -1},
                new double[]{1, 2, -1, -2},
                new double[]{-2, -1, 2, 1},
                new double[]{-1, -2, 1, 2}
        };

        final int stiffnessMatrixDimension = leftStiffnessMatrixCoefficients.length;
        double[][] components = new
double[stiffnessMatrixDimension][stiffnessMatrixDimension];

        for (int i = 0; i < stiffnessMatrixDimension; ++i) {
            for (int j = 0; j < stiffnessMatrixDimension; ++j) {
                components[i][j] = (squaredHeightY * leftStiffnessMatrixCoefficients[i][j]
                        + squaredHeightX * rightStiffnessMatrixCoefficients[i][j])
                        * commonMultiplier;
            }
        }

        return new DenseMatrix(components);
    }

    private Matrix generateWeightMatrix(double gamma) {
        final double commonMultiplierConstant = 36;
        double commonMultiplier = gamma
                * (solutionArea.grid.xValues[xStartIndex + 1] -
```

```
solutionArea.grid.xValues[xStartIndex])
                * (solutionArea.grid.yValues[yStartIndex + 1] -
solutionArea.grid.yValues[yStartIndex])
                / commonMultiplierConstant;

        final double[][] weightMatrixCoefficients = new double[][]{
                new double[]{4, 2, 2, 1},
                new double[]{2, 4, 1, 2},
                new double[]{2, 1, 4, 2},
                new double[]{1, 2, 2, 4}
        };

        final int weightMatrixDimension = weightMatrixCoefficients.length;
        double[][] components = new double[weightMatrixDimension][weightMatrixDimension];

        for (int i = 0; i < weightMatrixDimension; ++i) {
            for (int j = 0; j < weightMatrixDimension; ++j) {
                components[i][j] = commonMultiplier * weightMatrixCoefficients[i][j];
            }
        }

        return new DenseMatrix(components);
    }
}

class Node {
    int xIndex;
    int yIndex;
    int globalIndex;

    Node(int xIndex, int yIndex, int globalIndex) {
        this.xIndex = xIndex;
        this.yIndex = yIndex;
        this.globalIndex = globalIndex;
    }
}
public class SolutionArea {
    Area area;
    Grid grid;

    BoundaryConditions conditions;

    public SolutionArea(Area area, Grid grid, BoundaryConditions conditions) {
        this.area = area;
        this.grid = grid;
        this.conditions = conditions;
    }

    static class RowElement {
        int columnIndex;
        double value;

        RowElement(int columnIndex, double value) {
            this.columnIndex = columnIndex;
            this.value = value;
        }
    }

    public Slae assembleSlae() {
        int nodesCount = grid.xValues.length * grid.yValues.length;

        double[][] matrixComponents = new double[nodesCount][nodesCount];
        double[] vectorComponents = new double[nodesCount];

        int yValuesLastIndex = grid.yValues.length - 1;
        int xValuesLastIndex = grid.xValues.length - 1;

        for (int i = 0; i < yValuesLastIndex; ++i) {
            for (int j = 0; j < xValuesLastIndex; ++j) {
                FiniteElement element = new FiniteElement(this, j, i);
```

```
                Matrix localMatrix = element.getLocalMatrixCoefficients();
                Vector localVector = element.getLocalConstantTermsVector();

                Node[] nodes = new Node[NODES_COUNT];

                for (int k = 0, index = 0; k < EDGE_NODES_COUNT; ++k) {
                    int yIndex = element.yStartIndex + k;

                    for (int l = 0; l < EDGE_NODES_COUNT; ++index, ++l) {
                        int xIndex = element.xStartIndex + l;

                        nodes[index] = new Node(xIndex, yIndex, getNodeGlobalIndex(xIndex,
yIndex));
                    }
                }

                for (int k = 0; k < NODES_COUNT; ++k) {
                    vectorComponents[nodes[k].globalIndex] += localVector.getComponent(k);

                    for (int l = 0; l < NODES_COUNT; ++l) {
                        matrixComponents[nodes[k].globalIndex][nodes[l].globalIndex] +=
localMatrix.getComponent(k, l);
                    }
                }
            }
        }

        Matrix matrix = new SparseRowMatrix(matrixComponents);
        Vector vector = new Vector(vectorComponents);

        setConditions(matrix, vector);
        return new Slae(matrix, vector);
    }

    private int getNodeGlobalIndex(int xIndex, int yIndex) {
        return yIndex * grid.xValues.length + xIndex;
    }

    private void setConditions(Matrix matrix, Vector vector) {
        setSecondConditions(vector);
        setThirdConditions(matrix, vector);
        setFirstConditions(matrix, vector);
    }

    private void setFirstConditions(Matrix matrix, Vector vector) {
        List<List<Integer>> edgesType1 =
conditions.edges.get(BoundaryConditions.FIRST_CONDITION_NUMBER);

        for (List<Integer> edge : edgesType1) {
            int edgeIndex = edge.getFirst();
            BiFunction<Double, Double, Double> function = conditions.firstKind.get(edgeIndex);

            int xStart = edge.get(BoundaryConditions.X_START_POSITION);
            int xEnd = edge.get(BoundaryConditions.X_START_POSITION + 1);

            int yStart = edge.get(BoundaryConditions.Y_START_POSITION);
            int yEnd = edge.get(BoundaryConditions.Y_START_POSITION + 1);

            if (xEnd == xStart) {
                for (int i = yStart + 1; i <= yEnd; ++i) {
                    setFirstConditionOnElement(xStart, xEnd, i - 1, i, matrix, vector,
function);
                }
            } else {
                for (int i = xStart + 1; i <= xEnd; ++i) {
                    setFirstConditionOnElement(i - 1, i, yStart, yEnd, matrix, vector,
function);
                }
            }
        }
    }
```

```
    private void setSecondConditions(Vector vector) {
        List<List<Integer>> edgesType2 =
conditions.edges.get(BoundaryConditions.SECOND_CONDITION_NUMBER);

        for (List<Integer> edge : edgesType2) {
            BiFunction<Double, Double, Double> thetaFunction =
conditions.secondKind.get(edge.getFirst());

            int xStart = edge.get(BoundaryConditions.X_START_POSITION);
            int xEnd = edge.get(BoundaryConditions.X_START_POSITION + 1);

            int yStart = edge.get(BoundaryConditions.Y_START_POSITION);
            int yEnd = edge.get(BoundaryConditions.Y_START_POSITION + 1);

            if (xEnd == xStart) {
                for (int i = yStart + 1; i <= yEnd; ++i) {
                    int yElementStart = i - 1;
                    setSecondConditionOnElement(xStart, xEnd, yElementStart, i,
                            grid.yValues[i] - grid.yValues[yElementStart], vector,
thetaFunction);
                }
            } else {
                for (int i = xStart + 1; i <= xEnd; ++i) {
                    int xElementStart = i - 1;
                    setSecondConditionOnElement(xElementStart, i, yStart, yEnd,
                            grid.xValues[i] - grid.xValues[xElementStart], vector,
thetaFunction);
                }
            }
        }
    }

    private void setThirdConditions(Matrix matrix, Vector vector) {
        List<List<Integer>> edgesType3 =
conditions.edges.get(BoundaryConditions.THIRD_CONDITION_NUMBER);

        for (List<Integer> edge : edgesType3) {
            int edgeIndex = edge.getFirst();

            BiFunction<Double, Double, Double> betaFunction =
conditions.thirdKind.get(edgeIndex);
            double beta = conditions.beta.get(edgeIndex);

            int xStart = edge.get(BoundaryConditions.X_START_POSITION);
            int xEnd = edge.get(BoundaryConditions.X_START_POSITION + 1);

            int yStart = edge.get(BoundaryConditions.Y_START_POSITION);
            int yEnd = edge.get(BoundaryConditions.Y_START_POSITION + 1);

            if (xEnd == xStart) {
                for (int i = yStart + 1; i <= yEnd; ++i) {
                    int yElementStart = i - 1;
                    double commonMultiplier = beta * (grid.yValues[i] -
grid.yValues[yElementStart])
                            / BoundaryConditions.COMMON_LOCAL_MATRIX_DIVISOR;

                    setThirdConditionOnElement(xStart, xEnd, yElementStart, i,
commonMultiplier,
                            matrix, vector, betaFunction);
                }
            } else {
                for (int i = xStart + 1; i <= xEnd; ++i) {
                    int xElementStart = i - 1;
                    double commonMultiplier = beta * (grid.xValues[i] -
grid.xValues[xElementStart])
                            / BoundaryConditions.COMMON_LOCAL_MATRIX_DIVISOR;

                    setThirdConditionOnElement(xElementStart, i, yStart, yEnd,
commonMultiplier,
                            matrix, vector, betaFunction);
```

```
                }
            }
        }
    }

    private void setFirstConditionOnElement(int xStart, int xEnd, int yStart, int yEnd,
                                            Matrix matrix, Vector vector,
                                            BiFunction<Double, Double, Double> function) {
        int[] nodesGlobalIndexes = new int[EDGE_NODES_COUNT];

        nodesGlobalIndexes[0] = getNodeGlobalIndex(xStart, yStart);
        nodesGlobalIndexes[1] = getNodeGlobalIndex(xEnd, yEnd);

        matrix.resetRow(nodesGlobalIndexes[0]);
        matrix.resetRow(nodesGlobalIndexes[1]);

        matrix.setComponent(nodesGlobalIndexes[0], nodesGlobalIndexes[0], 1);
        matrix.setComponent(nodesGlobalIndexes[1], nodesGlobalIndexes[1], 1);

        vector.setComponent(nodesGlobalIndexes[0], function.apply(grid.xValues[xStart],
grid.yValues[yStart]));
        vector.setComponent(nodesGlobalIndexes[1], function.apply(grid.xValues[xEnd],
grid.yValues[yEnd]));
    }

    private void setSecondConditionOnElement(int xStart, int xEnd, int yStart, int yEnd,
                                             double elementEdgeLength, Vector vector,
                                             BiFunction<Double, Double, Double> function) {
        double[] theta = new double[EDGE_NODES_COUNT];
        int[] nodesGlobalIndexes = new int[EDGE_NODES_COUNT];

        nodesGlobalIndexes[0] = getNodeGlobalIndex(xStart, yStart);
        nodesGlobalIndexes[1] = getNodeGlobalIndex(xEnd, yEnd);

        theta[0] = elementEdgeLength
                * function.apply(grid.xValues[xStart], grid.yValues[yStart])
                / BoundaryConditions.COMMON_LOCAL_MATRIX_DIVISOR;
        theta[1] = elementEdgeLength
                * function.apply(grid.xValues[xEnd], grid.yValues[yEnd])
                / BoundaryConditions.COMMON_LOCAL_MATRIX_DIVISOR;

        Vector localVector = BoundaryConditions.LOCAL_MATRIX.multiplyByVector(new
Vector(theta));

        for (int j = 0; j < EDGE_NODES_COUNT; ++j) {
            vector.increaseComponent(nodesGlobalIndexes[j], localVector.getComponent(j));
        }
    }

    private void setThirdConditionOnElement(int xStart, int xEnd, int yStart, int yEnd,
                                            double commonMultiplier, Matrix matrix, Vector
vector,
                                            BiFunction<Double, Double, Double> function) {
        double[] vectorComponents = new double[EDGE_NODES_COUNT];
        int[] nodesGlobalIndexes = new int[EDGE_NODES_COUNT];

        nodesGlobalIndexes[0] = getNodeGlobalIndex(xStart, yStart);
        nodesGlobalIndexes[1] = getNodeGlobalIndex(xEnd, yEnd);

        vectorComponents[0] = commonMultiplier
                * function.apply(grid.xValues[xStart], grid.yValues[yStart]);
        vectorComponents[1] = commonMultiplier
                * function.apply(grid.xValues[xEnd], grid.yValues[yEnd]);

        Vector localVector = BoundaryConditions.LOCAL_MATRIX.multiplyByVector(new
Vector(vectorComponents));

        for (int j = 0; j < EDGE_NODES_COUNT; ++j) {
            vector.increaseComponent(nodesGlobalIndexes[j], localVector.getComponent(j));
        }
```

```
        DenseMatrix localMatrix = new DenseMatrix(BoundaryConditions.LOCAL_MATRIX);
        localMatrix.multiplyByScalar(commonMultiplier);

        for (int j = 0; j < EDGE_NODES_COUNT; ++j) {
            for (int k = 0; k < EDGE_NODES_COUNT; ++k) {
                matrix.increaseComponent(nodesGlobalIndexes[j], nodesGlobalIndexes[k],
                        localMatrix.getComponent(j, k));
            }
        }
    }


public interface Matrix {
    int getRowsCount();
    int getColumnsCount();
    double getComponent(int rowIndex, int columnIndex);
    void setComponent(int rowIndex, int columnIndex, double component);
    Vector getRow(int index);
    void resetRow(int index);
    void add(Matrix matrix);
    DenseMatrix assemble();
    void increaseComponent(int rowIndex, int columnIndex, double component);
    Vector multiplyByVector(Vector vector);
}


public class SparseRowMatrix implements Matrix {
    int rowsCount;
    int columnsCount;

    double[] values;
    int[] columnsIndexes;
    int[] pointers;

    public SparseRowMatrix(double[] values, int[] columnsIndexes, int[] pointers) {
        this.values = values;
        this.columnsIndexes = columnsIndexes;
        this.pointers = pointers;

        rowsCount = pointers.length - 1;
        int maxIndex = -1;

        for (int index : columnsIndexes) {
            if (index > maxIndex) {
                maxIndex = index;
            }
        }

        columnsCount = maxIndex + 1;
    }

    public SparseRowMatrix(DenseMatrix matrix) {
        rowsCount = matrix.getRowsCount();
        columnsCount = matrix.getColumnsCount();
        int maxNodesCount = rowsCount * columnsCount;

        values = new double[maxNodesCount];
        columnsIndexes = new int[maxNodesCount];
        pointers = new int[rowsCount + 1];

        for (int i = 0, index = 0; i < rowsCount; ++i) {
            pointers[i + 1] = pointers[i];

            for (int j = 0; j < columnsCount; ++j) {
                double component = matrix.rows[i].components[j];

                if (Double.compare(component, 0) != 0) {
                    values[index] = component;
                    columnsIndexes[index] = j;
                    ++pointers[i + 1];
```

```
                    ++index;
                }
            }
        }

        values = Arrays.copyOf(values, pointers[rowsCount]);
        columnsIndexes = Arrays.copyOf(columnsIndexes, pointers[rowsCount]);
    }

    public SparseRowMatrix(double[][] components) {
        rowsCount = components.length;
        columnsCount = components[0].length;
        int maxNodesCount = rowsCount * columnsCount;

        values = new double[maxNodesCount];
        columnsIndexes = new int[maxNodesCount];
        pointers = new int[rowsCount + 1];

        for (int i = 0, index = 0; i < rowsCount; ++i) {
            pointers[i + 1] = pointers[i];

            for (int j = 0; j < columnsCount; ++j) {
                double component = components[i][j];

                if (Double.compare(component, 0) != 0) {
                    values[index] = component;
                    columnsIndexes[index] = j;
                    ++pointers[i + 1];
                    ++index;
                }
            }
        }

        values = Arrays.copyOf(values, pointers[rowsCount]);
        columnsIndexes = Arrays.copyOf(columnsIndexes, pointers[rowsCount]);
    }

    public DenseMatrix assemble() {
        double[][] components = new double[rowsCount][columnsCount];

        for (int i = 0, pointer = 0; i < rowsCount; ++i) {
            int elementsCount = pointers[i + 1] - pointers[i];

            for (int j = 0; j < elementsCount; ++j) {
                components[i][columnsIndexes[pointer]] = values[pointer];
                ++pointer;
            }
        }

        return new DenseMatrix(components);
    }
}



public class Vector {
    double[] components;

    public Vector(int size) {
        components = new double[size];
    }

    public Vector(Vector vectorToCopy) {
        components = Arrays.copyOf(vectorToCopy.components, vectorToCopy.components.length);
    }

    public Vector(double[] components) {
        this.components = Arrays.copyOf(components, components.length);
    }

    public Vector(int size, double[] components) {
```

```
        this.components = Arrays.copyOf(components, size);
    }

    public int getSize() {
        return components.length;
    }

    public void add(Vector vector) {
        if (components.length < vector.components.length) {
            components = Arrays.copyOf(components, vector.components.length);
        }

        for (int i = 0; i < vector.components.length; i++) {
            components[i] += vector.components[i];
        }
    }

    public void subtract(Vector vector) {
        if (components.length < vector.components.length) {
            components = Arrays.copyOf(components, vector.components.length);
        }

        for (int i = 0; i < vector.components.length; i++) {
            components[i] -= vector.components[i];
        }
    }

    public void multiplyByScalar(double scalar) {
        for (int i = 0; i < components.length; i++) {
            components[i] *= scalar;
        }
    }

    public double getComponent(int index) {
        validateIndex(index);

        return components[index];
    }

    public void setComponent(int index, double component) {
        components[index] = component;
    }

    public void increaseComponent(int index, double addition) {
        components[index] += addition;
    }

    public static double getDotProduct(Vector vector1, Vector vector2) {
        int minSize = Math.min(vector1.components.length, vector2.components.length);

        double dotProduct = 0;

        for (int i = 0; i < minSize; i++) {
            dotProduct += vector1.components[i] * vector2.components[i];
        }

        return dotProduct;
    }
}


public class DenseMatrix implements Matrix {
    Vector[] rows;

    public DenseMatrix(int rowsCount, int columnsCount) {
        validateRowsCount(rowsCount);
        validateColumnsCount(columnsCount);

        rows = new Vector[rowsCount];
```

```
        for (int i = 0; i < rowsCount; i++) {
            rows[i] = new Vector(columnsCount);
        }
    }

    public DenseMatrix(DenseMatrix matrixToCopy) {
        int matrixToCopyRowsCount = matrixToCopy.getRowsCount();
        rows = new Vector[matrixToCopyRowsCount];

        for (int i = 0; i < matrixToCopyRowsCount; i++) {
            rows[i] = new Vector(matrixToCopy.rows[i]);
        }
    }

    public DenseMatrix(double[][] array) {
        validateRowsCount(array.length);

        int maxRowLength = 0;

        for (double[] rowArray : array) {
            maxRowLength = Math.max(maxRowLength, rowArray.length);
        }

        validateColumnsCount(maxRowLength);
        rows = new Vector[array.length];

        for (int i = 0; i < array.length; i++) {
            rows[i] = new Vector(maxRowLength, array[i]);
        }
    }

    public int getRowsCount() {
        return rows.length;
    }

    public int getColumnsCount() {
        return rows[0].getSize();
    }

    public Vector getRow(int index) {
        validateRowIndex(index);

        return new Vector(rows[index]);
    }

    @Override
    public void resetRow(int index) {
        Arrays.fill(rows[index].components, 0);
    }

    @Override
    public DenseMatrix assemble() {
        return this;
    }

    public double getComponent(int rowIndex, int columnIndex) {
        return rows[rowIndex].getComponent(columnIndex);
    }

    @Override
    public void setComponent(int rowIndex, int columnIndex, double component) {
        rows[rowIndex].setComponent(columnIndex, component);
    }
```

```
    @Override
    public void increaseComponent(int rowIndex, int columnIndex, double addition) {
        rows[rowIndex].setComponent(columnIndex,
rows[rowIndex].getComponent(columnIndex) + addition);
    }

    public void multiplyByScalar(double scalar) {
        for (Vector row : rows) {
            row.multiplyByScalar(scalar);
        }
    }

    public void add(Matrix matrix) {
        validateSizesEquality(matrix);

        int rowsCount = getRowsCount();

        for (int i = 0; i < rowsCount; i++) {
            rows[i].add(matrix.getRow(i));
        }
    }

    public void subtract(DenseMatrix matrix) {
        validateSizesEquality(matrix);

        int rowsCount = getRowsCount();

        for (int i = 0; i < rowsCount; i++) {
            rows[i].subtract(matrix.rows[i]);
        }
    }

    public Vector multiplyByVector(Vector vector) {
        int rowsCount = getRowsCount();
        Vector resultVector = new Vector(rowsCount);

        for (int i = 0; i < rowsCount; i++) {
            resultVector.setComponent(i, Vector.getDotProduct(rows[i], vector));
        }

        return resultVector;
    }
}


public class Slae {
    private final Matrix systemCoefficients;
    private final Vector constantTerms;

    public Slae(Matrix systemCoefficients, Vector constantTerms) {
        this.systemCoefficients = systemCoefficients;
        this.constantTerms = constantTerms;
    }

    public Vector solve() {
        return solveByLocallyOptimalScheme(systemCoefficients, constantTerms);
    }

    public static Vector solveByLocallyOptimalScheme(Matrix A, Vector b) {
        int size = b.getSize();

        Vector result = new Vector(size);
        Vector residual = new Vector(b);
        Vector direction = new Vector(residual);

        do {
```

```
            Vector auxiliaryVector = A.multiplyByVector(direction);
            double auxiliaryVectorSquaredNorm = Vector.getDotProduct(auxiliaryVector,
auxiliaryVector);

            double alpha = Vector.getDotProduct(auxiliaryVector, residual)
                    / auxiliaryVectorSquaredNorm;

            for (int i = 0; i < size; i++) {
                result.components[i] += alpha * direction.components[i];
            }

            for (int i = 0; i < size; i++) {
                residual.components[i] -= alpha * auxiliaryVector.components[i];
            }

            Vector nextResult = A.multiplyByVector(residual);
            double beta = -Vector.getDotProduct(auxiliaryVector, nextResult)
                    / auxiliaryVectorSquaredNorm;

            for (int i = 0; i < size; i++) {
                direction.components[i] = residual.components[i] + beta *
direction.components[i];
            }

            for (int i = 0; i < size; i++) {
                auxiliaryVector.components[i] = nextResult.components[i]
                        + beta * auxiliaryVector.components[i];
            }
        } while (!Double.valueOf(Vector.getDotProduct(residual, residual)).equals(0.));

        return result;
    }
```