МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра Прикладной математики
(полное название кафедры)

Утверждаю
ПМт
**Зав. кафедрой** _____
Ю.Г. Соловейчик
(подпись, инициалы, фамилия)

«___» _____ 201__ г.

**МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ**
по направлению высшего образования

01.04.02 Прикладная математика и информатика
(код и наименование направления подготовки магистра)

Факультет прикладной математики и информатики
(факультет)

Патрушева Ильи Игоревича
(фамилия, имя, отчество студента – автора работы)

Разработка и реализация вычислительных схем решения трёхмерной задачи многофазной фильтрации

(полное название темы магистерской диссертации)

**Руководитель от НГТУ**

Персова Марина Геннадьевна
(фамилия, имя, отчество)

д.т.н., профессор
(ученая степень, ученое звание)

_____
(подпись, дата)

**Автор выпускной квалификационной работы**

Патрушев Илья Игоревич
(фамилия, имя, отчество)

ФПМИ, ПММ-62
(факультет, группа)

_____
(подпись, дата)

Новосибирск 2018

АННОТАЦИЯ

ОТЧЕТ 66 С., 4 Ч., 20 РИС., 5 ТАБЛ., 18 ИСТОЧНИКОВ, 1 ПРИЛ.

МНОГОФАЗНАЯ ФИЛЬТРАЦИЯ, МЕТОД КОНЕЧНЫХ ЭЛЕМЕНТОВ, ЧИСЛЕННОЕ МОДЕЛИРОВАНИЕ, НЕФТЕДОБЫЧА

Объектом исследования является построение аппроксимации для решения трёхмерных задач многофазной фильтрации в пористых средах.

Цель работы – разработка программы для моделирования многофазного потока в пористой среде для случая несжимающихся изотермических фаз.

В работе представлена математическая модель и вычислительные схемы для процесса многофазной фильтрации в пористых средах с учётом многокомпонентности фаз.

Для расчёта давления в пористой среде при заданном распределении насыщенностей, а также, для расчёта потоков смеси через границы ячеек области используется метод конечных элементов. Для выполнения баланса втекающих и вытекающих потоков через границы ячеек области, находятся корректирующие добавки к рассчитанным по конечноэлементному решению потокам смеси из условия минимизации функционала.

Приведены два численных эксперимента: исследование использования полимерного заводнения в целях повышения нефтедобычи, и исследование влияния контакта водоносного и нефтенасыщенного слоев в окрестности добычи.

Степень внедрения – разработанная программа применяется для проведения численных экспериментов при моделировании трёхмерных задач многофазной фильтрации в НОЦ «Моделирования электромагнитных технологий».

ОГЛАВЛЕНИЕ

# ВВЕДЕНИЕ

В нефтедобывающей отрасли актуальными являются задачи моделирования многофазных потоков в пористых средах. Эффективные вычислительные схемы прямого трёхмерного моделирования месторождений и реализующие их программы являются необходимым инструментом для оперативного управления нефтедобычей, а также, инструментом для оптимизации современных технологий добычи трудноизвлекаемой нефти

На многих российских месторождениях прекращена добыча нефти методами, основанными на использовании естественного внутрипластового давления. Это такие методы добычи, при которых нефть фактически фонтанирует из установленной скважины, требуя при этом минимальные затраты для её добычи. Такие методы добычи называют первичными. Эффективность первичных методов в основном определяется точностью попадания скважины в целевой горизонт [1]. За счёт внутренней энергии пласта удаётся извлечь малую часть залежи нефти (20-30%) и большая часть полезного сырья остаётся в пласте. Для добычи остаточной нефти используют специальные методы (вторичные и третичные методы или методы повышающие нефтеотдачу пласта), которые используют искусственные воздействия на нефтенасыщенный пласт, например, закачивание в пласт через группы нагнетательных скважин вязких вытесняющих агентов [2-6].

Высокое значение имеет анализ текущего состояния месторождения для эффективного управления разработкой. Полученная информация о поле распределения насыщенностей фаз в среде, как результат проведения прямого моделирования процесса фильтрации, может служить входными данными для дальнейших задач оптимизации разработки нефтяного месторождения и определения параметров геологической среды. Оптимальное управление разработкой нефтяного месторождения с целью увеличения объёмов добычи практически невозможно без использования специализированных программно-математических пакетов, позволяющих проводить численное моделирование процессов, протекающих в геологических средах. В настоящее время

распространёными программными пакетами, которые используются для контроля за разработками месторождений, являются Roxar RSM, Schlumberger ECLIPSE, CMG STARS и другие. Все они имеют большой функционал и позволяют проводить моделирование состояния месторождений.

Программный комплекс Roxar в своём составе имеет несколько модулей моделирования состояния пласта [7]. Один из таких модулей RoxarRMS stream использует упрощённую фильтрационную модель, основанную на теории линий тока [8]. Программный модуль Tempest MORE EOS реализует метод стохастического моделирования петрофизических свойств, который может использоваться в детерминированной манере в совокупности с методом трендов.

Программный комплекс ECLIPSE, разработанный нефтесервисной компанией использует метод конечных объёмов [9], применение которого обеспечивает соблюдение сохранения потоков между ячейками расчётной области, но имеет ряд недостатков для моделирования месторождений в областях со сложной геометрией.

В данной работе описан метод численного моделирования задачи многофазной фильтрации с использованием метода конечных элементов, который позволяет строить высокоточные аппроксимации решения в областях с геометрией любой сложности, что особенно актуально для моделирования процессов, протекающих в геологической среде. Но метод конечных элементов не позволяет гарантировать выполнения закона сохранения масс, что отмечается в работах [10-13]. Для выполнения балансировки потоков между ячейками расчётной области решается задача минимизации функционала, для поиска корректирующих добавок к потокам, рассчитанным по конечноэлементому решению.

Описанная в данной работе математическая модель позволяет проводить расчёты задачи фильтрации для случая несжимаемых изотермических жидкостей с произвольным количеством фаз и их компонентного состава.

# 1. МАТЕМАТИЧЕСКАЯ МОДЕЛЬ И ЧИСЛЕННЫЙ МЕТОД

В данном разделе представлена математическая модель многофазной фильтрации для случая многокомпонентных несжимаемых и изотермических фаз. В первом пункте главы изложена идея и приведены основанные соотношения модели многофазной фильтрации с использованием метода конечных элементов без подробного описания технологий вычисления величин, необходимых для моделирования процесса, таких как давление, объёмы перетекающих фаз через границы ячеек расчётной области, величина шага дискретизации по времени и другие. Описание технологий расчётов каждой из этих величин приведены в отдельных параграфах.

## 1.1. Математическая модель многофазной фильтрации в пористой среде

Рассмотрим математическую модель процесса многофазной фильтрации в пористых средах с учётом многокомпонентности фаз.

Будем рассматривать процесс фильтрации в области $\Omega$, которая представляет собой пористую среду и характеризуется двумя параметрами: $\Phi$ – структурная пористость породы, и $\mathbf{K}$ – структурная проницаемость породы. Под структурной пористостью $\Phi$ понимают долю объёма пор в объёме породы, по которым могут протекать фильтруемые жидкости, измеряется в долях единицы. Структурная проницаемость породы $\mathbf{K}$ – это тензор, который характеризует способность породы пропускать жидкие флюиды через поры, которая зависит от типа самой породы (например, песчаник, известняк, кварцевый порошок и другие), измеряется в мкм$^2$ или в Дарси (Д). В общем случае, коэффициенты уравнения $\mathbf{K}$ и $\Phi$ являются пространственными функциями и изменяются во времени, но в данной работе будем считать параметры $\Phi$ и $\mathbf{K}$ неизменяющимися во времени кусочно-постоянными функциями. Будем полагать, что в любой момент времени поры среды полностью заполнены многофазной смесью.

Модель фильтрации может быть однофазной, когда рассматривается процесс течения в пористой среде одной жидкости, и многофазной, когда в среде протекает не одна, а смесь из нескольких жидкостей. Под фазой в этом

случае понимается жидкость, входящая в состав фильтруемой смеси, обладающая отличными от других фильтрационными свойствами. Каждая фаза смеси может состоять из нескольких компонент, причём одна компонента может входить в состав нескольких фаз.

Пусть $m$-ая фаза смеси состоит из $L^m$ компонент, и пусть $\chi^{lm}$ – доля содержания $l$-ой компоненты в $m$-ой фазе. Каждая $m$-ая фаза смеси характеризуется параметрами: $\rho^m$ – плотность (измеряется в кг/м$^3$), $\eta^m$ – динамическая вязкость(измеряется в Па·с), которые могут зависеть от компонентного состава фазы, и $\kappa^m$ – множитель к структурной проницаемости породы, который может зависеть от насыщенностей $S^m$ фаз в среде.

Фильтрация представляет собой движение жидкости в пористой среде под действием перепада давления. Скорость фильтрации многофазного потока описывается законом Дарси

$$\vec{u} = \mathbf{K}\sum_{m=1}^{M}\frac{\kappa^m}{\eta^m}\operatorname{grad}\left(P + P_c^m\right), \tag{1}$$

где $\vec{u}$ – скорость фильтрации [14], $M$ – количество фаз, $P$ – давление, а $P_c^m$ – капиллярное давление фазы $m$. Записывая для каждой компоненты смеси (фазы) закон сохранения массы, получаем систему уравнений:

$$-\operatorname{div}\left(\rho^m\mathbf{K}\frac{\kappa^m}{\eta^m}\operatorname{grad}\left(P + P_c^m\right)\right) = \frac{\partial}{\partial t}\left(\Phi\rho^m S^m\right) + \tilde{f},\ m = \overline{1,M}, \tag{2}$$

где $\tilde{f}$ – отбор или нагнетание фазы в области.

Будем считать, что моделируемый процесс изотермический и фазы – это несжимаемые жидкости, т.е. $\rho^m$ – постоянные величины. Тогда левые и правые части уравнений (2) можно разделить на $\rho^m$, а полученные уравнения сложить, и тогда, учитывая, что $\frac{\partial}{\partial t}\left(\sum_{m=1}^{M} S^m\right) = 0$ (поскольку $\sum_{m=1}^{M} S^m = 1$), получаем для давления $P$ эллиптическую краевую задачу:

$$-\mathrm{div}\left(\sum_{m=1}^{M}\mathbf{K}\frac{\kappa^m}{\eta^m}\mathrm{grad}\left(P+P_c^m\right)\right)=0, \tag{3}$$

$$P\big|_{\Gamma_1}=P_g, \tag{4}$$

$$\sum_{m=1}^{M}\mathbf{K}\frac{\kappa^m}{\eta^m}\frac{\partial P}{\partial n}\bigg|_{\Gamma_2}=\theta, \tag{5}$$

где $\Gamma_1$ удалённые границы, через которые может осуществляться переток смеси из $\Omega$ и в $\Omega$, и $\Gamma_2$ – непроницаемые границы (для них $\theta=0$) и границы скважин, через которые осуществляется отбор или нагнетание смеси области.

Предварительно определив в области $\Omega$ начальное распределение насыщенностей $S^m$, будем решать краевую задачу (3)-(5) с помощью метода конечных элементов [15]. Для этого разобьём расчётную область $\Omega$ на ячейки $\Omega_e$ [16]. В каждой ячейке $\Omega_e$ будем считать параметры среды и смеси постоянными. В результате конечноэлементной аппроксимации (использование метода конечных элементов для решения задачи (3)-(5) описано в п. 1.2) получаем в расчётной области $\Omega$ распределение поля давления $P$.

По найденному конечноэлементному решению на каждой грани $\Gamma$ ячеек $\Omega_e$ расчётной области $\Omega$ могут быть вычислены значения скорости фильтрации смеси $\vec{u}$ по соотношению (1), по которым могут быть рассчитаны объемы перетекающей смеси через грани ячеек[1]:

$$V=\left|\int_{\Gamma}\vec{u}\cdot\vec{n}\,\mathrm{d}\Gamma\right|\Delta t \tag{6}$$

где $\vec{n}$ – внешняя нормаль к грани $\Gamma$, это означает, что положительное значение интеграла будет соответствовать вытекающему объёму смеси из элемента $\Omega_e$ через границу $\Gamma$ ($V_{out}$), а отрицательное значение – втекающему объёму смеси в элемент $\Omega_e$ ($V_{in}$). Вычислив перетекающий объём смеси, можно рассчитать новые значения насыщенностей $S_{new}^m$ в каждой ячейке по соотношению:

[1] Подробнее о расчёте перетекающих потоков смеси изложено в п. 1.3

$$S_{new}^{m} = \frac{mes(\Omega_e)\Phi S^m + V_{in}^m - V_{out}^m}{mes(\Omega_e)\Phi S^m}, \tag{7}$$

где $V_{in}^m$ – втекающий, а $V_{out}^m$ – вытекающий объём $m$-ой фазы в ячейку $\Omega_e$, которые для каждой грани $\Gamma_i$ можно найти по соотношению

$$V_{\Gamma_i,\Omega_e}^m = V_{\Gamma_i,\Omega_e} \cdot \frac{\kappa^m}{\eta^m \sum_{i=1}^{M} \kappa^i / \eta^i}, \tag{8}$$

где $V_{\Gamma_i,\Omega_e}$ – перетекающий объём смеси из ячейки $\Omega_k$ в $\Omega_e$.

Величина шага по времени $\Delta t$ в соотношении (6) фиксируется для всей области такой, что за данный отрезок времени ни в одной ячейке $\Omega_e$ объём вытекающей фазы $V_{out}^m$ превышает объём этой фазы в ячейке[2]. При этом специальным образом обрабатывается ситуация, когда какой-то фазы в ячейке меньше некоторого заданного (критического, $S_{crit}^m$) значения. Кроме того, для снижения негативного влияния небалансов фаз, возникающих из-за погрешностей численного решения уравнения (3), используется метод балансировки потоков фаз через грани ячеек, который описан в п. 1.4.

Рассчитав перетекающие объёмы фаз, вместе с расчётом новых насыщенностей можно рассчитать новые значения долей $\chi^{lm}$ компонент фаз по формуле (модель полного перемешивания фазы с разными концентрациями полимера в одной ячейке):

$$\chi_{\Omega_e}^{lm} = \frac{n_{\Omega_e}^{lm} \cdot \mathrm{M}_l}{\sum_{k=1}^{L^m} n_{\Omega_e}^{km} \cdot \mathrm{M}_k}, \tag{9}$$

где $\mathrm{M}_l$ – молярная масса компоненты $l$, а $n_{\Omega_e}^{lm}$ – количество молей вещества в ячейке $\Omega_e$, которое можно найти по соотношению:

[2]Подробнее о выборе шага по времени изложено в п. 1.5

$$n_{\Omega_e}^{lm} = \left( \sum_{i \in I_{in,\Omega_e}} \rho_{\Omega_{k_i}}^{m} \chi_{\Omega_{k_i}}^{lm} V_{\Gamma_i,\Omega_e}^{m} + \rho_{\Omega_e}^{m} \chi_{\Omega_e}^{lm} \left( V_{\Omega_e}^{m} - \sum_{j \in I_{out,\Omega_e}} V_{\Gamma_j,\Omega_e}^{m} \right) \right) \Big/ \mathrm{M}_l \qquad (10)$$

где $\Omega_{k_i}$ – конечный элемент, из которого объем смеси $V_{\Gamma_i,\Omega_e}^{m}$ перетекает в ячейку $\Omega_e$ через грань $\Gamma_i$, $I_{in,\Omega_e}(I_{out,\Omega_e})$ – глобальные номера граней, соответствующие втекающим (вытекающим) объёмам фаз для ячейки $\Omega_e$.

Получив распределение насыщенностей фаз в ячейках области, необходимо рассчитать новое распределение поля давления $P$. Далее повторяя описанный алгоритм, получаем аппроксимацию течения многофазной смеси в пористой среде.

1.2. Построение конечноэлементной аппроксимации функции давления

1.2.1. Вариационная постановка и дискретный аналог

Эллиптическую краевую задачу (3)-(5) будем решать относительно давления $P$ методом конечных элементов.

Пусть функция $P = P(x, y, z)$ принадлежит гильбертову пространству $H$. Решать краевую задачу будем в слабой постановке Галёркина. Для этого потребуем выполнение условия ортогональности невязки решения уравнения (3) некоторому пространству пробных функций $\Upsilon$:

$$\int_{\Omega} -\mathrm{div}\left( \sum_{m=1}^{M} \mathbf{K} \frac{\kappa^m}{\eta^m} \mathrm{grad}\left( P + P_c^m \right) \right) \cdot \nu d\Omega = 0, \quad \forall \nu \in \Upsilon. \qquad (11)$$

Выберем в качестве $\Upsilon$ пространство $H_0^1$ – пространство функций, удовлетворяющих нулевых краевым условиям на границе $\Gamma_1$, и которые имеют интегрируемые с квадратом первые производные. Преобразуем выражение (11), применяя формулу Грина (интегрирование по частям), и с учётом краевых условий (5) получаем:

$$\int_{\Omega}\left(\sum_{m=1}^{M}\mathbf{K}\frac{\kappa^m}{\eta^m}\operatorname{grad}P\right)\cdot\operatorname{grad}\nu_0 d\Omega+\int_{\Gamma_2}\theta\nu d\Gamma_2=$$
$$=\int_{\Omega}\left(\sum_{m=1}^{M}\mathbf{K}\frac{\kappa^m}{\eta^m}\operatorname{grad}P_c^m\right)\cdot\operatorname{grad}\nu_0 d\Omega,\ \forall\nu_0\in H_0^1. \tag{12}$$

Разобьём расчётную область $\Omega$ на конечные элементы $\Omega_e$. Искать решение уравнения $P$ будем в проекции пространства $H$ на его конечномерное линейное подпространство $V^N=span\{\psi_1,\psi_2,...,\psi_N\}$, где в качестве $\psi_i$ выберем финитные базисные функции, ассоциированные с узлами конечноэлементной сетки. В этом случае функцию $P$ можно представить в виде линейной комбинации базисных функций $\psi_i$ с весами $q_j$:

$$P(x,y,z)=\sum_{i=1}^{N}q_i\psi_i(x,y,z), \tag{13}$$

Тогда, если подставить (13) в (12) и последовательно заменять пробную функцию $\nu_0$ базисными функциями $\psi_i$, получим конечноэлементную СЛАУ относительно вектора весов $\mathbf{q}$ вида:

$$\mathbf{Gq}=\mathbf{b}, \tag{14}$$

где матрица $\mathbf{G}$ и вектор правой части $\mathbf{b}$ вычисляются по соотношениям

$$G_{ij}=\int_{\Omega}\left(\mathbf{K}\sum_{m=1}^{M}\frac{\kappa^m}{\eta^m}\operatorname{grad}\psi_j\right)\cdot\operatorname{grad}\psi_i d\Omega, \tag{15}$$

$$b_i=\int_{\Omega}\left(\mathbf{K}\sum_{m=1}^{M}\frac{\kappa^m}{\eta^m}\operatorname{grad}P_c^m\right)\cdot\operatorname{grad}\psi_i d\Omega+\int_{\Gamma_2}\theta\psi_i d\Gamma_2. \tag{16}$$

Краевые условия первого рода (4) учитываются в СЛАУ (14) заменами соответствующих строк системы равенствами искомых весов $q_i$ значению краевого условия $P_g$ в $j$-ом узле.

### 1.2.2. Трилинейные базисные функции на шестигранных элементах

В данной работе используются регулярные согласованные шестигранные конечноэлементные сетки. Для того чтобы определить базисные функции на шестиграннике $\Omega_e$ свершинами $(\hat{x}_i, \hat{y}_i, \hat{z}_i)$, введём базисные функции $\hat{\varphi}_i$ на шаблонном элементе

$$\Omega^E = \left\{ (\xi, \eta, \zeta) \middle| \ -1 \le \xi \le 1, \ -1 \le \eta \le 1, \ -1 \le \zeta \le 1 \right\}, \tag{17}$$

$$\hat{\varphi}_i(\xi, \eta, \zeta) = W_{\mu(i)}(\xi) \cdot W_{\upsilon(i)}(\eta) \cdot W_{\vartheta(i)}(\zeta), \ i = 1..8, \tag{18}$$

где

$$W_1(\alpha) = \frac{1}{2}(1-\alpha), W_2(\alpha) = \frac{1}{2}(1+\alpha), \tag{19}$$

а целочисленные функции $\mu(i)$, $\upsilon(i)$ и $\vartheta(i)$ определяются соотношениями

$$\begin{aligned} \mu(i) &= ((i-1) \bmod 2) + 1, \\ \upsilon(i) &= \left( \left[ \frac{i-1}{2} \right] \bmod 2 \right) + 1, \\ \vartheta(i) &= \left[ \frac{i-1}{4} \right] + 1. \end{aligned} \tag{20}$$

Тогда базисные функции на шестиграннике можно определить с помощью соотношения:

$$\hat{\psi}_i(x, y, z) = \hat{\varphi}_i\left(\xi(x, y, z), \eta(x, y, z), \zeta(x, y, z)\right), \ i = 1..8, \tag{21}$$

где функции отображения $\xi = \xi(x, y, z)$, $\eta = \eta(x, y, z)$, $\zeta = \zeta(x, y, z)$ заданы неявно выражениями:

$$x = \sum_{i=1}^{8} \hat{x}_i \hat{\varphi}_i(\xi, \eta, \zeta), \ y = \sum_{i=1}^{8} \hat{y}_i \hat{\varphi}_i(\xi, \eta, \zeta), \ z = \sum_{i=1}^{8} \hat{z}_i \hat{\varphi}_i(\xi, \eta, \zeta). \tag{22}$$

Будем считать, что коэффициент $\mathbf{K}\sum_{m=1}^{M}\frac{\kappa^m}{\eta^m}$ постоянен на конечном элементе $\Omega_e$ и с учётом вида базисных функций $\hat{\psi}_i$ запишем соотношение для расчёта компонент матрицы СЛАУ:

$$\hat{G}_{ij}=\mathbf{K}\sum_{m=1}^{M}\frac{\kappa^m}{\eta^m}\int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1}\left(\mathbf{J}^{-1}\mathrm{grad}\hat{\varphi}_i(\xi,\eta,\zeta)\right)^{\mathrm{T}}\mathbf{J}^{-1}\mathrm{grad}\hat{\varphi}_j(\xi,\eta,\zeta)\left|J\right|d\xi d\eta d\zeta, \qquad (23)$$

где $J$ – определитель матрицы Якоби:

$$\mathbf{J}=\begin{pmatrix}\frac{\partial x}{\partial\xi} & \frac{\partial y}{\partial\xi} & \frac{\partial z}{\partial\xi}\\ \frac{\partial x}{\partial\eta} & \frac{\partial y}{\partial\eta} & \frac{\partial z}{\partial\eta}\\ \frac{\partial x}{\partial\zeta} & \frac{\partial y}{\partial\zeta} & \frac{\partial z}{\partial\zeta}\end{pmatrix}. \qquad (24)$$

Вычислять компоненты матрицы СЛАУ (23) будем численно с помощью метода Гаусса-3.

1.3. Расчёт потока смеси через границы конечных элементов

Потоком смеси через границу $\Gamma_i$ конечного элемента $\Omega_e$ будем называть интеграл

$$Q_i=\int_{\Gamma_i}\left(\mathbf{K}\sum_{m=1}^{M}\frac{\kappa^m}{\eta^m}\mathrm{grad}\left(P+P_c^m\right)\right)\cdot\vec{n}_{\Gamma_i}\,d\Gamma, \qquad (25)$$

где $\vec{n}_{\Gamma_i}$ – нормаль к грани $\Gamma_i$, зафиксированная некоторым образом, единим для содержащих её элементов. Направление потока через грань $\Gamma_i$ по отношению к содержащему её конечному элементу определим величиной $Sg_{\Gamma_i}^{\Omega_e}$, которая равна 1, если поток $Q_i$ втекает в элемент $\Omega_e$, и равна -1, если поток $Q_i$ из элемента $\Omega_e$ вытекает.

Отметим, что знак интеграла (25) не определяет направление течения жидкости по отношению к элементам области, потому что нормали $\vec{n}_{\Gamma_i}$ не

выбраны как внешние (или внутренние) по отношению к ним. Для того, чтобы определить, втекает поток смеси в элемент или вытекает из него, вводится специальная величина $Sg_{\Gamma_i}^{\Omega_e}$, которая фактически определяет соответствие внешнего направления нормали грани $\Gamma_i$ ячейки $\Omega_e$ зафиксированному направлению грани $\vec{n}_{\Gamma_i}$. Не смотря на то, что такое определение нормалей не достаточно удобно для изложения теоретического материала, это необходимо для существенной экономии вычислительных затрат при реализации метода балансировки потоков, который изложен в пункте 1.4.

Рассчитать интеграл (25) по конечноэлементному решению, где $\Gamma_i$ – грань шестигранника, можно по формуле (с учётом (13) и (24))

$$Q_i = \bar{\lambda} \int_{-1}^{1}\int_{-1}^{1} \left( \sum_{i=1}^{8} \mathbf{J}^{-1} \left( \frac{\partial \hat{\varphi}_i(\xi,\eta,\zeta)}{\partial \xi}, \frac{\partial \hat{\varphi}_i(\xi,\eta,\zeta)}{\partial \eta}, \frac{\partial \hat{\varphi}_i(\xi,\eta,\zeta)}{\partial \zeta} \right)^{\mathrm{T}} \Bigg|_{\hat{\Gamma}_i} \right) \cdot \vec{n}_{\Gamma_i} \left| J^{2\mathrm{D}} \right| d\Gamma, \qquad (26)$$

где $\bar{\lambda} = \mathbf{K} \sum_{m=1}^{M} \frac{\kappa^m}{\eta^m}$ – коэффициенты, рассчитанные для того элемента сетки, из которого поток $Q_{\Omega_e,\Gamma_i}$ вытекает (то есть $Sg_{\Gamma_i}^{\Omega_e} = -1$), $\hat{\Gamma}_i$ – грань шаблонного элемента $\Omega^E$ (17), соответствующая грани шестигранника $\Gamma_i$, $J^{2\mathrm{D}}$ – якобиан матрицы перехода от $\Gamma_i$ к $\hat{\Gamma}_i$. Интеграл (26) будем рассчитывать численно с помощью метода Гаусса-3.

Поскольку поток смеси через границу $\Gamma_i$ будет рассчитываться по конечноэлементному решению, то за счёт разрыва производной конечноэлементного решения интеграл (25) может иметь разные значения на конечных элементах, содержащих грань $\Gamma_i$. Поэтому будем использовать осреднённый поток

$$\left|\bar{Q}_i\right| = (1-\omega)\left|Q_{\Omega_e,\Gamma_i}\right| + \omega\left|Q_{\Omega_k,\Gamma_i}\right|, \qquad (27)$$

где $\Omega_e$ и $\Omega_k$ – элементы, содержащие грань $\Gamma_i$.

1.4. Балансировка потоков

Поскольку основная идея рассматриваемой схемы моделирования заключается в том, что фильтруемые жидкости пошагово «перетекают» из ячеек в ячейки, крайне необходимо, чтобы всех ячейках сетки выполнялся баланс потоков, то есть сумма втекающих в ячейку потоков равнялся сумме вытекающих потоков. Причём, ввиду численной погрешности и известного свойства накопления численных ошибок, желательно, чтобы баланс выполнялся с некоторой заданной точностью (например, $\varepsilon^{balance} = 10^{-10}$).

В связи с этим, будем искать корректирующие добавки $\delta Q_i$ к полученным по конечноэлементному решению осреднённым потокам $\bar{Q}_{\Omega_e,\Gamma_i}$, удовлетворяющие минимуму функционала:

$$\sum_{e=1}^{N^e} \beta_e \left( \sum_{i \in I_{\Omega_e}} \left( Sg_{\Gamma_i}^{\Omega_e} \cdot \left( \left| \bar{Q}_{\Omega_e,\Gamma_i} \right| + \delta Q_i \right) \right) \right)^2 + \sum_{i=1}^{N^f} \alpha_i \left( \delta Q_i \right)^2 \to \min_{\delta Q_i}, \tag{28}$$

где $e$ – номер конечного элемента, $N^e$ – количество конечных элементов, $N^f$ – количество граней, $I_{\Omega_e}$ – множество номеров граней элемента $\Omega_e$, $\beta_e$ и $\alpha_i$ – регуляризирующие коэффициенты. При этом коэффициенты $\beta_e$ выбираются из условия

$$\sum_{e=1}^{N^e} \beta_e \left| \sum_{i \in I_{\Omega_e}} \left( Sg_{\Gamma_i}^{\Omega_e} \cdot \left( \left| \bar{Q}_{\Omega_e,\Gamma_i} \right| + \delta Q_i \right) \right) \right| < \varepsilon^{balance}, \tag{29}$$

где $\varepsilon^{balance}$ некоторое заданное значение, определяющее предельно допустимый суммарный небаланс перетекающих объемов смеси. Найденные добавки $\delta Q_i$, должны быть минимально возможными, для обеспечения близости скорректированных сбалансированных объемов смеси к исходным несбалансированным.

В результате минимизации функционала (28) получаем СЛАУ вида

$$\left( \mathbf{B} + \boldsymbol{\alpha} \right) \mathbf{q} = \mathbf{d}, \tag{30}$$

где $\mathbf{q}$ – вектор, составленный из искомых значений $\delta Q_i$, $\boldsymbol{\alpha}$ – диагональная матрица с элементами $\alpha_i$ на главной диагонали, а компоненты матрицы $\mathbf{B}$ и вектора правой части $\mathbf{d}$ вычисляются с помощью соотношений

$$B_{ij} = \begin{cases} 2, \ \textit{если} \ i = j, \\ \beta_e \cdot Sg_{\Gamma_i}^{\Omega_e} \cdot Sg_{\Gamma_j}^{\Omega_e}, \ \textit{если} \ i, j \in I_{\Omega_e}, i \neq j, \\ 0, \ \textit{иначе}, \end{cases} \tag{31}$$

$$d_i = -\sum_{e=1}^{N^e} \left( Sg_{\Gamma_i}^{\Omega_e} \cdot \sum_{j \in I_{\Omega_e}} \bar{Q}_{\Omega_e, \Gamma_j} \right), \tag{32}$$

где $I_{\Omega_e}$ – множество глобальных номеров граней, которые содержит $\Omega_e$.

Обратим внимание, что компоненты матрицы $\mathbf{B}$ не зависят от модуля и направления, перетекающих через границы ячеек потоков, потому что величины $Sg_{\Gamma_i}^{\Omega_e}$ определяют лишь соответствие направления внешней нормали к грани $\Gamma_i$ элемента $\Omega_e$ с зафиксированным направлением грани $\vec{n}_{\Gamma_i}$ (смотри п. 1.4). За счёт этого, можно существенно сократить вычислительные затраты при реализации модуля балансировки потоков, используя для решения СЛАУ (30) прямые решатели, которые используют факторизацию матрицы СЛАУ (разложение матрицы СЛАУ на произведение двух треугольных матриц). При этом, в случае, если параметры регуляризации $\beta_e$ и $\alpha_i$ не изменяются, то разложение матрицы СЛАУ можно выполнить один раз.

С учётом полученных в результате решения СЛАУ (30) корректирующих добавок, получаем сбалансированные значения перетекающих потоков смеси через границы элементов, удовлетворяющие условию выполнения баланса с заданной точностью, в виде

$$\tilde{Q}_i = Sg_{\Gamma_i}^{\Omega_e} \cdot \left( \left| \bar{Q}_i \right| + \delta Q_i \right). \tag{33}$$

1.5. Определение объёмов перетекающих фаз и шага по времени

Обозначим через $\alpha^m$ величины долей вытекающих фаз

$$\alpha^m = \frac{\kappa^m}{\eta^m \sum_{k=1}^{M} \kappa^k / \eta^k} \tag{34}$$

Объём $m$-ой фазы, вытекшей из ячейки $\Omega_e$ через грань $\Gamma_i$ за время $\Delta t$, вычисляется по соотношению

$$V^m_{\Gamma_i,\Omega_e} = \alpha^m \cdot \left|\tilde{Q}_{\Gamma_i,\Omega_e}\right| \cdot \Delta t\,. \tag{35}$$

Будем фиксировать шаг по времени одинаковый для всех ячеек расчётной области. Величины суммарного втекающего в ячейку $\Omega_e$ и вытекающего из $\Omega_e$ объёма фаз вычисляются по соотношению

$$V^m_{in,\Omega_e} = \sum_{i \in I_{in,\Omega_e}} V^m_{\Gamma_i,\Omega_e}\,, \tag{36}$$

$$V^m_{out,\Omega_e} = \sum_{j \in I_{out,\Omega_e}} V^m_{\Gamma_j,\Omega_e}\,. \tag{37}$$

Естественным условием на выбор шага по времени $\Delta t$ является то, что суммарный вытекающий объём фазы из ячейки не должен превышать имеющийся объём фазы в ней. Записать это условие можно следующим соотношением

$$V^m_{out,\Omega_e} \le \mathrm{mes}\left(\Omega_e\right) \Phi S^m, \tag{38}$$

где $\Phi$ – структурная пористость ячейки $\Omega_e$, а $S^m$ –насыщенность $m$-ой фазы в ячейке $\Omega_e$. Вычислительно эффективнее находить шаг по времени будет по следующей формуле, которая не нарушает предыдущих рассуждений:

$$\Delta t = \min\left\{ \Delta t_{me} \;\middle|\; \Delta t_{me} = \frac{\mathrm{mes}\left(\Omega_e\right) \Phi S^m}{\alpha^m \cdot \sum_{i \in I_{out,\Omega_e}} \tilde{Q}_{\Gamma_i,\Omega_e}},\ m = 1..M,\ e = 1..N^e \right\} \tag{39}$$

Из условия (38) видно, что выбор шага $\Delta t$ зависит от малости величины насыщенности фазы $S^m$. То есть, если $S^m$ достаточно малая величина, то руководствуясь этим условием необходимо выбрать соответствующий маленький шаг по времени, что может повлечь за собой резкое замедление моделирования процесса фильтрации. Поэтому введём специальные величины, для контроля величины шага $\Delta t$:

- $S^m_{crit,\max}$ – максимальная критическая насыщенность $m$-ой фазы в ячейке $\Omega_e$, для которой выбирается шаг по времени $\Delta t$. То есть, если в $\Omega_e$ $S^m \le S^m_{crit,\max}$, то шаг $\Delta t_{me}$ исключается из множества (39). Пропуски выбора $\Delta t$ для таких насыщенностей может нарушать условие (38), поэтому будем вносить пары номеров элемента $e$ и номера фазы $m$ пропущенных насыщенностей во множество пар $I'_{abandon}$.

- $S^m_{crit,\min}$ – минимальная критическая насыщенность $m$-ой фазы, которая определяет величину насыщенности $S^m$ в ячейке, при которой насыщенность фазы в ячейке на столько мала, что данная фаза будет вытесняться из ячейки в потоке вместе с остальными вытекающими фазами (другими словами, данная фаза полностью покинет ячейку пропорционально вытекающему потоку смеси). Если $S^m \le S^m_{crit,\min}$, то шаг $\Delta t_{me}$ так же исключается из множества (39), и пара номеров элемента $e$ и номера фазы $m$ пропущенных насыщенностей будет внесена во множество пар $I''_{abandon}$.

Очевидно, что по определению $S^m_{crit,\min} < S^m_{crit,\max}$, и вообще говоря, множество $I''_{abandon} \subset I'_{abandon}$, но назначения данных величин не позволяет использовать только одну из них.

Пропуски величин шага $\Delta t_{me}$, для которых выполняется $S^m_{crit,\min} < S^m_{crit,\max}$ или $S^m \le S^m_{crit,\min}$, могут привести к нарушению условия (38). Это значит, что по выбранному шагу $\Delta t$ вытекающий объём фазы из ячейки может превышать имеющийся объём фазы в этой ячейке. Это может привести к тому, что новая насыщенность этой фазы по формуле (7) может стать отрицательной, если

$V_{in}^{m}=0$. Поэтому необходимо выбрать из множества пар $I'_{abandon}$ те, для которых $V_{out,\Omega_e}^{m}>\mathrm{mes}(\Omega_e)\Phi S^{m}$, обозначим это множество $\tilde{I}'_{abandon}$. И далее, для фаз, чьи номера оказались в объединении множеств $I_{abandon}=\tilde{I}'_{abandon}\cup I''_{abandon}$, необходимо произвести процедуру «выталкивания» фазы из ячейки под действием вытекающего потока смеси.

Процедура «выталкивания» заключается в следующем. Пусть $I^{e}_{missed}$ – множество $m'$ номеров фаз в ячейке $\Omega_e$ таких, что $(e,m')\in I_{abandon},\ \forall m'\in I^{e}_{missed}$. То есть необходимо из ячейки $\Omega_e$ фазы с номерами $m'\in I^{e}_{missed}$ полностью «вытолкнуть». Для ячейки $\Omega_e$ необходимо вычесть из известных вытекающих потоков смеси через границы $\Gamma_j$ ($j\in I_{out}$) объёмы выталкиваемых фаз (пропорционально вытекающим потокам смеси), то есть

$$\tilde{V}_{\Gamma_j,\Omega_e}=V_{\Gamma_j,\Omega_e}-\frac{\tilde{Q}_{\Gamma_j,\Omega_e}}{\sum\limits_{k\in I_{out}}\tilde{Q}_{\Gamma_k,\Omega_e}}\cdot\sum_{m'\in I^{e}_{missed}}V_{\Omega_e}^{m'},\tag{40}$$

где $V_{\Omega_e}^{m'}$ – объём фазы $m'$ в ячейке $\Omega_e$. Для фаз, которые выталкиваться не будут, вычислим новые доли по соотношению

$$\tilde{\alpha}^{m}=\frac{\alpha^{m}}{\sum\limits_{k\notin I^{e}_{missed}}\alpha^{k}}\tag{41}$$

Тогда новые объёмы вытекающих фаз из ячейки $\Omega_e$ через границу $\Gamma_j$ с учётом «выталкивания» будут определяться соотношениями:

$$V_{\Gamma_j,\Omega_e}^{m'}=\frac{\tilde{Q}_{\Gamma_j,\Omega_e}}{\sum\limits_{k\in I_{out}}\tilde{Q}_{\Gamma_k,\Omega_e}}\cdot V_{\Omega_e}^{m'},\ \ m'\in I^{e}_{missed},\tag{42}$$

$$V_{\Gamma_j,\Omega_e}^{m}=\tilde{\alpha}^{m}\cdot\tilde{V}_{\Gamma_j,\Omega_e},\ \ m\notin I^{e}_{missed}.\tag{43}$$

# 2. ОПИСАНИЕ РАЗРАБОТАННОЙ ПРОГРАММЫ

В этой главе описаны основные структуры данных и алгоритмы схемы аппроксимации течения многофазной смеси в пористой среде, описанной в первой главе, и реализующие их функции в разработанной программе.

## 2.1. Основные сведения

Разработанная в ходе данной работы программа предназначена для моделирования процесса многофазной фильтрации с учётом многокомпонентности фаз в расчётных областях, представляющих собой геологические структуры со скважинами.

Программа написана на языке C++, используя технологии объектно-ориентированного программирования. Среда разработки: Microsoft Visual Studio 2015 Enterprise. Программа поддерживает 64-х разрядные операционные системы семейства Windows.

Для решения СЛАУ в программе используется прямой решатель Pardiso из библиотеки Intel MKL.

*Входные данные*:

1) Регулярная конечноэлементная сетка с шестигранными ячейками.

   а. Список узлов сетки, представленных в виде записей трёх вещественных чисел – координат узлов.

   б. Список конечных элементов, представленных в виде записей восьми целых чисел – глобальных номеров узлов сетки.

   в. Список конечных элементов, представленных в виде записей шести целых чисел – глобальных номеров граней сетки.

   г. Список номеров материалов породы, к которым относятся элементы сетки.

2) Информация об удалённой границе и о границах скважин в области, представленная в виде списков соответствующих номеров граней сетки.

3) Информация о количестве и свойствах фильтруемых фаз.

4) Информация о компонентном составе фаз.

5) Информация о свойствах материалов породы.

6) Информация о режимах работы скважин (мощность добычи/нагнетания, время работы, составы закачиваемой жидкости).

7) Информация о зависимостях свойств фаз от внешних факторов.

8) Начальное распределение насыщенностей в области.

9) Настройки расчёта (максимальное количество итераций, время окончания расчёта, максимальный шаг по времени).

10) Информация о контрольных линиях и контрольных точках.

*Выходные данные*:

1) Поля распределения насыщенностей по времени.

2) Графики давления и насыщенностей фильтруемых фаз вдоль контрольных линий на временных слоях, и графики давления и насыщенностей как функций от времени в контрольных точках. Графики отборов смеси и фаз для каждой скважины в отдельности, графики поступления в область смеси и фаз через удалённые границы как функции от времени.

3) Информация о времени работы программы и времени работы отдельных модулей.

4) Настроечные параметры работы программы.

Разработанная программа содержит модули:

1) Модуль хранения шестигранной конечноэлементной сетки, с дополнительными подмодулями, выполняющие

   а. фиксацию направлений граней элементов;

   б. расчёт объёмов элементов;

2) Модуль расчёта распределения давления в области.

3) Модуль расчёта нового состояния ячеек при заданном распределении давления, который включает подмодули:

   а. расчёта осреднённых перетекающих потоков;

   б. фиксирования потоков на границах скважин из краевого условия;

   в. балансировки потоков;

   г. расчёта шага по времени;

   д. расчёта новых насыщенностей и компонентного состава фаз.

4) Модули вычисления базисных функций, вычисления обратных матриц к функциональным матрицам вида (24), якобианов, внешних нормалей к граням конечных элементов и прочие функции-утилиты;

5) Модуль численного интегрирования.

6) Модуль хранения и обработки информации о состоянии модели:

   а. поля давления;

   б. полей насыщенностей;

   в. компонентного состава фаз в ячейках;

   г. таблиц зависимостей свойств фаз;

   д. хранение и обработка свойств фаз в ячейках;

Далее подробнее опишем основные модули разработанной программы.

## 2.2. Структуры данных

В данном пункте приведены основные структуры данных, которые используются для хранения модели и её состояния, а также, некоторые структуры, используемые для хранения промежуточных данных, требуемых для выполнения вычислений.

Структуры данных, используемые для описания модели, определены в модуле «FEM_Structs». Для описания конечноэлементной сетки используются структуры «Point3D» и «Element3D». В структуре «Point3D» определены координаты узлов сетки. Структура «Element3D» содержит описание конечного элемента в виде восьми глобальных номеров узлов (поле «node»), в виде шести глобальных номеров граней (поле «faces»), и массив из шести чисел (поле «faces_direct»), которые могут принимать значения 1 или -1, которые определяют величину $Sg_{\Gamma_i}^{\Omega_e}$ (см. пункт 1.4). Используя эти структуры данных, сетка определяется массивами:

- vector<Element3D> Elements; – массив (размер: $N^e$) конечных элементов.
- vector<Point3D> Points; – массив (размер: $N$) узлов сетки.

Для хранения информации используется специальный модуль «FiltrProp», ссылку на который содержат вычислительные модули. В нём содержатся некоторые функции работы с файлами для чтения и записи данных, а также процедуры, выполняющие некоторые операции с данными, например вычисление новых значения параметров фаз, или проверка равенства единице суммы насыщенностей фаз в ячейке, и так далее.

Структуры данных используемые для описания состояния модели:

- vector<vector<double>> S; – массив (размер: $N^e \times M$ ), который содержит списки значений насыщенностей $S_{\Omega_e}^m$.

- vector<double> P; – массив (размер: $N$) весов разложения функции давления $q_i$, далее будем обозначать его $\{q_i\}_{i=1..N}$.

- vector<vector<Phase>> PhasesElem; – массив (размер: $N^e \times M \times 3$), в котором для каждого элемента $\Omega_e$ в структуре «Phase» хранятся параметры фаз: плотность $\rho^m$, вязкость $\eta^m$ и множитель к структурной проницаемости $\kappa^m$.

- vector<PhaseComponents> phasecomp; – массив (размер: $N^e \times N^\chi$), в котором для каждого элемента $\Omega_e$ в структуре «PhaseComponents» хранится компонентный состав фаз в элементе, далее будем обозначать его $\left\{\chi_e^{lm}\right\}_{e=1..N^e, m=1..M, l=1..L^m}$. Структура «PhaseComponents» хранит значения таблицы компонентного состава, при этом память выделяется только для ячеек таблицы, которые могут принимать не нулевые значения.

В программном модуле «Filtration» выполняются процедуры пересчёта нового состояния ячеек, и для этого в нём определены массивы:

- vector<double> flMix; – массив (размер: $N^f$), в котором для каждой грани $\Gamma_i$ хранится значение перетекающего (осреднённого и сбалансированного) потока смеси $\bar{Q}_i$ ( $\tilde{Q}_i$), далее будем обозначать его $\left\{\bar{Q}_i\right\}_{i=1..N^f}$ ( $\left\{\tilde{Q}_i\right\}_{i=1..N^f}$ ).

- vector<vector<double>> fluxOutPhase; – массив (размер: $N^f \times M$), в котором для каждой грани $\Gamma_i$ записаны значения потоков перетекающих фаз $\bar{Q}_i^m$ ( $\tilde{Q}_i^m$), далее будем обозначать его $\left\{\bar{Q}_i^m\right\}_{i=1..N^f,m=1..M}$ ( $\left\{\tilde{Q}_i^m\right\}_{i=1..N^f,m=1..M}$ ).

- vector<vector<double>>volumeOutPhase; – массив (размер: $N^f \times M$), в котором для каждой грани $\Gamma_i$ записаны значения объёмов перетекающих фаз $V_{\Gamma_i}^m$, далее будем обозначать его $\left\{V_i^m\right\}_{i=1..N^f,m=1..M}$.

- vector<double> alpha; – массив (размер: $N^f$), в котором для каждой грани $\Gamma_i$ записаны значения регуляризирующего параметра $\alpha_i$, далее будем обозначать его $\left\{\alpha_i\right\}_{i=1..N^f}$.

- vector<double> beta; – массив (размер: $N^e$), в котором для каждого элемента $\Omega_e$ записаны значения регуляризирующего параметра $\beta_e$, далее будем обозначать его $\left\{\beta_e\right\}_{e=1..N^e}$.

2.3. Модуль расчёта давления

Согласно пункту 1.2 по заданному распределению насыщенностей фаз можно рассчитать распределения функции давления $P$.

Входные данные для модуля расчёта давления:

1) Конечноэлементная сетка, массивы, в которых записаны свойства материалов породы $\Phi_{\Omega_e}$ и $\mathbf{K}_{\Omega_e}$, $e=1..N_e$.
2) Массивы, в которых хранится начальное распределение насыщенностей $S_{\Omega_e}^m$, $e=1..N_e$, $m=1..M$.
3) Массивы номеров граней, на которых заданы первые (и вторые) краевые условия и значения фиксируемого на них давления (потока давления).

Выходные данные:

1) Массив $\{q_i\}_{i=1..N}$ весов разложения функции давления $P$ (см. соотношение (13)), где $N$ – количество узлов в сетке.

Алгоритм работы модуля расчёта давления:

Шаг 1. В цикле по конечным элементам $\Omega_e, e=1..N^e$, выполняются следующие действия:

Шаг 1.1. По соотношению (23) расчёт локальную матрицу $\hat{G}$.

Шаг 1.2. Занесение локальной матрицы $\hat{G}$ в глобальную матрицу СЛАУ $\mathbf{G}$ по соответствию локальной и глобальной нумерации неизвестных на текущем конечном элементе.

Шаг 2. В цикле по граням, через которые производится откачка/закачка жидкости $\Gamma_i$, $i=1..N^{f,bc}$, выполнить следующие действия:

Шаг 2.1. По соотношению (16) рассчитать локальный вектор $\hat{b}$.

Шаг 2.2. Занесение локального вектора $\hat{b}$ в глобальный вектор матрицы СЛАУ $\mathbf{b}$ по соответствию локальной и глобальной нумерации неизвестных на текущем конечном элементе.

Шаг 3. В цикле по узлам, лежащих на контактируемых с удалённой границей гранях, выполняется учёт первых краевых условий (4) с симметризацией матрицы СЛАУ (Гауссово вычитание).

Шаг 4. Решение матрицы СЛАУ (14).

### 2.4. Модуль численного интегрирования

Реализация численного интегрирования выполнена с учётом минимизации вычислительных затрат. В данной задаче необходимо вычислять два типа интегралов: объёмные интегралы вида (23) и поверхностные интегралы вида (26). Поэтому разработано два подмодуля для расчёта двух- и трёхмерных интегралов.

Численное интегрирование производится с помощью трёхточечного метода Гаусса. Этот метод интегрирования сводит решение интеграла к

суммированию значений подынтегральной функции в точках интегрирования с весами (приведём соотношение для объёмного интеграла):

$$\int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} f(\xi,\eta,\zeta)d\xi d\eta d\zeta = \sum_{i=1}^{3}\sum_{j=1}^{3}\sum_{k=1}^{3}\alpha_i\alpha_j\alpha_k f\left(w_i,w_j,w_k\right),$$
$$\alpha_1=\frac{8}{9},\ \ \alpha_2=\frac{5}{9},\ \ \alpha_3=\alpha_2,\ \ \ w_1=0,\ \ w_2=\sqrt{0.6},\ \ w_3=-w_2 \qquad (44)$$

В данной работе все интегрируемые функции содержат произведения базисных функций $\hat{\varphi}$ (в том числе и при вычислении функциональной матрицы $\mathbf{J}$), которые определены на шаблонном элементе (17), и поэтому с целью минимизации вычислительных операций, заранее, один раз за время работы программы, вычисляются значения $\hat{\varphi}$ в точках интегрирования и сохраняются в соответствующие массивы. Также, для расчёта интегралов вида (26) аналогично сохраняются значения $\hat{\varphi}$ в точках интегрирования на гранях шаблонного элемента $\hat{\Gamma}$.

Входные данные для работы модуля численного интегрирования:

2) Координаты узлов $\left(\hat{x}_i,\hat{y}_i,\hat{z}_i\right)$ $i=1..8$ элемента $\Omega_e$.
3) Данные, определяющие подынтегральную функцию (например, массив весов $\hat{q}_i$ для (26)).

Выходные данные:

1) Значение интеграла.

Алгоритм работы модуля численного интегрирования:

Шаг 1. В цикле по точкам интегрирования выполнить следующие действия

Шаг 1.1. Расчёт матрицы $\mathbf{J}^{-1}$ иякобиана $|J|$ в точке интегрирования.

Шаг 1.2. В цикле по базисным функциям (по индексам $i$ и $j$) расчёт подынтегральной функции.

Шаг 2. Расчёт значения интеграла по соотношению (44).

2.5. Модуль расчёта нового состояния ячеек при заданном распределении давления

После того как найдено распределение давления, согласно математической модели, описанной в п. 1.1, можно рассчитать новые состояния ячеек, вычислив перетекающие объёмы фаз между ячейками.

Общий алгоритм расчёта нового состояния ячеек:

Шаг 1. Расчёт осреднённых перетекающих потоков смеси $\overline{Q}_i$, $i = 1..N^f$ через грани конечных элементов.

Шаг 2. Фиксирование значений потоков $\overline{Q}_k$, на границах скважины из краевого условия (5).

Шаг 3. Балансировка перетекающих потоков смеси.

Шаг 4. Расчёт перетекающих потоков фаз $\overline{Q}_i^m$, $i = 1..N^f$.

Шаг 5. Расчёт шага по времени $\Delta t$.

Шаг 6. Расчёт перетекающих объёмов фаз $V_i^m$, $i = 1..N^f$

Шаг 7. Расчёт новых насыщенностей и компонентного состава фаз в ячейках $\Omega_e$, $e = 1..N^e$.

Каждый шаг общего алгоритма выполняется отдельным подмодулем, далее опишем подробнее алгоритмы их работы.

2.5.1. Подмодуль расчёта осреднённых перетекающих потоков смеси

Входные данные:

2) Конечноэлементная сетка.

3) Массив $\{q_i\}_{i=1..N}$ весов разложения функции давления $P$.

Выходные данные:

1) Массив $\{\overline{Q}_i\}_{i=1..N^f}$ осреднённых потоков смеси.

Алгоритм работы этого подмодуля заключается в следующем. В цикле по граням $\Gamma_i$ каждого конечного элемента $\Omega_e$, $e = 1..N^e$ вычисляются потоки смеси $Q_{\Gamma_i,\Omega_e}$ по соотношению (26). Если для данной грани $\Gamma_i$ поток уже был

вычислен на смежном с ней элементе $\Omega_k$, то значение потока $\bar{Q}_i$ вычисляется по формуле (27), иначе $\bar{Q}_i = Q_{\Gamma_i,\Omega_e}$.

Отметим, что массив $\left\{\bar{Q}_i\right\}_{i=1..N^f}$ содержит значения потоков – знаковые вещественные числа, знак которых определяет направление течения потока по отношению к ориентациям соответствующих граней $\vec{n}_{\Gamma_i}$, и этот знак не показывает направления течения жидкости по отношению к внутренней (или внешней) нормали элемента.

2.5.2. Подмодуль балансировки перетекающих потоковсмеси

Входные данные:

1) Конечноэлементная сетка.
2) Массивы $\left\{\alpha_i\right\}_{i=1..N^f}$, $\left\{\beta_i\right\}_{i=1..N^e}$ – параметры регуляризации.
3) Массив $\left\{\bar{Q}_i\right\}_{i=1..N^f}$ осреднённых потоков смеси.
4) $\Gamma^W$ – список номеров граней, лежащих на границе скважин, через которые откачивается или закачивается жидкость.
5) $\varepsilon^{balance}$ – максимальный суммарный относительный небаланс потоков на элементах.

Выходные данные:

1) Массив $\left\{\tilde{Q}_i\right\}_{i=1..N^f}$ сбалансированных потоков смеси.

Во время первого запуска алгоритма необходимо инициализировать начальные значения параметров регуляризации $\left\{\alpha_i\right\}_{i=1..N^f}$, $\left\{\beta_e\right\}_{e=1..N^e}$.

Алгоритм балансировки потоков:

Шаг 1. В цикле по конечным элементам $\Omega_e$, $e=1..N^e$, выполняется следующее:

Шаг 1.1. По соотношениям (31) и (32), и используя параметры регуляризации $\left\{\beta_e\right\}_{e=1..N^e}$, вычисляются значения локальных матриц $\hat{\mathbf{B}}$ и добавок в правую часть $\hat{\mathbf{d}}$.

Шаг 1.2. По соответствию локальной и глобальной нумерации граней на текущем элементе заносятся компоненты локальной матрицы $\hat{\mathbf{B}}$ в глобальную матрицу $\mathbf{B}$, и компоненты локального вектора $\hat{\mathbf{d}}$ заносятся в глобальный вектор $\mathbf{d}$.

Шаг 2. В цикле по списку граней $\Gamma_i$, $i=1..N^f$, добавить компоненты диагональной матрицы $\boldsymbol{\alpha}$, со значениями $\{\alpha_i\}_{i=1..N^f}$ на главной диагонали, к матрице СЛАУ $(\mathbf{B}+\boldsymbol{\alpha})$.

Шаг 3. Зафиксировать значения неизвестных $\delta Q_k$, $k \in \Gamma^W$ в СЛАУ равными нулю.

Шаг 4. Из решения СЛАУ (30) находятся корректирующие добавки $\delta Q_i$.

Шаг 5. Проверка выполнения условия (29). Если условие (29) не выполняется, то выполняется корректировка параметров регуляризации $\{\alpha_i\}_{i=1..N^f}$ и $\{\beta_e\}_{e=1..N^e}$, и выполняется переход на Шаг 1.

Шаг 6. Вычисляются сбалансированные потоки $\tilde{Q}_i$ по соотношению (33).

Отметим, что для решения СЛАУ (30) выгодно использовать прямые решатели, использующие факторизацию матрицы СЛАУ – разложение на произведение треугольных матриц. В случае, когда коэффициенты $\{\alpha_i\}_{i=1..N^f}$ и $\{\beta_e\}_{e=1..N^e}$ перед очередным решением системы не изменены, то в этом случае можно не производить повторно факторизацию матрицы СЛАУ (самую вычислительно затратную процедуру), а выполнить только прямой и обратный ход с новой правой частью (решение систем с треугольными матрицами).

2.5.3. Подмодуль расчёта перетекающих потоков фаз

Данный подмодуль реализован с целью уменьшения вычислительных затрат. Перетекающие потоки фаз используются для расчёта шага по времени, и далее, с учётом выбранного шага по времени вычисляются перетекающие объёмы фаз. Поэтому в разработанной программе расчёт перетекающих потоков выполняется в отдельном подмодуле один раз за одну итерацию пересчёта нового состояния ячеек.

Входные данные:

1) Конечноэлементная сетка.
2) Массив $\left\{\tilde{Q}_i\right\}_{i=1..N^f}$ сбалансированных потоков смеси.
3) $\Gamma^W$ – список номеров граней, лежащих на границе скважин, через которые откачивается или закачивается жидкость.
4) $\left\{S_W^m\right\}_{m=1..M}$ – список насыщенностей фаз, находящихся за соответствующими их границами скважин (фазовый состав нагнетаемых смесей).
5) $\Gamma^B$ – список номеров граней, лежащих на удалённой границе $\Omega$.
6) $\left\{S_B^m\right\}_{m=1..M}$ – список насыщенностей фаз, находящихся за соответствующими их удалёнными границами (фазовый состав смеси, находящейся вокруг за границами $\Omega$).

Выходные данные:

1) Массив $\left\{Q_i^m\right\}_{i=1..N^f,m=1..M}$ перетекающих потоков фаз.

Алгоритм подмодуль расчёта перетекающих потоков фаз заключается в следующем. В цикле по граням $\Gamma_i$, где $i \in I_{out}$, каждого конечного элемента $\Omega_e$, $e=1..N^e$, вычисляются вытекающие из текущего элемента потоки фаз по соотношению $Q_i^m = \alpha^m \cdot \left|\tilde{Q}_i\right|$, где $\alpha^m$ определяется соотношением (34).

В цикле по граням из списков $\Gamma^W$ и $\Gamma^B$ выполняется следующее. Если поток через текущую грань $\Gamma_i$ направлен в $\Omega$, то по соответствующим величинам насыщенностей за границей $\left\{S_W^m\right\}_{m=1..M}$ (или $\left\{S_B^m\right\}_{m=1..M}$) рассчитываются значения долей $\alpha^m$ по соотношению (34) и вычисляются значения потоков $Q_i^m = \alpha^m \cdot \left|\tilde{Q}_i\right|$.

2.5.4. Подмодуль расчёта шага по времени

Входные данные:

1) Массив $\left\{Q_i^m\right\}_{i=1..N^f,m=1..M}$ перетекающих потоков фаз.

2) Массив $\left\{S^m_{\Omega_e}\right\}_{e=1..N^e,m=1..M}$ насыщенностей фаз в элементах.

3) Массив $\left\{S^m_{crit,\max}\right\}_{m=1..M}$ критических максимальных насыщенностей фаз.

4) $\Delta t^0$ – начальное значение шага по времени.

Выходные данные:

1) $\Delta t$ – шаг по времени.

2) Множество пар номеров элементов и фаз $I_{abandon}$, для которых будет производиться процедура «выталкивания» фазы из ячейки.

Алгоритм расчёта шага по времени:

Шаг 1. Шагу по времени $\Delta t$ присваивается начальное значение $\Delta t^0$.

Шаг 2. В цикле по номерам фаз $m=1..M$ для каждого элемента $\Omega_e$, $e=1..N^e$ выполняется следующее:

Шаг 2.1. Если насыщенность $S^m_{\Omega_e} < S^m_{crit,\max}$, то пара чисел $(e, m)$ заносится в массив $I'_{abandon}$ и переход к следующим номерам $e$ и $m$ (на шаг 2).

Шаг 2.2. Вычисляется суммарный вытекающий поток из текущего элемента $Q^m_{out,\Omega_e} = \sum_{i\in I_{out}} Q^m_i$.

Шаг 2.3. Вычисляются значения $\Delta t_{me}$ по соотношению (39), учитывая, что в знаменателе выражения для расчёта $\Delta t_{me}$ стоит $\alpha^m \cdot \sum_{i\in I_{out,\Omega_e}} \tilde{Q}_{\Gamma_i,\Omega_e} \equiv Q^m_{out,\Omega_e}$.

Шаг 2.4. Если выполняется условие $\Delta t_{me} < \Delta t$, то присваиваем шагу по времени $\Delta t$ значение $\Delta t_{me}$.

Шаг 3. В цикле по номерам $(e, m)$ из множества $I'_{abandon}$ вычисляется объём $\hat{V}^m_{out} = Q^m_{out,\Omega_e} \cdot \Delta t$. Если $\hat{V}^m_{out} > mes(\Omega_e) S^m_{\Omega_e} \Phi$, то $(e, m)$ включается во множество $I_{abandon}$.

2.5.5. Подмодуль расчёта перетекающих объёмов фаз.

Входные данные:

7) Массив $\left\{Q^m_i\right\}_{i=1..N^f,m=1..M}$ перетекающих потоков фаз.

8) $\Delta t$ – шаг по времени.

9) Множество пар номеров элементов и фаз $I_{abandon}$.

10) Массив $\left\{S^m_{crit,\min}\right\}_{m=1..M}$ критических минимальных насыщенностей фаз.

Выходные данные:

1) Массив $\left\{V_i^m\right\}_{i=1..N^f,m=1..M}$ – перетекающих объёмов фаз.

Перед началом алгоритма необходимо сформировать структуру $\left\{I^e_{missed}\right\}_{e=1..N^e}$, в которой для каждого конечного элемента находится список номеров фаз, для которых будет производиться процедура «выталкивания» (смотри п. 1.5). Для этого в цикле по номерам фаз $m=1..M$ для каждого элемента $\Omega_e$, $e=1..N^e$ при условии, что $(e,m)\in I_{abandon}$ или $S^m_{\Omega_e}<S^m_{crit,\min}$, то для текущего элемента $e$ в список $I^e_{missed}$ добавляется номер фазы $m$.

Алгоритм расчёта перетекающих объёмов фаз:

Шаг 1. В цикле по конечным элементам $\Omega_e$, $e=1..N^e$, выполняется следующее:

Шаг 1.1. Если для текущего элемента список $I^e_{missed}$ пуст, то рассчитываются вытекающие из текущего элемента объёмы по соотношению $V_i^m=Q_i^m\cdot\Delta t,\ i\in I_{out}$. Перейти к следующему элементу (вернуться на шаг 1).

Шаг 1.2. В цикле по номерам фаз $m'\in I^e_{missed}$, вычислить объёмы «выталкиваемых» фаз: $V^{m'}_{\Omega_e}=mes\left(\Omega_e\right)S^k_{\Omega_e}\Phi$.

Шаг 1.3. Рассчитать величины долей $\alpha^m$ по соотношению (34).

Шаг 1.4. Рассчитать величины новых долей $\tilde{\alpha}^m$ для фаз, которые «не выталкиваются» из ячейки ($m\notin I^e_{missed}$), по соотношению (41).

Шаг 1.5. В цикле по граням $\Gamma_i$ текущего элемента $\Omega_e$ рассчитать величины новых вытекающих объёмов смеси $\tilde{V}_{\Gamma_j,\Omega_e}$ по соотношению (40).

Шаг 1.6. В цикле по граням $\Gamma_i$ текущего элемента $\Omega_e$ рассчитать вытекающие объёмы фаз по соотношению (42), если номер фазы $m \in I^e_{missed}$, и по соотношению (43), если номер фазы $m \notin I^e_{missed}$.

2.5.5. Подмодуль расчёта новых насыщенностей и компонентного состава фаз

Входные данные:

1) Массив $\left\{\bar{S}^m_{\Omega_e}\right\}_{e=1..N^e, m=1..M}$ насыщенностей фаз в элементах.

2) Массив $\left\{\bar{\chi}^{lm}_e\right\}_{e=1..N^e, m=1..M, l=1..L^m}$ компонентного состава фаз в элементах.

3) Массив $\left\{V^m_i\right\}_{i=1..N^f, m=1..M}$ перетекающих объёмов фаз.

Выходные данные:

4) Массив $\left\{S^m_{\Omega_e}\right\}_{e=1..N^e, m=1..M}$ новых насыщенностей фаз в элементах.

5) Массив $\left\{\chi^{lm}_e\right\}_{e=1..N^e, m=1..M, l=1..L^m}$ нового компонентного состава фаз в элементах.

Алгоритм расчёта новых насыщенностей и компонентного состава фаз:

Шаг 1. В цикле по номерам фаз $m = 1..M$ для каждого элемента $\Omega_e$, $e = 1..N^e$ выполняется следующее:

Шаг 1.1. Расчёт текущего объёма фазы в элементе $\bar{V}^m_{\Omega_e} = mes\left(\Omega_e\right)\bar{S}^m_{\Omega_e}\Phi$.

Шаг 1.2. В цикле по граням $\Gamma_i$ текущего элемента выполнить следующее:

Шаг 1.2.1 Рассчитать величины $V^m_{in,\Omega_e}$ и $V^m_{out,\Omega_e}$ по соотношениям (36) и (37).

Шаг 1.2.2. Если через текущую грань $\Gamma_i$ жидкость вытекает из элемента $\Omega_e$ в смежный с ней элемент $\Omega_k$, то вычислить величины $n^{lm}_{in,\Omega_k} = \sum_{i \in I_{in,\Omega_k}} \rho^m_{\Omega_{e_i}} \chi^{lm}_{\Omega_{e_i}} V^m_{\Gamma_i,\Omega_k} \Big/ \mathrm{M}_l$, $l = 1..L^m$.

Шаг 1.3. Расчёт нового объёма фазы в элементе $V^m_{\Omega_e} = V^m_{\Omega_e} + V^m_{in,\Omega_e} - V^m_{out,\Omega_e}$.

Шаг 1.4. Расчёт новых насыщенностей $S_{\Omega_e}^m = \dfrac{V_{\Omega_e}^m}{\sum\limits_{k=1}^{M} V_{\Omega_e}^k}$.

Шаг 2. В цикле по номерам фаз $m = 1..M$ для каждого элемента $\Omega_e$, $e = 1..N^e$ выполняется следующее:

Шаг 2.1. Расчёт количества вещества в $\Omega_e$ с учётом втекших и вытекших объёмов: $n_{\Omega_e}^{lm} = \rho_{\Omega_e}^m \chi_{\Omega_e}^{lm} \left( V_{\Omega_e}^m - V_{out,\Omega_e}^m \right) \Big/ \mathrm{M}_l + n_{in,\Omega_e}^{lm}$, $l = 1..L^m$.

Шаг 2.1. Расчёт новых долей $\chi_{\Omega_e}^{lm}$ по соотношению (9).

# 3. ВЕРИФИКАЦИЯ

Верификация разработанной программы проводилась поэтапно, для исключения возможных ошибок реализации модулей программы.

Первый этап включал в себя тестирование модулей программы расчёта давления и потоков смеси через границы ячеек на тестах с известным аналитическим решением. Так, для проверки реализации модуля расчёта давления было проведено тестирование на полиномиальных функциях для подтверждения теоретических порядков аппроксимации и сходимости. Тестирование подтвердило первый порядок аппроксимации и второй порядок сходимости для трилинейного базиса на сетках с шестигранными ячейками. Аналогично, для проверки реализации модуля расчёта потока через грани элементов использовались тесты на полиномиальных функциях. В результате первого этапа тестирования, были исключены возможные ошибки работы модулей программы, которые выполняют конечноэлементные расчёты.

Подробнее о последующих этапах верификации разработанной программы описано в данной главе.

## 3.1. Сравнение с двумерной реализацией

Для того чтобы убедиться в корректности полученных результатов, в случае моделирования трёхмерных задач, существует способ сравнения результатов на осесимметричных модельных задачах, полученных в двух постановках. Вне зависимости от того, в какой постановке: двумерной или трёхмерной, решена задача, результаты моделирования должны совпадать с точностью погрешности аппроксимации.

Для проведения этого этапа проверки была реализована программа для моделирования процессов многофазной фильтрации в осесимметричной постановке (цилиндрических координатах).

Тестирование проводилось на одномерной модельной задаче двухфазной фильтрации (решение не зависит от координаты $z$), схема которой изображена на рисунке 3.3.1.

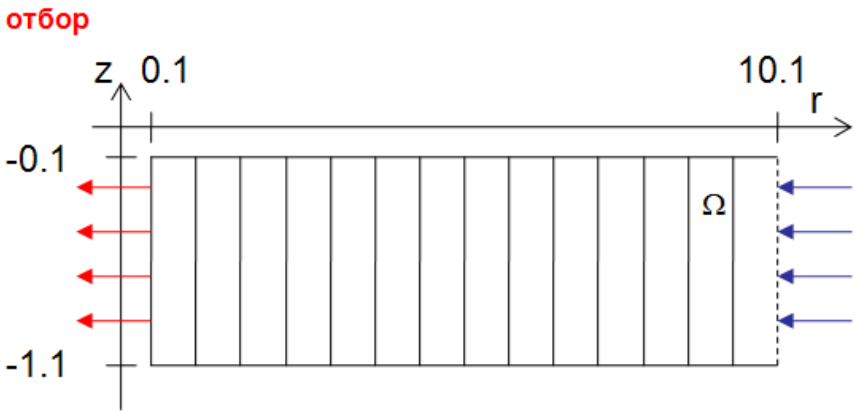



Рисунок 3.1. Схема модельной задачи для верификации реализаций. Вертикальными линиями схематически изображена конечноэлементная сетка.

Расчётная область тестовой задачи: $\Omega = [0.1, 10.1] \times [-1.1, -0.1]$. Вязкости фаз: $\eta^1 = 0.001$ м²/с, $\eta^2 = 0.005$ м²/с. Начальное распределение насыщенностей равномерное во всей области: $S^1 = 0.1$, $S^2 = 0.9$. На удалённой границе задано первое краевое условие $P|_{\Gamma_B} = 0$, а на границе со скважиной задан единичный отбор смеси: $\theta = 1$ м$^3$/с. На удалённой границе в область $\Omega$ поступает только фаза 1. Это означает, что со временем в $\Omega$ фаза 2 будет полностью вытеснена фазой 1. Поскольку решение не зависит, от координаты z, то результаты можно приводить вдоль линии на произвольной высоте, например z = –0.5. В связи с этим также отметим, что конечноэлементную сетку достаточно строить без дробления по координате z. Использовались вложенные конечноэлементные сетки h, h/2, h/4 и h/8, которые имеют 10, 20, 40 и 80 элементов по координате $r$ соответственно.

Для сравнения данных на рисунке 3.3.2 приведены распределения полей давления, посчитанные на сетке h. На рисунке 3.3 приведены графики насыщенностей $S^1(r,t)$ при времени $t = 5$ сек., $t = 15$ сек. и $t = 23$ сек. после начала отбора смеси из области на сетке h/2. Наибольшие отличия насыщенностей наблюдаются при времени $t = 15$ сек., при этом значения отличаются не более чем на 0.3%.

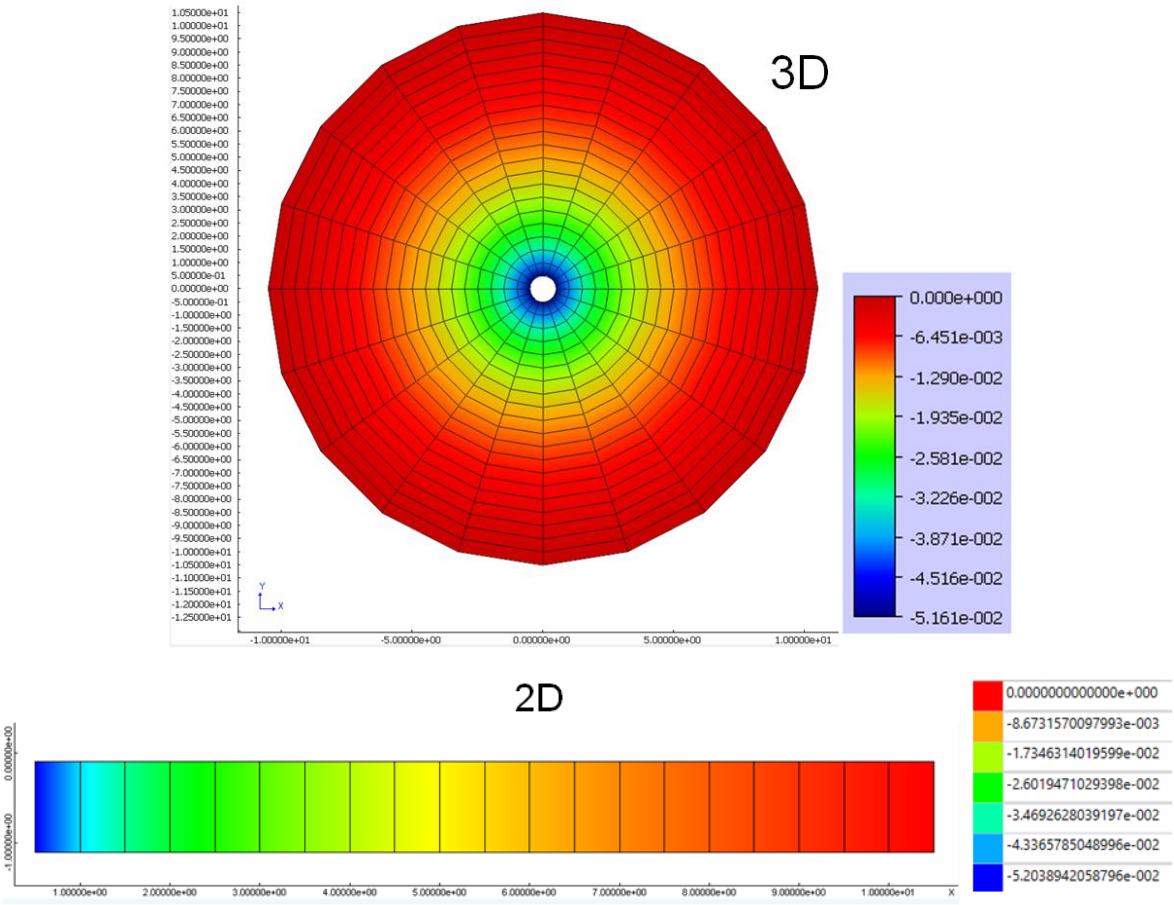



Рисунок 3.2. Сравнение 2D и 3D реализаций, распределение поля давления.

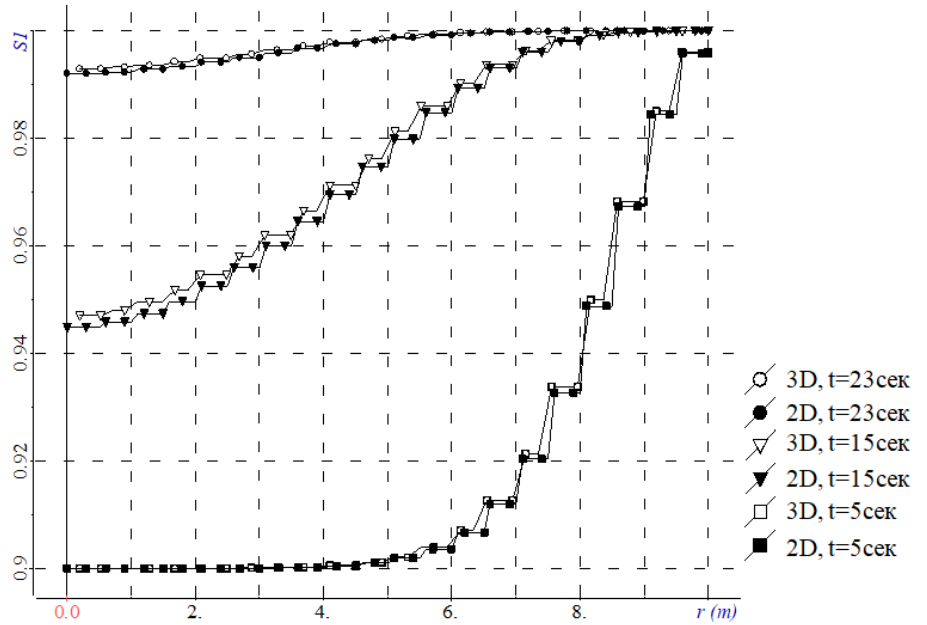



Рисунок 3.3. Сравнение 2D и 3D реализаций. Графики насыщенностей на временах 5, 15 и 23 секунды.

На рисунке 3.4 приведены графики насыщенностей $S^1(r,t)$ на разных временных слоях на вложенных сетках, посчитанных с помощью трёхмерной реализации. Из рисунка видна сходимость результатов вместе с дроблением сетки по пространству, что свидетельствует о корректности реализации вычислительной схемы.

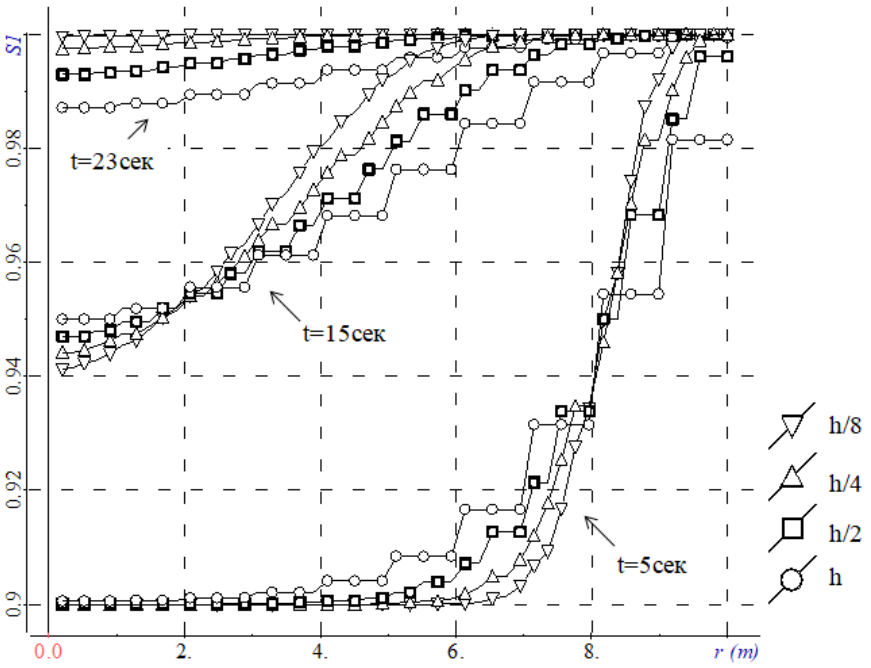



Рисунок 3.4. Графики насыщенностей, построенных на вложенных сетках, на временах 5, 15 и 23 секунды.

3.2. Балансировка потоков

Исследование проведено на задаче, эмулирующей работу добывающей и группы из нагнетательных скважин. Кровля и подошва нефтеносного пласта заданы на уровне –910 м и –912 м соответственно, ниже расположен непроницаемый пропласток толщиной 0.2 м, под которым находится водоносный слой. На рисунке 3.5 представлено распределение начальной насыщенности нефти в коллекторе (которая соответствует 17.6 тыс. м$^3$ нефти). На участке x<–400 м, y>–80 м непроницаемый участок отсутствует, и коллектор непосредственно контактирует с водоносным слоем. Отбор смеси из добывающей скважины равен 40 м$^3$/сут. Объем жидкости, закачиваемой в

нагнетательные скважины, распределен между ними равномерно, а суммарный объем закачиваемой жидкости равен отбору смеси из добывающей скважины. Максимальное время, до которого выполняется моделирование процесса фильтрации, – 40 лет.

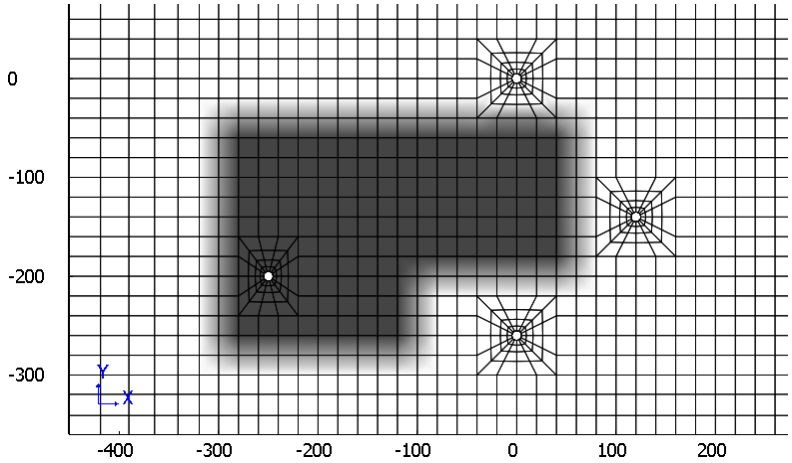
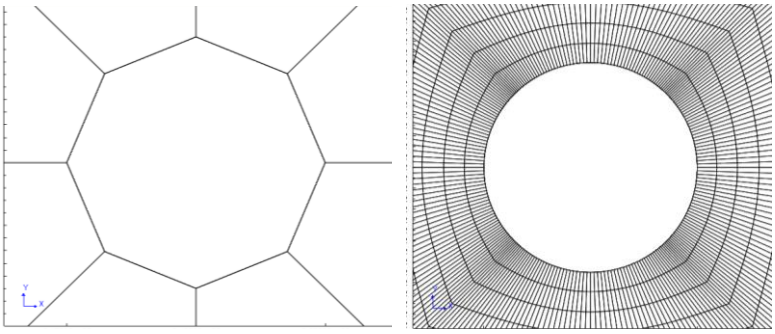



Рис. 3.5. Начальное распределения насыщенности нефти в коллекторе.

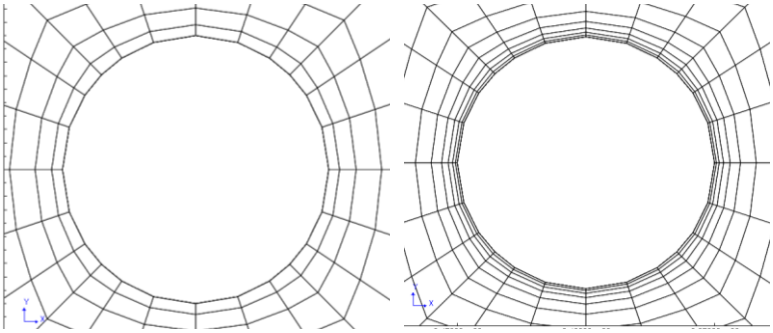
Расчёты проводились на пяти конечноэлементных сетках $\tau_h$, $\tau_{h/2}$, $\tau_{h/4}$ $\tau_{h/8}$, $\tau_{h/16}$ и шести сетках $\tilde{\tau}_{h/2}$, $\tilde{\tilde{\tau}}_{h/2}$, $\tilde{\tau}_{h/4}$, $\tilde{\tilde{\tau}}_{h/4}$, $\tilde{\tau}_{h/8}$, $\tilde{\tilde{\tau}}_{h/8}$. При этом сетки $\tau_{h/2}$, $\tau_{h/4}$, $\tau_{h/8}$, $\tau_{h/16}$ задавались почти вложенными по латерали к предыдущей (т.е. $\tau_{h/2}$ к $\tau_h$, $\tau_{h/4}$ к $\tau_{h/2}$ и т.д.) с уточняющейся аппроксимацией скважин, а сетки $\tilde{\tau}_i$ и $\tilde{\tilde{\tau}}_i$ были получены на основе сеток $\tau_{h/2}$, $\tau_{h/4}$, $\tau_{h/8}$ различным сгущением к скважинам. Вид нескольких характерных сеток в окрестности скважин изображен на рисунке 3.6. Параметры сеток приведены в таблице 3.1.

Рассмотрим три варианта расчета процесса фильтрации:

1. объемы перетекающей смеси модифицируются с использованием предложенного метода балансировки;
2. метод балансировки не используется, а для граней (5) осредненного решения (6) производится фиксация значений объемов закачиваемой или откачиваемой смеси на границах скважин;
3. метод балансировки и фиксация значений объемов закачиваемой или откачиваемой смеси не используются.



(а) (б)



(в) (г)

Рис. 3.6. Вид конечноэлементных сеток $\tau_h$ (а), $\tau_{h/16}$ (б), $\tilde{\tau}_{h/2}$ (в) и $\tilde{\tilde{\tau}}_{h/2}$ (г) в окрестности скважин.

Таблица 3.1 Параметры используемых конечноэлементных сеток

| Сетка | Количество узлов | Количество элементов |
|---|---|---|
| $\tau_h$, | 6964 | 4944 |
| $\tau_{h/2}$, | 21300 | 15420 |
| $\tau_{h/4}$, | 64980 | 47676 |
| $\tau_{h/8}$, $\tilde{\tau}_{h/8}$, $\tilde{\tilde{\tau}}_{h/8}$ | 202164 | 149628 |
| $\tau_{h/16}$ | 672724 | 500736 |
| $\tilde{\tau}_{h/2}$ | 23604 | 17148 |
| $\tilde{\tilde{\tau}}_{h/2}$ | 25140 | 18300 |
| $\tilde{\tau}_{h/4}$ | 66772 | 49020 |
| $\tilde{\tilde{\tau}}_{h/4}$ | 70356 | 51708 |

На Рис. 3.7 приведены графики отбора нефти при использовании метода балансировки объемов смеси на конечноэлементных сетках $\tau_h$, $\tau_{h/2}$, $\tau_{h/4}$, $\tau_{h/8}$ (обозначены $h$, $h/2$, $h/4$ и $h/8$, соответственно).

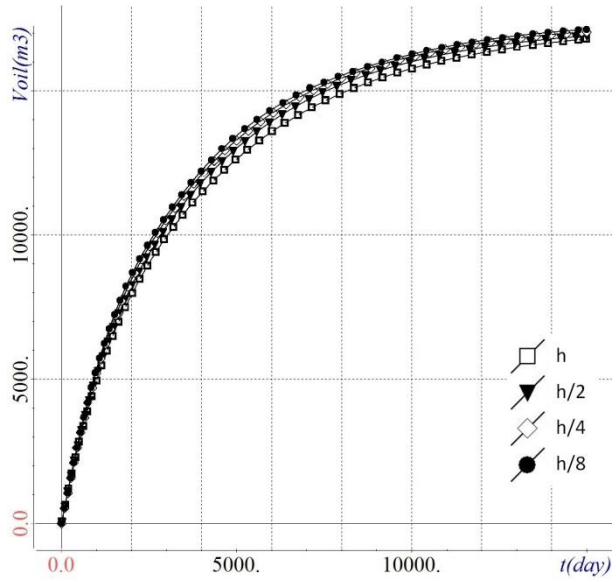



Рис. 3.7. График отбора нефти на сетках $\tau_h$, $\tau_{h/2}$, $\tau_{h/4}$ и $\tau_{h/8}$ при использовании метода балансировки.

На Рис. 3.8 приведены графики околоскважинной насыщенности нефти для сеток $\tau_h$, $\tau_{h/2}$, $\tau_{h/4}$, $\tau_{h/8}$. Видно, что околоскважинная насыщенность нефти почти не меняется с дроблением сетки.

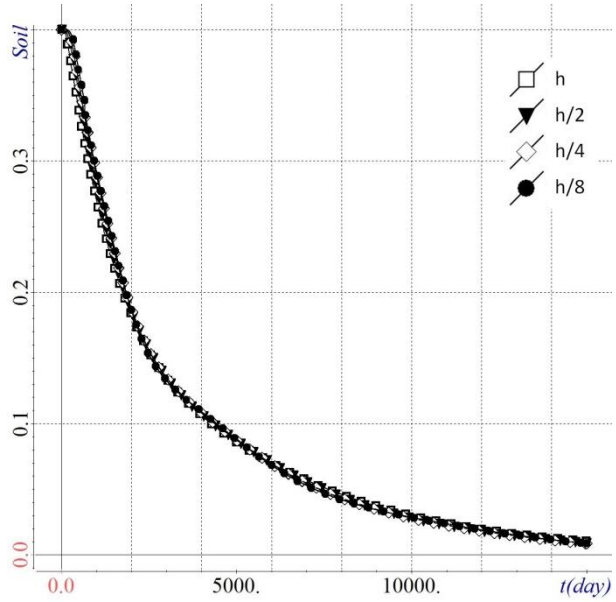



Рис. 3.8. График околоскважинной насыщенности нефти на сетках $\tau_h$, $\tau_{h/2}$, $\tau_{h/4}$ и $\tau_{h/8}$ при использовании метода балансировки.

На Рис. 3.9 приведены графики отбора без использования метода балансировки на сетках $\tau_h$, $\tau_{h/2}$, $\tau_{h/4}$, $\tau_{h/8}$, $\tau_{h/16}$ в сравнении с графиком отбора нефти с балансировкой потоков, рассчитанным на сетке $\tau_{h/8}$ (на Рис. 5 и далее результаты, полученные в этом расчете, обозначены как $h/8^*$).

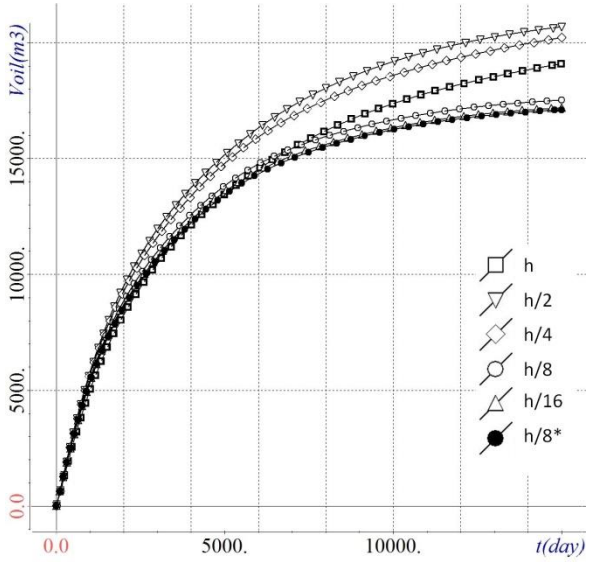



Рис. 3.9. График отбора нефти без использования метода балансировки на сетках $\tau_h$, $\tau_{h/2}$, $\tau_{h/4}$, $\tau_{h/8}$ и график отбора на сетке $\tau_{h/8}$ с использованием метода балансировки.

На Рис. 3.10 приведены графики суммарного баланса нефти в системе (сумма отобранной и остаточной нефти в области) без балансировки для сеток $\tau_h$, $\tau_{h/2}$, $\tau_{h/4}$, $\tau_{h/8}$ в сравнении с расчетом на сетке $\tau_{h/8}$ с балансировкой.

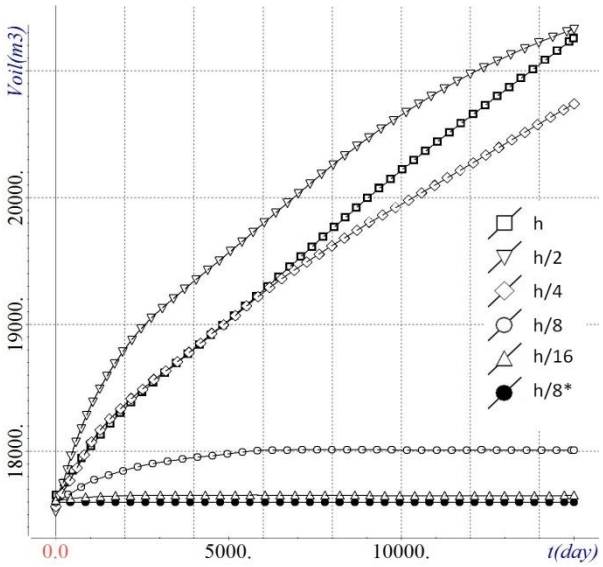



Рис. 3.10. Графики суммарного баланса нефти в системе.

На Рис. 3.11 приведены графики отбора нефти без использования метода балансировки и без фиксации значений объемов закачиваемой (или откачиваемой) жидкости на сетках $\tau_h$, $\tau_{h/2}$, $\tau_{h/4}$, $\tau_{h/8}$ в сравнении с графиками с балансировкой на сетке $\tau_{h/8}$.

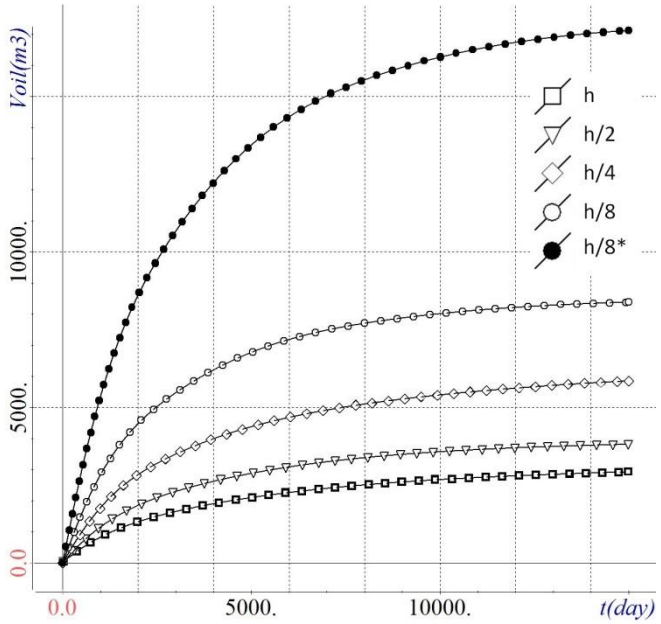



Рис. 3.11. Графики отбора нефти сбалансированных потоков и без использования метода балансировки и фиксации значений объемов закачиваемой (или откачиваемой) жидкости на сетках $\tau_h$, $\tau_{h/2}$, $\tau_{h/4}$, $\tau_{h/8}$.

На Рис. 3.12 приведены графики отбора нефти без использования метода балансировки и фиксации значений объемов закачиваемой или откачиваемой жидкости на сетках $\tilde{\tau}_{h/2}$, $\tilde{\tilde{\tau}}_{h/2}$, $\tilde{\tau}_{h/4}$, $\tilde{\tilde{\tau}}_{h/4}$, $\tilde{\tau}_{h/8}$ и $\tilde{\tilde{\tau}}_{h/8}$ (графики обозначены как $\tilde{h}/2$, $\tilde{\tilde{h}}/2$, $\tilde{h}/4$, $\tilde{\tilde{h}}/4$, $\tilde{h}/8$ и $\tilde{\tilde{h}}/8$ соответственно) со сгущением к добывающей и нагнетающим скважинам в сравнении с графиками с балансировкой.

По графикам, представленным на Рис. 3.9, видно, что с дроблением сетки графики отбора без использования метода балансировки, в целом, сходятся к «эталонному» решению, однако эта сходимость немонотонная: для сеток $\tau_{h/2}$ и $\tau_{h/4}$ графики отбора для варианта без балансировки лежат значительно дальше от графика с балансировкой, чем даже для сетки $\tau_h$, и только на сетке $\tau_{h/8}$ и $\tau_{h/16}$ наблюдается «регулярная» сходимость к результатам, полученным в варианте с

балансировкой. При этом только на сетке $\tau_{h/8}$ отклонения графика без балансировки от графика с балансировкой стало сопоставимым с отклонением графика, полученным в варианте с балансировкой на самой грубой сетке.

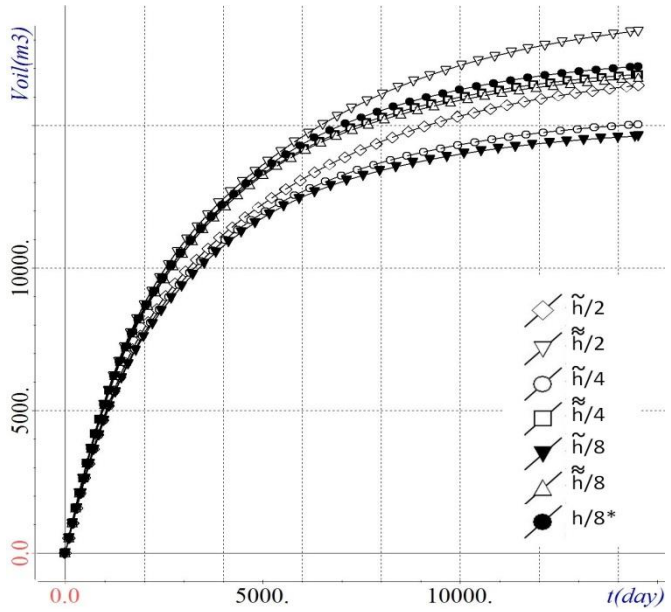



Рис. 3.12. График отбора нефти сбалансированных потоков и без использования метода балансировки и фиксации значений объемов закачиваемой (или откачиваемой) жидкости на сетках $\tilde{\tau}_{h/2}$, $\tilde{\tilde{\tau}}_{h/2}$, $\tilde{\tau}_{h/4}$, $\tilde{\tilde{\tau}}_{h/4}$, $\tilde{\tau}_{h/8}$ и $\tilde{\tilde{\tau}}_{h/8}$

Из графиков на Рис. 3.10 видны значительные потери объёмов в системе в расчётах без применения метода балансировки, которые, однако, с дроблением сеток уменьшаются вместе с падением погрешности.

Из представленных на Рис. 3.11 данных можно сделать вывод, что полученные значения отбора нефти при несбалансированных потоках и без фиксации расхода неприемлемо грубы, и даже на достаточно подробной сетке $\tau_{h/8}$ не соответствуют порядку решения, полученного с использованием метода балансировки потоков. Более того, мгновенный объем отобранной нефти (производная интегрального отбора) совершенно неадекватно отражает процесс добычи на начальном этапе. Главная причина этого – отсутствие сильного сгущения сетки в окрестности скважин, необходимого для данного варианта расчета.

Из рисунка 3.12 можно заметить, что при выбранном сгущении сеток графики отбора без использования метода балансировки и фиксации значений объемов закачиваемой и откачиваемой жидкости подходят гораздо ближе к графику, полученному в варианте с балансировкой, но вместе с тем для устойчивой сходимости все же требуются слишком подробные сетки.

В таблице 3.2 приведены значения затрат времени при моделировании процесса фильтрации смеси с использованием трех методов: «B» – балансировка потоков, «NB» – балансировка не используется, но объемы жидкости, перетекающие через границы скважин, фиксируются, «NBNF» – не используется балансировка и фиксация значений объемов перетекающей через границы скважин жидкости. Расчеты выполнялись на компьютере с процессором Intel Core i7-8700K с частотой 3.70GHz, имеющим 6 вычислительных ядер.

Таблица 3.2 Временные затраты на моделирование процесса фильтрации

| Метод, сетка | Время | Метод, сетка | Время |
|---|---|---|---|
| «B», $\tau_h$ | 16 с | «NBNF», $\tau_h$ | 8 с |
| «B», $\tau_{h/2}$ | 1 мин 30 с | «NBNF», $\tau_{h/2}$ | 56 с |
| «B», $\tau_{h/4}$ | 14 мин 24 с | «NBNF», $\tau_{h/4}$ | 10 мин 34 с |
| «B», $\tau_{h/8}$ | 2 ч 25 мин | «NBNF», $\tau_{h/8}$ | 1ч 46 мин |
| «NB», $\tau_h$ | 14 с | «NBNF», $\tilde{\tau}_{h/2}$ | 1 ч 14 мин |
| «NB», $\tau_{h/2}$ | 1 мин 43 с | «NBNF», $\tilde{\tilde{\tau}}_{h/2}$ | 8 ч 30 мин |
| «NB», $\tau_{h/4}$ | 15 мин | «NBNF», $\tilde{\tau}_{h/4}$ | 4 ч 51 мин |
| «NB», $\tau_{h/8}$ | 2 ч 24 мин | «NBNF», $\tilde{\tilde{\tau}}_{h/4}$ | 28 ч 38 мин |
| «NB», $\tau_{h/16}$ | 27 ч 28 мин | «NBNF», $\tilde{\tau}_{h/8}$ | 13 ч 9 мин |
| | | «NBNF», $\tilde{\tilde{\tau}}_{h/8}$ | 85 ч 14 мин |

Из таблицы 3.2 видно, что для получения необходимой точности решения (полученной на сетке $\tau_{h/2}$ в варианте «B») без использования метода

балансировки потоков для задач «NB», $\tau_{h/16}$ и «NBNF», $\tilde{\tilde{\tau}}_{h/8}$ вычислительные затраты увеличиваются на 3 порядка. С увеличением размера конечноэлементной сетки, учитывающей более сложную геометрию реальных нефтяных месторождений (количество нагнетающих и добывающих скважин может быть порядка 100), использование предложенного метода будет являться критически важным.

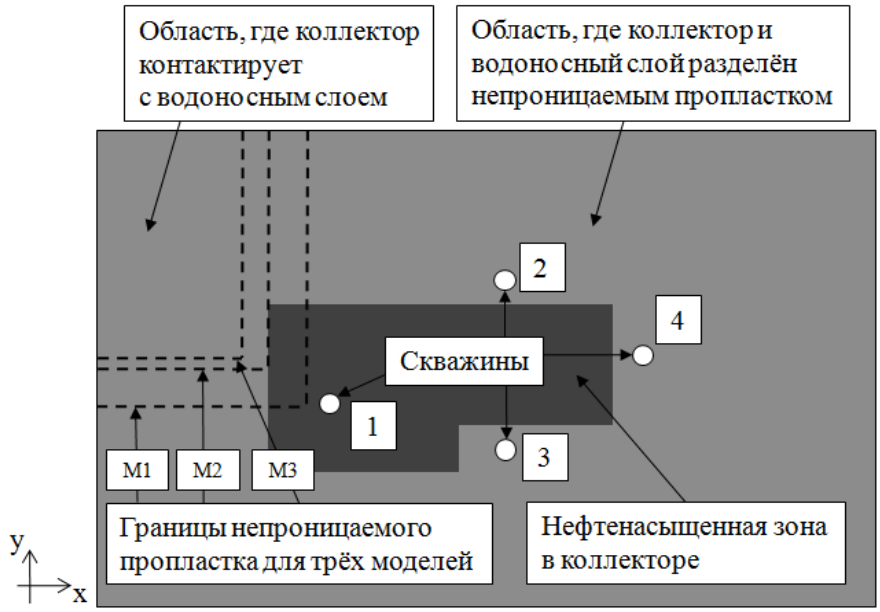
# 4. ИССЛЕДОВАНИЯ

В настоящее время большое количество российских нефтяных месторождений заканчивают добычу нефти с помощью первичных методов разработки, при которых нефть выходит на поверхность в основном за счёт внутренней энергии пласта. Эффективность такого способа добычи нефти в основном определяется точностью попадания добывающей скважины в целевой горизонт. Однако после первого этапа разработки значительная часть нефти может остаться в породе и для её добычи требуются специальные технологии, которые основаны на нагнетании в целевой пласт воды. Нагнетание в целевой пласт вытесняющих агентов может нести двойную функцию: первая – поддержание внутрипластового давления, а вторая – нейтрализация потоков воды из близлежащих водоносных пластов.

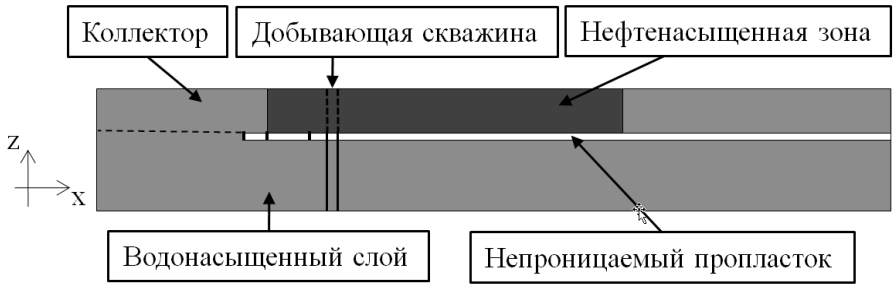
В данной главе приведены результаты двух численных экспериментов. В первом проводилось моделирования многофазного потока в пористой среде в технологиях нефтедобычи, использующих полимерное заводнение, а во втором проводилось моделирование многофазного потока при различных контактах водоносного и нефтенасыщенного слоев.

## 4.1. Моделирование многофазного потока в пористой среде в технологиях нефтедобычи, использующих полимерное заводнение

Рассмотрим четыре модели нефтяного месторождения с коллектором, где расположена нефтенасыщенная зона, и водоносным слоем, расположенным под коллектором. Геометрия нефтенасыщенной зоны в плане и положение четырех скважин (1 – добывающая скважина, 2, 3, 4 – нагнетательные скважины) показаны на рис. 4.1а. Между коллектором и находящимся под ним водонасыщенным слоем находится непроницаемый пропласток, который на части месторождения может отсутствовать, и тогда там коллектор контактирует с водонасыщенным слоем. На рис. 4.1б показан фрагмент соответствующего разреза.




(а)




(б)

Рис. 4.1. Схема модели месторождения с контактом нефтенасыщенного слоя с водоносным

Обозначим через M0 модель, в которой непроницаемый пропласток везде разделяет коллектор и водонасыщенный слой (т.е. водонасыщенный слой нигде не контактирует с коллектором). Через M1, M2 и M3 обозначим модели, в которых в части области $\left\{x < X_{M_k}, y > Y_{M_k}\right\}$ непроницаемый пропласток отсутствует и коллектор непосредственно контактирует с водоносным слоем. Значения величин $X_{M_k}$ и $Y_{M_k}$ для этих моделей представлены в таблице 4.1.

Задачей исследования является изучение влияния различных режимов нагнетания на эффективность нефтедобычи при различных контактах коллектора с водоносным слоем.

Таблица 4.1. – Параметры границ в плане непроницаемого слоя

| Модель | $X_{M_k}$, м | $Y_{M_k}$, м |
|---|---|---|
| М1 | -400 | -80 |
| М2 | -360 | -100 |
| М3 | -300 | -200 |

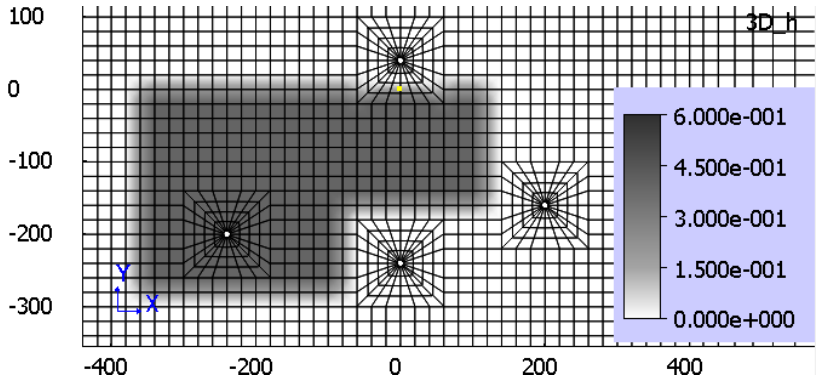
Расчёты процесса фильтрации проведем на модели, эмулирующей работу добывающей и трёх нагнетательных скважин. Структурная пористость моделируемого пласта равна 0.2, а структурная проницаемость – $0.4 \cdot 10^{-3}$ мкм$^2$. Во всех расчётах объем добываемой смеси равен 40 м$^3$/сут. Объемы жидкости, закачиваемые через нагнетательные скважины, приведены в таблице 2 для четырех режимов.

Таблица 4.2. – Режимы работы нагнетательных скважин

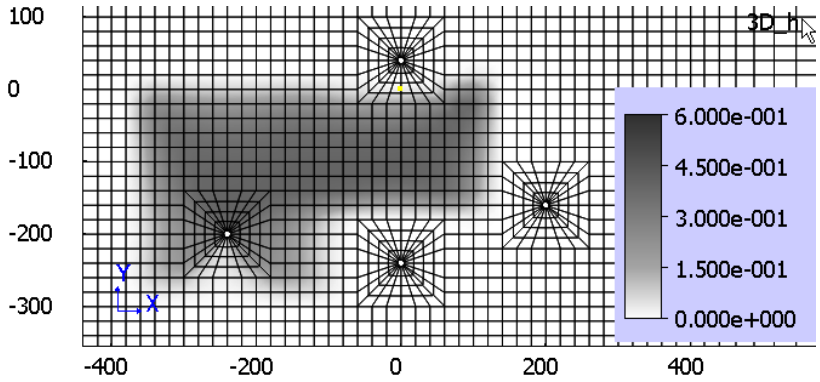
| Режим | Мощность нагнетания скважин, м$^3$/сут | | |
|---|---|---|---|
| | № 2 | № 3 | № 4 |
| 1 | 13.3 | 13.3 | 13.3 |
| 2 | 6.6 | 6.6 | 13.3 |
| 3 | 6.6 | 6.6 | 26.6 |
| 4 | 6.6 | 6.6 | 40 |

На рис. 4.2 изображена в плане используемая в исследовании конечноэлементная сетка и поля распределения насыщенностей в разные моменты времени для модели М1 и режима 1 работы нагнетательных скважин.

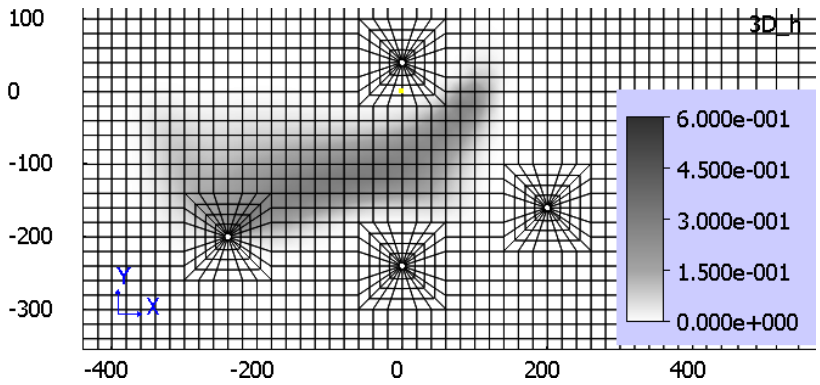
На рис. 4.3 изображены графики отборов нефти для всех рассматриваемых моделей в сравнении с моделью М0. На всех графиках пустыми маркерами обозначены отборы без дополнительного нагнетания в системе.




(а)

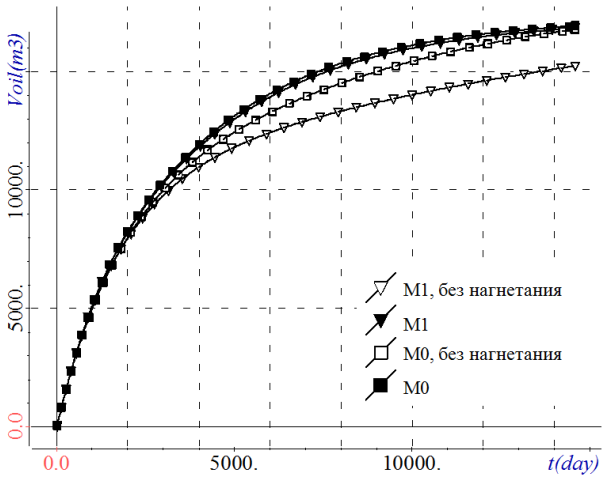



(б)

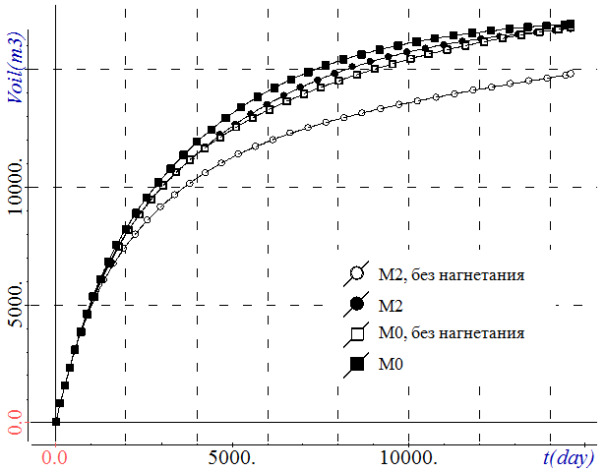



(в)

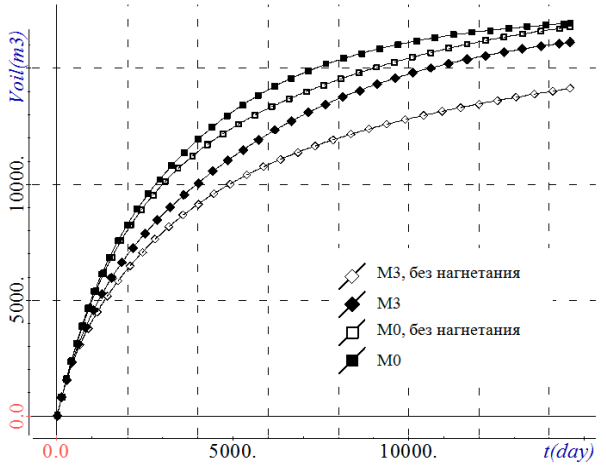
Рис. 4.2. Поле распределения насыщенности нефти в начальный момент времени (а), при t=1000 сут. (б) и при t=5000 сут. (в).




(а)




(б)




(в)

Рис. 4.3. Графики отбора нефти для трёх рассматриваемых моделей с нагнетанием и без нагнетания при режиме 1 работы нагнетательных скважин.

На рис. 4.4 приведены графики отборов для модели М3 для всех четырех рассматриваемых режимов нагнетания в сравнении с ситуацией, когда коллектор нигде не контактирует с водоносным слоем (модель М0).

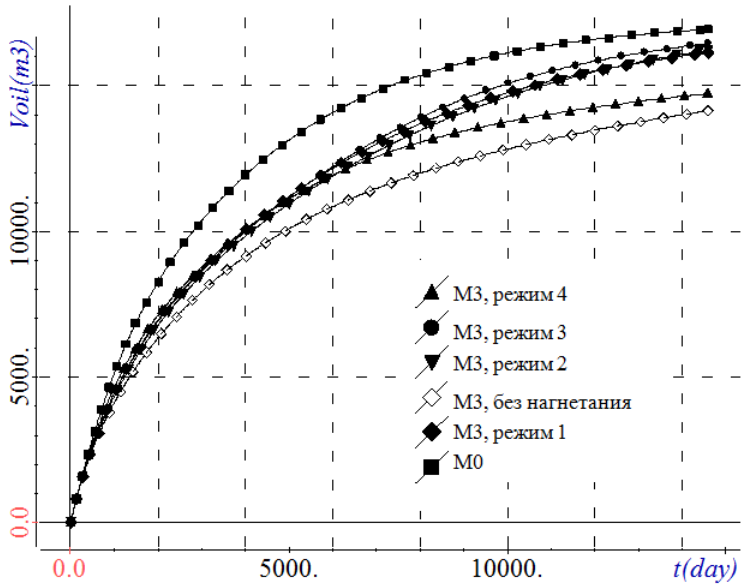



Рис. 4.4. Графики отбора нефти для модели М3 при всех режимах работы нагнетательных скважин.

Приведенные на Рис. 4.4 результаты показывают, что слишком больший напор нагнетательных скважин (режим 4) может существенно ухудшить результат, поскольку более сильный напор воды создаёт эффект, при котором добывающая скважина «не успевает» собрать всю нефть из области при высокой скорости вытеснения нефти водой. Также видно, что режим нагнетания 3 оказался немного выигрышнее остальных.

## 4.2. Моделирование многофазного потока в пористой среде в технологиях нефтедобычи, использующих полимерное заводнение

Повышение эффективности нефтеотдачи может достигаться за счёт внедрения в производственный процесс новых современных цифровых технологий.

Первичный этап разработки месторождения происходит за счёт притоков нефти к добывающей скважине под действием природной внутренней энергии пласта (естественного внутрипластового давления). На поздних стадиях разработки необходимо прибегать к вторичным и третичным методам добычи, которые используют искусственные методы воздействия на нефтенасыщенный пласт, например, закачивание в пласт более вязких вытесняющих агентов. Для каждого месторождения необходимо выбирать свои технологии и составы вытесняющих агентов, учитывая геологические свойства месторождения и свойства залегающих углеводородов. От выбора технологий вытеснения во многом зависит рентабельность разработки соответствующего месторождения.

Одним из методов разработки месторождений для добычи трудноизвлекаемой нефти, является закачка в пласт вытесняющего агента, обладающего повышенной вязкостью, например, водный раствор полимера. Такое воздействие на пласт может приносить несколько положительных эффектов. Во-первых, дополнительное нагнетание позволяет поддерживать внутрипластовое давление. Во-вторых, повышенная вязкость вытесняющего агента даёт преимущество течению нефтяной фазы через поры, поскольку по законам фильтрации менее вязкие флюиды быстрее мигрируют в пласте, другими словами, такие воздействия на пласт заставляют продвигаться нефть к добывающей скважине быстрее, чем водный раствор с полимером. Очевидно, что положительный экономический эффект будет определяться стоимостью и объёмами использованного полимера для получения дополнительного объема нефти, причём для разных типов залегающей нефти (которые могут отличаться, например, вязкостью) будут требоваться разные объёмы и составы вытесняющего агента. Поэтому ставится задача определения оптимального расхода полимера при использовании его водного раствора в качестве вытесняющего агента.

Характерный разброс вязкости залегающей нефти находится в довольно большом диапазоне: от 2.8 до 693.8 мПа·с [17]. Поэтому мы проведём три

группы экспериментов для подбора оптимальной концентрации полимера в вытесняющем агенте для трёх величин вязкостей нефти: 5, 50 и 500 мПа·с.

На практике могут использоваться разные концентрации полимера, поэтому мы возьмём три возможные концентрации из соображений, что стоимость полимера высока, и его общий закачиваемый объём должен быть минимальным. Поэтому будем исследовать агенты с концентрациями: 1, 0.1 и 0.01%.

Существует зависимость вязкости водяного раствора от концентрации полимера в нём. В общём случае эта зависимость является функцией не только концентрации полимера, но и линейной скорости фильтрации в пласте [18]. В данной работе, по результатам предварительных исследований будем считать, что средняя линейная скорость фильтрации в пласте составляет порядка десятков миллиметров в сутки, а соответствующая этой скорости зависимость вязкости вытесняющего агента задана таблично (график зависимости вязкости раствора от концентрации полимера изображён на Рис. 4.5).

Критериями выбора наилучшего варианта можно считать как величину дополнительного объема добытой нефти, так и приращение объема добываемой нефти на единицу массы используемого полимера.

Проведем расчёты процесса двухфазной фильтрации на модели, эмулирующей работу добывающей и трёх нагнетательных скважин. Геометрическое положение скважин и конфигурация нефтяного пятна изображены на рис. 2. Расчёты проведём для случая, когда суммарный объем жидкости, закачиваемой в нагнетательные скважины, равен отбору смеси из добывающей и составляет 40 $м^3$/сут.

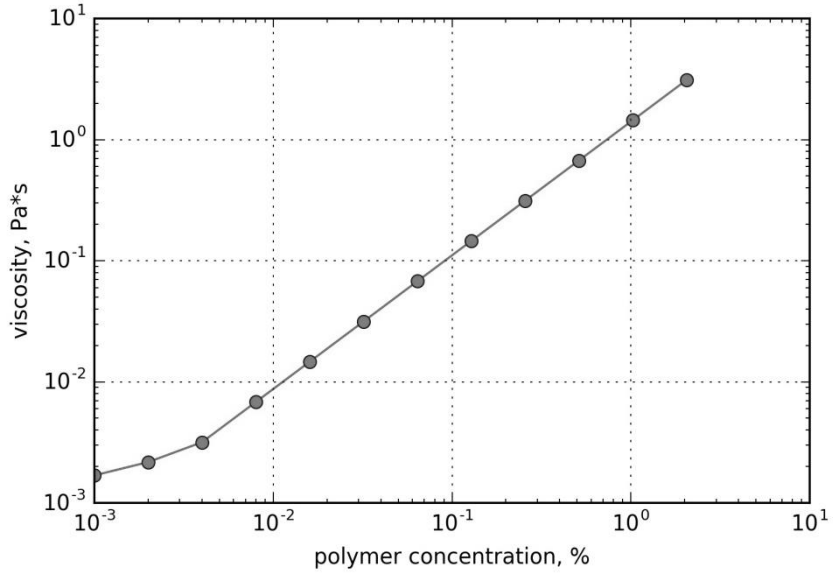


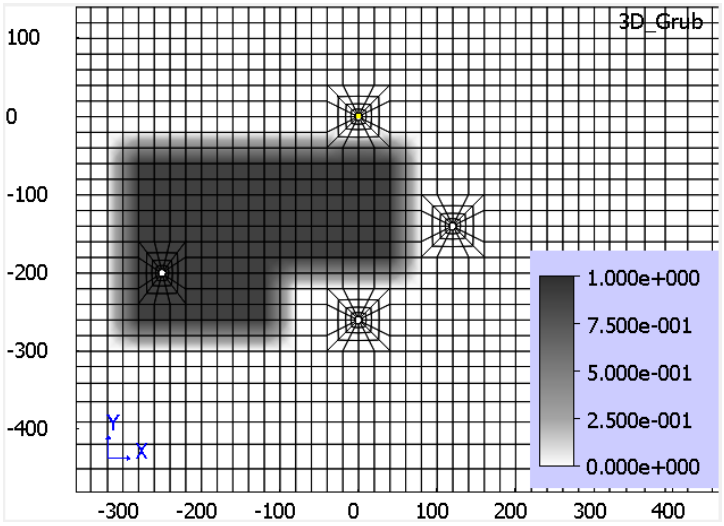

Рис. 4.5. Зависимость вязкости раствора от концентрации полимера.

Структурная пористость моделируемого пласта равна 0.2, а структурная проницаемость – $0.4 \cdot 10^{-3}$ мкм$^2$. Начальное распределение насыщенности нефти в расчётной области изображено на Рис. 2,а.

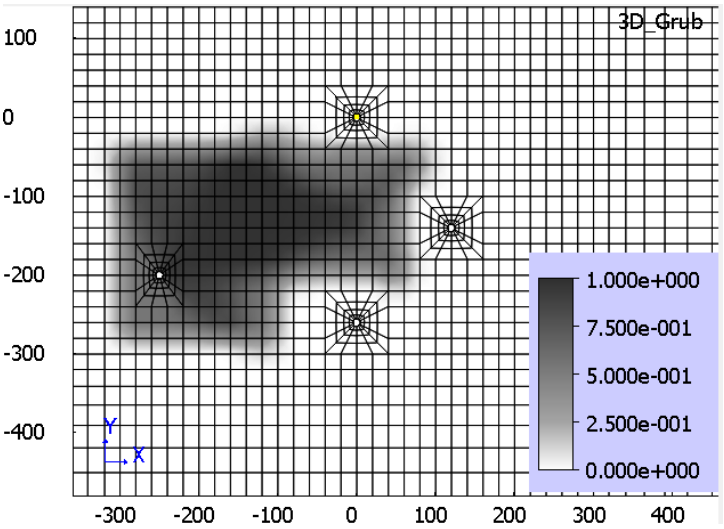
Рассмотрим три ситуации: в пласт на протяжении 625 суток закачивается водный раствор полимера с концентрациями 1, 0.1 и 0.01%, соответственно. После 625 суток во всех трёх ситуациях в нагнетательные скважины закачивается только вода (без примесей полимера).

На Рис. 4.6 и Рис. 4.7 приведены поля распределений насыщенностей нефти и концентрации полимера в разные моменты времени для варианта, когда вязкость нефти равна 50 мПа·с.

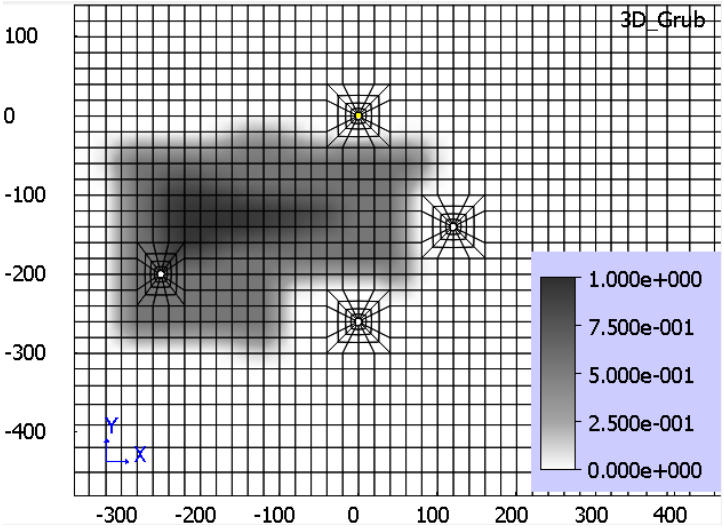
На Рис. 4.7а показана концентрация полимера в области в момент времени t=625 суток, после которого в область начинает закачивается только вода. К этому времени полный закаченный объём раствора полимера составляет 25 тыс. м$^3$ (т.е. примерно 250, 25 и 2.5 тонн полимера для концентраций 1, 0.1 и 0.01%).




(а)

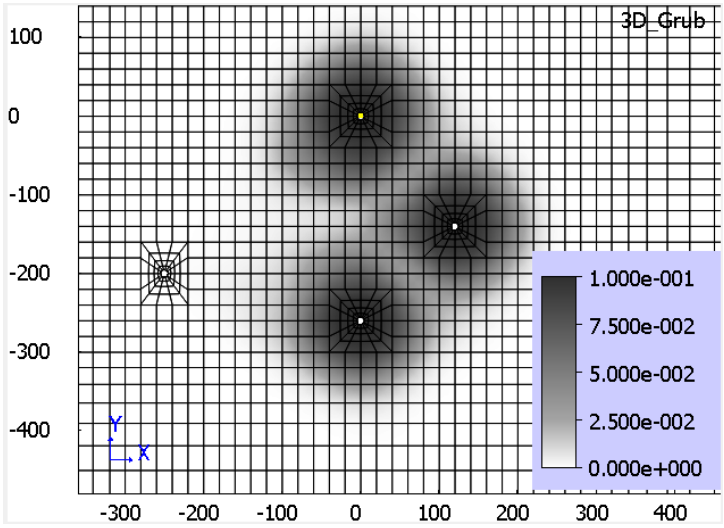



(б)

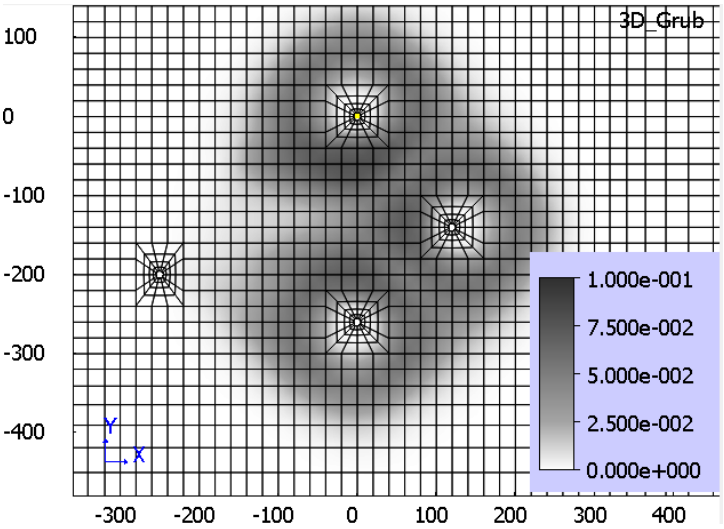



(в)

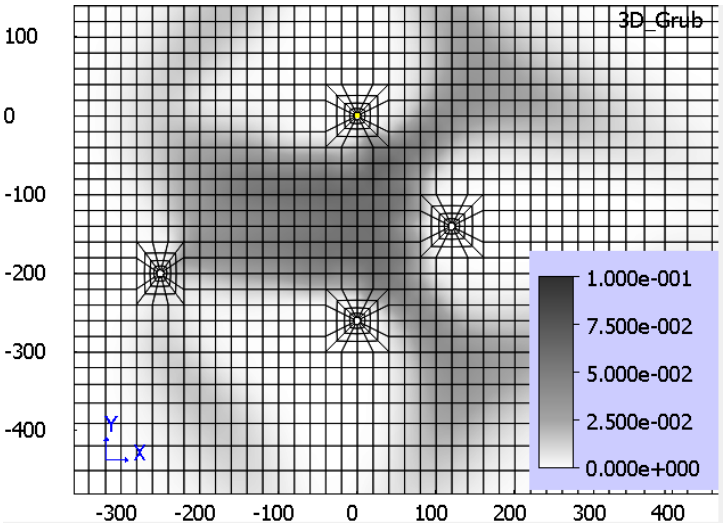
Рис. 4.6. Поле распределения насыщенности нефти в начальный момент времени (а), при t=1000 сут (б) и при t=4000 сут (в).




(а)




(б)



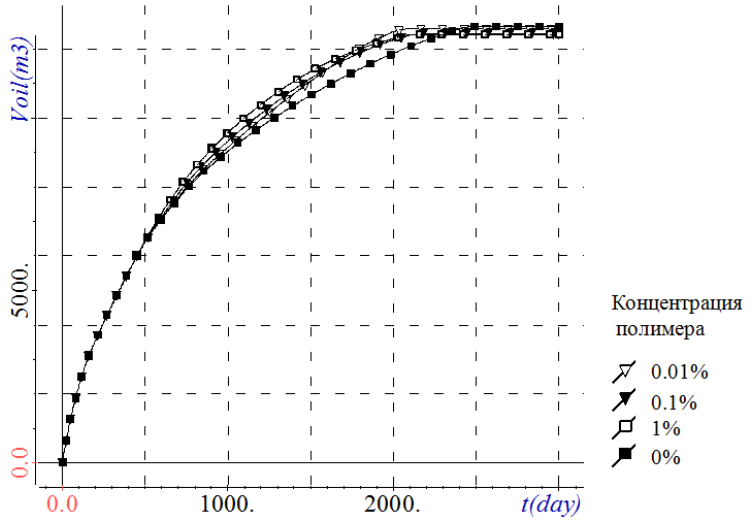

(в)

Рис. 4.7. Поле распределения концентрации полимера (в процентах) в момент времени при t=625 сут (а), при t=1000 сут (б), при t=4000сут. (в)

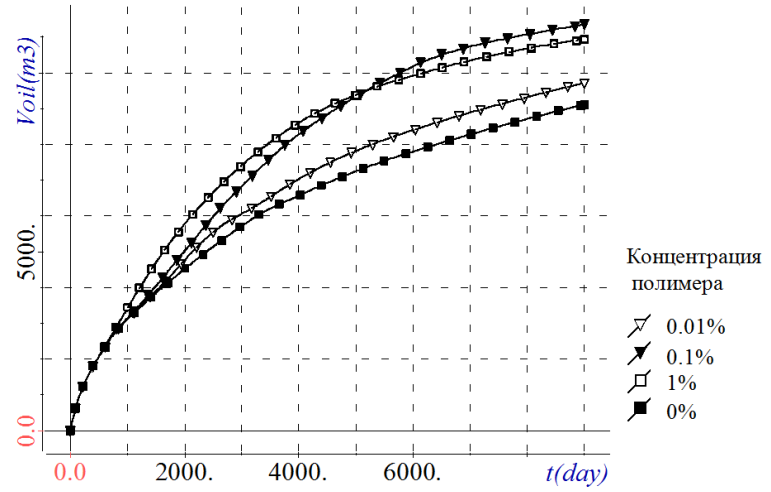
На Рис. 4.8 представлены графики, на которых показан отбор нефти из области при разных концентрациях полимера для разных вариантов вязкости нефти. По полученным результатам можно оценить оптимальный объём использованного полимера, рассчитав дополнительный объём добытой нефти на единицу массы закаченного полимера. Дополнительно добытый объём нефти для всех вариантов расчётов приведён в таблице 4.3. В ней для нефти с вязкостью 5 мПа·с приведены данные на момент времени t=1500 сут., а для 50 и 500 мПа·с – t=4000 сут. Для варианта с вязкостью нефти 5 мПа·с на момент времени t=1500 сут. для концентрации 1% дополнительно добытый объём нефти (в $м^3$) на единицу массы закаченного полимера (в тоннах) составил 2.7, для концентрации 0.1% – 19.6, для 0.01% – 162.8. Для варианта с вязкостью нефти 50 мПа·с на момент времени t=4000 сут. для концентрации 1% дополнительно добытый объём нефти на единицу массы закаченного полимера составил 8.2, для концентрации 0.1% – 68.7, 0.01% – 186. И для варианта с вязкостью нефти 500 мПа·с на момент времени t=4000 сут.: 1% – 5.5, 0.1% – 17.4, 0.01% – 64.

Таблица 4.3 – Дополнительно добытый объём нефти

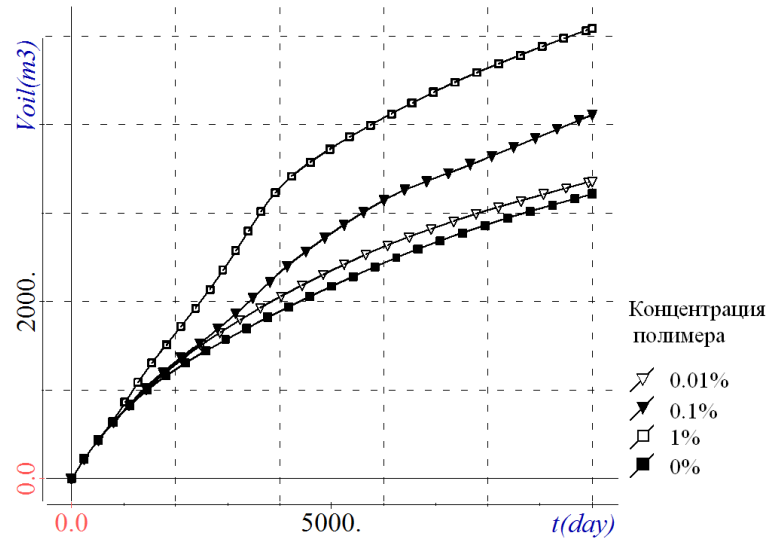
| Концентрация полимера, % | Для вязкости нефти 5 мПа·с, $м^3$ | Для вязкости нефти 50 мПа·с, $м^3$ | Для вязкости нефти 500 мПа·с, $м^3$ |
|---|---|---|---|
| 1 | 687 | 2038 | 1393 |
| 0.1 | 491 | 1714 | 437 |
| 0.01 | 407 | 465 | 160 |




(а)




(б)




(в)

Рис. 4.8. Графики отбора нефти при вязкостях нефти 5 мПа·с (а), 50 мПа·с (б) и 500 мПа·с (в) при разных концентрациях полимера в вытесняющем агенте.

Проведенное моделирование показывает, что использование полимерного заводнения позволяет получать весьма существенные добавочные объемы добываемой нефти. Эффективность использования полимеров, как и ожидалось, растет с увеличением вязкости нефти. Вместе с тем оптимальная концентрация полимера определяется неоднозначно и зависит не только от структуры геологической среды, вязкости нефти, морфологии месторождения, взаиморасположения добывающих и нагнетательных скважин, но и от приоритета той или иной цели, для достижения которой принимается решение об использовании полимеров. Так, например, если основной целью является получение максимального добавочного объема на единицу массы использованного полимера, то наиболее эффективным будет применение растворов с низкой концентрацией полимера. Особенно ярко это видно для маловязкой ($\eta = 5$ мПа·с) и средневязкой ($\eta = 50$ мПа·с) нефти. Но если более приоритетной целью является получение добавочных объемов добываемой нефти, то вполне оправданным становится использование растворов с более высокой концентрацией полимера.

Так, для случая высоковязкой нефти ($\eta = 500$ мПа·с) при использовании раствора с концентрацией 1% объем добытой нефти на момент времени t=4000 сут. увеличился более чем на 70%, в то время как для концентрации 0.1% – на 20%, а для концентрации 0.01% – только на 10% (хотя по критерию максимума дополнительного объема добываемой нефти на единицу массы использованного полимера лучшим является вариант с концентрацией полимера 0.01%).

Для средневязкой нефти ($\eta = 50$ мПа·с) ситуация совсем другая. Здесь дополнительно добытый объем нефти к моменту времени t=4000 сут. примерно одинаков для концентраций полимера 1% и 0.1%, и поэтому использовать раствор с концентрацией 1% в такой ситуации и рассмотренных нами условий не имеет смысла (так как этот вариант однозначно хуже по критерию затрат полимера на единицу дополнительно добытой нефти).

# ЗАКЛЮЧЕНИЕ

В работе описана математическая модель процессов многофазной фильтрации в пористых средах для случая несжимаемых жидкостей.

Применение в описанной вычислительной схеме метода конечных элементов позволяет проводить численные эксперименты на задачах со сложными геометрическими областями, произвольным количеством нагнетательных и добывающих скважин. В работе используются трилинейный базис на регулярных конечноэлементных сетках с шестигранными ячейками.

Поскольку численное решение при использовании метода конечных элементов не удовлетворяет условию сохранения баланса втекающих и вытекающих потоков через границы ячеек области, в работе описан способ балансировки потоков с помощью минимизации функционала для поиска корректирующих добавок к потокам, рассчитанным по конечноэлеметному решению. Если известны мощности нагнетания и добычи жидкости в области, такой подход позволяет полностью скорректировать потоки в области и в каждой ячейке в отдельности по заданным значениям мощностей.

В работе подробно описаны алгоритмы и вычислительные схемы для вычисления новых состояний ячеек области. В частности описан алгоритм автоматического выбора шага дискретизации по времени. Такой подход позволяет проводить длительные расчёты на временных сетках с нефиксированных шагом, что ускоряет процесс моделирования и исключает появление возможных ошибок, связанных некорректным выбором временного шага.

Основным результатом данной работы является программа, позволяющая проводить моделирование процессов многофазной фильтрации в пористых средах с учётом многокомпонентности фаз для случая изотермических и несжимаемым жидкостей.

С помощью разработанной программы были проведены численные эксперименты для решения двух задач, обладающих практическим интересом. Первая задача включала исследование влияния контакта водонасыщенного слоя

с коллектором вблизи в окрестности добывающей скважины. Проведенное моделирование показало, что при наличии контакта коллектора с водоносным слоем использование нагнетательных скважин может существенно повысить объем добываемой нефти. При этом оптимальный режим нагнетания должен выбираться в зависимости от месторождения и условий контакта коллектора с водоносным слоем.

Целью второго исследования являлось определение оптимального расхода полимера при использовании его водного раствора в качестве вытесняющего агента для разных вариантов вязкости нефти. Проведенное моделирование показывает, что использование полимерного заводнения позволяет получать весьма существенные добавочные объемы добываемой нефти.

## СПИСОК ЛИТЕРАТУРЫ


1. Kullawan K. Sequential geosteering decisions for optimization of real-time well placement / Kullawan K., Bratvold R.B., Bickel J.E. // Journal of Petroleum Science and EngineeringVolume – 2018 – Vol. 165 – pp. 90-104. – DOI: https://doi.org/10.1016/j.petrol.2018.01.068
2. Bautista J.F Prediction of formation damage at water injection wells due to channelization in unconsolidated formations / Bautista, J.F., Taleghani A.D. // Journal of petroleum science and engineering – 2018 – Vol. 64 – pp. 1-10 – DOI: 10.1016/j.petrol.2017.12.073
3. Pogaku R. Polymer flooding and its combinations with other chemical injection methods in enhanced oil recovery / Pogaku, R., Fuat N.H.M., Sakar, S., Cha Z.W., Musa, N., Tajudin, D.N.A.A., Morris, L.O. // POLYMER BULLETIN – 2018 – Vol. 75 – №4 – pp. 1753-1774 – DOI: 10.1007/s00289-017-2106-z
4. Clemens T. Optimizing Water-Injection Design in a Shallow Offshore Reservoir / Clemens T., Kienberger G., Persaud M., Suri A., Sharma M.M., Boschi M., Overland A.M. // SPE production & operations – 2017 – Vol. 32 – №4 – pp. 551-563
5. Aristov S. Integrated Approach to Managing Formation Damage in Waterflooding / Aristov S., Paul van den Hoek, Eddie Pun // SPE European Formation Damage Conference and Exhibition, Budapest, Hungary – 2015 –DOI: https://doi.org/10.2118/174174-MS
6. Ахметзянов Р.Р., Жильцов А.А., Самойлов В.В., Булгаков А.П., Самойлов Д.Ю. Как повысить эффективность системы поддержания пластового давления при разработке месторождений // ТЕРРИТОРИЯ НЕФТЕГАЗ. - 2015. - №2. - С. 14-17.
7. Roxar RSM [Электронный ресурс]. – Режим доступа: http://roxar.ru/
8. Сидельников К.А., Васильев В.В. Анализ применений математического моделирования пластовых систем на базе метода линий тока / Нефтегазовое дело. - 2005 - С. 1-11

9. Schlumberger ECLIPSE [Электронный ресурс]. – Режим доступа: http://www.sis.slb.ru/ECLIPSE/

10. Zhang R. Numerical simulation of water flooding in natural fractured reservoirs based on control volume finite element method / Zhang, R. H., Zhang, L. H., Luo, J. X., Yang, Z. D., & Xu, M. Y // Journal of Petroleum Science and Engineering. – 2016. – Vol. 146. – pp. 1211-1225. DOI: 10.1016/j.petrol.2016.08.024.

11. Abushaikha, A. S. Interface control volume finite element method for modelling multi-phase fluid flow in highly heterogeneous and fractured reservoirs / Abushaikha, A. S., Blunt, M. J., Gosselin, O. R., Pain, C. C., & Jackson, M. D. // Journal of Computational Physics – 2015. – Vol. 298. – pp. 41-61. DOI: 10.1016/j.jcp.2015.05.024.

12. Nick H.M. and Matthäi S.K. A Hybrid Finite-Element Finite-Volume Method with Embedded Discontinuities for Solute Transport in Heterogeneous Media // Vadose Zone Journal. – 2011. – Vol. 10. – №. 1. – pp. 299-312. DOI: 10.2136/vzj2010.0015.

13. Nick H.M. and Matthäi S.K. Comparison of Three FE-FV Numerical Schemes for Single- and Two-Phase Flow Simulation of Fractured Porous Media // Transport in Porous Media. – 2011. – Vol. 90. – №. 2. – pp. 421-444. DOI: 10.1007/s11242-011-9793-y.

14. Азиз Х., Сеттари Э. Математическое моделирование пластовых систем: Пер. с англ. — М.: Недра, 1982. — 407 с.

15. Соловейчик, Ю.Г. Метод конечных элементов для скалярных и векторных задач / Ю.Г. Соловейчик, М.Э. Рояк, М.Г. Персова. – Новосибирск : НГТУ, 2007. – 869 с.

16. Persova M. G., Vagin D. V., Abramov M. V. Finite element meshing for calculating the stress-strain behavior of structures with stress-raisers // 11 International forum on strategic technology (IFOST 2016) : proc., Novosibirsk, 1–3 June 2016. – Novosibirsk : NSTU, 2016. – Pt. 1. – P. 371-374. - ISBN 978-1-5090-0853-7. - DOI: 10.1109/IFOST.2016.7884131

17. Willhite Paul G. Waterflooding – Society of Petroleum Engineers Richardson, TX, 1986

18. Lopes L. F., B. Silveira M.O., Moreno R. B. Z. L. Rheological Evaluation of HPAM fluids for EOR Applications // International Journal of Engineering & Technology, Vol:14 No:03, June 2014 IJENS, pp. 35-41

# ПРИЛОЖЕНИЕ А. ТЕКСТ ПРОГРАММЫ

В данном разделе приведены фрагменты текста программы.

main.cpp

```
#define _CRT_SECURE_NO_WARNINGS 1

#include"filtr_cfg.h"
#include "Mesh.h"
#include"Filtr_Structs.h"
using namespace fltrStruct;
#include"FEM_compiler.h"
#include"Filtration.h"
#include"FEM_utils.h"

#include"ClassMethodGaussFast.h"
#include"ClassMethodGaussFast2D.h"

#include<iostream>
#include<iomanip>
#include "TimeLogger.h"

streambuf *bakOut;
ofstream fileOut;

int storing(string & pathOutput, int iT, int iT_print, double time,
FiltrProperties & prop, FiltrationClass & Filtr)
{
 ofstream ofp, ofpTime;
 string path, pathTime, pathLog;

 pathLog = pathOutput + "\\storlog_A";
 ofp.open(pathLog);
 ofp << 0 << endl;
 ofp.close();
 ofp.clear();
 path = pathOutput + "\\storageA";
 system(string("rmdir /s /q " + path).c_str());
 _mkdir(path.c_str());

 // ------------------
 // Выгрузка файлов для восстановления расчёта A
       prop.WriteS(path, iT);
       Filtr.WriteTotalVolumes(path);
 // ------------------

 pathTime = pathOutput + "\\timesA";
 ofpTime.open(pathTime);
 ofpTime << iT << endl;
 ofpTime << iT_print << endl;
 ofpTime << time << endl;
 ofpTime.close();
 ofpTime.clear();

 ofp.open(pathLog);
 ofp << 1 << endl;
 ofp.close();
 ofp.clear();
```

```
	pathLog = pathOutput + "\\storlog_B";
	ofp.open(pathLog);
	ofp << 0 << endl;
	ofp.close();
	ofp.clear();

	path = pathOutput + "\\storageB";
	system(string("rmdir /s /q " + path).c_str());
	_mkdir(path.c_str());

	// ------------------
	// Выгрузка файлов для восстановления расчёта B
		prop.WriteS(path, iT);
		Filtr.WriteTotalVolumes(path);
	// ------------------

	pathTime = pathOutput + "\\timesB";
	ofpTime.open(pathTime);
	ofpTime << iT << endl;
	ofpTime << iT_print << endl;
	ofpTime << time << endl;
	ofpTime.close();
	ofpTime.clear();

	ofp.open(pathLog);
	ofp << 1 << endl;
	ofp.close();
	ofp.clear();
	return 0;
	}

    int reStoring(string & pathInput, Restart &restart, FiltrProperties & prop,
FiltrationClass & Filtr)
    {
	ifstream infLog, inf;
	string path, pathLog;
	int textlog;
	int flagAorB; // 1 -> A, -1 -> B, 0 - error
	pathLog = pathInput + "\\storlog_A";
	infLog.open(pathLog);
	flagAorB = -1;
	if (infLog.good())
	{
		infLog >> textlog;
		if (infLog.good())
		{
			if (textlog == 1)
				flagAorB = 1;
		}
	}
	infLog.close();
	infLog.clear();
	if (flagAorB == 1)
	{
		path = pathInput + "\\timesA";
		inf.open(path);
		inf >> restart.it;
		inf >> restart.it_print;
		inf >> restart.Ttotal;
		inf.close();
```

```
        inf.clear();
        path = pathInput + "\\storageA";

        // ------------------
        // Чтение файлов для восстановления расчёта A
        prop.ReadS(path, restart.it);
        Filtr.ReadTotalVolumes(path);
        // ------------------

    }
    else
    {
        pathLog = pathInput + "\\storlog_B";
        infLog.open(pathLog);
        if (infLog.good())
        {
            infLog >> textlog;
            if (infLog.good())
            {
                if (textlog == 1)
                {
                    path = pathInput + "\\timesB";
                    inf.open(path);
                    inf >> restart.it;
                    inf >> restart.it_print;
                    inf >> restart.Ttotal;
                    inf.close();
                    inf.clear();
                    path = pathInput + "\\storageB";

                    // ------------------
                    // Чтение файлов для восстановления расчёта B
                    prop.ReadS(path, restart.it);
                    Filtr.ReadTotalVolumes(path);
                    // ------------------
                }
                else
                    flagAorB = 0;
            }
        }
        infLog.close();
        infLog.clear();
    }
    if (flagAorB == 0)
        return 1;
    else
        return 0;
}

int main()
{
    auto& timeLogger = TimeLogger::instance();
    timeLogger.start("All time");

    timeLogger.start("prepare");
    int ret;
    bool buildCrossSection;
    bool fres = true;
    const string fileStop = "filtr.stop";
    ifstream infStop;
```

```
double dt;

MoveCoutToFile();

Restart restart;
ret = restart.ReadSimple();
if (ret)
{
      cout << "STOP by restart " << ret;
      system("pause");
      return 1;
}

Config cfg;
ret = cfg.Read();
if (ret)
{
      cout << "Function cfg.Read() return " << ret;
      system("pause");
      return 1;
}
cfg.ConvertDataToSec();

string path = cfg.PathOutput + "\\" + cfg.OutputName;
string path3D = cfg.PathOutput + "\\" + cfg.OutputName3D;
string pathStorage = cfg.PathOutput + "\\storage";
string pathNebalance3D = cfg.PathOutput + "\\" + "nebalance3D";

if (!restart.On())
{
      system(string("rmdir /s /q " + path).c_str());
      _mkdir(path.c_str());

      // --- RESTART ---
      system(string("rmdir /s /q " + pathStorage).c_str());
      _mkdir(pathStorage.c_str());
      system(string("rmdir /s /q " + path3D).c_str());
      _mkdir(pathStorage.c_str());
}

MeshXYZ mesh;
ret = mesh.MeshRead(cfg.PathInput);
if (ret)
{
      cout << "Function mesh.MeshRead() return " << ret;
      system("pause");
      return 1;
}
FiltrProperties fltrProp;
fltrProp.InitMesh(mesh.kolel);
ret = fltrProp.ReadPhase(cfg.PathInputProp);
if (ret)
{
      cout << "Function FitrProperties::ReadPhase return " << ret;
      system("pause");
      return 1;
}
ret = fltrProp.ReadMaterials(cfg.PathInputProp);
if (ret)
{
```

```
            cout << "Function FitrProperties::ReadMaterials return " << ret;
            system("pause");
            return 1;
      }
      if (mesh.kolmat != fltrProp.kolmat)
      {
            cout << "kolmat in mesh (" << cfg.PathInput << ") != kolmat in
materials.txt (" << cfg.PathInputProp << ")" << endl;
            system("pause");
            return 1;
      }
      cout << "kolel " << mesh.kolel << " koluz " << mesh.koluz << endl;
      if (!restart.On())
      {
            MakeFolderFor3D(cfg.PathInput, path3D);
            PrintSmtr(path3D, mesh);
            Print_fields_cnf(path3D);
            if (cfg.fPrintNebalance3D)
            {
                  MakeFolderFor3D(cfg.PathInput, pathNebalance3D);
                  PrintSmtr(pathNebalance3D, mesh);
                  Print_fields_cnf(pathNebalance3D);
            }
      }
      vector<vector<double>> s_usred;
      s_usred.resize(mesh.koluz, vector<double>(fltrProp.kolphase, 0.));

      vector<double> times;
      if (!restart.On())
            times.push_back(0);

      mesh.MakeLinBF();
      mesh.MakeNode2elem();
      mesh.MakeFaces2node();
      mesh.MakeFacesDirect();

      MethodGaussFast_FEM femCalculator3D;
      MethodGaussFast_FEM2D femCalculator2D;
      CalcAmountElem(mesh, femCalculator3D);
      FEM_Pressure Task(&mesh, fltrProp, femCalculator3D, femCalculator2D);
      FiltrationClass Filtr(&mesh, fltrProp, femCalculator3D, femCalculator2D);

      readRules((cfg.PathInputProp + "/rules.txt").c_str(), Filtr.rules);

      fltrProp.ReadBC1(cfg.PathInput);
      fltrProp.ReadBC2(cfg.PathInput);

      vector<int> FaceRenum;

      Filtr.RenumerateFaces(FaceRenum);
      for (int i = 0; i < (int)Task.bc2faces.size(); i++)
      {
            fltrProp.bc2faces[i].face= FaceRenum[fltrProp.bc2faces[i].face];
      }
      for (int i = 0; i < (int)Task.bc1faces.size(); i++)
      {
            fltrProp.bc1faces[i].face = FaceRenum[fltrProp.bc1faces[i].face];
      }
      mesh.MakeFaces2node();
```

```
Filtr.BuildTmatrixGran();

ret = fltrProp.ReadWells(cfg.PathInput);
if (ret)
{
      cout << "fltrProp.ReadWells return " << ret << endl;
      return 1;
}
ret = fltrProp.ReadWellsSettings(cfg.PathInputProp);
if (ret)
{
      cout << "fltrProp.ReadWellsSettings return " << ret << endl;
      return 1;
}

ret = fltrProp.ReadSCritical(cfg.PathInputProp);
if (ret)
{
      cout << "fltrProp.ReadSCritical return " << ret << endl;
      return 1;
}
ret = fltrProp.ReadPhaseComponentsStruct(cfg.PathInputProp);
if (ret)
{
      cout << "fltrProp.ReadPhaseComponentsStruct return " << ret << endl;
      return 1;
}

ret = fltrProp.ReadRelationPhaseEnviron(cfg.PathInputProp);
if (ret)
{
      cout << "fltrProp.ReadPhaseComponentsStruct return " << ret << endl;
      return 1;
}

ret = fltrProp.ReadSbcCommon_interval_z(cfg.PathInput);
if (ret)
{
      ret = fltrProp.ReadSbcCommon(cfg.PathInput);
      if (ret)
            return 1;
}
else
{
      Filtr.InitSbcCommon_interval_bc1faces();
}

ret = fltrProp.ReadCompProp(cfg.PathInputProp);
if (ret)
{
      cout << "Function ReadCompProp return " << ret;
      system("pause");
      return 1;
}
fltrProp.InitMemory();
if (!restart.On())
{
      ret = fltrProp.ReadS0(cfg.PathInput);
      if (ret)
            return 1;
```

```
	}
	else
	{
		ret = reStoring(pathStorage, restart, fltrProp, Filtr);
		if (ret == 1)
		{
			cout << "fltrProp.reStoring return " << ret << endl;
			cout << "May be a both files storageA and storageB failed" <<
endl;
			system("pause");
			return 1;
		}
	}
	timeLogger.start("Output");
	//-------OUTPUT---------------
	ifstream inf_intersec("filtr.intersec");
	double intersec_coord = 0.0;
	int intersec_fc = 2;
	if (inf_intersec.good())
	{
		inf_intersec >> intersec_fc >> intersec_coord;
	}
	if (!inf_intersec.good())
	{
		intersec_coord = 0.0;
		intersec_fc = 2;
	}

	// Grapher
	string pathGraph = ".\\graph";
	if (!restart.On())
	{
		system(string("rmdir /s /q " + pathGraph).c_str());
		_mkdir(pathGraph.c_str());
	}
	auto grapher = Grapher(pathGraph, mesh);
	try {
		ifstream path_graph("lines.txt");
		ifstream path_graph2("time_points.txt");
		if (path_graph.good())
			grapher.ReadLines(path_graph);
		if(path_graph2.good())
			grapher.ReadTimePoints(path_graph2);
	}
	catch (exception& exc) {
		cerr << "Error in graph file: " << exc.what() << endl;
		exit(1);
	}

	timeLogger.stop("Output");
	if (!restart.On())
	{
		ofstream ofp_dt("dt.grph");
		ofp_dt << endl << endl;
		ofp_dt.close();
		ofp_dt.clear();

		ofp_dt.open("dt_time.grph");
		ofp_dt << endl << endl;
		ofp_dt.close();
```

```
            ofp_dt.clear();

            ofp_dt.open("nobalanceSing.grph");
            ofp_dt << endl << endl;
            ofp_dt.close();
            ofp_dt.clear();

            ofp_dt.open("nobalanceUnsing.grph");
            ofp_dt << endl << endl;
            ofp_dt.close();
            ofp_dt.clear();

            CalcUsredSaturation(mesh, fltrProp, s_usred);
            for (int iuz = 0; iuz < mesh.koluz; iuz++)
            {
                  s_usred[iuz].resize(fltrProp.kolphase + 1);
            }
            CalcUsredComp(mesh, fltrProp, s_usred);
            Print3D_S(path3D, s_usred, 0);
            PrintTimesFast(path3D, "times_main", 0, 0);
      }

      timeLogger.stop("prepare");

      int iT (0), iT_print(0);
      if (restart.On())
      {
            iT = restart.it;
            iT_print = restart.it_print;
            Filtr.Ttotal = restart.Ttotal;
      }
      for (iT; iT < cfg.NT && Filtr.Ttotal <= cfg.Time_all; iT++)
      {
            infStop.open(fileStop);
            if (infStop.good())
            {
                  printf_s("\n\nSTOP CALC");
                  cout << "STOP CALC" << endl;
                  break;
            }

            cout << "------------------------------" << endl;
            cout << "iT = " << iT << " time = " << scientific << Filtr.Ttotal /
24. / 3600. << defaultfloat << endl;
            printf_s("iT = %d time = %le\r", iT, Filtr.Ttotal / 24. / 3600.);
            cout << "------------------------------" << endl;
            cout << "CalcTask0" << endl;

            // проверка изменения
            ret = fltrProp.InitWellsSettings(Filtr.Ttotal);
            if (ret)
            {
                  for (int h = 0; h < fltrProp.kolholes; h++)
                        MakeHolesByConsumption(mesh,            femCalculator2D,
fltrProp.holes[h], fltrProp.holes_consumption[h]);
                  fltrProp.InitBC2ByHoles();
            }

            dt = 0;
            timeLogger.start("Iter");
```

```
            {
                  fltrProp.CalcPropDensity();
                  fltrProp.CalcPropEta();
                  fltrProp.CalcPropKoefPermeability();
                  fltrProp.PhasesElem0 = fltrProp.PhasesElem;

                  Task.CalcTask0(false, iT); // (не переключай на полиномы (там
Гаусс ч-з лябмду выключен))
                  fltrProp.PullP(Task.get_P());

                  timeLogger.start("Filtr_step");
                  Filtr.CALC_crit(cfg, iT, dt);
                  timeLogger.stop("Filtr_step");

            }
            timeLogger.stop("Iter");

            timeLogger.start("Output");

            // --- STORING ---
            storing(pathStorage,  iT  +  1,  iT_print,  Filtr.Ttotal,  fltrProp,
Filtr);

            timeLogger.start("Output");
            ofstream ofp("dt.grph", ios::app);
            ofp << iT << "\t" << setprecision(8) << scientific << dt / 3600. /
24. << endl;
            ofp.close();
            ofp.clear();

            ofp.open("dt_time.grph", ios::app);
            ofp << Filtr.Ttotal / 3600. / 24. << setprecision(8) << scientific
<< "\t" << dt / 3600. / 24. << endl;
            ofp.close();
            ofp.clear();

            {
                  iT_print++;
                  //-------OUTPUT--3D-----------
                  if (iT_print == 1)
                        PrintP(path3D, Task.get_P(), fltrProp.P.size(), 0);
                  PrintP(path3D, fltrProp.P, fltrProp.P.size(), iT_print);
                  CalcUsredSaturation(mesh, fltrProp, s_usred);
                  CalcUsredComp(mesh, fltrProp, s_usred);
                  Print3D_S(path3D, s_usred, iT_print);
                  PrintTimesFast(path3D, "times_main", iT_print, Filtr.Ttotal /
3600. / 24.);

                  // grapher
                  // Example
                  // x              -> [](const Point3D_t & p) { return p[0]; }
                  // norm(p)        ->  [](const  Point3D_t  &  p)  {  return
norm(p); }
                  //  sqrt(x^2+y^2)  ->  [](const  Point3D_t  &  p)  {  return
norm({p[0], p[1]}); }
                  grapher.ExportTimeSolution("_p", Filtr.Ttotal / 3600. / 24.,
fltrProp.P);
                  for (int ph = 0; ph < fltrProp.kolphase; ph++) {
                        grapher.ExportTimePiecewiseSolution("_s"  +  to_string(ph
+ 1),
```

```
                              Filtr.Ttotal / 3600. / 24. /*/ 60.*/,
                              [&](int          id_elem)          {          return
fltrProp.S0[id_elem][ph]; });
                  }

                  string sub_name = " ";
                  if (cfg.propuski_P != 1)
                        sub_name += to_string(iT_print) + " " + to_string(iT +
1);
                  else
                        sub_name += to_string(iT + 1);
                  sub_name += " " + getStringTime(Filtr.Ttotal);

                  grapher.ExportDerivateSolution(sub_name   +   "_dp",   [](const
Point3D_t & p) { return p[0]; }, fltrProp.P);
                  grapher.ExportSolution(sub_name + "_p", [](const Point3D_t &
p) { return p[0]; }, fltrProp.P);
                  for (int ph = 0; ph < fltrProp.kolphase; ph++) {
                        grapher.ExportPiecewiseSolution(sub_name    +    "_s"    +
to_string(ph + 1),
                              [](const Point3D_t & p) { return p[0]; },
                              [&](int          id_elem)          {          return
fltrProp.S0[id_elem][ph]; });
                  }
            }

            timeLogger.stop("Output");
            timeLogger.log(); // на случай предварительного завершения программы
      }
      timeLogger.stop("All time");
      timeLogger.log();
      ReturnCoutFromFile();
      printf_s("\n");
      system("pause");
      return 0;
    }
```

## FEM_compiler.h

```
#pragma once

#define FEM_compiler
#ifdef FEM_compiler

using namespace std;

#include"LoggerI.h"
#include"FEM_Structs.h"
#include"Filtr_Structs.h"
using namespace fltrStruct;
#include"FEM_utils.h"

#include"ClassMethodGauss.h"
#include"ClassMethodGaussFast.h"
#include"ClassMethodGaussFast2D.h"

#include"BaseFunc_lin.h"
#include"pardiso.h"
using namespace bfL1;
```

```
#include "TimeLogger.h"

class FEM_Pressure
{
private:
 int kolel, koluz, kolvar, kolvarall;
 static const int SizeLocMatrix = 8;
 bool fTestPolynom;
 MeshXYZ *mesh;
 FiltrProperties &prop;
 vector<Element3D> &Elements;
 vector<Point3D> &Points;
 vector<BCNodes> &bc1nodes;

 MethodGauss3D Gauss;
 MethodGauss2D Gauss2D;
 MethodGauss1D Gauss1D;

 string PathInput2D;

 MethodGaussFast_FEM &femCalculator3D;
 MethodGaussFast_FEM2D &femCalculator2D;

 bool Is_init_database;
 vector<int> ig, jg;
 vector<double> gg, ggl, di, pr, u, u_ist;

 int jsize;

 int CalcLocMatrixG(double *x, double *y, double *z, double M[][8]);
 int CalcLocMatrixM(double *x, double *y, double *z, double M[][8]);
 int CalcLocMatrixM2d(double *x, double *y, double M[][4]);

 int CalcGlobMatrix();
 void AddToGlobMatrix(int el, double M[][8], vector<double>& dest_gg,
vector<double>& dest_di);
 void AddToGlobVector(int el, double *x, vector<double>& dest);

 void MakeBC1Nodes();
 void MakeBC1NodesPolynom();
 void Build1BC();
 void Build1BCSimmetrization();
 void Build2BC();
 int CalcRigthPartWithU(double *x, double *y, double *z, double *F);
 int CalcRigthPartRm(int el, double * x, double * y, double * z, double *
F);
 int CalcLocBC2(int el, int num_face, double P, double *F);
 int CalcLocBC2Fast(int el, int num_face, double P, double *F);

 pardiso_solver prds;
 void SolveSlae(int n, const vector<int>& _ig, const vector<int>& _jg,
const vector<double>& _ggM, const vector<double>& _gglM, const vector<double>&
_diM, const vector<double>& _pr, vector<double>& res);
 void SolveSlaePardiso(int n, const vector<int>& _ig, const vector<int>&
_jg, const vector<double>& _ggM, const vector<double>& _gglM, const
vector<double>& _diM, const vector<double>& _pr, vector<double>& res);

public:
 vector<BCFaces> &bc1faces, &bc2faces;
```

```
      inline vector<double> & get_P() { return u; };
      int CalcTask0(bool gravitation, int it);
      int CalcTask0Plynom(vector<vector<double>> &_S, bool gravitation);
      int MakeBC2ByConsumption(double consumption);
      FEM_Pressure(
            MeshXYZ *_mesh,
            FiltrProperties &_prop,
            MethodGaussFast_FEM &_femCalculator3D,
            MethodGaussFast_FEM2D &_femCalculator2D
      );
      int PrintP(string & Outputpath, int iT);

      inline void initPathInput2D(const string &_PathInput2D) { PathInput2D =
_PathInput2D; }
     };

     void BuildPortret(int kolel, int kolvar, const vector<Element2D>& Elements,
vector<int>& ig, vector<int>& jg);
     void BuildPortret(int kolel, int kolvar, const vector<Element3D>& Elements,
vector<int>& ig, vector<int>& jg);

     void MultMatrixtoVect(const int dim, const int *ig, const int *jg, const
double *gg, const double *di, const double *v1, double *rez);

     int BuildPortrait(const int nc, const int krect3d, const vector<Element3D>
&Elements,const int *tig, const int *tjg, vector<int> &ig, vector<int> &jg, int
&jsize);

     #endif // FEM_compiler
```

## FEM_compiler.cpp

```
     #define _CRT_SECURE_NO_WARNINGS 1
     #include "FEM_compiler.h"
     #include "PortraitAL.h"
     #include "ArrayOf.h"

     double func(double x, double y, double z)
     {
      return x*x;
     }

     double funcpr(double x, double y, double z)
     {
      return -2 + x*x;
     }

     int FEM_Pressure::CalcLocMatrixG(double *x, double *y, double *z, double
G[][8])
     {
      ScalFunc3D f;
      for (int i = 0; i < SizeLocMatrix; i++)
            for (int j = 0; j <= i; j++)
            {
                  f = [i, j, x, y, z](double ksi, double eta, double zeta)
                  {
                        double J_1_T[3][3], det_J;
```

```
                        double grad1[3], grad2[3], Mg1[3], Mg2[3];
                        Jacobian3D(ksi, eta, zeta, x, y, z, J_1_T, det_J);
                        grad1[0] = d_phi3d(i, ksi, eta, zeta, 1);
                        grad1[1] = d_phi3d(i, ksi, eta, zeta, 2);
                        grad1[2] = d_phi3d(i, ksi, eta, zeta, 3);
                        grad2[0] = d_phi3d(j, ksi, eta, zeta, 1);
                        grad2[1] = d_phi3d(j, ksi, eta, zeta, 2);
                        grad2[2] = d_phi3d(j, ksi, eta, zeta, 3);
                        MultMatrixtoVec(J_1_T, grad1, Mg1);
                        MultMatrixtoVec(J_1_T, grad2, Mg2);
                        return (Mg1[0] * Mg2[0] + Mg1[1] * Mg2[1] + Mg1[2] *
Mg2[2]) * abs(det_J);
                  };
                  G[j][i] = G[i][j] = Gauss.Integration(f);
            }
      return 0;
     }

     int FEM_Pressure::CalcLocMatrixM(double * x, double * y, double * z, double
M[][8])
     {
      ScalFunc3D f;
      for (int i = 0; i < SizeLocMatrix; i++)
            for (int j = 0; j <= i; j++)
            {
                  f = [i, j, x, y, z](double ksi, double eta, double zeta)
                  {
                        double J_1_T[3][3], det_J;
                        Jacobian3D(ksi, eta, zeta, x, y, z, J_1_T, det_J);
                        return phi3d(i, ksi, eta, zeta) * phi3d(j, ksi, eta,
zeta) * abs(det_J);
                  };
                  M[j][i] = M[i][j] = Gauss.Integration(f);
            }
      return 0;
     }

     int FEM_Pressure::CalcLocMatrixM2d(double * x, double * y, double M[][4])
     {
      ScalFunc2D f;
      for (int i = 0; i < 4; i++)
            for (int j = 0; j <= i; j++)
            {
                  f = [i, j, x, y](double ksi, double eta)
                  {
                        double J_1_T[2][2], det_J;
                        Jacobian(ksi, eta, x, y, J_1_T, det_J);
                        return phi2d(i, ksi, eta) * phi2d(j, ksi, eta) *
abs(det_J);
                  };
                  M[j][i] = M[i][j] = Gauss2D.Integration(f);
            }
      return 0;
     }

     int FEM_Pressure::CalcRigthPartWithU(double * x, double * y, double * z,
double * F)
     {
      ScalFunc3D f;
      for (int i = 0; i < SizeLocMatrix; i++)
```

```
	{
		f = [i, x, y, z](double ksi, double eta, double zeta)
		{
			double J_1[3][3], det_J, xcrd(0.0), ycrd(0.0), zcrd(0.0);
			Jacobian3D(ksi, eta, zeta, x, y, z, J_1, det_J);
			for (int k = 0; k < 8; k++)
			{
				xcrd += x[k] * phi3d(k, ksi, eta, zeta);
				ycrd += y[k] * phi3d(k, ksi, eta, zeta);
				zcrd += z[k] * phi3d(k, ksi, eta, zeta);
			}
			return phi3d(i, ksi, eta, zeta) * funcpr(xcrd, ycrd, zcrd) *
abs(det_J);
		};
		F[i] = Gauss.Integration(f);
	}
	return 0;
}

int FEM_Pressure::CalcLocBC2Fast(int el, int num_face, double P, double *
F)
{
	double M2d[4][4];
	double shx[4], shy[4], shz[4]; // координаты грани по шаблону
	mesh->InitShapeGran(Elements[el].faces[num_face], shx, shy, shz);
	femCalculator2D.Calc_M(shx, shy, shz, M2d);


	for (int i = 0; i < 4; i++)
	{
		F[i] = 0.0;
		for (int j = 0; j < 4; j++)
		{
			F[i] += M2d[i][j];
		}
		F[i] *= P; // давление на границе равномерное
	}
	return 0;
}


int FEM_Pressure::CalcLocBC2(int el, int num_face, double P, double * F)
{
	double M2d[4][4];
	double shx[4], shy[4]; // координаты грани по шаблону
	int nd, lcrd[2];
	switch (num_face)
	{
	case 0:
	case 1:
		lcrd[0] = 1;
		lcrd[1] = 2;
		break;
	case 2:
	case 3:
		lcrd[0] = 0;
		lcrd[1] = 2;
		break;
	case 4:
	case 5:
```

```
            lcrd[0] = 0;
            lcrd[1] = 1;
            break;
      }
      for (int i = 0; i < 4; i++)
      {
            nd = Elements[el].node[Element3D::faces_node[num_face][i]];
            shx[i] = Points[nd].crd[lcrd[0]];
            shy[i] = Points[nd].crd[lcrd[1]];
      }
      CalcLocMatrixM2d(shx, shy, M2d);

      for (int i = 0; i < 4; i++)
      {
            F[i] = 0.0;
            for (int j = 0; j < 4; j++)
                  F[i] += M2d[i][j];
            F[i] *= P; // давление на границе равномерное
      }
      return 0;
     }

     int FEM_Pressure::CalcTask0(bool gravitation, int iT)
     {
      auto& timeLogger = TimeLogger::instance();
      timeLogger.start("Pressure task");

      if (!Is_init_database)
      {
            MakeBC1Nodes();
            mesh->nc = kolvar - mesh->tc;
            kolvar = mesh->nc;
            kolvarall = mesh->nc + mesh->tc;
            cout << "mesh->nc= " << mesh->nc << endl;
            cout << "mesh->tc= " << mesh->tc << endl;
            cout << "kolvar= " << kolvar << endl;
            cout << "kolvarall= " << kolvarall << endl;
            BuildPortrait(kolvar, kolel, Elements, mesh->tig, mesh->tjg, ig, jg,
jsize);
            prds.prepare(kolvar, ig.data(), jg.data());
            gg.resize(ig[kolvar], 0);
            gg.reserve(ig[kolvar]);
            di.resize(kolvar, 0);
            di.reserve(kolvar);
            pr.resize(kolvar, 0);
            pr.reserve(kolvar);
            u.resize(kolvarall, 0);
            u.reserve(kolvarall);
            Is_init_database = true;
      }
      else
      {
            for (int i = 0; i < ig[kolvar]; i++)
                  gg[i] = 0.0;
            for (int i = 0; i < kolvar; i++)
                  di[i] = pr[i] = u[i] = 0.0;
      }
      CalcGlobMatrix();
      Build2BC();
      Build1BCSimmetrization();
```

```
	cout << "SolveSlaePardiso" << endl;
	SolveSlaePardiso(kolvar, ig, jg, gg, gg, di, pr, u);
	timeLogger.stop("Pressure task");
	for (int m = kolvar; m<kolvarall; m++)
	{
		int i = m - kolvar;
		u[m] = 0.0;
		for (int j = mesh->tig[i] - 1; j<mesh->tig[i + 1] - 1; j++)
		{
			u[m] += mesh->tgg[j] * u[mesh->tjg[j] - 1];
		}
	}
	return 0;
}

int FEM_Pressure::CalcGlobMatrix()
{
	double x[8], y[8], z[8], M[8][8], F[8], koef;
	for (int el = 0; el < kolel; el++)
	{
		koef = 0.0;
		mesh->InitShapeElem(el, x, y, z);
		femCalculator3D.Calc_Matrix_G(x, y, z, M);
		for (int p = 0; p < prop.kolphase; p++)
			koef += prop.PhasesElem0[el][p].k /
prop.PhasesElem0[el][p].eta;
		koef *= prop.Mat[Elements[el].nmat].K;
		if (fTestPolynom)
			koef = 1.;

		for (int i = 0; i < SizeLocMatrix; i++)
			for (int j = 0; j < SizeLocMatrix; j++)
				M[i][j] *= koef;
		AddToGlobMatrix(el, M, gg, di);
		if (fTestPolynom)
		{
			//для аналитического теста
			CalcLocMatrixM(x, y, z, M);
			AddToGlobMatrix(el, M, gg, di);

			CalcRigthPartWithU(x, y, z, F);
			AddToGlobVector(el, F, pr);
		}
	}
	cout << endl;
	return 0;
}

FEM_Pressure::FEM_Pressure(
	MeshXYZ *_mesh,
	FiltrProperties &_prop,
	MethodGaussFast_FEM &_femCalculator3D,
	MethodGaussFast_FEM2D &_femCalculator2D
)
	:
	prop(_prop),
	femCalculator3D (_femCalculator3D),
	femCalculator2D(_femCalculator2D),
	Elements(_mesh->Elements),
	Points(_mesh->Points),
```

```
 bc1nodes(_prop.bc1nodes),
 bc1faces(_prop.bc1faces),
 bc2faces(_prop.bc2faces)
{
 kolel = _mesh->kolel;
 koluz = _mesh->koluz;
 kolvar = koluz;
 mesh = _mesh;
 Is_init_database = false;
 fTestPolynom = false;
}

int FEM_Pressure::PrintP(string & Outputpath, int iT)
{
 ofstream ofp;
 ofp.open(Outputpath + "\\ut." + to_string(iT) + ".0", ios::binary);
 ofp << setprecision(15);
 for (int i = 0; i < kolvarall; i++)
       ofp < u[i];
 ofp.clear();
 ofp.close();
 return 0;
}

int BuildPortrait(const int nc, const int krect3d, const vector<Element3D>
&Elements,const int *tig, const int *tjg, vector<int> &ig, vector<int> &jg, int
&jsize)
{
 int i, j, k, retc, node, nextnode;
 PortraitAL P(nc);
 ResizableArrayOf<int> indlist(64);
 int *iptr, *jptr;
 iptr = jptr = NULL;
 for (i = 0; i<krect3d; i++)
 {
       const Element3D& r = Elements[i];
       indlist.SetSize(0);
       for (j = 0; j<8; j++)
       {
             node = r.bf[j] + 1;
             if (node <= nc)
             {
                   if (!indlist.Find(node))
                         if (indlist.Add(node) == -1)
                               return RETCODE_OUTOFRANGE;
             }
             else
             {
                   node -= nc;
                   for (k = tig[node - 1] - 1; k<tig[node] - 1; k++)
                   {
                         nextnode = tjg[k];
                         if (!indlist.Find(nextnode))
                               if (indlist.Add(nextnode) == -1)
                                     return RETCODE_OUTOFRANGE;
                   }
             }
       }
       P.AddToPortrait(indlist.GetSize(), indlist.val);
 }
```

```
      if ((retc = P.BuildPortrait(iptr, jptr, jsize)) != 0)
            return retc;

      // На выходе нумерация с 0

      ig.resize(nc + 1);
      jg.resize(jsize);

      for (i = 0; i<nc; i++) { ig[i]=iptr[i]-1; }
      ig[i]=iptr[i]-1;

      for (i = 0; i<jsize; i++) { jg[i]=jptr[i]-1; }

      if (iptr) {delete[] iptr; iptr = NULL;}
      if (jptr) {delete[] jptr; jptr = NULL; }

      return RETCODE_OK;
     }

     int BuildPortraitFaces(const int nc, const int krect3d, const
vector<Element3D> &Elements, const int *tig, const int *tjg, vector<int> &ig,
vector<int> &jg, int &jsize)
     {
      int i, j, k, retc, node, nextnode;
      PortraitAL P(nc);
      ResizableArrayOf<int> indlist(64);
      int *iptr, *jptr;
      iptr = jptr = NULL;
      for (i = 0; i<krect3d; i++)
      {
            const Element3D& r = Elements[i];
            indlist.SetSize(0);
            for(j=0;j<6;j++)
            {
                  node = r.faces[j] + 1;
                  if (node <= nc)
                  {
                        if (!indlist.Find(node))
                              if (indlist.Add(node) == -1)
                                    return RETCODE_OUTOFRANGE;
                  }
                  else
                  {
                        node -= nc;
                        for (k = tig[node - 1] - 1; k<tig[node] - 1; k++)
                        {
                              nextnode = tjg[k];
                              if (!indlist.Find(nextnode))
                                    if (indlist.Add(nextnode) == -1)
                                          return RETCODE_OUTOFRANGE;
                        }
                  }
            }
            P.AddToPortrait(indlist.GetSize(), indlist.val);
      }

      if ((retc = P.BuildPortrait(iptr, jptr, jsize)) != 0)
            return retc;
```

```
      // На выходе нумерация с 0

      //ig=new int[nc + 1];
      //jg=new int[jsize];

      ig.resize(nc + 1);
      jg.resize(jsize);

      for (i = 0; i<nc; i++) { ig[i] = iptr[i] - 1; }
      ig[i] = iptr[i] - 1;

      for (i = 0; i<jsize; i++) { jg[i] = jptr[i] - 1; }

      if (iptr) { delete[] iptr; iptr = NULL; }
      if (jptr) { delete[] jptr; jptr = NULL; }

      return RETCODE_OK;
     }

     void AddToMatrix(const int &gnu_i, const int &gnu_j, const double &val,
vector<int> &ig, vector<int> &jg, vector<double>&di, vector<double>&gg)
     {
      int k;
      if (gnu_i == gnu_j)
      {
            di[gnu_i] += val;
      }
      else
      {
            if (gnu_i>gnu_j)
            {
                  for (k = ig[gnu_i]; jg[k]<gnu_j; k++);
                  gg[k] += val;
            }
      }
     }

     // В ig, jg нумерация с 0
     // В tig, tjg нумерация с 1
     void AddToMatrix(const int &i, const int &j, const double &val,vector<int>
&ig, vector<int> &jg, vector<double>&di, vector<double>&gg,const int nc, const
int *tig, const int *tjg, const double *tgg)
     {
      int _i, _j, _ti, _tj, c, d;
      _i = i + 1;
      _j = j + 1;
      if (_i>nc && _j <= nc)
      {
            _i -= nc;
            for (c = tig[_i - 1] - 1; c<tig[_i] - 1; c++)
            {
                  _ti = tjg[c];
                  AddToMatrix(_ti - 1, _j - 1, tgg[c] * val, ig, jg, di, gg);
            }
      }
      else if (_i <= nc && _j>nc)
      {
            _j -= nc;
            for (c = tig[_j - 1] - 1; c<tig[_j] - 1; c++)
            {
```

```
                _tj = tjg[c];
                AddToMatrix(_i - 1, _tj - 1, tgg[c] * val, ig, jg, di, gg);
            }
        }
        else if (_i>nc && _j>nc)
        {
            _i -= nc;
            _j -= nc;
            for (c = tig[_i - 1] - 1; c<tig[_i] - 1; c++)
            {
                _ti = tjg[c];
                for (d = tig[_j - 1] - 1; d<tig[_j] - 1; d++)
                {
                    _tj = tjg[d];
                    AddToMatrix(_ti - 1, _tj - 1, tgg[c] * tgg[d] * val, ig,
jg, di, gg);
                }
            }
        }
        else
        {
            AddToMatrix(_i - 1, _j - 1, val, ig, jg, di, gg);
        }
    }

    void FEM_Pressure::AddToGlobMatrix(int el, double M[][8], vector<double>&
dest_gg, vector<double>& dest_di)
    {
        for (int i = 0; i < Elements[el].bfkol; i++)
        {
            int ii = Elements[el].bf[i];
            for (int j = 0; j < Elements[el].bfkol; j++)
            {
                int jj = Elements[el].bf[j];
                AddToMatrix(ii, jj, M[i][j], ig, jg, dest_di, dest_gg, kolvar,
mesh->tig, mesh->tjg, mesh->tgg);
            }
        }
    }

    void AddToVector(const int &i, const double &val, vector<double> &pr,const
int nc, const int *tig, const int *tjg, const double *tgg)
                {
        int _i, _ti, c;
        _i = i + 1;
        if (_i <= nc)
                    {
            pr[_i - 1] += val;
                }
        else
        {
            _i -= nc;
            for (c = tig[_i - 1] - 1; c<tig[_i] - 1; c++)
            {
                _ti = tjg[c];
                pr[_ti - 1] += tgg[c] * val;
            }
        }
    }
```

```
void FEM_Pressure::AddToGlobVector(int el, double *x, vector<double>& dest)
{
 for (int i = 0; i < Elements[el].bfkol; i++)
 {
       AddToVector(Elements[el].bf[i], x[i], dest, kolvar, mesh->tig, mesh-
>tjg, mesh->tgg);
 }
}

void FEM_Pressure::MakeBC1Nodes()
{
 BCNodes node;
 int nel, nfc;
 for (int bc = 0; bc < prop.kolbc1faces; bc++)
 {
       node.P = bc1faces[bc].dP;
       nel = bc1faces[bc].el;
       nfc = bc1faces[bc].face;
       for (int nd = 0; nd < 4; nd++)
       {
             node.node = Elements[nel].bf[Element3D::faces_node[nfc][nd]];
             bc1nodes.push_back(node);
       }
 }
}

void FEM_Pressure::MakeBC1NodesPolynom()
{
 for (auto & u : bc1nodes)
       u.P     =     func(Points[u.node].crd[0],     Points[u.node].crd[1],
Points[u.node].crd[2]);
}

void FEM_Pressure::Build1BC()
{
 double ans;
 int ind;
 double BIG = 1e+0;
 ggl = gg;
 for (const auto & u : bc1nodes)
 {
       ans = u.P;
       ind = u.node;
       di[ind] = BIG;
       pr[ind] = BIG*ans;
       int i0 = ig[ind];
       int i1 = ig[ind + 1];
       for (int k = i0; k < i1; k++)
             ggl[k] = 0;
       for (int j = 1; j < kolvar; j++)
       {
             int j0 = ig[j];
             int j1 = ig[j + 1];
             for (int i = j0; i < j1; i++)
                   if (jg[i] == ind)
                         gg[i] = 0;
       }
 }
}
```

```
    void FEM_Pressure::Build1BCSimmetrization()
    {
     vector<int> Fku(kolvar, -1);
     for (int i = 0; i < bc1nodes.size(); i++)
           Fku[bc1nodes[i].node] = i;
     int k;
     for (int i = 0; i < kolvar; i++)
     {
           if (Fku[i] != -1)
           {
                 di[i] = 1.0;
                 pr[i] = bc1nodes[Fku[i]].P;
                 for (int j = ig[i]; j < ig[i + 1]; j++)
                 {
                       k = jg[j];
                       if (Fku[k] == -1) { pr[k] -= gg[j] * pr[i]; }
                       gg[j] = 0.0;
                 }
           }
           else
           {
                 for (int j = ig[i]; j < ig[i + 1]; j++)
                 {
                       k = jg[j];
                       if (Fku[k] != -1)
                       {
                             pr[i] -= gg[j] * pr[k];
                             gg[j] = 0.0;
                       }
                 }
           }
     }
    }

    void FEM_Pressure::Build2BC()
    {
     double F[4];
     int nd;
     for (const auto &u : bc2faces)
     {
           CalcLocBC2Fast(u.el, u.face, u.dP, F);

           for (int i = 0; i < 4; i++)
           {
                 nd = Elements[u.el].node[Element3D::faces_node[u.face][i]];
                 AddToVector(nd,  F[i],  pr,  mesh->nc,  mesh->tig,  mesh->tjg,
mesh->tgg);
           }
     }
    }

    void FEM_Pressure::SolveSlaePardiso(int n, const vector<int>& _ig, const
vector<int>& _jg, const vector<double>& _ggM, const vector<double>& _gglM, const
vector<double>& _diM, const vector<double>& _pr, vector<double>& res)
    {
     cout << "> factorize" << endl;
     prds.factorize(n,   (int   *)_ig.data(),   (int   *)_jg.data(),   (double
*)_ggM.data(), (double *)_diM.data(), 8);
     cout << "> solve" << endl;
     prds.solve_nrhs(1, (double *)_pr.data(), res.data());
```

```
 prds.stop_solver();
}

void MultMatrixtoVect(const int dim, const int *ig, const int *jg, const
double *gg, const double *di, const double *v1, double *rez)
{
 for (int i = 0; i < dim; i++)
 {
       int i0 = ig[i];
       int i1 = ig[i + 1];
       rez[i] = di[i] * v1[i];
       for (int k = i0; k < i1; k++)
       {
             int j = jg[k];
             rez[i] += gg[k] * v1[j];
             rez[j] += gg[k] * v1[i];
       }
 }
 return;
}
```

## Filtration.h

```
#pragma once
#define _CRT_SECURE_NO_WARNINGS 1

#define Filtration
#ifdef Filtration

#include"Mesh.h"
#include"stdafx.h"
#include"filtr_cfg.h"
#include"FEM_Structs.h"
#include"Filtr_Structs.h"
using namespace fltrStruct;
#include"ClassMethodGauss.h"
#include"ClassMethodGaussFast2D.h"
#include"ClassMethodGaussFast.h"
#include"BaseFunc_lin.h"
#include"LOS.h"
#include"Matrix_CSIR.h"
#include"pardiso.h"
#include"rule.h"

#include "TimeLogger.h"

#define PI 3.1415926535897932
#define Gconst 10.0
#define eps_flux 1e-8
#define eps_zero_balance_flux 1e-3
#define eps_not_balance_flux_for_check 1e-8
#define eps_reg_align 1e-12
#define kij 1
#define eps_1_in_crit_filtr 1e-7
#define eps_zero 1e-9
#define eps_expulsion 0.8

using namespace bfL1;
```

```
class FiltrationClass
{
private:
 MeshXYZ *mesh;
 vector<Element3D> &Elements;
 vector<Point3D> &Points;
 FiltrProperties &prop;

 bool    is_init_flux0,    is_init_fluxMixture,    is_init_volumeOutPhase,
is_init_fluxOutPhase, is_init_volumePhase;
 unordered_set<int> faces_bc, faces_free;
 vector<double> flMix;
 vector<short int> flNulls; // по номеру граней
 vector<vector<double>>  volumeOutPhase,  volumeOutPhasePre,  volumePhase,
volumePhasePre;
 vector<vector<double>> fluxOutPhase;
 bool use_aligment = true;
 vector<bool> used_fc;
 //-------------ALIGMENT-------------------------------
 bool use_aligment_slae = true; // слау или дерево
 bool init_fatorize = false;
 MatrixCSIR A_align;
 vector<double> b_align, x_align;
 vector<int> ibc_align;
 vector<double> nonbalance_elem;
 vector<double> nonbalance_node;
 vector<double> alpha_abs_buf; //
 vector<double> alpha_buf;
 vector<double> x_align_buf;
 void WriteNebalance(vector<double> & times, vector<double> & nebalance,
Config &cfg);
 vector<double> times_nonb;
 void PrintElemFlow(int ielem);
 void PrintElemFlowAlignment(int ielem);
 vector<double> alpha, alpha_abs;
 vector<double> beta;
 bool GoodChange(double flow_old, double flow_new, double max_abs_flow); //
 int CalcUsredSaturation(vector<double>& for_elem, vector<double>& for_uz);
 double MaxAbsFlow();
 double NonbalanceElem(int ielem);
 double NonbalanceMax();
 double NonbalanceMax(int & ind);
 double NonbalanceAll();
 double AlignmentMaxAbsFlow();
 double AlignmentNonbalanceElem(int ielem);
 double AlignmentNonbalanceMax();
 double AlignmentNonbalanceMax(int & ind);
 double AlignmentNonbalanceAll();
 void AlignmentFluxNew(vector<double> & FL, bool refactorize /*= true*/);

 //-------------------------------------------------
 MethodGauss2D Gauss2D;
 MethodGaussFast_FEM &femCalculator3D;
 MethodGaussFast_FEM2D &femCalculator2D;
 pardiso_solver prds;

 size_t kolel, koluz, kolvar, kolfc;
 size_t kolphase;
public:
 vector<BCFaces> &bc2faces, &bc1faces;
```

```
	vector<pair<double, double>> balance;

	double Ttotal;
	vector<double> flVbc2, flVbc1;
	vector<vector<double>> holesVolumePhasesTotal;
	vector<double> holesVolumeTotal;
	vector<rule> rules;
	FiltrProperties & GetProp();
	//-------------FLUX-----------------------------------
	void CalcFlux0UsredMixture(); // ! усреднение не gradP, а потока смеси
	int CheckDirectFlux();
	void NullerSmallflMix();
	void NullerCounter();
	int fluxDirect(int el, int fc);
	//-------------CALC_FILTRATION------------------------
	void CALC_crit(Config &cfg, int iT, double &dt);
	void CALC_test(Config & cfg, int iT, double & dt);
	void CalcFluxOutPhase();
	void CalcVolumePhase4(double dt);
	bool CalcdtCrit(double dt_start, double & dt_ret);
	void FiltrationV_Crit(double dt);
	void FiltrationV_Crit2(double dt);
	//-------------ALIGMENT-------------------------------
	bool align_refactorize = true;
	int max_iter_aligment = 10;
	int max_alpha_iter = 3;
	int max_beta_iter = 20;
	int max_alpha_cancel = 2;
	int first_accuracy_iter = 10;
	double ratio_sign_change = 1e-3;
	double ratio_bad_sign_change = 1e-2;
	double max_beta = 1e+30;
	double max_beta1 = 1e+15; // 1e+15
	double dump_beta = 10;
	double dump_alpha = 10;
	double max_ratio_func_ab = 2;
	double beta0 = 1;
	double alpha0 = 1;
	void AlignmentFluxMainNew(vector<double> & FL, int iter, Config &cfg);
	double CalcCounter(vector<double>& FL);
	void AlignmentFluxIO(vector<double> & FL);
	void MakeTrueFluxOnBC2(vector<double> & FL);
	void RenumerateFaces(vector<int> &FaceRenum);
	void BuildTmatrixGran();
	void PrintBalanceTable(const std::string & path);
	void PrintFluxIO(vector<double>& FL);
	void AddToGlobVector(int el, double *x, vector<double>& dest);
	int InitShapeElem(int el, double * x, double * y, double * z);
	int InitShapeGran(int el, int face, double * x, double * y, double * z);
	void SolveSlaePardiso(int n, const vector<int>& _ig, const vector<int>&
_jg, const vector<double>& _ggM, const vector<double>& _gglM, const
vector<double>& _diM, const vector<double>& _pr, vector<double>& res);
	FiltrationClass(
		MeshXYZ *_mesh,
		FiltrProperties &_prop,
		MethodGaussFast_FEM & _femCalculator3D,
		MethodGaussFast_FEM2D & _femCalculator2D
	);
	int InitSbcCommon_interval_bc1faces();
	int WriteMixtureFlux(string & pathInput);
```

```
 int ReadMixtureFlux(string & pathInput);
 int CheckBalance(vector<double> &FL);
 void FindFreeBCCounter();
 int PrintFluxV_IO_crit(string & pathInput, double t, int it);
 int PrintFluxV_IO_crit_hole(string & pathInput, double t, int it);
 int PrintFluxV_IO_prep(string & pathInput);
 int PrintFlux0Table(string & pathInput);
 int PrintS(string & pathInput, vector<vector<double>> &_S, int it);
 int PrintString(string & pathInput, string str, int it);
 int PrintMatandPhaseprop(string & pathInput);
 int PrintBetaNebalans(string & pathOutput, int iT);
 int WriteTotalVolumes(string & pathOutput);
 int ReadTotalVolumes(string & pathInput);
 double GetMassPhaseCompIn(int el, int iphs, int icmp);
 bool Mixing2(double tau);
 void CalcNewS();
 void CalcNewV2();
 bool fcont;
};

void FixUnknownsWithSimmetrization(int n, vector<int> &Fku, vector<int>
&ig, vector<int> &jg, vector<double> &di, vector<double> &ggl, vector<double>
&pr);

static void AddToGlobMatrix2(int shapes[6], double M[][6], const
vector<int> & ig, const vector<int> & jg, vector<double>& gg, vector<double>&
di);
static void AddToGlobVector2(int shapes[6], double *x, vector<double>&
dest);

#endif // Filtration
```

## Filtration.cpp

```
#include "Filtration.h"
#include "ElemNeib.h"

int sign(const double & fl)
{
 if (fl == 0)
       return 0;
 if (fl > 0)
       return 1;
 else
       return -1;
}

void FiltrationClass::FindFreeBCCounter()
{
 int kolbc, inded, neig;
 kolbc = bc1faces.size();
 for (int i = 0; i < kolbc; i++)
 {
       inded = Elements[bc1faces[i].el].faces[bc1faces[i].face];
       faces_bc.insert(inded);
 }
 kolbc = bc2faces.size();
 for (int i = 0; i < kolbc; i++)
```

```
 {
       inded = Elements[bc2faces[i].el].faces[bc2faces[i].face];
       faces_bc.insert(inded);
 }
 for (int el = 0; el < kolel; el++)
 {
       for (int fc = 0; fc < 6; fc++)
       {
             //neig = mesh->get_neighbor(el, fc);
             int n_neib = (int)mesh->ElemNeibVec[el].neib[fc].size();

             //if (neig == -1)
             if (!n_neib)
             {
                   inded = Elements[el].faces[fc];
                   if (faces_bc.count(inded) == 0)
                   {
                         faces_free.insert(inded);
                   }
             }
       }
 }
}

void FiltrationClass::CalcFlux0UsredMixture()
{
 static int iter = 0;
 int inded, indfc;
 double x[8], y[8], z[8], p[8], fl, fl_fast, k;
 double shx[4], shy[4], shz[4];
 vector<int> used_fc(kolfc, -1);
 if (!is_init_flux0)
 {
       flMix.reserve(kolfc);
       flMix.resize(kolfc);
       flNulls.resize(kolfc, 1);
       FindFreeBCCounter();
       is_init_flux0 = true;
 }
 else
 {
       for (int i = 0; i < kolfc; i++)
       {
             flMix[i] = 0.0;
             flNulls[i] = 1;
       }
 }
 for (int el = 0; el < kolel; el++)
 {
       InitShapeElem(el, x, y, z);
       for (int i = 0; i < 8; i++)
             p[i] = prop.P[Elements[el].bf[i]];

       for (int fc = 0; fc < 6; fc++)
       {
             indfc = Elements[el].faces[fc];
             mesh->InitShapeGran(indfc, shx, shy, shz);

             // Поток течёт в направлении антиградиента
```

```
                fl = -femCalculator2D.Calc_Flux(x, y, z, shx, shy, shz, p,
fc);
                fl *= Elements[el].faces_direct[fc]; // fl - со знаком в
соответствии с ориентацией грани
                {
                      k = 0.0;
                      for (int i = 0; i < kolphase; i++)
                            k       +=       prop.PhasesElem0[el][i].k       /
prop.PhasesElem0[el][i].eta;
                      k *= prop.Mat[Elements[el].nmat].K;
                      fl *= k;
                      if (used_fc[indfc] != -1)
                      {
                            if (sign(fl) != sign(flMix[indfc])) // если лежат
с разным знаком
                            {
                                  flMix[indfc] += fl;
                            }
                            else
                            {
                                  // вариант с некоторой пропорцией w
                                  //double w = 0.5;
                                  //flMix[indfc] = (1 - w) * flMix[indfc] + w
* fl;
                                  fl += flMix[indfc];
                                  fl /= 2.;
                                  flMix[indfc] = fl;
                            }
                      }
                      else
                      {
                            flMix[indfc] = fl;
                            used_fc[indfc] = el;
                      }
                }
          }
     }
     cout << endl;
     iter++;
    }

    void FiltrationClass::CalcFluxOutPhase()
    {
     vector<double> alfa(kolphase);
     bool ferr = false;
     int errfc = -1;
     double alfa_sr;
     int indfc;
     if (!is_init_fluxOutPhase)
     {
          fluxOutPhase.resize(kolfc, vector<double>(kolphase, 0.));
          used_fc.resize(kolfc, false);
          is_init_fluxOutPhase = true;
     }
     else
     {
          for (int i = 0; i < kolfc; i++)
               used_fc[i] = false;
     }
     for (int el = 0; el < kolel; el++)
```

```
	{
		alfa_sr = 0;
		auto & prop_el = prop.PhasesElem0[el];
		for (int ph = 0; ph < kolphase; ph++)
			alfa_sr += prop_el[ph].k / prop_el[ph].eta;
		for (int ph = 0; ph < kolphase; ph++)
			alfa[ph] = prop_el[ph].k / (prop_el[ph].eta * alfa_sr);
		for (int fc = 0; fc < 6; fc++)
			if (fluxDirect(el, fc) == 1)
			{
				indfc = Elements[el].faces[fc];
				if (!used_fc[indfc])
					used_fc[indfc] = true;
				else
				{
					ferr = true;
					errfc = indfc;
				}
				for (int ph = 0; ph < kolphase; ph++)
					fluxOutPhase[indfc][ph] = alfa[ph] *
abs(flMix[indfc]);
			}
	}
	if (ferr)
	{
		cout << "CalcFluxOutPhase : ERROR any times calcs flux phase for
face = " << errfc << endl;
	}
}

void FiltrationClass::CalcVolumePhase4(double dt)
{
	vector<double> alfa(kolphase), Vcrit(kolphase);
	array<double, 6> alfa_crit; // по количеству граней
	double alfa_sr, alfa_sr_new, alfa_sr_for_crit, Qmix_new;
	double Vmix_new, Vabandom;
	bool fTakeAway, fTakeAwayGlob;
	int ind_glob_fc;

// Изъято из публичного доступа

}

int FiltrationClass::fluxDirect(int el, int fc)
{
	int indfc = Elements[el].faces[fc];
	if (flNulls[indfc] == 0)
		return 0;
	if (flMix[indfc] > 0)
		return Elements[el].faces_direct[fc];
	else
		return -Elements[el].faces_direct[fc];
}

static void AddToGlobMatrix2(int shapes[6], double M[][6], const
vector<int> & ig, const vector<int> & jg, vector<double>& gg, vector<double>&
di)
{
	for (int i = 0; i < 6; i++) {
		di[shapes[i]] += M[i][i];
```

```
            int i0 = ig[shapes[i]];
            int i1 = ig[shapes[i] + 1];
            for (int j = 0; j < 6; j++) {
                  int jj = shapes[j];
                  for (int k = i0; k < i1; k++) {
                        if (jg[k] == jj) {
                              gg[k] += M[i][j];
                              break;
                        }
                  }
            }
      }
     }

     static void AddToGlobVector2(int shapes[6], double *x, vector<double>&
dest)
     {
      for (int i = 0; i < 6; i++) {
            dest[shapes[i]] += x[i];
      }
     }

     int      FiltrationClass::CalcUsredSaturation(vector<double>      &for_elem,
vector<double> &for_uz)
     {
      double sum;
      int kol;
      int elem;
      for_uz.resize(koluz);
      for (int uz = 0; uz < koluz; uz++)
            for_uz[uz] = 0.;
      for (int uz = 0; uz < koluz; uz++)
      {
            kol = mesh->node2elem[uz].size();
            sum = 0.0;
            for (int el = 0; el < kol; el++)
            {
                  elem = mesh->node2elem[uz][el];
                  for_uz[uz] += for_elem[elem] * mesh->amountElem[elem];
                  sum += mesh->amountElem[elem];
            }
            for_uz[uz] /= sum;
      }
      return 0;
     }

     void FiltrationClass::WriteNebalance(vector<double> & times, vector<double>
& nebalance, Config & cfg) {
      ofstream ouf;
      string path = cfg.PathOutput + "\\" + "nebalance3D" + "\\" + "times_main";
      ouf.open(path);
      ouf << times.size() << endl;
      for (int i = 0; i < times.size(); i++)
            ouf << times[i] << endl;
      ouf.clear();
      ouf.close();

      path   =   cfg.PathOutput   +   "\\"   +   "nebalance3D"   +   "\\sx."   +
to_string(times.size());
      ofstream ofp3(path, ios::binary);
```

```
      for (int i = 0; i < nebalance.size(); i++) {
            ofp3 < nebalance[i];
      }
      ofp3.close();
      ofp3.clear();
     }

     void    FiltrationClass::AlignmentFluxNew(vector<double>    &    FL,    bool
refactorize)
     {
      auto& timeLogger = TimeLogger::instance();
      auto & ig = A_align.ig;
      auto & jg = A_align.jg;
      auto & di = A_align.di;
      auto & gg = A_align.ggl;
      auto & b = b_align;
      auto & x = x_align;
      auto & ibc = ibc_align;

      if (!A_align.is_init_portrait) {
            BuildPortretbyFace(kolel, kolfc, Elements, ig, jg);
            A_align.is_init_portrait = true;
            A_align.InitDataMemory(kolfc, true);
            prds.prepare(kolfc, ig.data(), jg.data());
      }
      A_align.SetZeroData();
      b.clear();
      b.resize(kolfc, 0);

      // Сборка матрицы СЛАУ
      double M[6][6];
      for (int ielem = 0; ielem < kolel; ++ielem) {
            auto & elem_faces = Elements[ielem].faces;
            auto & elem_flow_direct = Elements[ielem].faces_direct;
            for (int iface = 0; iface < 6; ++iface) {
                  int main_direction = elem_flow_direct[iface];
                  for (int jface = 0; jface < 6; ++jface) {
                        M[iface][jface]        =        main_direction        *
elem_flow_direct[jface] * beta[ielem];
                  }
            }
            AddToGlobMatrix2(elem_faces, M, ig, jg, gg, di);
      }
      // Регуляризация
      for (int i = 0; i < kolfc; ++i) {
            if (flNulls[i] != 0) {
                  di[i] += alpha[i];
            }
      }


     // Изъято из публичного доступа

      cout << "FLcounter befor: " << FLcounter1 << " FLcounter after: " <<
FLcounter2 << endl;
     }
     void FiltrationClass::PrintElemFlow(int ielem) {
      for (int i = 0; i < 6; i++) {
            cerr << Elements[ielem].faces[i] << "  "
```

```
                  <<              flMix[Elements[ielem].faces[i]]              *
Elements[ielem].faces_direct[i]
                  << std::endl;
      }
     }

     void FiltrationClass::PrintElemFlowAlignment(int ielem) {
      for (int i = 0; i < 6; i++) {
            int iface = Elements[ielem].faces[i];
            int direct = Elements[ielem].faces_direct[i];
            cerr << Elements[ielem].faces[i] << "  "
                  << (flMix[iface] + x_align[iface]) * direct
                  << std::endl;
      }
     }

     void FiltrationClass::CALC_crit(Config & cfg, int iT, double &dt)
     {
      auto& timeLogger = TimeLogger::instance();
      double tau;
      bool fStopTauCycl;
      int ktau = 1;
      prop.RazvorNode.clear();
      prop.RazvorNode.resize(koluz, 0);
      cout << "CalcFluxMixture" << endl;
      prop.RmClear();
      timeLogger.start("Flow mix");
      CalcFlux0UsredMixture();
      NullerCounter();
      CheckDirectFlux();
      MakeTrueFluxOnBC2(flMix);
      timeLogger.stop("Flow mix");
      PrintFluxIO(flMix);
      cout << "AlignmentFlux" << endl;
      timeLogger.start("Flow alignment");
      AlignmentFluxMainNew(flMix, iT, cfg);
      timeLogger.stop("Flow alignment");
      PrintFluxIO(flMix);
      for (int it = 0; it < cfg.propuski_P; it++)
      {
            CalcFluxOutPhase();
            timeLogger.start("Flow choise dt");
            CalcdtCrit(cfg.fix_dT, dt);
            if (cfg.used_fix_dt && cfg.fix_dT < dt)
            {
                  cout << "used fix dt = " << cfg.fix_dT / 24. / 3600. << endl;
                  dt = cfg.fix_dT;
            }
            timeLogger.stop("Flow choise dt");
            timeLogger.start("Flow filtration");
            CalcVolumePhase4(dt);
            FiltrationV_Crit2(dt);
            Ttotal += dt;
            prop.CheckSaturationMain();
            timeLogger.stop("Flow filtration");
            timeLogger.start("Output");
            if (prop.kolholes != 0)
                  PrintFluxV_IO_crit_hole(cfg.pathOUT, Ttotal / 3600. / 24. /*/
60.*/, iT);
            else
```

```
                PrintFluxV_IO_crit(cfg.pathOUT, Ttotal / 3600. / 24., iT);
            timeLogger.stop("Output");
            prop.CalcPropDensity();
            if (rules.size() != 0)
            {
                tau = dt / ktau;
                fStopTauCycl = false;
                for (int k = 0; k < ktau && !fStopTauCycl; k++)
                {
                    fStopTauCycl = Mixing2(tau);
                    CalcNewV2();
                    CalcNewS();
                }
            }
            prop.SwapDataSaturationProperty();
            prop.CalcPropDensity();
            prop.CalcPropEta();
            prop.CalcPropKoefPermeability();
            if (Ttotal > prop.wells_time[prop.wells_time_ind])
            {
                prop.wells_time_ind++;
                break;
            }
        }
    }

    bool FiltrationClass::CalcdtCrit(double dt_start, double & dt_ret)
    {
        double dt_new = dt_start;
        double Vout, Vel, alfa_align, Vel0;
        double flphIn, flphOut, divflph, flphOutmax;
        int neig, indfc;
        bool change = false;
        int memory_who_make_dt[2] = { -1, -1 };
        int     countAbandondt(0),     countAbandonMin(0),     countDeficitFlux(0),
countElemPhExpulsion(0);
        for (int el = 0; el < kolel; el++)
        {
            alfa_align = 0.0;
            auto & prop_el = prop.PhasesElem0[el];
            for (int ph = 0; ph < kolphase; ph++)
                alfa_align += prop_el[ph].k / prop_el[ph].eta;
            for (int ph = 0; ph < kolphase; ph++)
            {
                if (prop.S0[el][ph] < prop.Scrit_min_zero[ph])
                {
                    prop.elPhSmallerCritSmin[el][ph] = true;
                    countAbandondt++;
                    continue;
                }
                else
                    prop.elPhSmallerCritSmin[el][ph] = false;
                if (prop.S0[el][ph] < prop.Scrit_max_dt[ph])
                {
                    flphOut = flphIn = 0.;
                    flphOutmax = fluxOutPhase[Elements[el].faces[0]][ph];
                    for (int fc = 0; fc < 6; fc++)
                    {
                        indfc = Elements[el].faces[fc];
```

```
                              flphOutmax            =            max(flphOutmax,
fluxOutPhase[indfc][ph]);
                              if (fluxDirect(el, fc) == 1)
                                    flphOut += fluxOutPhase[indfc][ph];
                              else
                                    flphIn += fluxOutPhase[indfc][ph];
                        }
                        if (flphOutmax < 1e-13)
                        {
                              prop.elPhAmptyBydt[el][ph] = true;
                              countAbandondt++;
                              continue;
                        }
                        if  (flphIn  /  flphOutmax  <  eps_flux  ||  flphOut  /
flphOutmax < eps_flux)
                              divflph = 0;
                        else
                              divflph = flphIn / flphOut;

                        if  (divflph  >  eps_expulsion  &&  divflph  <  1.  /
eps_expulsion && divflph == divflph)
                        {
                              prop.elPhAmptyBydt[el][ph] = false;
                              countElemPhExpulsion++;
                        }
                        else
                        {
                              prop.elPhAmptyBydt[el][ph] = true;
                              countAbandondt++;
                              continue;
                        }
                  }
                  else
                        prop.elPhAmptyBydt[el][ph] = false;
                  Vout = 0.0;
                  for (int ed = 0; ed < 6; ed++)
                        if (fluxDirect(el, ed) == 1) // вытекающий поток
                              Vout    +=    abs(flMix[Elements[el].faces[ed]]);//
вытекающий объём смеси
                  Vout *= (prop_el[ph].k / prop_el[ph].eta) / alfa_align;
                  Vel  =  mesh->amountElem[el]  *  prop.Mat[Elements[el].nmat].F  *
prop.S0[el][ph];
                  if (Vout * dt_new > Vel) // если вытекающий объём фазы больше
объёма фазы в ячейке
                  {
                        dt_new = Vel / Vout; // то уменьшаем dt_new
                        change = true;
                        memory_who_make_dt[0] = el;
                        memory_who_make_dt[1] = ph;
                  }
            }
      }
      dt_ret = dt_new;
      cout  <<  "make  dt  "  <<  dt_new  /  24.  /  3600.  <<  "  el  =  "  <<
memory_who_make_dt[0] << " ph = " << memory_who_make_dt[1] << endl;

      int indel;
      // проверка, хватает ли смеси при выбранном dt_new
      for (int el = 0; el < kolel; el++)
      {
```

```
            Vel0 = mesh->amountElem[el] * prop.Mat[Elements[el].nmat].F;
            Vout = 0.0;
            for (int ed = 0; ed < 6; ed++)
                  if (fluxDirect(el, ed) == 1) // вытекающий поток
                        Vout += abs(flMix[Elements[el].faces[ed]]);// вытекающий
объём смеси
            if (Vout * dt_new > Vel0)
            {
                  countDeficitFlux++;
                  indel = el;
                  cout << "indel " << indel << " Vel0 = "  << Vel0 << " Vout = "
<< Vout * dt_new << " " << Vout * dt_new - Vel0 << endl;
            }
      }
      if (countDeficitFlux != 0)
            cout << "Warning: kol elem with deficit flux with this dt = " <<
countDeficitFlux << endl;

      // Поиск фаз в элементах, в которых не хватает смеси при выбранном dt (из
помечанных)
      for (int el = 0; el < kolel; el++)
      {
            alfa_align = 0.0;
            auto & prop_el = prop.PhasesElem0[el];
            for (int ph = 0; ph < kolphase; ph++)
                  alfa_align += prop_el[ph].k / prop_el[ph].eta;
            Vel0 = mesh->amountElem[el] * prop.Mat[Elements[el].nmat].F;
            for (int ph = 0; ph < kolphase; ph++)
            {
                  if (prop.elPhAmptyBydt[el][ph])
                  {
                        Vout = 0.0;
                        for (int ed = 0; ed < 6; ed++)
                              if (fluxDirect(el, ed) == 1) // вытекающий поток
                                    Vout += abs(flMix[Elements[el].faces[ed]]);
                        Vout *= (prop_el[ph].k / prop_el[ph].eta) / alfa_align;
                        Vel = Vel0 * prop.S0[el][ph];

                        if (Vout * dt_new < Vel)
                        {
                              prop.elPhAmptyBydt[el][ph] = false;
                              countAbandondt--;
                        }
                  }

                  if                (!prop.elPhAmptyBydt[el][ph]                &&
prop.elPhSmallerCritSmin[el][ph])
                  {
                        Vout = 0.0;
                        for (int ed = 0; ed < 6; ed++)
                              if (fluxDirect(el, ed) == 1) // вытекающий поток
                                    Vout += abs(flMix[Elements[el].faces[ed]]);
                        Vout *= (prop_el[ph].k / prop_el[ph].eta) / alfa_align;

                        Vel = Vel0 * prop.S0[el][ph];

                        if (Vout * dt_new >= Vel)
                        {
                              prop.elPhAmptyBydt[el][ph] = true;
                              countAbandonMin++;
```

```
                        }
                  }
            }
      }
      if (countAbandondt != 0)
            cout << "kol abandom elem by Smax (dt) = " << countAbandondt <<
endl;
      if (countElemPhExpulsion != 0)
            cout << "kol expulsion elem (cancel Smax (dt)) = " <<
countElemPhExpulsion << endl;
      if (countAbandonMin != 0)
            cout << "kol abandom elem by Smin (zero) = " << countAbandonMin <<
endl;
      return change;
     }

     void FiltrationClass::FiltrationV_Crit2(double dt)
     {
      double Vel, Vph, Vin_ph, Vin, Vout_ph, VmSum;
      vector<double> Vm(kolphase), DeltaMassm(kolphase);
      int ind_fc, ind_neighb;
      int count_nebalanceVolume(0), count_nebalanceMass(0);
      int kolcomp, numcomp;
      int neib;
      double roV, sum_mol;
      double max_nebalanceVolume(-1.0), max_nebalanceMass(-1.0);
      if (!is_init_volumePhase)
      {
            volumePhase.resize(kolel, vector<double>(kolphase, 0.));
            volumePhasePre.resize(kolel, vector<double>(kolphase, 0.));
            is_init_volumePhase = true;
      }
      for (int el = 0; el < kolel; el++)
            prop.phasecomp[el].nu_nuller();
      for (int el = 0; el < kolel; el++)
      {
            auto & phcomp_el = prop.phasecomp[el];
            Vel = mesh->amountElem[el] * prop.Mat[Elements[el].nmat].F;
            VmSum = 0.0;
            for (int ph = 0; ph < kolphase; ph++)
            {
                  kolcomp = PhaseComponents::size_comp(ph);
                  Vin_ph = 0.0;
                  Vin = 0.0;
                  Vout_ph = 0.0;
                  for (int fc = 0; fc < 6; fc++) // расчитываем втекающие потоки
                  {
                        ind_fc = Elements[el].faces[fc];
                        if (flNulls[ind_fc] == 0)
                              continue;
                        if (fluxDirect(el, fc) == -1) // втекающий поток
                        {
                              Vin_ph += volumeOutPhase[ind_fc][ph];
                        }
                        else // вытекающий поток
                        {
                              Vout_ph += volumeOutPhase[ind_fc][ph];

                              // Добавляем молей соседу, в который вытекает из
этого элемента
```

```
                              neib = mesh->get_neighbor_by_fc(el, ind_fc);
                              if (neib != -1)
                              {
                                    auto & phcomp_neib = prop.phasecomp[neib];
                                    roV    =    prop.PhasesElem0[el][ph].ro    *
volumeOutPhase[ind_fc][ph];
                                    for (int icmp = 0; icmp < kolcomp; icmp++)
                                    {
                                          numcomp = PhaseComponents::num_comp(ph,
icmp);
                                          phcomp_neib.nu_local(ph, icmp) +=
                                                roV    *    phcomp_el.hi_local(ph,
icmp) / prop.CompMolMass[numcomp];
                                    }
                              }
                        }
                  }
                  Vph = Vel * prop.S0[el][ph];
                  //Vm[ph] = Vph + Vin_ph - Vout_ph;
                  Vm[ph] = Vph - Vout_ph; // П.2 в скобках

                  roV = prop.PhasesElem0[el][ph].ro * Vm[ph];

                  //  Расчтитаем имеющиеся в ячейке минус вытекщие моли для
каждой компоненты
                  for (int icmp = 0; icmp < kolcomp; icmp++)
                  {
                        numcomp = PhaseComponents::num_comp(ph, icmp);
                        phcomp_el.nu_local(ph, icmp) +=
                              roV     *     phcomp_el.hi_local(ph,     icmp)     /
prop.CompMolMass[numcomp];
                  }

                  Vm[ph] += Vin_ph;

                  DeltaMassm[ph] = (Vin_ph - Vout_ph) * prop.Phases[ph].ro;
                  if (abs(Vm[ph] / Vel) < 1e-14)
                        Vm[ph] = 0.0;
                  VmSum += Vm[ph];
            }

            double checkV = abs(VmSum - Vel) / Vel;
            if (checkV > 1e-5)
            {
                  count_nebalanceVolume++;
                  max_nebalanceVolume = max(max_nebalanceVolume, checkV);
            }
            for (int ph = 0; ph < kolphase; ph++)
            {
                  volumePhasePre[el][ph] = Vm[ph];
                  prop.S[el][ph] = Vm[ph] / VmSum;
            }
      } // el

      // Обработка молей для краевых условий
      int kolbc, el, fc, ind_glob_fc;
      for (int ih = 0; ih < prop.kolholes; ih++)
      {
            auto holescomp = prop.holes_components[ih];
            kolbc = prop.holes[ih].size();
```

```
            for (int i = 0; i < kolbc; i++)
            {
                  el = prop.holes[ih][i].el;
                  fc = prop.holes[ih][i].face;
                  if (fluxDirect(el, fc) == -1)
                  {
                        ind_glob_fc = Elements[el].faces[fc];
                        for (int ph = 0; ph < kolphase; ph++)
                        {
                              roV         =         prop.Phases[ph].ro         *
volumeOutPhase[ind_glob_fc][ph];
                              kolcomp = PhaseComponents::size_comp(ph);
                              for (int icmp = 0; icmp < kolcomp; icmp++)
                              {
                                    numcomp    =    PhaseComponents::num_comp(ph,
icmp);
                                    prop.phasecomp[el].nu_local(ph, icmp) +=
                                          roV  *  holescomp.hi_local(ph,  icmp)  /
prop.CompMolMass[numcomp];
                              }
                        }
                  }
            }
      }

      kolbc = prop.bc1faces.size();
      for (int i = 0; i < kolbc; i++)
      {
            el = prop.bc1faces[i].el;
            fc = prop.bc1faces[i].face;
            if (fluxDirect(el, fc) == -1)
            {
                  ind_glob_fc = Elements[el].faces[fc];
                  for (int ph = 0; ph < kolphase; ph++)
                  {
                        roV           =           prop.Phases[ph].ro           *
volumeOutPhase[ind_glob_fc][ph];
                        kolcomp = PhaseComponents::size_comp(ph);
                        for (int icmp = 0; icmp < kolcomp; icmp++)
                        {
                              numcomp = PhaseComponents::num_comp(ph, icmp);
                              prop.phasecomp[el].nu_local(ph, icmp) +=
                                    roV  *  prop.SbcCommon_components.hi_local(ph,
icmp) / prop.CompMolMass[numcomp];
                        }
                  }
            }
      }

      // Расчёт новых hi П.3
      for (int el = 0; el < kolel; el++)
      {
            for (int ph = 0; ph < kolphase; ph++)
            {
                  if                (!prop.elPhAmptyBydt[el][ph]                &&
!prop.elPhSmallerCritSmin[el][ph])
                  {
                        sum_mol = 0.0;
                        kolcomp = PhaseComponents::size_comp(ph);
                        for (int icmp = 0; icmp < kolcomp; icmp++)
```

```
                        {
                              numcomp = PhaseComponents::num_comp(ph, icmp);
                              sum_mol += prop.phasecomp[el].nu_local(ph, icmp) *
prop.CompMolMass[numcomp];
                        }
                        if (sum_mol > 1e-13)
                        {
                              for (int icmp = 0; icmp < kolcomp; icmp++)
                              {
                                    numcomp    =    PhaseComponents::num_comp(ph,
icmp);
                                    prop.phasecomp[el].hi_local(ph, icmp) =
                                          prop.phasecomp[el].nu_local(ph, icmp) *
prop.CompMolMass[numcomp] / sum_mol;
                              }
                        }
                  }
            }
      }

      if (count_nebalanceVolume != 0)
            cout   <<   "FiltrationV_Crit   :   kol   nebalce   volume   =   "   <<
count_nebalanceVolume << "  max = " << max_nebalanceVolume << endl;
     }

     FiltrationClass::FiltrationClass(
      MeshXYZ *_mesh,
      FiltrProperties &_prop,
      MethodGaussFast_FEM & _femCalculator3D,
      MethodGaussFast_FEM2D & _femCalculator2D
     )
      :
      prop(_prop),
      femCalculator3D(_femCalculator3D),
      femCalculator2D(_femCalculator2D),
      Elements(_mesh->Elements),
      Points(_mesh->Points),
      bc1faces(_prop.bc1faces),
      bc2faces(_prop.bc2faces)
     {
      kolel = _mesh->kolel;
      koluz = _mesh->koluz;
      kolvar = koluz;
      kolphase = prop.kolphase;
      kolfc = _mesh->koled;
      mesh = _mesh;
      Gauss2D.ChoiseMethodGauss(4);

      is_init_flux0 = false;
      is_init_fluxMixture = false;
      is_init_volumeOutPhase = false;
      is_init_fluxOutPhase = false;
      is_init_volumePhase = false;
      Ttotal = 0.;
      flVbc2.resize(kolphase, 0.);
      flVbc1.resize(kolphase, 0.);
     }

     void BuildPortretbyEdge(int kolel, int koled, const vector<Element2D>&
Elements, vector<int>& ig, vector<int>& jg)
```

```
{
 int i;
 const int lockoled = 4;
 ig.resize(koled + 1);
 vector<int>::iterator it;
 vector<set<int>> list(koled);
 for (int ielem = 0; ielem < kolel; ielem++)
 {
       for (int i = 0; i < lockoled; i++)
       {
             int k = Elements[ielem].edge[i];
             for (int j = 0; j < lockoled; j++)
             {
                   if (i == j)
                         continue;
                   int ind1 = k;
                   int ind2 = Elements[ielem].edge[j];
                   if (ind2 < ind1)
                   {
                         ind1 = ind2;
                         ind2 = k;
                   }
                   //вставка
                   list[ind2].insert(ind1);
             }
       }
 }
 ig[0] = 0;
 for (i = 0; i < koled; i++)
 {
       ig[i + 1] = ig[i];
       for (auto & uz : list[i])
       {
             jg.push_back(uz);
             //ig[i + 1]++;
       }
       ig[i + 1] += (int)list[i].size();
       /*if (list[i].size() != 3)
             cout << list[i].size() << endl;*/
 }
}
```

## Filtr_Structs.h

```
#pragma once
#include"stdafx.h"
#include"FEM_Structs.h"

namespace fltrStruct
{
 enum ePropPhase
 {
       density,
       viscosity,
       koef_permeability,
       nonPropPhase
 };
```

```
enum ePropEnviron
{
	pressure,
	temperature,
	press_temp,
	saturation,
	velocityFiltr,
	compSelf,
	nonPropEnviron
};

enum ePropPhase To_ePropPhase(string &str);
enum ePropEnviron To_ePropEnviron(string &str);

struct Phase
{
	double ro;		// плотность
	double k;		// множитель структурной проницаемости
	double eta;		// вязкость
	double lyambda; // теплопроводность
	double c;		// теплоёмкость

	inline double get(ePropPhase properties)
	{
		switch (properties)
		{
		density:
			return ro;
			break;
		viscosity:
			return eta;
			break;
		koef_permeability:
			return k;
			break;
		default:
			break;
		}
		return -1;
	}
};

struct Material
{
	double K; // проницаемость
	double F; // пористость
};

struct TrubaBC
{
	int el, face;
	double dP;
};

int FindIntervalInDoubleVec(vector<double> &vec, double elem);

struct Table
{
	int sz;
	vector<double> arg, val;
```

```
	void Read(ifstream &inf)
	{
		int i;
		inf >> sz;
		arg.resize(sz);
		val.resize(sz);
		for (i = 0; i < sz; i++) { inf >> arg[i] >> val[i]; }
	}
	double GetValue(double x)
	{
		int i;
		double y, cff;
		y = 0.0;
		if (sz)
		{
			i = FindIntervalInDoubleVec(arg, x);
			if (i != -1)
			{
				cff = (x - arg[i]) / (arg[i + 1] - arg[i]);
				y = (1.0 - cff)*val[i] + cff*val[i + 1];
			}
			else if (x < arg[0]) { y = val[0]; }
			else { y = val[sz - 1]; }
/*(x>arg[sz-1])*/
		}
		return y;
	}
};

class PhaseComponents
{
private:
	static vector<string> namePhase, nameComp;
	static vector<vector<int>> tableInd;
	static vector<vector<int>> tableIndOpt;
	static vector<vector<int>> tableIndOptNumComp;
	static vector<double> start_phasecomp;
	vector<double> phasecomp_hi, phasecomp_nu;
public:
	static int kolphase, kolcomp, kolphcomp;
	static int ReadPhaseComponentsStruct(std::ifstream &inf);
	int ReadPhaseComponents(std::ifstream &inf);
	static inline int size_comp(int ph)
	{
		return tableIndOptNumComp[ph].size();
	}
	static inline int num_comp(int iphase, int icomp_local)
	{
		return tableIndOptNumComp[iphase][icomp_local];
	}
	inline void nu_nuller()
	{
		for (auto & i : phasecomp_nu)
			i = 0.0;
	}
	inline double& hi(int iphase, int icomp_glob)
	{
		return phasecomp_hi[tableInd[iphase][icomp_glob]];
	}
	inline double& nu(int iphase, int icomp_glob)
```

```
            {
                  return phasecomp_nu[tableInd[iphase][icomp_glob]];
            }
            inline double& hi_local(int iphase, int icomp_local)
            {
                  return phasecomp_hi[tableIndOpt[iphase][icomp_local]];
            }
            inline double& nu_local(int iphase, int icomp_local)
            {
                  return phasecomp_nu[tableIndOpt[iphase][icomp_local]];
            }
            inline void set_phasecomp_default()
            {
                  phasecomp_hi = start_phasecomp;
                  phasecomp_nu.resize(kolphcomp);
            }
            inline int set_phasecomp(vector<double> &val)
            {
                  if (kolphcomp == val.size())
                  {
                        phasecomp_hi = val;
                        phasecomp_nu.resize(kolphcomp);
                        return 0;
                  }
                  else
                        return 1;
            }
            bool check_hi();
            inline PhaseComponents& operator= (const PhaseComponents &val)
            {
                  if (phasecomp_hi != val.phasecomp_hi)
                  {
                        phasecomp_hi = val.phasecomp_hi;
                  }
                  if (phasecomp_nu != val.phasecomp_nu)
                  {
                        phasecomp_nu = val.phasecomp_nu;
                  }
                  return *this;
            }

            friend   bool   operator==(const   PhaseComponents   &left,   const
PhaseComponents &right)
            {
                  return      left.phasecomp_hi == right.phasecomp_hi
                              && left.phasecomp_nu == right.phasecomp_nu;
            }
            inline void set_value(int iphase, int icomp_glob, double val)
            {
                  phasecomp_hi[tableInd[iphase][icomp_glob]] = val;
            }
            inline double& value_local(int iphase, int icomp_loc)
            {
                  return phasecomp_hi[tableIndOpt[iphase][icomp_loc]];
            }
      };

      struct RelationPhaseEnviron
      {
            ePropPhase propPhase;
```

```
            ePropEnviron propEnviron;
            int info; // например, номер фазы от которой зависит коеф.
проницаемости
            int table_ind;
            string table_NameFile;
      };

      struct WellsSettings
      {
            double time;
            double consump;
            int num_comp;
            vector<double> s;
      };

     #define eps_check_saturation 1e-14

      class FiltrProperties
      {
      public:
            int kolphase, kolmat, kolel, kolrelationprop, kolholes;
            std::vector<Material> Mat;
            std::vector<Phase> Phases;
            int kolbc1faces, kolbc2faces, kolbc2wells;
            vector<BCFaces> bc1faces, bc2faces;
            vector<BCNodes> bc1nodes;
            vector<double> CompMolMass,CompRo;
            vector<Table> tableRelations;
            std::vector<unordered_map<ePropPhase,         RelationPhaseEnviron>>
relationprop; // вектор - по количеству фаз
            vector<PhaseComponents> phasecomp;
            vector<double> RazvorNode;
            vector<double> Scrit_min_zero, Scrit_max_dt;
            vector<vector<bool>> elPhSmallerCritSmin, elPhAmptyBydt;
            vector<vector<TrubaBC>> holes;
            vector<vector<WellsSettings>> wells_settings;
            int wells_time_ind;
            vector<double> wells_time;
            vector<double> holes_consumption;
            vector<vector<double>> holes_suturation;
            vector<int> holes_num_comp;
            vector<PhaseComponents> components_data;
            vector<PhaseComponents> holes_components;
            vector<double> SbcCommon_interval_z;
            vector<vector<double>> SbcCommon_interval;
            vector<vector<int>> SbcCommon_interval_indbc1;
            vector<double> SbcCommon;
            PhaseComponents SbcCommon_components;
            vector<vector<double>> S0, S;
            bool fInitEta, fInitKoefPermeability, fInitDensity;
            vector<bool> fRelatEta, fRelatKoefPermeability, fRelatDensity;
            vector<vector<Phase>> PhasesElem0, PhasesElem;
            vector<double> P;
            vector<vector<double>> Rm;
            vector<double> dP;
            vector<double> SredElVelos;
            FiltrProperties();
            ~FiltrProperties();
            void InitMesh(int _kolel);
            void InitMemory();
```

```
            int ReadPhaseComponentsStruct(string & pathInput);
            int ReadRelationPhaseEnviron(string & pathInput);
            int ReadPhase(std::string &pathInput);
            int ReadMaterials(std::string &pathInput);
            int ReadP(std::string &pathInput, int kolvar);
            int PullP(vector<double> &_Psours);
            int ReadS0(string & pathInput);
            int ReadS(string & pathInput, int iT);
            int ReadSbcCommon(string & pathInput);
            int ReadSbcCommon_interval_z(string & pathInput); // задаёт внешние
насыщенности по z сверху вниз, в файле задаётся нижняя граница
            int ReadSCritical(string & pathInput);
            int ReadBC1(string & pathInput);
            int ReadBC2(string & pathInput);
            int InitBC2ByHoles();
            int ReadWells(string & pathInput);
            int ReadWellsSettings(string & pathInput);
            int InitWellsSettings(double time);
            int InitS0(const vector<double> & s0);
            int ReadD0(string & pathInput);
            int ConvertConsumptToSec();
            int InitPhasesElem_default();
            inline     void     SwapDataSaturationProperty()     {     S0.swap(S);
PhasesElem0.swap(PhasesElem); };
            void CalcPropEta();
            void CalcPropKoefPermeability();
            void CalcPropDensity();
            void RmClear();
            int ReadCompProp(string & pathInput);
            int CheckSaturationMain();
            int CheckSaturationCrit();
            int WriteS(string & pathOutput, int iT);
     };
    }
```

## Filtr_Structs.cpp

```
#include "Filtr_Structs.h"

int fltrStruct::PhaseComponents::kolphase;
int fltrStruct::PhaseComponents::kolcomp;
int fltrStruct::PhaseComponents::kolphcomp;
vector<vector<int>> fltrStruct::PhaseComponents::tableInd;
vector<vector<int>> fltrStruct::PhaseComponents::tableIndOpt;
vector<vector<int>> fltrStruct::PhaseComponents::tableIndOptNumComp;
vector<double> fltrStruct::PhaseComponents::start_phasecomp;
vector<string> fltrStruct::PhaseComponents::namePhase;
vector<string> fltrStruct::PhaseComponents::nameComp;

fltrStruct::FiltrProperties::FiltrProperties()
{
 fInitEta = fInitKoefPermeability = fInitDensity = false;
}

fltrStruct::FiltrProperties::~FiltrProperties()
{
}

void fltrStruct::FiltrProperties::CalcPropKoefPermeability()
```

```
	{
	 for (int ph = 0; ph < kolphase; ph++)
	 {
		if (fRelatKoefPermeability[ph])
		{
			auto relation = relationprop[ph][koef_permeability];
			auto table = tableRelations[relation.table_ind];
			switch (relation.propEnviron)
			{
			case saturation:
				for (int el = 0; el < kolel; el++)
					PhasesElem[el][ph].k	=	S0[el][ph]	*
table.GetValue(S0[el][relation.info - 1]);
				break;
			default:
				break;
			}
		}
		else
		{
			for (int el = 0; el < kolel; el++)
				PhasesElem[el][ph].k = S0[el][ph];
		}
	 }
	}

	void fltrStruct::FiltrProperties::CalcPropDensity()
	{
	 int count = kolphase;
	 for (int ph = 0; ph < kolphase; ph++)
	 {
		if (fRelatDensity[ph])
		{
			auto relation = relationprop[ph][density];
			auto table = tableRelations[relation.table_ind];
			switch (relation.propEnviron)
			{
			case saturation:
				for (int el = 0; el < kolel; el++)
					PhasesElem[el][ph].ro
=
table.GetValue(S0[el][relation.info - 1]);
				break;
			default:
				break;
			}
		}
		else
		{
			if (!fInitDensity)
			{
				for (int el = 0; el < kolel; el++)
					PhasesElem[el][ph].ro = Phases[ph].ro;
				count--;
				if (count == 0)
					fInitDensity = true;
			}
		}
	 }
	}
```

```
void fltrStruct::FiltrProperties::RmClear()
{
 if (Rm.size() != kolel)
       Rm.resize(kolel, vector<double>(kolphase, 0.0));
 for (auto & i : Rm)
       fill(i.begin(), i.end(), 0.0);
}

int fltrStruct::FiltrProperties::ReadCompProp(string & pathInput)
{
 ifstream inf;
 string path;
 int kol;

 path = pathInput + "/CompMolMass.txt";
 inf.open(path.c_str());
 if (!inf)
 {
       cout << "Can't open file " << path << endl;
       return 1;
 }
 inf >> kol;
 if (!inf.good())
 {
       cout << "Can't open file " << path << endl;
       return 1;
 }
 if (kol != PhaseComponents::kolcomp)
 {
       cout     <<     "FiltrProperties::ReadCompMolMass     :     kol     !=
phasecomp.kolcomp"<<"      kol=      "<<kol<<"      phasecomp.kolcomp=      "<<
PhaseComponents::kolcomp << endl;
       return 1;
 }
 CompMolMass.resize(kol);
 for (int i = 0; i < kol; i++)
 {
       inf >> CompMolMass[i];
 }
 inf.close();
 inf.clear();

 path = pathInput + "/CompRo.txt";
 inf.open(path.c_str());
 if (!inf)
 {
       cout << "Can't open file " << path << endl;
       return 1;
 }
 inf >> kol;
 if (!inf.good())
 {
       cout << "Can't open file " << path << endl;
       return 1;
 }
 if (kol != PhaseComponents::kolphase)
 {
       cout     <<     "FiltrProperties::ReadCompMolMass     :     kol     !=
phasecomp.kolcomp"  <<  "  kol=  "  <<  kol  <<  "  phasecomp.kolcomp=  "  <<
PhaseComponents::kolphase << endl;
```

```
		return 1;
	}
	CompRo.resize(kol);
	for (int i = 0; i < kol; i++)
	{
		inf >> CompRo[i];
	}
	inf.close();
	inf.clear();

	return 0;
}

int fltrStruct::FiltrProperties::CheckSaturationMain()
{
	int res = 0;
	double sumS;
	for (int el = 0; el < kolel; el++)
	{
		sumS = 0.0;
		for (int ph = 0; ph < kolphase; ph++)
			sumS += S0[el][ph];
		if (abs(1 - abs(sumS)) > eps_check_saturation)
			res++;
	}
	if (res != 0)
		cout << "sum suturation != 1.0 kol elem = " << res << endl;
	return res;
}

int fltrStruct::FiltrProperties::CheckSaturationCrit()
{
	for (int el = 0; el < kolel; el++)
		for (int ph = 0; ph < kolphase; ph++)
		{
			if (S0[el][ph] < Scrit_min_zero[ph])
				elPhSmallerCritSmin[el][ph] = true;
			else
				elPhSmallerCritSmin[el][ph] = false;
		}
	return 0;
}
```

## ClassMethodGaussFast.h

```
#pragma once

#include"GaussConsts.h"
#include<vector>
#include"BaseFunc_lin.h"

using namespace bfL1;

class MethodGaussFast_FEM
{
private:
	static const int Nbf = 8;
```

```
      static const int NMethodGauss = 3;
      static const int Ngp = NMethodGauss * NMethodGauss * NMethodGauss;

      const double *p, *gauss_koeff;
      double g_phi[Ngp][Nbf];       // значения базисных функций в точках гаусса
      double g_d_phi[Ngp][Nbf][3];
      double(*g_J_d_phi)[8][3];          // для якобина - пока псевдоним, пока
функции трилинейные
      double AGk[Ngp];                   // All Gauss koeff
      double AGp[Ngp][3];                      // All Gauss points
      void ChoiseNumPoint();
      void CalculateGaussArrays();
     public:
      MethodGaussFast_FEM();

      void Calc_J(double *x, double *y, double *z, double(*J_1_T)[3], double
*det_J_abs, int n_of_point);
      double Calc_detJ_abs(double *x, double *y, double *z, int n_of_point);
      void Calc_Matrix_G(double *x, double *y, double *z, double(*mg)[Nbf]);
      void Calc_Matrix_M(double *x, double *y, double *z, double(*mg)[Nbf]);
      double CalcAmountHexagon(double *x, double *y, double *z);
     };
```

## ClassMethodGaussFast.cpp

```
     #include "ClassMethodGaussFast.h"

     void MethodGaussFast_FEM::ChoiseNumPoint()
     {
      switch (NMethodGauss)
      {
      case 3:
            p = gauss_3_point.data();
            gauss_koeff = gauss_3_koef.data();
            break;
      case 4:
            p = gauss_4_point.data();
            gauss_koeff = gauss_4_koef.data();
            break;
      default:
            printf("Only Guss 3 or 4, sorry\n");
            break;
      }
     }

     MethodGaussFast_FEM::MethodGaussFast_FEM()
     {
      int i, j, k, m = 0;
      ChoiseNumPoint();
      for (k = 0; k < NMethodGauss; k++)
            for (j = 0; j < NMethodGauss; j++)
                  for (i = 0; i < NMethodGauss; i++)
                  {
                        AGp[m][0] = p[i];
                        AGp[m][1] = p[j];
                        AGp[m][2] = p[k];
                        AGk[m]    =    gauss_koeff[i]    *    gauss_koeff[j]    *
gauss_koeff[k];
```

```
                        m++;
                    }
      CalculateGaussArrays();
     }

     void MethodGaussFast_FEM::Calc_J(double * x, double * y, double * z,
double(*J_1_T)[3], double * det_J_abs, int n_of_point)
     {
      int i, j;
      double J[3][3], det_J;
      memset(J, 0, sizeof(double) * 9);
      for (i = 0; i < 8; i++)
      {
            J[0][0] += x[i] * g_J_d_phi[n_of_point][i][0]; //  d_xi[i];
            J[0][1] += x[i] * g_J_d_phi[n_of_point][i][1]; //  d_eta[i];
            J[0][2] += x[i] * g_J_d_phi[n_of_point][i][2]; //  d_zeta[i];

            J[1][0] += y[i] * g_J_d_phi[n_of_point][i][0]; //  d_xi[i];
            J[1][1] += y[i] * g_J_d_phi[n_of_point][i][1]; //  d_eta[i];
            J[1][2] += y[i] * g_J_d_phi[n_of_point][i][2]; //  d_zeta[i];

            J[2][0] += z[i] * g_J_d_phi[n_of_point][i][0]; //  d_xi[i];
            J[2][1] += z[i] * g_J_d_phi[n_of_point][i][1]; //  d_eta[i];
            J[2][2] += z[i] * g_J_d_phi[n_of_point][i][2]; //  d_zeta[i];
      }
      // вычисляем Якобиан (определитель)
      det_J = J[0][0] * J[1][1] * J[2][2] - J[0][0] * J[1][2] * J[2][1] +
J[1][0] * J[2][1] * J[0][2]
            - J[1][0] * J[0][1] * J[2][2] + J[2][0] * J[0][1] * J[1][2] -
J[2][0] * J[1][1] * J[0][2];
      // модуль Якобиана
      *det_J_abs = fabs(det_J);
      // матрица, обратная к транспонированной матрице Якоби
      J_1_T[0][0] = (J[1][1] * J[2][2] - J[2][1] * J[1][2]) / det_J;
      J_1_T[1][0] = (J[2][1] * J[0][2] - J[0][1] * J[2][2]) / det_J;
      J_1_T[2][0] = (J[0][1] * J[1][2] - J[1][1] * J[0][2]) / det_J;
      J_1_T[0][1] = (-J[1][0] * J[2][2] + J[2][0] * J[1][2]) / det_J;
      J_1_T[1][1] = (J[0][0] * J[2][2] - J[2][0] * J[0][2]) / det_J;
      J_1_T[2][1] = (-J[0][0] * J[1][2] + J[1][0] * J[0][2]) / det_J;
      J_1_T[0][2] = (J[1][0] * J[2][1] - J[2][0] * J[1][1]) / det_J;
      J_1_T[1][2] = (-J[0][0] * J[2][1] + J[2][0] * J[0][1]) / det_J;
      J_1_T[2][2] = (J[0][0] * J[1][1] - J[1][0] * J[0][1]) / det_J;
     }

     double MethodGaussFast_FEM::Calc_detJ_abs(double * x, double * y, double *
z, int n_of_point)
     {
      double J[3][3], det_J;
      memset(J, 0, sizeof(double) * 9);
      for (int i = 0; i < 8; i++)
      {
            J[0][0] += x[i] * g_J_d_phi[n_of_point][i][0]; //  d_xi[i];
            J[0][1] += x[i] * g_J_d_phi[n_of_point][i][1]; //  d_eta[i];
            J[0][2] += x[i] * g_J_d_phi[n_of_point][i][2]; //  d_zeta[i];

            J[1][0] += y[i] * g_J_d_phi[n_of_point][i][0]; //  d_xi[i];
            J[1][1] += y[i] * g_J_d_phi[n_of_point][i][1]; //  d_eta[i];
            J[1][2] += y[i] * g_J_d_phi[n_of_point][i][2]; //  d_zeta[i];

            J[2][0] += z[i] * g_J_d_phi[n_of_point][i][0]; //  d_xi[i];
```

```
            J[2][1] += z[i] * g_J_d_phi[n_of_point][i][1]; //  d_eta[i];
            J[2][2] += z[i] * g_J_d_phi[n_of_point][i][2]; //  d_zeta[i];
      }
      det_J = J[0][0] * J[1][1] * J[2][2] - J[0][0] * J[1][2] * J[2][1] +
J[1][0] * J[2][1] * J[0][2]
            - J[1][0] * J[0][1] * J[2][2] + J[2][0] * J[0][1] * J[1][2] -
J[2][0] * J[1][1] * J[0][2];
      return fabs(det_J);
     }

     void MethodGaussFast_FEM::Calc_Matrix_G(double * x, double * y, double * z,
double(*mg)[Nbf])
     {
      int i, j, c, i1, j1;
      double scal, gauss_mult, J_1_T[3][3], det_J_abs;
      double  grad_all[Nbf][3];  //  градиенты  от  баз.  функций  в  точке
интегрирования
      memset(mg, 0, sizeof(double) * Nbf*Nbf);
      for (i = 0; i < Ngp; i++)
      {
            Calc_J(x, y, z, J_1_T, &det_J_abs, i);
            gauss_mult  =  AGk[i]  *  det_J_abs;  //  A_i*A_j*A_k*|J|,  A  -
коэффициенты метода гаусса

            for  (j  =  0;  j  <  Nbf;  j++)  //  градиенты  от  базисных  ф-ций
преобразуются по правилу...
                  for (c = 0; c < 3; c++)
                        grad_all[j][c]  =  J_1_T[c][0]  *  g_d_phi[i][j][0]  +
J_1_T[c][1] * g_d_phi[i][j][1] + J_1_T[c][2] * g_d_phi[i][j][2];

            for (i1 = 0; i1 < Nbf; i1++)
                  for (j1 = 0; j1 < Nbf; j1++)
                  {
                        scal = 0.;
                        for (c = 0; c < 3; c++)
                              scal  +=  grad_all[i1][c]  *  grad_all[j1][c];  //
transpose
                        mg[j1][i1] += scal * gauss_mult;
                  }
      }
     }

     void MethodGaussFast_FEM::Calc_Matrix_M(double * x, double * y, double * z,
double(*mg)[Nbf])
     {
      int i, c, i1, j1;
      double scal, gauss_mult, J_1_T[3][3], det_J_abs;
      memset(mg, 0, sizeof(double) * Nbf*Nbf);
      for (i = 0; i < Ngp; i++)
      {
            Calc_J(x, y, z, J_1_T, &det_J_abs, i);
            gauss_mult  =  AGk[i]  *  det_J_abs;  //  A_i*A_j*A_k*|J|,  A  -
коэффициенты метода гаусса

            for (i1 = 0; i1 < Nbf; i1++)
                  for (j1 = 0; j1 < Nbf; j1++)
                  {
                        mg[i1][j1] += g_phi[i][i1] * g_phi[i][i1] * gauss_mult;
                  }
      }
```

```
    }

    double MethodGaussFast_FEM::CalcAmountHexagon(double * x, double * y,
double * z)
    {
     double res (0.);
     for (int i = 0; i < Ngp; i++)
           res += AGk[i] * Calc_detJ_abs(x, y, z, i);
     return res;
    }

    void MethodGaussFast_FEM::CalculateGaussArrays()
    {
     int i, j, k,imu, inu, ith, m, n;
     double x, y, z;
     m = 0;
     for (k = 0; k < NMethodGauss; k++)
     {
           z = p[k];
           for (j = 0; j < NMethodGauss; j++)
           {
                 y = p[j];
                 for (i = 0; i < NMethodGauss; i++)
                 {
                       x = p[i];
                       for (n = 0; n < Nbf; n++)
                       {
                             imu = ind_3d_map::mu(n);
                             inu = ind_3d_map::nu(n);
                             ith = ind_3d_map::th(n);
                             g_phi[m][n] = phi3d(imu, inu, ith, x, y, z);
                             g_d_phi[m][n][0] = bfL1::d_phi3d(imu, inu, ith, x,
y, z, 1);
                             g_d_phi[m][n][1] = bfL1::d_phi3d(imu, inu, ith, x,
y, z, 2);
                             g_d_phi[m][n][2] = bfL1::d_phi3d(imu, inu, ith, x,
y, z, 3);
                       }
                       m++;
                 }
           }
     }
     g_J_d_phi = g_d_phi;
    }
```