

# Содержание

<b>1 Лекция 1 (02.09)</b>	<b>3</b>
<b>2 Лекция 2 (09.09)</b>	<b>4</b>
2.1 Уравнение Пуассона . . . . .	4
2.1.1 Постановка задачи . . . . .	4
2.1.2 Метод решения . . . . .	4
2.1.2.1 Нахождение численного решения . . . . .	4
2.1.2.2 Практическое определения порядка аппроксимации . . . . .	5
2.1.3 Программная реализация . . . . .	6
2.1.3.1 Функция верхнего уровня . . . . .	6
2.1.3.2 Детали реализации . . . . .	7
2.2 Задание для самостоятельной работы . . . . .	9
2.2.1 Порядок сходимости . . . . .	9
2.2.2 Двумерное уравнение Пуассона . . . . .	9
<b>3 Лекция 3 (16.09)</b>	<b>11</b>
3.1 Двухслойные схемы для нестационарных уравнений . . . . .	11
3.1.1 Определение . . . . .	11
3.1.1.1 Явная схема . . . . .	11
3.1.1.2 Неявная схема . . . . .	11
3.1.1.3 Схема Кранка–Николсон . . . . .	12
3.1.1.4 Обобщённая двухслойная схема . . . . .	12
3.1.2 Дискретизация по времени как итерационный процесс . . . . .	13
3.1.2.1 Двухслойный итерационный процесс . . . . .	13
3.1.2.2 Устойчивость итерационного процесса . . . . .	13
3.1.2.3 Источники возмущений . . . . .	14
3.2 Методы исследования устойчивости расчётных схем . . . . .	15
3.2.1 Матричный метод . . . . .	15
3.2.1.1 Явная схема для нестационарного уравнения диффузии . . . . .	15
3.2.1.2 Неявная схема для нестационарного уравнения диффузии . . . . .	16
3.2.2 Метод дискретных возмущений . . . . .	17
3.2.2.1 Явная схема против потока для уравнения переноса . . . . .	17
3.2.3 Метод Неймана . . . . .	17
3.2.3.1 Неявная противопотоковая схема для уравнения переноса . . . . .	18
3.2.3.2 Противопотоковая схема Кранка–Николсон для уравнения переноса . . . . .	19
3.2.3.3 Явная схема для уравнения нестационарной конвекции-диффузии . . . . .	20
3.2.3.4 Неявная схема для уравнения нестационарной конвекции-диффузии . . . . .	21
3.2.4 Общие рекомендации к выбору устойчивых расчётных схем . . . . .	21
3.3 Программная реализация схемы для уравнения переноса . . . . .	22
3.3.1 Постановка задачи . . . . .	22

3.3.2	Функция верхнего уровня . . . . .	23
3.3.3	Расчётные функции . . . . .	24
3.3.3.1	Явная схема . . . . .	24
3.3.3.2	Неявная схема . . . . .	24
3.3.3.3	Схема Кранка-Николсон . . . . .	25
3.3.4	Анализ результатов работы . . . . .	25
3.4	Задание для самостоятельной работы . . . . .	27
3.4.1	Постановка задачи . . . . .	27
3.4.1.1	Тестовый пример 1 . . . . .	27
3.4.1.2	Тестовый пример 2 . . . . .	27
3.4.2	Расчётная схема . . . . .	28
<b>4</b>	<b>Лекция 4 (30.09)</b>	<b>31</b>
4.1	Моделирование течения вязкой несжимаемой жидкости методом конечных разностей . . . . .	31
4.1.1	Система уравнений Навье-Стокса . . . . .	31
4.1.2	Схема расчёта . . . . .	32
4.1.2.1	Метод SIMPLE . . . . .	32
4.1.3	Пространственная аппроксимация . . . . .	34
4.1.3.1	Разнесённая сетка . . . . .	34
4.1.3.2	Уравнения движения . . . . .	35
4.1.3.3	Уравнение для поправки давления . . . . .	37
4.1.3.4	Уравнение для поправки скорости . . . . .	39
4.1.3.5	Учёт граничных условий . . . . .	39
4.2	Программа для расчёта течения в каверне по схеме SIMPLE . . . . .	41
4.2.1	Постановка задачи . . . . .	42
4.2.2	Функция верхнего уровня . . . . .	43
4.2.3	Поля класса решателя . . . . .	44
4.2.4	Инициализация решателя . . . . .	44
4.2.5	Шаг итерации SIMPLE . . . . .	45
4.2.6	Сборка системы уравнений для поправки давления . . . . .	45
4.2.7	Сборка системы уравнений для пробной скорости . . . . .	45
4.3	Задание для самостоятельной работы . . . . .	46
<b>5</b>	<b>Лекция 5 (6.10)</b>	<b>47</b>
5.1	Оптимальные значения параметров алгоритма SIMPLE . . . . .	47
5.2	Нестационарное уравнение Навье-Стокса . . . . .	47
5.2.1	Схема расчёта по алгоритму SIMPLE . . . . .	47
5.3	Завихренность и функция тока . . . . .	49
5.3.1	Определение завихренности и функции тока на разнесённой сетке . . . . .	50
5.4	Задание для самостоятельной работы . . . . .	51

<b>6 Лекция 6 (13.10)</b>	<b>52</b>
6.1 Оптимизация методов решения СЛАУ . . . . .	52
6.1.1 Метод Якоби . . . . .	52
6.1.2 Метод Зейделя . . . . .	53
6.1.3 Метод последовательных верхних релаксаций (SOR) . . . . .	53
6.1.4 Формат хранения разреженных матриц CSR . . . . .	54
6.2 Задача об обтекании препятствия . . . . .	56
6.2.1 Расчётная сетка . . . . .	56
6.2.2 Граничные условия . . . . .	57
6.2.2.1 Входное сечение . . . . .	57
6.2.2.2 Условия симметрии . . . . .	59
6.2.2.3 Условия прилипания . . . . .	59
6.2.2.4 Выходные граничные условия . . . . .	60
6.2.3 Баланс сил. Коэффициенты сил . . . . .	62
6.2.3.1 Сопротивление . . . . .	62
6.2.3.2 Подъёмная сила . . . . .	63
6.2.3.3 Вычисление коэффициентов сил на разнесённой сетке . . . . .	64
6.3 Задание для самостоятельной работы . . . . .	65
<b>7 Лекция 7 (20.10)</b>	<b>69</b>
7.1 Инициализация решения . . . . .	69
7.1.1 Задача о потенциале течения . . . . .	69
7.1.2 Аппроксимация на разнесённой сетке . . . . .	69
7.2 Конвективный теплообмен . . . . .	70
7.2.1 Уравнение теплопроводности . . . . .	70
7.2.2 Дискретизация по времени . . . . .	71
7.2.3 Аппроксимация на разнесённой сетке . . . . .	71
7.2.4 Граничные условия . . . . .	72
7.2.4.1 Условия первого рода . . . . .	72
7.2.4.2 Условия второго рода . . . . .	72
7.2.4.3 Условия третьего рода . . . . .	73
7.2.4.4 Универсальность условий третьего рода . . . . .	73
7.2.5 Коэффициент теплообмена . . . . .	73
7.3 Тестовые примеры . . . . .	74
7.3.1 Задача о равномерном течении . . . . .	74
7.3.1.1 Учёт граничных условий . . . . .	74
7.3.1.2 Анализ результатов . . . . .	75
7.3.2 Течение Пуазейля . . . . .	75
7.3.3 Стационарное обтекание квадратного препятствия . . . . .	75
7.3.3.1 Функция верхнего уровня . . . . .	76
7.3.3.2 Учёт неактивных ячеек . . . . .	77
7.3.3.3 Расчёт коэффициентов сопротивления . . . . .	77

7.3.3.4	Результаты расчёта . . . . .	78
7.3.4	Нестационарное обтекание квадратного препятствия с теплообменом . . . . .	78
7.3.4.1	Функция верхнего уровня . . . . .	79
7.3.4.2	Учёт нестационарности . . . . .	79
7.3.4.3	Расчёт температурного поля . . . . .	80
7.3.4.4	Вычисление коэффициента теплообмена . . . . .	80
7.3.4.5	Результаты расчёта . . . . .	80
7.4	Задание для самостоятельной работы . . . . .	82
<b>8</b>	<b>Лекция 8 (28.10)</b>	<b>87</b>
8.1	Метод конечных объёмов . . . . .	87
8.1.1	Уравнение Пуассона . . . . .	87
8.1.1.1	Обработка внутренних граней . . . . .	88
8.1.1.2	Учёт граничных условий . . . . .	89
8.1.2	Одномерный случай . . . . .	90
8.1.3	Сборка системы линейных уравнений . . . . .	91
8.1.3.1	Алгоритм сборки в цикле по ячейкам . . . . .	92
8.1.3.2	Алгоритм сборки в цикле по граням . . . . .	93
8.2	Работа с конечнообъёмной сеткой . . . . .	94
8.2.1	Двумерная сетка . . . . .	94
8.2.1.1	Определение двумерной конечнообъёмной сетки . . . . .	95
8.2.1.2	Вспомогательные таблицы связности . . . . .	95
8.2.1.3	Геометрические свойства сетки . . . . .	96
8.2.2	Трёхмерная сетка . . . . .	97
8.2.2.1	Определение трёхмерной конечнообъёмной сетки . . . . .	97
8.2.2.2	Геометрические свойства сетки . . . . .	98
8.2.3	Интегрирование сеточной функции . . . . .	99
8.3	Пример расчётной программы . . . . .	99
8.3.1	Работа с сеткой . . . . .	100
8.3.2	Функция верхнего уровня . . . . .	100
8.3.3	Инициализация решения . . . . .	101
8.3.4	Реализация решения . . . . .	101
8.3.4.1	Сборка матрицы . . . . .	101
8.3.4.2	Сборка правой части . . . . .	101
8.3.4.3	Вычисление нормы отклонения от точного решения . . . . .	102
8.4	Задание для самостоятельной работы . . . . .	102
<b>9</b>	<b>Лекция 9 (11.11)</b>	<b>105</b>
9.1	Вычисление нормальной производной на скошенных сетках . . . . .	105
9.2	Решение системы Уравнений Навье-Стокса методом конечных объёмов . . . . .	105
9.2.1	Схема SIMPLE . . . . .	105
9.2.2	Вычисление градиента давления методом наименьших квадратов . . . . .	105
9.2.3	Интерполяция Rhie-Chow нормальной компоненты скорости . . . . .	105

9.2.4	Порядок вычисления на итерации . . . . .	105
9.3	Пример расчётной программы. Течение в каверне . . . . .	105
9.4	Задание для самостоятельной работы . . . . .	105
<b>10</b>	<b>Лекция 10 (18.11)</b>	<b>108</b>
10.1	Примеры сборки СЛАУ методом конечных объёмов . . . . .	108
10.2	Нестационарная задача об обтекании кругового цилиндра . . . . .	108
10.3	Задание для самостоятельной работы . . . . .	108
10.3.1	Уравнение для температуры . . . . .	109
10.3.2	Коэффициент теплоотдачи . . . . .	109
10.3.3	Порядок реализации . . . . .	110
<b>11</b>	<b>Лекция 11 (25.11)</b>	<b>112</b>
11.1	Метод взвешенных невязок . . . . .	112
11.2	Метод Бубнова–Галёркина . . . . .	112
11.2.1	Степенные базисные функции . . . . .	112
11.3	Метод конечных элементов . . . . .	112
11.3.1	Узловые базисные функции . . . . .	112
11.3.2	Одномерное уравнение Пуассона . . . . .	112
11.3.2.1	Слабая интегральная постановка задачи . . . . .	112
11.3.2.2	Линейный одномерный базис . . . . .	112
11.3.2.3	Элементные матрицы . . . . .	112
11.3.2.4	Сборка глобальных матриц и векторов . . . . .	112
11.3.3	Двумерное уравнение Пуассона . . . . .	112
11.3.3.1	Треугольный элемент. Линейный двумерный базис . . . . .	112
11.3.3.2	Пример сборки матрицы для двумерной задачи . . . . .	112
<b>12</b>	<b>Лекция 12 (02.12)</b>	<b>113</b>
12.1	Метод конечных элементов . . . . .	113
12.1.1	Четырёхугольный элемент. Билинейный двумерный базис . . . . .	113
12.1.2	Численное интегрирование внутри конечного элемента . . . . .	113
12.2	Разбор программной реализации МКЭ . . . . .	113
12.2.1	Рабочий объект . . . . .	113
12.2.2	Конечноэлементный сборщик . . . . .	113
12.2.3	Концепция конечного элемента . . . . .	114
12.2.3.1	Определение линейного треугольного элемента . . . . .	114
12.2.3.2	Определение линейного элемента на отрезке . . . . .	115
12.2.3.3	Определение билинейного элемента на четырёхугольнике . . . . .	115
12.2.3.4	Геометрические свойства элемента . . . . .	115
12.2.3.5	Элементный базис . . . . .	115
12.2.3.6	Калькулятор элементных матриц . . . . .	116
12.3	Задание для самостоятельной работы . . . . .	116

<b>13 Лекция 13 (09.12)</b>	<b>118</b>
13.1 Узловые элементы высокого порядка точности . . . . .	118
13.2 Эрмитовы элементы . . . . .	118
13.3 Разбор программной реализации МКЭ третьего порядка . . . . .	118
13.4 Задание для самостоятельной работы . . . . .	118
<b>14 Лекция 14 (02.03)</b>	<b>120</b>
14.1 МКЭ решение для системы Навье-Стокса . . . . .	120
<b>15 Лекция 15 (09.03)</b>	<b>121</b>
15.1 TVD-схемы для неструктурированных сеток . . . . .	121
15.2 Задание для самостоятельной работы . . . . .	121
<b>16 Лекция 16 (23.03)</b>	<b>122</b>
16.1 Условие устойчивости . . . . .	122
16.2 FEM-TVD алгоритм . . . . .	122
16.3 Задание для самостоятельной работы . . . . .	122
<b>17 Лекция 17 (30.03)</b>	<b>124</b>
17.1 Метод конечных элементов со стабилизацией . . . . .	124
17.2 Задание для самостоятельной работы . . . . .	124
<b>18 Лекция 18 (06.04)</b>	<b>126</b>
18.1 Стабилизация методом характеристик . . . . .	126
18.1.1 Конечноэлементная процедура . . . . .	126
18.2 Задание для самостоятельной работы . . . . .	127
<b>A Формулы и обозначения</b>	<b>128</b>
A.1 Векторы . . . . .	129
A.1.1 Обозначение . . . . .	129
A.1.2 Набла–нотация . . . . .	129
A.2 Интегрирование . . . . .	131
A.2.1 Формула Гаусса–Остроградского . . . . .	131
A.2.2 Интегрирование по частям . . . . .	131
A.2.3 Численное интегрирование в заданной области . . . . .	132
A.3 Интерполяционные полиномы . . . . .	133
A.3.1 Многочлен Лагранжа . . . . .	133
A.3.1.1 Узловые базисные функции . . . . .	133
A.3.1.2 Интерполяция в параметрическом отрезке . . . . .	134
A.3.1.3 Интерполяция в параметрическом треугольнике . . . . .	137
A.3.1.4 Интерполяция в параметрическом квадрате . . . . .	139
A.4 Геометрические алгоритмы . . . . .	142
A.4.1 Преобразование координат . . . . .	142
A.4.1.1 Матрица Якоби . . . . .	142

A.4.1.2	Дифференцирование в параметрической плоскости . . . . .	143
A.4.1.3	Интегрирование в параметрической плоскости . . . . .	144
A.4.1.4	Двумерное линейное преобразование. Параметрический треугольник .	144
A.4.1.5	Двумерное билинейное преобразование. Параметрический квадрат .	145
A.4.1.6	Трёхмерное линейное преобразование. Параметрический тетраэдр .	145
A.4.2	Свойства многоугольника . . . . .	145
A.4.2.1	Площадь многоугольника . . . . .	145
A.4.2.2	Интеграл по многоугольнику . . . . .	147
A.4.2.3	Центр масс многоугольника . . . . .	147
A.4.3	Свойства многогранника . . . . .	148
A.4.3.1	Объём многогранника . . . . .	148
A.4.3.2	Интеграл по многограннику . . . . .	148
A.4.3.3	Центр масс многогранника . . . . .	148
A.4.4	Поиск многоугольника, содержащего заданную точку . . . . .	148
<b>B</b>	<b>Работа с инфраструктурой проекта CFDCourse</b>	<b>149</b>
B.1	Сборка и запуск . . . . .	150
B.1.1	Сборка проекта CFDCourse . . . . .	150
B.1.1.1	Подготовка . . . . .	150
B.1.1.2	VisualStudio . . . . .	150
B.1.1.3	VSCode . . . . .	152
B.1.2	Запуск конкретного теста . . . . .	153
B.1.3	Сборка релизной версии . . . . .	155
B.2	Git . . . . .	157
B.2.1	Основные команды . . . . .	157
B.2.2	Порядок работы с репозиторием CFDCourse . . . . .	158
B.3	Paraview . . . . .	160
B.3.1	Данные на одномерных сетках . . . . .	160
B.3.2	Изолинии для двумерного поля . . . . .	163
B.3.3	Данные на двумерных сетках в виде поверхности . . . . .	164
B.3.4	Числовых значения в точках и ячейках . . . . .	165
B.3.5	Векторные поля . . . . .	165
B.3.6	Значение функции вдоль линии . . . . .	167
B.4	Hybmesh . . . . .	170
B.4.1	Работа в Windows . . . . .	170
B.4.2	Работа в Linux . . . . .	170

# 1 Лекция 1 (02.09)

## 2 Лекция 2 (09.09)

### 2.1 Уравнение Пуассона

Решение одномерной задачи Пуассона с граничными условиями первого рода методом конечных разностей. Понятие о точности аппроксимации сеточной схемы.

#### 2.1.1 Постановка задачи

Рассматривается одномерное дифференциальное уравнение вида

$$-\frac{\partial^2 u}{\partial x^2} = f(x) \quad (2.1)$$

в области  $x \in [a, b]$  с граничными условиями первого рода

$$\begin{cases} u(a) = u_a, \\ u(b) = u_b. \end{cases} \quad (2.2)$$

Необходимо:

- Запрограммировать расчётную схему для численного решения этого уравнения методом конечных разностей на сетке с постоянным шагом,
- С помощью вычислительных экспериментов подтвердить порядок аппроксимации расчётной схемы.

#### 2.1.2 Метод решения

##### 2.1.2.1 Нахождение численного решения

В области решения  $[a, b]$  введём равномерную сетку из  $N$  ячеек. Шаг сетки будет равен  $h = (b - a)/N$ . Узлы сетки запишем в виде сеточного вектора  $\{x_i\}$  длины  $N + 1$ , где  $i = \overline{0, N}$ . Определим сеточный вектор  $\{u_i\}$  неизвестных, элементы которого определяют значение искомого численного решения в  $i$ -ом узле сетки.

Разностная схема второго порядка для уравнения (2.1) имеет вид

$$\frac{-u_{i-1} + 2u_i - u_{i+1}}{h^2} = f_i, \quad i = \overline{1, N-1}. \quad (2.3)$$

Здесь  $\{f_i\}$  – известный сеточный вектор, определяемый через известную аналитическую функцию  $f(x)$  в правой части уравнения (2.1) как

$$f_i = f(x_i). \quad (2.4)$$

Аппроксимация граничных условий (2.2) первого рода даёт дополнительные сеточные уравнения

для граничных узлов

$$\begin{aligned} u_0 &= u_a, \\ u_N &= u_b \end{aligned} \quad (2.5)$$

Линейные уравнения (2.3), (2.5) составляют систему вида

$$\sum_{j=0}^N A_{ij} u_j = b_i, \quad i = \overline{0, N}$$

с матричными коэффициентами

$$A_{ij} = \begin{cases} 1, & i = 0, j = 0; \\ 2/h^2, & i = \overline{1, N-1}, j = i; \\ -1/h^2, & i = \overline{1, N-1}, j = i-1; \\ -1/h^2, & i = \overline{1, N-1}, j = i+1; \\ 1, & i = N, j = N; \\ 0, & \text{иначе.} \end{cases} \quad (2.6)$$

и правой частью

$$b_i = \begin{cases} u_a, & i = 0; \\ u_b, & i = N; \\ f_i, & i = \overline{1, N-1}. \end{cases} \quad (2.7)$$

Искомый вектор находится путём решения этой системы.

### 2.1.2.2 Практическое определение порядка аппроксимации

Порядок аппроксимации показывает скорость приближения численного решения к точному с уменьшением сетки. Поэтому для подтверждения порядка необходимо

- Знать точное решение,
- Уметь вычислять функционал (норму,  $\|\cdot\|$ ), характеризующий отклонение точного решения от численного,
- Сделать несколько расчётов на сетках с разной  $N$  и заполнить таблицу  $\|\{u_i - u^e(x_i)\}\|(N)$ ,
- На основе этой таблицы построить график в логарифмических осях и по углу наклона кривой сделать вывод о порядке аппроксимации.

Выберем произвольную функцию  $u^e$  (достаточно сильно изменяющуюся на целевом отрезке  $[a, b]$ ).

Далее путём прямого вычисления определим параметры задачи  $f$ ,  $u_a$ ,  $u_b$  такие, для которых функция  $u^e$  является точным решением задачи (2.1), (2.2).

Зададимся числом разбиений  $N$  и решим задачу для выбранным параметров. В результате определим сеточный вектор численного решения  $\{u_i\}$ .

В качестве нормы выберем стандартное отклонение. В интегральном виде для многомерной функции  $y(\mathbf{x})$  в области  $\mathbf{x} \in D$  оно имеет вид

$$\|y(\mathbf{x})\|_2 = \sqrt{\frac{1}{|D|} \int_D y(\mathbf{x})^2 d\mathbf{x}}. \quad (2.8)$$

Упрощая до одномерного случая

$$\|y(x)\|_2 = \sqrt{\frac{1}{b-a} \int_a^b y(x)^2 dx}.$$

Вычислим этот интеграл численно на введённой ранее равномерной сетке  $\{x_i\}$ :

$$\|\{y_i\}\|_2 = \sqrt{\frac{1}{b-a} \sum_{i=0}^N w_i y_i^2},$$

где  $\{w_i\}$  – вес (или "площадь влияния")  $i$ -ого узла:

$$w_i = \begin{cases} h/2, & i = 0, N; \\ h, & i = \overline{1, N-1}, \end{cases}$$

такая что

$$\sum_{i=0}^N w_i = b - a.$$

Окончательно среднеквадратичная норма отклонения численного решения от точного запишется в виде

$$\|\{u_i - u^e(x_i)\}\|_2 = \sqrt{\frac{1}{b-a} \sum_{i=0}^N w_i (u_i - u_i^e)^2}. \quad (2.9)$$

### 2.1.3 Программная реализация

Тестовая программа для решения одномерного уравнения Пуассона реализована в файле `poisson_solve_test.cpp`.

В качестве аналитической тестовой функции используется

$$u^e = \sin(10x^2)$$

на отрезке  $x \in [0, 1]$ .

#### 2.1.3.1 Функция верхнего уровня

объявлена как

```
111 TEST_CASE("Poisson 1D solver", "[poisson1"]){
```

В программе в цикле по набору разбиений `n_cells`

```
123   for (size_t n_cells: {10, 20, 50, 100, 200, 500, 1000}){
```

создаётся решатель для тестовой задачи, использующий заданное число ячеек

```
125     TestPoisson1Worker worker(n_cells);
```

вычисляется среднеквадратичная норма отклонения численного решения от точного

```
128     double n2 = worker.solve();
```

полученное численное решение (вместе с точным) сохраняется в vtk файле

```
poisson1_ncells={10,20,...}.vtk
```

```
131     worker.save_vtk("poisson1_ncells=" + std::to_string(n_cells) + ".vtk");
```

а полученная норма печатается в консоль напротив количества ячеек

```
134     std::cout << n_cells << " " << n2 << std::endl;
```

В результате работы программы в консоли должна отобразиться таблица вида

```
--- cfd24_test [poisson1] ---
10 0.179124
20 0.0407822
50 0.00634718
100 0.00158055
200 0.000394747
500 6.31421e-05
1000 1.57849e-05
```

где первый столбец – это количество ячеек, а второй – полученная для этого количества ячеек норма. Нарисовав график этой таблицы в логарифмических осях подтвердим второй порядок аппроксимации (рис. 1).

Открыв один из сохранённых в процессе работы файлов vtk `poisson1_ncells=? vtk` в paraview можно посмотреть полученные графики. В файле представлены как точное “exact”, так и численное решение “numerical” (рис. 2).

### 2.1.3.2 Детали реализации

Основная работа по решению задачи проводится в классе `TestPoisson1Worker`.

В его конструкторе происходит инициализация сетки (приватного поля класса) на отрезке  $[0, 1]$  с заданным разбиением `n_cells`:

	A	B	C	D	E	F	G	H	I
1	N	Norm	Log10(N)	Log10(Norm)					
2	10	0.179124	1	-0.74684622					
3	20	0.0407822	1.301029996	-1.38952935					
4	50	0.00634718	1.698970004	-2.19741919					
5	100	0.00158055	2	-2.80119176					
6	200	0.000394747	2.301029996	-3.40368116					
7	500	6.31E-05	2.698970004	-4.19968098					
8	1000	1.58E-05	3	-4.80175817					
9									
10									
11									
12									
13									
14									
15									
16									
17									
18									
19									

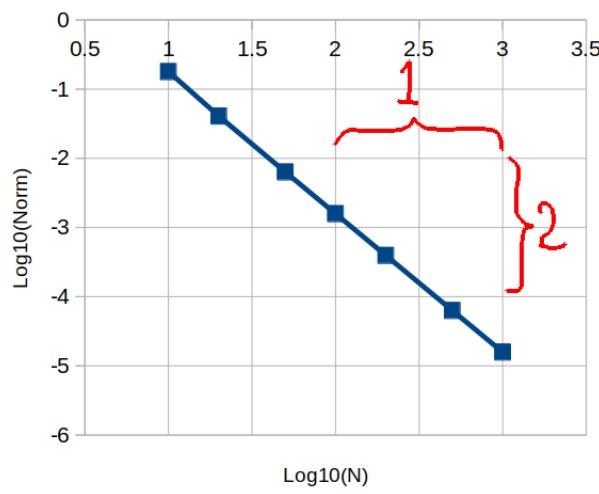


Рис. 1: Сходимость с уменьшением разбиения при решении одномерного уравнения Пуассона

```
14 class TestPoisson1Worker{
```

В методе

`solve()` производится численное решения задачи и вычисления нормы. Для этого последовательно

- Строится матрица левой части и вектор правой части определяющей системы уравнений. Матрицы хранятся в разреженном формате CSR, удобном для последовательного чтения.
- Вызывается решатель СЛАУ. Решение записывается в приватное поле класса `u`.
- Вызывается функция вычисления нормы.

```
29 double solve(){
30     // 1. build SLAE
31     CsrMatrix mat = approximate_lhs();
32     std::vector<double> rhs = approximate_rhs();
33
34     // 2. solve SLAE
35     AmgcMatrixSolver solver;
36     solver.set_matrix(mat);
37     solver.solve(rhs, u);
38
39     // 3. compute norm2
40     return compute_norm2();
41 }
```

Функции нижнего уровня (используемые в методе `solve`):

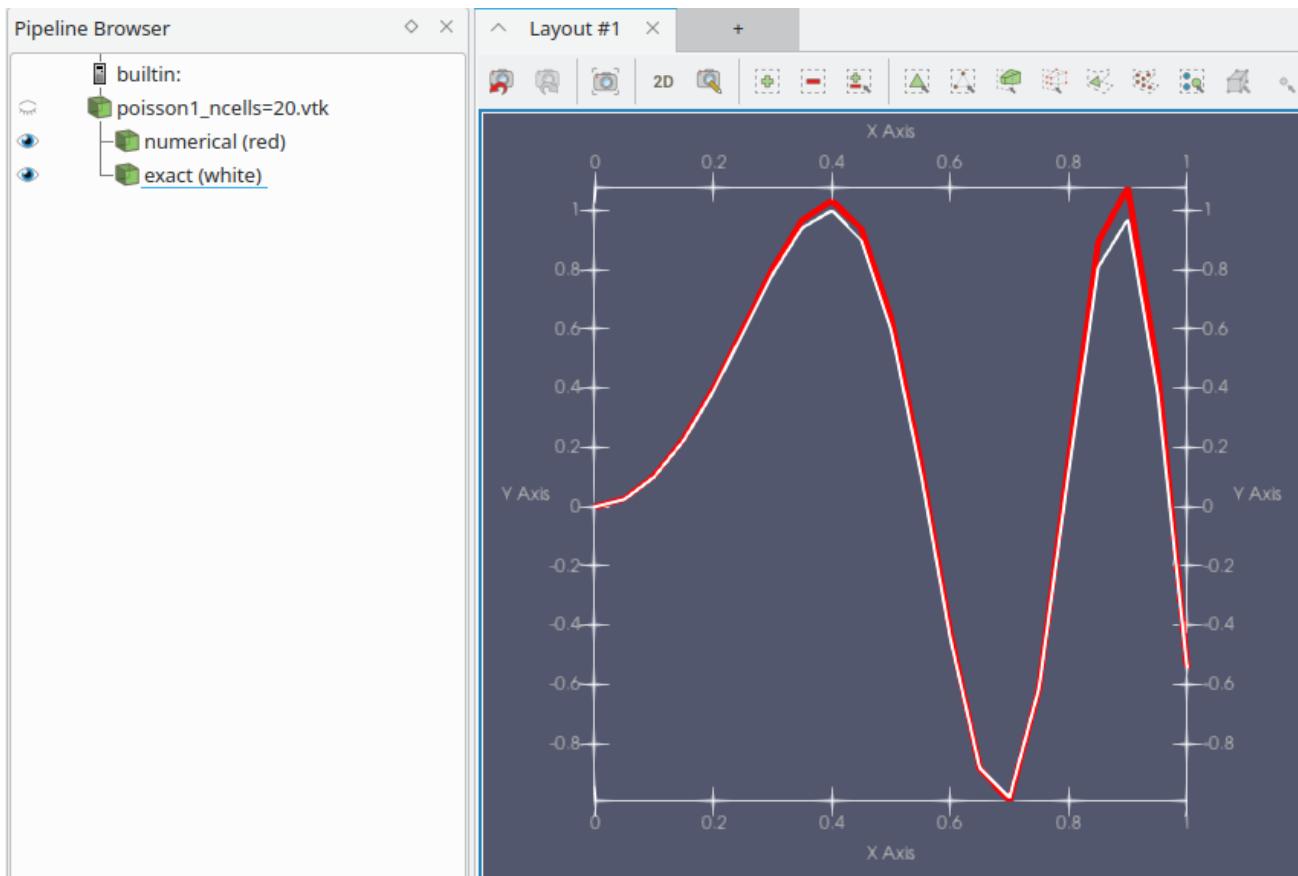


Рис. 2: Сравнение точного и численного решений уравнения Пуассона

- Сборка левой части СЛАУ. Реализует формулу (2.6). Для заполнения матрицы используется формат

`cfd::LodMatrix`, удобный для непоследовательной записи, который в конце конвертируется CSR.

```

63     CsrMatrix approximate_lhs() const{
64         // constant h = x[1] - x[0]
65         double h = grid.point(1).x() - grid.point(0).x();
66
67         // fill using 'easy-to-construct' sparse matrix format
68         LodMatrix mat(grid.n_points());
69         mat.add_value(0, 0, 1);
70         mat.add_value(grid.n_points()-1, grid.n_points()-1, 1);
71         double diag = 2.0/h/h;
72         double nondiag = -1.0/h/h;
73         for (size_t i=1; i<grid.n_points()-1; ++i){
74             mat.add_value(i, i-1, nondiag);
75             mat.add_value(i, i+1, nondiag);
76             mat.add_value(i, i, diag);
77         }
78

```

```

79     // return 'easy-to-use' sparse matrix format
80     return mat.to_csr();
81 }
```

- Сборка правой части СЛАУ. Реализует формулу (2.7).

```

83     std::vector<double> approximate_rhs() const{
84         std::vector<double> ret(grid.n_points());
85         ret[0] = exact_solution(grid.point(0).x());
86         ret[grid.n_points()-1] = exact_solution(grid.point(grid.n_points()-1).x());
87         for (size_t i=1; i<grid.n_points()-1; ++i){
88             ret[i] = exact_rhs(grid.point(i).x());
89         }
90         return ret;
91     }
```

- Вычисление нормы. Реализует формулу (2.9).

```

93     double compute_norm2() const{
94         // weights
95         double h = grid.point(1).x() - grid.point(0).x();
96         std::vector<double> w(grid.n_points(), h);
97         w[0] = w[grid.n_points()-1] = h/2;
98
99         // sum
100        double sum = 0;
101        for (size_t i=0; i<grid.n_points(); ++i){
102            double diff = u[i] - exact_solution(grid.point(i).x());
103            sum += w[i]*diff*diff;
104        }
105
106        double len = grid.point(grid.n_points()-1).x() - grid.point(0).x();
107        return std::sqrt(sum / len);
108    }
```

## 2.2 Задание для самостоятельной работы

### 2.2.1 Порядок сходимости

Построить график, подтверждающий второй порядок точности разностной схемы (2.3).

### 2.2.2 Двумерное уравнение Пуассона

Написать тест, аналогичный [poisson1], но для двумерной задачи на двумерной регулярной сетке

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = f(x, y).$$

использовать разностную схему

$$\frac{-u_{k[i-1,j]} + 2u_{k[i,j]} - u_{k[i+1,j]}}{h_x^2} + \frac{-u_{k[i,j-1]} + 2u_{k[i,j]} - u_{k[i,j+1]}}{h_y^2} = f_{k[i,j]},$$

где

$$k[i, j] = i + (n_x + 1)j \quad (2.10)$$

– функция, переводящая парный  $(i, j)$  индекс узла регулярной сетки ( $i$ ) для оси x,  $j$  для оси y) в сквозной индекс  $k$  сеточного вектора,  $n_x$  – количество ячеек сетки в направлении x.

При вычислении весов  $w_k$  для вычисления среднеквадратичного отклонения учесть наличие граничных и угловых точек:

$$w_k = \begin{cases} h_x h_y / 4, & \text{для угловых точек;} \\ h_x h_y / 2, & \text{для граничных неугловых точек;} \\ h_x h_y, & \text{для внутренних точек.} \end{cases}$$

Четыре угловые точки определяются как

$$i[k], j[k] = (0, 0), (0, n_y), (n_x, n_y), (n_x, 0)$$

Граничные неугловые точки:

$$\begin{aligned} i[k], j[k] = & \overline{1, n_x - 1}, 0; & \text{нижняя сторона,} \\ & \overline{n_x, 1, n_y - 1}; & \text{правая сторона,} \\ & \overline{1, n_x - 1, n_y}; & \text{верхняя сторона,} \\ & \overline{0, 1, n_y - 1}; & \text{левая сторона.} \end{aligned}$$

Функции, переводящие сквозной индекс в пару  $i, j$ , имеют вид

$$\begin{aligned} i[k] &= \text{mod}(k, (n_x + 1)), && // \text{ остаток от деления,} \\ j[k] &= \lfloor k / (n_x + 1) \rfloor, && // \text{ целая часть от деления.} \end{aligned} \quad (2.11)$$

Использовать класс `cfd::RegularGrid2D` для задания сетки. Функции перевода индексов узлов из сквозных в парные и обратно реализованы в классе двумерной регулярной сетки:

- `cfd::RegularGrid2D::to_split_point_index`
- `cfd::RegularGrid2D::to_linear_point_index`

В случае, если решатель системы линейных уравнений не решает построенную матрицу, использовать функцию `cfd::dbg::print` для отлажочной печати матрицы в консоль (размерность задачи должна быть небольшой).

### 3 Лекция 3 (16.09)

#### 3.1 Двухслойные схемы для нестационарных уравнений

##### 3.1.1 Определение

Рассмотрим дифференциальное уравнение вида

$$\frac{\partial u}{\partial t} + Lu = f, \quad (3.1)$$

где  $L$  – произвольный пространственный дифференциальный оператор. При использовании двухслойной схемы аппроксимации производная по времени записывается в виде конечной разности с шагом  $\tau$ , которая может приближать производную в одном из трёх моментов времени:

$$\begin{aligned} \frac{u(t + \tau) - u(t)}{\tau} &= \left. \frac{\partial u}{\partial t} \right|_t + o(\tau) && \text{– разность вперёд;} \\ \frac{u(t) - u(t + \tau)}{\tau} &= \left. \frac{\partial u}{\partial t} \right|_{t+\tau} + o(\tau) && \text{– разность назад;} \\ \frac{u(t + \tau) - u(t - \tau)}{2\tau} &= \left. \frac{\partial u}{\partial t} \right|_{t+\frac{\tau}{2}} + o(\tau^2) && \text{– симметричная разность.} \end{aligned} \quad (3.2)$$

Момент времени  $t$  будем называть текущим временем`ы`**м** слоем, момент  $t + \tau$  – следующим, а момент  $t + \tau/2$  – промежуточным. Считается, что значение функции на текущий момент времени  $u(t)$  известно, а значение на следующий момент  $u(t + \tau)$  подлежит определению.

##### 3.1.1.1 Явная схема

При использовании разности назад уравнение (3.1) в полудискретизированном (то есть дискретизованном только по времени, но не по пространству) виде запишется как

$$\frac{u(x, t + \tau) - u(x)}{\tau} + Lu(x, t) = f(x, t)$$

или, после переноса всех известных слагаемых вправо

$$u(x, t + \tau) = (E - \tau L) u(x, t) + \tau f(x, t). \quad (3.3)$$

Здесь  $E$  – единичный оператор. Схема (3.3) называется явной схемой и имеет первый порядок точности.

##### 3.1.1.2 Неявная схема

Выбрав разность назад из выражения (3.2) полудискретизированная схема для уравнения (3.1) примет вид

$$\frac{u(x, t + \tau) - u(x)}{\tau} + Lu(x, t + \tau) = f(x, t + \tau).$$

В результате преобразования получим неявную схему первого порядка точности

$$(E + \tau L) u(x, t + \tau) = u(x, t) + \tau f(x, t + \tau). \quad (3.4)$$

### 3.1.1.3 Схема Кранка–Николсон

Подставим симметричную разность из (3.2) в уравнение (3.1). Формально получим

$$\frac{u(x, t + \tau) - u(x)}{\tau} + Lu(x, t + \frac{\tau}{2}) = f(x, t + \frac{\tau}{2}).$$

Для определения выражения функций на промежуточном временном слое распишем значение  $u$  на текущем и следующем слоях в ряд Тейлора относительно значения на момент  $t + \tau/2$ :

$$\begin{aligned} u(t) &= u\left(t + \frac{\tau}{2}\right) - \frac{\tau}{2} \frac{\partial u}{\partial t}\Big|_{t+\frac{\tau}{2}} + o(\tau^2) \\ u(t + \tau) &= u\left(t + \frac{\tau}{2}\right) + \frac{\tau}{2} \frac{\partial u}{\partial t}\Big|_{t+\frac{\tau}{2}} + o(\tau^2) \end{aligned}$$

Взяв полусумму этих выражений получим аппроксимацию функции на промежуточном слое:

$$u\left(x, t + \frac{\tau}{2}\right) = \frac{1}{2}u(x, t) + \frac{1}{2}u(x, t + \tau) + o(\tau^2) \quad (3.5)$$

Аналогичная запись справедлива и для свободного члена  $f$ . Если оператор  $L$  – нестационарный или нелинейный, то аппроксимацию (3.5) следует записывать для всего выражения  $Lu$ :

$$(Lu)_{t+\frac{\tau}{2}} = \frac{1}{2}(Lu)_t + \frac{1}{2}(Lu)_{t+\tau} + o(\tau^2)$$

С учётом (3.5) симметричная разностная схема запишется как

$$\frac{u(x, t + \tau) - u(x)}{\tau} + \frac{1}{2}Lu(x, t) + \frac{1}{2}Lu(x, t + \tau) = \frac{1}{2}f(x, t) + \frac{1}{2}f(x, t + \tau)$$

или

$$\left(E + \frac{\tau}{2}L\right)u(x, t + \tau) = \left(E - \frac{\tau}{2}L\right)u(x, t) + \frac{\tau}{2}(f(x, t) + f(x, t + \tau)). \quad (3.6)$$

Такая схема называется схемой Кранка–Николсон и имеет второй порядок аппроксимации по времени.

В случае, если оператор  $L$  зависит от времени, то в левой части схемы (3.6) его нужно брать на следующем временном слое, а в правой – на текущем.

### 3.1.1.4 Обобщённая двухслойная схема

Выражения (3.3), (3.4), (3.6) можно записать в обобщённой форме

$$(E + \theta\tau L)u(x, t + \tau) = (E + (\theta - 1)\tau L)u(x, t) + (1 - \theta)f(x, t) + \theta f(x, t + \tau). \quad (3.7)$$

Коэффициент  $\theta$  – степень неявности схемы:

- $\theta = 0$  – явная схема (3.3),
- $\theta = 1$  – полностью неявная схема (3.4),
- $\theta = 1/2$  – схема Кранка–Николсон (3.6).

Отметим, что только при  $\theta = 1/2$  схема (3.7) имеет второй порядок точности по времени. Для других значений (в том числе промежуточных) схема будет иметь ошибку первого порядка  $o(\tau)$ .

### 3.1.2 Дискретизация по времени как итерационный процесс

#### 3.1.2.1 Двухслойный итерационный процесс

Простой двухслойный итерационный процесс определяется как

$$u^{n+1} = Au^n + b, \quad (3.8)$$

где  $n$  – индекс итерационного слоя,  $A$  – оператор преобразования,  $b$  – свободный член.

Определение значения функции на следующий момент времени  $u(t + \tau)$  по двухслойной схеме (3.7) можно представить как простой итерационный процесс (3.8), где

$$A = (E + \theta\tau L)^{-1} (E + (\theta - 1)\tau L),$$

$$b = (E + \theta\tau L)^{-1} (\theta f(x, t + \tau) + (1 - \theta) f(x, t)).$$

Итерационный процесс называется сходящимся, если

$$\lim_{n \rightarrow \infty} \|u^{n+1} - u^n\| = 0.$$

#### 3.1.2.2 Устойчивость итерационного процесса

Рассмотрим два простых итерационных процесса, имеющих на нулевом слое значение  $u^0 = 1$ :

$$\begin{aligned} \text{(I)} : \quad & u^{n+1} = 2u^n - 1, \\ \text{(II)} : \quad & u^{n+1} = 0.5u^n + 0.5. \end{aligned}$$

Оба этих процесса при выбранном начальном приближении, очевидно, сходятся. На каждой итерации справедливо  $u^n = 1$ . Возмутим начальное условие: пусть

$$u^0 = 1 + \varepsilon,$$

и проведём итерации.

	(I)	(II)
$u^1$	$1 + 2\varepsilon$	$1 + \frac{\varepsilon}{2}$
$u^2$	$1 + 4\varepsilon$	$1 + \frac{\varepsilon}{4}$
$u^3$	$1 + 8\varepsilon$	$1 + \frac{\varepsilon}{8}$
...		
$u^\infty$	$\infty$	1

Видно, что процесс (I) теряет сходимость и стремится к бесконечности, в то время, как процесс (II) сохраняет свои свойства.

Свойство итерационных процессов уменьшать малые возмущения называется устойчивостью. В примере выше процесс (I) является неустойчивым, а процесс (II) – устойчивым.

Нетрудно видеть, что для рассматриваемого скалярного итерационного процесса, условие устойчивости запишется в виде  $|A| \leq 1$ .

### 3.1.2.3 Источники возмущений

На практике возникновение возмущений в решениях неизбежно: они могут быть следствием ошибок дискретизации функций и операторов, погрешностей решения СЛАУ, ошибок при проведении арифметических операций на числах с плавающей точкой и т.д. Поэтому любой итерационный процесс, используемый для решений математических задач, должен быть устойчив.

Возникновение непреднамеренных ошибок вследствие компьютерного округления можно проиллюстрировать на примере программы, в которой рассматривается сходящийся для любого начального условия, но неустойчивый итерационный процесс

$$u^{n+1} = 10u^n - 9u^0.$$

```
double u0 = 0.625;
double u = u0;
for (int i=0; i<1000; ++i){
    u = 10*u - 9*u0;
}
std::cout << u << std::endl;
```

Если начальное значение может быть точно представлено в числах с плавающей точкой (путём конечной суммы степеней двойки), то арифметическая ошибка не возникает. Так, представленный выше код на выходе печатает ожидаемое  $u = 0.625$ . Потому что начальное приближение может быть разложено как  $u^0 = 2^{-1} + 2^{-3}$ .

Однако, если заменить начальное приближение на любое число, которое не может быть записано

точно во floating-point формате, то процесс быстро уходит в бесконечность. Например, для  $u^0 = 0.626$  бесконечные (непредставимые в машинном формате) значения появляются на 324-ой итерации, а при переключении на работу в числах одинарной точности ‘float’ – уже на 46-ой.

## 3.2 Методы исследования устойчивости расчётных схем

### 3.2.1 Матричный метод

Итерационные процессы, возникающие при численном решении дифференциальных уравнений се-точными методами, имеют матричную природу. То есть оператор преобразования  $A$  в выражении (3.8) – это матрица, а функции  $u$  и  $b$  – векторы-столбцы.

Как было показано выше, условием устойчивости скалярного итерационного процесса является неравенство  $|A| \leq 1$ . Аналогом этого условия для матричного процесса является ограничение на спектральный радиус  $S(A)$ :

$$S(A) = \max_j |\lambda_j| \leq 1, \quad (3.9)$$

где  $\lambda_j$  – собственные числа матрицы  $A$ .

Для некоторых видов матриц, возникающих при аппроксимации простейших дифференциальных уравнений, собственные числа известны.

#### 3.2.1.1 Явная схема для нестационарного уравнения диффузии

Например, рассмотрим одномерное нестационарное уравнение диффузии с граничными условиями первого рода

$$\begin{aligned} \frac{\partial u}{\partial t} &= k \frac{\partial^2 u}{\partial x^2}, \\ u(x, 0) &= u_0(x), \\ u(x_a, t) &= u_a, \\ u(x_b, t) &= u_b. \end{aligned}$$

Используем явную дискретизацию по времени и аппроксимацию второго порядка по пространству. Тогда разностная схема запишется в виде:

$$\hat{u}_i = u_i + \gamma(u_{i-1} - 2u_i + u_{i+1}), \quad i = \overline{1, N-1}, \quad (3.10)$$

где введено обозначение для значения функции на следующем временном слое  $\hat{u} = u(t + \tau)$  и  $\gamma = \tau k/h^2$ . В матричном виде схема имеет вид

$$\hat{u} = Au, \quad A = \begin{pmatrix} 1 & 0 & & & \\ \gamma & 1 - 2\gamma & \gamma & & \\ & \gamma & 1 - 2\gamma & \gamma & \\ & & \ddots & \ddots & \ddots \\ & & & \gamma & 1 - 2\gamma & \gamma \\ & & & & 0 & 1 \end{pmatrix}.$$

Первая и последняя строки этой матрицы – следствие учёта граничных условий первого рода.

Собственные числа для полученной трёхдиагональной матрицы преобразования в правой части имеют вид

$$\lambda_j = 1 - 4\gamma \sin^2 \left( \frac{j\pi}{2N} \right), \quad j = \overline{1, N-1}$$

Тогда, исходя из выражения (3.9), запишем условие устойчивости для явной схемы (3.10)

$$\gamma \leq \frac{1}{2}$$

### 3.2.1.2 Неявная схема для нестационарного уравнения диффузии

Аналогично, рассмотрим неявную схему

$$\hat{u}_i - \gamma(\hat{u}_{i-1} - 2\hat{u}_i + \hat{u}_{i+1}) = u_i, \quad i = \overline{1, N-1}, \quad (3.11)$$

В матричном виде

$$\hat{u} = A^{-1}u, \quad A = \begin{pmatrix} 1 & 0 & & & \\ -\gamma & 1 + 2\gamma & -\gamma & & \\ & -\gamma & 1 + 2\gamma & -\gamma & \\ & & \ddots & \ddots & \ddots \\ & & & -\gamma & 1 + 2\gamma & -\gamma \\ & & & & 0 & 1 \end{pmatrix}.$$

Собственные числа такой матрицы имеют вид

$$\lambda_j = 1 + 4\gamma \sin^2 \left( \frac{j\pi}{2N} \right), \quad j = \overline{1, N-1}$$

Поскольку в правой части итерационного процесса используется матрица, обратная к  $A$ , а собственные числа обратных матриц равны  $1/\lambda_j$ , то условие устойчивости примет вид

$$\lambda_j \geq 1.$$

Очевидно, что оно выполняется всегда. Поэтому неявная схема (3.11) безусловно устойчива.

### 3.2.2 Метод дискретных возмущений

Метод дискретных возмущений заключается в использовании в качестве начального приближения нулевого вектора, с возмущением  $\varepsilon$  в одном из узлов:

$$u^0 = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \varepsilon \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

и дальнейшем анализом распространения этого возмущения с прохождением по временным слоям. Во многом этот метод аналогичен тому алгоритму, по которому мы иллюстрировали устойчивость простейшего скалярного итерационного процесса (3.1.2.2).

#### 3.2.2.1 Явная схема против потока для уравнения переноса

Для иллюстрации рассмотрим одномерное уравнение переноса

$$\frac{\partial u}{\partial t} + V \frac{\partial u}{\partial x} = 0 \quad (3.12)$$

и явную противопоточную схему для него (при условии  $V > 0$ )

$$\hat{u}_i = u_i - C(u_i - u_{i-1}), \quad (3.13)$$

где число Куранта определено как  $C = \tau V / h$ .

Пусть  $u_i = \varepsilon$ . Тогда

$$\begin{aligned} \hat{u}_i &= (1 - C)\varepsilon & \Rightarrow & \quad 0 \leq C \leq 2, \\ \hat{u}_{i+1} &= C\varepsilon & \Rightarrow & \quad -1 \leq C \leq 1. \end{aligned}$$

Поскольку  $C$  по определению больше нуля, то условием устойчивости для схемы (3.13) будет выражение

$$C \leq 1.$$

### 3.2.3 Метод Неймана

Запишем обратное преобразование Фурье для функции  $u(x)$ :

$$u(x) = \int v(\kappa) e^{i\kappa x} d\kappa,$$

$\kappa$  – волновое число,  $i$  – мнимая единица,  $v(\kappa)$  – Фурье образ исходной функции.

Зададим такое начальное возмущение, которое имеет единичную амплитуду на одной частоте, соответствующей волновому числу  $\kappa_0$ :

$$v(\kappa) = \delta(\kappa - \kappa_0),$$

$\delta(x)$  – функция Дирака. Кроме того, учтём, что  $x_i = ih$ . Тогда выбранное начальное возмущение на одной выбранной частоте, взятое в  $i$ -ом узле, примет вид

$$u_i = e^{\mathbf{i}i\theta}, \quad \theta = \kappa_0 h \quad (3.14)$$

На следующем временном шаге это возмущение примет вид:

$$\hat{u}_i = Ge^{\mathbf{i}i\theta}. \quad (3.15)$$

$G$  – коэффициент усиления. Он показывает во сколько раз увеличилась амплитуда выбранного возмущения за один шаг по времени. Для того, чтобы все возмущения затухали, необходимо

$$|G| \leq 1, \quad \forall \theta$$

### 3.2.3.1 Неявная противотоковая схема для уравнения переноса

Для примера анализа устойчивости методом Неймана опять рассмотрим задачу (3.12), но на этот раз рассмотрим чисто неявную аппроксимацию

$$\hat{u}_i + C(\hat{u}_i - \hat{u}_{i-1}) = u_i. \quad (3.16)$$

Подставим (3.14), (3.15)

$$Ge(i) + C(Ge(i) - Ge(i-1)) = e(i).$$

где для краткости введено обозначение

$$e(i) = e^{\mathbf{i}\theta i}.$$

Поделим на  $e(i)$  с использованием свойств этой степенной функции. Тогда

$$G + CG(1 - e(-1)) = 1 \quad \Rightarrow$$

$$G = (1 + C(1 - e(-1)))^{-1}.$$

По определению комплексной экспоненты имеем

$$e(-1) = \cos \theta - \mathbf{i} \sin \theta.$$

Требуется показать, что  $|G| \leq 1$ . Отсюда

$$\begin{aligned} |1 + C(1 - \cos \theta + i \sin \theta)| &\geq 1 \quad \Rightarrow \\ |1 + C(1 - \cos \theta) + C i \sin \theta|^2 &\geq 1 \quad \Rightarrow \\ 1 + C^2(1 - \cos \theta)^2 + 2C(1 - \cos \theta) + C^2 \sin^2 \theta &\geq 1 \quad \Rightarrow \\ C^2(1 - \cos \theta) + 2C + C^2(1 + \cos \theta) &\geq 0 \quad \Rightarrow \\ C^2 + 2C &\geq 0. \end{aligned}$$

По определению число Куранта больше 0, поэтому последнее выражение выполняется всегда. Отсюда следует вывод, что неявная разностная схема вида (3.16) безусловно устойчива.

### 3.2.3.2 Противопотоковая схема Кранка-Николсон для уравнения переноса

Для того же самого уравнения (3.12) рассмотрим схему Кранка-Николсон (3.5):

$$\hat{u}_i + \frac{C}{2} (\hat{u}_i - \hat{u}_{i-1}) = u_i - \frac{C}{2} (u_i - u_{i-1}). \quad (3.17)$$

Так же подставим (3.14), (3.15) и поделим на  $e(i)$ :

$$\begin{aligned} G + \frac{CG}{2} (1 - e(-1)) &= 1 - \frac{C}{2} (1 - e(-1)) \quad \Rightarrow \\ G = \frac{1-p}{1+p}, \quad p &= \frac{C}{2} (1 - e(-1)). \end{aligned}$$

Для выполнения условия устойчивости  $|G| \leq 1$ , необходимо

$$\begin{aligned} |1 - p|^2 &\leq |1 + p|^2 \quad \Rightarrow \\ (1 - \Re(p))^2 + \Im(p)^2 &\leq (1 + \Re(p))^2 + \Im(p)^2 \quad \Rightarrow \\ \Re(p) &\geq 0 \end{aligned}$$

Здесь  $\Re(p)$ ,  $\Im(p)$  – действительная и мнимая часть комплексного числа.

Поскольку число Куранта больше нуля, то и действительная часть выражения  $p$  неотрицательная для любого  $\theta$ .

$$\Re(p) = \frac{C}{2} (1 - \cos \theta) \geq 0.$$

Получаем, что схема видеа (3.17) безусловно устойчива.

### 3.2.3.3 Явная схема для уравнения нестационарной конвекции-диффузии

Рассмотрим уравнение конвекции-диффузии

$$\frac{\partial u}{\partial t} + V \frac{\partial u}{\partial x} = k \frac{\partial^2 u}{\partial x^2} \quad (3.18)$$

Сначала напишем чисто явную схему второго порядка по пространству:

$$\frac{\hat{u}_i - u_i}{\tau} + V \frac{u_{i+1} - u_{i-1}}{2h} = k \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} \quad (3.19)$$

Введем число Куранта  $C = V\tau/h$  и параметр  $\gamma = k\tau/h^2$ . Тогда

$$\hat{u}_i = \left( \gamma - \frac{C}{2} \right) u_{i+1} + \left( \gamma + \frac{C}{2} \right) u_{i-1} + (1 - 2\gamma) u_i.$$

Далее подставим (3.14), (3.15)

$$Ge(i) = \left( \gamma - \frac{C}{2} \right) e(i+1) + \left( \gamma + \frac{C}{2} \right) e(i-1) + (1 - 2\gamma) e(i).$$

Поделим на  $e(i)$  с использованием свойств этой степенной функции. Тогда

$$G = \gamma(e(1) + e(-1)) - \frac{C}{2}(e(1) - e(-1)) + (1 - 2\gamma)$$

По определению комплексной экспоненты имеем

$$\begin{aligned} e(1) &= \cos \theta + i \sin \theta, \\ e(-1) &= \cos \theta - i \sin \theta, \end{aligned}$$

Отсюда

$$G = 2\gamma \cos \theta - iC \sin \theta + (1 - 2\gamma)$$

Запишем квадрат модуля комплексного числа  $G$ :

$$\begin{aligned} |G|^2 &= (1 - 2\gamma(1 - \cos \theta))^2 + C^2 \sin^2 \theta = \\ &= 1 + 4\gamma^2(1 - \cos \theta)^2 - 4\gamma(1 - \cos \theta) + C^2(1 - \cos^2 \theta). \end{aligned}$$

Требование  $|G| \leq 1$  эквивалентно  $|G|^2 \leq 1$ , или

$$\begin{aligned} 1 + 4\gamma^2(1 - \cos \theta)^2 - 4\gamma(1 - \cos \theta) + C^2(1 - \cos^2 \theta) &\leq 1 \quad \Rightarrow \\ 4\gamma^2(1 - \cos \theta)^2 - 4\gamma(1 - \cos \theta) + C^2(1 - \cos^2 \theta) &\leq 0 \quad \Rightarrow \\ 4\gamma^2(1 - \cos \theta) - 4\gamma + C^2(1 + \cos \theta) &\leq 0 \quad \Rightarrow \\ (C^2 - 4\gamma^2) \cos \theta + 4\gamma^2 - 4\gamma + C^2 &\leq 0 \end{aligned}$$

Поскольку неравенство должно выполняться для всех  $\theta$ , а полученное выражение линейно за-

висит от  $\cos \theta$ , то будет достаточно рассмотреть два экстремальных значения косинуса, из которых окончательно запишем два условия устойчивости для явной дискретизации уравнения конвекции-диффузии вида (3.19):

$$\begin{aligned}\cos \theta = 1 &\Rightarrow C \leq \sqrt{2\gamma}, \\ \cos \theta = -1 &\Rightarrow \gamma \leq 1/2.\end{aligned}\quad (3.20)$$

Обычно вместо первого из условий (3.20) применяют более жёсткое (в случае  $2\gamma < 1$ ) условие

$$C \leq 2\gamma,$$

которое с учётом определений сводится к условию на шаг по пространству, формулируемому в терминах сеточного числа Рейнольдса  $\text{Re}_c$ :

$$\frac{Vh}{k} \equiv \text{Re}_c \leq 2.$$

### 3.2.3.4 Неявная схема для уравнения нестационарной конвекции-диффузии

Аналогичным образом рассмотрим неявную диффузии схему для уравнения (3.18) вида

$$\frac{\hat{u}_i - u_i}{\tau} + V \frac{u_{i+1} - u_{i-1}}{2h} = k \frac{\hat{u}_{i+1} - 2\hat{u}_i + \hat{u}_{i-1}}{h^2} \quad (3.21)$$

Подставляя представление для возмущения с волновым числом  $\theta$ , получим

$$G = \frac{1 - iC \sin \theta}{1 - 2\gamma(\cos \theta - 1)}$$

Для устойчивости необходимо

$$\begin{aligned}|1 - iC \sin \theta|^2 &\leq |1 - 2\gamma(\cos \theta - 1)|^2 \quad \Rightarrow \\ 1 + C^2 \sin^2 \theta &\leq 1 + 4\gamma^2(1 - \cos \theta)^2 + 4\gamma(1 - \cos \theta) \quad \Rightarrow \\ C^2(1 + \cos \theta) &\leq 4\gamma^2(1 - \cos \theta) + 4\gamma \quad \Rightarrow \\ \cos \theta(C^2 + 4\gamma^2) + C^2 - 4\gamma - 4\gamma^2 &\leq 0\end{aligned}$$

Наибольшего значения выражение слева достигает при  $\theta = 0$ . Тогда единственное условие устойчивости примет вид

$$C \leq \sqrt{2\gamma}.$$

### 3.2.4 Общие рекомендации к выбору устойчивых расчётных схем

Теоретический анализ условий устойчивости возможен лишь для простейших уравнений с постоянными шагами дискретизации. В практических приложениях, имеющих дело, как правило, с неструктурированными сетками и сложными нелинейными системами уравнений, параметры устойчивого счёта приходится определять эмпирически. Однако, такой теоретический анализ позволяет выделить принципы, которыми следует руководствоваться для построения устойчивых схем.

## Неявные схемы более устойчивы, чем явные

- Это можно видеть, сравнив результаты анализа для безусловно устойчивой неявной схемы (3.2.1.2) для уравнения диффузного переноса и для условно устойчивой явной схемы (3.2.1.1).
- Для уравнения переноса с разностью против потока явная (3.2.2.1) схема условно устойчива, в то время как неявная (3.2.3.1) – устойчива безусловно.
- Даже если только часть схемы неявная, это повышает устойчивость. Так, явная (3.2.3.3) схема для уравнения конвекции-диффузии имеет два условия устойчивости, в то время как схема, неявная по диффузии (3.2.3.4) – только одно.
- Аналогично, явная (3.2.2.1) схема против потока для уравнения переноса условно устойчива, а схема Кранка-Николсон (3.2.3.2) для того же уравнения устойчива при любых параметрах.

**Конвективное слагаемое провоцирует неустойчивость, а диффузионное – напротив, добавляет устойчивость**

- Так, схемы с центральными разностями для уравнения конвекции-диффузии (и явная (3.2.3.3), и полунеявная (3.2.3.4)), условно устойчивы. Явная схема с центральными разностями для чистого уравнения переноса всегда неустойчива. В последнем можно убедиться, подставив  $k = 0$  в условия устойчивости для уравнений конвекции-диффузии.

## 3.3 Программная реализация схемы для уравнения переноса

### 3.3.1 Постановка задачи

Рассматриваются три схемы по времени для противопотоковой аппроксимации уравнения переноса (3.12):

- явная схема (3.13) (тест называется `[transport1-explicit]`),
- неявная схема (3.16) (`[transport1-implicit]`),
- схема Кранка–Николсон (3.17) (`[tranport1-cn]`).

Уравнение решается на отрезке  $x \in [0, 1]$  с единичной скоростью  $V = 1$  на сетке из 1000 ячеек.

Временные итерации продолжаются до момента времени  $t = 0.5$ .

Начальным условием является функция вида

$$u(x, 0) = e^{-x^2/\sigma^2}, \quad \sigma = 0.1$$

Точное решение уравнения, с которого будут сниматься граничные условия и производится сравнения полученного численного решения, запишется как

$$u(x, t) = u(x - t, 0) = e^{-(x-t)^2/\sigma^2}.$$

На каждом шаге по времени функция сохраняется в vtk-формате. В конце выводится значение отклонения от точного решения на конечный момент времени.

В качестве цели решения обозначим построение решения и визуальное сравнение решений при числе  $C = 0.9$  по трём разным схемам. А также построение графика сходимости отклонения точного решения от численного при изменении числа Куранта и фиксированном шаге по пространству (то есть сходимость при уменьшении шага по времени).

Программы реализованы в файле `transport_solve_test.cpp`.

### 3.3.2 Функция верхнего уровня

Для всех трёх программ функция верхнего уровня имеет один и тот же вид. Рассмотрим на примере первой из них:

```
95 TEST_CASE("Transport 1D solver, explicit", "[transport1-explicit]"){
```

В начале происходит установка параметров численной схемы:

- конечного момента времени,
- скорости переноса,
- длины расчётной области,
- разбиения по пространству,
- числа Куранта

```
98 const double tend = 0.5;
99 const double V = 1.0;
100 const double L = 1.0;
101 size_t n_cells = 100;
102 double Cu = 0.9;
```

Далее вычисляются используемые шаги:

- шаг по пространству (из длины области и разбиения),
- шаг по времени (из шага по пространству и числа Куранта)

```
103 double h = L/n_cells;
104 double tau = Cu * h / V;
```

Потом устанавливается рабочий класс, в котором будет производится решение

```
107 TestTransport1WorkerExplicit worker(n_cells);
```

Конструируется класс, используемый для связного сохранения полей на разные моменты времени. Этот класс создаёт `transport1-explicit.vtk.series` со списком всех сохранённых полей и отнесёнными к ним моментами времени, который можно впоследствии открыть в Paraview и использовать функции анимации для воспроизведения поведения решения во времени.

```
110 VtkUtils::TimeSeriesWriter writer("transport1-explicit");
```

Далее нужно в этот класс сохранить решение на начальный момент времени. Для этого туда сначала добавляется запись о нулевом моменте времени

```
111 std::string out_filename = writer.add(0);
```

В переменную

`out_filename` записывается конкретное имя vtk-файла, куда следует сохранить решение. Уже это имя используется для сохранения решения на текущий (начальный) момент времени.

```
112 worker.save_vtk(out_filename);
```

Далее начинается цикл по времени, продолжающийся до тех пор, пока внутреннее время решателя не достигнет конечного

```
115 while (worker.current_time() < tend - 1e-6) {
```

Внутри вызывается функция решения, которая продвигает внутреннее время решателя на  $\tau$ , обновляет актуальное состояние вектора решения и возвращает текущую норму.

```
117 norm = worker.step(tau);
```

Потом повторяется процедура сохранения текущего состояния решателя

```
119 out_filename = writer.add(worker.current_time());  
120 worker.save_vtk(out_filename);
```

После завершения цикла в консоль печатается установленное разбиение по времени и полученное отклонение от точного решения на конечный момент времени

```
122     std::cout << 1.0/tau << " " << norm << std::endl;
```

### 3.3.3 Расчётные функции

Три класса-решателя для трёх заявленных задач:

`TestTransport1WorkerExplicit`, `TestTransport1WorkerImplicit`,

`TestTransport1WorkerCN` наследуются от одного абстрактного класса

`ATestTransport1Worker`. В этом абстрактном классе реализованы все общие для всех решателей функции: создание сетки, сохранение в vtk, расчёт нормы, продвижение по времени.

Этот класс также хранит в себе параметры, полностью определяющие текущее состояние решения:

- расчётную сетку,
- шаг по времени (это параметр, производный от сетки, он сохранён в отдельное поле для удобства расчётов),
- вектор решения на текущий момент,
- текущее время.

Эти поля хранятся в ‘protected’ секции, таким образом все производные классы имеют к этим полям полный доступ.

```
67 protected:  
68     Grid1D _grid;  
69     double _h;  
70     std::vector<double> _u;  
71     double _time = 0;
```

Функция решения, также реализована в абстрактном классе. Она продвигает текущее время и вызывает виртуальный метод `impl_step`, который изменяет значение вектора решения, а в конце вызывает функцию вычисления ошибки.

```
27     double step(double tau){  
28         _time += tau;  
29         impl_step(tau);  
30         return compute_norm2();  
31     }
```

Функция `impl_step` уже зависит от конкретной схемы и реализована в производных классах

### 3.3.3.1 Явная схема

Её решатель реализован в классе `TestTransport1WorkerExplicit`. Рабочая функция по порядку:

- копирует текущий вектор значений во вспомогательный вектор `u_old`. Этот шаг добавлен сюда для ясности. Вообще говоря, его можно было избежать.
- устанавливает граничное условие в левой точке
- далее в цикле по точкам реализует расчётную схему (3.13).

```
86 void impl_step(double tau) override {  
87     std::vector<double> u_old(_u);  
88     _u[0] = exact_solution(_grid.point(0).x());  
89     for (size_t i=1; i<_grid.n_points(); ++i){  
90         _u[i] = u_old[i] - tau/_h*(u_old[i] - u_old[i-1]);  
91     }  
92 }
```

### 3.3.3.2 Неявная схема

Её решатель реализован в классе `TestTransport1WorkerImplicit`. Поскольку здесь для нахождения решения требуется решить СЛАУ, то порядок действий включает в себя:

- формирование класса-решателя.
- формирование столбца свободных членов
- вызова функции решения СЛАУ для найденного столбца правой части. Ответ записывается во внутреннее поле класса `_u`

```
135 void impl_step(double tau) override {  
136     AmgMatrixSolver& slv = build_solver(tau);  
137     std::vector<double> rhs = build_rhs(tau);  
138     slv.solve(rhs, _u);  
139 }
```

Для построения и инициализации решателя необходимо собрать матрицу правой части системы уравнений (3.16). Матрица зависит от шага по времени (через число Куранта), при этом шаг по времени является аргументом функции

`build_solver`, которая приходит от пользователя решателя через аргумент функции `step()`.

Таким образом, в логике работы приложения, нам придётся пересобирать матрицу на каждой временной итерации. При этом, почти всегда шаги по времени постоянны для временных слоёв. То

есть одну и ту же операцию (сборку матрицы) при одним и тех же аргументах (шаге по времени) придётся повторять.

Поскольку сборка матрицы – дорогая операция, то результат работы функции `build_solver` мы кэшируем (сохраняем во внутреннее поле класса `_solver`). С тем чтобы на следующем временном слое в случае, если шаг по времени не изменился (`_last_used_tau == tau`), просто вернуть ответ, посчитанный ранее.

```
144 AmgMatrixSolver& build_solver(double tau){  
145     if (_last_used_tau != tau){  
146         CsrMatrix mat = build_lhs(tau);  
147         _solver.set_matrix(mat);  
148         _last_used_tau = tau;  
149     }  
150     return _solver;  
151 }
```

Сама сборка двухдиагональной матрицы происходит в функции `build_lhs`. В первой и последней строке учитываются граничные условия, а строки, соответствующие внутренним узлам, заполняются согласно схеме (3.16)

```
153 virtual CsrMatrix build_lhs(double tau){  
154     LodMatrix mat(_u.size());  
155     mat.set_value(0, 0, 1.0);  
156     mat.set_value(_u.size()-1, _u.size()-1, 1.0);  
157     double diag = 1.0 + tau/_h;  
158     double nondiag = -tau/_h;  
159     for (size_t i=1; i<_u.size()-1; ++i){  
160         mat.set_value(i, i, diag);  
161         mat.set_value(i, i-1, nondiag);  
162     }  
163     return mat.to_csr();  
164 }
```

Сборка правой части СЛАУ происходит в функции `build_rhs`. Согласно схеме (3.16) правый столбец равен значению функции на предыдущем временном слое. В коде мы создаём столбец `rhs` как копию вектора `_u`. А далее переписываем первый и последний элемент с тем, чтобы учесть граничные условия.

```
166 virtual std::vector<double> build_rhs(double tau){  
167     std::vector<double> rhs(_u);  
168     rhs[0] = exact_solution(_grid.point(0).x());
```

```

169     rhs.back() = exact_solution(_grid.point(_grid.n_points()-1).x());
170
171     return rhs;
}

```

### 3.3.3.3 Схема Кранка-Николсон

Её решатель реализован в классе `TestTransport1WorkerCN`.

По аналогии с предыдущей программой, здесь требуется решить СЛАУ, возникающую из схемы (3.17). Таким образом, вся логика работы этого класса (включая кэширование решателя) повторяет логику работы рассмотренного ранее класса для чисто неявной схемы `TestTransport1WorkerImplicit`. Отличаются эти классы только реализацией функций построения матрицы и правой части. Поэтому настоящий класс наследуется от `TestTransport1WorkerImplicit`

```

200 class TestTransport1WorkerCN: public TestTransport1WorkerImplicit{

```

и переопределяет только функции сборки левой части (3.17) с учётом граничных условий

```

204     CsrMatrix build_lhs(double tau) override{
205
206     LodMatrix mat(_u.size());
207
208     mat.set_value(0, 0, 1.0);
209
210     mat.set_value(_u.size()-1, _u.size()-1, 1.0);
211
212     double diag = 1.0 + 0.5*tau/_h;
213
214     double nondiag = -0.5*tau/_h;
215
216     for (size_t i=1; i<_u.size()-1; ++i){
217
218         mat.set_value(i, i, diag);
219
220         mat.set_value(i, i-1, nondiag);
221
222     }
223
224     return mat.to_csr();
225 }

```

и правой части (3.17) с учётом граничных условий

```

217     std::vector<double> build_rhs(double tau) override{
218
219     std::vector<double> rhs(_u);
220
221     rhs[0] = exact_solution(_grid.point(0).x());
222
223     rhs.back() = exact_solution(_grid.point(_grid.n_points()-1).x());
224
225     for (size_t i=1; i<rhs.size()-1; ++i){
226
227         rhs[i] -= 0.5 * tau / _h * (_u[i] - _u[i-1]);
228
229     }
230
231     return rhs;
232 }

```

### 3.3.4 Анализ результатов работы

Сравнение полученных ответов (по явной и неявной схемам) с точным решением представлено на рис. 3.

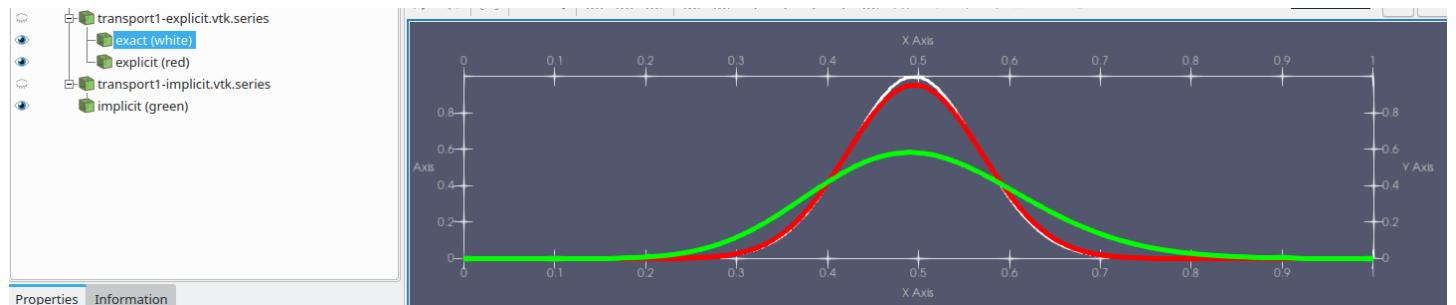


Рис. 3: Сравнение точного (белая линия) решения и численных решений по явной (красная) и неявной (зелёная) схемам

Чтобы получить такую картинку необходимо открыть в Paraview сгенерированные в результате работы программ выходные файлы `transport1_explicit.vtk.series` и `transport1_implicit.vtk.series`. И далее проделать преобразования, описанные в пункте B.3.1.

Для построения графиков сходимости, необходимо преобразовать написанные программы, запустив цикл по различным значениям числа Куранта

```
for (double Cu: { ... }{
    // solution
    ...

    std::cout << 1.0/tau << " " << norm << std::endl;
}
```

и построить график полученной таблицы в логарифмических осях. При задании диапазона изменений  $C$  следует учитывать, что явная схема устойчива только при  $C \leq 1$ , в то время как две другие схемы безусловно устойчивы.

Графики сходимости с уменьшением шага по времени представлены на рис. 4.

Видно, что для явной схемы с уменьшением шага по времени ответ отдаляется от точного, а для неявной – наоборот, приближается.

Это объясняется тем, что в случае явной схемы ошибки по времени и по пространству имеют разный знак и (в случае их равенства) компенсируют друг друга. А для неявной эти ошибки имеют одинаковый знак.

В пределе (с минимальным шагом по времени) все три схемы сходятся к одной и той же ошибке (ошибке схемы по пространству).

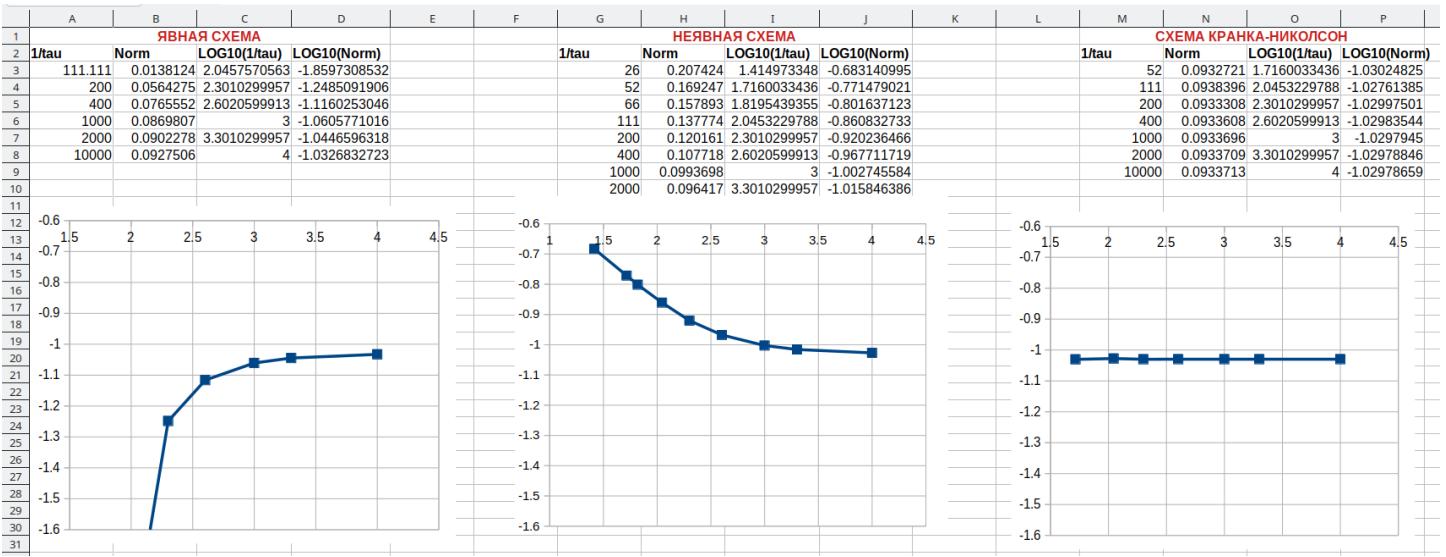


Рис. 4: Сходимость решения уравнения переноса с уменьшением  $\tau$

### 3.4 Задание для самостоятельной работы

#### 3.4.1 Постановка задачи

Написать двумерный решатель для уравнения переноса

$$\frac{\partial u}{\partial t} + U \frac{\partial u}{\partial x} + V \frac{\partial u}{\partial y} = 0.$$

Решение проводить в квадрате  $x, y \in [-1, 1]$ .

Требуется

- расчитать и нарисовать в Paraview нестационарное решение (см. [B.3.3](#));
- построить график, иллюстрирующий увеличение нормы ошибки с продвижением по времени;
- исследовать устойчивость схемы. Эмпирическим путём выяснить, какое максимально возможный шаг по времени можно брать при фиксированном разбиении по пространству;
- построить график, иллюстрирующий сходимость нормы ошибки при уменьшении шага по времени при фиксированном разбиении по пространству.

##### 3.4.1.1 Тестовый пример 1

На этапе первичного тестирования использовать значения скорости

$$U = 1, \quad V = 0.$$

А в качестве начального решения брать простой "столбик" ([рис. 5](#))

$$u(x, y, 0) = u_0(x, y) = \begin{cases} 1, & -1 \leq x \leq -0.8, -0.1 \leq y \leq 0.1, \\ 0, & \text{иначе.} \end{cases}$$

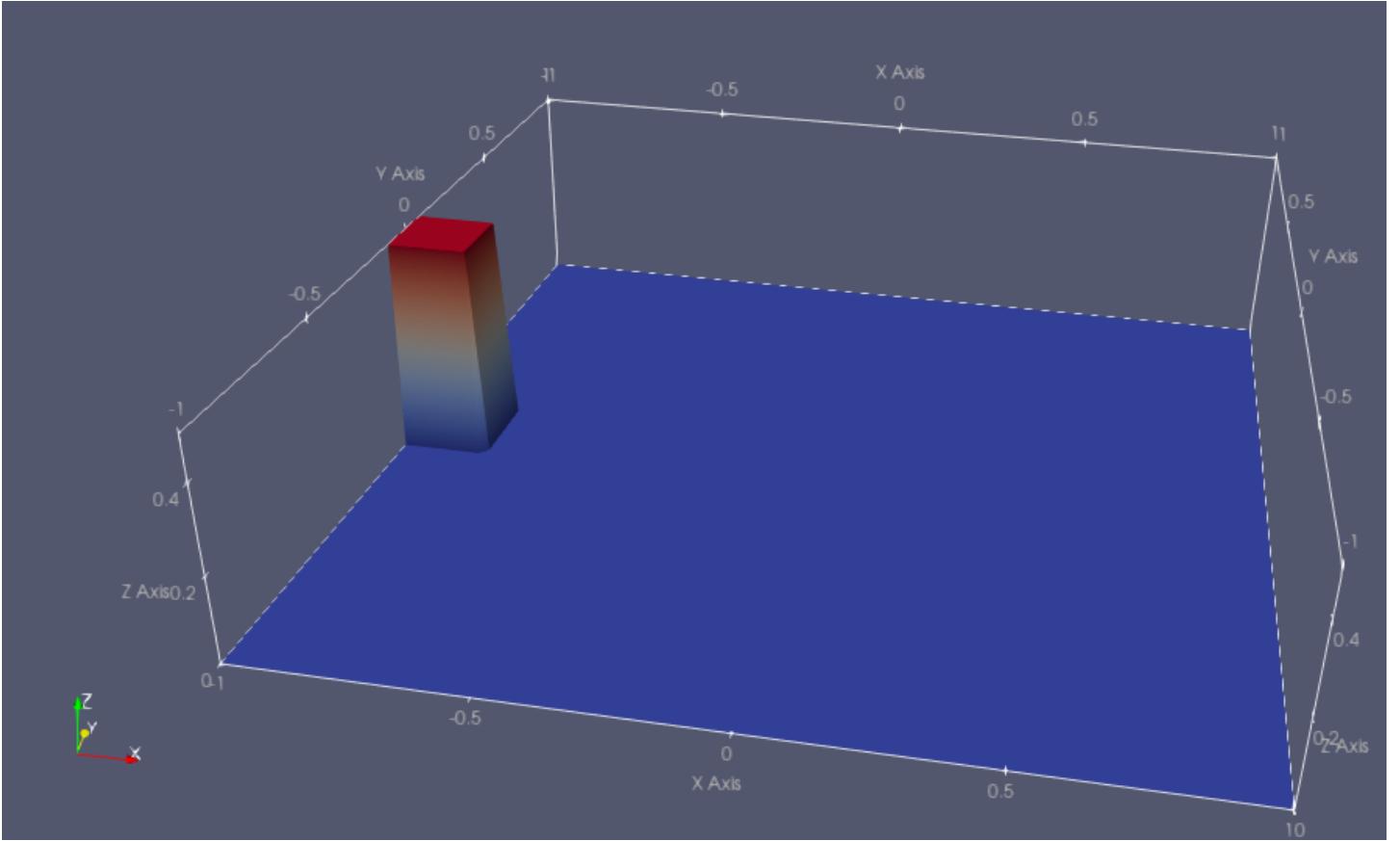


Рис. 5: Начальные условия для первого тестового примера

Точным решением будет функция

$$u^e(x, y, t) = u_0(x - t, y)$$

То есть этот столбик будет двигаться вправо с единичной скоростью и за время 2 полностью покинет расчётную область.

### 3.4.1.2 Тестовый пример 2

После того, как этот тест будет пройден, использовать постановку с непостоянной по пространству скоростью

$$U(x, y) = -y, \quad V(x, y) = x.$$

и начальным решением вида (рис. 6)

$$\begin{aligned} r_0(x, y) &= \sqrt{(x - 0.5)^2 + y^2}; \quad \sigma = 0.1; \\ u(x, y, 0) &= u_0(x, y) = e^{-r_0^2(x, y)/\sigma^2} \end{aligned}$$

В процессе решения этот "холмик" будет двигаться по окружности, описывая полный оборот за время  $t = 4\pi$ .

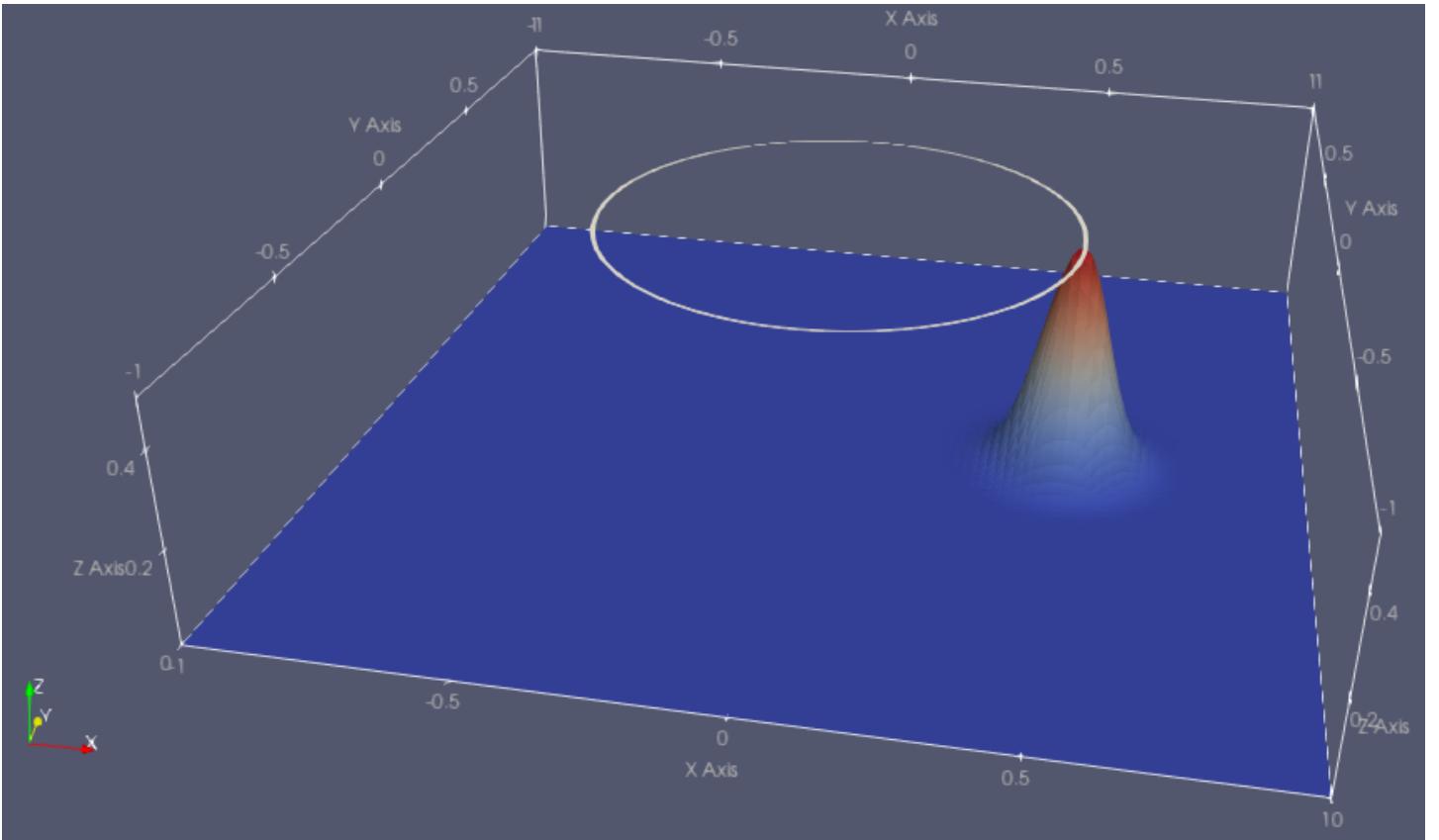


Рис. 6: Начальные условия для второго тестового примера

Точное решение на момент времени  $t$  будет иметь вид

$$\begin{aligned}x_c(t) &= 0.5 \cos(0.5t); \\y_c(t) &= 0.5 \sin(0.5t); \\r(x, y, t) &= \sqrt{(x - x_c(t))^2 + (y - y_c(t))^2}; \\u^e(x, y, t) &= e^{-r^2(x, y, t)/\sigma^2}\end{aligned}$$

### 3.4.2 Расчёчная схема

Использовать противопотоковую явную схему:

$$\frac{\hat{u}_k - u_k}{\tau} + |U_k| \frac{u_k - u_{\text{upx}[k]}}{h_x} + |V_k| \frac{u_k - u_{\text{upy}[k]}}{h_y} = 0$$

Здесь  $\text{upx}[k]$ ,  $\text{upy}[k]$  – значения индексов, расположенных против потока относительно узла  $k$  в направлениях  $x$  и  $y$  соответственно.

Поскольку скорость в настоящей постановке непостоянная и зависит от точки пространства, то вычислять индекс узла, расположенного против потока приходится в зависимости от значения скорости. С использованием ранее введённых алгоритмов перехода от парных  $(i, j)$  индексов к сквозному

индексу  $k$  (2.10) и обратно (2.11) запишем

$$\begin{aligned} i &= i[k]; \\ j &= j[k]; \\ \text{upx}[k] &= \begin{cases} k[i-1, j], & U_k \geq 0, \\ k[i+1, j], & U_k < 0, \end{cases} \\ \text{upy}[k] &= \begin{cases} k[i, j-1], & V_k \geq 0, \\ k[i, j+1], & V_k < 0. \end{cases} \end{aligned}$$

В схеме скорости переноса взяты по абсолютному значению. Это связано с зависимостью направления конечной разности от знака скорости. Так если  $U_k > 0$ , то для дискретизации производной по  $x$  используется разность назад:

$$U \frac{\partial u}{\partial x} \approx U_k \frac{u_{k[i,j]} - u_{k[i-1,j]}}{h_x} = |U_k| \frac{u_{k[i,j]} - u_{k[i-1,j]}}{h_x} = |U_k| \frac{u_{k[i,j]} - u_{\text{upx}[k]}}{h_x}$$

Если же  $U_k < 0$ , то используется разность вперёд

$$U \frac{\partial u}{\partial x} \approx U_k \frac{u_{k[i+1,j]} - u_{k[i,j]}}{h_x} = -U_k \frac{u_{k[i,j]} - u_{k[i+1,j]}}{h_x} = |U_k| \frac{u_{k[i,j]} - u_{k[i+1,j]}}{h_x} = |U_k| \frac{u_{k[i,j]} - u_{\text{upx}[k]}}{h_x}$$

На границах использовать условия первого рода. Можно просто нули, поскольку они соответствуют постановке.

## 4 Лекция 4 (30.09)

### 4.1 Моделирование течения вязкой несжимаемой жидкости методом конечных разностей

#### 4.1.1 Система уравнений Навье-Стокса

Будем рассматривать стационарную двумерную систему уравнений Навье-Стокса для вязкой несжимаемой жидкости. В безразмерном консервативном виде в декартовой системе координат она имеет вид

$$\frac{\partial u^2}{\partial x} + \frac{\partial uv}{\partial y} = -\frac{\partial p}{\partial x} + \frac{1}{Re} \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right), \quad (4.1)$$

$$\frac{\partial uv}{\partial x} + \frac{\partial v^2}{\partial y} = -\frac{\partial p}{\partial y} + \frac{1}{Re} \left( \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right), \quad (4.2)$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0. \quad (4.3)$$

Неизвестными являются поля скорости:  $u$  – в направлении оси  $x$ ,  $v$  – в направлении оси  $y$ , и давления  $p$ .

Число Рейнольдса определено через характерную скорость  $U$ , [м/с] и характерный линейный размер  $L$ , [м] как

$$Re = \frac{UL\rho}{\mu},$$

где  $\rho$ , [кг/м<sup>3</sup>] – постоянная (вследствии несжимаемости) плотность жидкости, а  $\mu$ , [Па·с] – динамическая вязкость жидкости.

Характерное значение для давление выписывается в виде:  $p^0 = \rho U^2$ , [Па].

Для решения этой системы будем использовать метод конечных разностей с аппроксимацией по пространству второго порядка и последовательное (раздельное) решение входящих в неё уравнений.

Глядя на вид уравнений (4.1) – (4.3) можно выделить несколько проблем, которые необходимо решить при построении расчётной схемы:

- нелинейность конвективного оператора в (4.1), (4.2),
- отсутствие явного уравнения для определения давления,
- аппроксимация первых производных для давления и скорости со вторым порядком точности.

Для решения первой проблемы будем использовать итерационный процесс с линеаризацией – то есть записывать уравнение на итерационном слое используя значения неизвестных полей с прошлого слоя. Вторую проблему будем решать с помощью алгоритма SIMPLE связывания давления и скорости (Pressure-Velocity Coupling). Решать третью проблему будем с помощью пространственной аппроксимации на разнесённой сетке (Staggered Grid).

## 4.1.2 Схема расчёта

Стационарную задачу (4.1)-(4.3) будем решать методом установления. Для этого в первые два уравнения введём фиктивную производную по времени, которую распишем по неявной двухслойной схеме с шагом  $\tau$ . Тогда задача на одном итерационном слое примет вид

$$\frac{\hat{u} - u}{\tau} + \frac{\partial u \hat{u}}{\partial x} + \frac{\partial v \hat{u}}{\partial y} = -\frac{\partial \hat{p}}{\partial x} + \frac{1}{\text{Re}} \left( \frac{\partial^2 \hat{u}}{\partial x^2} + \frac{\partial^2 \hat{u}}{\partial y^2} \right), \quad (4.4)$$

$$\frac{\hat{v} - v}{\tau} + \frac{\partial u \hat{v}}{\partial x} + \frac{\partial v \hat{v}}{\partial y} = -\frac{\partial \hat{p}}{\partial y} + \frac{1}{\text{Re}} \left( \frac{\partial^2 \hat{v}}{\partial x^2} + \frac{\partial^2 \hat{v}}{\partial y^2} \right), \quad (4.5)$$

$$\frac{\partial \hat{u}}{\partial x} + \frac{\partial \hat{v}}{\partial y} = 0. \quad (4.6)$$

При записи была произведена линеаризация конвективного слагаемого: один из множителей в производной был отнесён на предыдущий временной слой. В остальном схема неявная.

На временном слое значения  $u, v, p$  известны, а  $\hat{u}, \hat{v}, \hat{p}$  подлежат определению.

Критерием выхода из итерационного процесса является пороговое условие на невязку, вычисленную с использованием найденных на слое значений неизвестных:

$$\begin{aligned} r_u &= \frac{\partial \hat{u} \hat{u}}{\partial x} + \frac{\partial \hat{u} \hat{v}}{\partial y} + \frac{\partial \hat{p}}{\partial x} - \frac{1}{\text{Re}} \left( \frac{\partial^2 \hat{u}}{\partial x^2} + \frac{\partial^2 \hat{u}}{\partial y^2} \right), \\ r_v &= \frac{\partial \hat{u} \hat{v}}{\partial x} + \frac{\partial \hat{v} \hat{v}}{\partial y} + \frac{\partial \hat{p}}{\partial y} - \frac{1}{\text{Re}} \left( \frac{\partial^2 \hat{v}}{\partial x^2} + \frac{\partial^2 \hat{v}}{\partial y^2} \right), \\ \max(\|r_u\|, \|r_v\|) &< \varepsilon. \end{aligned} \quad (4.7)$$

### 4.1.2.1 Метод SIMPLE

Приведём алгоритм для явного выражения уравнения для давления из уравнения неразрывности (4.6).

Распишем искомые переносные в виде суммы

$$\begin{aligned} \hat{u} &= u^* + u', \\ \hat{v} &= v^* + v', \\ \hat{p} &= p + p'. \end{aligned} \quad (4.8)$$

Пусть введённые выше поля  $u^*, v^*$  удовлетворяют уравнениям

$$u^* + \tau \frac{\partial u u^*}{\partial x} + \tau \frac{\partial v u^*}{\partial y} - \frac{\tau}{\text{Re}} \left( \frac{\partial^2 u^*}{\partial x^2} + \frac{\partial^2 u^*}{\partial y^2} \right) = -\tau \frac{\partial p}{\partial x} + u, \quad (4.9)$$

$$v^* + \tau \frac{\partial u v^*}{\partial x} + \tau \frac{\partial v v^*}{\partial y} - \frac{\tau}{\text{Re}} \left( \frac{\partial^2 v^*}{\partial x^2} + \frac{\partial^2 v^*}{\partial y^2} \right) = -\tau \frac{\partial p}{\partial y} + v. \quad (4.10)$$

Тогда уравнение для поправки  $u'$  запишем вычтя последнее выражение из уравнения (4.4), умноженного на  $\tau$ :

$$u' + \tau \frac{\partial uu'}{\partial x} + \tau \frac{\partial vu'}{\partial y} - \frac{\tau}{\text{Re}} \left( \frac{\partial^2 u'}{\partial x^2} + \frac{\partial^2 \hat{u}'}{\partial y^2} \right) = -\tau \frac{\partial p'}{\partial x}. \quad (4.11)$$

Основная идея алгоритма SIMPLE заключается в приближённом представлении выражения (4.11) в явном виде относительно поправки. Для этого все дифференциальные операторы, включающие в себя поправку скорости, из выражения убираются, а для компенсации в правую часть добавляется множитель  $d^u$ :

$$u' \approx -\tau d^u(x, y) \frac{\partial p'}{\partial x}. \quad (4.12)$$

Аналогичные рассуждения в отношении поправки поперечной скорости  $v'$  приводят к выражению

$$v' \approx -\tau d^v(x, y) \frac{\partial p'}{\partial y}. \quad (4.13)$$

К точному определению значения полей  $d^u, d^v$  вернёмся позднее, когда будем расписывать эти выражения на матричном уровне.

Далее используем уравнение неразрывности (4.6). Подставим в него разложения (4.8) и используем (4.12)-(4.13). Тогда получим уравнение Пуассона с непостоянным по пространству векторным коэффициентом диффузии  $(d^u, d^v)$  относительно поправки давления  $p'$ :

$$-\left[ \frac{\partial}{\partial x} \left( d^u \frac{\partial p'}{\partial x} \right) + \frac{\partial}{\partial y} \left( d^v \frac{\partial p'}{\partial y} \right) \right] = -\frac{1}{\tau} \left( \frac{\partial u^*}{\partial x} + \frac{\partial v^*}{\partial y} \right). \quad (4.14)$$

Определим порядок вычислений на итерационном слое. Напомним, что значения  $u, v, p$  с предыдущего слоя нам известно и задача состоит в нахождении значений  $\hat{u}, \hat{v}, \hat{p}$  на текущем слое.

1. Из уравнений (4.9), (4.10) вычисляются значения  $u^*, v^*$ ;
2. Они используются для вычисления правой части уравнения (4.14), в результате решения которого находится поправка давления  $p'$ ;
3. Дифференцируя найденную поправку давления найдём поправки скорости  $u', v'$  из выражений (4.12), (4.13);
4. Окончательно выразим значения переменных для текущего слоя из (4.8). Для улучшения стабильности алгоритма значение давления вычисляют с некоторым коэффициентом релаксации  $\alpha_p$ :

$$\hat{p} = p + \alpha_p p';$$

5. Далее проводится вычисление невязки с использованием найденных значений  $\hat{u}, \hat{v}, \hat{p}$  из выражения (4.7). Если она недостаточно мала, то выполняется присваивание  $u = \hat{u}, v = \hat{v}, p = \hat{p}$  и возвращение на шаг 1.

Полученные на каждом шаге итерационного процесса компоненты скорости  $\hat{u}, \hat{v}$  точно удовлетворяют уравнению неразрывности (4.6) в “чёрных” узлах сетки, но уравнения движения (4.4), (4.5) выполняются лишь приближённо.

Всего в алгоритме SIMPLE есть два параметра: коэффициент релаксации давления  $\alpha_p$  и фиктивный шаг по времени  $\tau$  (который можно трактовать как коэффициент релаксации скорости).

### 4.1.3 Пространственная аппроксимация

Для численной реализации алгоритма решения необходимо провести пространственную аппроксимацию полудискретизованных выражений (4.9), (4.10), (4.12), (4.13), (4.14).

#### 4.1.3.1 Разнесённая сетка

Будем использовать структурированную четырёхугольную сетку с постоянным шагом по пространству. При этом неизвестные параметры будем задавать по схеме, представленной на рис. 7.

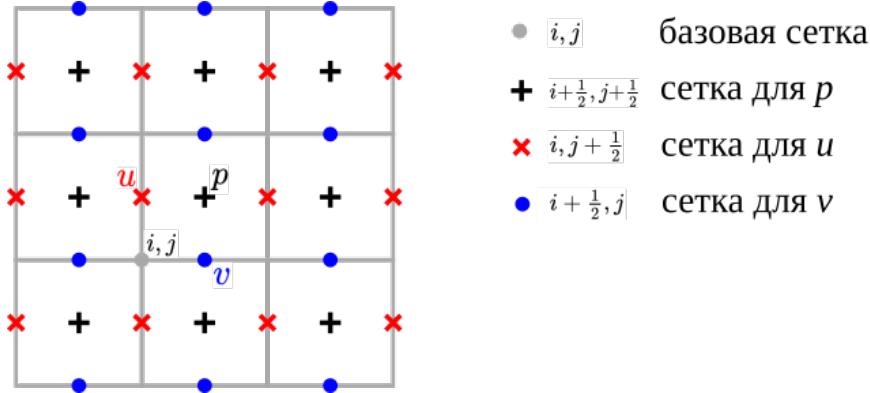


Рис. 7: Разнесённая сетка

Введём разбиение сетки:  $n_x$  — количество ячеек в направлении  $x$ ,  $n_y$  — количество ячеек в направлении  $y$ .

Очевидно, что при использовании такого разнесённого шаблона, количество точек, в которых заданы значения, будет различным для разных параметров. Так количество узловых значений давления будет равно  $n_x \times n_y$ , продольной скорости  $u - (n_x + 1) \times n_y$ , а поперечной  $v - n_x \times (n_y + 1)$ .

Использование такого расположения узловых точек даёт преимущество при аппроксимации первых производных. Так, конечная разность

$$\frac{\partial p}{\partial x} \Big|_{i,j+\frac{1}{2}} = \frac{p_{i+\frac{1}{2},j+\frac{1}{2}} - p_{i-\frac{1}{2},j+\frac{1}{2}}}{h_x} + o(h_x^2)$$

будем симметричной в узле  $i, j + \frac{1}{2}$ , где задана компонента скорости  $u$ , и поэтому будет иметь там второй порядок точности.

Выражения (4.9), (4.12) аппроксимируются на сетке для  $u$ , выражения (4.10), (4.13) — на сетке для  $v$ , а (4.14) — на сетке для  $p$ .

Введём сквозную линейную нумерацию узлов сетки: нулевой узел разместим в левом нижнем углу, далее будем индексировать слева направо и потом снизу вверх. Для основной сетки перевод двумерного индекса  $i, j$  в сквозной индекс будет проводится по формуле

$$k(i, j) = j(n_x + 1) + i. \quad (4.15)$$

Для сеток, на которых заданы сеточные параметры, такой перевод примет вид

$$k(i + \frac{1}{2}, j + \frac{1}{2}) = jn_x + i, \quad - \text{сетка для давления } p \quad (4.16)$$

$$k(i, j + \frac{1}{2}) = j(n_x + 1) + i, \quad - \text{сетка для продольной скорости } u \quad (4.17)$$

$$k(i + \frac{1}{2}, j) = jn_x + i, \quad - \text{сетка для поперечной скорости } v \quad (4.18)$$

#### 4.1.3.2 Уравнения движения

Запишем конечноразностную аппроксимацию уравнения (4.9) для пробной скорости  $u^*$  в “красных” узлах сетки  $(i, j + \frac{1}{2})$ :

$$\begin{aligned} & u_{i,j+\frac{1}{2}}^* + \frac{\tau}{h_x} \left( (uu^*)_{i+\frac{1}{2},j+\frac{1}{2}} - (uu^*)_{i-\frac{1}{2},j+\frac{1}{2}} \right) \\ & + \frac{\tau}{h_y} \left( (vu^*)_{i,j+1} - (vu^*)_{i,j} \right) \\ & - \frac{1}{\text{Re}} \frac{\tau}{h_x^2} \left( u_{i-1,j+\frac{1}{2}}^* - 2u_{i,j+\frac{1}{2}}^* + u_{i+1,j+\frac{1}{2}}^* \right) \\ & - \frac{1}{\text{Re}} \frac{\tau}{h_y^2} \left( u_{i,j-\frac{1}{2}}^* - 2u_{i,j+\frac{1}{2}}^* + u_{i,j+\frac{3}{2}}^* \right) \\ & = u_{i,j+\frac{1}{2}} - \frac{\tau}{h_x} \left( p_{i+\frac{1}{2},j+\frac{1}{2}} - p_{i-\frac{1}{2},j+\frac{1}{2}} \right). \end{aligned} \quad (4.19)$$

В приведённом выражении за исключением конвективных слагаемых вида  $uu$  все остальные сеточные вектора используются на своих сетках. Конвективные слагаемые распишем через полусуммы вида:

$$u_{i+\frac{1}{2}} = \frac{u_i + u_{i+1}}{2} + o(h^2)$$

Тогда

$$\begin{aligned} (uu^*)_{i+\frac{1}{2},j+\frac{1}{2}} &= \left( u_{i+\frac{1}{2},j+\frac{1}{2}} \right) \left( u_{i+\frac{1}{2},j+\frac{1}{2}}^* \right) = \frac{1}{4} \left( u_{i,j+\frac{1}{2}} + u_{i+1,j+\frac{1}{2}} \right) \left( u_{i,j+\frac{1}{2}}^* + u_{i+1,j+\frac{1}{2}}^* \right), \\ (uu^*)_{i-\frac{1}{2},j+\frac{1}{2}} &= \frac{1}{4} \left( u_{i,j+\frac{1}{2}} + u_{i-1,j+\frac{1}{2}} \right) \left( u_{i,j+\frac{1}{2}}^* + u_{i-1,j+\frac{1}{2}}^* \right), \\ (vu^*)_{i,j+1} &= \frac{1}{4} \left( v_{i+\frac{1}{2},j+1} + v_{i-\frac{1}{2},j+1} \right) \left( u_{i,j+\frac{3}{2}}^* + u_{i,j+\frac{1}{2}}^* \right), \\ (vu^*)_{i,j} &= \frac{1}{4} \left( v_{i+\frac{1}{2},j} + v_{i-\frac{1}{2},j} \right) \left( u_{i,j+\frac{1}{2}}^* + u_{i,j-\frac{1}{2}}^* \right). \end{aligned}$$

Схему (4.19) можно записать в виде системы линейных уравнений вида

$$A^u u^* = b^u. \quad (4.20)$$

Сеточная матрица  $A^u$  будет иметь  $(n_x + 1)n_y$  строк. Для строки, соответствующей  $(i, j + \frac{1}{2})$  узлу ненулевыми будут столбцы, соответствующие узлам:

- $(i, j + \frac{1}{2})$ ,
- $(i + 1, j + \frac{1}{2})$ ,
- $(i - 1, j + \frac{1}{2})$ ,
- $(i, j + \frac{3}{2})$ ,
- $(i, j - \frac{1}{2})$ .

В случае использования стандартной нумерации узлов структурированной сетки, когда нулевой индекс соответствуют левому нижнему узлу и далее нумерация идёт с быстрым индексом  $i$ , то матрица будет пятидиагональной.

Подставим полученные выражения в конвективную часть выражения (4.19). Множитель при диагональном элементе  $u_{i,j+\frac{1}{2}}^*$  будет равен:

$$\frac{\tau}{4} \left( \underbrace{\frac{u_{i,j+\frac{1}{2}} - u_{i-1,j+\frac{1}{2}}}{h_x}}_{\frac{\partial u}{\partial x}\Big|_{i-\frac{1}{2},j+\frac{1}{2}}} + \underbrace{\frac{u_{i+1,j+\frac{1}{2}} - u_{i,j+\frac{1}{2}}}{h_x}}_{\frac{\partial u}{\partial x}\Big|_{i+\frac{1}{2},j+\frac{1}{2}}} + \underbrace{\frac{v_{i+\frac{1}{2},j+1} - v_{i+\frac{1}{2},j}}{h_y}}_{\frac{\partial v}{\partial y}\Big|_{i+\frac{1}{2},j+\frac{1}{2}}} + \underbrace{\frac{v_{i-\frac{1}{2},j+1} - v_{i-\frac{1}{2},j}}{h_y}}_{\frac{\partial v}{\partial y}\Big|_{i-\frac{1}{2},j+\frac{1}{2}}} \right)$$

Сумма первого и четвёртого слагаемых представляет собой разностный аналог уравнения неразрывности (4.6), записанной для “чёрного” узла сетки  $i - \frac{1}{2}, j + \frac{1}{2}$  относительно компонент скорости с предыдущей итерации. Как было сказано ранее, в настоящем алгоритме уравнение неразрывности для итоговых по результатам итерации скорости в этих узлах выполняется точно. Поэтому эта сумма в точности будет равна нулю. Аналогичный результат получится и для суммы второго и третьего слагаемых. Отсюда следует вывод, что конвективное слагаемое не даёт вклад в диагональ итоговой матрицы (как и следовало ожидать от симметричной аппроксимации).

Окончательно запишем все пять ненулевых вхождений в строку матрицы:

$$A^u [k(i, j + \frac{1}{2}), k(i, j + \frac{1}{2})] = 1 + \frac{2\tau}{\text{Re}} \left( \frac{1}{h_x^2} + \frac{1}{h_y^2} \right) \quad - \text{ основная диагональ,} \quad (4.21)$$

$$A^u [k(i, j + \frac{1}{2}), k(i + 1, j + \frac{1}{2})] = -\frac{\tau}{\text{Re}} \frac{1}{h_x^2} + \frac{\tau}{4h_x} \left( u_{i,j+\frac{1}{2}} + u_{i+1,j+\frac{1}{2}} \right) \quad - \text{ первая верхняя диагональ,}$$

$$A^u [k(i, j + \frac{1}{2}), k(i - 1, j + \frac{1}{2})] = -\frac{\tau}{\text{Re}} \frac{1}{h_x^2} - \frac{\tau}{4h_x} \left( u_{i,j+\frac{1}{2}} + u_{i-1,j+\frac{1}{2}} \right) \quad - \text{ первая нижняя диагональ,}$$

$$A^u [k(i, j + \frac{1}{2}), k(i, j + \frac{3}{2})] = -\frac{\tau}{\text{Re}} \frac{1}{h_y^2} + \frac{\tau}{4h_y} \left( v_{i+\frac{1}{2},j+1} + v_{i-\frac{1}{2},j+1} \right) \quad - \text{ вторая верхняя диагональ,}$$

$$A^u [k(i, j + \frac{1}{2}), k(i, j - \frac{1}{2})] = -\frac{\tau}{\text{Re}} \frac{1}{h_y^2} - \frac{\tau}{4h_y} \left( v_{i+\frac{1}{2},j} + v_{i-\frac{1}{2},j} \right) \quad - \text{ вторая нижняя диагональ.}$$

Здесь  $k(i, j)$  – функция перевода двумерного индекса в сквозной (4.17).

Аналогичные выкладки для второго из уравнений движения (4.10) дают систему уравнений

$$A^v v^* = b^v, \quad (4.22)$$

элементы пятидиагональной матрицы которой имеют вид

$$\begin{aligned} A^v [k(i + \frac{1}{2}, j), k(i + \frac{1}{2}, j)] &= 1 + \frac{2\tau}{\operatorname{Re} h_x^2} \left( \frac{1}{h_x^2} + \frac{1}{h_y^2} \right) \quad - \text{ основная диагональ,} \\ A^v [k(i + \frac{1}{2}, j), k(i + \frac{3}{2}, j)] &= -\frac{\tau}{\operatorname{Re} h_x^2} \frac{1}{h_x^2} + \frac{\tau}{4h_x} \left( u_{i+1,j+\frac{1}{2}} + u_{i+1,j-\frac{1}{2}} \right) \quad - \text{ первая верхняя диагональ,} \\ A^v [k(i + \frac{1}{2}, j), k(i - \frac{1}{2}, j)] &= -\frac{\tau}{\operatorname{Re} h_x^2} \frac{1}{h_x^2} - \frac{\tau}{4h_x} \left( u_{i,j+\frac{1}{2}} + u_{i,j-\frac{1}{2}} \right) \quad - \text{ первая нижняя диагональ,} \\ A^v [k(i + \frac{1}{2}, j), k(i + \frac{1}{2}, j + 1)] &= -\frac{\tau}{\operatorname{Re} h_y^2} \frac{1}{h_y^2} + \frac{\tau}{4h_y} \left( v_{i+\frac{1}{2},j} + v_{i+\frac{1}{2},j+1} \right) \quad - \text{ вторая верхняя диагональ,} \\ A^v [k(i + \frac{1}{2}, j), k(i + \frac{1}{2}, j - 1)] &= -\frac{\tau}{\operatorname{Re} h_y^2} \frac{1}{h_y^2} - \frac{\tau}{4h_y} \left( v_{i+\frac{1}{2},j} + v_{i+\frac{1}{2},j-1} \right) \quad - \text{ вторая нижняя диагональ.} \end{aligned} \quad (4.23)$$

Правая часть аппроксимируется в виде

$$b^{v*}[k(i + \frac{1}{2}, j)] = 1 - \frac{\tau}{h_y} \left( p_{i+\frac{1}{2},j+1} - p_{i+\frac{1}{2},j} \right).$$

Используется функция перевода двумерного индекса в сквозной из (4.18).

#### 4.1.3.3 Уравнение для поправки давления

Распишем уравнение (4.14) на “чёрной” сетке методом конечных разностей. Для первого слагаемого получим

$$\begin{aligned} \frac{\partial}{\partial x} \left( d^u \frac{\partial p'}{\partial x} \right) \Big|_{i+\frac{1}{2},j+\frac{1}{2}} &\approx \frac{1}{h_x} \left( d^u_{i+1,j+\frac{1}{2}} \frac{\partial p'}{\partial x} \Big|_{i+1,j+\frac{1}{2}} - d^u_{i,j+\frac{1}{2}} \frac{\partial p'}{\partial x} \Big|_{i,j+\frac{1}{2}} \right) \\ &= \frac{1}{h_x} \left( d^u_{i+1,j+\frac{1}{2}} \frac{p'_{i+\frac{3}{2},j+\frac{1}{2}} - p'_{i+\frac{1}{2},j+\frac{1}{2}}}{h_x} - d^u_{i,j+\frac{1}{2}} \frac{p'_{i+\frac{1}{2},j+\frac{1}{2}} - p'_{i-\frac{1}{2},j+\frac{1}{2}}}{h_x} \right). \end{aligned} \quad (4.24)$$

Аналогично расписываются остальные слагаемые. В результате получим систему линейных уравнений вида

$$A^p p' = b^p, \quad (4.25)$$

где ненулевые коэффициенты пятидиагональной матрицы примут вид

$$A^p[k(i + \frac{1}{2}, j + \frac{1}{2}), k(i + \frac{1}{2}, j + \frac{1}{2})] = \frac{1}{h_x^2} \left( d_{i+1,j+\frac{1}{2}}^u + d_{i,j+\frac{1}{2}}^u \right) + \frac{1}{h_y^2} \left( d_{i+\frac{1}{2},j}^v + d_{i+\frac{1}{2},j+1}^v \right), \quad (4.26)$$

$$A^p[k(i + \frac{1}{2}, j + \frac{1}{2}), k(i + \frac{3}{2}, j + \frac{1}{2})] = -\frac{1}{h_x^2} d_{i+1,j+\frac{1}{2}}^u,$$

$$A^p[k(i + \frac{1}{2}, j + \frac{1}{2}), k(i - \frac{1}{2}, j + \frac{1}{2})] = -\frac{1}{h_x^2} d_{i,j+\frac{1}{2}}^u,$$

$$A^p[k(i + \frac{1}{2}, j + \frac{1}{2}), k(i + \frac{1}{2}, j + \frac{3}{2})] = -\frac{1}{h_y^2} d_{i+\frac{1}{2},j+1}^v,$$

$$A^p[k(i + \frac{1}{2}, j + \frac{1}{2}), k(i + \frac{1}{2}, j - \frac{1}{2})] = -\frac{1}{h_y^2} d_{i+\frac{1}{2},j}^v.$$

(4.27)

Столбец свободных членов аппроксимируется в виде

$$b^p[k(i + \frac{1}{2}, j + \frac{1}{2})] = -\frac{1}{\tau} \left( \frac{u_{i+1,j+\frac{1}{2}}^* - u_{i,j+\frac{1}{2}}^*}{h_x} + \frac{v_{i+\frac{1}{2},j+1}^* - v_{i+\frac{1}{2},j}^*}{h_y} \right). \quad (4.28)$$

Здесь используется функция перевода двумерного индекса в сквозной из (4.16).

Далее определим значения  $d^u, d^v$ . Согласно идее алгоритма SIMPLE  $d^u$  должна быть такой функцией, которая максимально приближает выражение (4.11) к (4.12).

Пространственная аппроксимация выражения (4.11) приводит к системе уравнений

$$A^u u' = -\tau \frac{\partial p'}{\partial x}$$

где матрица  $A^u$  – та же самая матрица, которая использовалась при аппроксимации уравнения движения (4.19).

Сравнивая предыдущее выражение с (4.12) сделаем вывод, что  $d^u$  должна быть такой, чтобы

$$d^u \frac{\partial p'}{\partial x} \approx (A^u)^{-1} \frac{\partial p'}{\partial x}.$$

То есть поэлементное умножение сеточного вектора  $d^u$  на другой вектор должно действовать похоже на умножение обратной к  $A^u$  матрицы на этот же самый вектор.

Исходя из свойств матрицы  $A^u$  (4.21) можно положить

$$d^u = (\text{diag}(A^u))^{-1} = \left( 1 + \frac{2\tau}{\text{Re}} \left( \frac{1}{h_x^2} + \frac{1}{h_y^2} \right) \right)^{-1} \quad (4.29)$$

и аналогично из (4.23)

$$d^v = (\text{diag}(A^v))^{-1} = \left( 1 + \frac{2\tau}{\text{Re}} \left( \frac{1}{h_x^2} + \frac{1}{h_y^2} \right) \right)^{-1}. \quad (4.30)$$

Равенство коэффициентов  $d^u = d^v$  – следствие использования симметричной аппроксимации конвективного слагаемого в уравнениях движения.

Таким образом мы получили выражения для коэффициентов уравнения для поправки давления, которые зависят только от разбиения сетки. В случае, если разбиение равномерное ( $h_x = \text{const}$ ,  $h_y = \text{const}$ ), то все значения коэффициентов одинаковы. Однако, для неравномерных разбиений, они будут зависеть от пространства и задаваться на “красной” (для  $d^u$ ) и “синей” (для  $d^v$ ) сетках.

В результате использования (4.29), (4.30) левая часть системы уравнений (4.25) будет постоянна на всех итерациях, что удобно для инициализации алгебраических решателей этой системы (можно провести инициализацию один раз до начала счёта).

Это отличает эту систему от двух других систем, возникающих из аппроксимации уравнений движения (4.20), (4.22), левые части которых зависят от значений с предыдущих итерационных слоёв. Этот момент обуславливает выбор решателей для этих систем, которые в эффективных гидродинамических кодах обычно отличаются, от решателя для системы (4.25).

#### 4.1.3.4 Уравнение для поправки скорости

И наконец рассмотрим аппроксимацию выражений (4.12), (4.13), которые примут явный вид

$$u'_{i,j+\frac{1}{2}} = -\tau d^u_{i,j+\frac{1}{2}} \frac{p'_{i+\frac{1}{2},j+\frac{1}{2}} - p'_{i-\frac{1}{2},j+\frac{1}{2}}}{h_x}, \quad (4.31)$$

$$v'_{i+\frac{1}{2},j} = -\tau d^v_{i+\frac{1}{2},j} \frac{p'_{i+\frac{1}{2},j+\frac{1}{2}} - p'_{i+\frac{1}{2},j-\frac{1}{2}}}{h_x}. \quad (4.32)$$

$$(4.33)$$

#### 4.1.3.5 Учёт граничных условий

Для уравнений Навье-Стокса на каждой границе расчётной области требуется столько условий, сколько есть уравнений движения. Для двумерной задачи (4.1)-(4.3) нужно задать два граничных условия.

При использовании разнесённой сетки граница области проходит по граням основной сетки. На нижней и верхней границах расчётной области присутствуют узлы для  $v$ , но отсутствуют узлы для  $u$ . На правой и левой границах, наоборот, есть узлы с заданными компонентами  $u$ , но нет узлов с компонентами  $v$ . Узловые значения для давления  $p$  никогда не бывают граничными.

Для простоты пока будем рассматривать только случай с заданными значениями двух компонент скорости на каждой из границ задачи:

$$\begin{aligned} u(x, y)|_{x,y \in \Gamma} &= u^\Gamma(x, y), \\ v(x, y)|_{x,y \in \Gamma} &= v^\Gamma(x, y). \end{aligned}$$

В схеме SIMPLE частные граничные условия для скорости учитываются при решении задачи для пробных скоростей  $u^*, v^*$ . Тогда для поправки скорости  $u', v'$  на границах будут справедливы соответствующие однородные граничные условия (нулевые значения в нашем случае):

$$\begin{aligned} u^*(x, y)|_{x,y \in \Gamma} &= u^\Gamma(x, y), \\ v^*(x, y)|_{x,y \in \Gamma} &= v^\Gamma(x, y), \\ u'(x, y)|_{x,y \in \Gamma} &= 0, \\ v'(x, y)|_{x,y \in \Gamma} &= 0. \end{aligned} \tag{4.34}$$

Для учёта граничных условий по скорости требуется модифицировать системы линейных уравнений (4.20), (4.22).

Рассмотрим нижнюю границу  $j = 0$ .

На нижней границе явно присутствуют узлы “синей” сетки. Значит можно явно установить значения для скорости  $v$  путём постановки нулей с единицой на диагонали в строке матрицы и отнесением необходимого граничного значение в правый вектор столбец системы (4.22):

$$A^v[k(i + \frac{1}{2}, 0), s] = \delta_{ks}, \quad \forall i, \forall s \tag{4.35}$$

$$b^v[k(i + \frac{1}{2}, 0)] = v^\Gamma.$$

Такая модификация просто заменяет  $k(i + \frac{1}{2}, 0)$ -ое уравнение системы (4.22) на выражение

$$v^*_{i+\frac{1}{2}, 0} = v^\Gamma.$$

Узлов для компонент  $u$  на нижней границе нет. Рассмотрим первый ряд точек “красной” сетки:  $(i, \frac{1}{2})$ . Если бы мы захотели заполнить коэффициенты системы линейных уравнений (4.20) по выведенным выше формулам (4.21) для узла, расположенного в этом ряду, мы бы столкнулись с необходимостью установки значения в фиктивную колонку: последнее из уравнений (4.21) предписывает нам установить значение по адресу  $[k(i, \frac{1}{2}), k(i, -\frac{1}{2})]$ , который, очевидно, не присутствует в матрице.

Действительно,  $k(i, \frac{1}{2})$ -ая строка системы уравнений (4.21) имеет вид

$$Du^*_{i, \frac{1}{2}} + U^1 u^*_{i+1, \frac{1}{2}} + L^1 u^*_{i-1, \frac{1}{2}} + U^2 u^*_{i, \frac{3}{2}} + L^2 u^*_{i, -\frac{1}{2}} = b^u_{i, \frac{1}{2}}, \tag{4.36}$$

где  $D$  – коэффициент с основной диагональю,  $U^{1,2}, L^{1,2}$  – коэффициенты с двух верхних и двух нижних диагоналях, вычисляемые по формулам (4.21). Вторая нижняя диагональ у этой строки матрицы отсутствует. Она соответствует вкладу от узла  $(i, -\frac{1}{2})$ , который лежит вне области расчёта, на полшага ниже нижней границе.

Тем не менее, такой фиктивный узел мы можем использовать для записи аппроксимации

$$u^*_{i, 0} = u^\Gamma = \frac{u^*_{i, \frac{1}{2}} + u^*_{i, -\frac{1}{2}}}{2} + o(h_x^2).$$

или

$$u^*_{i, -\frac{1}{2}} \approx 2u^\Gamma - u^*_{i, \frac{1}{2}}.$$

Подставляя это выражение в строку (4.36) получим

$$(D - L^2)u_{i,\frac{1}{2}}^* + U^1 u_{i+1,\frac{1}{2}}^* + L^1 u_{i-1,\frac{1}{2}}^* + U^2 u_{i,\frac{3}{2}}^* = b_{i,\frac{1}{2}}^u + 2u^\Gamma.$$

Таким образом, добавление коэффициента в фиктивную колонку строки матрицы при наличие условия первого рода на границе равносильно вычитанию этого коэффициента из диагонального элемента этой строки и вычитанием удвоенного граничного значения из правой части. В случае нижней границы получим

$$A^u[k(i, \frac{1}{2}), k(i, \frac{1}{2})] = A^u[k(i, -\frac{1}{2})], \quad (4.37)$$

$$b^u[k(i, \frac{1}{2})] = 2u^\Gamma.$$

Приёмы (4.35), (4.37) используются и на остальных границах для постановки граничных условий для скорости.

При сборке системы линейных уравнений для поправки давления (4.25) так же возникает проблема с обращением к фиктивным узлам. Например, при рассмотрении левой стенки ( $i = 0$  третье из уравнений (4.26) описывает несуществующий столбец  $k(-\frac{1}{2}, j + \frac{1}{2})$ . Если обратиться к выражению (4.24), то будет видно, что это слагаемое пришло в результате расписывания граничной производной  $p'$ , которая, исходя из выражения (4.12) пропорциональна граничному значению  $u'$ , то есть, вспоминая (4.34), равна нулю:

$$\left. \frac{\partial p'}{\partial x} \right|_{0,j+\frac{1}{2}} = -\frac{1}{\tau d^u} u'_{0,j+\frac{1}{2}} = 0.$$

То есть добавлять слагаемые, соответствующие фиктивным узлам, в матрицу  $A^p$  не нужно. Не нарушая общности выведённых ранее выражений (4.26), просто модифицируем значения коэффициентов  $d^u, d^v$ :

$$d^u_{0,j+\frac{1}{2}} = d^u_{n_x+1,j+\frac{1}{2}} = 0, \quad (4.38)$$

$$d^v_{i+\frac{1}{2},0} = d^u_{i+\frac{1}{2},n_y+1} = 0.$$

В исходных уравнениях (4.1)-(4.3) давление присутствует только в виде своих производных. Если в задаче нигде не задано явное граничное условие для давления, то решение для давления будет определено только с точностью до константы. Чтобы убрать эту неопределённость рекомендуется явно положить давление нулю в любом узле. Например, в случае нулевого узла, по аналогии с (4.35) запишем:

$$A^p[k(\frac{1}{2}, \frac{1}{2}), s] = \delta_{ks}, \quad (4.39)$$

$$b^p[k(\frac{1}{2}, \frac{1}{2})] = 0.$$

## 4.2 Программа для расчёта течения в каверне по схеме SIMPLE

#### 4.2.1 Постановка задачи

Для иллюстрации работы алгоритма рассмотрим задачу о течении в каверне. Постановку задачи представлена на рис. 8.

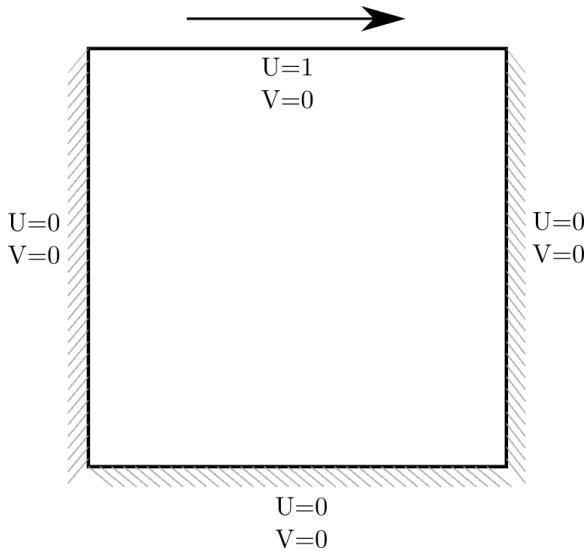


Рис. 8: Область расчёта задачи о каверне

Задача реализована в тесте `[cavern2-simple]` в файле `cavern_2d_simple_test.cpp`.

Программа проводит итерации стартуя от начального нулевого состояния  $u = v = p = 0$  до тех пор, пока невязка не достигнет заданного порога. На каждой итерации поле давления и векторное поле скорости сохраняются на основной сетке в файл `cavern2.vtk.series`.

Итоговый результат (для  $\varepsilon = 10^{-2}$ ) представлен на рис. 9.

Для отображения вектора поля скорости в Paraview см. справку в [B.3.5](#).

Для работы с разнесённой сеткой в классе `cfd::RegularGrid2D` представлены функции

- `cfd::RegularGrid2D::cell_centered_grid()` – построить сетку по центрам ячеек (“чёрную” сетку для  $p$ ),
- `cfd::RegularGrid2D::xface_centered_grid()` – построить сетку по центрам  $x$ -граней (“синюю” сетку для  $v$ ),
- `cfd::RegularGrid2D::yface_centered_grid()` – построить сетку по центрам  $y$ -граней (“красную” сетку для  $u$ ),

и функции перевода индексов

- `cfd::RegularGrid2D::cell_centered_grid_index_ip_jp` – посчитать линейный индекс “чёрной” сетки ([4.16](#)),
- `cfd::RegularGrid2D::xface_grid_index_ip_j` – посчитать линейный индекс “синей” сетки ([4.18](#)),
- `cfd::RegularGrid2D::yface_grid_index_i_jp` – посчитать линейный индекс “красной” сетки ([4.17](#)).

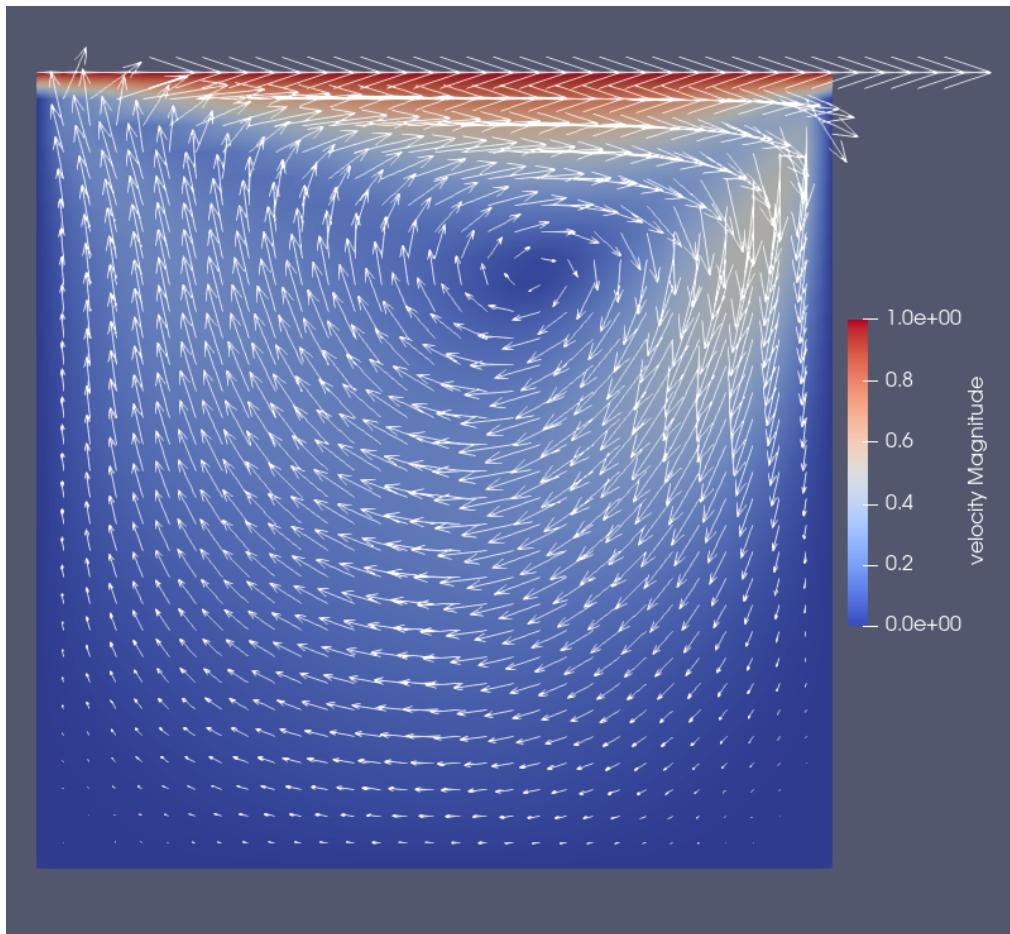


Рис. 9: Область расчёта задачи о каверне

#### 4.2.2 Функция верхнего уровня

```
446 TEST_CASE("Cavern 2D, SIMPLE algorithm", "[cavern2-simple]"){
```

Сначала устанавливаются параметры задачи: число Рейнольдса,

```
450 double Re = 100;
```

параметры алгоритма SIMPLE,

```
451 double tau = 0.03;
452 double alpha = 0.8;
```

разбиение сетки,

```
453 size_t n_cells = 30;
```

максимальное количество итераций

```
454     size_t max_it = 10000;
```

и значение невязки, при котором итерации прекращаются

```
455     double eps = 1e-0;
```

Затем происходит инициализация решателя, который определён в классе `Cavern2DSimpleWorker`

```
458     Cavern2DSimpleWorker worker(Re, n_cells, tau, alpha);
```

и параметров сохранения. Здесь первым параметром является флаг сохранения точных сеточных значений, который установлен в `false`, а также имя файла с итоговым результатом. Таким образом сохраняться будет только решение, интерполированное на основную сетку. Для целей отладки программы (для просмотра действительных, не интерполированных полей решения) следует первый флаг установить в `true`. Тогда помимо `cavern2.vtk.series`, будут создаваться также файлы `cavern2-u`, `cavern2-v`, `cavern2-p`.

```
459     worker.initialize_saver(false, "cavern2");
```

Потом происходит установка начальных значений искомых сеточных векторов:  $u = v = p = 0$

```
462     std::vector<double> u_init(worker.u_size(), 0.0);
463     std::vector<double> v_init(worker.v_size(), 0.0);
464     std::vector<double> p_init(worker.p_size(), 0.0);
465     worker.set_uvp(u_init, v_init, p_init);
```

и начинается итерационный процесс.

```
470     for (it=1; it < max_it; ++it){
```

Внутри цикла выполняется шаг итерационного процесса, который возвращает значение итоговой невязки в переменную `nrm`.

```
471     double nrm = worker.step();
```

На печать выводится индекс итерации, значение невязки и значение давления в правом верхнем узле (для контроля сходимости)

```
474     std::cout << it << " " << nrm << " " << worker.pressure().back() << std::endl;
```

Сохраняется состояние решателя на пройденную итерацию

```
477     worker.save_current_fields(it);
```

и производится проверка на сходимость

```
480     if (nrm < eps){  
481         break;  
482     }
```

В конце производится проверка: при установленных параметрах решение должно сойтись за 9 итераций:

```
484     CHECK(it == 9);
```

#### 4.2.3 Поля класса решателя

Класс `Cavern2DSimpleWorker` хранит в себе набор полей, характеризующих состояние итерационного процесса. Некоторые из этих полей (параметры решателя) постоянны (`const`) и определяются непосредственно перед вызовом конструктора в инициализаторе. Другие меняются с продвижением по итерациям.

Среди постоянных полей заданы 4 сетки: основная

`_grid`, “чёрная” сетка `_cc_grid` (cell-centered) для давления, “красная” сетка `_yf_grid` (y-face) для  $u$ , “синяя” сетка `_xf_grid` (x-face) для  $v$  (рис. 7).

```
32     const RegularGrid2D _grid;  
33     const RegularGrid2D _cc_grid;  
34     const RegularGrid2D _xf_grid;  
35     const RegularGrid2D _yf_grid;
```

Далее заданы скалярные параметры: число Рейнольдса, шаги сетки и параметры алгоритма SIMPLE

```
36     const double _hx;  
37     const double _hy;  
38     const double _Re;
```

```
39 const double _tau;
40 const double _alpha_p;
```

Далее следуют сеточные вектора, характеризующие текущее состояние решателя: найденные на последней итерации давление и скорости.

```
42 std::vector<double> _p;
43 std::vector<double> _u;
44 std::vector<double> _v;
```

Также определяется данные для решения системы уравнений для нахождения  $p'$  (4.25): значения  $d^u, d^v$ , а так же инициализированный решатель системы уравнений. Поскольку используется постоянные шаги по времени,  $d^u, d^v$  являются скалярами.

```
46 double _du;
47 double _dv;
48 AmgMatrixSolver _p_stroke_solver;
```

Хранятся левая и правая части систем уравнений (4.20), (4.22) для определения пробных значений скорости и расчета невязки.

```
50 CsrMatrix _mat_u;
51 CsrMatrix _mat_v;
52 std::vector<double> _rhs_u;
53 std::vector<double> _rhs_v;
```

Указатели на классы, помогающие сохранять найденные вектора в vtk - формат. Эти классы инициализируются только в случае, если пользователь указал на необходимость сохранения.

```
55 std::shared_ptr<VtkUtils::TimeSeriesWriter> _writer_u;
56 std::shared_ptr<VtkUtils::TimeSeriesWriter> _writer_v;
57 std::shared_ptr<VtkUtils::TimeSeriesWriter> _writer_p;
58 std::shared_ptr<VtkUtils::TimeSeriesWriter> _writer_all;
```

#### 4.2.4 Инициализация решателя

В секции инициализации конструктора создаются сетки в единичном квадрате и переписываются параметры решения. Далее в теле конструктора вычисляются значения  $d^u, d^v$  по формулам (4.29),

(4.30) и собирается решатель для  $p'$ . Как было указано ранее, матрица системы  $A^p$  не меняется с продвижением по итерациям, поэтому этот решатель можно собрать один раз до начала счёта.

```

77 Cavern2DSimpleWorker::Cavern2DSimpleWorker(double Re, size_t n_cells, double tau,
    ↪ double alpha_p):
78     _grid(0, 1, 0, 1, n_cells, n_cells),
79     _cc_grid(_grid.cell_centered_grid()),
80     _xf_grid(_grid.xface_centered_grid()),
81     _yf_grid(_grid.yface_centered_grid()),
82     _hx(1.0/n_cells),
83     _hy(1.0/n_cells),
84     _Re(Re),
85     _tau(tau),
86     _alpha_p(alpha_p)
87 {
88     _du = 1.0 / (1 + 2.0*_tau/_Re * (1.0/_hx/_hx + 1.0/_hy/_hy));
89     _dv = 1.0 / (1 + 2.0*_tau/_Re * (1.0/_hx/_hx + 1.0/_hy/_hy));
90     assemble_p_stroke_solver();
91 }
```

Начальные значения устанавливаются через вызов функции `set_upv`. Эти начальные значения будут использоваться в качестве значений с предыдущего итерационного слоя на первой итерации.

В функции происходит переписывание переданных векторов в приватные поля класса.

```

102 double Cavern2DSimpleWorker::set_upv(const std::vector<double>& u, const
    ↪ std::vector<double>& v, const std::vector<double>& p){
103     _u = u;
104     _v = v;
105     _p = p;
```

После этого данных в классе-решателе достаточно, для сборки матриц  $A^u, A^v$  и правых частей  $b^u, b^v$  для системы уравнений (4.20), (4.22).

```

106     assemble_u_slae();
107     assemble_v_slae();
```

Если посмотреть на выражение для невязки (4.7) убрав в нём крышки над переменными, то можно убедится, что оно аппроксимируется в виде

$$r_u = \frac{1}{\tau} (A^u u - b^u).$$

Поэтому после сборки систем уравнений движения, можно вычислить невязку, характеризующую отклонение установленного в этой процедуре решения от желаемого:

```
108 // residuals
109 double nrm_u = compute_residual(_mat_u, _rhs_u, _u)/_tau;
110 double nrm_v = compute_residual(_mat_v, _rhs_v, _v)/_tau;
111
112 return std::max(nrm_u, nrm_v);
113 }
```

#### 4.2.5 Шаг итерации SIMPLE

Осуществляется в процедуре

```
115 double Cavern2DSimpleWorker::step(){
116     // Predictor step: U-star
117     std::vector<double> u_star = compute_u_star();
118     std::vector<double> v_star = compute_v_star();
119     // Pressure correction
120     std::vector<double> p_stroke = compute_p_stroke(u_star, v_star);
121     // Velocity correction
122     std::vector<double> u_stroke = compute_u_stroke(p_stroke);
123     std::vector<double> v_stroke = compute_v_stroke(p_stroke);
124     // Set final values
125     std::vector<double> u_new = vector_sum(u_star, 1.0, u_stroke);
126     std::vector<double> v_new = vector_sum(v_star, 1.0, v_stroke);
127     std::vector<double> p_new = vector_sum(_p, _alpha_p, p_stroke);
128
129     return set_uvp(u_new, v_new, p_new);
130 }
```

и представляет собой буквальное пошаговое следование алгоритму SIMPLE (4.1.2.1). В конце опять вызывается функция `set_uvp` для сборки матриц для следующей итерации и подсчёта невязки на текущей итерации.

#### 4.2.6 Сборка системы уравнений для поправки давления

Сборка системы уравнений (4.25) осуществляется в процедуре

```
160 void Cavern2DSimpleWorker::assemble_p_stroke_solver(){
```

Сборка происходит с использованием матрицы формата `cfd::LodMatrix`, удобного для непоследовательной записи.

```
161     LodMatrix mat(p_size());
```

Заполнение происходит в цикле по раздвоенным индексам  $ij$  “чёрной” сетки для давления:

```
162     for (size_t j = 0; j < _cc_grid.ny() + 1; ++j)
163         for (size_t i = 0; i < _cc_grid.nx() + 1; ++i){
```

Внутри цикла устанавливаются флаги, характеризующие граничный статус текущего узла

```
164     bool is_left = (i == 0);
165     bool is_right = (i == _cc_grid.nx());
166     bool is_bottom = (j == 0);
167     bool is_top = (j == _cc_grid.ny());
```

Вычисляется значение сквозного индекса по формуле (4.16)

```
169     size_t ind0 = _grid.cell_centered_grid_index_ip_jp(i, j);
```

и значения коэффициентов в формулах (4.26). Поскольку сетка равномерная, эти значения не меняются для разных узлов

```
170     double coef_x = _du/_hx/_hx;
171     double coef_y = _dv/_hy/_hy;
```

Далее формулы (4.26) применяются для заполнения матриц с учётом аппроксимированного граничного условия (4.38). Так, запись

```
172     // x
173     if (!is_right){
174         size_t ind1 = _grid.cell_centered_grid_index_ip_jp(i+1, j);
175         mat.add_value(ind0, ind0, coef_x);
176         mat.add_value(ind0, ind1, -coef_x);
177     }
```

для всех неправых узлов с линейным индексом

`ind0` вычисляет индекс узла, расположенного правее него с линейным индексом `ind1`, добавляет слагаемое в диагональный (первое из уравнений (4.26)) и вычитает из недиагонального (четвёртое из уравнений (4.26)) элемента строки `ind0`. Для правых узлов работает граничное условие (4.26) и выполнять эту процедуру не нужно.

После заполнения в матрицу вводится граничное условие (4.39)

```
195     mat.set_unit_row(0);
```

И матрица передаётся в решатель СЛАУ предварительно сконвертированная в формат

```
cfd::CsrMatrix
```

```
196     _p_stroke_solver.set_matrix(mat.to_csr());
```

Правая часть собирается заново на каждой итерации по формуле (4.28). Её реализация представлена в функции

```
345 std::vector<double> Cavern2DSimpleWorker::compute_p_stroke(const std::vector<double>&
→ u_star, const std::vector<double>& v_star){
```

Сначала собирается правая часть системы (4.25) по формуле (4.28):

```
347     for (size_t i = 0; i < _grid.nx(); ++i)
348         for (size_t j = 0; j < _grid.ny(); ++j){
349             size_t ind0 = _grid.cell_centered_grid_index_ip_jp(i, j);
350             size_t ind_left = _grid.yface_grid_index_i_jp(i, j);
351             size_t ind_right = _grid.yface_grid_index_i_jp(i+1, j);
352             size_t ind_bot = _grid.xface_grid_index_ip_j(i, j);
353             size_t ind_top = _grid.xface_grid_index_ip_j(i, j+1);
354             rhs[ind0] = -(u_star[ind_right] - u_star[ind_left])/_tau/_hx - (v_star[ind_top] -
→ v_star[ind_bot])/_tau/_hy;
355         }
```

Потом осуществляется установка граничного условия (4.39)

```
356     rhs[0] = 0;
```

и вызывается решатель СЛАУ

```
357     std::vector<double> p_stroke;
358     _p_stroke_solver.solve(rhs, p_stroke);
359     return p_stroke;
360 }
```

#### 4.2.7 Сборка системы уравнений для пробной скорости

Сборка системы (4.20) (как правой, так и левой частей) реализована в функции

```
199 void Cavern2DSimpleWorker::assemble_u_slae(){
```

Основной цикл идёт по негрничным узлам “красной” сетки, в котором реализуются формулы (4.21)

```

233 for (size_t j=0; j < _grid.ny(); ++j)
234 for (size_t i=1; i < _grid.nx(); ++i){
235     size_t row_index = _grid.yface_grid_index_i_jp(i, j); // [i, j+1/2]
236
237     double u0_plus    = u_ip_jp(i, j); // u[i+1/2, j+1/2]
238     double u0_minus   = u_ip_jp(i-1, j); // u[i-1/2, j+1/2]
239     double v0_plus    = v_i_j(i, j+1); // v[i, j+1]
240     double v0_minus   = v_i_j(i, j); // v[i, j]
241
242     // u_(i, j+1/2)
243     add_to_mat(row_index, {i, j}, 1.0);
244     // + tau * d(u0*u)/ dx
245     add_to_mat(row_index, {i+1, j}, _tau/2.0/_hx*u0_plus);
246     add_to_mat(row_index, {i-1, j}, -_tau/2.0/_hx*u0_minus);
247     // + tau * d(v0*u)/dy
248     add_to_mat(row_index, {i, j+1}, _tau/2.0/_hy*v0_plus);
249     add_to_mat(row_index, {i, j-1}, -_tau/2.0/_hy*v0_minus);
250     // - tau / Re * d^2u/dx^2
251     add_to_mat(row_index, {i, j}, 2.0*_tau/_Re/_hx/_hx);
252     add_to_mat(row_index, {i+1, j}, -_tau/_Re/_hx/_hx);
253     add_to_mat(row_index, {i-1, j}, -_tau/_Re/_hx/_hx);
254     // - tau / Re * d^2u/dy^2
255     add_to_mat(row_index, {i, j}, 2.0*_tau/_Re/_hy/_hy);
256     add_to_mat(row_index, {i, j+1}, -_tau/_Re/_hy/_hy);
257     add_to_mat(row_index, {i, j-1}, -_tau/_Re/_hy/_hy);
258     // = u0_(i, j+1/2)
259     _rhs_u[row_index] += _u[row_index];
260     // - tau * dp/dx
261     _rhs_u[row_index] -= _tau/_hx*(p_ip_jp(i, j) - p_ip_jp(i-1, j));
262 }
```

Как было отмечено в пункте 4.1.3.5, граничные условия первого рода в этом уравнении учитываются двумя разными способами: узлы расположенные непосредственно на границе (нижней и верхней) учитываются по схеме (4.35), которая реализована в цикле

```

222 for (size_t j=0; j< _grid.ny(); ++j){
223     size_t index_left = _grid.yface_grid_index_i_jp(0, j);
224     add_to_mat(index_left, {0, j}, 1.0);
225     _rhs_u[index_left] = 0.0;
```

```

226
227     size_t index_right = _grid.yface_grid_index_i_jp(_grid.nx(), j);
228     add_to_mat(index_right, {_grid.nx(), j}, 1.0);
229     _rhs_u[index_right] = 0.0;
230 }

```

А фиктивные узлы, возникающие при обработке узлов расположенных в полушаге от границ (левой и правой), обрабатываются по схеме (4.37). Эта схема реализована в виде препроцессинга алгоритма добавления элемента в матрицу в лямбда-функции

```

204 auto add_to_mat = [&](size_t row_index, std::array<size_t, 2> ij_col, double value){
205     if (ij_col[1] == _grid.ny()){
206         // ghost index => top boundary condition: u = 1
207         size_t ind1 = _grid.yface_grid_index_i_jp(ij_col[0], ij_col[1]-1);
208         mat.add_value(row_index, ind1, -value);
209         _rhs_u[row_index] -= 2.0*value;
210     } else if (ij_col[1] == (size_t)-1){
211         // ghost index => bottom boundary condition: u = 0
212         size_t ind1 = _grid.yface_grid_index_i_jp(ij_col[0], ij_col[1]+1);
213         mat.add_value(row_index, ind1, -value);
214         _rhs_u[row_index] -= 2.0*value;
215     } else {
216         size_t ind1 = _grid.yface_grid_index_i_jp(ij_col[0], ij_col[1]);
217         mat.add_value(row_index, ind1, value);
218     }
219 };

```

Эта лямбда вызывается везде, где нужно добавить в строку `row_index` и колонку, соответствующую узлу `ij_col`, значение `value`. Она перехватывает ситуации с “фиктивным” узлом ( $j = -1, j = n_y$ ) и применяет алгоритм (4.37).

## 4.3 Задание для самостоятельной работы

- Подобрать оптимальные параметры алгоритма SIMPLE  $\tau, \alpha_p$  для задачи в каверне, при которых сходимость происходит за наименьшее число итераций. Для этого лучше понизить пороговый  $\varepsilon = 0.01$ . Сравнить полученные вами эмпирически значения с рекомендованными. Увеличить разбиение и отметить, как величина шага по пространству влияет на количество требуемых итераций. Для ускорения параметрических расчётов лучше собирать программу в “релизной” (B.1.3) версии и убрать сохранение в vtk внутри каждой итерации.

2. Нарисовать поле невязок  $r_u, r_v$  в динамике. Отметить в каком из уравнений и в каких местах области расчёта наблюдаются наибольшие проблемы со сходимостью.
3. Решить аналогичную задачу, в которой скорость не только на верхней, но и на нижней стенке равна  $U = 1$ . Для этого завести новый тест

[cavern2-simple-sym] и новый тестовый рабочий класс `Cavern2DSimpleSymWorker`, который отнаследовать от существующего класса `Cavern2DSimpleWorker`. Для того, чтобы при реализации нового класса не повторять существующий код, а пользоваться механизмом перегрузок виртуальных функций, необходимо будет произвести небольшой рефакторинг: ввести новый приватный метод класса `Cavern2DSimpleWorker`

```
1 virtual double get_bottom_velocity() const;
```

## 5 Лекция 5 (6.10)

### 5.1 Оптимальные значения параметров алгоритма SIMPLE

Введем обозначение

$$E = \frac{4\tau}{\operatorname{Re} H(h_x^2, h_y^2)}$$

где  $H(a, b)$  – среднее гармоническое, определяемое как

$$H(a, b) = \frac{2ab}{a+b}.$$

Значение  $E$  – есть диагональное компонента матрицы диффузии в аппроксимированных уравнениях движения (второе слагаемое в правой части первой формулы (4.21)).

При независимом задании релаксаций по скорости и давлению, оптимальной сходимости соответствуют значения

$$\begin{aligned} E = 1 \quad \Rightarrow \quad \tau &= \frac{\operatorname{Re} H(h_x^2, h_y^2)}{4}, \\ \alpha_p &= 0.8. \end{aligned}$$

Ещё более эффективной сходимости соответствуют параметры

$$E \approx 4, \quad \alpha_p = \frac{1}{1+E}. \quad (5.1)$$

Это выражение соответствует алгоритму SIMPLEC (согласованный алгоритм, SIMPLE Consistent).

### 5.2 Нестационарное уравнение Навье-Стокса

Запишем безразмерную систему (4.1) – (4.3) в нестационарной постановке

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{\partial u^2}{\partial x} + \frac{\partial uv}{\partial y} &= -\frac{\partial p}{\partial x} + \frac{1}{\operatorname{Re}} \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right), \\ \frac{\partial v}{\partial t} + \frac{\partial uv}{\partial x} + \frac{\partial v^2}{\partial y} &= -\frac{\partial p}{\partial y} + \frac{1}{\operatorname{Re}} \left( \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right), \\ \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} &= 0. \end{aligned} \quad (5.2)$$

Характерное время, на которое было произведено обезразмериваение, равно  $t^0 = L/U$ .

#### 5.2.1 Схема расчёта по алгоритму SIMPLE

Производную по времени будет аппроксимировать по двухслойной неявной схеме.

$$\frac{\partial u}{\partial t} = \frac{\hat{u} - \check{u}}{\Delta t} + o(\Delta t),$$

где символом  $\check{\cdot}$  обозначены значения с предыдущего временного слоя.

Внутри каждого временного слоя будем исполнять итерационный процесс по типу (4.4) – (4.6) с добавлением дискретизованной производной по времени:

$$\begin{aligned} \frac{\hat{u} - \check{u}}{\Delta t} + \frac{\hat{u} - u}{\tau} + \frac{\partial u \hat{u}}{\partial x} + \frac{\partial v \hat{u}}{\partial y} &= -\frac{\partial \hat{p}}{\partial x} + \frac{1}{\text{Re}} \left( \frac{\partial^2 \hat{u}}{\partial x^2} + \frac{\partial^2 \hat{u}}{\partial y^2} \right), \\ \frac{\hat{v} - \check{v}}{\Delta t} + \frac{\hat{v} - v}{\tau} + \frac{\partial u \hat{v}}{\partial x} + \frac{\partial v \hat{v}}{\partial y} &= -\frac{\partial \hat{p}}{\partial y} + \frac{1}{\text{Re}} \left( \frac{\partial^2 \hat{v}}{\partial x^2} + \frac{\partial^2 \hat{v}}{\partial y^2} \right), \\ \frac{\partial \hat{u}}{\partial x} + \frac{\partial \hat{v}}{\partial y} &= 0. \end{aligned} \quad (5.3)$$

Далее на основе этих уравнений проведём рассуждения, аналогичные приведённым в п. 4.1.2.1. Уравнения для пробной скорости типа (4.9), (4.10) примут вид

$$\begin{aligned} \left( 1 + \frac{\tau}{\Delta t} \right) u^* + \tau \frac{\partial u u^*}{\partial x} + \tau \frac{\partial v u^*}{\partial y} - \frac{\tau}{\text{Re}} \left( \frac{\partial^2 u^*}{\partial x^2} + \frac{\partial^2 u^*}{\partial y^2} \right) &= -\tau \frac{\partial p}{\partial x} + u + \frac{\tau}{\Delta t} \check{u}, \\ \left( 1 + \frac{\tau}{\Delta t} \right) v^* + \tau \frac{\partial u v^*}{\partial x} + \tau \frac{\partial v v^*}{\partial y} - \frac{\tau}{\text{Re}} \left( \frac{\partial^2 v^*}{\partial x^2} + \frac{\partial^2 v^*}{\partial y^2} \right) &= -\tau \frac{\partial p}{\partial y} + v + \frac{\tau}{\Delta t} \check{v}. \end{aligned} \quad (5.4)$$

Уравнения для поправок скорости (4.12), (4.13) и давления (4.14) можно оставить в неизменном виде если модифицировать входящие в них множители  $d^u, d^v$ . По аналогии с (4.29), (4.30) запишем

$$\begin{aligned} d^u &= (\text{diag}(A^u))^{-1} = \left( 1 + \frac{\tau}{\Delta t} + \frac{2\tau}{\text{Re}} \left( \frac{1}{h_x^2} + \frac{1}{h_y^2} \right) \right)^{-1} \\ d^v &= (\text{diag}(A^v))^{-1} = \left( 1 + \frac{\tau}{\Delta t} + \frac{2\tau}{\text{Re}} \left( \frac{1}{h_x^2} + \frac{1}{h_y^2} \right) \right)^{-1}. \end{aligned} \quad (5.5)$$

Здесь  $A^u, A^v$  – матрицы левых частей выражений (5.4).

Схема расчёта на временнóм слое остаётся аналогичной стационарному случаю, с той разницей, что первая итерация использует значение расчётных полей с предыдущего шага по времени. Порядок действий на временном слое:

1. Присвоить  $u = \check{u}, v = \check{v}, p = \check{p}$ ;
2. Из уравнений (5.4) вычислить значения  $u^*, v^*$ ;
3. Определить поправку давления  $p'$  из уравнения (4.14) с использованием (5.5);
4. Найти поправки скорости  $u', v'$  из выражений (4.12), (4.13) с использованием (5.5);
5. Выразить значения переменных для текущего слоя из (4.8); Для определения давления использовать сглаживание с коэффициентом  $\alpha_p$ ;
6. Найти невязку с использованием найденных значений  $\hat{u}, \hat{v}, \hat{p}$  из выражения (4.7). Если она недостаточно мала, то выполняется присваивание  $u = \hat{u}, v = \hat{v}, p = \hat{p}$  и возвращение на шаг 2. Если сходимость достигнута, то перейти на следующий шаг по времени. Для этого выполнить  $\check{u} = \hat{u}, \check{v} = \hat{v}, \check{p} = \hat{p}$  и перейти на шаг 1.

### 5.3 Завихренность и функция тока

Для несжимаемых течений ( $\nabla \cdot \mathbf{u} = 0$ ) можно ввести векторный потенциал скорости:

$$\nabla \times \Psi = \mathbf{u}$$

Также определим векторное поле завихренности как

$$\boldsymbol{\Omega} = \nabla \times \mathbf{u}$$

Компоненты этого вектора характеризуют вращательную составляющую скорости в плоскостях, перпендикулярных соответствующим базисным векторам. Например, компонента  $\Omega_z$  характеризует вращение в плоскости  $xy$ .

Подставив векторный потенциал в выражение для завихренности, получим связь между двумя введёнными векторными полями

$$\boldsymbol{\Omega} = \nabla \times (\nabla \times \Psi) = \nabla(\nabla \cdot \Psi) - \nabla^2 \Psi.$$

Далее рассмотрим двумерные течения в декартовой системе координат.  $z$ -компоненты последнего выражения (с учётом  $\partial/\partial z = 0$ ) сократится до

$$\Omega_z = -\nabla^2 \Psi_z$$

или, введя обозначения  $\Omega_z = \omega$ ,  $\Psi_z = \psi$ :

$$\begin{aligned} \frac{\partial \psi}{\partial y} &= u, \\ -\frac{\partial \psi}{\partial x} &= v, \\ \omega &= \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} \end{aligned} \tag{5.6}$$

и расписывая оператор Лапласа покоординатно

$$-\left(\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2}\right) = \omega. \tag{5.7}$$

Скалярное поле  $\psi$  называется функцией тока и его изолинии совпадают с линиями тока течения. Действительно, введём прямоугольную систему координат  $(\mathbf{n}, \mathbf{s})$  вдоль линии тока (вектор  $\mathbf{s}$  – касательный к линии тока,  $\mathbf{n}$  – нормаль) и перепишем определение (5.6) в этой системе:

$$\begin{aligned} \frac{\partial \psi}{\partial s} &= u_n, \\ -\frac{\partial \psi}{\partial n} &= u_s. \end{aligned} \tag{5.8}$$

По определению, вдоль линии тока  $u_n = 0$ , отсюда получим  $\psi = \text{const.}$

### 5.3.1 Определение завихренности и функции тока на разнесённой сетке

Исходя из определения завихренности (5.6), легко видеть что аппроксимировать вторым порядком точности её проще всего на основной сетке  $ij$ . Для внутренних узлов можно записать:

$$\omega_{i,j} = \frac{v_{i+\frac{1}{2},j} - v_{i-\frac{1}{2},j}}{h_x} - \frac{u_{i,j+\frac{1}{2}} - u_{i,j-\frac{1}{2}}}{h_y}. \quad (5.9)$$

Для граничных узлов возможно (при известных значениях скорости на границах) использовать направленные разности. Например, для  $i = 0$ :

$$\omega_{0,j} = \frac{v_{\frac{1}{2},j} - v_{0,j}}{h_x/2} - \frac{u_{i,j+\frac{1}{2}} - u_{i,j-\frac{1}{2}}}{h_y}. \quad (5.10)$$

Получив сеточный вектор для завихрённости  $\omega$  можно, исходя из (5.7) записать разностную схему для определения функции тока во внутренних узлах сетки:

$$\frac{-\psi_{i-1,j} + 2\psi_{i,j} - \psi_{i+1,j}}{h_x^2} + \frac{-\psi_{i,j-1} + 2\psi_{i,j} - \psi_{i,j+1}}{h_y^2} = \omega_{i,j}. \quad (5.11)$$

На границах необходимо воспользоваться соотношениями (5.8). Из этих соотношениях функция тока вычисляется с точностью до константы (для каждой границы своей). Одну из этих констант можно положить нулём. Остальные вычисляются прямым интегрированием. Например, рассмотрим течение в области высотой  $Y$  с непротекаемыми горизонтальными границами и двумя препятствиями (рис. 10)

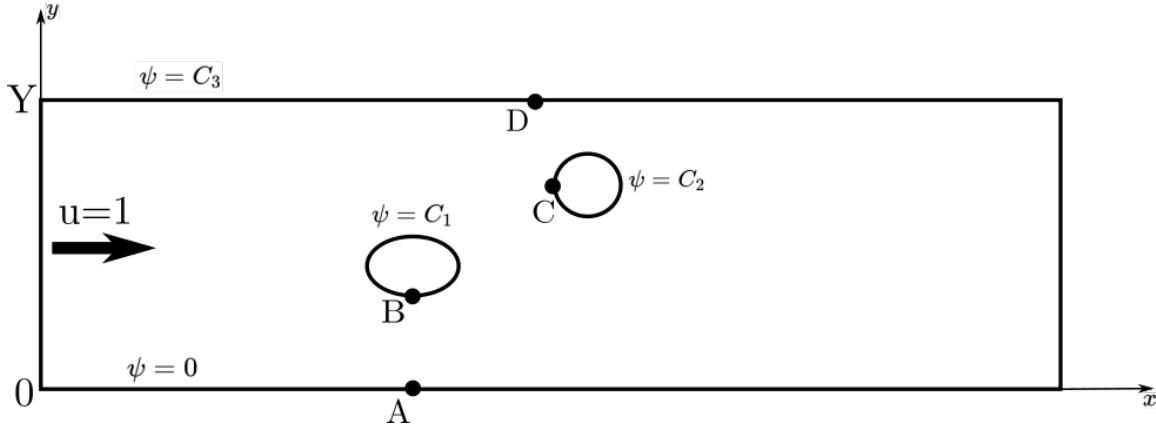


Рис. 10: Область течения в задаче обтекания

Пусть скорость набегающего потока равна единице на всей входной границе. На нижней границе положим  $\psi_A = 0$ . На входной границе получим

$$\frac{\partial \psi}{\partial y} = 1 \quad \Rightarrow \quad \psi_{in}(y) = y + \psi_A = y. \quad (5.12)$$

Исходя из значения  $\psi_{in}$  на верхней границе можно записать

$$\psi_D = \psi_{in}(Y) = Y.$$

Для определения  $\psi$  на первом из двух препятствий рассмотрим траекторию  $AB$ :

$$\frac{\partial \psi}{\partial s} = u_n \quad \Rightarrow \quad \psi_B = \int_A^B u_n ds + \psi_A = Q_{AB},$$

где  $s$  - касательная к этой траектории,  $n$  – нормаль к ней,  $Q_{AB}$  – расход жидкости поперёк траектории  $AB$ . Вследствии несжимаемости можно использовать любую из возможных положений точек  $A, B$  на границах и любую из реализаций траектории.

Аналогично для второго тела можно записать

$$\psi_C = Q_{AC} = Q_{AB} + Q_{BC}.$$

## 5.4 Задание для самостоятельной работы

Для расчитанного ранее течения в каверне расчитать скалярные поля функции тока и завихренности.

- Сначала для вычисления завихренности использовать формулы (5.9) для внутренних и (5.10) для граничных узлов.
- Затем полученную завихренность использовать для вычисления  $\psi$ . Для этого необходимо собрать и решить систему линейных уравнений, где внутренним узлам будет соответствовать разностная схема (5.11), а для граничных – условие  $\psi_{i,j} = 0$ .
- Полученные поля сохранить в vtk, добавив строки

```
VtkUtils::add_point_data(psi, "psi", filepath); // найденный вектор psi
VtkUtils::add_point_data(omega, "omega", filepath); // найденный вектор omega
```

в функцию сохранения на основной сетке

```
void Cavern2DSimpleWorker::save_current_fields(size_t iter){
    if (_writer_all){
        ...
    }
}
```

# 6 Лекция 6 (13.10)

## 6.1 Оптимизация методов решения СЛАУ

В рассмотренном ранее методе расчёта двумерного течения вязкой несжимаемой жидкости на каждой итерации необходимо решить три системы слийных уравнений: (4.20), (4.22), (4.25). Причём левые части первых двух систем уравнений меняются от итерации к итерации, в то время как левая часть третьей остаётся постоянной.

Последнее обстоятельство накладывает некоторые ограничения на оптимальный выбор решателя сеточных систем линейных уравнений.

В частности, для систем (4.20), (4.22) не следует использовать решатели с большим временем инициализации (этап инициализации требуется решателю на этапе задания матрицы левой части).

Для системы (4.25), напротив, можно использовать решатели с дорогой инициализацией, так как она проводится один раз до начала итераций SIMPLE.

В рассмотренных ранее примерах использовался алгебраический многосеточный итерационный решатель, который имеет существенное время инициализации. Ниже рассмотрим некоторые более простые итерационные способы решения систем уравнений, которые, хотя и имеют значительно худшую сходимость, но не требуют дорогой инициализации.

Поскольку итерации для решения СЛАУ являются внутренними относительно SIMPLE-итераций, то при использовании этих решателей не требуется доводить их до полной сходимости. Достаточно сделать один-два шага. А полную сходимость можно "переложить" на вышестоящий итерационный процесс.

### 6.1.1 Метод Якоби

Будем рассматривать систему уравнений вида

$$\sum_{j=0}^{N-1} A_{ij} u_j = r_i, \quad i = \overline{0, N-1}$$

относительно неизвестного сеточного вектора  $\{u\}$ .

В классическом виде алгоритм Якоби формулируется в виде

$$\hat{u}_i = \frac{1}{A_{ii}} \left( r_i - \sum_{j \neq i} A_{ij} u_j \right)$$

Произведём некоторые преобразования

$$\begin{aligned} \hat{u}_i &= \frac{1}{A_{ii}} \left( r_i - \sum_j A_{ij} u_j + A_{ii} u_i \right) \\ &= u_i + \frac{1}{A_{ii}} \left( r_i - \sum_j A_{ij} u_j \right) \end{aligned}$$

Таким образом, программировать итерацию этого алгоритма, обновляющую значения массива

$\{u\}$ , можно в виде

```

 $\hat{u} = u;$ 
for  $i = \overline{0, N - 1}$ 
 $\hat{u}_i += \frac{1}{A_{ii}} \left( r_i - \sum_{j=0}^{N-1} A_{ij} u_j \right)$ 
endfor
 $u = \hat{u};$ 

```

### 6.1.2 Метод Зейделя

Формулируется в виде

$$\hat{u}_i = \frac{1}{A_{ii}} \left( r_i - \sum_{j < i} A_{ij} \hat{u}_j - \sum_{j > i} A_{ij} u_j \right).$$

Поскольку этот метод неявный относительно уже найденных на итерации значений, то в отличии от метода Якоби этот алгоритм не требует создания временного массива  $\hat{u}$  при программировании. Псевдокод для реализации итерации этого метода можно записать как

```

for  $i = \overline{0, N - 1}$ 
 $u_i += \frac{1}{A_{ii}} \left( r_i - \sum_{j=0}^{N-1} A_{ij} u_j \right)$ 
endfor

```

### 6.1.3 Метод последовательных верхних релаксаций (SOR)

Этот метод основан на добавлении к решению результатов итераций Зейделя с коэффициентом  $\omega > 1$ . То есть он изменяет решение по тому же принципу, что и метод Зейделя, но искусственно увеличивает эту добавку.

Формулируется этот метод в виде

$$\hat{u}_i = (1 - \omega) u_i + \frac{\omega}{A_{ii}} \left( r_i - \sum_{j < i} A_{ij} \hat{u}_j - \sum_{j > i} A_{ij} u_j \right).$$

Для устойчивости метода необходимо  $\omega < 2$ . Обычно используют  $\omega = 1.95$ .

Итерация этого метода по аналогии с методом Зейделя может быть запрограммирована в виде

```

for  $i = \overline{0, N - 1}$ 
     $u_i += \frac{\omega}{A_{ii}} \left( r_i - \sum_{j=0}^{N-1} A_{ij} u_j \right)$ 
endfor

```

#### 6.1.4 Формат хранения разреженных матриц CSR

При реализации решателей систем сеточных уравнений важно учитывать разреженный характер используемых в левой части. То есть избегать хранения и ненужных операций с нулевыми элементами матрицы.

Хотя рассмотренные ранее алгоритмы конечноразностных аппроксимаций на структурированных сетках давали трёх- (для одномерных задач) или пятидиагональную (для двумерных) сеточную матрицу, здесь будем рассматривать общие форматы хранения, не привязанные к конкретному шаблону.

Любой общий формат хранения должен хранить информацию о шаблоне матрице (адресах ненулевых элементов) и значениях матричных коэффициентов в этом шаблоне.

В CSR (Compressed sparse rows) формате все ненулевые элементы хранятся в линейном массиве `vals`. А шаблон матрицы – в двух массивах

- массиве колонок `cols` – значений колонок для соответствующих ему значений из массива `vals`,
- массиве адресов `addr` – индексах массива `vals`, с которых начинается описание соответствующей строки.

В конце массива `addr` добавляется общая длина массива `vals`.

Таким образом, длины массивов `vals`, `cols` равны количеству ненулевых элементов матрицы, а длина массива `addr` равна количеству строк в матрице плюс один.

Для облегчения процедур поиска описание каждой строки должно идти последовательно с увеличением индекса колонки.

Для примера рассмотрим следующую матрицу

$$\begin{pmatrix} 2.0 & 0 & 0 & 1.0 \\ 0 & 3.0 & 5.0 & 4.0 \\ 0 & 0 & 6.0 & 0 \\ 0 & 7.0 & 0 & 8.0 \end{pmatrix}$$

Массивы, описывающие матрицу в формате CSR примут вид

	$row = 0$	$row = 1$	$row = 2$	$row = 3$	
<code>vals</code> =	2.0, 1.0,	3.0, 5.0, 4.0,	6.0,	7.0, 8.0	
<code>cols</code> =	0, 3,	1, 2, 3,	2,	1, 3	
<code>addr</code> =	0,	2,	5,	6,	8

Рассмотрим реализацию базовых алгоритмов для матриц, заданных в этом формате.

Пусть матрица задана следующими массивами:

```
std::vector<double> vals; // массив значений  
std::vector<size_t> cols; // массив столбцов  
std::vector<size_t> addr; // массив адресов
```

Число строк в матрице:

```
size_t nrows = addr.size() - 1;
```

Число элементов в шаблоне (ненулевых элементов)

```
size_t n_nonzeros = vals.size();
```

Число ненулевых элементов в заданной строке ‘irow’

```
size_t n_nonzeros_in_row = addr[irow + 1] - addr[irow];
```

Умножение матрицы на вектор ‘v’ (длина этого вектора должна быть равна числу строк в матрице). Здесь реализуется суммирование вида

$$r_i = \sum_{j=0}^{N-1} A_{ij} v_j,$$

при этом избегаются лишние операции с нулями

```
// число строк в матрице и длина вектора v  
size_t nrows = addr.size() - 1;  
// массив ответов. Инициализируем нулями  
std::vector<double> r(nrows, 0);  
// цикл по строкам  
for (size_t irow = 0; irow < nrows; ++irow){  
    // цикл по ненулевым элементам строки irow  
    for (size_t a = addr[irow]; a < addr[irow + 1]; ++a){  
        // получаем индекс колонки  
        size_t icol = cols[a];  
        // значение матрицы на позиции [irow, icol]  
        double val = vals[a];  
        // добавляем к ответу  
        r[irow] += val * v[icol];  
    }  
}
```

Поиск значения элемента матрицы по адресу `(irow, icol)` с учётом локально сортированного вектора `cols`

```

using iter_t = std::vector<size_t>::const_iterator;
// указатели на начало и конец описания строки в массиве cols
iter_t it_start = cols.begin() + addr[irow];
iter_t it_end = cols.begin() + addr[irow+1];
// поиск значения icol в отсортированной последовательности [it_start, it_end)
iter_t fnd = std::lower_bound(it_start, it_end, icol);
if (fnd != it_end && *fnd == icol){
    // если нашли, то определяем индекс найденного элемента в массиве cols
    size_t a = fnd - cols.begin();
    // и возвращаем значение из vals по этому индексу
    return vals[a];
} else {
    // если не нашли, значит элемент [irow, icol] находится вне шаблона. Возвращаем 0
    return 0;
}

```

## 6.2 Задача об обтекании препятствия

### 6.2.1 Расчётная сетка

Рассмотрим постановку граничных условий и особенности пространственной аппроксимации для задачи о внешнем обтекании. Поскольку рассматриваемые нами методы пока ограничены аппроксимациями на структурированной прямоугольной сетке, то будем рассматривать такую область расчёта, которую легко можно отобразить на такой сетке. Пусть внешняя область расчёта и обтекаемое препятствие представляют из себя прямоугольники.

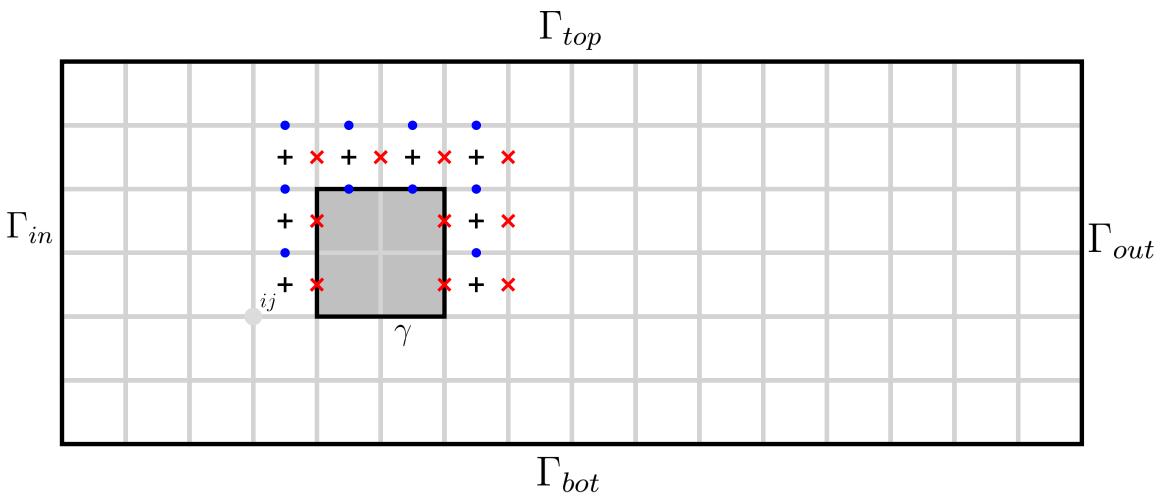


Рис. 11: Область расчёта и разнесённая сетка для задачи обтекания

По аналогии с рис. 7 введём в этой области прямоугольную сетку (рис. 11).

В случае сохранения естественной нумерации узлов и ячеек сетки часть их этих пронумерованных ячеек выпадает из области расчёта (попадает внутрь препятствия). Для таких случаев существует два способа работы с нумерацией:

- Можно сохранить естественную нумерацию, при этом часть ячеек пометить как неактивные (например, введя специальный массив признаков `actnum`,  $i$ -ый элемент которого равен единице для активной ячейки и нулю для неактивной). Количество элементов в сеточных векторах тогда будет равно общему количеству всех ячеек (и активных и неактивных). Но значения этих векторов в неактивных ячейках будут фиктивными (нулями).
- Можно нумеровать лишь активные ячейки (и узлы), тем самым нарушив естественную нумерацию.

Оба этих подхода имеют свои очевидные плюсы и минусы. Первый подход сохраняет простые зависимости для перевода двумерного индекса в сквозной и диагональную структуру сеточных матриц. Второй подход более экономичен в хранении данных.

### 6.2.2 Граничные условия

Рассмотрим постановку со следующими граничными условиями:

- во входном сечении зададим равномерный профиль скорости

$$(x, y) \in \Gamma_{in} : u = 1, v = 0; \quad (6.1)$$

- на нижней и верхней границах – условие симметрии (идеального скольжения). Это условие моделирует зеркальное отражение расчётной области относительно соответствующих границ  $\Gamma_{top}, \Gamma_{bot}$ .

$$(x, y) \in \Gamma_{top}, \Gamma_{bot} : \frac{\partial u}{\partial n} = 0, v = 0; \quad (6.2)$$

- на самом обтекаемом деле – условия прилипания

$$(x, y) \in \gamma : u = 0, v = 0; \quad (6.3)$$

- в выходном сечении – условия выхода потока. Их точную формулировку определим позднее.

На каждом шаге алгоритма SIMPLE требуется решить три дифференциальных уравнения (4.9), (4.10), (4.14) относительно неизвестных  $u^*, v^*, p'$ . Значит из представленных выше граничных условий требуется выразить граничные значения для этих трёх неизвестных сеточных векторов и рассказать способ их учёта при сборке соответствующих систем линейных уравнений.

#### 6.2.2.1 Входное сечение

При разложении скорости на пробное значение и поправку (4.8) условия для скорости (6.1) раскладываются следующим образом:

$$(x, y) \in \Gamma_{in} : u^* = 1, v^* = 0, u' = v' = 0 \quad (6.4)$$

Условия первого рода для пробной скорости учитываются при решении уравнений (4.9), (4.10). Для сеточного вектора  $u^*$ , узлы которого лежат непосредственно на границе, учёт этого условия сводится к модификации соответствующей строки матрицы  $A^u$  и правой части  $b^u$ . В строке  $k = k [0, j + \frac{1}{2}]$ :

$$A_{km}^u = \delta_{km}, \quad b_k^u = 1. \quad (6.5)$$

Для сеточного вектора  $v^*$  учёт производится с помощью выражения для значения в фиктивном узле  $k_1 = k [-\frac{1}{2}, j]$  через значение в настоящем узле  $k_0 = k [\frac{1}{2}, j]$ :

$$\frac{v_{k_0}^* + v_{k_1}^*}{2} = 0 \quad \Rightarrow \quad v_{k_1}^* = -v_{k_0}^*.$$

Поэтому при сборке матрицы  $A^v$  по формулам (4.23) при необходимости добавить значение  $a$  в колонку, соответствующую фиктивному узлу, требуется добавить это значение в диагональ с обратным знаком:

$$A_{k_0, k_1}^v += a \quad \Rightarrow \quad A_{k_0, k_0}^v -= a \quad (6.6)$$

Из условий на поправку скорости  $u' = v' = 0$  и уравнений (4.12), (4.13) следует граничное условие для поправки давления

$$x, y \in \Gamma_{in} : \frac{\partial p'}{\partial x} = 0 \quad (6.7)$$

Из этого условия получаем соотношение для давления в фиктивном узле  $k_1 = k [-\frac{1}{2}, j + \frac{1}{2}]$  через значение в реальном узле  $k_0 = k [\frac{1}{2}, j + \frac{1}{2}]$ :

$$p'_{k_1} = p'_{k_0}$$

Тогда добавление значения  $a$  в фиктивную колонку  $k_1$  эквивалентно

$$A_{k_0, k_1}^p += a \quad \Rightarrow \quad A_{k_0, k_0}^p += a \quad (6.8)$$

Следует понимать, что выражение (4.12) является приближением, используемым в расчётной схеме SIMPLE. В действительности, использование условия (6.7) (в случае нулевого начального приближения давления) приводит к нулевой производной для всего давления (а не только поправки)

$$x, y \in \Gamma_{in} : \frac{\partial p}{\partial x} = 0.$$

Это выражение никак не следует из постановки задачи. Действительно, если расписать уравнение (4.1) с учётом условий (6.1) и уравнения неразрывности (4.3), то получим соотношение

$$x, y \in \Gamma_{in} : \frac{\partial p}{\partial x} = \frac{1}{Re} \frac{\partial^2 u}{\partial x^2} = -\frac{1}{Re} \frac{\partial}{\partial x} \left( \frac{\partial v}{\partial y} \right). \quad (6.9)$$

(при выводы учтено, что  $\partial v / \partial y = -\partial u / \partial x = 0$ ). Однако, практика показывает, что в большинстве случаев, условий типа (6.7) оказывается достаточно. Выражение (6.9) равно нулю, если поперечная компонента скорости не появляется сразу за входным сечением. То есть течение остается прямолинейным на начальном участке расчётной области. Чтобы это исполнялось, входное сечение необходимо

размещать на таком расстоянии от препятствия, на котором поток еще не чувствует его присутствия (не начинает разворачиваться).

### 6.2.2.2 Условия симметрии

Однородные условия для скорости (6.2) расписываются как

$$(x, y) \in \Gamma_{in} : \frac{\partial u^*}{\partial y} = \frac{\partial u'}{\partial y} = 0, v^* = v' = 0.$$

Из условия на  $u^*$  запишем соотношение для фиктивного узла около нижней границы, которое будем использовать при сборке матрицы  $A^u$ :

$$\begin{aligned} k_0 &= k \left[ i + \frac{1}{2}, \frac{1}{2} \right], \quad k_1 = k \left[ i + \frac{1}{2}, -\frac{1}{2} \right], \\ u_{k_1}^* &= u_{k_0}^*, \\ A_{k_0, k_1}^u &+ a \Rightarrow A_{k_0, k_0}^u + a. \end{aligned}$$

Условие на  $v^*$  можно использовать явно:

$$\begin{aligned} k &= k \left[ i + \frac{1}{2}, \frac{1}{2} \right], \\ A_{k,s}^u &= \delta_{ks}, \quad b_k^u = 0. \end{aligned}$$

Границное условие для поправки давления можно получить из уравнения движения (4.2) (в неконсервативном виде) с учётом уравнения неразрывности. Используя

$$\begin{aligned} (x, y) \in \Gamma_{top,bot} : v &= 0 \Rightarrow \frac{\partial v}{\partial x} = 0 \Rightarrow \frac{\partial^2 v}{\partial x^2} = 0, \\ : \frac{\partial u}{\partial y} &= 0 \Rightarrow \frac{\partial}{\partial x} \frac{\partial u}{\partial y} = 0 \Rightarrow \frac{\partial^2 v}{\partial y^2} = 0, \end{aligned}$$

получим

$$(x, y) \in \Gamma_{top,bot} : \frac{\partial p}{\partial y} = 0.$$

При использовании нулевого начального приближения давления, для поправки давления так же справедливо

$$(x, y) \in \Gamma_{top,bot} : \frac{\partial p'}{\partial y} = 0.$$

Учёт этого условия на матричном уровне аналогичен процедуре (6.8).

### 6.2.2.3 Условия прилипания

Учёт условий прилипания на границе обтекаемого тела (6.3) в целом аналогичен алгоритму учёта входной границы. Для компонент скорости, узлы которых лежат на границе работает процедура (6.5) (с нулём в правой части). В случае если узлы не лежат на границе, то используется процедура (6.6).

Для поправки давления так же используется однородное условие второго рода (6.7) И все комментарии к этому условию, указанные в пункте 6.2.2.1, остаются справедливыми.

#### 6.2.2.4 Выходные граничные условия

На выходной границе отсутствует возможность указать какие-либо физические условия для искомых переменных. При этом, как правило, поведение течения в этой области большого интереса не представляет. Поэтому здесь требуется написать такие выражения, учёт которых не оказывал бы влияния на течение в основной области расчёта. Отсюда возникает проблема формулировки неотражающих граничных условий. Цель состоит в том, чтобы жидкость выходила из области расчёта естественным для себя образом, не подстраиваясь под выходную границу.

Простейшим решением этой проблемы является использование уравнения переноса на выходной границе:

$$(x, y) \in \Gamma_{out} : \begin{aligned} \frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} &= 0, \\ \frac{\partial v}{\partial t} + u \frac{\partial v}{\partial x} &= 0. \end{aligned} \quad (6.10)$$

В стационарном случае ( $\partial u, v / \partial t = 0$ ) из этих условий следует, что поперечная скорость равна нулю:

$$\frac{\partial u}{\partial x} = 0 \Rightarrow \frac{\partial v}{\partial y} = 0 \Rightarrow v = v|_{\Gamma_{bot}} = 0.$$

По аналогии с уравнениями движения (4.4) в уравнение на выходной границе так же добавим фиктивную производную по времени. Тогда условия для компонент скорости на итерации SIMPLE примут вид (для стационарного случая)

$$(x, y) \in \Gamma_{out} : \frac{\hat{u} - u}{\tau} + u \frac{\partial \hat{u}}{\partial x} = 0, \quad (6.11)$$

$$\hat{v} = 0.$$

Подставим разложение (4.8) и запишем условия для уравнений пробной скорости

$$(x, y) \in \Gamma_{out} : u^* + \tau u \frac{\partial u^*}{\partial x} = u, \quad (6.12)$$

$$v^* = 0. \quad (6.13)$$

Пробная скорость в алгоритме SIMPLE не удовлетворяет уравнению неразрывности. Поэтому нет гарантий, что найденная  $u^*$  сохраняет баланс массы в расчётной области. На практике это означает, что количество жидкости, которое втекает через  $\Gamma_{in}$  не равно количеству жидкости, которое вытекает через  $\Gamma_{out}$ .

Но финальная по итогам SIMPLE итерации скорость должна сохранять баланс массы. То есть

$$\Delta Q = \int_{\Gamma_{in}} \hat{u} ds - \int_{\Gamma_{out}} \hat{u} ds = 0.$$

Раскладывая это выражение через пробную скорость и поправку с учётом нулевого значения  $u'$  на входной границе (6.4), получим

$$\int_{\Gamma_{out}} u' ds = \int_{\Gamma_{in}} u^* ds - \int_{\Gamma_{out}} u^* ds$$

Положим, что  $u'$  на выходной границе постоянна. Это предположение не влияет на итоговый результат SIMPLE итераций, так как при его сходимости поправки скорости обнуляются. Тогда запишем значение поправки скорости на выходной границе

$$(x, y) \in \Gamma_{out} : u' = \left( \int_{\Gamma_{in}} u^* ds - \int_{\Gamma_{out}} u^* ds \right) / |\Gamma_{out}| \quad (6.14)$$

Подставляя это выражение в (4.12), получим граничные условия на поправку давления

$$(x, y) \in \Gamma_{out} : d^u \frac{\partial p'}{\partial x} = -\frac{u'}{\tau} \quad (6.15)$$

Таким образом, мы вывели граничные условия для всех трёх дифференциальных уравнений: (6.12), (6.13), (6.15).

Отметим, что если выходных границ несколько, то выражение (6.14) следует записывать для каждой из границ. При этом следует дополнительно задавать долю расхода  $C_i$ , вытекающую через каждую из границ. Пусть  $\Gamma_{out} = \Gamma_{o1} \cap \Gamma_{o2}$ . Тогда

$$\begin{aligned} (x, y) \in \Gamma_{o1} : u' &= \left( C_1 \int_{\Gamma_{in}} u^* ds - \int_{\Gamma_{o2}} u^* ds \right) / |\Gamma_{o1}| \\ (x, y) \in \Gamma_{o2} : u' &= \left( C_2 \int_{\Gamma_{in}} u^* ds - \int_{\Gamma_{o1}} u^* ds \right) / |\Gamma_{o2}| \\ C_1 + C_2 &= 1. \end{aligned}$$

**Учёт условия для  $u^*$  (6.12)** Просто перепишем уравнение в строках СЛАУ, соответствующих выходным узлам  $k_0 = k [n_x, j + \frac{1}{2}]$ . Для этого аппроксимируем конвективную производную по схеме против потока (с противопоточным узлом  $k_1 = k [n_x - 1, j + \frac{1}{2}]$ ).

$$u_{k_0}^* + \tau U_{k_0} \frac{u_{k_0}^* - u_{k_1}^*}{h_x} = u_{k_0},$$

где  $U$  - скорость переноса в  $k_0$ -ом узле. Она должна быть всегда больше нуля (иначе схема перестаёт быть противопотоковой). Можно просто положить её равной среднерасходной (единице в нашем случае). А можно взять из предыдущей итерации с проверкой на положительность:

$$U_{k_0} = \max(0, u_{k_0}).$$

На матричном уровне получим:

$$A_{k0,s}^u = \begin{cases} 1 + \frac{\tau U_{k_0}}{h_x}, & s = k_0 \\ -\frac{\tau U_{k_0}}{h_x}, & s = k_1 \\ 0, & \text{иначе,} \end{cases}, \quad (6.16)$$

$$b_{k_0}^u = u_{k_0}$$

**Учёт условия для  $v^*$  (6.13)** будем осуществлять за счёт введения фиктивного узла:

$$k_0 = k [n_x - \frac{1}{2}, j], \quad k_1 = k [n_x + \frac{1}{2}, j],$$

$$\frac{v_{k_0}^* + v_{k_1}^*}{2} = v_{\Gamma_{out}}^* = 0 \Rightarrow v_{k_1}^* = -v_{k_0}^*.$$

Отсюда добавление элемента  $a$  в фиктивную колонку будет осуществляться в виде

$$A_{k_0, k_1}^v += a \Rightarrow A_{k_0, k_0}^v -= a.$$

**Учёт условия для  $p'$  (6.15)** Также введём фиктивный узел  $k_1$  и расположенные левее от него реальный узел  $k_0$ :

$$k_0 = k [n_x - \frac{1}{2}, j + \frac{1}{2}], \quad k_1 = k [n_x + \frac{1}{2}, j + \frac{1}{2}],$$

Из (6.15)

$$d^u \frac{p'_{k_1} - p'_{k_0}}{h_x} = -\frac{u'}{\tau} \Rightarrow p'_{k_1} = p'_{k_0} - \frac{h_x}{\tau d^u} u'$$

На матричном уровне добавление фиктивной колонки даёт

$$A_{k_0, k_1}^v += a \Rightarrow A_{k_0, k_0}^v += a, \quad b_{k_0}^v += a \frac{h_x u'}{\tau d^u}. \quad (6.17)$$

## 6.2.3 Баланс сил. Коэффициенты сил

### 6.2.3.1 Сопротивление

Проинтегрируем уравнение движения (4.1) по области расчёта  $D$ :

$$\int_D \frac{\partial u^2}{\partial x} d\mathbf{x} + \int_D \frac{\partial uv}{\partial y} d\mathbf{x} = - \int_D \frac{\partial p}{\partial x} d\mathbf{x} + \frac{1}{Re} \int_D \nabla^2 u d\mathbf{x}.$$

Интегрирование по частям даёт:

$$\begin{aligned} \int_D \frac{\partial f}{\partial x} d\mathbf{x} &= \int_{\Gamma_{out}} f ds - \int_{\Gamma_{in}} f ds + \int_{\gamma} f n_x ds \\ \int_D \frac{\partial f}{\partial y} d\mathbf{x} &= \int_{\Gamma_{top}} f ds - \int_{\Gamma_{bot}} f ds + \int_{\gamma} f n_y ds, \\ \int_D \nabla^2 f d\mathbf{x} &= \int_{\Gamma} \frac{\partial f}{\partial n} ds + \int_{\gamma} \frac{\partial f}{\partial n} ds, \quad \Gamma = \Gamma_{in} \cap \Gamma_{out} \cap \Gamma_{bot} \cap \Gamma_{top} \end{aligned}$$

Учтём, что на обтекаемом теле скорости равны нулю, а верхняя и нижняя границы непротекаемы.

Тогда

$$\int_{\Gamma_{in}} (u^2 + p) ds - \int_{\Gamma_{out}} (u^2 + p) ds = \int_{\gamma} p n_x ds - \frac{1}{Re} \int_{\gamma} \frac{\partial u}{\partial n} ds$$

Полученное выражение есть баланс сил в  $x$  направлении. Слева стоит сила, обусловленная перепадом динамического давления (если считать профили скорости на входе и на выходе примерно одинаковыми, то останется только перепад статического давления). А справа - силы сопротивления потоку вследствии наличия препятствия. И эти силы уравновешивают друг друга. Первое слагаемое в правой части – есть сопротивление формы, второе – сопротивление трения из-за эффектов вязкости.

Коэффициенты этих сил имеют следующее выражение:

$$\begin{aligned} C_x^p &= 2 \int_{\gamma} p n_x ds \quad \text{– коэффициент сопротивления формы} \\ C_x^f &= -\frac{2}{Re} \int_{\gamma} \frac{\partial u}{\partial n} ds \quad \text{– коэффициент сопротивления трения} \\ C_x &= C_x^p + C_x^f \quad \text{– коэффициент сопротивления} \end{aligned} \tag{6.18}$$

Чтобы из этих безразмерных коэффициентов получить реальные силы, измеряемые в Ньютонах, нужно умножить их на  $\frac{1}{2}\rho U^2 L^2$ .

### 6.2.3.2 Подъёмная сила

Аналогично проинтегрируем уравнение движение в направлении  $y$  (4.2). С учётом граничных условий получим выражение для баланса сил в поперечном направлении

$$\int_{\Gamma_{bot}} p ds - \int_{\Gamma_{top}} p ds = \int_{\gamma} p n_y ds - \frac{1}{Re} \int_{\gamma} \frac{\partial v}{\partial n} ds$$

и соответствующие коэффициенты

$$\begin{aligned} C_y^p &= 2 \int_{\gamma} p n_y \, ds \\ C_y^f &= -\frac{2}{\text{Re}} \int_{\gamma} \frac{\partial v}{\partial n} \, ds \\ C_y &= C_y^p + C_y^f \quad \text{— коэффициент подъёмной силы} \end{aligned} \tag{6.19}$$

### 6.2.3.3 Вычисление коэффициентов сил на разнесённой сетке

Вычисления коэффициентов  $C_x, C_y$  по формулам (6.18), (6.19) сводятся к интегрированию давления и производных скорости по поверхности обтекаемого тела. Само вычисление интеграла происходит простым суммированием:

$$\int_{\gamma} f \, ds \approx \sum_i f_i |\gamma_i|, \tag{6.20}$$

где  $\gamma_i$  — отрезок границы  $\gamma$ , а  $f_i$  — значение функции в центре этого отрезка. В случае равномерной сетки  $|\gamma_i| = h_x, h_y$  для горизонтальных и вертикальных отрезков соответственно. Задача сводится к определению значению функций  $p, \partial u, v/\partial n$  в центрах отрезков.

**Горизонтальная граница** Зафиксируем узел  $i + \frac{1}{2}, j$  на такой границе. На этой границе  $n_x$  равна нулю, и вклад в  $C_x^p$  он не даёт. Для вычисления вклада в  $C_x^f$  нужно вычислить  $\partial u / \partial y$ . Для верхней границе запишем:

$$\begin{aligned} u_{i+\frac{1}{2},j} &= 0 \quad \text{из граничных условий прилипания,} \\ u_{i+\frac{1}{2},j+\frac{1}{2}} &= \frac{u_{i,j+\frac{1}{2}} + u_{i+1,j+\frac{1}{2}}}{2} \end{aligned}$$

отсюда

$$\frac{\partial u}{\partial n} = -\frac{\partial u}{\partial y} = \frac{u_{i+\frac{1}{2},j} - u_{i+\frac{1}{2},j+\frac{1}{2}}}{h_y/2} = -\frac{u_{i,j+\frac{1}{2}} + u_{i+1,j+\frac{1}{2}}}{h_y}$$

Аналогично для нижней границы

$$\frac{\partial u}{\partial n} = \frac{\partial u}{\partial y} = \frac{u_{i+\frac{1}{2},j} - u_{i+\frac{1}{2},j-\frac{1}{2}}}{h_y/2} = -\frac{u_{i,j-\frac{1}{2}} + u_{i+1,j-\frac{1}{2}}}{h_y}$$

Для вычисления вклада в  $C_y^p$  необходимо определить давление в  $i + \frac{1}{2}, j$ . На границе  $\gamma$  мы использовали условие  $\partial p / \partial n = 0$  (см п. 6.2.2.3). Отсюда на верхней границе:

$$\frac{\partial p}{\partial n} = \frac{p_{i+\frac{1}{2},j} - p_{i+\frac{1}{2},j+\frac{1}{2}}}{h_y} = 0 \quad \Rightarrow \quad p_{i+\frac{1}{2},j} = p_{i+\frac{1}{2},j+\frac{1}{2}}.$$

Тогда подинтегральное выражение равно

$$p n_y = -p_{i+\frac{1}{2},j+\frac{1}{2}}$$

Аналогично на нижней границе получим:

$$p n_y = p_{i+\frac{1}{2}, j-\frac{1}{2}}$$

Вклад горизонтальной границы в коэффициент  $C_y^f$  вычисляется через значение  $\partial v / \partial y$ , которое равно нулю из-за условий прилипания.

**Вертикальная граница** Теперь зафиксируем узел  $i, j + \frac{1}{2}$  на вертикальной границе. Вклад этой границы в коэффициенты  $C_y^p, C_x^f$  равны нулю: первого из-за значения  $n_y$ , а второго из-за  $\partial u / \partial x = -\partial v / \partial y = 0$ .

Для коэффициента  $C_x^p$  напишем:

$$\begin{aligned} p n_x &= p_{i-\frac{1}{2}, j+\frac{1}{2}} && \text{— левая граница,} \\ p n_x &= -p_{i+\frac{1}{2}, j+\frac{1}{2}} && \text{— правая граница.} \end{aligned} \quad (6.21)$$

Для  $C_y^f$ :

$$\begin{aligned} \frac{\partial v}{\partial n} &= -\frac{v_{i-\frac{1}{2}, j} + v_{i-\frac{1}{2}, j+1}}{h_x} && \text{— левая граница,} \\ \frac{\partial v}{\partial n} &= -\frac{v_{i+\frac{1}{2}, j} + v_{i+\frac{1}{2}, j+1}}{h_x} && \text{— правая граница.} \end{aligned} \quad (6.22)$$

### 6.3 Задание для самостоятельной работы

В SIMPLE-решателе для течения вязкой жидкости в каверне

[cavern2-simple] рассмотреть простые итерационные подходы к решению систем уравнений для  $u^*, v^*$ :

- метод Якоби (6.1.1),
- метод Зейделя (6.1.2),
- метод SOR (6.1.3).

Реализовать означенные решатели в виде функций вида:

```
// Single Jacobi iteration for mat*u = rhs SLAE. Writes result into u
void jacobi_step(const cfd::CsrMatrix& mat, const std::vector<double>& rhs,
→ std::vector<double>& u){
    ...
}
```

которые делают одну итерацию соответствующего метода без проверок на сходимость. Аргумент **u** используется как начальное значение искомого сеточного вектора. Туда же пишется итоговый результат.

Эти функции необходимо вызывать вместо

## AmgMatrixSolver::solve\_slae

в соответствующих решателях `Cavern2DSimpleWorker::compute_u_star`,  
`Cavern2DSimpleWorker::compute_v_star`.

Если требуется сделать несколько шагов, то вызывать несколько раз подряд.

Все алгоритмы основаны на вычислении выражения вида

$$\frac{1}{A_{ii}} \left( r_i - \sum_{j=0}^{N-1} A_{ij} u_j \right),$$

поэтому рекомендуется выделить отдельную функцию, которая бы вычисляла это выражение и использовалась всеми тремя решателями

```
double row_diff(size_t irow, const cfd::CsrMatrix& mat, const std::vector<double>&
→ rhs, const std::vector<double>& u){
    const std::vector<size_t>& addr = mat.addr();      // массив адресов
    const std::vector<size_t>& cols = mat.cols();      // массив колонок
    const std::vector<double>& vals = mat.vals();      // массив значений
    ...
}
```

Использовать параметры решателя:

$$\text{Re} = 100, \quad E = 4, \quad n_x = n_y = 50, \quad \varepsilon = 10^{-2}.$$

Сделать замеры времени исполнения:

- `total` – общее время работы итераций SIMPLE,
- `assemble` – время сборки систем уравнений для  $u^*$ ,  $v^*$ ,
- `p-solver` – время решения системы для  $p'$ ,
- `uv-solvers` – время решения систем для  $u^*$ ,  $v^*$ .

Замеры проводить в Release-версии сборки и с отключенными функциями сохранения в vtk.

Для замера времени исполнения участка кода воспользоваться функциями

- `cfд::dbg::Tic` – вызвать до начала участка кода
- `cfд::dbg::Toc` – вызвать после окончания участка кода

Так, чтобы замерить время `total`, нужно обрамить SIMPLE - цикл следующими вызовами

```
// iterations loop
dbg::Tic("total");    // запустить таймер total
size_t it = 0;
```

```

for (it=1; it < max_it; ++it){
    double nrm = worker.step();
    ...
}
dbg::Toc("total"); // остановить таймер total

```

Замеры времени p-solver и uv-solver делать в функции `Cavern2DSimpleWorker::step`:

```

dbg::Tic("uv-solvers");
std::vector<double> u_star = compute_u_star();
std::vector<double> v_star = compute_v_star();
dbg::Toc("uv-solvers");
dbg::Tic("p-solver");
std::vector<double> p_stroke = compute_p_stroke(u_star, v_star);
dbg::Toc("p-solver");

```

Замеры времени для сборки левых частей СЛАУ – в функции `Cavern2DSimpleWorker::set_uvp`:

```

dbg::Tic("assemble");
assemble_u_slae();
assemble_v_slae();
dbg::Toc("assemble");

```

При правильном задании функций замеров, по окончанию работы в консоль должен напечататься отчёт о времени исполнения вида:

```

total: 6.670 sec
uv-solvers: 5.220 sec
assemble: 1.210 sec
p-solver: 0.181 sec

```

Заполнить таблицу

	Кол-во итераций решателя СЛАУ	Кол-во итераций SIMPLE	total, s	assemble, s	p solver, s	uv solvers, s
Amg	—					
Якоби	1					
Якоби	2					
Якоби	4					
Зейдель	1					
Зейдель	2					
Зейдель	4					
SOR	1					

Здесь Amg - исходный решатель.

Сравнить полученное время исполнения со временем, которое занимает исходный метод. Подобрать оптимальный с точки зрения времени исполнения метод решения СЛАУ и его настройки (количество внутренних итераций).

# 7 Лекция 7 (20.10)

## 7.1 Инициализация решения

Схема решения SIMPLE является итерационной, а значит для начала расчётов ей требуется какое-то начальное приближение параметров, описывающих течение. Для решения задачи в каверне мы использовали нулевое приближение ( $u = v = p = 0$ ). Однако, при расчёте открытых течений, такое приближение не является оптимальным, потому что не соответствует входным граничным условиям. В таких задачах удобно в качестве начального приближения использовать потенциальное решение.

### 7.1.1 Задача о потенциале течения

Введем потенциал  $\phi$  векторного поля скорости как  $\mathbf{u} = \nabla\phi$ . В двумерной декартовой системе координат получим

$$\begin{aligned} u &= \frac{\partial\phi}{\partial x}, \\ v &= \frac{\partial\phi}{\partial y}. \end{aligned} \tag{7.1}$$

Для модели несжимаемой жидкости из уравнения неразрывности (4.3) получим уравнение для потенциала

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0. \tag{7.2}$$

В качестве граничных условий на всех непротекаемых поверхностях используем условие  $v_n = 0$ :

$$\left. \frac{\partial\phi}{\partial n} \right|_{wall} = 0 \tag{7.3}$$

Во входном и выходном сечениях используем условие постоянной нормальной скорости  $v_n$ , вычисляемой из известного расхода  $Q$ :

$$\left. \frac{\partial\phi}{\partial n} \right|_{io} = v_n = Q / |\Gamma_{io}|. \tag{7.4}$$

После решения задачи (7.2) – (7.4) компоненты скорости находятся прямым дифференцированием по формулам (7.1).

### 7.1.2 Аппроксимация на разнесённой сетке

Исходя из необходимости вычислять выражения (7.1), значение потенциала удобно аппроксимировать в “чёрных” узлах сетки (рис. 7).

Тогда сеточное уравнение для узла  $i + \frac{1}{2}, j + \frac{1}{2}$  для выражения (7.2) будет записано в виде

$$\frac{-\phi_{i-\frac{1}{2},j+\frac{1}{2}} + 2\phi_{i+\frac{1}{2},j+\frac{1}{2}} - \phi_{i+\frac{3}{2},j+\frac{1}{2}}}{h_x^2} + \frac{-\phi_{i+\frac{1}{2},j-\frac{1}{2}} + 2\phi_{i+\frac{1}{2},j+\frac{1}{2}} - \phi_{i+\frac{1}{2},j+\frac{3}{2}}}{h_y^2} = 0. \tag{7.5}$$

Граничные условия (7.3), (7.4) используются для вычисления значений в фиктивных узлах сетки.

Так, пусть левая граница  $i = 0$  есть входная граница течения. Тогда

$$\frac{\partial \phi}{\partial n} \Big|_{left} = -\frac{\partial \phi}{\partial x} \Big|_{left} = \frac{\phi_{-\frac{1}{2}, j+\frac{1}{2}} - \phi_{\frac{1}{2}, j+\frac{1}{2}}}{h_x} = v_n.$$

Отсюда получим

$$\phi_{-\frac{1}{2}, j+\frac{1}{2}} = h_x v_n + \phi_{\frac{1}{2}, j+\frac{1}{2}}.$$

Поэтому уравнение (7.5) для левых узлов сетки примет вид

$$\frac{\phi_{\frac{1}{2}, j+\frac{1}{2}} - \phi_{\frac{3}{2}, j+\frac{1}{2}}}{h_x^2} + \frac{-\phi_{\frac{1}{2}, j-\frac{1}{2}} + 2\phi_{\frac{1}{2}, j+\frac{1}{2}} - \phi_{\frac{1}{2}, j+\frac{3}{2}}}{h_y^2} = \frac{v_n}{h_x}.$$

Если нижняя граница сетки  $j = 0$  непротекаемая, то условие (7.3) даёт

$$\phi_{i+\frac{1}{2}, -\frac{1}{2}} = \phi_{i+\frac{1}{2}, \frac{1}{2}}$$

и уравнение (7.5) запишется как

$$\frac{-\phi_{i-\frac{1}{2}, j+\frac{1}{2}} + 2\phi_{i+\frac{1}{2}, j+\frac{1}{2}} - \phi_{i+\frac{3}{2}, j+\frac{1}{2}}}{h_x^2} + \frac{\phi_{i+\frac{1}{2}, \frac{1}{2}} - \phi_{i+\frac{1}{2}, \frac{3}{2}}}{h_y^2} = 0.$$

Поскольку задача в задаче для потенциала используются только граничные условия второго рода, то для получения однозначного решения необходимо явно указать значение в одном из узлов. Например

$$\phi_{\frac{1}{2}, \frac{1}{2}} = 0.$$

После вычисления сеточного вектора  $\{\phi\}$  значения компонент скорости получаются из формул (7.1):

$$u_{i,j+\frac{1}{2}} = \frac{\phi_{i+\frac{1}{2}, j+\frac{1}{2}} - \phi_{i-\frac{1}{2}, j+\frac{1}{2}}}{h_x}$$

$$u_{0,j+\frac{1}{2}} = v_n$$

## 7.2 Конвективный теплообмен

### 7.2.1 Уравнение теплопроводности

Дополним нестационарную систему уравнений течения (5.2) уравнением теплообмена

$$\rho c_p \left( \frac{\partial T}{\partial t} + u \frac{\partial T}{\partial x} + v \frac{\partial T}{\partial y} \right) = \lambda \left( \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right).$$

Здесь  $T$  – температура течения, К;  $\rho$  – плотность жидкости, кг/м<sup>3</sup>;  $c_p$  – теплоёмкость, Дж/кг/К;  $\lambda$  – теплопроводность, Вт/м/К.

В безразмерном виде это уравнение примет вид

$$\frac{\partial T}{\partial t} + u \frac{\partial T}{\partial x} + v \frac{\partial T}{\partial y} = \frac{1}{\text{Pe}} \left( \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right). \quad (7.6)$$

где безразмерная температура  $T$  вычислена через размерную  $T^{dim}$  как

$$T = \frac{T^{dim} - T^0}{\Delta T},$$

а число Пекле Pe есть

$$\text{Pe} = \frac{UL}{a}, \quad a = \frac{\lambda}{\rho c_p}.$$

### 7.2.2 Дискретизация по времени

Пользуясь обозначениями из п. 5.2.1 запишем неявную дискретизацию по времени уравнения (7.6) в виде

$$\frac{\hat{T} - \check{T}}{\Delta t} + \hat{u} \frac{\partial \hat{T}}{\partial x} + \hat{v} \frac{\partial \hat{T}}{\partial y} = \frac{1}{\text{Pe}} \left( \frac{\partial^2 \hat{T}}{\partial x^2} + \frac{\partial^2 \hat{T}}{\partial y^2} \right). \quad (7.7)$$

Полученное уравнение не содержит значений  $u, v$  с текущего итерационного слоя, и значит может быть решено один раз в конце шага по времени, когда сходимость уже достигнута.

### 7.2.3 Аппроксимация на разнесённой сетке

Пространственную аппроксимацию уравнения (7.7) будем проводить на разнесённой сетке в центральных (“чёрных”) узлах сетки (рис. 7). При этом конвективную производную будем приближать с помощью симметричной разности. Полученная конечная разность для узла  $i + \frac{1}{2}, j + \frac{1}{2}$  примет вид

$$\begin{aligned} & \frac{1}{\Delta t} \hat{T}_{i+\frac{1}{2},j+\frac{1}{2}} + \hat{u}_{i+\frac{1}{2},j+\frac{1}{2}} \frac{\hat{T}_{i+\frac{3}{2},j+\frac{1}{2}} - \hat{T}_{i-\frac{1}{2},j+\frac{1}{2}}}{2h_x} \\ & + \hat{v}_{i+\frac{1}{2},j+\frac{1}{2}} \frac{\hat{T}_{i+\frac{1}{2},j+\frac{3}{2}} - \hat{T}_{i+\frac{1}{2},j-\frac{1}{2}}}{2h_y} \\ & + \frac{1}{\text{Pe}} \frac{-\hat{T}_{i+\frac{3}{2},j+\frac{1}{2}} + 2\hat{T}_{i+\frac{1}{2},j+\frac{1}{2}} - \hat{T}_{i-\frac{1}{2},j+\frac{1}{2}}}{h_x^2} \\ & + \frac{1}{\text{Pe}} \frac{-\hat{T}_{i+\frac{1}{2},j+\frac{3}{2}} + 2\hat{T}_{i+\frac{1}{2},j+\frac{1}{2}} - \hat{T}_{i+\frac{1}{2},j-\frac{1}{2}}}{h_x^2} \\ & = \frac{1}{\Delta t} \check{T}_{i+\frac{1}{2},j+\frac{1}{2}}. \end{aligned} \quad (7.8)$$

Значения компонент скорости в центрах ячеек вычисляются с помощью ближайшей полусуммы

$$\hat{u}_{i+\frac{1}{2},j+\frac{1}{2}} \approx \frac{\hat{u}_{i,j+\frac{1}{2}} + \hat{u}_{i+1,j+\frac{1}{2}}}{2},$$

$$\hat{v}_{i+\frac{1}{2},j+\frac{1}{2}} \approx \frac{\hat{v}_{i+\frac{1}{2},j} + \hat{v}_{i+\frac{1}{2},j+1}}{2}.$$

#### 7.2.4 Границные условия

Учёт граничных условий производится за счёт вычисления значений в фиктивных узлах около границ.

Пусть требуется учесть условие на левой стенке ( $i = 0$ ). Тогда соответствующий фиктивный узел будет иметь индекс  $-\frac{1}{2}, j$ . Ниже приведём его выражения для трёх типов граничных условий.

##### 7.2.4.1 Условия первого рода

Пусть

$$T|_{left} = T^\Gamma \quad (7.9)$$

Тогда

$$\frac{T_{-\frac{1}{2},j+\frac{1}{2}} + T_{\frac{1}{2},j+\frac{1}{2}}}{2} = T^\Gamma.$$

Отсюда

$$T_{-\frac{1}{2},j+\frac{1}{2}} = -T_{\frac{1}{2},j+\frac{1}{2}} + 2T^\Gamma.$$

Таким образом, если в матрицу  $A^T$  левой части выражения (7.8) в фиктивную колонку  $k [-\frac{1}{2}, j + \frac{1}{2}]$  требуется добавить какое-то значение  $a$ , это равносильно добавлению этого выражения с обратным знаком в диагональ и удвоенного выражения, умноженного на граничное значение, в правую часть  $b^T$ :

$$k_0 = k \left[ \frac{1}{2}, j + \frac{1}{2} \right], \quad k_1 = k \left[ -\frac{1}{2}, j + \frac{1}{2} \right],$$

$$A_{k_0,k_1}^T = a \quad \Rightarrow \quad A_{k_0,k_0}^T = -a, \quad b_{k_0}^T = 2aT^\Gamma. \quad (7.10)$$

##### 7.2.4.2 Условия второго рода

Если на левой границе задано условие второго рода

$$\frac{\partial T}{\partial n} \Big|_{left} = - \frac{\partial T}{\partial x} \Big|_{left} = q \quad (7.11)$$

То вычисление фиктивного узла производится из конечной разности вида

$$\frac{T_{-\frac{1}{2},j+\frac{1}{2}} - T_{\frac{1}{2},j+\frac{1}{2}}}{h_x} = q.$$

Отсюда

$$T_{-\frac{1}{2},j+\frac{1}{2}} = T_{\frac{1}{2},j+\frac{1}{2}} + h_x q$$

Тогда

$$A_{k_0, k_1}^T + = a \Rightarrow A_{k_0, k_0}^T + = a, \quad b_{k_0}^T - = h_x q \quad (7.12)$$

#### 7.2.4.3 Условия третьего рода

Пусть на левой границе задано условие второго рода

$$\frac{\partial T}{\partial n} \Big|_{left} = - \frac{\partial T}{\partial x} \Big|_{left} = \alpha T + \beta \quad (7.13)$$

Расписывая производную и вычисляя значение температуры на стенке через полусумму, получим

$$\frac{T_{-\frac{1}{2}, j+\frac{1}{2}} - T_{\frac{1}{2}, j+\frac{1}{2}}}{h_x} = \alpha \frac{T_{-\frac{1}{2}, j+\frac{1}{2}} + T_{\frac{1}{2}, j+\frac{1}{2}}}{2} + \beta.$$

Отсюда выразим значение в фиктивном узле

$$T_{-\frac{1}{2}, j+\frac{1}{2}} = \frac{2 + \alpha h_x}{2 - \alpha h_x} T_{\frac{1}{2}, j+\frac{1}{2}} + \frac{2\beta h_x}{2 - \alpha h_x}$$

Тогда

$$A_{k_0, k_1}^T + = a \Rightarrow A_{k_0, k_0}^T + = \frac{2 + \alpha h_x}{2 - \alpha h_x} a, \quad b_{k_0}^T - = \frac{2\beta h_x}{2 - \alpha h_x} a. \quad (7.14)$$

#### 7.2.4.4 Универсальность условий третьего рода

Условие третьего рода (7.13) можно использовать для моделирования условий первого и второго рода. Так, условия второго рода (7.11) получаются, если положить  $\alpha = 0, \beta = q$ . А условия первого (7.9), – если  $\alpha = \varepsilon^{-1}, \beta = -\varepsilon^{-1}T^\Gamma$ , где  $\varepsilon$  – малое положительное число.

Если подставить эти выражения в формулу (7.14), то можно убедится, что они дадут выражения (7.12) и (7.10) (в пределе при  $\varepsilon \rightarrow 0$ ) соответственно.

#### 7.2.5 Коэффициент теплообмена

На границах, где заданы условия первого рода (7.9) можно вычислить тепловой поток, тем самым определив, сколько тепловой энергии требуется для поддержания этой постоянной температуры.

Безразмерный интегральный коэффициент теплообмена (интегральное число Нуссельта) определяется как

$$Nu = \int_{\gamma} \frac{\partial T}{\partial n} ds. \quad (7.15)$$

Для получения размерной мощности из этого безразмерного коэффициента (измеряемой в Ваттах), необходимо умножить его на  $\lambda \Delta TL$ .

Вычисление интегрального числа Нуссельта из определения (7.15) происходит по той же схеме, что и вычисление коэффициентов сил (6.20). При этом нормальная производная на границе  $\partial T / \partial n$

вычисляется в виде

$$(x, y) \in \gamma_i : \quad \frac{\partial T}{\partial n} \approx \frac{T^\Gamma - T_k}{h/2}, \quad (7.16)$$

где  $\gamma_i$  – отрезок границы,  $k$  – индекс ячейки, прилегающей к этому отрезку,  $h$  – шаг сетки, поперёк границы ( $h_x$  для вертикальных границ и  $h_y$  – для горизонтальных).

## 7.3 Тестовые примеры

### 7.3.1 Задача о равномерном течении

Рассмотрим задачу о стационарном прямолинейном течении с граничными условиями

$$\begin{aligned} (x, y) \in \Gamma_{in} : & \quad u = 1, v = 0, \\ (x, y) \in \Gamma_{top,bot} : & \quad \frac{\partial u}{\partial n} = 0, v = 0, \\ (x, y) \in \Gamma_{out} : & \quad u \frac{\partial u}{\partial x} = 0, v = 0. \end{aligned}$$

Очевидно, что точным решением этой задачи будут  $u = 1, v = 0, p = 0$ . В случае использования алгоритма инициализации (п. 7.1) мы бы сразу получили этот ответ. Но здесь в качестве теста будем начинать итерации из состояния покоя  $u = v = p = 0$ .

Задача решается в области  $[0, 2] \times [-\frac{1}{2}, \frac{1}{2}]$  с использованием алгоритма SIMPLEC с  $E = 4$  и разбиением на единицу длины  $n_{un} = 20$ .

Программа реализована в teste `linear2-simple` в файле `linear_2d_simple_test.cpp`.

Программа по расчёту этой задачи отличается от рассмотренной ранее задачи в каверне (п. 4.2) только наличием условий входного и выходного сечений.

**Шаг алгоритма SIMPLE** В функции `step()`, описывающей основной шаг алгоритма SIMPLE, добавлено вычисление граничных значений поправки скорости  $u'$  из (6.14) необходимых для соблюдения баланса массы (`compute_u_stroke_outflow`).

```
116 double Linear2DSimpleWorker::step(){
117     // Predictor step: U-star
118     std::vector<double> u_star = compute_u_star();
119     std::vector<double> v_star = compute_v_star();
120     std::vector<double> u_stroke_outflow = compute_u_stroke_outflow(u_star);
121     // Pressure correction
122     std::vector<double> p_stroke = compute_p_stroke(u_star, v_star, u_stroke_outflow);
123     // Velocity correction
124     std::vector<double> u_stroke = compute_u_stroke(p_stroke, u_stroke_outflow);
125     std::vector<double> v_stroke = compute_v_stroke(p_stroke);
126     // Set final values
127     std::vector<double> u_new = vector_sum(u_star, 1.0, u_stroke);
128     std::vector<double> v_new = vector_sum(v_star, 1.0, v_stroke);
129     std::vector<double> p_new = vector_sum(_p, _alpha_p, p_stroke);
```

```

130
131     return set_uvp(u_new, v_new, p_new);
132 }

```

В дальнейшем эти условия используются для расчёта поправки давления и для расчёта самой поправки скорости.

### 7.3.1.1 Учёт граничных условий

**Вычисление поправки скорости на выходной границе** Функция

`compute_u_stroke_outflow`, реализующая вычисление формулы (6.14), имеет вид

```

378 std::vector<double> Linear2DSimpleWorker::compute_u_stroke_outflow(const
379   ↵ std::vector<double>& u_star) const{
380   double qin = 0;
381   double qout = 0;
382   for (size_t j=0; j<_grid.ny(); ++j){
383     size_t ind_left = _grid.yface_grid_index_i_jp(0, j);
384     size_t ind_right = _grid.yface_grid_index_i_jp(_grid.nx(), j);
385     qin += u_star[ind_left]*_hy;
386     qout += u_star[ind_right]*_hy;
387   }
388   double Lout = _grid.Ly();
389   double diff_u = (qin - qout)/Lout;
390   std::vector<double> ret(_grid.ny(), diff_u);
391   return ret;
392 }

```

. Здесь в цикле по вертикальным граням вычисляются расходы по входному и выходному сечениям (`qin`, `qout`), далее находится поправка скорости (`diff_u`), постоянная для всех выходных отрезков, и возвращается вектор, содержащий эту поправку для всех выходных отрезков.

**Учёт граничных условий для  $u^*$**  Для учёта граничных условий входа и выхода при сборке уравнения для  $u^*$  в функции `assemble_u_slae` используется цикл

```

222   for (size_t j=0; j< _grid.ny(); ++j){
223     // left boundary:  $u = 1$ 
224     {
225       size_t index_left = _grid.yface_grid_index_i_jp(0, j);
226       mat.set_value(index_left, index_left, 1.0);
227       _rhs_u[index_left] = 1.0;
228     }
229     // right boundary:  $u \cdot du/dx = 0$ 

```

```

230    {
231        size_t index_right = _grid.yface_grid_index_i_jp(_grid.nx(), j);
232        size_t index_right_minus = _grid.yface_grid_index_i_jp(_grid.nx()-1, j);
233        double u0 = std::max(0.0, _u[index_right]);
234        double coef = _tau*u0/_hx;
235        mat.set_value(index_right, index_right, 1.0 + coef);
236        mat.set_value(index_right, index_right_minus, -coef);
237        _rhs_u[index_right] = u0;
238    }
239 }

```

Здесь для левой границы согласно (6.5) жёстко устанавливается единичное значение. А для правой границы используются соотношения (6.16).

**Учёт граничных условий для  $p'$**  Найденные поправки скорости на выходной границе должны быть учтены при решении задачи для  $p'$  согласно (6.17) Сборка матрицы левой части при этом останется неизменной. Действительно, распишем производную на правой (выходной) границе

$$d^u \frac{\partial p'}{\partial x}_{n_x, j+\frac{1}{2}} \approx d^u \frac{p'_{k_1} - p'_{k_0}}{h_x}$$

входящую в выражение (4.24). Где  $k_0 = k[n_x - \frac{1}{2}, j + \frac{1}{2}]$  – реальный, а  $k_1 = k[n_x + \frac{1}{2}, j + \frac{1}{2}]$  – находящийся правее него фиктивный узлы сетки. Наличие этой производной требует добавления выражения  $d^u/h_x^2$  в реальный (диагональный) столбец  $k_0$  и выражения  $-d^u/h_x^2$  в фиктивный столбец  $k_1$  матрицы  $A^p$  в строке  $k_0$ . Следуя алгоритму (6.17) добавление значения в фиктивный столбец равносильно добавлению этого же значения в диагональный столбец. То есть два этих значения взаимоуничтожаются. Останется только модифицировать столбец правых членов. Альтернативно можно просто подставить значение производной (6.15) в дисcretизованное выражение (4.24), и, поскольку оно не содержит в себе  $p'$ , унести его в правую часть с обратным знаком и делением на  $h_x$ .

Учёт граничных условий в правой части осуществляется в функции `assemble_p_stroke_solver` за счёт модификации правой части для узлов, расположенных около выходной границы:

```

367 // outflow compensation
368 if (i == _grid.nx()-1{
369     rhs[ind0] -= (u_stroke_outflow[j]) / _tau / _hx;
370 }

```

**Учёт граничных условий для  $u'$**  Явным образом предварительно найденные граничные значения для поправки скорости присваиваются в функции

"compute\_u\_stroke":

```

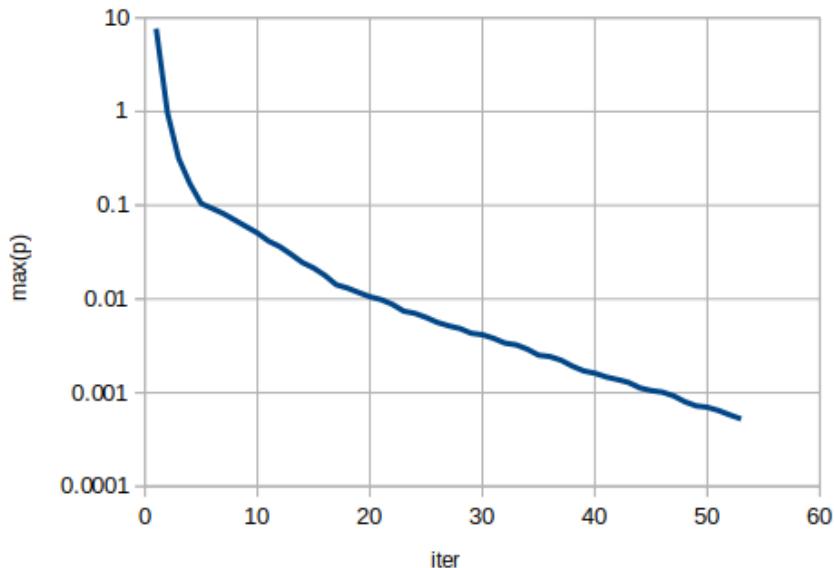
402 // outflow
403 for (size_t j=0; j<_grid.ny(); ++j){
404     size_t ind0 = _grid.yface_grid_index_i_jp(_grid.nx(), j);
405     u_stroke[ind0] = u_stroke_outflow[j];
406 }

```

### 7.3.1.2 Анализ результатов

До невязки  $\varepsilon = 10^{-2}$  задача сходится за 53 итерации. При этом для скорости точный ответ получается уже на первой итерации, а всё остальное время происходит подстройка давления.

Максимальное значение давление в зависимости от итерации приведено на графике ниже



### 7.3.2 Течение Пуазейля

TODO

### 7.3.3 Стационарное обтекание квадратного препятствия

В тесте `obstacle2-simple` из файла

`obstacle_2d_simple_test.cpp` рассматривается задача о стационарном обтекании квадратного препятствия (см. постановку в п. 6.2). По окончании расчёта в консоль печатаются коэффициенты сопротивления и подъёмной силы.

Используется естественная нумерация узлов с неактивными ячейками. Класс

`RegularGrid2D` предлагает следующие методы, связанные с неактивными ячейками:

- 

`void RegularGrid2D::deactivate_cells(Point bot_left, Point top_right)` – установить область неактивных ячеек;

- `bool RegularGrid2D::is_active_cell(size_t icell)` – проверить, является ли ячейка активной.

Кроме того, в задаче появились внутренние границы. То есть для постановки граничных условий уже не достаточно использовать крайние значение индексов  $i, j$ , а нужен механизм для получения граничных отрезков сетки. Для этого введены следующие функции

- `RegularGrid2d::boundary_yfaces()` – получить список всех вертикальных граничных фасок (возвращает парные индексы в соответствии `yface_centered_grid`).
- `RegularGrid2d::boundary_xfaces()` – получить список всех горизонтальных граничных фасок (возвращает парные индексы в соответствии `xface_centered_grid`).
- `RegularGrid2d::yface_type(size_t yface_index)` – узнать тип вертикальной грани по её глобальному индексу. Возвращает перечисление

```
enum struct FaceType{
    Internal,      // внутренняя
    Boundary,      // граничная
    Deactivated    // неактивная (находится внутри неактивной области)
};
```

- `RegularGrid2d::xface_type(size_t xface_index)` – узнать тип горизонтальной грани по её глобальному индексу.

### 7.3.3.1 Функция верхнего уровня

Здесь сначала происходит установка параметров расчёта: числа Рейнольдса, параметра  $E$ , количества итераций, порога сходимости и разбиения единичного интервала.

```
729 double Re = 20;
730 double E = 4.0;
731 size_t max_it = 10000;
732 double eps = 1e-1;
733 size_t n_unit = 10; // partition per unit length
```

Далее строится сетка

```
736 RegularGrid2D grid(0, 12, -2, 2, 12*n_unit, 4*n_unit);
```

В этом примере сетка строится в четырёхугольнике  $[0, 12] \times [-2, 2]$ . Потом для описания квадратного препятствия происходит деактивация ячеек, находящихся в единичном квадрате с нижней левой координатой  $(2, -0.5)$  и верхней правой координатой  $(3, 0.5)$ .

```
737     grid.deactivate_cells({2, -0.5}, {3, 0.5});
```

Потом создаётся решатель, инициализируются функции сохранения и вызывается алгоритм потенциальной инициализации расчётных полей.

```
738     Obstacle2DSimpleWorker worker(Re, grid, E);
739     worker.initialize_saver(false, "obstacle2");
740
741     // initial condition
742     worker.initialize();
```

Затем идёт стандартный цикл по SIMPLE-итерациям

```
745     size_t it = 0;
746     for (it=1; it < max_it; ++it){
747         double nrm = worker.step();
748
749         // print norm
750         std::cout << it << " " << nrm << std::endl;
751
752         // break if residual is low enough
753         if (nrm < eps){
754             break;
755         }
756     }
```

По окончании цикла вызывается функция сохранения решения в vtk

```
758     worker.save_current_fields(it);
```

В конце происходит расчёт коэффициентов сил и их печать в консоль

```
760     Obstacle2DSimpleWorker::Coefficients coefs = worker.coefficients();
761     std::cout << "==== Drag" << std::endl;
762     std::cout << "Cpx = " << coefs.cpx << std::endl;
763     std::cout << "Cfx = " << coefs.cfx << std::endl;
764     std::cout << "Cx = " << coefs.cx << std::endl;
765     std::cout << "==== Lift" << std::endl;
766     std::cout << "Cpy = " << coefs.cpy << std::endl;
```

```

767 std::cout << "Cfy = " << coefs.cfy << std::endl;
768 std::cout << "Cy  = " << coefs.cy  << std::endl;

```

Результирующее поле течения сохраняется в файл `obstacle2.vtk.series`.

### 7.3.3.2 Учёт неактивных ячеек

Неактивные ячейки учитываются во всех алгоритмах сборки систем линейных уравнений. Рассмотрим на примере сборки матрицы для пробной скорости, реализованной в функции `assemble_u_slae`.

```

326 void Obstacle2DSimpleWorker::assemble_u_slae(){

```

Рассмотрим цикл сборки внутренних узлов “красной” сетки для  $u$  (или, что тоже самое, цикл по всем вертикальным граням основной сетки)

```

371 for (size_t j=0; j < _grid.ny(); ++j)
372 for (size_t i=1; i < _grid.nx(); ++i){

```

Сначала вычисляется индекс строки (сквозной индекс текущей грани):

```

373     size_t row_index = _grid.yface_grid_index_i_jp(i, j); // [i, j+1/2]

```

Эта грань может быть либо внутренней, либо граничной (принадлежать внутренней вертикальной границе), либо неактивной. Выполняется проверка, является ли эта грань внутренней

```

374 if (_grid.yface_type(row_index) == RegularGrid2D::FaceType::Internal){

```

Если да, то выполняется обычная процедура сборки

```

375     double u0_plus    = u_ip_jp(i, j); // _u[i+1/2, j+1/2]
376     double u0_minus   = u_ip_jp(i-1, j); // _u[i-1/2, j+1/2]
377     double v0_plus    = v_i_j(i, j+1); // _v[i, j+1]
378     double v0_minus   = v_i_j(i, j); // _v[i, j]

379
380     // u_(i, j+1/2)
381     add_to_mat(row_index, {i, j}, 1.0);
382     // + tau * d(u0*u)/dx
383     add_to_mat(row_index, {i+1, j}, _tau/2.0/_hx*u0_plus);
384     add_to_mat(row_index, {i-1, j}, -_tau/2.0/_hx*u0_minus);
385     // + tau * d(v0*u)/dy
386     add_to_mat(row_index, {i, j+1}, _tau/2.0/_hy*v0_plus);
387     add_to_mat(row_index, {i, j-1}, -_tau/2.0/_hy*v0_minus);
388     // - tau / Re * d^2u/dx^2

```

```

389 add_to_mat(row_index, {i, j}, 2.0*_tau/_Re/_hx/_hx);
390 add_to_mat(row_index, {i+1, j}, -_tau/_Re/_hx/_hx);
391 add_to_mat(row_index, {i-1, j}, -_tau/_Re/_hy/_hy);
392 //      - tau / Re * d^2u/dy^2
393 add_to_mat(row_index, {i, j}, 2.0*_tau/_Re/_hy/_hy);
394 add_to_mat(row_index, {i, j+1}, -_tau/_Re/_hy/_hy);
395 add_to_mat(row_index, {i, j-1}, -_tau/_Re/_hy/_hy);
396 // = u0_(i, j+1/2)
397 _rhs_u[row_index] += _u[row_index];
398 //      - tau * dp/dx
399 _rhs_u[row_index] -= _tau/_hx*(p_ip_jp(i, j) - p_ip_jp(i-1, j));

```

Если нет (то есть грань либо неактивная, либо принадлежит внутренней границе), то в диагональ ставится единица, в правую часть 0.

```

400 } else {
401     mat.set_value(row_index, row_index, 1.0);
402     _rhs_u[row_index] = 0;
403 }

```

Это отражает тот факт, что на внутренних границах  $u = 0$  из-за условий прилипания, а для неактивных мы пишем тривиальное уравнение, просто чтобы матрица не была вырождена.

Учёт условий прилипания на внутренних горизонтальных границах осуществляется через фиктивный узел в лямбда-функции "add\_to\_mat", которая перехватывает все ситуации, когда алгоритм требует добавить что-либо в фиктивную колонку матрицы.

```
331 auto add_to_mat = [&](size_t row_index, std::array<size_t, 2> ij_col, double value){
```

Такие ситуации могут произойти либо при сборке около вертикальной грани, находящейся рядом с верхней границей:

```

332 if (ij_col[1] == _grid.ny()){
333     // ghost index => top boundary condition: du/dn = 0
334     size_t ind1 = _grid.yface_grid_index_i_jp(ij_col[0], ij_col[1]-1);
335     mat.add_value(row_index, ind1, value);

```

либо около вертикальной грани, находящейся рядом с нижней границей границией:

```

336 } else if (ij_col[1] == (size_t)-1){
337     // ghost index => bottom boundary condition: du/dn = 0
338     size_t ind1 = _grid.yface_grid_index_i_jp(ij_col[0], ij_col[1]+1);
339     mat.add_value(row_index, ind1, value);

```

либо около вертикальной грани, находящейся непосредственно над или под препятствием. В этом случае индекс фиктивной колонки, в которую требуется поставить будет соответствовать неактивной вертикальной грани. Мы вычисляем этот индекс

```
340 } else {  
341     size_t ind1 = _grid.yface_grid_index_i_jp(ij_col[0], ij_col[1]);
```

Если он неактивный, то следуем по процедуре добавления фиктивного узла около границы с нулевым значением.

```
342     if (_grid.yface_type(ind1) == RegularGrid2D::FaceType::Deactivated){  
343         // ghost index => obstacle boundary u = 0  
344         mat.add_value(row_index, row_index, -value);  
345     } else {
```

Иначе – это нормальная колонка и мы добавляем туда значение по стандартной процедуре

```
346     mat.add_value(row_index, ind1, value);
```

### 7.3.3.3 Расчёт коэффициентов сопротивления

Расчёт коэффициентов сил по формулам (6.18), (6.19) осуществляется в процедуре `coefficients()`. Она возвращает структуру, куда входят все шесть искомых значений

```
33 struct Coefficients{  
34     double cpx;  
35     double cpy;  
36     double cfx;  
37     double cfy;  
38     double cx;  
39     double cy;  
40 };
```

Процедура, объявленная как

```
661 Obstacle2DSimpleWorker::Coefficients Obstacle2DSimpleWorker::coefficients() const{
```

производит вычисления четырёх интегралов по простой квадратуре (6.20). Результаты агрегируются в переменные

```
662     double sum_cpx = 0;  
663     double sum_cpy = 0;
```

```
664     double sum_cfx = 0;  
665     double sum_cfy = 0;
```

Операции проводятся в циклах по внутренним граничным отрезкам. Сначала рассматриваются вертикальные границы:

```
668     for (const RegularGrid2D::split_index_t& yface: _grid.boundary_yfaces()) {
```

Здесь в переменную

`yface` попадают все парные индексы вертикальных граней, лежащих на границах. Сначала нужно отфильтровать границы, лежащие во входном и выходном сечениях

```
669     if (yface[0] == 0) {  
670         // input => ignore  
671     } else if (yface[0] == _grid.nx()) {
```

На вертикальных границах актуально вычисление коэффициентов  $C_x^p$ ,  $C_y^f$  (из пункта 6.2.3.3). Для их определения на каждой сеточной грани мы должны определить  $p_{nx}$ ,  $\partial v / \partial n$  по формулам (6.21), (6.22).

```
674     double pnx, dvdn;
```

Для использования этих формул нужно определить, является ли это левой или правой границей обтекаемого тела. Мы вычисляем индексы ячеек, лежащих слева и справа. Если левая ячейка активна, значит это левая граница, если правая активна, значит это правая граница.

```
675     size_t left_cell = _grid.cell_centered_grid_index_ip_jp(yface[0]-1, yface[1]);  
676     size_t right_cell = _grid.cell_centered_grid_index_ip_jp(yface[0], yface[1]);
```

Далее левой границы

```
677     if (_grid.is_active_cell(left_cell)) {  
678         pnx = _p[left_cell];  
679         dvdn = -v_ip_jp(yface[0]-1, yface[1]) / (_hx/2.0);
```

для правой

```
680     } else if (_grid.is_active_cell(right_cell)) {  
681         pnx = -_p[right_cell];  
682         dvdn = -v_ip_jp(yface[0], yface[1]) / (_hx/2.0);
```

иначе (если это и не правая и не левая граница) бросается исключение, потому что так быть не должно: у любой внутренней границы должна быть хоть одна соседняя активная ячейка

```

683 } else {
684     _THROW_UNREACHABLE_;
685 }
```

После вычисления  $p n_x$ ,  $\partial v / \partial n$  они добавляются в искомые интегралы согласно (6.20):

```

686     sum_cpx += pnx * _hy;
687     sum_cfy += dvdn * _hy;
```

Далее аналогичная процедура проводится для горизонтальных граней, в результате которой вычисляются интегралы `sum_cpy`,  
`sum_cfx`.

```

692 for (const RegularGrid2D::split_index_t& xface: _grid.boundary_xfaces()){
693     if (xface[1] == 0){
694         // bottom => ignore
695     } else if (xface[1] == _grid.ny()){
696         // top => ignore
697     } else {
698         double pny, dudn;
699         size_t bot_cell = _grid.cell_centered_grid_index_ip_jp(xface[0], xface[1]-1);
700         size_t top_cell = _grid.cell_centered_grid_index_ip_jp(xface[0], xface[1]);
701         if (_grid.is_active_cell(bot_cell)){
702             pny = -_p[bot_cell];
703             dudn = -u_ip_jp(xface[0], xface[1]-1)/(_hy/2.0);
704         } else if (_grid.is_active_cell(top_cell)){
705             pny = -_p[top_cell];
706             dudn = -u_ip_jp(xface[0], xface[1])/(_hy/2.0);
707         } else {
708             _THROW_UNREACHABLE_;
709         }
710         sum_cpy += pny * _hx;
711         sum_cfx += dudn * _hx;
712     }
713 }
```

В конце функции искомые коэффициенты вычисляются через уже найденные интегралы согласно (6.18), (6.19):

```

715 Coefficients coefs;
716 coefs.cpx = 2.0*sum_cpx;
717 coefs.cpy = 2.0*sum_cpy;
718 coefs.cfx = -2.0/_Re*sum_cfx;
719 coefs.cfy = -2.0/_Re*sum_cfy;
720 coefs.cx = coefs.cpx + coefs.cfx;
721 coefs.cy = coefs.cpy + coefs.cfy;
722 return coefs;

```

### 7.3.3.4 Результаты расчёта

Картина течения, полученная для сетки

`n_part = 10` при  $Re = 20$ , представлена на рис. 12. Полученные коэффициенты сопротивления:

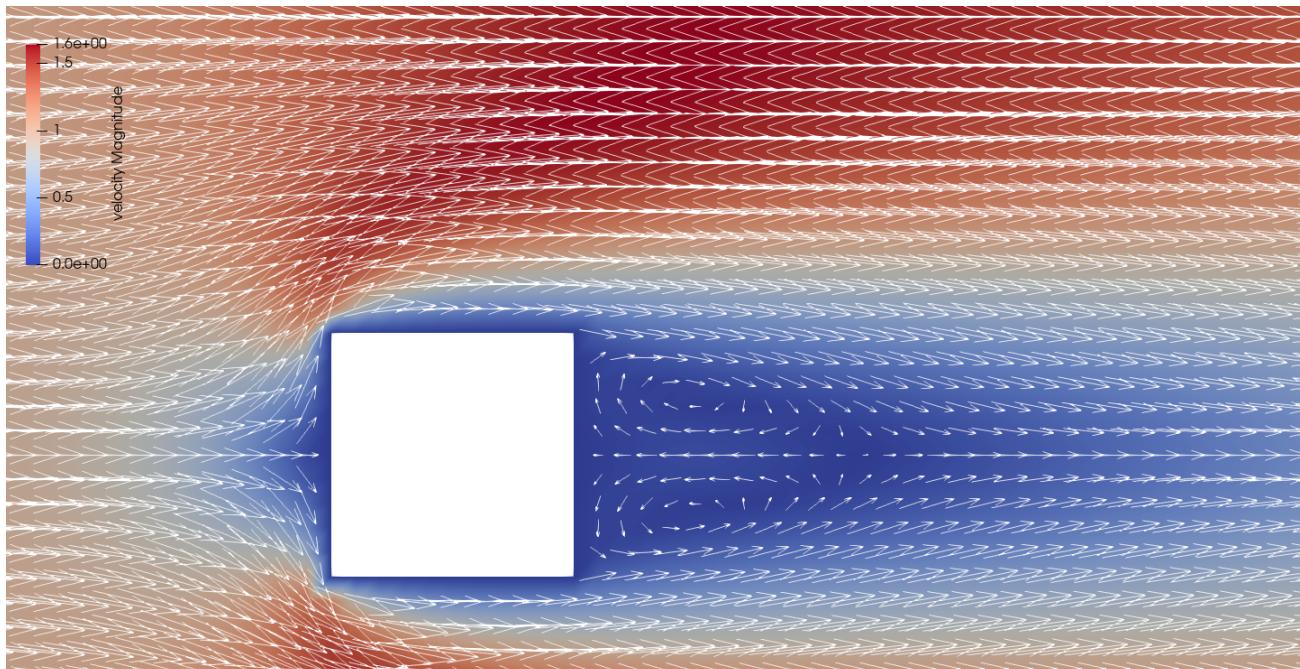


Рис. 12: Обтекание квадратного препятствия в стационарном режиме

```

==== Drag
Cpx = 2.97224
Cfx = 1.08639
Cx = 4.05863
==== Lift
Cpy = -6.815e-10
Cfy = -1.64419e-10
Cy = -8.45919e-10

```

Для  $\varepsilon = 10^{-1}$  решение сошлось за 29 итераций.

### 7.3.4 Нестационарное обтекание квадратного препятствия с теплообменом

Эта задача реализована в файле `obstacle_nonstat_2d_simple_test.cpp` в тесте `[obstacle2-nonstat-simple]`.

Программа решает задачу в той же области, которая рассматривалась в предыдущем пункте, но в нестационарной постановке (5.2) и с добавлением температуры (7.6). Граничные условия для температуры имеют вид

$$\begin{aligned}(x, y) \in \Gamma_{in} : \quad & T = 0, \\(x, y) \in \gamma : \quad & T = 1, \\(x, y) \in \Gamma_{out,top,bot} : \quad & \frac{\partial T}{\partial n} = 0.\end{aligned}$$

Поля течения для разных моментов времени пишутся в файл `obstacle-nonstat.vtk.series`. Кроме того, в файл `c.txt` пишутся вычисленные на разные моменты времени коэффициенты сопротивления и интегральное число Нуссельта.

#### 7.3.4.1 Функция верхнего уровня

В начале обозначим параметры задачи: числа Рейнольдса и Пекле, разбиение единичного отрезка, шаг по времени  $\Delta t$  и конечное время, параметр внутреннего итерационного процесса  $E$ , максимальное количество итераций во внутреннем итерационном процессе и порог по невязке:

```
866 double Re = 100;
867 double Pe = 100;
868 size_t n_unit = 10; // partition per unit length
869 double time_step = 0.25;
870 double end_time = 5;
871 double E = 4.0;
872 size_t max_it = 10000;
873 double eps = 1e-0;
```

Далее проводится создание сетки (так же, как и в предыдущем примере) и начальная инициализация решателя

```
876 RegularGrid2D grid(0, 12, -2, 2, 12*n_unit, 4*n_unit);
877 grid.deactivate_cells({2, -0.5}, {3, 0.5});
878 ObstacleNonstat2DSimpleWorker worker(Re, Pe, grid, E, time_step);
879 worker.initialize_saver(false, "obstacle2-nonstat");
880
881 // initial condition
882 worker.initialize();
883 worker.save_current_fields(0);
```

После всех инициализаций начинается цикл по времени

```
886 for (double time=time_step; time<end_time+1e-6; time+=time_step){
```

Отметим, что поскольку значению  $t = 0$  соответствует начальное состояние решения, то цикл начинается сразу с первого шага  $t = \Delta t$ .

Внутри цикла по времени производится цикл внутренних итераций SIMPLE

```
887 size_t it = 0;
888 for (it=1; it < max_it; ++it){
889     double nrm = worker.step();
890
891     // break inner iterations if residual is low enough
892     if (nrm < eps){
893         break;
894     } else if (it == max_it -1) {
895         std::cout << "WARNING: internal SIMPLE interations not converged with nrm = "
896             << nrm << std::endl;
897     }
898 }
```

Далее, если текущее время кратно 1.0, производится сохранение решения в файл vtk и запись коэффициентов сил в файл:

```
900 if (std::abs(time - round(time)) < 1e-6){
901     worker.save_current_fields(time);
902 }
```

Печатается информация о сходимости текущей итерации

```
903 std::cout << convergence_report(time, it);
```

и производится переход на следующий шаг по времени:

```
906 worker.to_next_time_step();
```

#### 7.3.4.2 Учёт нестационарности

Согласно пункту 5.2 наличие производной по времени учитывается:

- При вычислении коэффициентов  $d^u$ ,  $d^v$  (5.5):

```
116 _du = 1.0 / (1 + _tau/_time_step + 2.0*_tau/_Re * (1.0/_hx/_hx + 1.0/_hy/_hy));
117 _dv = 1.0 / (1 + _tau/_time_step + 2.0*_tau/_Re * (1.0/_hx/_hx + 1.0/_hy/_hy));
```

- При сборке систем уравнений для  $u^*$ ,  $v^*$  (5.4) как прибавка к диагонали

```
423 add_to_mat(row_index, {i, j}, 1.0 + _tau/_time_step);
```

и правой части

```
441 _rhs_u[row_index] += (_tau/_time_step)*_u_old[row_index];
```

- А так же в граничных условиях на выходе. В этом случае условия (6.10) для  $u$  по аналогии с (6.11) аппроксимируются к виду

$$(x, y) \in \Gamma_{out} : \frac{\hat{u} - \check{u}}{\Delta t} + \frac{\hat{u} - u}{\tau} + u \frac{\partial \hat{u}}{\partial x} = 0.$$

Для упрощения по прежнему будем использовать “стационарное” условие для поперечной скорости  $v = 0$ . Тогда для  $u^*$  можно записать

$$(x, y) \in \Gamma_{out} : \left(1 + \frac{\tau}{\Delta t}\right) u^*_{n_x,j+\frac{1}{2}} + \tau U_{n_x,j+\frac{1}{2}} \frac{u^*_{n_x,j+\frac{1}{2}} - u^*_{n_x-1,j+\frac{1}{2}}}{h_x} = u_{n_x,j+\frac{1}{2}} + \frac{\tau}{\Delta t} \check{u}_{n_x,j+\frac{1}{2}}$$

Это выражение и добавляется в соответствующие строки матрицы и правой части

```
400 // right boundary: du/dt + u*du/dx = 0
401 {
402     size_t index_right = _grid.yface_grid_index_i_jp(_grid.nx(), j);
403     size_t index_right_prev = _grid.yface_grid_index_i_jp(_grid.nx()-1, j);
404     double u0 = std::max(0.0, _u[index_right]);
405     double coef = _tau*u0/_hx;
406     mat.set_value(index_right, index_right, 1.0 + _tau/_time_step + coef);
407     mat.set_value(index_right, index_right_prev, -coef);
408     _rhs_u[index_right] = u0 + _tau/_time_step * _u_old[index_right];
409 }
```

- При переходе на следующий шаг по времени в функции происходит вычисление текущего значения температуры, и присваивание значений  $\check{u}$ ,  $\check{v}$ .

```
247 double ObstacleNonstat2DSimpleWorker::to_next_time_step(){
248     _t = compute_temperature();
249     _u_old = _u;
```

```

250     _v_old = _v;
251     return set_uvp(_u, _v, _p);
252 }

```

Вызов `set_uvp` здесь осуществляется для пересборки актуальных матриц.

### 7.3.4.3 Расчёт температурного поля

Осуществляется в функции

```

624 std::vector<double> ObstacleNonstat2DSimpleWorker::compute_temperature() const{

```

Для сборки системы используется цикл по ячейкам сетки

```

651 for (size_t j=0; j < _grid.ny(); ++j)
652 for (size_t i=0; i < _grid.nx(); ++i){
653     size_t row_index = _grid.cell_centered_grid_index_ip_jp(i, j);
654     if (_grid.is_active_cell(row_index)){
655         double u_left = _u[_grid.yface_grid_index_i_jp(i, j)];
656         double u_right = _u[_grid.yface_grid_index_i_jp(i+1, j)];
657         double v_bot = _v[_grid.xface_grid_index_ip_j(i, j)];
658         double v_top = _v[_grid.xface_grid_index_ip_j(i, j+1)];
659
660         // 1.0/time_step T(i+1/2, j+1/2)
661         add_to_mat(row_index, {i, j}, 1.0 / _time_step);
662         //      + d(u0*T)/ dx
663         add_to_mat(row_index, {i+1,j}, u_right/2.0/_hx);
664         add_to_mat(row_index, {i-1,j}, -u_left/2.0/_hx);
665         //      + d(v0*T)/dy
666         add_to_mat(row_index, {i, j+1}, v_top/2.0/_hy);
667         add_to_mat(row_index, {i, j-1}, -v_bot/2.0/_hy);
668         //      - 1 / Re * d^2u/dx^2
669         add_to_mat(row_index, {i, j}, 2.0/_Pe/_hx/_hx);
670         add_to_mat(row_index, {i+1, j}, -1.0/_Pe/_hx/_hx);
671         add_to_mat(row_index, {i-1, j}, -1.0/_Pe/_hx/_hx);
672         //      - 1 / Re * d^2u/dy^2
673         add_to_mat(row_index, {i, j}, 2.0/_Pe/_hy/_hy);
674         add_to_mat(row_index, {i, j+1}, -1.0/_Pe/_hy/_hy);
675         add_to_mat(row_index, {i, j-1}, -1.0/_Pe/_hy/_hy);
676         // = 1.0 / time_step*Told
677         rhs[row_index] += 1.0 / _time_step*_t[row_index];
678     } else {
679         mat.set_value(row_index, row_index, 1.0);

```

```

680     rhs[row_index] = 0;
681 }
682 }
```

для активных ячеек используются формулы (7.8), а для неактивных – тривиальное уравнение  $T = 0$ .

Учёт граничных условий осуществляется за счёт фиктивных узлов в функции `add_to_mat`

```

627 auto add_to_mat = [&](size_t row_index, std::array<size_t, 2> ij_col, double value){
```

Для левой границы условия первого рода (7.10) с  $T^\Gamma = 0$

```

629 // left boundary: T=0
630 mat.add_value(row_index, row_index, -value);
```

для нижней, верхней и выходной – условия второго рода с  $q = 0$  (7.12):

```

632 // right boundary: dT/dn = 0
633 mat.add_value(row_index, row_index, value);
```

Для границы обтекаемого тела – условия первого рода (7.10) с  $T^\Gamma = 1$ :

```

645 // internal boundary: T = 1
646 mat.add_value(row_index, row_index, -value);
647 rhs[row_index] -= 2*value;
```

После сборки правой и левой частей происходит решение СЛАУ и возвращается ответ

```

683 std::vector<double> temperature;
684 AmgMatrixSolver::solve_slae(mat.to_csr(), rhs, temperature);
685 return temperature;
```

#### 7.3.4.4 Вычисление коэффициента теплообмена

Производится в той же функции, где и другие коэффициенты (см. п.7.3.3.3)

```

779 ObstacleNonstat2DSimpleWorker::Coefficients
    → ObstacleNonstat2DSimpleWorker::coefficients() const{
```

В цикле по вертикальным границам

```
787     for (const RegularGrid2D::split_index_t& yface: _grid.boundary_yfaces()) {
```

на левой границе обтекаемого тела согласно формуле (7.16) имеем

```
799         dtdn = (1.0 - _t[left_cell]) / (_hx/2.0);
```

Здесь `left_cell` – индекс ячейки, лежащей слева от рассматриваемого участка границы, а  $T^\Gamma = 1$  – значение температуры из граничного условия. Аналогичные выражения использованы для правых

```
803         dtdn = (1.0 - _t[right_cell]) / (_hx/2.0);
```

нижних

```
826         dtdn = (1.0 - _t[bot_cell]) / (_hy/2.0);
```

и верхних фасок

```
830         dtdn = (1.0 - _t[top_cell]) / (_hy/2.0);
```

После определения нормальной производной по температуре она добавляется в интеграл согласно (6.20). Например, для горизонтальных границ

```
836     sum_nu += dtdn * _hx;
```

#### 7.3.4.5 Результаты расчёта

Настоящую задачу будем решать с параметрами  $\text{Re} = 100$ ,  $\text{Pe} = 100$ ,  $\varepsilon = 10^{-1}$ ,  $\Delta t = 0.1$ ,  $t_{end} = 200$ ,  $n = 10$ .

На рис. 13 представлено поле температуры на разные моменты времени. Видно, что сначала нагреваемая жидкость продвигается вниз по потоку, потом, на момент времени  $t \approx 50$  поток устанавливается, но, в районе  $t \approx 100$  решение теряет устойчивость и за препятствием образуется дорожка Кармана.

На рис. 14 представлено зависимость количества проведённых итераций от времени. На начальном этапе, пока течение развивается от состояния потенциального обтекания (начальное условие), количество итераций велико, затем, с достижением решения локального установления, решение начинает сходится за одну итерацию. Но после  $t > 100$ , когда начинает развиваться неустойчивость, количество итераций вновь возрастает. Количество итераций на слое характеризует степень изменения решения при продвижении к следующему шагу по времени.

Коэффициенты сопротивления, подъёмной силы и теплоотдачи нарисованы на рис. 15. Переход к нестационарному режиму течения характеризуется повышением теплоотдачи и сопротивлению и появлению заметных осцилляций в подъёмной силе.

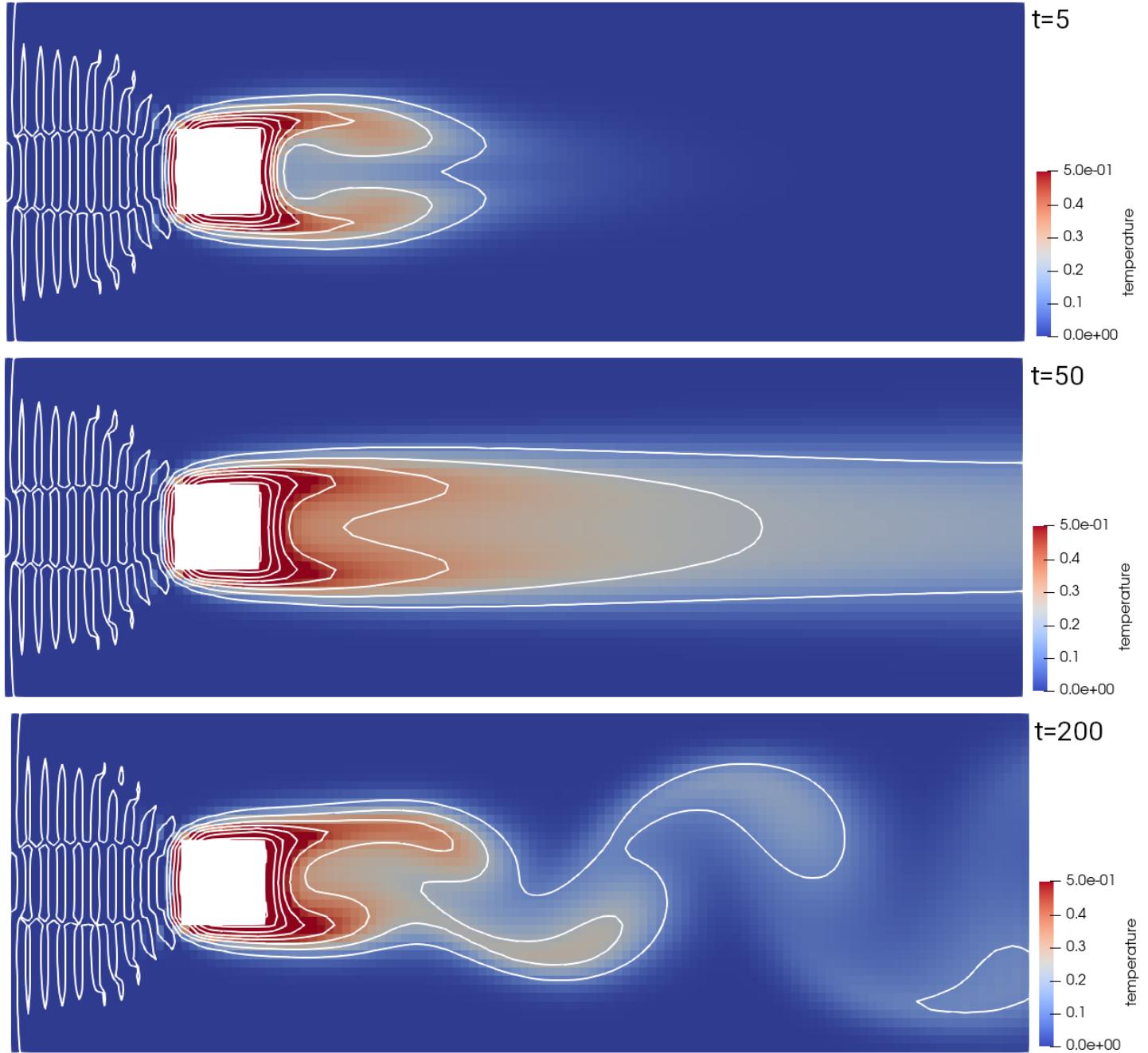


Рис. 13: Температурное поле при решении нестационарной задачи обтекания. Моменты времени  $t = 5$ ,  $t = 50$ ,  $t = 200$

Следует обратить внимание на небольшую рябь в поле температур, заметную слева от препятствия на рис. 13. Её хорошо видно на трёхмерном отображении (см. рис. 16). Если обратить внимание на легенду, то можно заметить, что температура в этой области даже становится отрицательной, что физически невозможно в рамках поставленных граничных условий.

Причина этой ряби кроется в симметричной разности, которую мы использовали для аппроксимации конвективного слагаемого уравнения температуры (см. (7.10)). Известно, что симметричные разности склонны давать осциллирующее решение (даже если оно и устойчиво). Если бы мы использовали схему против потока, то этой нефизичности в решении бы не было, но при этом решение бы имело первый порядок точности по пространству.

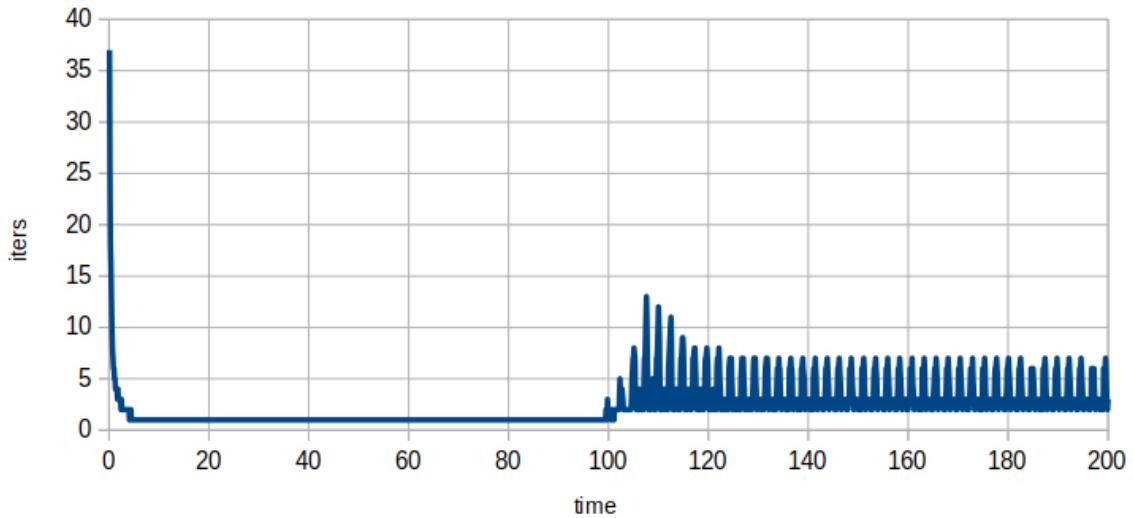


Рис. 14: Зависимость количества внутренних итераций SIMPLE от момента времени

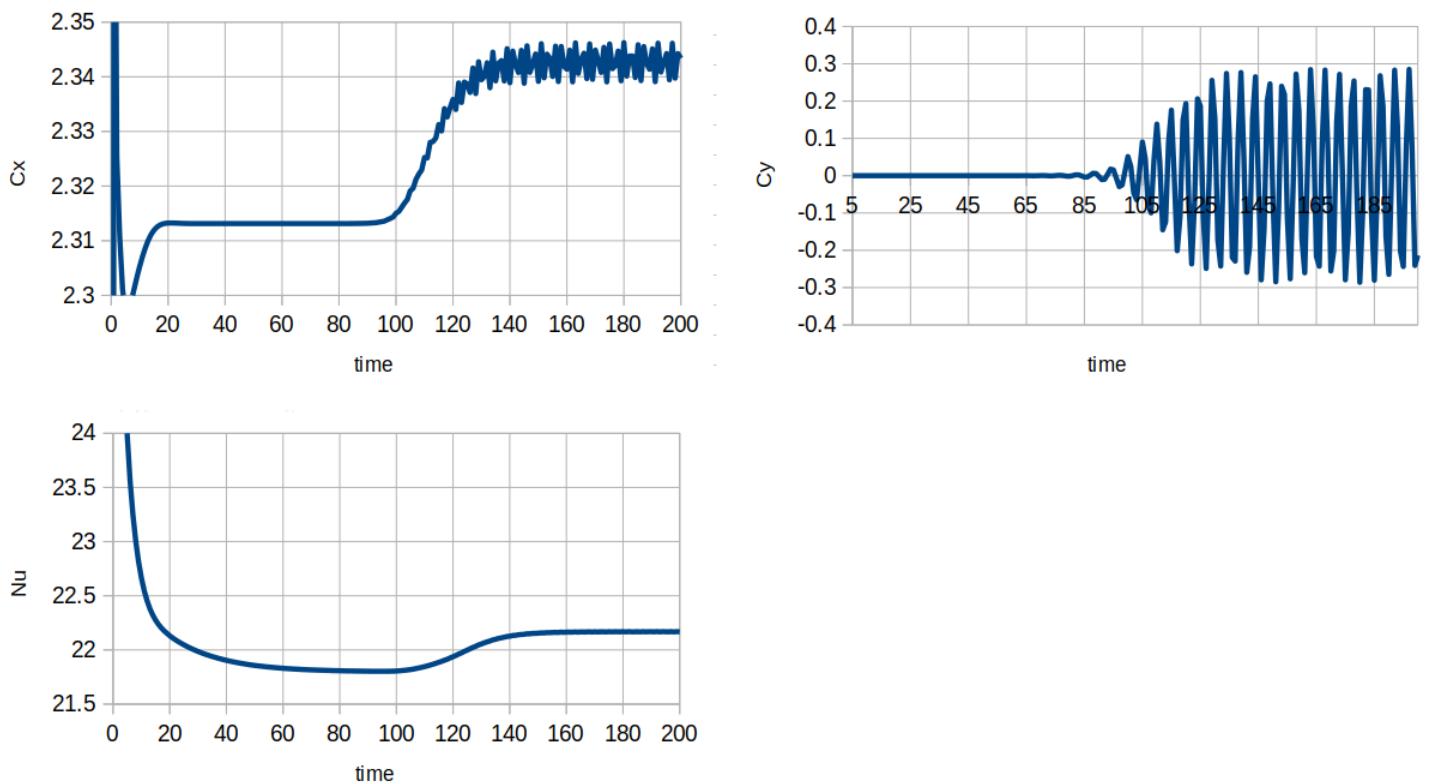


Рис. 15: Эволюция коэффициентов сопротивления  $C_x$ , подъёмной силы  $C_y$  и теплоотдачи Nu

## 7.4 Задание для самостоятельной работы

Решить задачу с двумя обтекаемыми телами: одно расположено в прямоугольнике

$$\gamma_1 : \begin{cases} 2.0 \leq x \leq 2.5, \\ -0.7 \leq y \leq 0.3, \end{cases}$$

второе –

$$\gamma_2 : \begin{cases} 4 \leq x \leq 4.5, \\ -0.3 \leq y \leq 0.7. \end{cases}$$

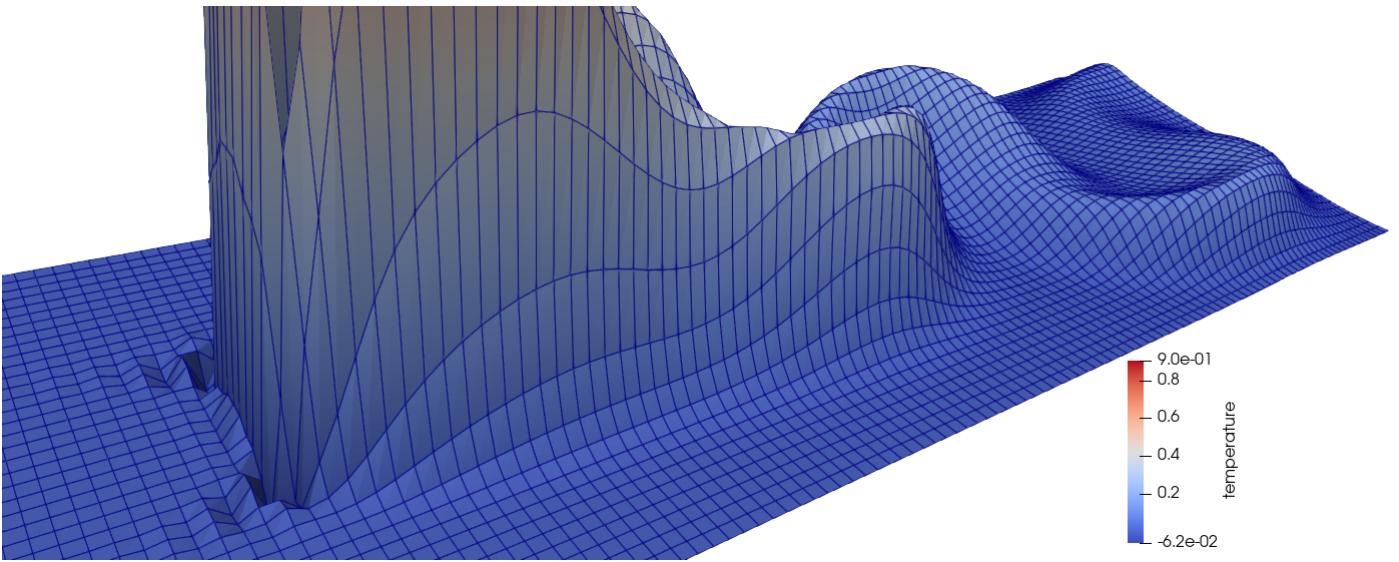


Рис. 16: Температура на момент  $t = 150$  в трёхмерном отображении. Нефизичные осцилляции в области перед препятствием

Использовать условия для температуры:

$$(x, y) \in \gamma_1 : \quad T = 0.5,$$

$$(x, y) \in \gamma_2 : \quad T = 1.0.$$

Область расчёта  $[0, 15] \times [-2, 2]$ :

```
RegularGrid2D grid(0, 15, -2, 2, 15*n_unit, 4*n_unit);
```

Остальные граничные условия использовать те же, что и в рассмотренной в п. 7.3.4 задаче. Параметры задачи:  $Re = 100$ ,  $Pe = 100$ ,  $\varepsilon = 10^{-1}$ ,  $\Delta t = 0.1$ ,  $t_{end} = 200$ ,  $n = 10$ .

Подсчитать коэффициенты сопротивления и теплоотдачи для каждого из двух тел в отдельности. Нарисовать графики из изменения со временем.

Делать на основе программы из файла `obstacle_nonstat_2d_simple_test.cpp`.

**Задание сетки с неактивными ячейками** Обтекаемые препятствия следует задавать при определении сетки. В рассмотренном примере из предыдущего пункта это делалось в строке

```
877    grid.deactivate_cells({2, -0.5}, {3, 0.5});
```

В настоящей задачи нужно эту функцию вызвать два раза, указав там по очереди обе нужные области.

**Задание граничных условий на температуру** Поскольку в задаче граничные условия на обтекаемых телах отличаются по своему значению, то следует модифицировать алгоритм их задания. Граничные условия на температуру задаются в функции `compute_temperature` в строке

```

645    // internal boundary: T = 1
646    mat.add_value(row_index, row_index, -value);
647    rhs[row_index] -= 2*value;

```

Согласно форме (7.10) изменения в левой части не зависит от величины граничной температуры, а в правой – пропорционально ей. Таким образом, для первого тела (где  $T^\Gamma = 0.5$ ) добавка в правую часть будет иметь вид

```
rhs[row_index] -= 2*value*0.5,
```

а для второго – останется такой же, как и раньше.

При этом нужно уметь отличать грани, принадлежащие первому телу от граний, принадлежащих второму. Для этого в классе `ObstacleNonstat2DSimpleWorker` можно объявить функцию, которая определяет ближайшую к ячейке границу. Саму функцию можно реализовать просто используя координаты центра ячейки `_grid.cell_center(icell)`. Например:

```
// => 1 если ячейка icell близка к первому обтекаемому телу и 2 - если ко второму
int ObstacleNonstat2DSimpleWorker::gamma_closest_to_cell(size_t icell){
    double x = _grid.cell_center(icell).x();
    if (x < 3.25) { // центр между первым и вторым телами
        return 1;
    } else {
        return 2;
    }
}
```

Тогда можно написать

```
double t_gamma = (gamma_closest_to_cell(row_index) == 1) ? 0.5 : 1.0;
rhs[row_index] -= 2*t_gamma*value;
```

Здесь запись `double a = (cond) ? 0.5 : 1.0;` есть сокращение от

```
double a;
if (cond){
    a = 0.5;
} else {
    a = 1.0
}
```

**Вычисление коэффициентов** Во-первых следует модифицировать структуру, хранящую коэффициенты, сделав там отдельные записи для каждого тела:

```

struct Coefficients{
    double cpx1, cpx2;
    double cpy1, cpy2;
    double cfx1, cfx2;
    double cfy1, cfy2;
    double cx1, cx2;
    double cy1, cy2;
    double nu1, nu2;
};


```

Во-вторых, в функции сохранения этих коэффициентов в файл в функции

```
ObstacleNonstat2DSimpleWorker::save_current_fields
```

```

cx_writer << time << " ";
cx_writer << coefs.cx1 << " " << coefs.cy1 << " " << coefs.nu1 << " ";
cx_writer << coefs.cx2 << " " << coefs.cy2 << " " << coefs.nu2 << std::endl;


```

Соответственно можно поправить легенду в функции `initialize_saver()`.

Сами коэффициенты следует вычислять в функции

```
coefficients(). Там нужно завести агрегаторы на оба тела:
```

```

double sum_cpx1 = 0;
double sum_cpy1 = 0;
double sum_cfx1 = 0;
double sum_cfy1 = 0;
double sum_nu1 = 0;
double sum_cpx2 = 0;
double sum_cpy2 = 0;
double sum_cfx2 = 0;
double sum_cfy2 = 0;
double sum_nu2 = 0;


```

И далее заполнять их в зависимости от близости ячейки. Так же следует учесть значение граничной температуры при вычислении  $\partial T / \partial n$  по формуле (7.16)

Например, для вертикальных граней после определения ячейки `left_cell`:

```

int gamma_i = gamma_closest_to_cell(left_cell);
double t_gamma = (gamma_i == 1) ? 0.5 : 1.0;


```

далее учесть при вычислении `dtdn` (два раза)

```

dtdn = (t_gamma - _t[left_cell]) / (_hx/2.0);


```

а также при выборе агрегатора

```
if (gamma_i == 1){  
    sum_cpx1 += pnx * _hy;  
    sum_cfy1 += dvdn * _hy;  
    sum_nu1 += dtdn * _hy;  
}  
else {  
    sum_cpx2 += pnx * _hy;  
    sum_cfy2 += dvdn * _hy;  
    sum_nu2 += dtdn * _hy;  
}
```

Аналогичную процедуру следует проделать и для горизонтальных граней.

В конце нужно правильным образом заполнить все поля переменной **coef**.

```
coefs.cpx1 = 2.0*sum_cpx1;  
coefs.cpx2 = 2.0*sum_cpx2;  
...
```

# 8 Лекция 8 (28.10)

## 8.1 Метод конечных объёмов

### 8.1.1 Уравнение Пуассона

Пространственную аппроксимацию дифференциальных операторов методом конечных объёмов рассмотрим на примере многомерного уравнения Пуассона

$$-\nabla^2 u = f, \quad (8.1)$$

которое требуется решить в области  $D$ . Разобъём эту область на непересекающиеся подобласти  $E_i$ ,  $i = \overline{0, N - 1}$  (рис. 17). Центры ячеек обозначим как  $\mathbf{c}_i$ .

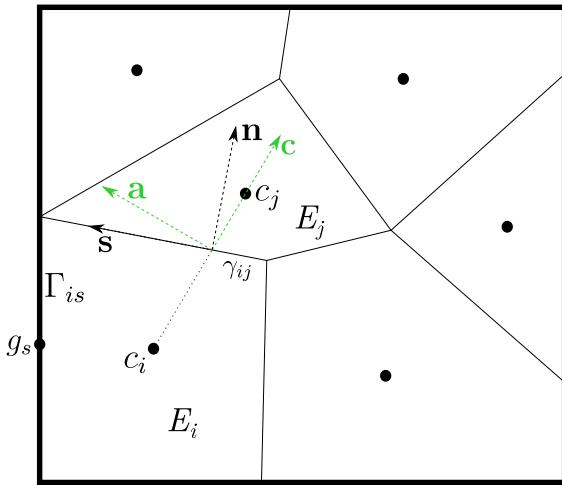


Рис. 17: Конечнообъёмная сетка

Проинтегрируем исходное уравнение по одной из подобластей  $E_i$ :

$$-\int_{E_i} \nabla^2 u \, ds = \int_{E_i} f \, d\mathbf{x}.$$

К интегралу в левой части применим формулу интегрирования по частям (A.13). Получим

$$-\int_{\partial E_i} \frac{\partial u}{\partial n} \, ds = \int_{E_i} f \, d\mathbf{x}. \quad (8.2)$$

Здесь  $\partial E_i$  – совокупность всех границ подобласти  $E_i$ , а  $\mathbf{n}$  – внешняя к подобласти нормаль.

Граница ячейки  $E_i$  состоит из внутренних граней  $\gamma_{ij}$  (индекс  $j$  здесь соответствует индексу соседней ячейки) и граней  $\Gamma_{is}$ , лежащих на внешней границе расчётной области  $D$ . Тогда интеграл по общей границе ячейки распишется через сумму интегралов по плоским поверхностям

$$\int_{\partial E_i} \frac{\partial u}{\partial n} \, ds = \sum_j \int_{\gamma_{ij}} \frac{\partial u}{\partial n} \, ds + \sum_s \int_{\Gamma_{is}} \frac{\partial u}{\partial n} \, ds.$$

Аппроксимируем производную  $\partial u / \partial n$  на каждой из граней константой. Тогда её можно вынести из под интегралов и предыдущее выражение записать в виде

$$\int_{\partial E_i} \frac{\partial u}{\partial n} ds \approx \sum_j |\gamma_{ij}| \left( \frac{\partial u}{\partial n} \right)_{\gamma_{ij}} + \sum_s |\Gamma_{is}| \left( \frac{\partial u}{\partial n} \right)_{\Gamma_{is}} \quad (8.3)$$

Аналогично, анализируя интеграл правой части (8.2), приблизим значение функции правой части  $f$  внутри элемента  $E_i$  константой  $f_i$ , которую отнесём к центру элемента. Тогда

$$\int_{E_i} f d\mathbf{x} \approx f_i |E_i|. \quad (8.4)$$

### 8.1.1.1 Обработка внутренних граней

Рассмотрим значение нормальной производной по грани  $\gamma_{ij}$ , входящее в первое слагаемое правой части (8.3). Для двумерного случая распишем градиент  $u$  в системе координат, образованной единичными векторами нормали  $\mathbf{n}$  и касательной  $\mathbf{s} = (-n_y, n_x)$  к грани  $\gamma_{ij}$ :

$$\nabla u = \frac{\partial u}{\partial n} \mathbf{n} + \frac{\partial u}{\partial s} \mathbf{s}.$$

Теперь введём другую систему координат, которая образована векторами  $\mathbf{c}$  – нормированный вектор  $\mathbf{c}_j - \mathbf{c}_i$  и перпендикулярного к нему единичного вектора  $\mathbf{c}^\perp = \mathbf{a} = (-c_y, c_x)$  (см. зелёные вектора на рис. 17). В этой системе

$$\nabla u = \frac{\partial u}{\partial c} \mathbf{c} + \frac{\partial u}{\partial a} \mathbf{a}.$$

Пользуясь формулами поворота систем координат  $(\mathbf{c}, \mathbf{a}) \rightarrow (\mathbf{n}, \mathbf{s})$  искомую производную можно записать

$$\frac{\partial u}{\partial n} = \frac{\partial u}{\partial c} \cos(\widehat{\mathbf{c}, \mathbf{n}}) + \frac{\partial u}{\partial a} \sin(\widehat{\mathbf{c}, \mathbf{n}}). \quad (8.5)$$

Можно рассмотреть и обратный поворот  $(\mathbf{n}, \mathbf{s}) \rightarrow (\mathbf{c}, \mathbf{a})$ :

$$\frac{\partial u}{\partial c} = \frac{\partial u}{\partial n} \cos(\widehat{\mathbf{n}, \mathbf{c}}) + \frac{\partial u}{\partial s} \sin(\widehat{\mathbf{n}, \mathbf{c}}).$$

Тогда

$$\frac{\partial u}{\partial n} = \frac{\partial u}{\partial c} \frac{1}{\cos(\widehat{\mathbf{n}, \mathbf{c}})} - \frac{\partial u}{\partial s} \tan(\widehat{\mathbf{n}, \mathbf{c}}). \quad (8.6)$$

Таким образом, мы получили два соотношения для определения производной  $\partial u / \partial n$ : (8.5), (8.6).

Отметим, что в трёхмерном случае эти формулы так же остаются справедливыми. При этом векторы  $\mathbf{s}$  и  $\mathbf{a}$  следует строить в плоскости, образованной векторами  $\mathbf{n}$  и  $\mathbf{c}$ :

$$\mathbf{s} = \frac{(\mathbf{n} \times \mathbf{c}) \times \mathbf{n}}{|(\mathbf{n} \times \mathbf{c}) \times \mathbf{n}|}, \quad \mathbf{a} = \frac{(\mathbf{n} \times \mathbf{c}) \times \mathbf{c}}{|(\mathbf{n} \times \mathbf{c}) \times \mathbf{c}|}.$$

При выводе этих формул используется тот факт, что результат векторного произведения перпендикулярен плоскости, образованной его аргументами.

Определим значения функции  $u$  в точках  $c_i, c_j$  как  $u_i, u_j$ . Тогда входящая в оба соотношения

(8.5), (8.6) производная  $\partial u / \partial c$  может быть приближена конечной разностью

$$\frac{\partial u}{\partial c} \approx \frac{u_j - u_i}{|\mathbf{c}_j - \mathbf{c}_i|}.$$

Вторые слагаемые в правых частях (8.5), (8.6) можно в первом приближении отбросить, если считать, что угол между векторами  $\mathbf{c}$  и  $\mathbf{n}$  близок к нулю:  $\widehat{\mathbf{n}, \mathbf{c}} \approx 0$ . Тогда искомую производную можно записать в виде:

$$\frac{\partial u}{\partial n} \approx \frac{u_j - u_i}{h_{ij}}, \quad (8.7)$$

где эффективное расстояние  $h_{ij}$  между узлами  $c_j$  и  $c_i$  для приближения (8.5) запишется как

$$h_{ij} = \frac{|\mathbf{c}_j - \mathbf{c}_i|}{\cos(\widehat{\mathbf{n}, \mathbf{c}})} = \frac{|\mathbf{c}_j - \mathbf{c}_i|^2}{(\mathbf{c}_j - \mathbf{c}_i) \cdot \mathbf{n}}. \quad (8.8)$$

а для (8.6) –

$$h_{ij} = |\mathbf{c}_j - \mathbf{c}_i| \cos(\widehat{\mathbf{n}, \mathbf{c}}) = (\mathbf{c}_j - \mathbf{c}_i) \cdot \mathbf{n} \quad (8.9)$$

Здесь для упрощений было использовано соотношение  $\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos(\widehat{\mathbf{a}, \mathbf{b}})$  и единичная длина вектора нормали:  $|\mathbf{n}| = 1$ .

Если вектора  $\mathbf{c}$  и  $\mathbf{n}$  сонаправлены, то  $\cos(\widehat{\mathbf{n}, \mathbf{c}}) = 0$  и тогда формулы (8.8), (8.9) идентичны. Если же при этом равны и расстояния от точек  $c_j$ ,  $c_i$  до границы  $\gamma_{ij}$ , то конечная разность (8.7) является симметричной и поэтому имеет второй порядок аппроксимации. Сетки, которые сохраняют такие свойства, называются реби-сетками (perpendicular bisector). Строятся такие сетки на основе ячеек Вороного.

Для сильно скошенных сеток кажется, что использование формулы (8.8) безопаснее чем (8.9). Потому что отброшенное из формулы (8.6) слагаемое

$$\frac{\partial u}{\partial s} \tan(\widehat{\mathbf{n}, \mathbf{c}})$$

стремится к бесконечности в вырожденном случае  $\widehat{\mathbf{n}, \mathbf{c}} \rightarrow \frac{\pi}{2}$ . Однако следует понимать, что обе эти формулы имеют одинаковый первый порядок точности (это следует из разложения синуса и тангенса вокруг нуля).

### 8.1.1.2 Учёт граничных условий

Для вычисления второго слагаемого в правой части (8.3) следует расписать значение нормальной к границе производной вида

$$\left( \frac{\partial u}{\partial n} \right)_{\Gamma_{is}}.$$

Это делается с помощью граничных условий. Далее рассмотрим постановку трёх видов граничных условий.

**Граничные условия первого рода** Пусть на центре грани  $\Gamma_{is}$  задано значение искомой функции

$$\mathbf{x} \in \Gamma_{is} : \quad u(\mathbf{x}) = u^\Gamma.$$

Аппроксимацию производных будем проводить из тех же соображений, которые использовали при анализе внутренних граней. Только вместо центра соседнего элемента  $c_j$  будем использовать центр грани  $g_s$ . В первом приближении, отбрасывая касательные производные, придём к формуле аналитической (8.7):

$$\frac{\partial u}{\partial n} \approx \frac{u^\Gamma - u_i}{h_{is}}, \quad (8.10)$$

где эффективное расстояние  $h_{is}$  зависимости от использованного подхода (8.5) или (8.6) вычисляется по одному из соотношений

$$h_{is} = \frac{|\mathbf{g}_s - \mathbf{c}_i|^2}{(\mathbf{g}_s - \mathbf{c}_i) \cdot \mathbf{n}}, \quad (8.11)$$

$$h_{is} = (\mathbf{g}_s - \mathbf{c}_i) \cdot \mathbf{n}. \quad (8.12)$$

**Границные условия второго рода** Учёт условий второго рода тривиален. Если на центре грани  $\Gamma_{is}$  задано значение нормальной производной

$$\mathbf{x} \in \Gamma_{is} : \quad \frac{\partial u}{\partial n} = q, \quad (8.13)$$

то это значение просто подставляется вместо соответствующей производной в (8.3).

**Границные условия третьего рода** Теперь рассмотрим условия третьего рода

$$\mathbf{x} \in \Gamma_{is} : \quad \frac{\partial u}{\partial n} = \alpha u + \beta.$$

Распишем производную в форме (8.10):

$$\frac{u^\Gamma - u_i}{h_{is}} = \alpha u^\Gamma + \beta,$$

откуда выразим  $u^\Gamma$ :

$$u^\Gamma = \frac{u_i + \beta h_{is}}{1 - \alpha h_{is}}.$$

Подставляя это выражение в исходное граничное условие получим

$$\frac{\partial u}{\partial n} \approx \frac{\alpha}{1 - \alpha h_{is}} u_i + \frac{\beta}{1 - \alpha h_{is}}. \quad (8.14)$$

### 8.1.2 Одномерный случай

Рассмотрим результат конечнообъёмной аппроксимации задачи (8.1) в одномерном случае на равномерной сетке с шагом  $h$  (рис. 18).

У внутренней ячейки  $i$  есть две границы:  $\gamma_{i,i-1}$  и  $\gamma_{i,i+1}$ . Нормали по этим границам аппроксими-

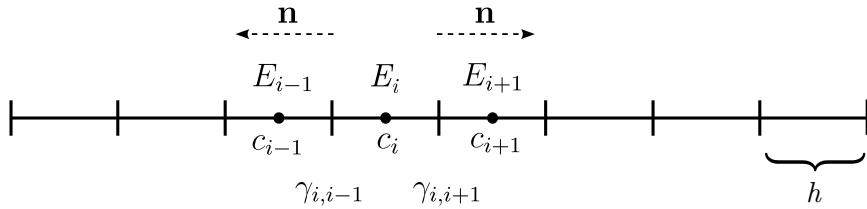


Рис. 18: Одномерная конечнообъёмная сетка

руются по формулам (8.7):

$$\gamma_{i,i-1} : \frac{\partial u}{\partial n} = \frac{u_{i-1} - u_i}{h}$$

$$\gamma_{i,i+1} : \frac{\partial u}{\partial n} = \frac{u_{i+1} - u_i}{h}$$

Объём ячейки в одномерном случае равен её длине  $h$ . Площадь грани следует положить единице с тем, чтобы

$$|E_i| = |\gamma| h = h.$$

Тогда, подставляя эти значения в (8.2), получим знакомую конечноразностную схему аппроксимации уравнения Пуассона

$$\frac{-u_{i-1} + 2u_i - u_{i+1}}{h} = f_i h,$$

которая имеет второй порядок точности. Разница с методом конечных разностей здесь состоит в том, что значения сеточных векторов  $\{u\}$ ,  $\{f\}$  здесь приписаны к центрам ячеек, а не к их узлам. Это отличие проявит себя в аппроксимации граничных условий. Так, если на левой границе задано условие первого рода, то соответствующее уравнение согласно (8.10) примет вид

$$-\frac{u^\Gamma - u_0}{h/2} - \frac{u_1 - u_0}{h} = f_0 h.$$

В методе конечных разностей это условие выразилось бы в виде  $u_0 = u^\Gamma$ .

### 8.1.3 Сборка системы линейных уравнений

Подставим все полученные аппроксимации (8.7), (8.10), (8.13), (8.14) в уравнение (8.2):

$$-\sum_j \frac{|\gamma_{ij}|}{h_{ij}} (u_j - u_i) - \sum_{s \in I} \frac{|\Gamma_{is}|}{h_{is}} (u^\Gamma - u_i) - \sum_{s \in II} |\Gamma_{is}| q - \sum_{s \in III} \frac{|\Gamma_{is}|}{1 - \alpha h_{is}} (\alpha u_i + \beta) = f_i |E_i|.$$

Здесь первое слагаемое в левой части отвечает за потоки через внутренние границы, второе – граничные условия первого рода, третье – граничные условия второго рода и четвёртое – граничные условия третьего рода. Далее перенесём все известные значения в правую часть и окончательно

получим линейное уравнение для  $i$ -го конечного объёма:

$$\begin{aligned} & \sum_j \frac{|\gamma_{ij}|}{h_{ij}} (u_i - u_j) + \sum_{s \in \text{I}} \frac{|\Gamma_{is}|}{h_{is}} u_i - \sum_{s \in \text{III}} \frac{\alpha |\Gamma_{is}|}{1 - \alpha h_{is}} u_i \\ &= f_i |E_i| + \sum_{s \in \text{I}} \frac{|\Gamma_{is}|}{h_{is}} u^\Gamma + \sum_{s \in \text{II}} |\Gamma_{is}| q + \sum_{s \in \text{III}} \frac{\beta |\Gamma_{is}|}{1 - \alpha h_{is}}. \end{aligned} \quad (8.15)$$

Таким образом мы получили систему из  $N$  (по количеству подобластей) линейных уравнений относительно неизвестного сеточного вектора  $\{u_i\}$

$$Au = b.$$

#### 8.1.3.1 Алгоритм сборки в цикле по ячейкам

Матрицу  $A$  и правую часть  $b$  системы (8.15) можно собирать в цикле по ячейкам: строчка за строчкой. Такой алгоритм выглядел бы следующим образом

```

for  $i = \overline{0, N - 1}$  – цикл по строкам СЛАУ
     $b_i = |E_i| f_i$ 
    for  $j \in \text{nei}(i)$  – цикл по ячейкам, соседним с ячейкой  $i$ 
         $v = |\gamma_{ij}| / h_{ij}$ 
         $A_{ii} += v$ 
         $A_{ij} -= v$ 
    endfor
    for  $s \in \text{bnd1}(i)$  – цикл по граням ячейки  $i$  с условиями первого рода
         $v = |\Gamma_{is}| / h_{is}$ 
         $A_{ii} += v$ 
         $b_i += u^\Gamma v$ 
    endfor
    for  $s \in \text{bnd2}(i)$  – цикл по граням ячейки  $i$  с условиями второго рода
         $b_i += q |\Gamma_{is}|$ 
    endfor
    for  $s \in \text{bnd3}(i)$  – цикл по граням ячейки  $i$  с условиями третьего рода
         $v = |\Gamma_{is}| / (1 - \alpha h_{is})$ 
         $A_{ii} -= \alpha v$ 
         $b_i += \beta v$ 
    endfor
endfor

```

Первым недостатком такого алгоритма является наличие вложенных циклов. Во-вторых, коэффициент, отвечающий за поток через внутреннюю грань  $\gamma_{ij}$ , равный  $|\gamma_{ij}| / h_{ij}$  в таком алгоритме будет учитываться дважды: в строке  $i$  и в строке  $j$ .

### 8.1.3.2 Алгоритм сборки в цикле по граням

Вместо общего цикла по ячейкам, будем использовать цикл по граням. В таком цикле коэффициенты потоков будут вычисляться один раз и вставляться сразу в две строки матрицы, соответствующие соседним с гранью ячейкам. Вложенных циклов в такой постановке удаётся избежать, потому что у грани есть только две соседние ячейки (в то время как у ячейки может быть произвольное количество соседних граней).

Разделим все грани на исходной сетки на внутренние и граничные (отдельный набор для каждого вида граничных условий). Тогда для внутренних граней можно записать

```

for  $s \in \text{internal}$            – цикл по внутренним граням
     $i, j = \text{nei\_cells}(s)$    – две ячейки, соседние с текущей гранью
     $v = |\gamma_{ij}|/h_{ij}$ 
     $A_{ii} += v; A_{jj} += v$    – диагональные коэффициенты матрицы
     $A_{ij} -= v; A_{ji} -= v$    – внедиагональные коэффициенты матрицы
endfor

```

Граничные условия учитываются в отдельных циклах. Здесь будем учитывать, что у грани, принадлежащей границе области, есть только одна соседняя ячейка. Условия первого рода:

```

for  $s \in \text{bnd1}$            – грани с условиями первого рода
     $i = \text{nei\_cells}(s)$    – соседняя с граничной гранью ячейка
     $v = |\Gamma_{is}|/h_{is}$ 
     $A_{ii} += v$ 
     $b_i += u^\Gamma v$ 
endfor

```

Условия второго рода:

```

for  $s \in \text{bnd2}$            – грани с условиями второго рода
     $i = \text{nei\_cells}(s)$    – соседняя с граничной гранью ячейка
     $b_i += |\Gamma_{is}|q$ 
endfor

```

Условия третьего рода:

```

for  $s \in \text{bnd3}$            – грани с условиями третьего рода
     $i = \text{nei\_cells}(s)$    – соседняя с граничной гранью ячейка
     $v = |\Gamma_{is}|/(1 + \alpha h_{is})$ 
     $A_{ii} -= \alpha v$ 
     $b_i += \beta v$ 
endfor

```

Первое слагаемое в правой части (8.15) учтём отдельным циклом:

```

for  $i = \overline{0, N - 1}$  – цикл по ячейкам
     $b_i = |E_i|f_i$ 
endfor

```

(8.20)

## 8.2 Работа с конечнообъёмной сеткой

Конечнообъёмная сетка разбивает область решения  $D$  на непересекающиеся подобласти (ячейки)  $E_i$  с плоскими гранями. Выпуклость каждой из подобластей математически не требуется, но желательна для качественной аппроксимации нормальных производных через грань. Для ячеек, узлов и граней сетки вводится нумерация.

Для реализации сборки системы линейных уравнений по алгоритму (8.16) – (8.20) необходимо уметь вычислять следующие параметры конечнообъёмной сетки:

- таблица связности грань-ячейка
- объём ячейки
- центр ячейки
- площадь грани
- центр грани
- нормаль к грани.

### 8.2.1 Двумерная сетка

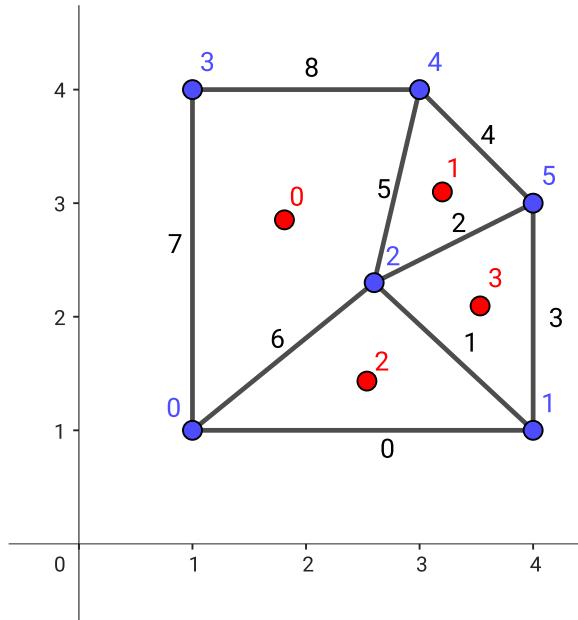


Рис. 19: Двумерная конечнообъёмная сетка. Нумерация узлов (синим), ячеек (красным) и граней (чёрным)

### 8.2.1.1 Определение двумерной конечнообъемной сетки

Подобласти представляют собой полигоны с произвольным количеством узлов. Для однозначного задания такой сетки достаточно таблицы с координатами узлов и таблицы, определяющей ячейки как последовательности узлов против часовой стрелки (таблица связности “ячейка–узел”). Для сетки, представленной на рис. 20 эти таблицы будут иметь вид, представленный в таблицах 1, 2.

Узел	X	Y	Ячейка	Узлы
0	1	1	0	0 2 4 3
1	4	1	1	2 5 4
2	2.6	2.3	2	0 1 2
3	1	4	3	1 5 2
4	3	4		
5	4	3		

Таблица 2: Таблица “ячейка–узел”

Таблица 1: Таблица узлов

### 8.2.1.2 Вспомогательные таблицы связности

На основании таблицы “ячейка–узел” можно собрать все присутствующие в сетке грани в таблицу связности “грань–узел” (см. таблицу 3). Направление отрезка границы здесь выбирается произвольно.

Грань	Начальный узел	Конечный узел
0	0	1
1	1	2
2	2	5
3	1	5
4	4	5
5	2	4
6	0	2
7	0	3
8	3	4

Таблица 3: Таблица связности “грань–узел”

Так же необходимо собрать таблицу связности “грань–ячейка” (таблица 4). Очевидно, у внутренних граней будет две соседние ячейки, а у граничных – одна. При сборке этой таблицы будем учитывать направление отрезка грани. На первую позицию будем помещать ячейку, расположенную справа от отрезка, а на вторую – слева. Первую позицию будем называть отрицательной стороной (поскольку она расположена против направления нормали к этой грани), а вторую – положительной. Если с какой-то из сторон ячейка отсутствует (для граничных граней), то на соответствующее место поставим невалидный индекс (в зависимости от реализации это может быть  $-1$ , максимальное положительное число и т.п.). Наличие невалидного индекса в таблице “грань–ячейка” можно использовать для выделения граничных граней.

Грань	Ячейка справа (отрицательная)	Ячейка слева (положительная)
0	–	2
1	3	2
2	3	1
3	–	3
4	1	–
5	1	0
6	2	0
7	0	–
8	0	–

Таблица 4: Таблица связности “грань–ячейка”

### 8.2.1.3 Геометрические свойства сетки

После определения всех связной, можно приступить к вычислению геометрических параметров сетки. На основании таблицы узлов (таб. 1) и таблицы “грань–узел” (таб. 3) вычислим площади граней  $|\gamma|$  (в двумерном случае площадь грани – есть длина отрезка грани) по формуле

$$|\gamma| = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2},$$

где  $x_{0,1}, y_{0,1}$  – координаты начальной и конечной точек отрезка грани (таб. 5). Центр грани  $\mathbf{g}$  (таб. 6) определяется как геометрический центр соответствующего отрезка

$$g_x = \frac{x_0 + x_1}{2}, \quad g_y = \frac{y_0 + y_1}{2}.$$

Вектор единичной нормали к граням  $\mathbf{n}$  (таб. 7), смотрящий влево от направленного отрезка грани, вычисляется по формуле

$$n_x = -\frac{y_1 - y_0}{|\gamma|}, \quad n_y = \frac{x_1 - x_0}{|\gamma|}.$$

Грань	Площадь
0	3
1	1.91
2	1.57
3	2
4	1.41
5	1.75
6	2.06
7	3
8	2

Таблица 5: Площадь граней

Таблица 6: Центр граней

Грань	$n_x$	$n_y$
0	0	1
1	-0.68	-0.73
2	-0.45	0.89
3	-1	0
4	0.71	0.71
5	-0.97	0.23
6	-0.63	0.78
7	-1	0
8	0	1

Таблица 7: Нормаль к граням

Для вычисления объёмов (площадей) (таб. 8) и центров масс (таб. 9) ячеек воспользуемся таблицей “ячейка–узел”, где задана нумерация узлов ячеек против часовой стрелки. Алгоритмы вычисления представлены в пп. А.4.2.1, А.4.2.3.

Ячейка	Объём
0	4.1
1	1.05
2	1.95
3	1.4

Таблица 8: Объём ячеек

Ячейка	Центр X	Центр Y
0	1.81	2.85
1	3.2	3.1
2	2.53	1.43
3	3.53	2.1

Таблица 9: Центр ячеек

## 8.2.2 Трёхмерная сетка

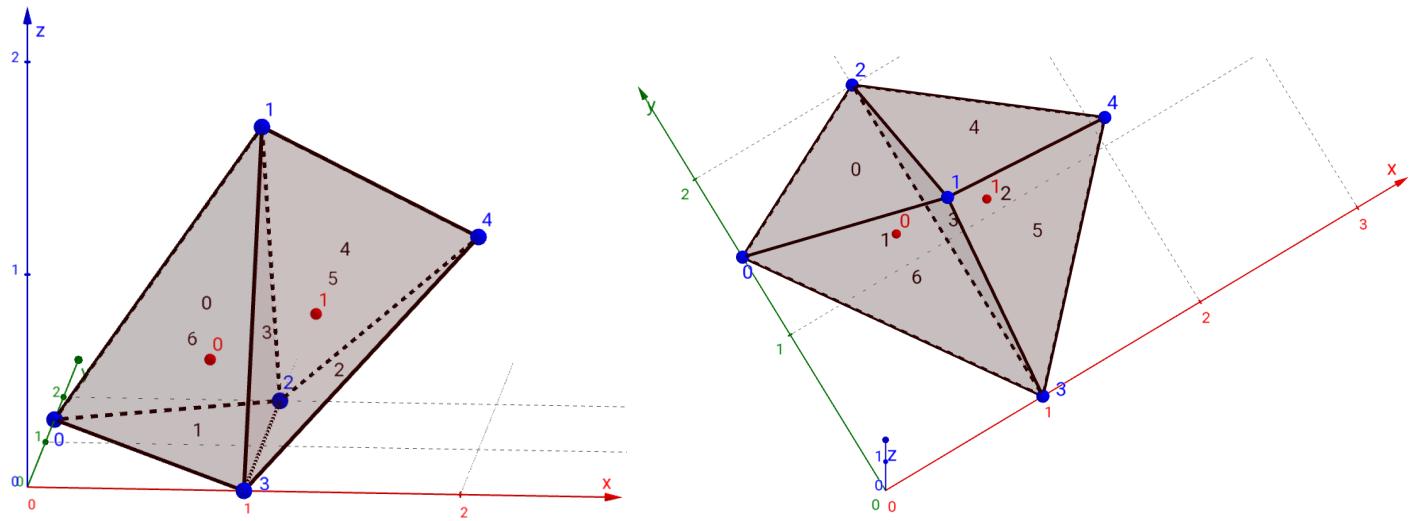


Рис. 20: Трёхмерная конечнообъёмная сетка в двух проекциях. Нумерация узлов (синим), ячеек (красным) и граней (чёрным)

### 8.2.2.1 Определение трёхмерной конечнообъёмной сетки

Подобласти представляют собой многогранники с произвольным количеством многоугольных граней. Для примера рассмотрим сетку, представленную на рисунке 20. Чтобы задать такую сетку однозначно, необходимы

- таблица узлов (таб. 10);
- таблица “грань-узел” (таб. 11), где набор условий задан в последовательном порядке. Направление закрутки здесь произвольное;
- таблица “грань-ячейка” (таб. 12). Порядок задания ячеек зависит от направления закрутки, заданной в предыдущей таблице (“грань-узел”). Будем смотреть на грань со стороны нормали (с этой стороны обход узлов будет против часовой стрелки). Тогда на первом месте должна стоять ячейка, находящаяся за плоскостью грани (отрицательная сторона), а на втором – перед плоскостью грани (положительная сторона). Если с одной из сторон нет ячейки (для границ), то на это место ставится невалидный индекс.

Узел	X	Y	Z
0	0	1.5	0
1	1	1	1.5
2	1	2	0
3	1	0	0
4	2	1	1

Таблица 10: Таблица узлов

Грань	Узлы
0	0 1 2
1	0 3 2
2	3 2 4
3	3 2 1
4	2 1 4
5	3 4 1
6	3 0 1

Таблица 11: Таблица “грань–узел”

Грань	Ячейка снизу (отрицательная)	Ячейка сверху (положительная)
0	0	–
1	–	0
2	1	–
3	0	1
4	1	–
5	1	–
6	–	0

Таблица 12: Таблица “грань–ячейка”

### 8.2.2.2 Геометрические свойства сетки

Площадь (таб. 13) и центр (таб. 14) граней вычисляются согласно алгоритмам из пп. A.4.2.1, A.4.2.3. Для определения нормали к плоской грани (таб. 15) необходимо взять три последовательные, не лежащие на одной прямой точки грани из упорядоченной таблицы связности “грань–узел”:  $\mathbf{p}_0$ ,  $\mathbf{p}_1$ ,  $\mathbf{p}_2$ , и провести векторное умножение:

$$\mathbf{n} = \frac{\mathbf{k}}{|\mathbf{k}|}, \quad \mathbf{k} = (\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0).$$

Грань	Площадь	Грань	$g_x$	$g_y$	$g_z$	Грань	$n_x$	$n_y$	$n_z$
0	0.98	0	0.67	1.5	0.5	0	-0.38	0.77	0.51
1	1	1	0.67	1.17	0	1	0	0	1
2	1.41	2	1.33	1	0.33	2	0.71	0	-0.71
3	1.5	3	1	1	0.5	3	1	0	0
4	0.94	4	1.33	1.33	0.83	4	0.27	0.8	0.53
5	0.94	5	1.33	0.67	0.83	5	0.27	-0.8	0.53
6	1.44	6	0.67	0.83	0.5	6	0.78	0.52	-0.35

Таблица 13: Площадь граней

Таблица 14: Центр граней

Таблица 15: Нормаль к граням

Вычисление объёма (таб. 16) и центра масс (таб. 17) ячеек осуществляется по формулам из пп. A.4.3.1, A.4.3.3 соответственно.

Ячейка	Объём
0	0.5
1	0.5

Таблица 16: Объём ячеек

Ячейка	$c_x$	$c_y$	$c_z$
0	0.75	1.13	0.38
1	1.25	1	0.63

Таблица 17: Центр ячеек

### 8.2.3 Интегрирование сеточной функции

Пусть задана сеточная функция  $\{u_i\}$ , которая аппроксимирует функцию  $u$ . Интеграл по области расчёта от этой функции можно расписать через сумму интегралов по каждой ячейке и далее воспользоваться тем фактом, что значение аппроксимированной функции внутри ячейки постоянно:

$$\int_D u \, d\mathbf{x} = \sum_{i=0}^{N-1} \int_{E_i} u \, d\mathbf{x} \approx \sum_{i=0}^{N-1} u_i |E_i|. \quad (8.21)$$

## 8.3 Пример расчётной программы

Рассмотрим пример решения двумерного уравнения (8.1) с граничными условиями первого рода. Для тестирования методики действовать будем по аналогии из п.2.1.2.2:

- Зададим точное решение в виде

$$u^e = \cos(10x^2) \sin(10y) + \sin(10x^2) \cos(10x);$$

- Расчитаем правую часть прямым дифференцированием

$$f = -\frac{\partial^2 u^e}{\partial x^2} - \frac{\partial^2 u^e}{\partial y^2};$$

- Используя подсчитанную  $f$  применим алгоритм метода конечных объёмов для получения численного решения  $u$ ;
- Для вычисления отклонения численного решения от точного подсчитаем интеграл вида (2.8):

$$\|u - u^e\|_2 = \sqrt{\frac{1}{D} \int_D (u - u^e)^2 \, d\mathbf{x}}. \quad (8.22)$$

Пример программы лежит в файле `poisson_fvm_solve_test.cpp` в тесте

[ ]. Программа использует регулярную двумерную сетку в единичном квадрате квадрате с разбиением в 20 ячеек по каждой оси. После расчёта файл с численным и точным решениями сохраняется в файл `poisson2_fvm.vtk`, а на печать выводится количество ячеек в сетке и полученная норма отклонения. Результат работы программы представлен на рис. 21. Поскольку вектор решений задан в центрах ячеек, то его отображение имеет мозаичный вид.

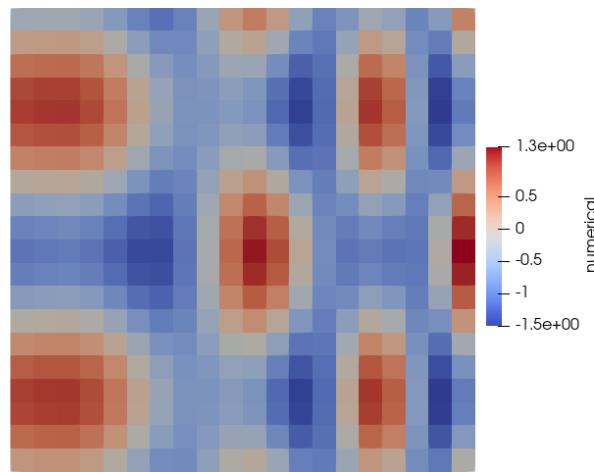


Рис. 21: Результат расчёта

### 8.3.1 Работа с сеткой

Несмотря на то, что на вход подаётся регулярная сетка, все алгоритмы используют сетку через абстрактный интерфейс `IGrid`, который определён для сеток произвольной структуры и размерности. Этот интерфейс (полностью объявленный в файле

`grid/i_grid.hpp`) предоставляет следующие функции, используемые в алгоритмах (8.16) – (8.20):

- `IGrid::face_normal` – вектор нормали для заданной грани;
- `IGrid::tab_face_cell` – таблица связности грань–ячейка. Возвращает пару индексов ячеек. Первый из этих индексов соответствует ячейке, лежащей в направлении, противоположенном направлению нормали заданной грани. Для граничных граней один из этих индексов (в зависимости от направления нормали этой грани) равен глобальной константе `INVALID_INDEX`;
- `IGrid::face_center` – центр грани;
- `IGrid::face_area` – значение площади заданной грани;
- `IGrid::cell_center` – центр ячейки;
- `IGrid::cell_volume` – объём ячейки.

Конкретная реализация этих функций зависит от вида сетки. Так, для структурированной сетки `RegularGrid2D` их (триivialная для таких сеток) реализация находится в файле `grid/regular_grid2.cpp`. Для произвольной двумерной неструктурированной сетки `UnstructuredGrid2D` общие алгоритмы, описанные в пункте 8.2 реализованы в файле `grid/unstructured_grid2d.cpp`

### 8.3.2 Функция верхнего уровня

На верхнем уровне создаётся сетка класса

`RegularGrid2D`, которая затем используется при конструировании рабочего класса `TestPoisson2FvmWorker`.

```

168 TEST_CASE("Poisson-fvm 2D solver", "[poisson2-fvm]"){
169     std::cout << std::endl << "--- cfd24_test [poisson2-fvm] --- " << std::endl;
170
171     size_t nx = 20;
172     RegularGrid2D grid(0.0, 1.0, 0.0, 1.0, nx, nx);
173     TestPoisson2FvmWorker worker(grid);

```

Далее вызывается решатель, который возвращает величину нормы:

```

174     double nrm = worker.solve();

```

результат сохраняется в файл

```

175     worker.save_vtk("poisson2_fvm.vtk");

```

печатается количество ячеек и полученная норма

```

176     std::cout << grid.n_cells() << " " << nrm << std::endl;

```

и полученная норма проверяется с предварительно расчитанным для заданных параметров значением

```

178     CHECK(nrm == Approx(0.04371).margin(1e-4));

```

### 8.3.3 Инициализация решения

Рассмотрим рабочий класс

`TestPoisson2FvmWorker`. В его объявлении реализованы два статических метода: заданное точное решение

```

20     static double exact_solution(Point p){
21         double x = p.x();
22         double y = p.y();
23         return cos(10*x*x)*sin(10*y) + sin(10*x*x)*cos(10*x);
24     }

```

и подсчитанная правая часть, соответствующая этому решению

```

25     static double exact_rhs(Point p){
26         double x = p.x();
27         double y = p.y();
28         return (20*sin(10*x*x)+(400*x*x+100)*cos(10*x*x))*sin(10*y)

```

```
29     +(400*x*x+100)*cos(10*x)*sin(10*x*x)
30     +(400*x*sin(10*x)-20*cos(10*x))*cos(10*x*x);
31 }
```

В полях класса хранятся: ссылка на абстрактную сетку

```
37 const IGrid& _grid;
```

список внутренних граней

```
38 std::vector<size_t> _internal_faces;
```

и список граничных граней с условиями первого рода

```
45 std::vector<DirichletFace> _dirichlet_faces;
```

Каждая из этих граней описана структурой вида

```
39 struct DirichletFace{
40     size_t iface;
41     size_t icell;
42     double value;
43     Vector outer_normal;
44 };
```

в которой хранятся индекс грани, индекс ячейки, соседней с этой гранью, граничное значение функции в центре этой грани и направление внешней к области нормали.

Поле `_grid` передается в класс пользователем, в то время как списки `_internal_faces` и `_dirichlet_faces` собираются при конструировании рабочего класса.

```
53 TestPoisson2FvmWorker::TestPoisson2FvmWorker(const IGrid& grid): _grid(grid){
```

Чтобы отличить внутреннюю грань от граничной, необходимо проверить таблицу связности грань–ячейка. Если для грани одно из значений этой таблицы равно

`INVALID_INDEX`, значит с соответствующей стороны нет ячейки, а грань является граничной. Эта проверка проводится в цикле по граням

```
55 for (size_t iface=0; iface<_grid.n_faces(); ++iface){
56     size_t icell_negative = _grid.tab_face_cell(iface)[0];
57     size_t icell_positive = _grid.tab_face_cell(iface)[1];
```

Если обе ячейки валидны, значит грань внутренняя и её индекс следует добавить в список `_internal_faces`

```

58     if (icell_positive != INVALID_INDEX && icell_negative != INVALID_INDEX){
59         // internal faces list
60         _internal_faces.push_back(iface);

```

Для граничных граней создаётся и заполняется структура `DirichletFace`. Сначала заполняются те поля, которые не зависят от направления: индекс грани и граничное значение (которое берётся из точного решения):

```

63     DirichletFace dir_face;
64     dir_face.iface = iface;
65     dir_face.value = exact_solution(_grid.face_center(iface));

```

Далее, если нормаль грани направлена вовне расчётной области (ячейка по направлению нормали невалида), то индекс соседней ячейки берётся с отрицательного направления, а внешняя к области нормаль совпадает с нормалью грани

```

66     if (icell_positive == INVALID_INDEX){
67         dir_face.icell = icell_negative;
68         dir_face.outer_normal = _grid.face_normal(iface);

```

иначе индекс ячейки берётся из положительного направления, а внешняя к области нормаль противоположна нормали грани

```

69     } else {
70         dir_face.icell = icell_positive;
71         dir_face.outer_normal = -_grid.face_normal(iface);
72     }

```

### 8.3.4 Реализация решения

Решение осуществляется вызовом функции

```

78 double TestPoisson2FvmWorker::solve(){
79     // 1. build SLAE
80     CsrMatrix mat = approximate_lhs();
81     std::vector<double> rhs = approximate_rhs();
82     // 2. solve SLAE
83     AmgcMatrixSolver solver;
84     solver.set_matrix(mat);
85     solver.solve(rhs, _u);
86     // 3. compute norm2

```

```
87     return compute_norm2();  
88 }
```

в которой последовательно строятся левая и правая часть СЛАУ, вызывается решатель СЛАУ и производится сравнение с точным решением.

#### 8.3.4.1 Сборка матрицы

В функции сборки левой части СЛАУ

```
104 CsrMatrix TestPoisson2FvmWorker::approximate_lhs() const{
```

реализуются алгоритмы (8.16), (8.17) в той их части, которая касается коэффициентов матрицы. Сначала согласно (8.16) проходит цикл по внутренним ячейкам

```
107     for (size_t iface: _internal_faces){
```

Для грани берётся нормаль

```
108     Vector normal = _grid.face_normal(iface);
```

Вычисляются соседние с гранью ячейки и их центры

```
109     size_t negative_side_cell = _grid.tab_face_cell(iface)[0];  
110     size_t positive_side_cell = _grid.tab_face_cell(iface)[1];  
111     Point ci = _grid.cell_center(negative_side_cell);  
112     Point cj = _grid.cell_center(positive_side_cell);
```

находится эффективное расстояние между ними по модели (8.9)

```
113     double h = dot_product(normal, cj-ci);
```

и далее проводится заполнение матрицы найденным значением потока `coef`:

```
114     double coef = _grid.face_area(iface) / h;  
115  
116     mat.add_value(negative_side_cell, negative_side_cell, coef);  
117     mat.add_value(positive_side_cell, positive_side_cell, coef);  
118     mat.add_value(negative_side_cell, positive_side_cell, -coef);  
119     mat.add_value(positive_side_cell, negative_side_cell, -coef);  
120 }
```

Учёт условий первого рода проводится в цикле по соответствующим граням согласно алгоритму (8.17) и модели вычисления эффективного расстояния (8.12)

```

122 for (const DirichletFace& dir_face: _dirichlet_faces){
123     size_t icell = dir_face.icell;
124     size_t iface = dir_face iface;
125     Point gs = _grid.face_center(iface);
126     Point ci = _grid.cell_center(icell);
127     Vector normal = dir_face.outer_normal;
128     double h = dot_product(normal, gs-ci);
129     double coef = _grid.face_area(iface) / h;
130     mat.add_value(icell, icell, coef);
131 }
```

### 8.3.4.2 Сборка правой части

В сборке правой части

```

135 std::vector<double> TestPoisson2FvmWorker::approximate_rhs() const{
136     std::vector<double> rhs(_grid.n_cells(), 0.0);
```

сначала прогоняется алгоритм (8.20)

```

138 for (size_t icell=0; icell < _grid.n_cells(); ++icell){
139     double value = exact_rhs(_grid.cell_center(icell));
140     double volume = _grid.cell_volume(icell);
141     rhs[icell] = value * volume;
142 }
```

а затем учитываются граничные условия первого рода согласно (8.17)

```

144 for (const DirichletFace& dir_face: _dirichlet_faces){
145     size_t icell = dir_face.icell;
146     size_t iface = dir_face iface;
147     Point gs = _grid.face_center(iface);
148     Point ci = _grid.cell_center(icell);
149     Vector normal = dir_face.outer_normal;
150     double h = dot_product(normal, gs-ci);
151     double coef = _grid.face_area(iface) / h;
152     rhs[icell] += dir_face.value * coef;
153 }
```

### 8.3.4.3 Вычисление нормы отклонения от точного решения

Вычисление выражения (8.22) с использованием алгоритма (8.21) производится в функции

```
157 double TestPoisson2FvmWorker::compute_norm2() const{
158     double norm2 = 0;
159     double full_area = 0;
160     for (size_t icell=0; icell<_grid.n_cells(); ++icell){
161         double diff = _u[icell] - exact_solution(_grid.cell_center(icell));
162         norm2 += _grid.cell_volume(icell) * diff * diff;
163         full_area += _grid.cell_volume(icell);
164     }
165     return std::sqrt(norm2/full_area);
166 }
```

## 8.4 Задание для самостоятельной работы

**Получить решения для неструктурированных сеток** В папке

`test_data` корневой директории репозитория лежат скрипты построения сеток в программе `HybMesh`:

- `pebigrd.py` – pebi–сетка,
- `tetragrid.py` – сетка, состоящая из произвольных трех- и четырехугольников.

Инструкции по запуску этих скриптов смотри п. 8.4. Эти скрипты строят равномерную неструктурированную сетку в единичном квадрате и записывают её в файл `vtk`, который впоследствии можно загрузить в расчётную программу. В каждом из скриптов есть параметр `N`, означающий примерное количество ячеек в итоговой сетке. Меняя его значение можно строить сетки разного разрешения.

Необходимо с помощью этих скриптов построить сетки из  $\sim 1000$  ячеек. Далее на этих сетках решить задачу из п. 8.3 и сравнить поля решений и полученные нормы для этих сеток и равномерной структурированной сетки сходного разрешения.

Работать на основе теста `poisson2-fvm` в файле

`poisson2_fvm_solve_test.cpp`. Можно создать отдельный тест, использующий те же классы для работы.

Для загрузки построенной сетки в решатель необходимо файл с сеткой поместить в каталог `test_data` и далее загрузить её в класс `UnstructuredGrid2D`. Нижеследующий код прочитает файл `test_data/pebigrd.vtk` и создаст рабочий класс с использованием прочитанной сетки

```
std::string fn = test_directory_file("pebigrd.vtk");
UnstructuredGrid2D grid = UnstructuredGrid2D::vtk_read(fn);
TestPoisson2FvmWorker worker(grid);
```

**Получить порядок аппроксимации** Для трёх видов сеток: структурированной, ребристой и произвольной построить график сходимости аналогичный рис. 1. Для этого провести серию расчётов с различным количеством ячеек в диапазоне  $N \in [500, 10^5]$ .

Следует иметь ввиду, что на графике сходимости по оси абсцисс отложено линейное разбиение, вычисляемое как  $n = 1/h$ , где  $h$  – это характерный линейный размер ячейки. Для неструктурированных двумерных сеток этот линейный размер сетки можно вычислить через среднюю площадь ячейки  $A$  как  $h = \sqrt{A}$ , которую в свою очередь можно получить, разделив общую площадь на количество ячеек:  $A = |D|/N$ . Тогда, в случае единичного квадрата, линейное разбиение будет равно  $n = \sqrt{N}$ .

**Реализовать граничные условия второго рода** Решить ту же задачу используя граничные условия второго рода на ребристой сетке и сравнить полученные результаты с решением задачи первого рода.

Для этого нужно в рабочий класс добавить функцию вычисления нормальной производной. Эта функция будет на вход принимать точку, лежащую на границе единичного квадрата, и возвращать  $\partial u^e / \partial n$ , вычисленную прямым дифференцированием известного точного решения. Следует учитывать направление внешней нормали. Например, на вертикальной границе

$$\begin{aligned} x = 0 : \quad \frac{\partial u}{\partial n} &= -\frac{\partial u}{\partial x}, \\ x = 1 : \quad \frac{\partial u}{\partial n} &= \frac{\partial u}{\partial x}, \end{aligned}$$

Для определения конкретной границы следует анализировать входную точку. Поскольку работа ведётся в числах с плавающей точкой, сравнение следует делать с некоторым допуском:

```
struct TestPoisson2FvmWorker{
    // точная производная по x
    static double exact_dudx(Point p){
        ...
    }

    // точная производная по y
    static double exact_dudy(Point p){
        ...
    }

    static double exact_dudn(Point p){
        double x = p.x();
        double y = p.y();
        if (std::abs(x) < 1e-6){
            // левая граница. Вернуть -du/dx
            return -exact_dudx(p);
        } else if (std::abs(x-1) < 1e-6){
```

```

    // правая граница. Вернуть du/dx
    return exact_dudx(p);
} else if ...
}

```

Для реализации алгоритма (8.18) предварительно нужно собрать информацию о гранях, которую следует хранить в массиве структур (по аналогии с `DirichletFace`)

```

struct NeumannFace{
    size_t iface;
    size_t icell;
    double value;
};

std::vector<NeumannFace> _neumann_faces;

```

В отличие от условий Дирихле, структура для условий Неймана не содержит нормалей, поскольку в формулах (8.18) они не используются. Заполнять массив `_neumann_faces` следует в конструктуре `TestPoisson2FvmWorker` вместо `_dirichlet_faces`.

Алгоритм учёта условий второго рода не изменяет матрицу, поэтому обработку граничных граней следует убрать из функции `approximate_lhs`, а в функции сборки правой части `approximate_rhs` нужно реализовать формулы (8.18).

Задача в такой постановке имеет бесконечное множество решений, отличающихся на константу. Для получения однозначного ответа нужно задать точное решение в одной из ячеек. Выберем нулевую ячейку. Тогда нулевое уравнение СЛАУ нужно преобразовать к виду

$$u_0 = u^e(\mathbf{c}_0) \Rightarrow A_{0j} = \delta_{0j}, \quad b_0 = u^e(\mathbf{c}_0).$$

Для этого в функции `approximate_lhs` после окончания сборки следует вызвать метод

```
mat.set_unit_row(0);
```

а в конце `approximate_rhs` –

```
rhs[0] = exact_solution(_grid.cell_center(0));
```

# 9 Лекция 9 (11.11)

## 9.1 Вычисление нормальной производной на скошенных сетках

TODO

## 9.2 Решение системы Уравнений Навье-Стокса методом конечных объёмов

TODO

### 9.2.1 Схема SIMPLE

TODO

### 9.2.2 Вычисление градиента давления методом наименьших квадратов

TODO

### 9.2.3 Интерполяция Rhie-Chow нормальной компоненты скорости

TODO

### 9.2.4 Порядок вычисления на итерации

TODO

## 9.3 Пример расчётной программы. Течение в каверне

Представлена в файле `cavern_2d_fvm_simple_test.cpp` в тесте `[cavern2-fvm-simple]`  
TODO

## 9.4 Задание для самостоятельной работы

На основе теста `[cavern2-fvm-simple]` из файла `cavern_2d_fvm_simple_test.cpp` сравнить результат решения задачи о течении в каверне на структурированной, ребристой и произвольной сетках. Использовать количество элементов  $N \approx 2000$ . Построение и чтение сеток проводить по аналогии с п.8.4.

1. Нарисовать результат решения (давление и векторы скорости) на различных итерациях для всех использованных сеток. Отметить количество требуемых итераций до сходимости с  $\varepsilon = 10^{-2}$ .
2. На основе полученного решения построить и сохранить в выходной vtk файл дивергенцию скорости и невязку решения.
3. Построить графики сходимости невязки (печатается в консоль при итерациях) от номера итерации для трёх сеток.

**Вычисление дивергенции скорости** Распишем значения дивергенции скорости в ячейке  $E_i$  как среднеинтегральное и далее воспользуемся формулой Гаусса-Остроградского (A.9):

$$(\nabla \cdot \mathbf{u})_i \approx \frac{1}{|E_i|} \int_{E_i} \nabla \cdot \mathbf{u} d\mathbf{x} = \frac{1}{|E_i|} \sum_j \int_{\gamma_{ij}} u_n ds \approx \frac{1}{|E_i|} \sum_j (u_n)_{ij} |\gamma_{ij}|,$$

где  $\gamma_{ij}$  – все грани ячейки  $E_i$ , а  $(u_n)_{ij}$  – значение нормальной (внешней нормали по отношению к ячейке  $E_i$ ) скорости на этой грани. Расчёт по этой формуле нужно вести в цикле по граням, определяя для каждой грани пару соседних ячеек.

```

d = {0, ...}                                – массив дивергенций. Длина равна количеству ячеек
for s = 0, Nf - 1                           – цикл по всем граням
    i, j = nei_cells(s)                     – ячейки, соседние с гранью: против и по нормали
    c = (u_n)_s |γ_s|
    if (i ≠ INVALID_INDEX)                 – если слева от грани есть ячейка
        d_i += c / |E_i|                   – добавляем, так как нормаль внешняя для ячейки i
    endif
    if (j ≠ INVALID_INDEX)                 – если справа от грани есть ячейка
        d_j -= c / |E_j|                   – вычитаем, так как нормаль внутренняя для ячейки j
    endif
endfor

```

Массив нормальных скоростей к граням сетки доступен как поле рабочего класса `Cavern2DFvmSimpleWorker::_un_face`. Методы сетки `IGrid`, необходимые для программирования этой формулы:

- `IGrid::n_cells()`, `IGrid::n_faces()` – количество ячеек и граней сетки,
- `IGrid::tab_face_cell(iface)` – индексы пары соседних с гранью ячеек. Первая – против нормали, вторая – по. Для граничных граней один из возвращаемых индексов равен `INVALID_INDEX`.
- `IGrid::cell_volume(icell)` – объём ячейки  $E_i$ ,
- `IGrid::face_area(iface)` – площадь грани  $\gamma_s$ .

Вычисление массива дивергенций можно производить непосредственно в процедуре перед сохранением решения

`Cavern2DFvmSimpleWorker::save_current_fields` с тем, чтобы сразу сохранить полученный массив дивергенций в файл вызовом

```
VtkUtils::add_cell_data(d, "div", filepath);
```

**Вычисление массива невязок** происходит в функции установки текущих значений решения `set_uvp`:

```
131 std::vector<double> res_u = compute_residual_vec(_mat_uv, _rhs_u, _u);
132 std::vector<double> res_v = compute_residual_vec(_mat_uv, _rhs_v, _v);
133 for (size_t icell=0; icell < _grid.n_cells(); ++icell){
134     double coef = 1.0 / _tau / _grid.cell_volume(icell);
135     res_u[icell] *= coef;
136     res_v[icell] *= coef;
137 }
```

Для его сохранения в файл нужно либо повторить эту процедуру в функции сохранения, а лучше сделать `rhs_u`, `rhs_v` полями рабочего класса, чтобы иметь к ним доступ в методах класса. Для их сохранения нужно в процедуре сохранения `save_current_fields` вызвать

```
VtkUtils::add_cell_data(res_u, "res_u", filepath);
VtkUtils::add_cell_data(res_v, "res_v", filepath);
```

# 10 Лекция 10 (18.11)

## 10.1 Примеры сборки СЛАУ методом конечных объёмов

TODO

## 10.2 Нестационарная задача об обтекании кругового цилиндра

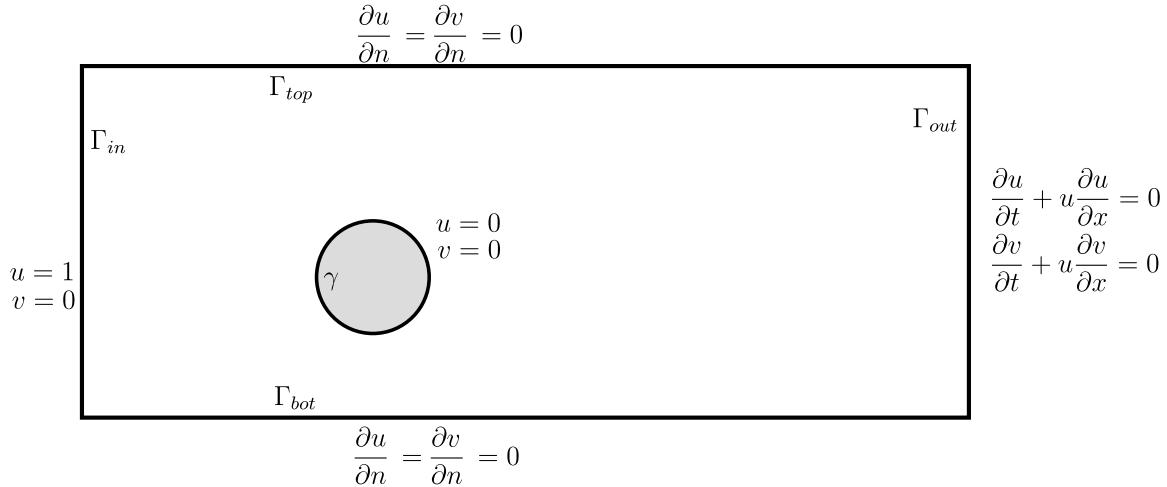


Рис. 22: К постановке задачи об обтекании цилиндра

$$\left(1 + \frac{\tau}{\Delta t}\right) u^* + \tau \nabla \cdot (\mathbf{u} u^*) - \frac{\tau}{\text{Re}} \nabla^2 u^* = u + \frac{\tau}{\Delta t} \ddot{u} - \tau \frac{\partial p}{\partial x}. \quad (10.1)$$

## 10.3 Задание для самостоятельной работы

1. Дополнить тест `[cylinder2-fvm-simple]` из файла `cylinder_fvm_simple_test.cpp` уравнением для температуры вида (7.6). Использовать граничные условия для температуры вида (пользуясь обозначениями с рис. 22):

$$\begin{aligned} \Gamma_{in} : \quad T &= 0, \\ \Gamma_{top}, \Gamma_{bot} : \quad \frac{\partial T}{\partial n} &= 0, \\ \Gamma_{out} : \quad \frac{\partial T}{\partial t} + u \frac{\partial T}{\partial x} &= 0, \\ \gamma : \quad T &= 1. \end{aligned}$$

2. Добавить процедуру вычисления интегрального числа Нуссельта Nu на временндом слое.
3. Провести расчёты для  $\text{Re} = 100$  и трёх чисел Пекле  $\text{Pe} = 30, 100, 1000$  до момента времени  $t_{end} = 60$ . Расчёты проводить в релизной сборке.
4. Сравнить полученные картины течения и графики  $\text{Nu}(t)$
5. В Paraview построить и сравнить сечения  $|\mathbf{u}|$  и  $T$  поперёк пограничного слоя цилиндра (см. п. B.3.6).

### 10.3.1 Уравнение для температуры

После дискретизации по времени уравнения (7.6) с шагом  $\Delta t$  оно примет вид

$$\hat{T} + \Delta t \nabla \cdot (\mathbf{u} \hat{T}) - \frac{\Delta t}{\text{Pe}} \nabla^2 \hat{T} = \check{T}, \quad (10.2)$$

где  $\hat{T}$  – значение температуры на следующем временному слое. Как видим, левая часть уравнения для температуры отличается от левой части уравнения переноса для скорости (10.1) только множителями перед слагаемыми. Типы граничных условий для этих уравнений также идентичны. Значит матрицу левой части для уравнения температуры можно собирать по той же процедуре, что и левую часть для  $u^*$ . В тестовом рабочем классе эта процедура имеет сигнатуру

```
CsrMatrix CylinderFvmSimpleWorker::assemble_uv_lhs(double coef_u, double coef_conv,
→ double coef_diff) const;
```

где коэффициенты `coef_u`, `coef_conv`, `coef_diff` – скалярные коэффициенты перед слагаемыми свободным, конвективным и диффузным слагаемыми левой части. Для сборки уравнения для  $u^*$  эти коэффициенты равны

$$c_u = 1 + \frac{\tau}{\Delta t}, \quad c_{conv} = \tau, \quad c_{diff} = \frac{\tau}{\text{Re}}.$$

Тогда для аппроксимации левой части уравнения для температуры нужно вызывать эту функцию с коэффициентами

$$c_u = 1, \quad c_{conv} = \Delta t, \quad c_{diff} = \frac{\Delta t}{\text{Pe}}.$$

Правая часть для внутренних точек коллокации будет аппроксимирована по стандартной конечно-объёмной процедуре:

$$b_i^T = \int_{E_i} \check{T} d\mathbf{x} = |E_i| \check{T}_i, \quad i = \overline{0, N_{cells}}.$$

Границные условия для температуры содержат только одно неоднородное условие:  $T = 1$  на  $\gamma$ . Поэтому для точек коллокации на границе  $\gamma$  в правую часть следует поставить единицу

$$b_i^T = 1, \quad i \in \gamma.$$

### 10.3.2 Коэффициент теплоотдачи

Из определения (7.15) интегральное число Нуссельта расписывается как

$$\text{Nu} = \int_{\gamma} \frac{\partial T}{\partial n} ds = \sum_{s \in \gamma} \int_{\gamma_s} \frac{\partial T}{\partial n} ds = \sum_{s \in \gamma} \left( \frac{\partial T}{\partial n} \right)_s |\gamma_s|, \quad (10.3)$$

где  $s$  – индексы всех граней, принадлежащих границе  $s$ , а  $\mathbf{n}$  – нормаль к границе, внешняя к области расчёта.

### 10.3.3 Порядок реализации

1. В рабочий класс необходимо добавить два новых поля: параметр  $\text{Pe}$  ( `_Pe` ) и массив  $T$  ( `_temperature` ). Параметр  $\text{Pe}$  необходимо привести в конструктор по аналогии с  $\text{Re}$ , а массив  $T$  инициализировать нулями внутри конструктора (длина массива равна количеству точек коллокации, вычисляемой через метод `vec_size` ).
2. Для вычисления поля температуры добавить в рабочий класс новый метод

```
std::vector<double> CylinderFvmSimpleWorker::compute_temperature() const;
```

который будет использовать текущую температуру

`_temperature` в качестве значения  $\tilde{T}$  и возвращать новую температуру, посчитанную по формуле (10.2). Этот метод будет иметь следующую реализацию

```
// 1. === Левая часть
CsrMatrix mat_temp = assemble_uv_lhs(1, _time_step, _time_step/_Pe);
// 2. === Правая часть
std::vector<double> rhs_temp(vec_size(), 0);
// 2.1 Внутренние точки коллокации
for (size_t icell=0; icell<_grid.n_cells(); ++icell){
    rhs_temp[icell] = _grid.cell_volume(icell) * _temperature[icell];
}
// 2.2 Границные условия на цилиндре
for (size_t icolloc: _boundary_info.cyl){
    rhs_temp[icolloc] = 1;
}
// 2.3 Границные условия на выходе
for (size_t icolloc: _boundary_info.output){
    rhs_temp[icolloc] = _temperature[icolloc];
}
// 3. === Решение СЛАУ
// 3.1 Массив с ответом.
// Инициализируем текущей температурой в качестве первого приближения
std::vector<double> new_temperature(_temperature)
// 3.2 Вызываем решатель СЛАУ
AmgCMatrixSolver::solve_slae(mat_temp, rhs_temp, new_temperature);
// 4. === Возвращаем ответ
return new_temperature;
```

3. Вызывать этот метод следует в процедуре

`to_next_time_step`, которая вызывается после окончания итераций на временнmм слое. Возвращаемый процедурой массив нужно присвоить полю `_temperature`, тем самым обновив значение для температуры в классе.

4. В процедуре `save_current_fields` добавить строчку

```
VtkUtils::add_cell_data(_temperature, "temperature", filepath, _grid.n_cells());
```

для сохранения температуры в файл vtk.

5. Добавить публичный метод `double CylinderFvmSimpleWorker::compute_nu() const`, который расчитывает интегральный коэффициент теплоотдачи по текущему полю `_temperature`. В реализации нужно запрограммировать формулу (10.3). Необходимые методы для вычислений:

- Все индексы точек коллокации, лежащие на цилиндре

```
std::vector<size_t> cyl_collocs = _boundary_info.cyl;
```

- Индекс грани по индексу граничной точки коллокации `icolloc`

```
size_t iface = _collocations.face_index(icolloc);
```

- Производная температуры по нормали к грани с индексом `iface`

```
double dtdn = _dfdnd_computer.compute(iface, _temperature);
```

Поскольку в формуле (10.3) используется не нормаль к грани, а внешняя нормаль к области, надо проверить, совпадают ли эти нормали. И если не совпадают (противонаправлены), то взять `dtdn` с обратным знаком:

```
if (_grid.tab_face_cell(iface)[0] == INVALID_INDEX){  
    dtdn *= -1.0;  
}
```

- Для вычисления площади грани с индексом `iface`

```
double area = _grid.face_area(iface);
```

6. Вызывать этот метод нужно в цикле по времени в функции верхнего уровня после обновления температурного поля:

```
worker.to_next_time_step();
```

Например, для печати значения Nu в консоль следует написать

```
std::cout << "time=" << time << " Nu=" << worker.compute_nu() << std::endl;
```

Можно осуществлять этот вызов не на каждой временной итерации, а с некоторым заданным шагом.

# 11 Лекция 11 (25.11)

11.1 Метод взвешенных невязок

11.2 Метод Бубнова–Галёркина

11.2.1 Степенные базисные функции

11.3 Метод конечных элементов

11.3.1 Узловые базисные функции

11.3.2 Одномерное уравнение Пуассона

11.3.2.1 Слабая интегральная постановка задачи

11.3.2.2 Линейный одномерный базис

11.3.2.3 Элементные матрицы

Матрица масс

$$M_{ij}^E = \int_E \phi_i(x) \phi_j(x) d\mathbf{x} = \int_{E^p} \phi_i(\boldsymbol{\xi}) \phi_j(\boldsymbol{\xi}) |J(\boldsymbol{\xi})| d\boldsymbol{\xi} \quad (11.1)$$

Вектор нагрузок

$$L_i^E = \int_E \phi_i(x) d\mathbf{x} = \int_{E^p} \phi_i(\boldsymbol{\xi}) |J(\boldsymbol{\xi})| d\boldsymbol{\xi} \quad (11.2)$$

Матрица жёсткости

$$S_{ij}^E = \int_E \nabla \phi_i \cdot \nabla \phi_j d\mathbf{x} = \int_{E^p} \nabla \phi_i \cdot \nabla \phi_j |J(\boldsymbol{\xi})| d\boldsymbol{\xi} \quad (11.3)$$

11.3.2.4 Сборка глобальных матриц и векторов

11.3.3 Двумерное уравнение Пуассона

11.3.3.1 Треугольный элемент. Линейный двумерный базис

Матрица масс (из (11.1)):

$$M_{ij}^E = \int_0^1 \int_0^{1-\xi} \phi_i(\xi, \eta) \phi_j(\xi, \eta) |J| d\eta d\xi = \frac{|J|}{24} \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix} \quad (11.4)$$

11.3.3.2 Пример сборки матрицы для двумерной задачи

# 12 Лекция 12 (02.12)

## 12.1 Метод конечных элементов

### 12.1.1 Четырёхугольный элемент. Билинейный двумерный базис

### 12.1.2 Численное интегрирование внутри конечного элемента

## 12.2 Разбор программной реализации МКЭ

Численное решение уравнения Пуассона с граничными условиями первого рода реализовано в файле `poisson_fem_solve_test.cpp`. Будем рассматривать решение двумерной задачи на треугольниках (тест `[poisson2-fem-tri]`). В этом тесте определяется двумерная аналитическая функция

$$f(x, y) = \cos(10x^2) \sin(10y) + \sin(10x^2) \cos(10x),$$

и формулируется уравнение Пуассона с граничными условиями первого рода, для которого эта функция является точным решением. Далее уравнение Пуассона решается численно и полученный численный результат сравнивается с точным ответом. Норма полученной ошибки печатается в консоль.

В функции верхнего уровня происходит чтение неструктурированной сетки, создание рабочего объекта, вызов решения с возвращением нормы полученной ошибки и вывод данных: сохранение двумерного поля решения в vtk-файл и печать нормы в консоль:

```
241 UnstructuredGrid2D grid = UnstructuredGrid2D::vtk_read(grid_fn, true);
242 TestPoissonLinearTriangleWorker worker(grid);
243 double nrm = worker.solve();
244 worker.save_vtk("poisson2_fem.vtk");
245 std::cout << grid.n_cells() << " " << nrm << std::endl;
```

Основная работа происходит в классе `TestPoissonLinearTriangleWorker`.

### 12.2.1 Рабочий объект

Класс `TestPoissonLinearTriangleWorker` наследуется от `ITestPoisson2FemWorker`. В этом классе сформулированы двумерные аналитические функции, служащие правой частью, точным решением и условиями первого рода уравнения Пуассона. А этот класс в свою очередь наследуется от `ITestPoissonFemWorker`, в котором и происходит решение уравнения.

```
53 double ITestPoissonFemWorker::solve(){
54     // 1. build SLAE
55     CsrMatrix mat = approximate_lhs();
56     std::vector<double> rhs = approximate_rhs();
57     // 2. Dirichlet bc
58     for (size_t ibas: dirichlet_bases()){
59         mat.set_unit_row(ibas);
```

```

60     Point p = _fem.reference_point(ibas);
61     rhs[ibas] = exact_solution(p);
62 }
63 // 3. solve SLAE
64 AmgCMatrixSolver solver({ {"precond.relax.type", "gauss_seidel"} });
65 solver.set_matrix(mat);
66 solver.solve(rhs, _u);
67 // 4. compute norm2
68 return compute_norm2();
69 }
```

Для получения решения сначала собирается левая и правая часть системы линейных уравнений, потом происходит учёт граничных условий первого рода , вызывается решатель системы уравнений и вычислитель нормы ошибки.

Функция сборки матрицы левой части реализует сборку глобальной матрицы жёсткости через набор локальных матриц

```

84 CsrMatrix ITestPoissonFemWorker::approximate_lhs() const{
85     CsrMatrix ret(_fem.stencil());
86     for (size_t ielem=0; ielem < _fem.n_elements(); ++ielem){
87         const FemElement& elem = _fem.element(ielem);
88         std::vector<double> local_stiff = elem.integrals->stiff_matrix();
89         _fem.add_to_global_matrix(ielem, local_stiff, ret.vals());
90     }
91     return ret;
92 }
```

Основой для сборки служит специальный объект `_fem` класса `FemAssembler` – сборщик. Этот объект сначала используется для задания шаблона итоговой матрицы, потом в цикле по элементам вычисляются локальные матрицы и с помощью метода этого класса

`FemAssembler::add_to_global_matrix` локальные матрицы добавляются в глобальную.

По аналогичной процедуре работает и сборка правой части `approximate_rhs`.

### 12.2.2 Конечноэлементный сборщик

Конечноэлементный сборщик `FemAssembler` – основной класс, хранящий всю информацию о текущей конечноэлементной аппроксимации: массив конечных элементов и их связность. Эта информация подаётся ему при конструировании (реализация в файле `cfd24/fem/fem_assembler.hpp` ).

```

12 FemAssembler(size_t n_bases,
13                 const std::vector<FemElement>& elements,
14                 const std::vector<std::vector<size_t>>& tab_elem_basis);
```

## Связность

`tab_elem_basis` имеет формат “элемент-глобальный базис” и определяет глобальный индекс для каждого локального базисного индекса. В рассмотренных нами узловых конечных элементах базис связан с узлом сетки. То есть эта таблица – это связность локальной и глобальной нумерации узлов сетки для каждой ячейки сетки.

Конечноэлементный сборщик создаётся в методе

`TestPoissonLinearTriangleWorker::build_fem` итогового рабочего класса (то есть сборщик специфичен для конкретной сетки и конкретного выбора типов элементов). Далее он прорасывается в конструктор базового рабочего класса.

### 12.2.3 Концепция конечного элемента

Класс конечного элемента `FemElement` определён в файле `fem/fem_element.hpp` как

```
80 struct FemElement{
81     std::shared_ptr<const IElementGeometry> geometry;
82     std::shared_ptr<const IElementBasis> basis;
83     std::shared_ptr<const IElementIntegrals> integrals;
84 };
```

Главная задача объекта этого класса – вычисление элементных матриц, которые впоследствии используются сборщиком для создания глобальных матриц. Для расчёта элементных матриц в свою очередь требуется

- Геометрия элемента, включающая в себя правило отображения элемента из физической в параметрическую область,
- Набор локальных базисных функций, заданных в параметрическом пространстве на указанной геометрии,
- Непосредственно правило интегрирования в параметрической области.

Каждый из этих трёх алгоритмов определён через интерфейсы

- `IElementGeometry`
- `IElementBasis`
- `IElementIntegrals`

Определение конечного элемента заключается в задании конкретных реализаций этих интерфейсов.

#### 12.2.3.1 Определение линейного треугольного элемента

. Так, в рассматриваемом нами тесте `"[poisson2-fem-tri]"`, используются только линейные треугольные элементы. Используется следующее определение элемента:

```

224 auto geom = std::make_shared<TriangleLinearGeometry>(p0, p1, p2);
225 auto basis = std::make_shared<TriangleLinearBasis>();
226 JacobiMatrix jac = geom->jacobi({0, 0});
227 auto integrals = std::make_shared<TriangleLinearIntegrals>(jac);
228 FemElement elem{geom, basis, integrals};

```

Здесь последовательно определяются:

- треугольная геометрия `geom` – путём задания трёх точек в физической плоскости `p0, p1, p2`,
- линейный треугольный базис `basis`,
- правила интегрирования `integrals` по параметрическому треугольнику с использованием точных формул. Эти формулы зависят только от матрицы Якоби `jac`, которая вычисляется с использованием геометрических свойств элемента (в данном случае матрица Якоби постоянная, поэтому её можно вычислять в любой точке параметрической плоскости)

Из этих трёх алгоритмов собирается конечный элемент `elem`.

### 12.2.3.2 Определение линейного элемента на отрезке

В тесте, который решает аналогичную задачу на одномерных линейных элементах `"[poisson1-fem-segm]"`, конечный элемент собирается из процедур, определённых для сегмента:

```

182 auto geom = std::make_shared<SegmentLinearGeometry>(p0, p1);
183 auto basis = std::make_shared<SegmentLinearBasis>();
184 auto integrals = std::make_shared<SegmentLinearIntegrals>(geom->jacobi({})); 
185 FemElement elem{geom, basis, integrals};

```

### 12.2.3.3 Определение билинейного элемента на четырёхугольнике

. В тесте, использующим четырёхугольные элементы

`"[poisson2-fem-quad]"` определение конечного элемента имеет вид

```

270 auto geom = std::make_shared<QuadrangleLinearGeometry>(p0, p1, p2, p3);
271 auto basis = std::make_shared<QuadrangleLinearBasis>();
272 const Quadrature* quadrature = quadrature_square_gauss2();
273 auto integrals = std::make_shared<NumericElementIntegrals>(quadrature, geom,
274   basis);
FemElement elem{geom, basis, integrals};

```

Задание геометрии и базиса здесь аналогично ранее рассмотренным элементам. А для интегрирования

`integrals` используется квадратурная формула. Расчёт интеграла по квадратурной формуле определяется классом `NumericElementIntegrals`. Для его определения нужно задать непосредственно квадратурную формулу `quadrature` (здесь используется формула, точная для полиномов второго порядка, заданных на параметрическом квадратате) а также геометрию и базис элемента. Последние нужны для вычисления подинтегральных выражений.

#### 12.2.3.4 Геометрические свойства элемента

Интерфейс `IElementGeometry`, заданный в файле `cfd24/fem/fem_element.hpp`, определяет геометрические свойства элемента:

```
13 class IElementGeometry{
14 public:
15     virtual ~IElementGeometry() = default;
16
17     virtual JacobiMatrix jacobi(Point xi) const = 0;
18     virtual Point to_physical(Point xi) const { _THROW_NOT_IMP_; }
19     virtual Point to_parametric(Point p) const { _THROW_NOT_IMP_; }
20 };
```

Для вычисления элементных матриц главным геометрическим свойством элемента является функция для вычисления матрицы Якоби (`jacobi`).

Рассмотрим реализацию этого класса для линейного треугольного элемента (в файле `cfd24/fem/elem2d/triangle_linear.hpp`)

```
11 class TriangleLinearGeometry: public IElementGeometry{
12 public:
13     TriangleLinearGeometry(Point p0, Point p1, Point p2);
14
15     JacobiMatrix jacobi(Point xi) const override;
16     Point to_physical(Point xi) const override;
17 private:
18     Point _p0, _p1, _p2;
19     JacobiMatrix _jac;
20 };
```

Для конструирования геометрии необходимо задать три точки, определяющие треугольник в физическом пространстве. Матрица Якоби в этом случае является постоянной и вычисляется один раз в конструкторе по формуле (A.31) с использованием переданных в конструктор точек. Хранится вычисленная матрица в приватном поле `_jac`.

### 12.2.3.5 Элементный базис

Интерфейс для определения локального элементного базиса имеет вид

```
33 class IElementBasis{
34 public:
35     virtual ~IElementBasis() = default;
36
37     virtual size_t size() const = 0;
38     virtual std::vector<Point> parametric_reference_points() const = 0;
39     virtual std::vector<BasisType> basis_types() const = 0;
40     virtual std::vector<double> value(Point xi) const = 0;
41     virtual std::vector<Vector> grad(Point xi) const = 0;
42     virtual std::vector<std::array<double, 6>> upper_hessian(Point xi) const {
43         ↳ _THROW_NOT_IMP_; }
43 };
```

Этот интерфейс работает только с параметрическим пространством и определяет следующие методы:

- `size` – количество базисных функций;
- `parametric_reference_points` – вектор из параметрических координат точек, приписанных к соответствующим базисам;
- `value` – значение базисных функций в заданной точке;
- `grad` – градиент (в параметрическом пространстве) базисных функций по заданным точкам.

Конкретная реализация для линейного треугольного элемента `TriangleLinearBasis` (в файле `cfd24/fem/elem2d/triangle_linear.cpp`) включает в себя линейный Лагранжев базис в двумерном пространстве согласно (A.23):

```
31 size_t TriangleLinearBasis::size() const {
32     return 3;
33 }
```

```
35 std::vector<Point> TriangleLinearBasis::parametric_reference_points() const {
36     return {Point(0, 0), Point(1, 0), Point(0, 1)};
37 }
```

```

43 std::vector<double> TriangleLinearBasis::value(Point xi_) const {
44     double xi = xi_.x();
45     double eta = xi_.y();
46     return { 1 - xi - eta, xi, eta };
47 }

```

```

49 std::vector<Vector> TriangleLinearBasis::grad(Point xi) const {
50     return { Vector(-1, -1), Vector(1, 0), Vector(0, 1) };
51 }

```

### 12.2.3.6 Калькулятор элементных матриц

Интерфейс `IElementIntegrals` предоставляет методы для вычисления интегралов

```

48 class IElementIntegrals{
49 public:
50     virtual ~IElementIntegrals() = default;
51
52     virtual std::vector<double> load_vector() const { _THROW_NOT_IMP_; }
53     virtual std::vector<double> mass_matrix() const { _THROW_NOT_IMP_; }
54     virtual std::vector<double> stiff_matrix() const { _THROW_NOT_IMP_; }
55     virtual std::vector<double> transport_matrix(
56         const std::vector<double>& vx,
57         const std::vector<double>& vy={},
58         const std::vector<double>& vz={}) const { _THROW_NOT_IMP_; }
59     virtual std::vector<double> mass_matrix_stab_supg(
60         const std::vector<double>& vx,
61         const std::vector<double>& vy={},
62         const std::vector<double>& vz={}) const { _THROW_NOT_IMP_; }
63     virtual std::vector<double> load_vector_stab_supg(
64         const std::vector<double>& vx,
65         const std::vector<double>& vy={},
66         const std::vector<double>& vz={}) const { _THROW_NOT_IMP_; }
67     virtual std::vector<double> stiff_matrix_stab_supg(
68         const std::vector<double>& vx,
69         const std::vector<double>& vy={},
70         const std::vector<double>& vz={}) const { _THROW_NOT_IMP_; }
71     virtual std::vector<double> transport_matrix_stab_supg(
72         const std::vector<double>& vx,

```

```

73     const std::vector<double>& vy={},
74     const std::vector<double>& vz={}) const { _THROW_NOT_IMP_; }
75 };

```

До сих пор были рассмотрены две элементные матрицы: матрица масс (11.1) и матрица жёсткости (11.3). Для вычисления этих матриц используются функции

`mass_matrix`, `stiff_matrix`. Возвращают эти функции локальные квадратные матрицы с числом строк, равным количеству базисов в элементе. Выходные матрицы развернуты в линейный массив. Так, для треугольного элемента с тремя базисными функциями на выходе будет массив из девяти элементов:

$$m_{00}, m_{01}, m_{02}, m_{10}, m_{11}, m_{12}, m_{20}, m_{21}, m_{22}.$$

Два подхода к вычислению элементных интегралов: точное и численное интегрирование, отражены в разных реализациях этого интерфейса.

Точные формулы интегрирования зависят от вида элемента. Так, для линейного треугольного элемента аналитическое интегрирование реализовано в классе `TriangleLinearIntegrals` (в файле `cfd24/fem/elem2d/triangle_linear.hpp`). Все интегралы будут зависеть только от матрицы Якоби, которая и передётся этому классу в конструктор. Например, вычисление матрицы масс (по (11.4)) запрограммировано в виде

```

58 std::vector<double> TriangleLinearIntegrals::mass_matrix() const {
59     double s0 = _jac.modj/12.0;
60     double s1 = _jac.modj/24.0;
61     return {s0, s1, s1,
62             s1, s0, s1,
63             s1, s1, s0};
64 }

```

Для численного интегрирования по квадратурным формулам вида (A.14) реализация этого интерфейса `NumericElementIntegrals` находится в файле `cfd24/fem/fem_numeric_integrals.hpp`. Для вычисления локальных матриц по квадратурной формуле необходимо знать квадратурные узлы и веса, а также уметь вычислять подинтегральное выражения. Конструктор этого класса имеет сигнатуру

```

11 NumericElementIntegrals(
12     const Quadrature* quad,
13     std::shared_ptr<const IElementGeometry> geom,
14     std::shared_ptr<const IElementBasis> basis);

```

Первый аргумент – это конкретная квадратура (специфичная для каждой геометрии элемента), два других аргумента описывают геометрию и набор базисов элемента (эти интерфейсы были разобраны ранее). Такой инициализации оказывается достаточно для вычисления локальных матриц для

любого конечного элемента.

### 12.3 Задание для самостоятельной работы

- Определить порядок аппроксимации двумерного уравнения Пуассона на треугольных элементах. Решение реализовано в тесте `[poisson2-fem-tri]`. Для построения треугольных сеток различного разрешения использовать скрипт `trigrid.py`.
- Решить уравнение Пуассона на сетке, содержащей как треугольники, так и четырёхугольники. Для построения таких сеток использовать скрипт `tetragrid.py`. Определить порядок аппроксимации и сравнить его с решателем на треугольной сетке из предыдущего пункта.
- С помощью расчёта на серии сгущающихся сеток исследовать влияние точности квадратурной формулы на полученную итоговую ошибку в решении уравнения Пуассона из предыдущего пункта (Здесь необходимо использовать квадратурные формулы в том числе и для треугольных элементов, где интегралы до сих пор вычислялись аналитически).

Для решения уравнения Пуассона **на сетке треугольников и четырёхугольников** необходимо создать новый тест `"poisson2-fem-tri-quad"`:

```
TEST_CASE("Poisson-fem 2D solver, triangles & quadrangles", "[poisson2-fem-tri-quad]")
```

и новый рабочий класс

```
struct TestPoissonLinearTriQuadWorker: public ITestPoisson2FemWorker{
    static FemAssembler build_fem(const IGrid& grid);
    TestPoissonLinearTriQuadWorker(const IGrid& grid):
        ITestPoisson2FemWorker(grid, build_fem(grid)){ }
};
```

Учёт наличия разных элементов следует учитывать при реализации функции `build_fem`. Если в ячейке три узла, то следует собирать треугольный элемент по алгоритму, описанному в п. 12.2.3.1, а если четыре, то согласно п. 12.2.3.3.

Для задания правил **интегрирования с использованием квадратурных формул разного порядка точности** следует использовать класс

`NumericElementIntegrals` в качестве реализации интерфейса

`IElementIntegrals`. Пример его использования указан в листинге к п. 12.2.3.3. Реализованные в программе квадратуры:

- Квадратурные формулы для параметрического треугольника, точные для полиномов первой, второй, третьей и четвёртой степеней. (то есть имеющие второй, третий, четвёртый и пятый порядки аппроксимации):

```
quadrature_triangle_gauss1();
quadrature_triangle_gauss2();
```

```
quadrature_triangle_gauss3();  
quadrature_triangle_gauss4();
```

- Квадратурные формулы для параметрического квадрата

```
quadrature_square_gauss1();  
quadrature_square_gauss2();  
quadrature_square_gauss3();  
quadrature_square_gauss4();
```

# 13 Лекция 13 (09.12)

## 13.1 Узловые элементы высокого порядка точности

## 13.2 Эрмитовы элементы

## 13.3 Разбор программной реализации МКЭ третьего порядка

## 13.4 Задание для самостоятельной работы

Провести сравнительный анализ на порядок аппроксимации решения двумерного уравнения Пуасона с граничными условиями первого рода следующих конечноэлементных схем

1. Линейные треугольные элементы (рис. 28а)
2. Квадратичные треугольные элементы (рис. 28б)
3. Кубические треугольные элементы (рис. 28в)
4. Квадратичные четырехугольные элементы (рис. 31б)
5. Неполные 8-узловые квадратичные четырёхугольные элементы (рис. 31г). В качестве базисных функций использовать полиномы вида

$$P(\xi, \eta) = A^{(00)} + A^{(10)}\xi + A^{(01)}\eta + A^{(11)}\xi\eta + A^{(20)}\xi^2 + A^{(02)}\eta^2 + A^{(21)}\xi^2\eta + A^{(12)}\xi\eta^2 + \cancel{A^{(32)}\xi^2\eta^2}$$

6. Неполные 9-узловые кубические треугольные элементы (рис. 28г). В качестве базисных функций использовать полиномы вида

$$P(\xi, \eta) = A^{(00)} + A^{(10)}\xi + A^{(01)}\eta + \cancel{A^{(11)}\xi\eta} + A^{(20)}\xi^2 + A^{(02)}\eta^2 + A^{(21)}\xi^2\eta + A^{(12)}\xi\eta^2 + A^{(30)}\xi^3 + A^{(03)}\eta^3$$

Все необходимые тесты находятся в файле `poisson_fem_solve_test.cpp`.

Для линейных треугольных элементов использовать тест `[poisson2-fem-tri]`.

Для квадратичных треугольных и квадратичных четырёхугольных элементов – `[poisson2-fem-quadratic]`.

Для кубических треугольных элементов – `[poisson2-fem-cubic]`.

Для неполных квадратичных четырёхугольных элементов в качестве основы взять тест `poisson2-fem-quadratic`. При этом необходимо изменить тип использованного четырёхугольного элемента: вместо класса `QuadrangleQuadraticBasis` использовать

`QuadrangleQuadratic8Basis`. Кроме того, поскольку центральная точка четырёхугольника больше не используется, нужно убрать последние (девятые) записи в таблицах связности `tab_elem_basis` для четырёхугольных элементов и уменьшить общее количество базисных функций до

```
size_t n_bases = grid.n_points() + grid.n_faces();
```

Для неполных кубических треугольных элементов систему базисных функций нужно вычислить самостоятельно используя алгоритм из п.А.3.1.3. На основе полученных соотношений нужно создать класс

```
class TriangleCubicNo11Basis: public IElementBasis{
public:
    size_t size() const override;
    std::vector<Point> parametric_reference_points() const override;
    std::vector<BasisType> basis_types() const override;
    std::vector<double> value(Point xi) const override;
    std::vector<Vector> grad(Point xi) const override;
};
```

и реализовать все необходимые функции:

- **size** – общее количество базисных функций. Здесь будет девять,
- **parametric\_reference\_points** – параметрические координаты девяти узловых точек (соблюдая порядок локальной индексации),
- **basis\_types** – типы базисных функций. Здесь все базисы узловые (**BasisType::Nodal**),
- **value** – значение девяти базисных функций в заданной параметрической точке. Здесь нужно подставить полученные при вычислении базисы,
- **grad** – градиенты вычисленных базисных функций. Для заполнения нужно подсчитать аналитические производные по  $\xi$  и  $\eta$  вычисленных базисов.

Реализовать этот класс можно взяв в качестве основы уже реализованный класс **TrinagleCubic9Basis** из файла **cfd24/fem/elem2d/triangle\_cubic.hpp**. После того, как базис будет реализован, его нужно использовать в тесте **poisson2-fem-cubic**:

```
auto basis = std::make_shared<TriangleCubicNo11Basis>();
```

Аналогично ранее рассмотренному неполному квадратичному элементу, следует убрать последний базис из таблицы связности

```
tab_elem_basis.push_back({
    bas0, bas1, bas2,
    bas3, bas4, bas5, bas6, bas7, bas8
});
```

и сократить общее количество базисных функций

```
size_t n_bases = grid.n_points() + 2*grid.n_faces();
```

## 14 Лекция 14 (02.03)

### 14.1 МКЭ решение для системы Навье-Стокса

TODO

# 15 Лекция 15 (09.03)

## 15.1 TVD-схемы для неструктурированных сеток

TODO

## 15.2 Задание для самостоятельной работы

TODO

# 16 Лекция 16 (23.03)

## 16.1 Условие устойчивости

$$1 + \tau(1 - \theta) \min_i \frac{l_{ii}}{m_i} \geq 0. \quad (16.1)$$

## 16.2 FEM-TVD алгоритм

TODO

$$\begin{aligned} P_i^+ &= \sum_{j \neq i} \min\{0, k_{ij}\} \min\{0, u_j - u_i\} \\ P_i^- &= \sum_{j \neq i} \min\{0, k_{ij}\} \max\{0, u_j - u_i\} \\ Q_i^+ &= \sum_{j \neq i} \max\{0, k_{ij}\} \max\{0, u_j - u_i\} \\ Q_i^- &= \sum_{j \neq i} \max\{0, k_{ij}\} \min\{0, u_j - u_i\} \end{aligned} \quad (16.2)$$

$$d_{ij} = d_{ji} = \max\{0, -k_{ij}, -k_{ji}\}. \quad (16.3)$$

$$l_{ij} = k_{ij} + d_{ij} \quad (16.4)$$

$$f_{ji}^a = \begin{cases} \min\{F(r_i^+)d_{ij}, l_{ji}\}, & \text{если } u_i \geq u_j, \\ \min\{F(r_i^-)d_{ij}, l_{ji}\}, & \text{если } u_i < u_j, \end{cases}, \quad f_{ij}^a = -f_{ji}^a, \quad f_{ii}^a = -\sum_{i \neq j} f_{ij}^a. \quad (16.5)$$

$F(r)$  – функция-ограничитель.

$$F(r) = \begin{cases} 0 & -upwind \\ 1 & -symmetric \\ \max(0, \min(r, 1)) & -minmod \\ \max(0, \min(2, |r|), \min(1, 2|r|)) & -superbee \end{cases} \quad (16.6)$$

## 16.3 Задание для самостоятельной работы

В тестовом примере `[transport2-fem-upwind-explicit]` из файла

`transport-fem-solve-test.cpp` реализовано решение двумерного уравнения переноса методом конечных объёмов с явной алгебраической противопотоковой схемой.

В этом тесте используя явную схему необходимо

1. Реализовать TVD-схему с ограничителями minmod и superbee;
2. Создать сравнительную анимацию численного решения, полученного этими тремя схемами;
3. Нарисовать норму отклонения точного решения от численного в зависимости от времени для трёх схем. В качестве нормы рассмотреть среднеквадратичное отклонение и отклонение максимума от единицы;

4. Изучить влияние шага по времени на точность результата на финальный момент времени ( $t = 0.5$ ). Для этого необходимо провести серию расчётов с шагами по времени  $\tau = k^\tau \tau_r$ , где  $\tau_r$  – это максимально допустимый условием (16.1) шаг по времени, а  $k^\tau$  – варьируемый коэффициент, изменяющийся от 0.1 до тех пор, пока решение остаётся устойчивым (может быть и большим единицы). По результатом этих расчётов нужно для трёх схем (upwind, minmod, superbee) построить графики нормы отклонения от параметра  $k^\tau$ .

Вычисление матрицы антидиффузии  $F^a$  по формулам (16.5) необходимо вести в цикле по упорядоченным связям  $ij$  (`_edges`). Для быстрого доступа к коэффициентам разреженной матрицы, связанным с текущей связью, необходимо использовать поля `..._addr` структуры `Edge`. Процедура заполнения матрицы  $F^a$  в цикле по граням должна иметь примерно такой вид

```
CsrMatrix Fa(_fem.stencil());    // заполнить шаблон нулевыми значениями
for (const Edge& e: _edges){
    // compute
    double fa_ji = ...;
    double fa_ij = -fa_ji;

    // assign
    Fa.vals()[e.ij_addr] = fa_ij;
    Fa.vals()[e.ii_addr] -= fa_ij;
    Fa.vals()[e.ji_addr] = fa_ji;
    Fa.vals()[e.jj_addr] -= fa_ji;
}
```

# 17 Лекция 17 (30.03)

## 17.1 Метод конечных элементов со стабилизацией

$$G_1 = \int_{\Omega} \nabla^2 \phi_j (\mathbf{v} \cdot \nabla \phi_i) d\mathbf{x}, \quad (17.1)$$

## 17.2 Задание для самостоятельной работы

В тестовом примере `[convdif-fem-supg]` из файла `convdif-fem-test.cpp` производится численное решение одномерного нестационарного уравнения конвекции-диффузии

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} - \varepsilon \frac{\partial^2 u}{\partial x^2} = 0$$

в области  $x \in [0, 4]$  с точным решением вида

$$u^e(x, t) = \frac{1}{\sqrt{4\pi\varepsilon(t+t_0)}} \exp\left(-\frac{(x-v_xt)^2}{4\varepsilon(t+t_0)}\right)$$

Точное решение используется для формулировки начальных ( $t = 0$ ) и граничных ( $x = 0, 4$ ) условий первого рода. Результат расчёта сохраняется в файл `convdif-supg.vtk.series`.

Задача полудискретизуется по схеме Кранка-Николсон ( $\theta = 1/2$ ) с шагом по времени, вычисляемым через число Куранта  $Cu = 0.5$ .

После дискретизации задача сводится к СЛАУ относительно неизвестного сеточного вектора  $u$

$$(A + sA^s)u = (B + sB^s)\ddot{u}.$$

Матрицы рассчитываются по процедуре Бубнова-Галёркина и SUPG стабилизаторов (матрицы с индексом  $s$ ). Множитель  $s$  – параметр SUPG-стабилизации.

$$\begin{aligned} A &= M + \tau\theta K + \varepsilon\tau\theta S, & B &= M - \tau(1-\theta)K - \varepsilon\tau(1-\theta)S, \\ A^s &= M^s + \tau\theta K^s + \varepsilon\tau\theta S^s, & B^s &= M^s - \tau(1-\theta)K^s - \varepsilon\tau(1-\theta)S^s, \end{aligned}$$

Стабилизирующие матрицы вычислялись по следующим формулам:

$$M^s = \int_{\Omega} \phi_j \mathbf{v} \cdot \nabla \phi_i d\mathbf{x}$$

$$K^s = \int_{\Omega} \mathbf{v} \cdot \nabla \phi_j \mathbf{v} \cdot \nabla \phi_i d\mathbf{x}$$

$$S^s = \int_{\Omega} \nabla^2 \phi_j \mathbf{v} \cdot \nabla \phi_i d\mathbf{x}$$

Последний интеграл из-за использования линейных базисов был равен нулю.

Сборки необходимых матриц осуществляется в процедуре `asseble_solver`. После её окончания

производится учёт граничных условий первого рода на матричном уровне.

Необходимо:

1. В одномерном тесте `[convdifff-supg]` с помощью анимированных графиков продемонстрировать наличие осцилляций при выбранных параметрах решения при отсутствии стабилизации ( $s = 0$ ), а так же эффект от SUPG-слагаемого ( $s > 0$ );
2. Нарисовать в сравнении результаты расчёта на конечный промежуток времени для различных  $s$ ;
3. Подобрать оптимальную величину параметра  $s$ , минимизирующую норму отклонения численного решения от точного;
4. Написать аналогичный тест для двумерного случая (решение в единичном квадрате). Точное решение в этом случае будет иметь вид

$$u^e(x, t) = \frac{1}{4\pi\varepsilon(t + t_0)} \exp\left(-\frac{(x - v_x t)^2 + (y - v_y t)^2}{4\varepsilon(t + t_0)}\right)$$

Использовать  $v_x = v_y = 1$ .

5. Проиллюстрировать работу программы на скошенной и структурированной двумерных сетках.

При программировании двумерного решателя нужно

- изменить значение точного решения и скорости (функции `velocity`, `nonstat_solution`);
- внести изменения в процедуру постановки граничных условий в функциях `assemble_solver`, `assemble_rhs` (сместо двух точек начала и конца одномерной области необходимо пройтись по всем граничным узлам двумерной сетки);

# 18 Лекция 18 (06.04)

## 18.1 Стабилизация методом характеристик

Введём обозначение

$$u^\theta = \theta \hat{u} + (1 - \theta) u.$$

Окончательно полудискретизованное уравнение конвекции-диффузии со стабилизирующим слагающимся примет вид

$$\frac{\hat{u} - u}{\tau} = -\mathbf{v} \cdot \nabla u + \varepsilon \nabla^2 u^\theta + \frac{\tau}{2} \nabla \cdot ((\mathbf{v} \cdot \nabla u) \mathbf{v}) - (1 - \theta) \varepsilon \mathbf{v} \cdot \nabla (\nabla^2 u). \quad (18.1)$$

### 18.1.1 Конечноэлементная процедура

Далее по стандартной процедуре взвешенных невязок домножим уравнение (18.1) на систему пробных функций  $\phi_i$  и проинтегрируем по области расчёта с применением формулы интегрирования по частям (A.10) к трём последним слагаемым правой части:

$$\begin{aligned} \int_{\Omega} \frac{\hat{u} - u}{\tau} \phi_i d\mathbf{x} &= - \int_{\Omega} (\mathbf{v} \cdot \nabla u) \phi_i d\mathbf{x} \\ &\quad + \varepsilon \left( - \int_{\Omega} \nabla u^\theta \cdot \nabla \phi_i d\mathbf{x} + \int_{\partial\Omega} \frac{\partial u^\theta}{\partial n} \phi_i ds \right) \\ &\quad + \frac{\tau}{2} \left( - \int_{\Omega} (\mathbf{v} \cdot \nabla u) (\mathbf{v} \cdot \nabla \phi_i) d\mathbf{x} + \int_{\partial\Omega} (\mathbf{v} \cdot \nabla u) v_n \phi_i ds \right) \\ &\quad + \varepsilon (1 - \theta) \left( \int_{\Omega} \nabla^2 u \nabla \cdot (\phi_i \mathbf{v}) d\mathbf{x} - \int_{\partial\Omega} v_n \phi_i \nabla^2 u ds \right) \end{aligned}$$

Согласно подходу Бубнова-Галёркина разложим искомую функцию на систему базисных функций, равную системе пробных функций

$$u(\mathbf{x}) = \sum_i u_i \phi_i(\mathbf{x})$$

Получим матричное выражение для перехода на следующий временной слой

$$\begin{aligned} \mathbf{L}\hat{u} &= \mathbf{R}u \\ \mathbf{L} &= \mathbf{M} + \tau \varepsilon \theta (\mathbf{S} - \mathbf{B}^S) \\ \mathbf{R} &= \mathbf{M} - \tau \mathbf{K} - \tau (1 - \theta) \varepsilon (\mathbf{S} - \mathbf{B}^S) + \frac{\tau^2}{2} (-\mathbf{K}^S + \mathbf{B}^{KS}) + \varepsilon \tau (1 - \theta) (\mathbf{G}_2 - \mathbf{B}^{G_2}) \end{aligned} \quad (18.2)$$

где введена новая матрица  $\mathbf{G}_2$ :

$$\mathbf{G}_2 = \int_{\Omega} \nabla^2 \phi_j \nabla \cdot (\mathbf{v} \phi_i) d\mathbf{x},$$

которая равна ранее рассмотренной матрице  $\mathbf{G}_1$  (17.1) в случае бездивергентного поля скорости:  $\nabla \cdot \mathbf{v} = 0$  и точно также обращается в ноль при использовании линейных конечных элементов.

## 18.2 Задание для самостоятельной работы

В тестовом примере [convdiff-fem-cg] из файла

convdiff-fem-test.cpp производится численного решение той же одномерной задачи, которая рассматривалась ранее в п. 17.2. Решение производится методом конечных элементов со стабилизацией методом характеристик. Задача полудискретизуется по схеме Кранка-Николсон ( $\theta = 1/2$ ) с шагом по времени, вычисленным через число Куранта  $Cu = 0.5$ .

Необходимо:

1. С помощью анимированных графиков сравнить полученное численное решение одномерной задачи ( [convdiff-fem-cg] ) с точным решением, а также с решением, полученным с помощью SUPG-стабилизации;
2. Сделать расчёты одномерной задачи с различными шагами по времени  $\tau$ . Продемонстрировать отличия в полученных решениях на анимированных графиках. Нарисовать зависимость нормы отклонения численного решения от точного на финальный момент времени

$$n_2 = \|u - u^e\|_2$$

от шага по времени  $\tau$ .

3. Написать аналогичный тест для двумерного случая (решение в единичном квадрате) и неконстантного поля скорости:  $\mathbf{v} = (-y + 0.5, x - 0.5)$ . Точное решение в этом случае будет иметь вид

$$\bar{x} = 0.5 + (x_0 - 0.5) \cos(t) - (y_0 - 0.5) \sin(t) \quad \text{– текущее положение пика}$$

$$\bar{y} = 0.5 + (x_0 - 0.5) \sin(t) + (y_0 - 0.5) \cos(t)$$

$$r^2 = (x - \bar{x})^2 + (y - \bar{y})^2 \quad \text{– расстояние от пика}$$

$$u^e = \frac{1}{4\pi\varepsilon(t + t_0)} \exp\left(-\frac{r^2}{4\varepsilon(t + t_0)}\right) \quad \text{– решение}$$

Использовать следующие параметры:

- начальное положение пика  $x_0 = 0.8, y_0 = 0.5$ ,
- сдвиг по времени  $t_0 = 0.3$ ,
- коэффициент диффузии  $\varepsilon = 10^{-3}$ ,
- временной интервал  $t \in [0, 2\pi]$ .

Анимировать решение, полученное на структурированной и скошенной сетках с количеством ячеек  $N \approx 5000$ .

4. Для структурированной сетки нарисовать зависимость  $n_2$  на момент  $t = 2\pi$  в зависимости от шага сетки  $h$  для фиксированного  $Cu = 0.5$  (в качестве характерной скорости при вычислении числа Куранта использовать единицу).
5. Нарисовать зависимость  $n_2$  от шага по времени  $\tau$  для фиксированной двумерной сетки  $N \approx 5000$ . Использовать структурированную и скошенную сетки.

## **A   Формулы и обозначения**

## A.1 Векторы

### A.1.1 Обозначение

Геометрические вектора обозначаются жирным шрифтом  $\mathbf{v}$ . Скалярные координаты вектора – через нижний индекс с обозначением оси координат:  $(v_x, v_y, v_z)$ . Если вектор  $\mathbf{u}$  – вектор скорости, то его декартовые координаты имеют специальное обозначение  $\mathbf{u} = (u, v, w)$ . Единичные вектора, соответствующие осям координат, обозначаются знаком  $\hat{\cdot}$ :  $\hat{\mathbf{x}}$ ,  $\hat{\mathbf{y}}$ ,  $\hat{\mathbf{z}}$ . Координатные векторы обозначаются по символу первой оси. Например,  $\mathbf{x} = (x, y, z)$  или  $\xi = (\xi, \eta, \zeta)$ .

Операции в векторами имеют следующее обозначение (расписывая в декартовых координатах):

- Умножение на скалярную функцию

$$f\mathbf{u} = (fu_x)\hat{\mathbf{x}} + (fu_y)\hat{\mathbf{y}} + (fu_z)\hat{\mathbf{z}}; \quad (\text{A.1})$$

- Скалярное произведение

$$\mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y + u_z v_z; \quad (\text{A.2})$$

- Векторное произведение

$$\mathbf{u} \times \mathbf{v} = \begin{vmatrix} \hat{\mathbf{x}} & \hat{\mathbf{y}} & \hat{\mathbf{z}} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix} = (u_y v_z - u_z v_y) \hat{\mathbf{x}} - (u_x v_z - u_z v_x) \hat{\mathbf{y}} + (u_x v_y - u_y v_x) \hat{\mathbf{z}}. \quad (\text{A.3})$$

В двумерном случае можно считать, что  $u_z = v_z = 0$ . Тогда результатом векторного произведения согласно (A.3) будет вектор, направленный перпендикулярно плоскости  $xy$ :

$$\mathbf{u} \times \mathbf{v} = (u_x v_y - u_y v_x) \hat{\mathbf{z}}.$$

При работе с двумерными задачами, где ось  $\mathbf{z}$  отсутствует, обычно результатом векторного произведения считают скаляр

$$2D : \mathbf{u} \times \mathbf{v} = u_x v_y - u_y v_x. \quad (\text{A.4})$$

Геометрический смысл этого скаляра: площадь параллелограмма, построенного на векторах  $\mathbf{u}$  и  $\mathbf{v}$ .

### A.1.2 Набла–нотация

Символ  $\nabla$  – есть псевдовектор, который выражает покоординатные производные. Для декартовой системы координат  $(x, y, z)$  он запишется в виде

$$\nabla = \left( \frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right).$$

В радиальной  $(r, \phi, z)$ :

$$\nabla = \left( \frac{\partial}{\partial r}, \frac{1}{r} \frac{\partial}{\partial \phi}, \frac{\partial}{\partial z} \right).$$

В цилиндрической  $(r, \theta, \phi)$ :

$$\nabla = \left( \frac{\partial}{\partial r}, \frac{1}{r} \frac{\partial}{\partial \theta}, \frac{1}{r \sin \theta} \frac{\partial}{\partial \phi} \right).$$

Удобство записи дифференциальных выражений с использованием  $\nabla$  заключается в независимости записи от вида системы координат. Но если требуется обозначить производную по конкретной координате, то, по аналогии с обычными векторами, это делается через нижний индекс:

$$\nabla_n f = \frac{\partial f}{\partial n}.$$

Для этого символа справедливы все векторные операции, описанные ранее. Так, применение  $\nabla$  к скалярной функции аналогично умножению вектора на скаляр (A.1) (здесь и далее приводятся покоординатные выражения для декартовой системы):

$$\nabla f = (\nabla_x f, \nabla_y f, \nabla_z f) = \frac{\partial f}{\partial x} \hat{\mathbf{x}} + \frac{\partial f}{\partial y} \hat{\mathbf{y}} + \frac{\partial f}{\partial z} \hat{\mathbf{z}}. \quad (\text{A.5})$$

Результатом этой операции является вектор.

Скалярное умножение  $\nabla$  на вектор  $\mathbf{v}$  по аналогии с (A.2) – есть дивергенция:

$$\nabla \cdot \mathbf{v} = \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} \quad (\text{A.6})$$

результат которой – скалярная функция.

Двойное применение  $\nabla$  к скалярной функции – это оператор Лапласа:

$$\nabla \cdot \nabla f = \nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2} \quad (\text{A.7})$$

Ротор – аналог векторного умножения (A.3):

$$\nabla \times \mathbf{v} = \begin{vmatrix} \hat{\mathbf{x}} & \hat{\mathbf{y}} & \hat{\mathbf{z}} \\ \nabla_x & \nabla_y & \nabla_z \\ v_x & v_y & v_z \end{vmatrix} = \left( \frac{\partial v_z}{\partial y} - \frac{\partial v_y}{\partial z} \right) \hat{\mathbf{x}} - \left( \frac{\partial v_z}{\partial x} - \frac{\partial v_x}{\partial z} \right) \hat{\mathbf{y}} + \left( \frac{\partial v_y}{\partial x} - \frac{\partial v_x}{\partial y} \right) \hat{\mathbf{z}}. \quad (\text{A.8})$$

## A.2 Интегрирование

### A.2.1 Формула Гаусса–Остроградского

Формула Гаусса–Остроградского, связывающая интегрирование по объёму  $E$  с интегрированием по границе этого объёма  $\Gamma$ , для векторного поля  $\mathbf{v}$  имеет вид

$$\int_E \nabla \cdot \mathbf{v} d\mathbf{x} = \int_{\Gamma} v_n ds, \quad (\text{A.9})$$

где  $\mathbf{n}$  – внешняя по отношению к области  $E$  нормаль. Смысл этой формулы можно проиллюстрировать на одномерном примере. Пусть одномерное векторное поле  $v_x = f(x)$  на отрезке  $E = [a, b]$  задано функцией, представленной на рис. 23. Разобьем область на  $N = 3$  равномерных подобластей

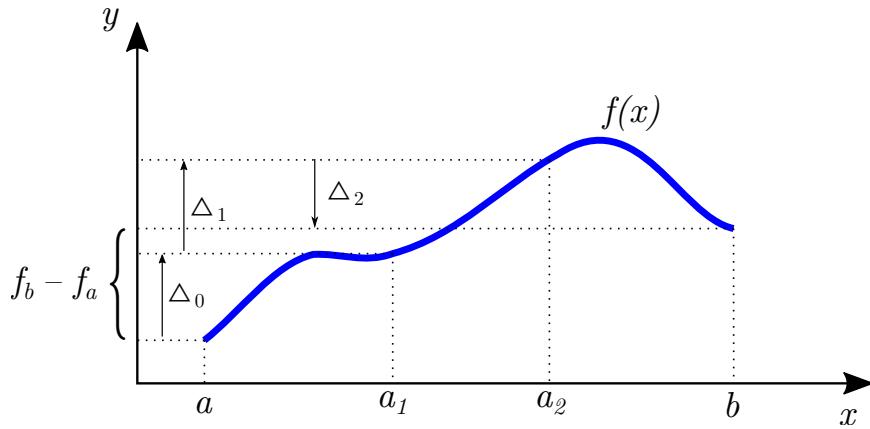


Рис. 23: Формула Гаусса–Остроградского в одномерном случае

длины  $h$ . Тогда расписывая интеграл как сумму, а производную через конечную разность, получим

$$\int_E \frac{\partial f}{\partial x} dx \approx \sum_{i=0}^2 h \left( \frac{\partial f}{\partial x} \right)_{i+\frac{1}{2}} \approx \sum_{i=0}^2 (f_{i+1} - f_i) = \Delta_0 + \Delta_1 + \Delta_2 = f_b - f_a.$$

Очевидно что, при устремлении  $N \rightarrow \infty$  правая часть предыдущего выражения не изменится. То есть, сумма всех изменений функции в области есть изменение функции по её границам:

$$\int_a^b \frac{\partial f}{\partial x} dx = f(b) - f(a).$$

А формула (A.9) – есть многомерное обобщение этого выражения.

### A.2.2 Интегрирование по частям

Подставив в (A.9)  $\mathbf{v} = f\mathbf{u}$ , где  $f$  – некоторая скалярная функция, и расписав дивергенцию в виде

$$\nabla \cdot (f\mathbf{u}) = f\nabla \cdot \mathbf{u} + \mathbf{u} \cdot \nabla f$$

получим формулу интегрирования по частям

$$\int_E \mathbf{u} \cdot \nabla f \, d\mathbf{x} = \int_{\Gamma} f u_n \, ds - \int_E f \nabla \cdot \mathbf{u} \, d\mathbf{x} \quad (\text{A.10})$$

Распишем некоторые частные случаи для формулы (A.10). Для  $\mathbf{u} = (n_x, 0, 0)$  получим

$$\int_E \frac{\partial f}{\partial x} \, d\mathbf{x} = \int_{\Gamma} f \cos(\hat{\mathbf{n}}, \hat{\mathbf{x}}) \, ds \quad (\text{A.11})$$

При  $\mathbf{u} = \nabla g$

$$\int_E f (\nabla^2 g) \, d\mathbf{x} = \int_{\Gamma} f \frac{\partial g}{\partial n} \, ds - \int_E \nabla f \cdot \nabla g \, d\mathbf{x} \quad (\text{A.12})$$

При  $f = 1$  и  $\mathbf{u} = \nabla g$

$$\int_E \nabla^2 g \, d\mathbf{x} = \int_{\Gamma} \frac{\partial g}{\partial n} \, ds \quad (\text{A.13})$$

### A.2.3 Численное интегрирование в заданной области

Квадратурная формула

$$\int_E f(\mathbf{x}) \, d\mathbf{x} = \sum_{i=0}^{N-1} w_i f(\mathbf{x}_i) \quad (\text{A.14})$$

Она определяется заданием узлов интегрирования  $\mathbf{x}_i$  и соответствующих весов  $w_i$ .

## A.3 Интерполяционные полиномы

### A.3.1 Многочлен Лагранжа

#### A.3.1.1 Узловые базисные функции

Рассмотрим функцию  $f(\xi)$ , заданную в области  $D$ . Внутри этой области зададим  $N$  узловых точек  $\xi_i, i = \overline{0, N-1}$ . Приближение функции  $f$  будем искать в виде

$$f(\xi) \approx \sum_{i=0}^{N-1} f_i \phi_i(\xi), \quad (\text{A.15})$$

где  $f_i = f(\xi_i)$ ,  $\phi_i$  – узловая базисная функция. Потребуем, чтобы это выражение выполнялось точно для всех заданных узлов интерполяции  $\xi = \xi_i$ . Тогда, исходя из определения (A.15), запишем условие на узловую базисную функцию

$$\phi_i(\xi_j) = \begin{cases} 1, & i = j, \\ 0, & i \neq j. \end{cases} \quad (\text{A.16})$$

Дополнительно потребуем, чтобы формула (A.15) была точной для постоянных функций

$$f(\xi) = \text{const} \Rightarrow f_i = \text{const}.$$

Тогда для любого  $\xi$  должно выполняться условие

$$\sum_{i=0}^{N-1} \phi_i(\xi) = 1, \quad \xi \in D. \quad (\text{A.17})$$

Задача построения интерполяционной функции состоит в конкретном определении узловых базисов  $\phi_i(\xi)$  по заданному набору узловых точек  $\xi_i$  и значениям функции в них  $f_i$ . Будем искать базисы в виде многочленов вида

$$\phi_i(\xi) = \sum_a A_i^{(a)} \xi^a = A_i^{(0)} + A_i^{(1)} \xi + A_i^{(2)} \xi^2 + \dots, \quad i = \overline{0, N-1}. \quad (\text{A.18})$$

Определять коэффициенты  $A_i^{(a)}$  будем из условий (A.16), которое даёт  $N$  линейных уравнений относительно неизвестных  $A_i^{(a)}$  для каждого  $i = \overline{0, N-1}$ . Таким образом, в выражениях (A.18) должно быть ровно  $N$  слагаемых. Будем использовать последовательный набор степеней:  $a = \overline{0, N-1}$ . Выпишем систему линейных уравнений для 0-ой базисной функции

$$\begin{aligned} \phi_0(\xi_0) &= A_0^{(0)} + A_0^{(1)} \xi_0 + A_0^{(2)} \xi_0^2 + A_0^{(3)} \xi_0^3 + \dots = 1, \\ \phi_0(\xi_1) &= A_0^{(0)} + A_0^{(1)} \xi_1 + A_0^{(2)} \xi_1^2 + A_0^{(3)} \xi_1^3 + \dots = 0, \\ \phi_0(\xi_2) &= A_0^{(0)} + A_0^{(1)} \xi_2 + A_0^{(2)} \xi_2^2 + A_0^{(3)} \xi_2^3 + \dots = 0, \\ &\dots \end{aligned}$$

или в матричном виде

$$\begin{pmatrix} 1 & \xi_0 & \xi_0^2 & \xi_0^3 & \dots \\ 1 & \xi_1 & \xi_1^2 & \xi_1^3 & \dots \\ 1 & \xi_2 & \xi_2^2 & \xi_2^3 & \dots \\ 1 & \xi_3 & \xi_3^2 & \xi_3^3 & \dots \\ \dots & & & & \end{pmatrix} \begin{pmatrix} A_0^{(0)} \\ A_0^{(1)} \\ A_0^{(2)} \\ A_0^{(3)} \\ \vdots \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ \vdots \end{pmatrix}$$

Записывая аналогичные выражения для остальных базисных функций, получим систему матричных уравнений вида  $CA = E$ :

$$\begin{pmatrix} 1 & \xi_0 & \xi_0^2 & \xi_0^3 & \dots \\ 1 & \xi_1 & \xi_1^2 & \xi_1^3 & \dots \\ 1 & \xi_2 & \xi_2^2 & \xi_2^3 & \dots \\ 1 & \xi_3 & \xi_3^2 & \xi_3^3 & \dots \\ \dots & & & & \end{pmatrix} \begin{pmatrix} A_0^{(0)} & A_1^{(0)} & A_2^{(0)} & A_3^{(0)} & \dots \\ A_0^{(1)} & A_1^{(1)} & A_2^{(1)} & A_3^{(1)} & \\ A_0^{(2)} & A_1^{(2)} & A_2^{(2)} & A_3^{(2)} & \\ A_0^{(3)} & A_1^{(3)} & A_2^{(3)} & A_3^{(3)} & \\ \vdots & & & & \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & \dots \\ 0 & 1 & 0 & 0 & \\ 0 & 0 & 1 & 0 & \\ 0 & 0 & 0 & 1 & \\ \vdots & & & & \end{pmatrix}$$

Отсюда матрица неизвестных коэффициентов  $A$  определится как

$$A = C^{-1} = \begin{pmatrix} 1 & \xi_0 & \xi_0^2 & \xi_0^3 & \dots \\ 1 & \xi_1 & \xi_1^2 & \xi_1^3 & \dots \\ 1 & \xi_2 & \xi_2^2 & \xi_2^3 & \dots \\ 1 & \xi_3 & \xi_3^2 & \xi_3^3 & \dots \\ \dots & & & & \end{pmatrix}^{-1}. \quad (\text{A.19})$$

Подставляя полином (A.18) в условие согласованности (A.17), получим требование

$$\sum_{i=0}^{N-1} A_i^{(a)} = \begin{cases} 1, & a = 0, \\ 0, & a = \overline{1, N-1}. \end{cases}$$

То есть сумма всех свободных членов в интерполяционных полиномах должна быть равна единице, а сумма коэффициентов при остальных степенях – нулю. Можно показать, что это свойство выполняется для любой матрицы  $A = C^{-1}$ , в случае, если первый столбец матрицы  $C$  состоит из единиц. То есть условие согласованности требует наличие свободного члена с интерполяционном полиноме.

### A.3.1.2 Интерполяция в параметрическом отрезке

Будем рассматривать область интерполяции  $D = [-1, 1]$ . В качестве первых двух узлов интерполяции возьмем границы области:  $\xi_0 = -1$ ,  $\xi_1 = 1$ .

**Линейный базис** Будем искать интерполяционный базис в виде

$$\phi_i(\xi) = A_i^{(0)} + A_i^{(1)}\xi.$$

на основе двух условий:

$$\phi_i(-1) = A_i^{(0)} - A_i^{(1)} = \delta_{0i}, \quad \phi_i(1) = A_i^{(0)} + A_i^{(1)}\delta_{1i}.$$

Составим матрицу  $C$ , записав эти условия в матричном виде

$$C = \left( \begin{array}{c|cc} & A^{(0)} & A^{(1)} \\ \hline \phi(-1) & 1 & -1 \\ \phi(1) & 1 & 1 \end{array} \right)$$

и, согласно (A.19), найдём матрицу коэффициентов

$$A = \begin{pmatrix} A_0^{(0)} & A_1^{(0)} \\ A_0^{(1)} & A_1^{(1)} \end{pmatrix} = C^{-1} = \left( \begin{array}{c|cc} & \phi_0 & \phi_1 \\ \hline 1 & \frac{1}{2} & \frac{1}{2} \\ \xi & -\frac{1}{2} & \frac{1}{2} \end{array} \right).$$

Отсюда узловые базисные функции примут вид (рис. 24)

$$\begin{aligned} \phi_0(\xi) &= \frac{1-\xi}{2}, \\ \phi_1(\xi) &= \frac{1+\xi}{2}. \end{aligned} \tag{A.20}$$

Окончательно интерполяционная функция из определения (A.15) примет вид

$$f(\xi) \approx \frac{1-\xi}{2}f(-1) + \frac{1+\xi}{2}f(1).$$

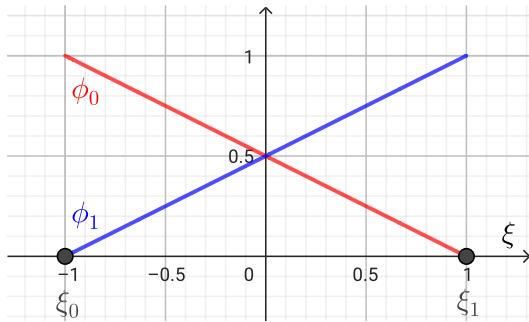


Рис. 24: Линейный базис в параметрическом отрезке

**Квадратичный базис** Будем искать интерполяционный базис в виде

$$\phi_i(\xi) = A_i^{(0)} + A_i^{(1)}\xi + A_i^{(2)}\xi^2.$$

По сравнению с линейным случаем, в форму базиса добавился ещё один неизвестный коэффициент  $A_i^{(2)}$ , поэтому в набор условий (A.16) требуется ещё одно уравнение (ещё одна узловая точка). Поме-

стим её в центр параметрического сегмента  $\xi_2 = 0$ . Далее будем действовать по аналогии с линейным случаем:

$$C = \left( \begin{array}{c|ccc} & A^{(0)} & A^{(1)} & A^{(2)} \\ \hline \phi(-1) & 1 & -1 & 1 \\ \phi(1) & 1 & 1 & 1 \\ \phi(0) & 1 & 0 & 0 \end{array} \right) \Rightarrow A = C^{-1} = \left( \begin{array}{c|ccc} & \phi_0 & \phi_1 & \phi_2 \\ \hline 1 & 0 & 0 & 1 \\ \xi & -\frac{1}{2} & \frac{1}{2} & 0 \\ \xi^2 & \frac{1}{2} & \frac{1}{2} & -1 \end{array} \right).$$

Узловые базисные функции для квадратичной интерполяции примут вид (рис. 25)

$$\begin{aligned} \phi_0(\xi) &= \frac{\xi^2 - \xi}{2}, \\ \phi_1(\xi) &= \frac{\xi^2 + \xi}{2}, \\ \phi_2(\xi) &= 1 - \xi^2. \end{aligned} \quad (\text{A.21})$$

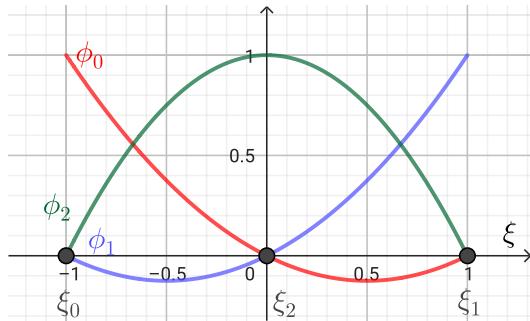


Рис. 25: Квадратичный базис в параметрическом отрезке

**Кубический базис** Интерполяционный базис будет иметь вид

$$\phi_i(\xi) = A_i^{(0)} + A_i^{(1)}\xi + A_i^{(2)}\xi^2 + A_i^{(3)}\xi^3.$$

Для нахождения четырёх коэффициентов нам понадобится четыре узла интерполяции. Две из них – это границы параметрического отрезка. Остальные две разместим так, чтобы разбить отрезок на равные интервалы:  $\xi_2 = -\frac{1}{3}$ ,  $\xi_3 = \frac{1}{3}$ . Далее вычислим матрицу коэффициентов:

$$C = \left( \begin{array}{c|cccc} & A^{(0)} & A^{(1)} & A^{(2)} & A^{(3)} \\ \hline \phi(-1) & 1 & -1 & 1 & -1 \\ \phi(1) & 1 & 1 & 1 & 1 \\ \phi(-\frac{1}{3}) & 1 & -\frac{1}{3} & \frac{1}{9} & -\frac{1}{27} \\ \phi(\frac{1}{3}) & 1 & \frac{1}{3} & \frac{1}{9} & \frac{1}{27} \end{array} \right) \Rightarrow A = C^{-1} = \frac{1}{16} \left( \begin{array}{c|cccc} & \phi_0 & \phi_1 & \phi_2 & \phi_3 \\ \hline 1 & -1 & -1 & 9 & 9 \\ \xi & 1 & -1 & -27 & 27 \\ \xi^2 & 9 & 9 & -9 & -9 \\ \xi^3 & -9 & 9 & 27 & -27 \end{array} \right)$$

Узловые базисные функции для квадратичной интерполяции примут вид (рис. 26)

$$\begin{aligned}\phi_0(\xi) &= \frac{1}{16} (-1 + \xi + 9\xi^2 - 9\xi^3), \\ \phi_1(\xi) &= \frac{1}{16} (-1 - \xi + 9\xi^2 + 9\xi^3), \\ \phi_2(\xi) &= \frac{1}{16} (9 - 27\xi - 9\xi^2 + 27\xi^3), \\ \phi_3(\xi) &= \frac{1}{16} (9 + 27\xi - 9\xi^2 - 27\xi^3),\end{aligned}\tag{A.22}$$

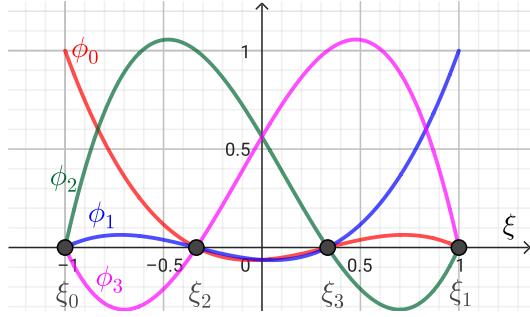


Рис. 26: Кубический базис в параметрическом отрезке

На рис. 27 представлено сравнение результатов аппроксимации функции  $f(x) = -x + \sin(2x + 1)$  линейным, квадратичным и кубическим базисом. Видно, что все интерполяционные приближения точно попадают в функцию в своих узлах интерполяции, а между узлами происходит аппроксимация полиномом соответствующей степени.

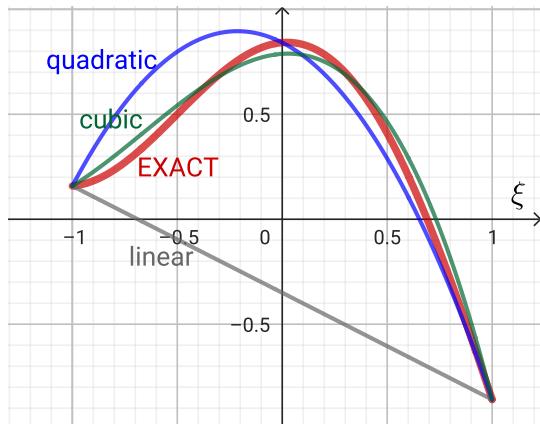


Рис. 27: Результат интерполяции

### A.3.1.3 Интерполяция в параметрическом треугольнике

Теперь рассмотрим двумерное обобщение формулы

#### Линейный базис

$$\phi_i(\xi, \eta) = A_i^{(00)} + A_i^{(10)}\xi + A_i^{(01)}\eta.$$

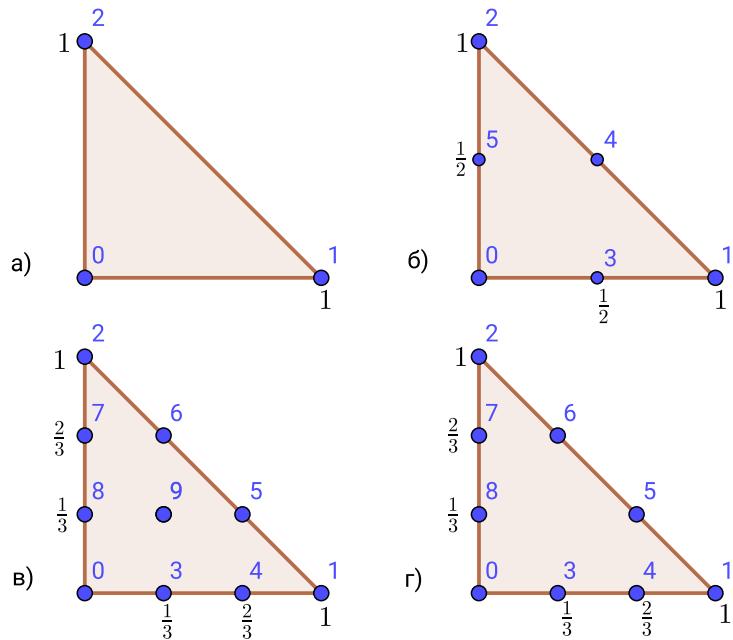


Рис. 28: Расположение узловых точек в параметрическом треугольнике. а) линейный базис, б) квадратичный базис, в) кубический базис, г) неполный кубический базис

$$C = \left( \begin{array}{c|ccc} & A^{(00)} & A^{(10)} & A^{(01)} \\ \hline \phi(0,0) & 1 & 0 & 0 \\ \phi(1,0) & 1 & 1 & 0 \\ \phi(0,1) & 1 & 0 & 1 \end{array} \right) \Rightarrow A = C^{-1} = \left( \begin{array}{c|ccc} & \phi_0 & \phi_1 & \phi_2 \\ \hline 1 & 1 & 0 & 0 \\ \xi & -1 & 1 & 0 \\ \eta & -1 & 0 & 1 \end{array} \right)$$

$$\begin{aligned} \phi_0(\xi, \eta) &= 1 - \xi - \eta, \\ \phi_1(\xi, \eta) &= \xi, \\ \phi_2(\xi, \eta) &= \eta, \end{aligned} \tag{A.23}$$

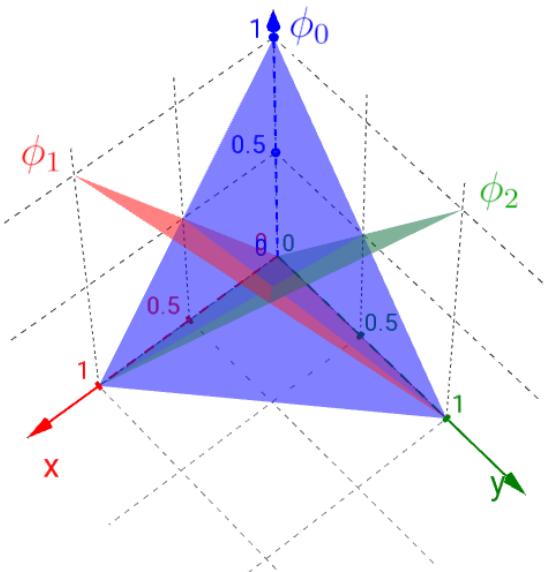


Рис. 29: Линейный базис в параметрическом треугольнике

## Квадратичный базис

$$\phi_i(\xi, \eta) = A_i^{(00)} + A_i^{(10)}\xi + A_i^{(01)}\eta + A_i^{(11)}\xi\eta + A_i^{(20)}\xi^2 + A_i^{(02)}\eta^2.$$

$$C = \left( \begin{array}{c|cccccc} & A^{(00)} & A^{(10)} & A^{(01)} & A^{(11)} & A^{(20)} & A^{(02)} \\ \hline \phi(0,0) & 1 & 0 & 0 & 0 & 0 & 0 \\ \phi(1,0) & 1 & 1 & 0 & 0 & 1 & 0 \\ \phi(0,1) & 1 & 0 & 1 & 0 & 0 & 1 \\ \phi(\frac{1}{2},0) & 1 & \frac{1}{2} & 0 & 0 & \frac{1}{4} & 0 \\ \phi(\frac{1}{2},\frac{1}{2}) & 1 & \frac{1}{2} & \frac{1}{2} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \phi(0,\frac{1}{2}) & 1 & 0 & \frac{1}{2} & 0 & 0 & \frac{1}{4} \end{array} \right) \Rightarrow A = \left( \begin{array}{c|cccccc} & \phi_0 & \phi_1 & \phi_2 & \phi_3 & \phi_4 & \phi_5 \\ \hline 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ \xi & -3 & -1 & 0 & 4 & 0 & 0 \\ \eta & -3 & 0 & -1 & 0 & 0 & 4 \\ \xi\eta & 4 & 0 & 0 & -4 & 4 & -4 \\ \xi^2 & 2 & 2 & 0 & -4 & 0 & 0 \\ \eta^2 & 2 & 0 & 2 & 0 & 0 & -4 \end{array} \right)$$

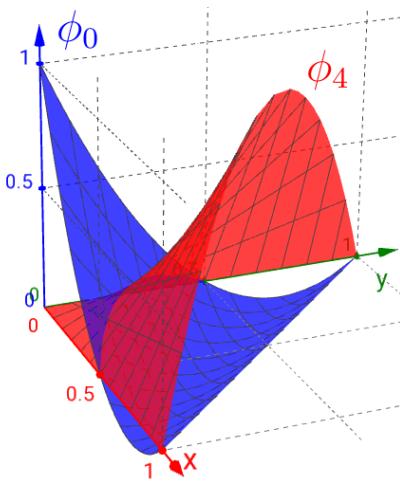


Рис. 30: Квадратичные функции  $\phi_0, \phi_4$  в параметрическом треугольнике

## Кубический базис TODO

### Неполный кубический базис TODO

#### A.3.1.4 Интерполяция в параметрическом квадрате

##### Билинейный базис

$$\phi_i = A_i^{00} + A_i^{10}\xi + A_i^{01}\eta + A_i^{11}\xi\eta.$$

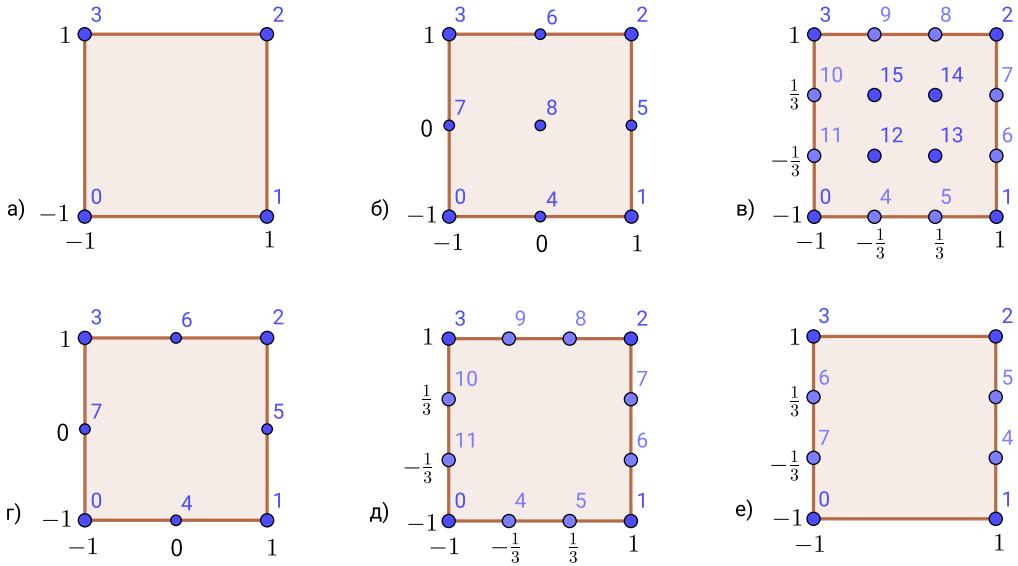


Рис. 31: Расположение узловых точек в параметрическом квадрате

$$C = \left( \begin{array}{c|cccc} & A^{(00)} & A^{(10)} & A^{(01)} & A^{(11)} \\ \hline \phi(-1, -1) & 1 & -1 & -1 & 1 \\ \phi(1, -1) & 1 & 1 & -1 & -1 \\ \phi(1, 1) & 1 & 1 & 1 & 1 \\ \phi(-1, 1) & 1 & -1 & 1 & -1 \end{array} \right) \Rightarrow A = C^{-1} = \frac{1}{4} \left( \begin{array}{c|ccccc} & \phi_0 & \phi_1 & \phi_2 & \phi_3 \\ \hline 1 & 1 & 1 & 1 & 1 \\ \xi & -1 & 1 & 1 & -1 \\ \eta & -1 & -1 & 1 & 1 \\ \xi\eta & 1 & -1 & 1 & -1 \end{array} \right)$$

$$\begin{aligned} \phi_0(\xi, \eta) &= \frac{1 - \xi - \eta + \xi\eta}{4} \\ \phi_1(\xi, \eta) &= \frac{1 + \xi - \eta - \xi\eta}{4} \\ \phi_2(\xi, \eta) &= \frac{1 + \xi + \eta + \xi\eta}{4} \\ \phi_3(\xi, \eta) &= \frac{1 - \xi + \eta - \xi\eta}{4} \end{aligned} \tag{A.24}$$

**Определение двумерных базисов через комбинацию одномерных** Обратим внимание, что в искомые билинейные базисные функции линейны в каждом из направлений  $\xi, \eta$ , если брать их по отдельности. Значит можно представить эти функции как комбинацию одномерных линейных базисов (A.20) в каждом из направлений. Узлы двумерного параметрического квадрата можно выразить через узлы линейного базиса в параметрическом одномерном сегменте, рассмотренном в п. A.3.1.2:

$$\boldsymbol{\xi}_0 = (\xi_0^{1D}, \xi_0^{1D}), \quad \boldsymbol{\xi}_1 = (\xi_1^{1D}, \xi_0^{1D}), \quad \boldsymbol{\xi}_2 = (\xi_1^{1D}, \xi_1^{1D}), \quad \boldsymbol{\xi}_3 = (\xi_0^{1D}, \xi_1^{1D}).$$

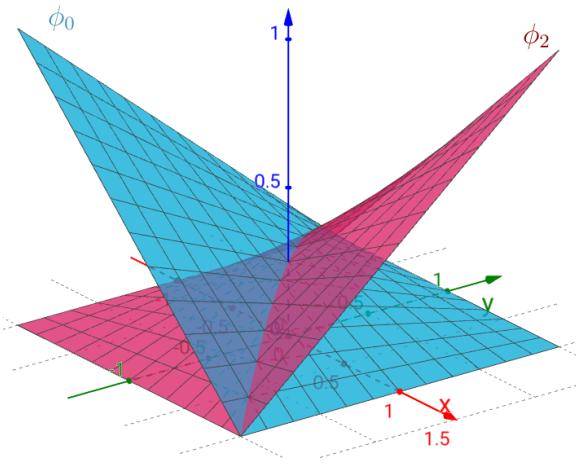


Рис. 32: Билинейные функции  $\phi_0, \phi_2$  в параметрическом квадрате

Значит и соответствующие базисные функции можно выразить через линейный одномерный базис  $\phi^{1D}$  из соотношений (A.20):

$$\begin{aligned}\phi_0(\xi, \eta) &= \phi_0^{1D}(\xi)\phi_0^{1D}(\eta) = \frac{1-\xi}{2}\frac{1-\eta}{2}, \\ \phi_1(\xi, \eta) &= \phi_1^{1D}(\xi)\phi_0^{1D}(\eta) = \frac{1+\xi}{2}\frac{1-\eta}{2}, \\ \phi_2(\xi, \eta) &= \phi_1^{1D}(\xi)\phi_1^{1D}(\eta) = \frac{1+\xi}{2}\frac{1+\eta}{2}, \\ \phi_3(\xi, \eta) &= \phi_0^{1D}(\xi)\phi_1^{1D}(\eta) = \frac{1-\xi}{2}\frac{1+\eta}{2}.\end{aligned}$$

Раскрыв скобки можно убедится, что мы получили тот же билинейный базис, что и ранее (A.24).

**Биквадратичный базис** Применим этот метод для вычисления биквадратичного базиса, определённого в точках на рис. 31б. В качестве основе возьмём квадратичный одномерный базис  $\phi_i^{1D}$  из (A.21).

$$\begin{aligned}\phi_0(\xi, \eta) &= \phi_0^{1D}(\xi)\phi_0^{1D}(\eta) = \frac{\xi^2 - \xi}{2}\frac{\eta^2 - \eta}{2}, & \phi_1(\xi, \eta) &= \phi_1^{1D}(\xi)\phi_0^{1D}(\eta) = \frac{\xi^2 + \xi}{2}\frac{\eta^2 - \eta}{2}, \\ \phi_2(\xi, \eta) &= \phi_1^{1D}(\xi)\phi_1^{1D}(\eta) = \frac{\xi^2 + \xi}{2}\frac{\eta^2 + \eta}{2}, & \phi_3(\xi, \eta) &= \phi_0^{1D}(\xi)\phi_1^{1D}(\eta) = \frac{\xi^2 - \xi}{2}\frac{\eta^2 + \eta}{2}, \\ \phi_4(\xi, \eta) &= \phi_2^{1D}(\xi)\phi_0^{1D}(\eta) = (1 - \xi^2)\frac{\eta^2 - \eta}{2}, & \phi_5(\xi, \eta) &= \phi_1^{1D}(\xi)\phi_2^{1D}(\eta) = \frac{\xi^2 + \xi}{2}(1 - \eta^2), \\ \phi_6(\xi, \eta) &= \phi_2^{1D}(\xi)\phi_1^{1D}(\eta) = (1 - \xi^2)\frac{\eta^2 + \eta}{2}, & \phi_7(\xi, \eta) &= \phi_0^{1D}(\xi)\phi_2^{1D}(\eta) = \frac{\xi^2 - \xi}{2}(1 - \eta^2), \\ \phi_8(\xi, \eta) &= \phi_2^{1D}(\xi)\phi_2^{1D}(\eta) = (1 - \xi^2)(1 - \eta^2).\end{aligned}\tag{A.25}$$

### Бикубический базис

#### Неполный биквадратичный базис

#### Неполный бикубический базис

## A.4 Геометрические алгоритмы

### A.4.1 Преобразование координат

Рассмотрим преобразование из двумерной параметрической системы координат  $\xi$  в физическую систему  $x$ . Такое преобразование полностью определяется покоординатными функциями  $x(\xi)$ . Далее получим соотношения, связывающие операции дифференцирования и интегрирования в физической и параметрической областях.

#### A.4.1.1 Матрица Якоби

Будем рассматривать двумерное преобразование  $(\xi, \eta) \rightarrow (x, y)$ . Линеаризуем это преобразование (разложим в ряд Фурье до линейного слагаемого)

$$x(\xi_0 + d\xi, \eta_0 + d\eta) \approx x_0 + \left. \frac{\partial x}{\partial \xi} \right|_{\xi_0, \eta_0} d\xi + \left. \frac{\partial x}{\partial \eta} \right|_{\xi_0, \eta_0} d\eta,$$

$$y(\xi_0 + d\xi, \eta_0 + d\eta) \approx y_0 + \left. \frac{\partial y}{\partial \xi} \right|_{\xi_0, \eta_0} d\xi + \left. \frac{\partial y}{\partial \eta} \right|_{\xi_0, \eta_0} d\eta,$$

где  $x_0 = x(\xi_0, \eta_0)$ ,  $y_0 = y(\xi_0, \eta_0)$ . Переписывая это выражение в векторном виде, получим

$$\mathbf{x}(\xi_0 + d\xi) - \mathbf{x}_0 = J(\xi_0) d\xi. \quad (\text{A.26})$$

Матрица  $J$  (зависящая от точки приложения в параметрической плоскости) называется матрицей Якоби:

$$J = \begin{pmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{pmatrix} = \begin{pmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{pmatrix} \quad (\text{A.27})$$

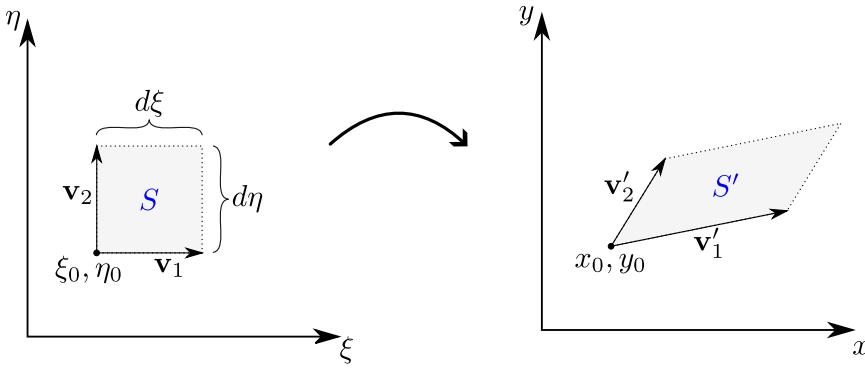


Рис. 33: Преобразование элементарного объёма

**Якобиан** Определитель матрицы Якоби (якобиан), взятый в конкретной точке параметрической плоскости  $\xi_0$ , показывает, во сколько раз увеличился элементарный объём около этой точки в результате преобразования. Действительно, рассмотрим два перпендикулярных элементарных вектора в параметрической системе координат:  $\mathbf{v}_1 = (d\xi, 0)$  и  $\mathbf{v}_2 = (0, d\eta)$  отложенных от точки  $\xi_0$  (см. рис. 33). В результате преобразования по формуле (A.26) получим следующие преобразования концевых точек

и векторов:

$$\begin{aligned} (\xi_0, \eta_0) &\rightarrow (x_0, y_0), \\ (\xi_0 + d\xi, \eta_0) &\rightarrow (x_0 + J_{11}d\xi, y_0 + J_{21}d\xi) \Rightarrow \mathbf{v}_1 \rightarrow \mathbf{v}'_1 = (J_{11}d\xi, J_{21}d\xi), \\ (\xi_0, \eta_0 + d\eta) &\rightarrow (x_0 + J_{12}d\eta, y_0 + J_{22}d\eta) \Rightarrow \mathbf{v}_2 \rightarrow \mathbf{v}'_2 = (J_{12}d\eta, J_{22}d\eta). \end{aligned}$$

Элементарный объём равен площади параллелограмма, построенного на элементарных векторах. В параметрической плоскости согласно (A.4) получим

$$|S| = \mathbf{v}_1 \times \mathbf{v}_2 = d\xi d\eta,$$

и аналогично для физической плоскости:

$$|S'| = \mathbf{v}'_1 \times \mathbf{v}'_2 = (J_{11}J_{22} - J_{12}J_{21})d\xi d\eta = |J|d\xi d\eta$$

Сравнивая два последних соотношения приходим к выводу, что элементарный объём в результате преобразования увеличился в  $|J|$  раз. Тогда можно записать

$$dx dy = |J| d\xi d\eta \quad (\text{A.28})$$

#### A.4.1.2 Дифференцирование в параметрической плоскости

Пусть задана некоторая функция  $f(x, y)$ . Распишем её производную по параметрическим координатам:

$$\begin{aligned} \frac{\partial f}{\partial \xi} &= \frac{\partial f}{\partial x} \frac{\partial x}{\partial \xi} + \frac{\partial f}{\partial y} \frac{\partial y}{\partial \xi}, \\ \frac{\partial f}{\partial \eta} &= \frac{\partial f}{\partial x} \frac{\partial x}{\partial \eta} + \frac{\partial f}{\partial y} \frac{\partial y}{\partial \eta}. \end{aligned}$$

Вспоминая определение (A.27), запишем

$$\begin{pmatrix} \frac{\partial f}{\partial \xi} \\ \frac{\partial f}{\partial \eta} \end{pmatrix} = J^T \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix} = \begin{pmatrix} J_{11} & J_{21} \\ J_{12} & J_{22} \end{pmatrix} \begin{pmatrix} \frac{\partial f}{\partial \xi} \\ \frac{\partial f}{\partial \eta} \end{pmatrix}$$

Обратная зависимость примет вид

$$\begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix} = (J^T)^{-1} \begin{pmatrix} \frac{\partial f}{\partial \xi} \\ \frac{\partial f}{\partial \eta} \end{pmatrix} = \frac{1}{|J|} \begin{pmatrix} J_{22} & -J_{21} \\ -J_{12} & J_{11} \end{pmatrix} \begin{pmatrix} \frac{\partial f}{\partial \xi} \\ \frac{\partial f}{\partial \eta} \end{pmatrix}$$

#### A.4.1.3 Интегрирование в параметрической плоскости

Пусть в физической области  $x, y$  задана область  $D_x$ . Интеграл функции  $f(x, y)$  по этой области можно расписать, используя замену (A.28)

$$\int_{D_x} f(x, y) dx dy = \int_{D_\xi} f(\xi, \eta) |J(\xi, \eta)| d\xi d\eta, \quad (\text{A.29})$$

где  $f(\xi, \eta) = f(x(\xi, \eta), y(\xi, \eta))$ , а  $D_\xi$  – образ области  $D_x$  в параметрической плоскости.

#### A.4.1.4 Двумерное линейное преобразование. Параметрический треугольник

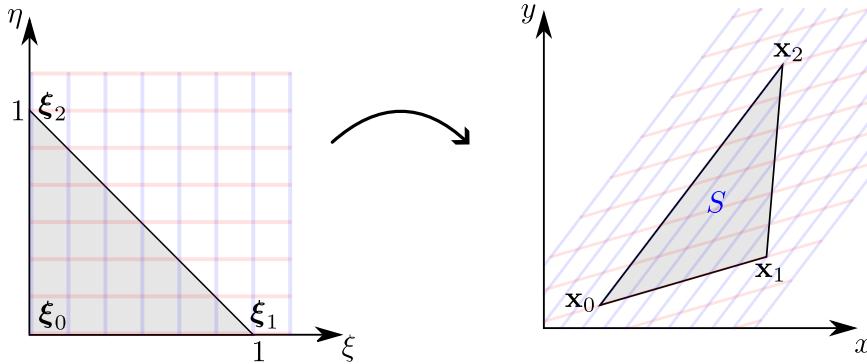


Рис. 34: Преобразование из параметрического треугольника

Рассмотрим двумерное преобразование, при котором определяющие функции являются линейными. То есть представимыми в виде

$$\begin{aligned} x(\xi, \eta) &= A_x \xi + B_x \eta + C_x, \\ y(\xi, \eta) &= A_y \xi + B_y \eta + C_y. \end{aligned}$$

Для определения шести констант, определяющих это преобразование, достаточно выбрать три любые (не лежащие на одной прямой) точки:  $(\xi_i, \eta_i) \rightarrow (x_i, y_i)$  для  $i = 0, 1, 2$ . В результате получим систему из шести линейных уравнений (три точки по две координаты), из которой находятся константы  $A_{x,y}, B_{x,y}, C_{x,y}$ . Пусть три точки в параметрической плоскости образуют единичный прямоугольный треугольник (рис. 34):

$$\xi_0, \eta_0 = (0, 0), \quad \xi_1, \eta_1 = (1, 0), \quad \xi_2, \eta_2 = (0, 1).$$

Тогда система линейных уравнений примет вид

$$\begin{aligned} x_0 &= C_x, & y_0 &= C_y, \\ x_1 &= A_x + C_x, & y_1 &= A_y + C_y, \\ y_2 &= B_x + C_x, & y_2 &= B_y + C_y. \end{aligned}$$

Определив коэффициенты преобразования из этой системы, окончательно запишем преобразование

$$\begin{aligned} x(\xi, \eta) &= (x_1 - x_0)\xi + (x_2 - x_0)\eta + x_0, \\ y(\xi, \eta) &= (y_1 - y_0)\xi + (y_2 - y_0)\eta + y_0. \end{aligned} \quad (\text{A.30})$$

Матрица Якоби этого преобразования (A.27) не будет зависеть от параметрических координат  $\xi, \eta$ :

$$J = \begin{pmatrix} x_1 - x_0 & x_2 - x_0 \\ y_1 - y_0 & y_2 - y_0 \end{pmatrix}. \quad (\text{A.31})$$

Якобиан преобразования будет равен удвоенной площади треугольника  $S$ , составленного из определяющих точек в физической плоскости:

$$|J| = (x_1 - x_0)(y_2 - y_0) - (y_1 - y_0)(x_2 - x_0) = (\mathbf{x}_1 - \mathbf{x}_0) \times (\mathbf{x}_2 - \mathbf{x}_0) = 2|S|. \quad (\text{A.32})$$

Распишем интеграл по треугольнику  $S$  по формуле (A.29). Вследствии линейности преобразования якобиан постоянен и, поэтому, его можно вынести его из-под интеграла:

$$\int_S f(x, y) dx dy = |J| \int_0^1 \int_0^{1-\xi} f(\xi, \eta) d\eta d\xi. \quad (\text{A.33})$$

#### A.4.1.5 Двумерное билинейное преобразование. Параметрический квадрат

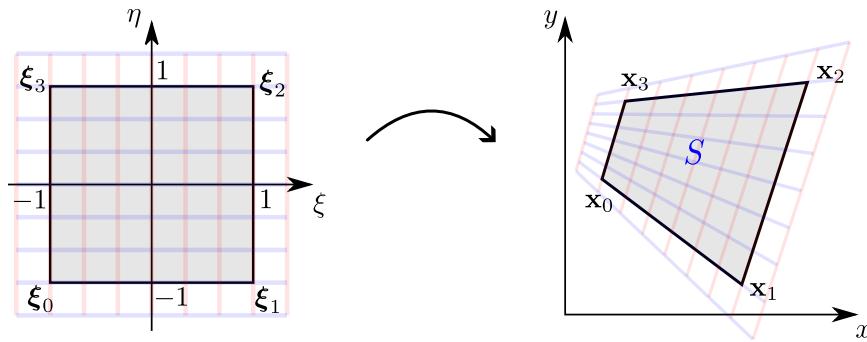


Рис. 35: Преобразование из параметрического квадрата

#### A.4.1.6 Трёхмерное линейное преобразование. Параметрический тетраэдр

TODO

#### A.4.2 Свойства многоугольника

##### A.4.2.1 Площадь многоугольника

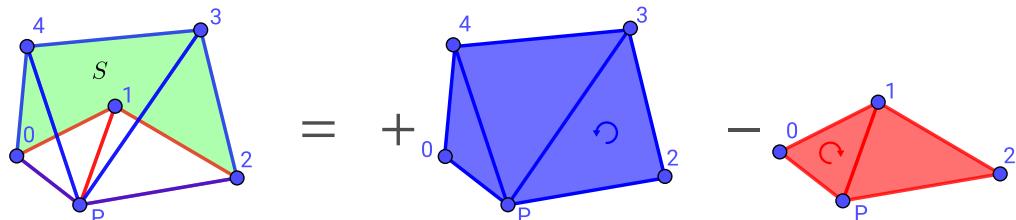


Рис. 36: Площадь произвольного многоугольника

Рассмотрим произвольный несамопересекающийся  $N$ -угольник  $S$ , заданный координатами своих узлов  $\mathbf{x}_i$ ,  $i = \overline{0, N-1}$ , пронумерованных последовательно против часовой стрелки (рис. 36). Далее введём произвольную точку  $\mathbf{p}$  и от этой точки будем строить ориентированные треугольники до граней многоугольника:

$$\Delta_i^p = (\mathbf{p}, \mathbf{x}_i, \mathbf{x}_{i+1}), \quad i = \overline{0, N-1},$$

(для корректности записи будем считать, что  $\mathbf{x}_N = \mathbf{x}_0$ ). Тогда площадь исходного многоугольника  $S$  будет равна сумме знаковых площадей треугольников  $\Delta_i^p$ :

$$|S| = \sum_{i=0}^{N-1} |\Delta_i^p|, \quad |\Delta_i^p| = \frac{(\mathbf{x}_i - \mathbf{p}) \times (\mathbf{x}_{i+1} - \mathbf{p})}{2}.$$

Знак площади ориентированного треугольника зависит от направления закрутки его узлов: она положительна для закрутки против часовой стрелки и отрицательна, если узлы пронумерованы по часовой стрелке. В частности, на рисунке 36 видно, что треугольники, отмеченные красным:  $P01, P12$ , будут иметь отрицательную площадь, а синие треугольники  $P23, P34, P40$  – положительную. Сумма этих площадей с учётом знака даст искомую площадь многоугольника.

Для сокращения вычислений воспользуемся произвольностью положения  $\mathbf{p}$  и совместим её с точкой  $\mathbf{x}_0$ . Тогда треугольники  $\Delta_0^p, \Delta_{N-1}^p$  выродятся (будут иметь нулевую площадь). Обозначим такую последовательную триангуляцию как

$$\Delta_i = (\mathbf{x}_0, \mathbf{x}_i, \mathbf{x}_{i+1}), \quad i = \overline{1, N-2}. \quad (\text{A.34})$$

Знаковая площадь ориентированного треугольника будет равна

$$|\Delta_i| = \frac{(\mathbf{x}_i - \mathbf{x}_0) \times (\mathbf{x}_{i+1} - \mathbf{x}_0)}{2}. \quad (\text{A.35})$$

Тогда окончательно формула определения площади примет вид

$$|S| = \sum_{i=1}^{N-2} |\Delta_i|. \quad (\text{A.36})$$

**Плоский полигон в пространстве** Если плоский полигон  $S$  расположен в трёхмерном пространстве, то правая часть формулы (A.35) согласно определению векторного произведения в трёхмерном пространстве (A.3) – есть вектор. Чтобы получить скалярную площадь, нужно спроектировать этот вектор на единичную нормаль к плоскости многоугольника:

$$\mathbf{n} = \frac{\mathbf{k}}{|\mathbf{k}|}, \quad \mathbf{k} = (\mathbf{x}_1 - \mathbf{x}_0) \times (\mathbf{x}_2 - \mathbf{x}_0).$$

Эта формула записана из предположения, что узел  $\mathbf{x}_2$  не лежит на одной прямой с узлами  $\mathbf{x}_0, \mathbf{x}_1$ . Иначе вместо  $\mathbf{x}_2$  нужно выбрать любой другой узел, удовлетворяющий этому условию. Тогда площадь ориентированного треугольника, построенного в трёхмерном пространстве запишется через смешанное произведение:

$$|\Delta_i| = \frac{((\mathbf{x}_i - \mathbf{x}_0) \times (\mathbf{x}_{i+1} - \mathbf{x}_0)) \cdot \mathbf{n}}{2}. \quad (\text{A.37})$$

Формула для определения площади полигона (A.36) будет по прежнему верна. При этом итоговый знак величины  $S$  будет положительным, если закрутка полигона положительная (против часовой стрелки) при взгляде со стороны вычисленной нормали  $\mathbf{n}$ .

#### A.4.2.2 Интеграл по многоугольнику

Рассмотрим интеграл функции  $f(x, y)$  по  $N$ -угольнику  $S$ , заданному последовательными координатами своих узлов  $\mathbf{x}_i$ . Введём последовательную триангуляцию согласно (A.34). Тогда интеграл по многоугольнику можно расписать как сумму интегралов по ориентированным треугольникам:

$$\int_S f(x, y) dx dy = \sum_{i=1}^{N-2} \int_{\Delta_i} f(x, y) dx dy. \quad (\text{A.38})$$

Далее для вычисления интегралов в правой части воспользуемся преобразованием к параметрическому треугольнику (п. A.4.1.4). Следуя формуле интегрирования (A.33), распишем интеграл по  $i$ -ому треугольнику:

$$\int_{\Delta_i} f(x, y) dx dy = |J_i| \int_0^1 \int_0^{1-\xi} f_i(\xi, \eta) d\eta d\xi,$$

где якобиан  $|J_i|$  согласно (A.32) есть удвоенная площадь ориентированного треугольника  $\Delta_i$  (положительная при закрутке против часовой стрелки и отрицательная иначе):

$$|J_i| = 2|\Delta_i| = (\mathbf{x}_i - \mathbf{x}_0) \times (\mathbf{x}_{i+1} - \mathbf{x}_0),$$

а функция  $f_i(\xi, \eta)$  есть функция от преобразованных согласно (A.30) переменных:

$$f_i(\xi, \eta) = f((\mathbf{x}_i - \mathbf{x}_0)\xi + (\mathbf{x}_{i+1} - \mathbf{x}_0)\eta + \mathbf{x}_0).$$

Окончательно запишем

$$\int_S f(x, y) dx dy = 2 \sum_{i=1}^{N-2} |\Delta_i| \int_0^1 \int_0^{1-\xi} f_i(\xi, \eta) d\eta d\xi. \quad (\text{A.39})$$

Отметим, что эта формула работает и в том случае, когда полигон расположен в трёхмерном пространстве (знаковую площадь при этом следует вычислять по (A.37)).

#### A.4.2.3 Центр масс многоугольника

По определению, координаты центра масс  $\mathbf{c}$  области  $S$  равны среднеинтегральным значениям координатных функций. То есть

$$c_x = \frac{1}{|S|} \int_S x dx dy, \quad c_y = \frac{1}{|S|} \int_S y dx dy.$$

Далее распишем интеграл в правой части через последовательную триангуляцию согласно (A.38) с учётом линейного преобразования (A.30):

$$\begin{aligned}
\int_S x \, dx dy &= \sum_{i=1}^{N-2} \int_{\Delta_i} x \, dx dy \\
&= \sum_{i=1}^{N-2} |J_i| \int_0^1 \int_0^{1-\xi} ((x_i - x_0)\xi + (x_{i+1} - x_0)\eta + x_0) d\eta d\xi \\
&= \sum_{i=1}^{N-2} \frac{|J_i|}{2} \frac{x_0 + x_i + x_{i+1}}{3} \\
&= \sum_{i=1}^{N-2} |\Delta_i| \frac{x_0 + x_i + x_{i+1}}{3}.
\end{aligned}$$

Итого, с учётом (A.36), координаты центра масс примут вид

$$\mathbf{c} = \frac{\sum_{i=1}^{N-2} \frac{\mathbf{x}_0 + \mathbf{x}_i + \mathbf{x}_{i+1}}{3} |\Delta_i|}{\sum_{i=1}^{N-2} |\Delta_i|}.$$

Если полигон расположен в двумерном пространстве  $xy$ , то знаковая площадь треугольников вычисляется по формуле (A.35). В случае трёхмерного пространства должна использоваться формула (A.37).

#### A.4.3 Свойства многогранника

##### A.4.3.1 Объём многогранника

TODO

##### A.4.3.2 Интеграл по многограннику

TODO

##### A.4.3.3 Центр масс многогранника

TODO

#### A.4.4 Поиск многоугольника, содержащего заданную точку

TODO

## B Работа с инфраструктурой проекта CFDCourse

## B.1 Сборка и запуск

### B.1.1 Сборка проекта CFDCourse

Описанная ниже процедура собирает проект в отладочной конфигурации. Для проведения необходимых модификаций для сборки релизной версии смотри [B.1.3](#).

#### B.1.1.1 Подготовка

1. Для сборки проекта необходимо установить `git` и `cmake>=3.0`

В Windows необходимо скачать и установить дистрибутивы:

- <https://github.com/git-for-windows/git/releases/download/v2.38.1.windows.1/Git-2.38.1-64-bit.exe>
- [https://github.com/Kitware/CMake/releases/download/v3.24.2/cmake-3.24.2-windows-x86\\_64.msi](https://github.com/Kitware/CMake/releases/download/v3.24.2/cmake-3.24.2-windows-x86_64.msi)

При установке cmake проследите, что бы путь к `cmake.exe` сохранился в системных путях. Msi установщик спросит об этом в диалоге.

В **линуксе** используйте менеджеры пакетов, предоставляемые вашим дистрибутивом. Также проследите чтобы были доступны компилятор `g++` и отладчик `gdb`.

2. Создайте папку в системе для репозиториев. Например `D:/git_repos/`
3. Возьмите необходимые заголовочные библиотеки boost из <https://disk.yandex.ru/d/GwTZUvfAqPsZ> и распакуйте архив в папку для репозиториев (`D:/git_repos/boost`). Проследите, чтобы внутри папки boost сразу шли папки с кодом (`accumulators`, `algorithm`, ...) и заголовочные файлы (`align.hpp`, `aligned_storage.hpp`, ...) без дополнительных уровней вложения.
4. Откройте терминал (git bash в Windows).
5. С помощью команды `cd` в терминале перейдите в папку для репозиториев

```
> cd D:/git_repos
```

6. Клонируйте репозиторий

```
> git clone https://github.com/kalininei/CFDCourse24
```

В директории (`D:/git_repos` в примере) появится папка `CFDCourse24`, которая является корневой папкой проекта

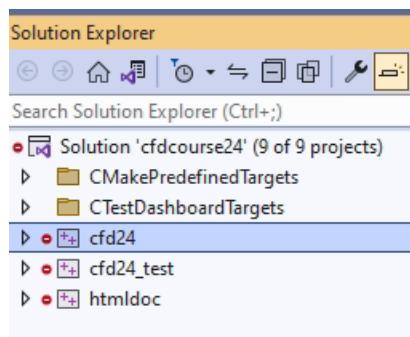
#### B.1.1.2 VisualStudio

1. Создайте папку `build` в корне проекта `CFDCourse24`
2. Скопируйте скрипт `winbuild64.bat` в папку `build`. Далее вносить изменения только в скопированном файле.

3. Скрипт написан для версии **Visual Studio 2019**. Если используется другая версия, измените в скрипте значение переменной **CMGenerator** на соответствующие вашей версии. Значения для разных версий Visual Studio написаны ниже

```
SET CMGenerator="Visual Studio 17 2022"
SET CMGenerator="Visual Studio 16 2019"
SET CMGenerator="Visual Studio 15 2017"
SET CMGenerator="Visual Studio 14 2015"
```

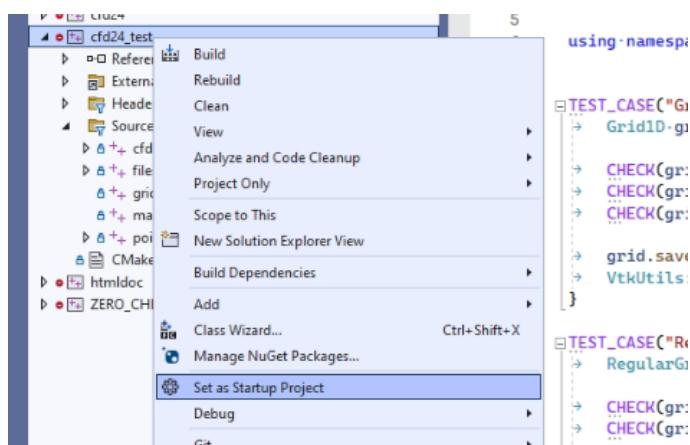
4. Запустите скрипт **winbuild64.bat** из папки **build**. Нужен доступ к интернету. В процессе будет скачано около 200Мб пакетов, поэтому первый запуск может занять время
5. После сборки в папке **build** появится проект **VisualStudio cfdcourse24.sln**. Его нужно открыть в **VisualStudio**. Дерево решения должно иметь следующий вид;



Проекты:

- **cf2d24** – расчётная библиотека
- **cf2d24\_test** – модульные тесты для расчётных функций

6. Проект **cf2d24\_test** необходимо назначить запускаемым проектом. Для этого нажать правой кнопкой мыши по проекту и в выпадающем меню выбрать соответствующий пункт. После этого заголовок проекта должен стать жирным.



7. Скомпилировать решение. Несколько способов:

- **Ctrl+Shift+B**,

- **Build->Build Solution** в основном меню,
- **Build Solution** в меню решения в дереве решения,
- **Build** в меню проекта **cf2d4\_test**.

8. Запустить тесты (проект

**cf2d4\_test**) нажав **F5** (или кнопку отладки в меню). После отработки должно высветиться сообщение об успешном прохождении всех тестов.

9. Бинарные файлы будут скомпилированы в папку **CFDCourse24/build/bin/Debug**. В случае работы через отладчик выходная директория, куда будут скидываться все файлы (в частности, vtk), должна быть **CFDCourse24/build/src/test/**.

### B.1.1.3 VSCode

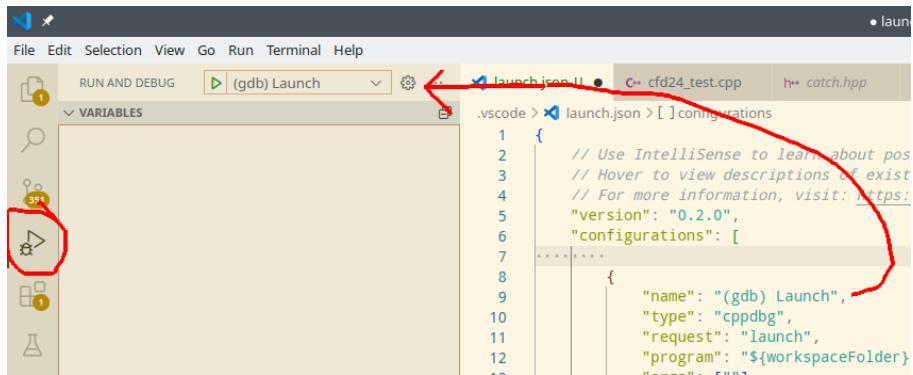
1. Открыть корневую папку проекта через **File->Open Folder**
2. Установить предлагаемые расширения cmake, c++
3. Для настройки отладки создайте конфигурацию launch.json следующего вида

```

5   "version": "0.2.0",
6   "configurations": [
7     {
8       "name": "(gdb) Launch",
9       "type": "cppdbg",
10      "request": "launch",
11      "program": "${workspaceFolder}/build/bin/cfd24_test",
12      "args": [""], ←
13      "stopAtEntry": false,
14      "cwd": "${fileDirname}",
15      "environment": [],
16      "externalConsole": false,
17      "MIMode": "gdb",
18      "setupCommands": [
19        {
20          "description": "Enable pretty-printing for gdb",
21          "text": "-enable-pretty-printing",
22          "ignoreFailures": true
23        },
24        {
25          "description": "Set Disassembly Flavor to Intel",
26          "text": "-gdb-set disassembly-flavor intel",
27          "ignoreFailures": true
28        }
29      ],
30    }
31  ]

```

- Для этого перейдите в меню **Run and Debug** (**Ctrl+Shift+D**), нажмите **create launch.json**, выберите пункт **Node.js**.
- После этого в корневой папке появится файл **.vscode/launch.json**.
- Откройте этот файл в **vscode**, нажмите **Add configuration**, **(gdb) Launch** или **(Windows) Launch** в зависимости от ОС.
- Далее напишите имя программы как показано на картинке.
- Используйте поле args для установки аргументов запуска.
- Выберите созданную конфигурацию для запуска отладчика по **F5**.



На скриншотах представлены настройки в случае работы в линуксе. Для работы под виндоус

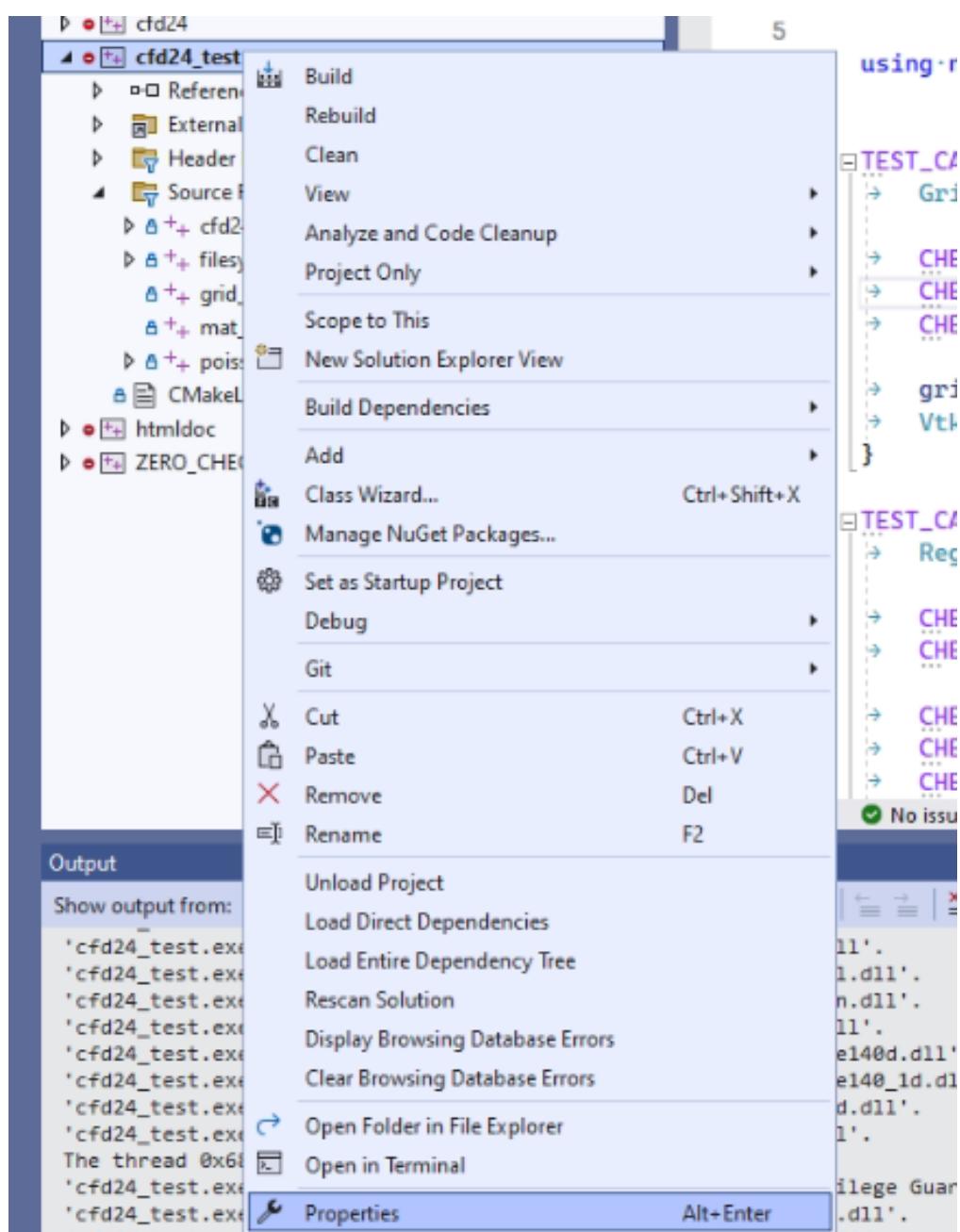
```
"name" : "(Windows) Launch",
"program": "${workspaceFolder}/build/bin/Debug/cfd24_test.exe"
```

### B.1.2 Запуск конкретного теста

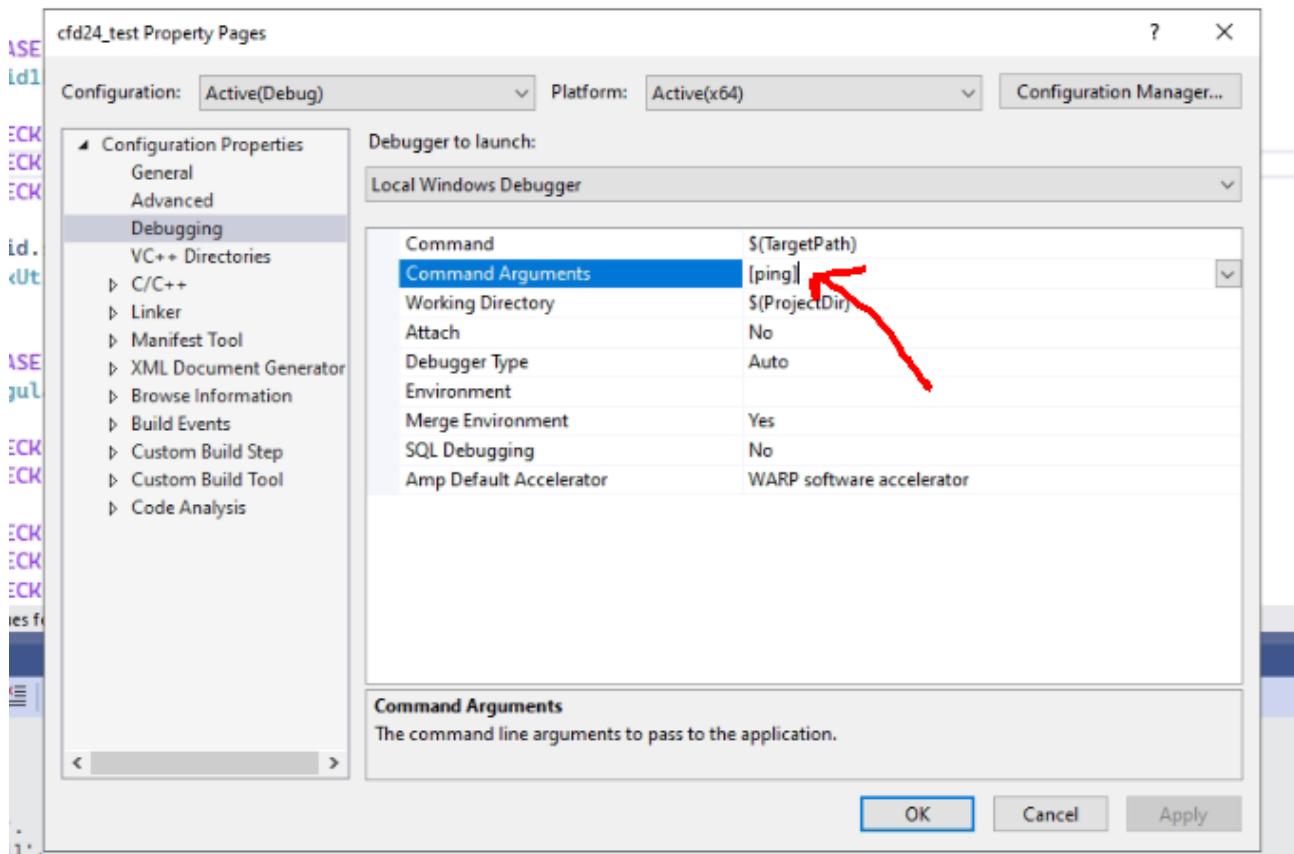
По умолчанию программа `cfд_test` прогоняет все объявленные в проекте тесты. Иногда может возникнуть необходимость запустить только конкретный тест в целях отладки или проверки. Для этого нужно передать программе аргумент с тегом для этого теста.

Тег для теста – это второй аргумент в макросе `TEST_CASE`, записанный в квадратных скобках. Добавлять нужно вместе со скобками. Например, `[ping]`.

Чтобы добавить аргумент в `VisualStudio`, необходимо в контекстном меню проекта `cfд_test` выбрать опции отладки



и там в поле Аргументы прописать нужный тэг.



В VSCode аргументы нужно добавлять в файле `.vscode/launch.json` в поле `args` в кавычках (см. картинку [B.1.1.3](#) с настройками `launch.json`).

### B.1.3 Сборка релизной версии

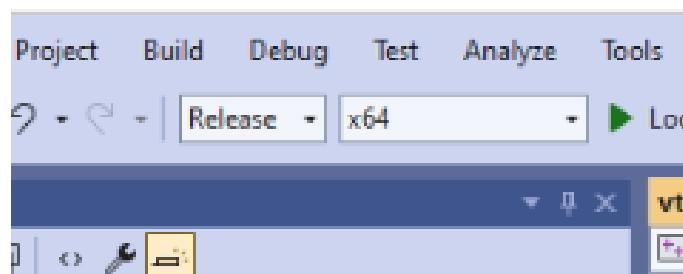
Релизная сборка программ даёт многократное увеличение производительности, но при этом отладка приложений в таком режиме невозможна.

#### Visual Studio

1. Создать папку `build-release` рядом с папкой `build`.
2. Скопировать в неё файл `winbuild64.bat` из папки `build`.
3. В скопированном файле произвести замену `Debug` на `Release`

```
-DCMAKE_BUILD_TYPE=Release ..
```

4. Запустить `winbuild64.bat` из новой папки
5. Открыть `build-release/cfdcourse24.sln` в `Visual Studio`
6. В проекте студии установить релизную сборку



7. Это новое решение, не связанное настройками с `debug`-версией. Поэтому нужно заново настроить запускаемым проектом `cfd_test` и, если нужно, настроить аргументы отладки.
8. Бинарные файлы будут скомпилированы в папку `CFDCourse24/build_release/bin/Release`. В случае работы через отладчик выходная директория – `CFDCourse24/build_release/src/test/`.

## VSCode

1. Выбрать релизную сборку в `build variant`
2. Нажать `Build`
3. Нажать `Launch`



## B.2 Git

### B.2.1 Основные команды

Все команды выполнять в терминале (`git bash` для виндоус), находясь в корневой папке проекта CFDCourse24.

- Для смены директории использовать команду `cd`. Например, находясь в папке A перейти в папку A/B/C

```
> cd B/C
```

- Подняться на директорию выше

```
> cd ..
```

- Просмотр статуса текущего репозитория: текущую ветку, все изменённые файлы и т.п.

```
> git status
```

- Сохранить и скоммитить изменения в текущую ветку

```
> git add .  
> git commit -m "message"
```

“message” – произвольная информация о текущем коммите, которая будет приписана к этому коммиту

- Переключиться на ветку main

```
> git checkout main
```

работает только в том случае, если все файлы скоммичены и статус ветки ‘Up to date’

- Создать новую ветку ответвлённую от последнего коммита текущей ветки и переключиться на неё

```
> git checkout -b new-branch-name
```

new-branch-name – имя новой ветки. Пробелы не допускаются

Эта команда работает даже если есть нескоммиченные изменения. Если необходимо скоммитить изменения в новую ветку, сразу за этой командой нужно вызвать

```
> git add .  
> git commit -m "message"
```

- Сбросить все нескоммиченные изменения. Вернуть файлы в состояние последнего коммита

```
> git reset --hard
```

Все изменения будут утеряны

- **Получить последние изменения** из удалённого хранилища с обновлением текущей ветки

```
> git pull
```

Работает только если статус текущей ветки 'Up to date'.

Если требуется получить изменения, но не обновлять локальную ветку:

```
> git fetch
```

Обновленная ветка будет доступна по имени origin/имя ветки.

- **Просмотр истории** коммитов в текущей ветке (последний коммит будет наверху)

```
> git log
```

- **Просмотр доступных веток** в текущем репозитории

```
> git branch
```

- **Просмотр** актуального состояния дерева репозитория в gui режиме

```
> git gui
```

Далее в меню

Repository->**Visualize all branch history**. В этом же окне можно посмотреть изменения файлов по сравнению с последним коммитом.

Альтернативно, при работе в виндоус можно установить программу GitExtensions и работать в ней.

## B.2.2 Порядок работы с репозиторием CFDCourse

Основная ветка проекта –

**main**. После каждой лекции (в течении 1-2 дней) в эту ветку будет отправлен коммит с сообщением **after-lect{index}**. Этот коммит будет содержать краткое содержание лекции, задание по итогам лекции и необходимые для этого задания изменения кода.

Если предполагается работа с кодом на лекции, то перед лекцией в эту ветку будет отправлен коммит с сообщением **before-lect{index}**. Этот коммит содержит изменения кода для работы на лекции.

Таким образом, **после лекции** необходимо выполнить следующие команды (находясь в ветке **main**)

```
> git reset --hard # очистить локальную копию от изменений,  
# сделанных на лекции (если они не представляют ценности)  
> git pull # получить изменения
```

Перед началом лекции, если была сделана какая то работа по заданиям,

```
> git checkout -b work-lect{index} # создать локальную ветку, содержащую задание  
> git add .  
> git commit -m "{свой комментарий}" # скоммитить свои изменения в эту ветку  
> git checkout main # вернуться на ветку main  
> git pull # получить изменения
```

Даже если задание выполнено не до конца, вы в любой момент можете переключиться на ветку с заданием и его доделать

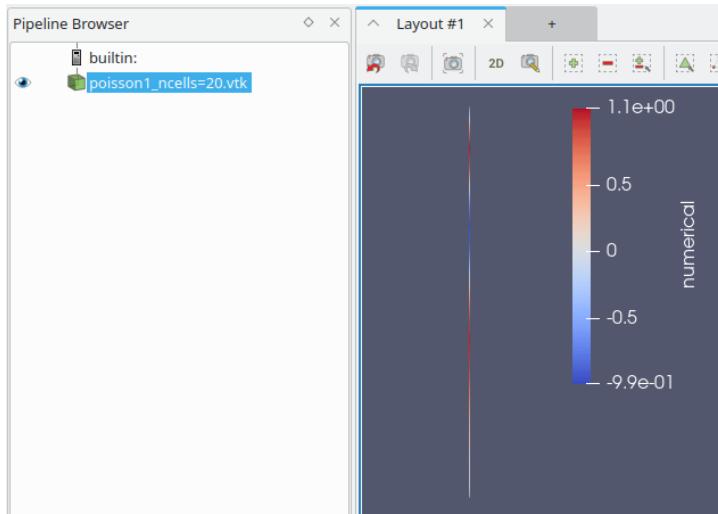
```
> git checkout work-lect{index}
```

Если ничего не было сделано (или все изменения не представляют ценности), можно повторить алгоритм “после лекции”.

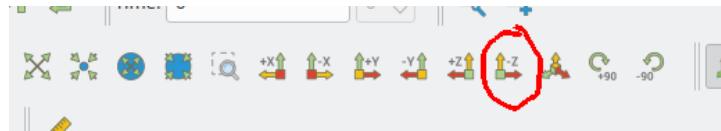
## B.3 Paraview

### B.3.1 Данные на одномерных сетках

Заданные на сетке данные паравью показывает цветом. Поэтому при загрузке одномерных сеток можно видеть картинку типа

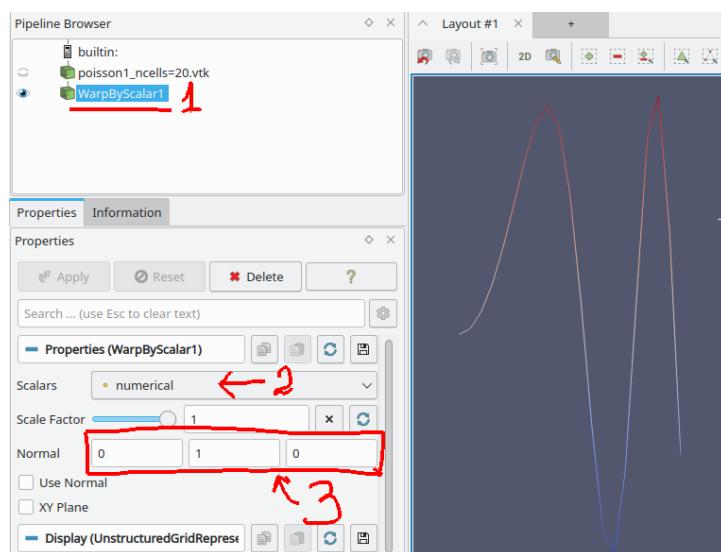


Развернуть изображение в плоскость xy



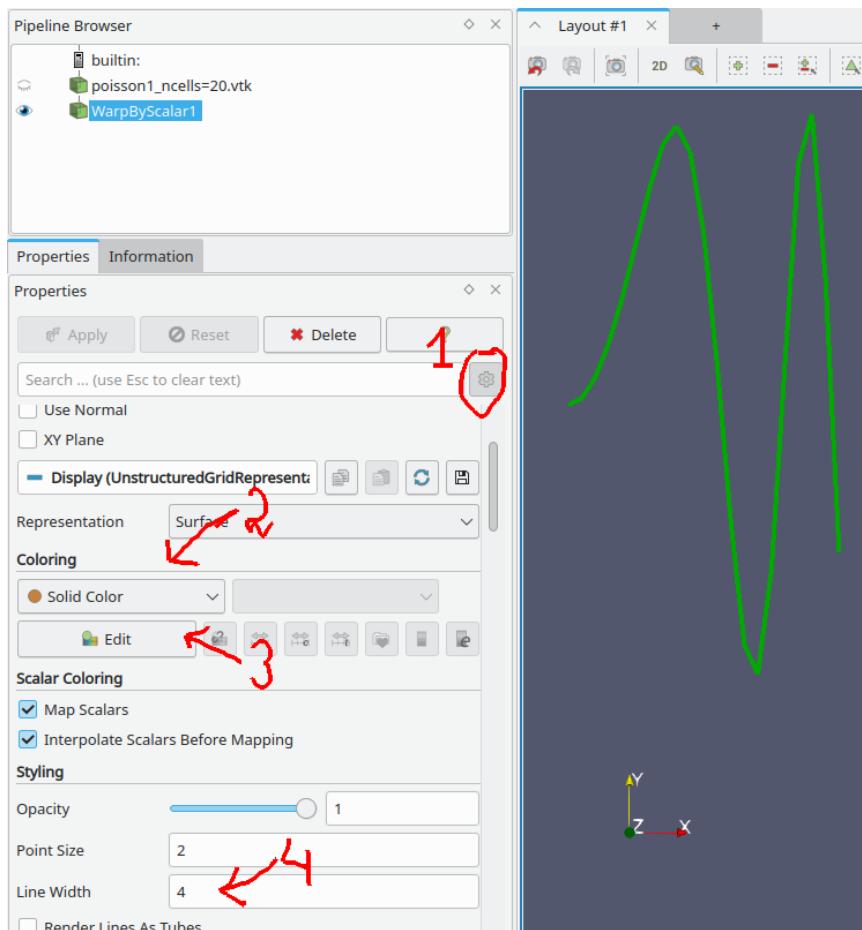
**Отобразить данные в виде у-координаты** Для того, что бы данные отображались в качестве значения по оси ординат, к загруженному файлу необходимо

1. применить фильтр WarpByScalar (В меню Filters->Alphabetical->Warp By Scalar )
2. в меню настройки фильтра указать поле данных, для отображения (numerical в примере ниже)
3. И настроить нормаль, вдоль которой будут проецироваться данные (в нашем случае ось у)



## Цвет и толщина линии

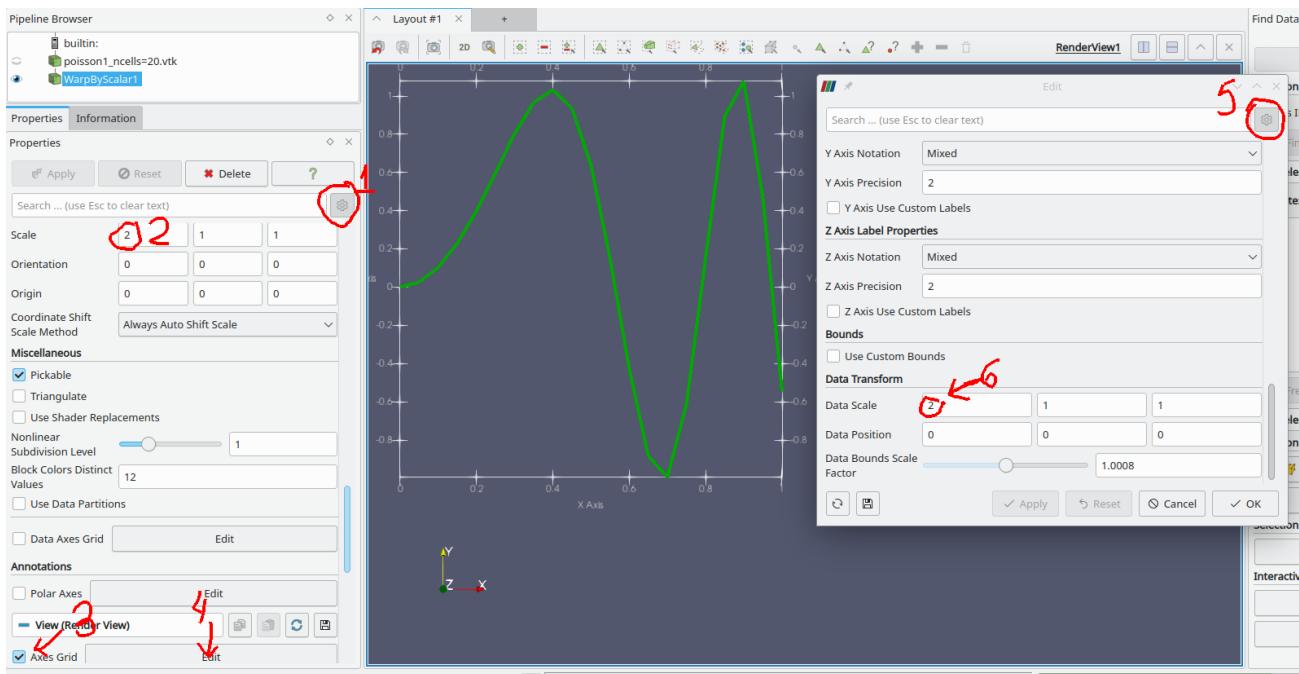
1. Включить подробные опции фильтра
2. Сменить стиль на **Solid Color**
3. В меню **Edit** выбрать желаемый цвет
4. В строке **Line Width** указать толщину линии



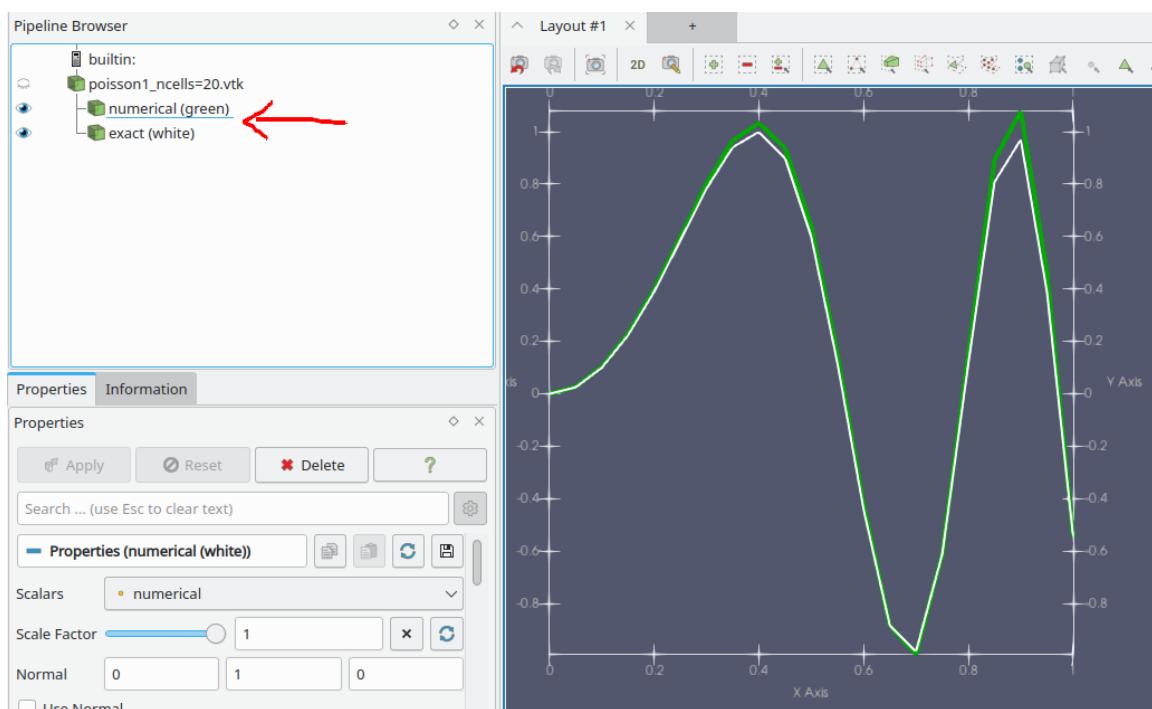
## Настройка масштабов и отображение осей координат

1. Отметьте подробные настройки фильтра
2. В поле **Transforming/Scale** Установите желаемые масштабы (в нашем случае растянуть в два раза по оси x)
3. Установите галку на отображение осей
4. откройте меню настройки осей
5. В нём включите подробные настройки
6. И также поставьте растяжение осей

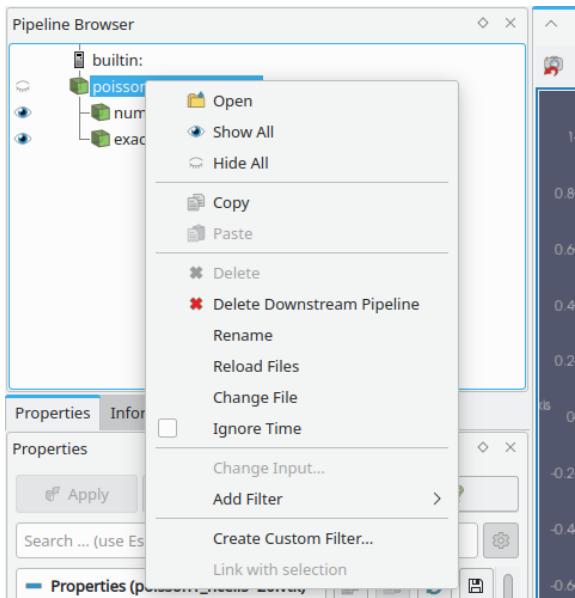
В случае, если масштабировать график не нужно, достаточно выполнить шаг 3.



**Построение графиков для нескольких данных** Если требуется нарисовать рядом несколько графиков для разных данных из одного файла, примените фильтр **Warp By Scalar** для этого файла ещё раз, изменив поле **Scalars** в настройке фильтра. Для наглядности измените имя узла в Pipeline Browser на осмысленные

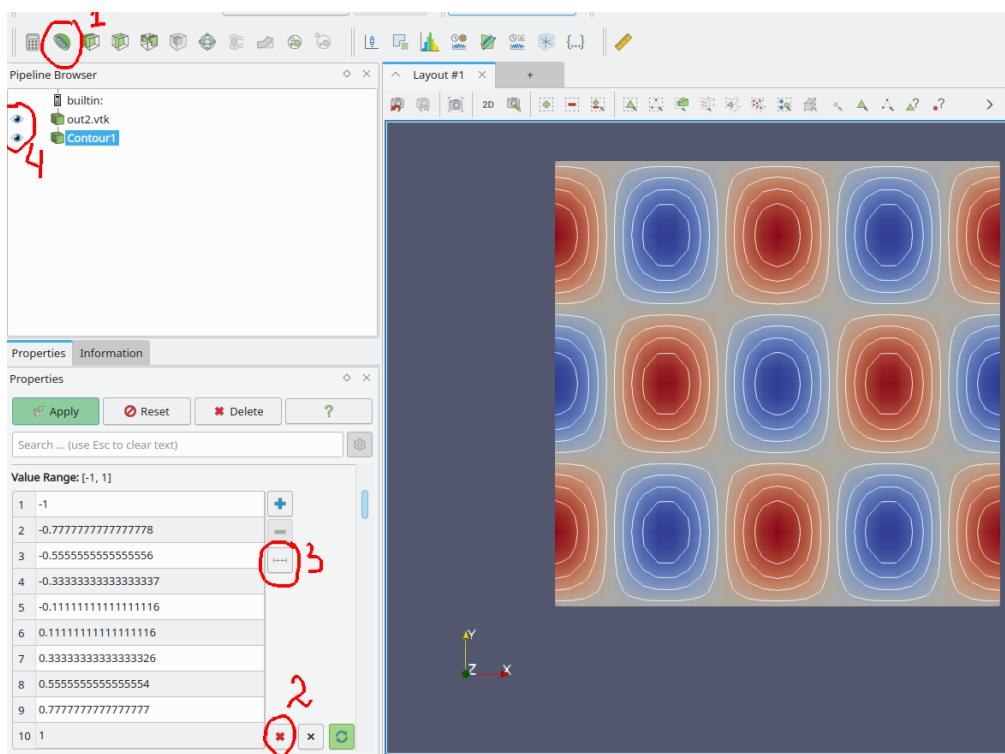


**Обновление данных при изменении исходного файла** В случае, если исходный файл был изменён, нужно в контекстном меню узла соответствующего файла выбрать **Reload Files** (или нажать F5). Если те же самые фильтры нужно применить для просмотра другого файла нужно в этом меню нажать **Change File**.

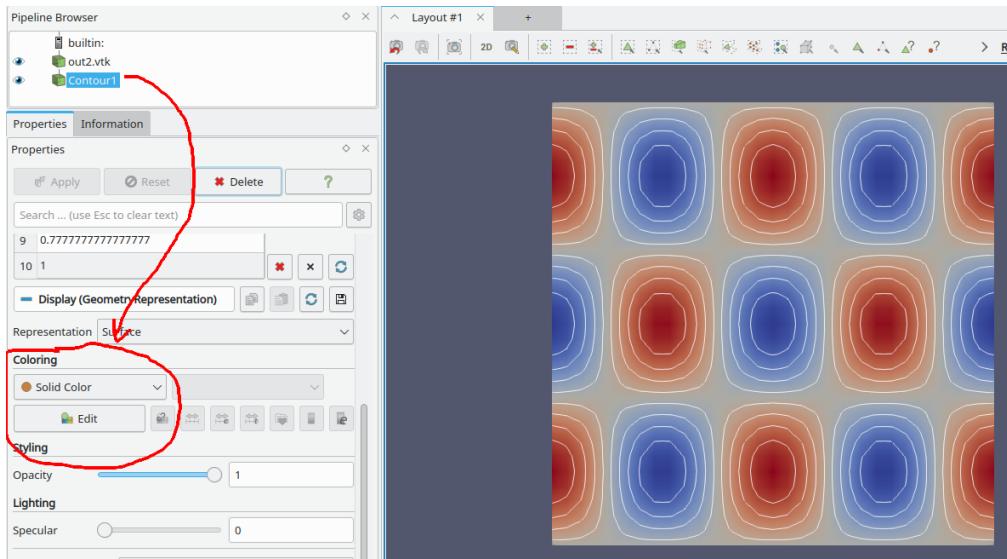


### B.3.2 Изолинии для двумерного поля

- Нажмите иконку **Contour** (или **Filters/Contour**) В настройках фильтра Contour by выберите данные, по которым нужно строить изолинии.
- В настройках фильтра удалите все существующие записи о значениях для изолиний.
- Добавьте равномерные значения. В появившемся меню установите необходимое количество изолиний и их диапазон.
- Если необходимо, включите одновременное отображения цветного поля и изолиний.



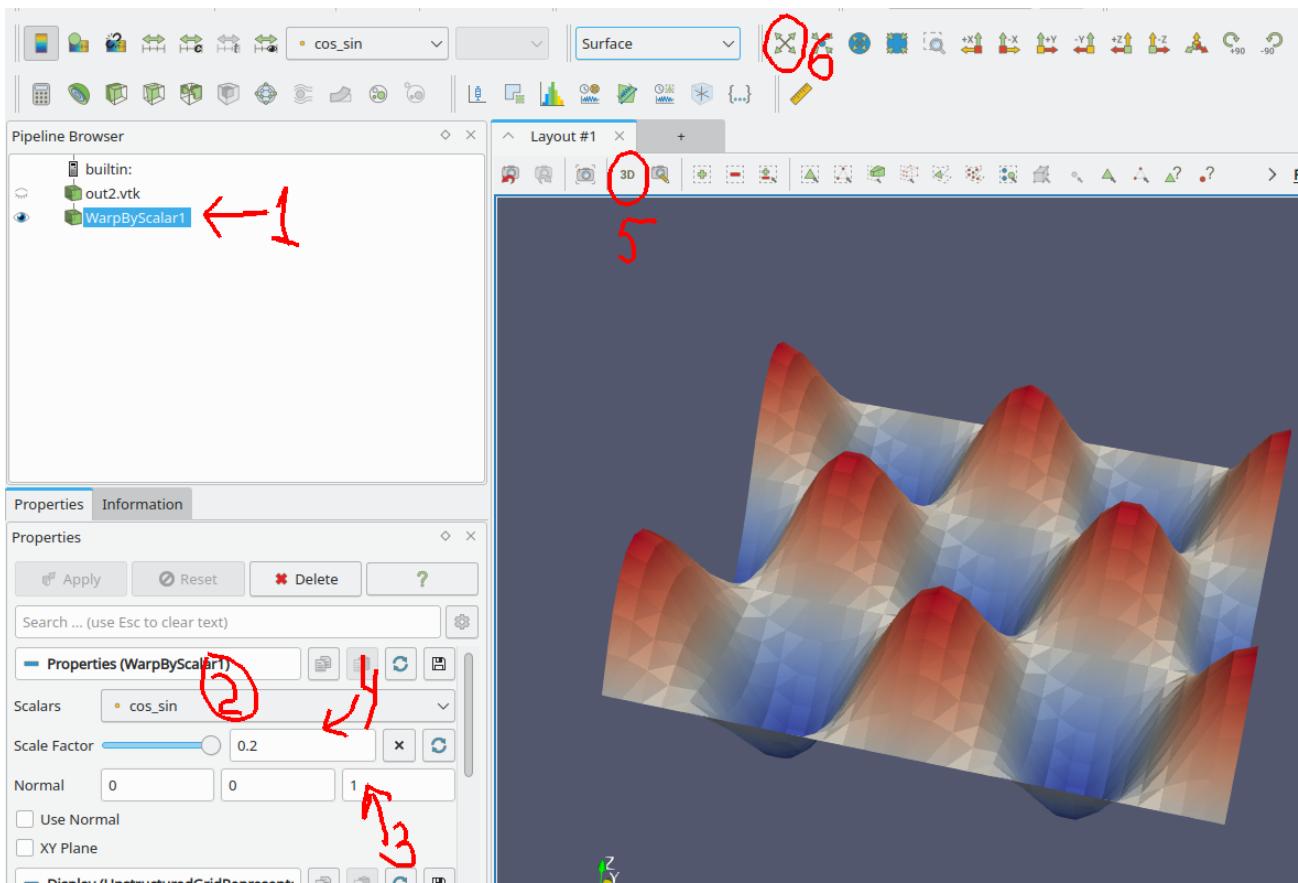
**Задание цвета и толщины изолинии** В случае, если нужно сделать изолинии одного цвета, установите поле **Coloring/Solid color** в настройках фильтра. Там же в меню **Edit** можно выбрать цвет. Для установления толщины линии включите подробные настройки и найдите там опцию **Styling/Line Width**.



### B.3.3 Данные на двумерных сетках в виде поверхности

По аналогии с одномерным графиком (п. B.3.1), двумерные поля так же можно отобразить, проектируя данные на геометрическую координату для получения объёмного графика. Для этого

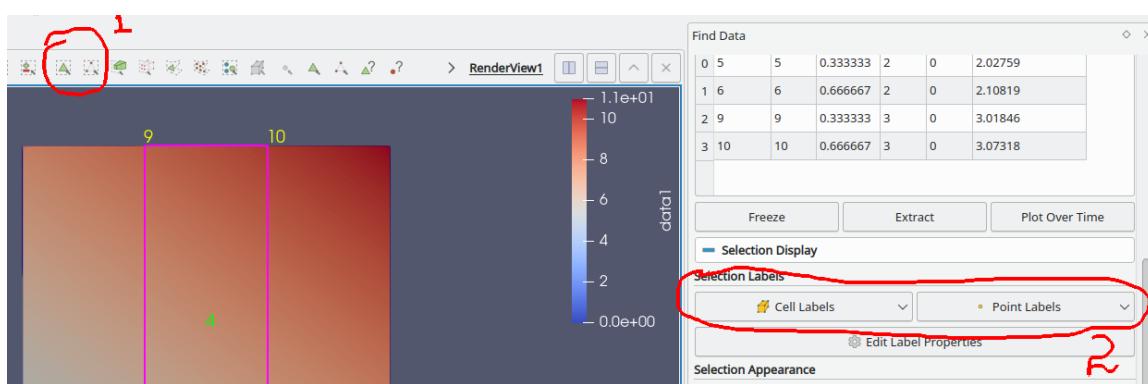
1. Включите фильтр **Filters/Warp By Scalar**
2. В настройках фильтра установите данные, которые будут проектироваться на координату z
3. Установите нормаль для проецирования (ось z)
4. Если нужно, выберите масштабирования для этой координаты
5. После нажатия **Apply** включите трёхмерное отображение
6. Если данные не видно, обновите экран.



### B.3.4 Числовых значений в точках и ячейках

Иногда в процессе отладки или анализа результатов расчёта требуется знать точное значение поля в заданном узле или ячейке сетки. Для этого

1. Включить режим выделения точек или ячеек (иконка (1 на рисунке) или горячие клавиши **s**, **d**). Выделить мышкой интересующую область
2. В окне **Find data** (или **Selection Inspector** для старых версий Paraview) отметить поле, которое должно отображаться в центрах ячеек и в точках (2 на рисунке). Если такого окна нет, включить его из основного меню **View**.

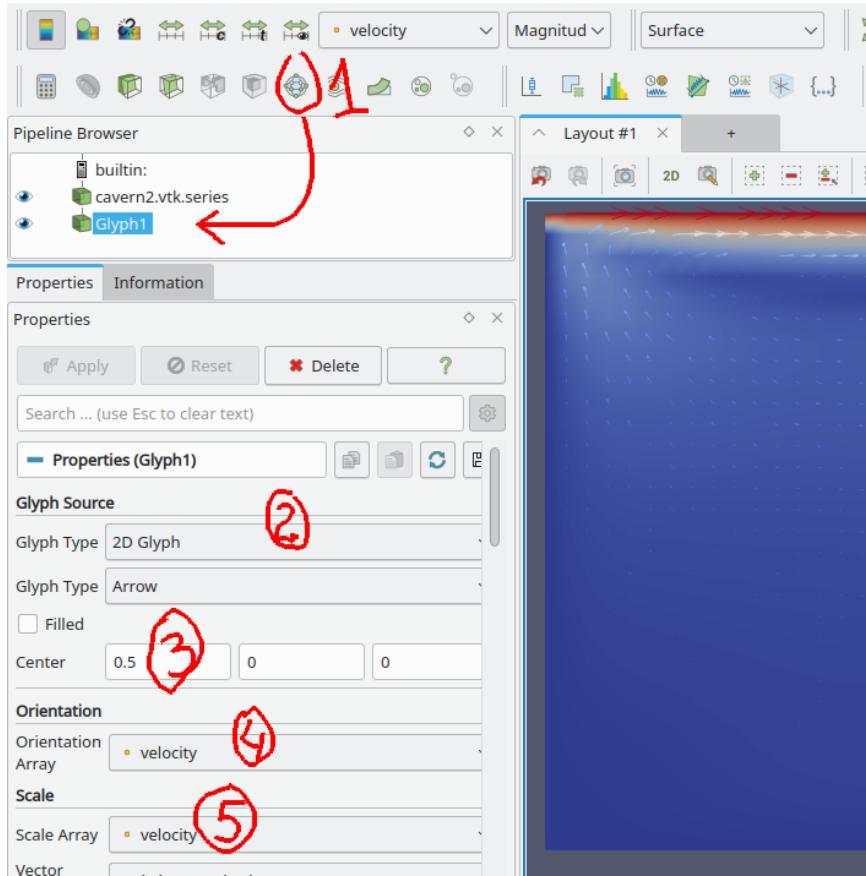


### B.3.5 Векторные поля

Открыть файл vtk или vtk.series, который содержит векторное поле. Далее

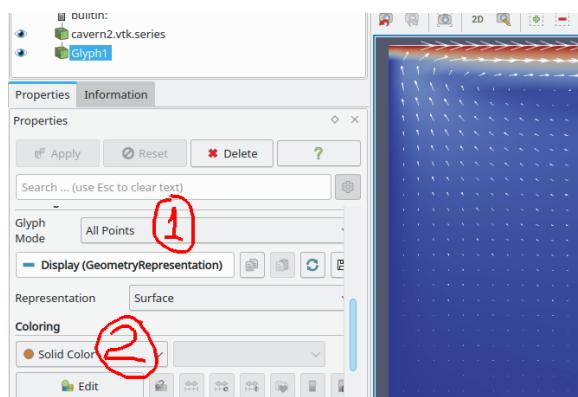
1. Создать фильтр **Glyph**

2. Задать двумерный тип стрелки
3. Сместить центр стрелки, чтобы она исходила из точки, к которой приписана
4. Отметить необходимое векторное поле в качестве ориентации
5. Отметить необходимое векторное поле для масштабирования Нажать **Apply**.



## Настройка отображения стрелок

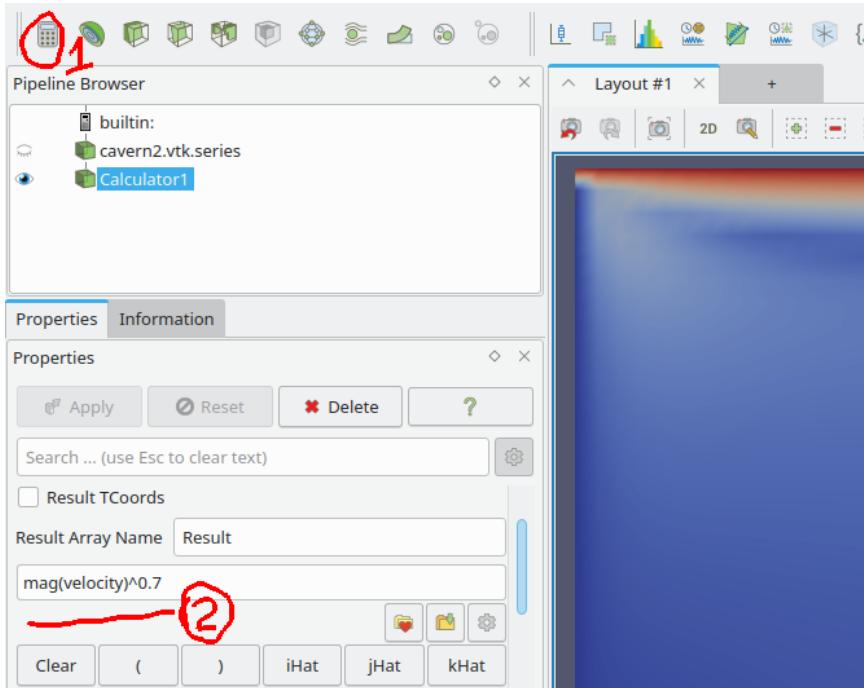
1. Выбрать необходимый **Glyph-mode**. Если сетка небольшая, то можно **All Points**.
2. Установить белый цвет для стрелок. Нажать **Apply**.



**Уменьшения разброса по длине стрелок** Если разброс по длинам стрелок слишком велик, его можно подправлять, введя новую функцию  $|v|^\alpha$  – длина вектора в степени меньше единицы (например,  $\alpha = 0.7$ ). Такую функцию можно создать через калькулятор

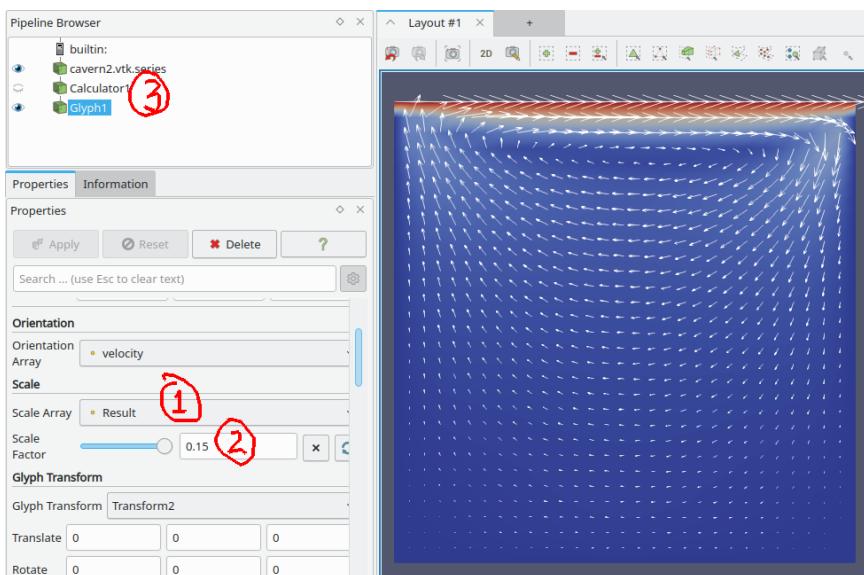
1. Начиная от загруженного файла создать фильтр **Calculator**

2. Там вбить необходимую формулу



Созданную функцию нужно прокинуть в **Glyph** в качестве коэффициента масштабирования

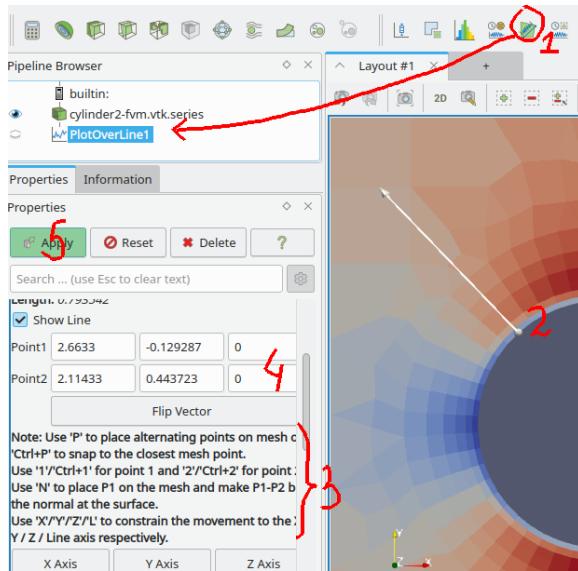
1. В **Scale Array** фильтра **Glyph** указать уже результат работы **Calculator**-а (**Result** по умолчанию),
2. Подтянуть значение **Scale Factor** до приемлимого
3. Не забыть отключить вспомогательное поле **Calculator** из отображения



### B.3.6 Значение функции вдоль линии

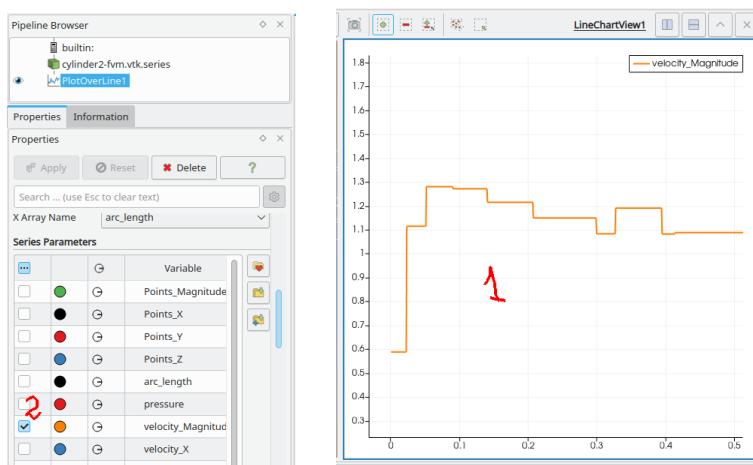
1. Выбрать фильтр **Plot Over Line** иконкой или в меню **Filters**
2. Установить начальную и конечную точку сечения

3. Можно использовать привязку к узлам сетки с помощью горячих клавиш (в подсказках написано)
4. Можно установить координаты руками в соответствующем поле. Для двумерных задач проследить, что координата Z равна нулю
5. Нажать **Apply**



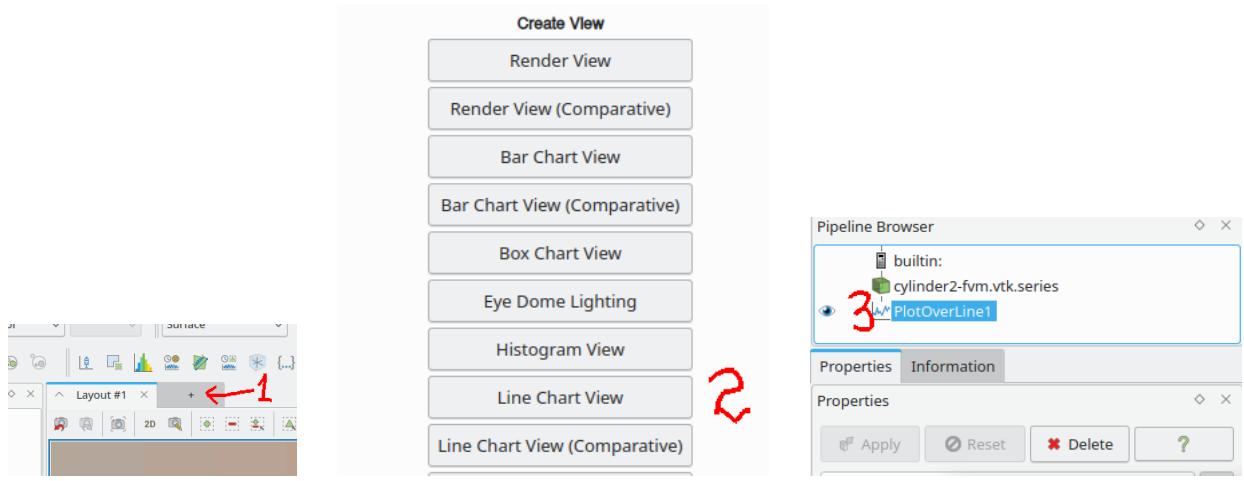
## Настройка графика

1. После установок появится дополнительное окно типа **Line Chart View** с нарисованным графиком.
2. Сделав это окно активным в настройках фильтра **PlotOverLine** можно выбрать, какие поля рисовать (**Series Parameters**)



## Отрисовка в отдельном окне

1. Открыть новую вкладку
2. Выбрать **Line Chart View**
3. Выбрать предварительно созданный фильтр с одномерным графиком



## B.4 Hybmesh

Генератор сеток на основе композитного подхода. Работает на основе python-скриптов. Полная документация <http://kalininei.github.io/HybMesh/index.html>

### B.4.1 Работа в Windows

Инсталлятор программы следует скачать по ссылке <https://github.com/kalininei/HybMesh/releases> и установить стандартным образом.

Для запуска скрипта построения `script.py` нужно открыть консоль, перейти в папку с нужным скриптом, оттуда выполнить (при условии, что программа была установлена в папку `C:\Program Files`):

```
> "C:\Program Files\HybMesh\bin\hybmesh.exe" -sx script.py
```

### B.4.2 Работа в Linux

Версию для линукса нужно собирать из исходников. Либо, если собрать не получилось, можно строить сетки в Windows и переносить полученные vtk-файлы на рабочую систему.

Перед сборкой в систему необходимо установить dev-версии пакетов `suitesparse` и `libxml2`. Также должны быть доступны компиляторы

`gcc-c++` и `gcc-fortan` и `cmake`. Программа работает со скриптами `python2`. Лучше установить среду anaconda (<https://docs.anaconda.com/free/anaconda/install/index.html>) И в ней создать окружение с `python-2.7`:

```
> conda create -n py27 python=2.7      # создать среду с именем py27
> conda activate py27                  # активировать среду py27
> pip install decorator              # установить пакет decorator
```

Сначала следует склонировать репозиторий в папку с репозиториями гита:

```
> cd D:/git_repos
> git clone https://github.com/kalininei/HybMesh
```

Поскольку программа не предназначена для запуска из под анаконды, в сборочные скрипты нужно внести некоторые изменения. В корневом сборочном файле `HybMesh/CMakeLists.txt` нужно закомментировать все строки в диапазоне

```
# ===== Python check
...
# ===== Windows installer options
```

а в файле `HybMesh/src/CMakeLists.txt` последнюю строку

```
#add_subdirectory(bindings)
```

Далее, находясь в корневой директории репозитория HybMesh, запустить сборку

```
> mkdir build  
> cd build  
> cmake .. -DCMAKE_BUILD_TYPE=Release  
> make -j8  
> sudo make install
```

Для запуска скриптов нужно создать скрипт-прокладку

```
import sys  
sys.path.append("/path/to/HybMesh/src/py/") # вставить полный путь к Hybmesh/src/py  
execfile(sys.argv[1])
```

и сохранить его в любое место. Например в `path/to/HybMesh/hybmesh.py`.

Для запуска скрипта построения сетки следует перейти в папку, где находится нужный скрипт `script.py`, убедится, что анаконда работает в нужной среде (то есть `conda activate py27` был вызван), и запустить

```
> python /path/to/HybMesh/hybmesh.py script.py
```