

Содержание

1 Лекция 1 (10.02)	3
1.1 Задание для самостоятельной работы	3
2 Лекция 2 (17.02)	4
2.1 Расчётная сетка	4
3 Лекция 3 (24.02)	5
3.1 Структурированная расчётная сетка	5
3.2 Метод конечных разностей. Уравнение Пуассона	5
3.2.1 Постановка задачи	5
3.2.2 Метод решения	5
3.2.2.1 Нахождение численного решения	5
3.2.2.2 Практическое определения порядка аппроксимации	6
3.2.3 Программная реализация	7
3.2.3.1 Функция верхнего уровня	7
3.2.3.2 Детали реализации	8
3.3 Задание для самостоятельной работы	11
3.3.1 Одномерное уравнение Пуассона	11
3.3.2 Двумерное уравнение Пуассона	11
4 Лекция 4 (02.03)	13
4.1 Форматы хранения разреженных матриц	13
4.1.1 CSR-формат	13
4.1.2 Массив словарей	15
5 Лекция 5 (09.03)	17
5.1 Решение СЛАУ	17
5.1.1 Метод Якоби	17
5.1.2 Метод Зейделя	17
5.1.3 Метод последовательных верхних релаксаций (SOR)	18
5.2 Задание для самостоятельной работы	18
6 Лекция 6 (23.03)	22
6.1 Метод конечных объёмов	22
6.1.1 Уравнение Пуассона	22
6.1.1.1 Обработка внутренних граней	23
6.1.1.2 Учёт граничных условий первого рода	24
6.1.2 Одномерный случай	24
6.1.3 Сборка системы линейных уравнений	25
6.1.3.1 Алгоритм сборки в цикле по ячейкам	25
6.1.3.2 Алгоритм сборки в цикле по граням	26

6.2	Задание для самостоятельной работы	26
7	Лекция 7 (30.03)	28
7.1	Граничные условия второго рода	28
7.2	Граничные условия третьего рода	28
7.2.1	Универсальность условий третьего рода	29
7.3	Задание для самостоятельной работы	29
8	Лекция 8 (06.04)	30
8.1	Дополнительные точки коллокации на границах	30
8.1.1	Пример	31
8.2	Задание для самостоятельной работы	32
9	Лекция 9 (13.04)	34
9.1	Учёт скопленности сетки в двумерной МКО-аппроксимации	34
9.1.1	Уточнённая аппроксимация нормальной производной	34
9.1.2	Интерполяция значения функции во вспомогательных точках	35
9.1.3	Производная по границе	36
9.1.4	Сборка СЛАУ для уравнения Пуассона	36
9.2	Задание для самостоятельной работы	37
10	Лекция 10 (20.04)	39
10.1	Учёт скопленности сетки в 3D	39
11	Лекция 11 (27.04)	40
11.1	Метод взвешенных невязок	40
11.2	Метод Бубнова–Галёркина	40
11.2.1	Степенные базисные функции	40
11.3	Метод конечных элементов	40
11.3.1	Узловые базисные функции	40
11.3.2	Одномерное уравнение Пуассона	40
11.3.2.1	Слабая интегральная постановка задачи	40
11.3.2.2	Линейный одномерный (пирамидальный) базис	40
12	Лекция 12 (04.05)	41
12.1	Поэлементная сборка конечноэлементных матриц	41
12.1.0.1	Элементные матрицы	41
13	Лекция 13 (11.05)	42
13.1	Вычисление элементных интегралов в параметрическом пространстве	42
13.2	Двумерное уравнение Пуассона	42
13.2.0.1	Треугольный элемент. Линейный двумерный базис	42
13.3	Разбор программной реализации МКЭ	42
13.3.1	Рабочий объект	43

13.3.2 Конечноэлементный сборщик	44
13.3.3 Концепция конечного элемента	44
13.3.3.1 Определение линейного одномерного элемента	45
13.3.3.2 Геометрические свойства элемента	46
13.3.3.3 Элементный базис	47
13.3.3.4 Калькулятор элементных матриц	48
13.4 Задание для самостоятельной работы	49
14 Лекция 14 (02.11)	50
14.1 Двухслойные схемы для нестационарных уравнений	50
14.1.1 Определение	50
14.1.1.1 Явная схема	50
14.1.1.2 Неявная схема	50
14.1.1.3 Схема Кранка–Николсон	51
14.1.1.4 Обобщённая двухслойная схема	51
14.2 Схемы высокого порядка точности	52
14.2.1 Многослойные схемы. Схемы Адамса	52
14.2.1.1 Явные схемы Адамса–Башфорта	52
14.2.1.2 Неявные схемы Адамса–Мультона	53
14.2.2 Схемы Рунге–Кутта	54
14.3 Методы исследования устойчивости расчётных схем	54
14.3.1 Дискретизация по времени как итерационный процесс	54
14.3.1.1 Двухслойный итерационный процесс	54
14.3.1.2 Устойчивость итерационного процесса	54
14.3.1.3 Источники возмущений	55
14.3.2 Матричный метод	56
14.3.2.1 Явная схема для нестационарного уравнения диффузии	56
14.3.2.2 Неявная схема для нестационарного уравнения диффузии	57
14.3.3 Метод дискретных возмущений	58
14.3.3.1 Явная схема против потока для уравнения переноса	58
14.3.4 Метод Неймана	58
14.3.4.1 Неявная противопотоковая схема для уравнения переноса	59
14.3.4.2 Противопотоковая схема Кранка–Николсон для уравнения переноса	60
14.3.4.3 Явная схема для уравнения нестационарной конвекции-диффузии	61
14.3.4.4 Неявная схема для уравнения нестационарной конвекции-диффузии	62
14.3.5 Общие рекомендации к выбору устойчивых расчётных схем	62
14.4 Программная реализация схемы для уравнения переноса	63
14.4.1 Постановка задачи	63
14.4.2 Функция верхнего уровня	64
14.4.3 Расчётные функции	66
14.4.3.1 Явная схема	68
14.4.3.2 Неявная схема	68

14.4.3.3 Схема Кранка-Николсон	69
14.4.4 Анализ результатов работы	69
14.5 Задание для самостоятельной работы	70
14.5.1 Постановка задачи	70
14.5.1.1 Тестовый пример 1	70
14.5.1.2 Тестовый пример 2	71
14.5.2 Расчёчная схема	72
15 Лекция 15 (09.11)	74
15.1 Аппроксимация уравнения переноса с ограничением потока	74
15.1.1 Схемы первого и второго порядка точности	74
15.1.2 Условие TVD	75
15.1.3 Нелинейные TVD схемы	75
15.2 TVD-схемы для неструктурированных конечнообъёмных сеток	77
15.2.1 Прямая интерполяция противопоточного значения	78
15.2.2 Интерполяция противопоточного значения через значение градиентов	78
15.2.3 Определение градиентов в узлах коллокации. Метод наименьших квадратов	79
15.2.4 Реализация для явной схемы	80
15.3 Задание для самостоятельной работы	81
16 Лекция 16 (16.11)	82
16.1 Алгебраический подход к построению нелинейных TVD схем	82
16.2 Схемы с искусственной вязкостью	82
16.2.1 Направленная искусственная вязкость	82
16.2.2 SUPG	82
16.3 Задание для самостоятельной работы	82
17 Лекция 17 (23.11)	84
17.1 Моделирование течения вязкой несжимаемой жидкости методом конечных разностей	84
17.1.1 Система уравнений Навье-Стокса	84
17.1.2 Схема расчёта	85
17.1.2.1 Метод SIMPLE	85
17.1.3 Пространственная аппроксимация	87
17.1.3.1 Разнесённая сетка	87
17.1.3.2 Уравнения движения	88
17.1.3.3 Уравнение для поправки давления	90
17.1.3.4 Уравнение для поправки скорости	92
17.1.3.5 Учёт граничных условий	92
17.1.4 Оптимальные значения параметров алгоритма SIMPLE	95
17.2 Программа для расчёта течения в каверне по схеме SIMPLE	95
17.2.1 Постановка задачи	95
17.2.2 Функция верхнего уровня	97
17.2.3 Поля класса решателя	97

17.2.4	Инициализация решателя	98
17.2.5	Шаг итерации SIMPLE	98
17.2.6	Сборка системы уравнений для поправки давления	98
17.2.7	Сборка системы уравнений для пробной скорости	99
17.3	Задание для самостоятельной работы	99
A	Формулы и обозначения	100
A.1	Векторы	101
A.1.1	Обозначение	101
A.1.2	Набла–нотация	101
A.2	Интегрирование	103
A.2.1	Формула Гаусса–Остроградского	103
A.2.2	Интегрирование по частям	103
A.2.3	Численное интегрирование в заданной области	104
A.3	Интерполяционные полиномы	105
A.3.1	Многочлен Лагранжа	105
A.3.1.1	Узловые базисные функции	105
A.3.1.2	Интерполяция в параметрическом отрезке	106
A.3.1.3	Интерполяция в параметрическом треугольнике	109
A.3.1.4	Интерполяция в параметрическом квадрате	111
A.4	Геометрические алгоритмы	114
A.4.1	Линейная интерполяция	114
A.4.2	Преобразование координат	114
A.4.2.1	Матрица Якоби	115
A.4.2.2	Дифференцирование в параметрической плоскости	116
A.4.2.3	Интегрирование в параметрической плоскости	117
A.4.2.4	Двумерное линейное преобразование. Параметрический треугольник .	117
A.4.2.5	Двумерное билинейное преобразование. Параметрический квадрат .	118
A.4.2.6	Трёхмерное линейное преобразование. Параметрический тетраэдр .	118
A.4.3	Свойства многоугольника	118
A.4.3.1	Площадь многоугольника	118
A.4.3.2	Интеграл по многоугольнику	120
A.4.3.3	Центр масс многоугольника	120
A.4.4	Свойства многогранника	121
A.4.4.1	Объём многогранника	121
A.4.4.2	Интеграл по многограннику	121
A.4.4.3	Центр масс многогранника	121
A.4.5	Поиск многоугольника, содержащего заданную точку	121
B	Работа с инфраструктурой проекта CFDCourse	122
B.1	Сборка и запуск	123
B.1.1	Сборка проекта CFDCourse	123
B.1.1.1	Подготовка	123

B.1.1.2	VisualStudio	123
B.1.1.3	VSCode	125
B.1.2	Запуск конкретного теста	126
B.1.3	Сборка релизной версии	128
B.2	Git	130
B.2.1	Основные команды	130
B.2.2	Порядок работы с репозиторием CFDCourse	131
B.3	Paraview	133
B.3.1	Данные на одномерных сетках	133
B.3.2	Изолинии для двумерного поля	136
B.3.3	Данные на двумерных сетках в виде поверхности	137
B.3.4	Числовых значения в точках и ячейках	138
B.3.5	Векторные поля	138
B.3.6	Значение функции вдоль линии	140
B.4	Hybmesh	143
B.4.1	Работа в Windows	143
B.4.2	Работа в Linux	143

1 Лекция 1 (10.02)

1.1 Задание для самостоятельной работы

1. Клонировать репозиторий `CFDCourse25` на компьютер, собрать проект и запустить программу `cfd25_test` (см. пункт [B.1](#))
2. В файле `cfd25_test.cpp` написать простой `TEST_CASE`, проверяющий результат простого арифметического действия (например, $2 + 2$). О запуске тестов см. пункт [B.1.2](#)

2 Лекция 2 (17.02)

2.1 Расчётная сетка

TODO

3 Лекция 3 (24.02)

3.1 Структурированная расчётная сетка

TODO

3.2 Метод конечных разностей. Уравнение Пуассона

3.2.1 Постановка задачи

Рассматривается одномерное дифференциальное уравнение вида

$$-\frac{\partial^2 u}{\partial x^2} = f(x) \quad (3.1)$$

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

$$\begin{cases} u(a) = u_a, \\ u(b) = u_b. \end{cases} \quad (3.2)$$

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.

3.2.2 Метод решения

3.2.2.1 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 (3.1) имеет вид

$$\frac{-u_{i-1} + 2u_i - u_{i+1}}{h^2} = f_i, \quad i = \overline{1, N-1}. \quad (3.3)$$

Здесь $\{f_i\}$ – известный сеточный вектор, определяемый через известную аналитическую функцию $f(x)$ в правой части уравнения (3.1) как

$$f_i = f(x_i). \quad (3.4)$$

Аппроксимация граничных условий (3.2) первого рода даёт дополнительные сеточные уравнения

для граничных узлов

$$\begin{aligned} u_0 &= u_a, \\ u_N &= u_b \end{aligned} \quad (3.5)$$

Линейные уравнения (3.3), (3.5) составляют систему вида

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

с матричными коэффициентами

$$A_{ij} = \begin{cases} 1, & i = 0, j = 0; \\ 2/h^2, & i = \overline{1, N-1}, j = i; \\ -1/h^2, & i = \overline{1, N-1}, j = i-1; \\ -1/h^2, & i = \overline{1, N-1}, j = i+1; \\ 1, & i = N, j = N; \\ 0, & \text{иначе.} \end{cases} \quad (3.6)$$

и правой частью

$$b_i = \begin{cases} u_a, & i = 0; \\ u_b, & i = N; \\ f_i, & i = \overline{1, N-1}. \end{cases} \quad (3.7)$$

Искомый вектор находится путём решения этой системы.

3.2.2.2 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 (3.1), (3.2).

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

$$\|y(\mathbf{x})\|_2 = \sqrt{\frac{1}{|D|} \int_D y(\mathbf{x})^2 d\mathbf{x}}. \quad (3.8)$$

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

$$\|\{u_i - u^e(x_i)\}\|_2 = \sqrt{\frac{1}{b-a} \sum_{i=0}^N w_i (u_i - u_i^e)^2}. \quad (3.9)$$

3.2.3 Программная реализация

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

В качестве аналитической тестовой функции используется

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

на отрезке $x \in [0, 1]$.

3.2.3.1 Функция верхнего уровня

объявлена как

```
113 TEST_CASE("Poisson 1D solver, Finite Difference Method", "[poisson1-fdm]") {
```

В программе в цикле по набору разбиений `n_cells`

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

создаётся решатель для тестовой задачи, использующий заданное число ячеек

```
127     TestPoisson1Worker worker(n_cells);
```

вычисляется среднеквадратичная норма отклонения численного решения от точного

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

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

```
poisson1_n={10,20,...}.vtk
```

```
133     worker.save_vtk("poisson1_fdm_n=" + std::to_string(n_cells) + ".vtk");
```

а полученная норма печатается в консоль напротив количества ячеек

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

В результате работы программы в консоли должна отобразиться таблица вида

```
--- [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).

3.2.3.2 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									

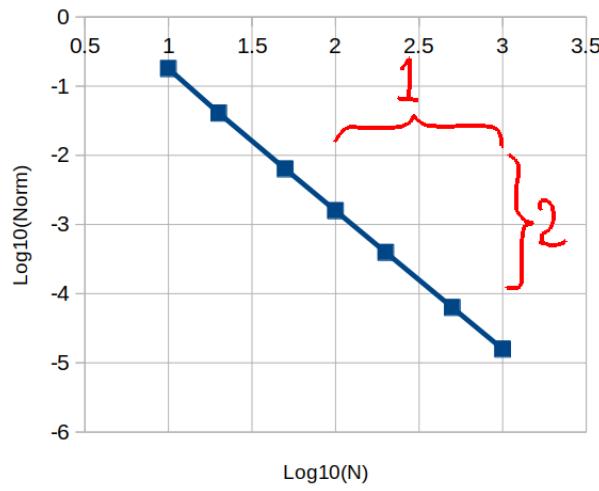


Рис. 1: Сходимость с уменьшением разбиения при решении одномерного уравнения Пуассона

```
15 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.

```
30 double solve(){
31     // 1. build SLAE
32     CsrMatrix mat = approximate_lhs();
33     std::vector<double> rhs = approximate_rhs();
34
35     // 2. solve SLAE
36     AmgMatrixSolver solver;
37     solver.set_matrix(mat);
38     solver.solve(rhs, u);
39
40     // 3. compute norm2
41     return compute_norm2();
42 }
```

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

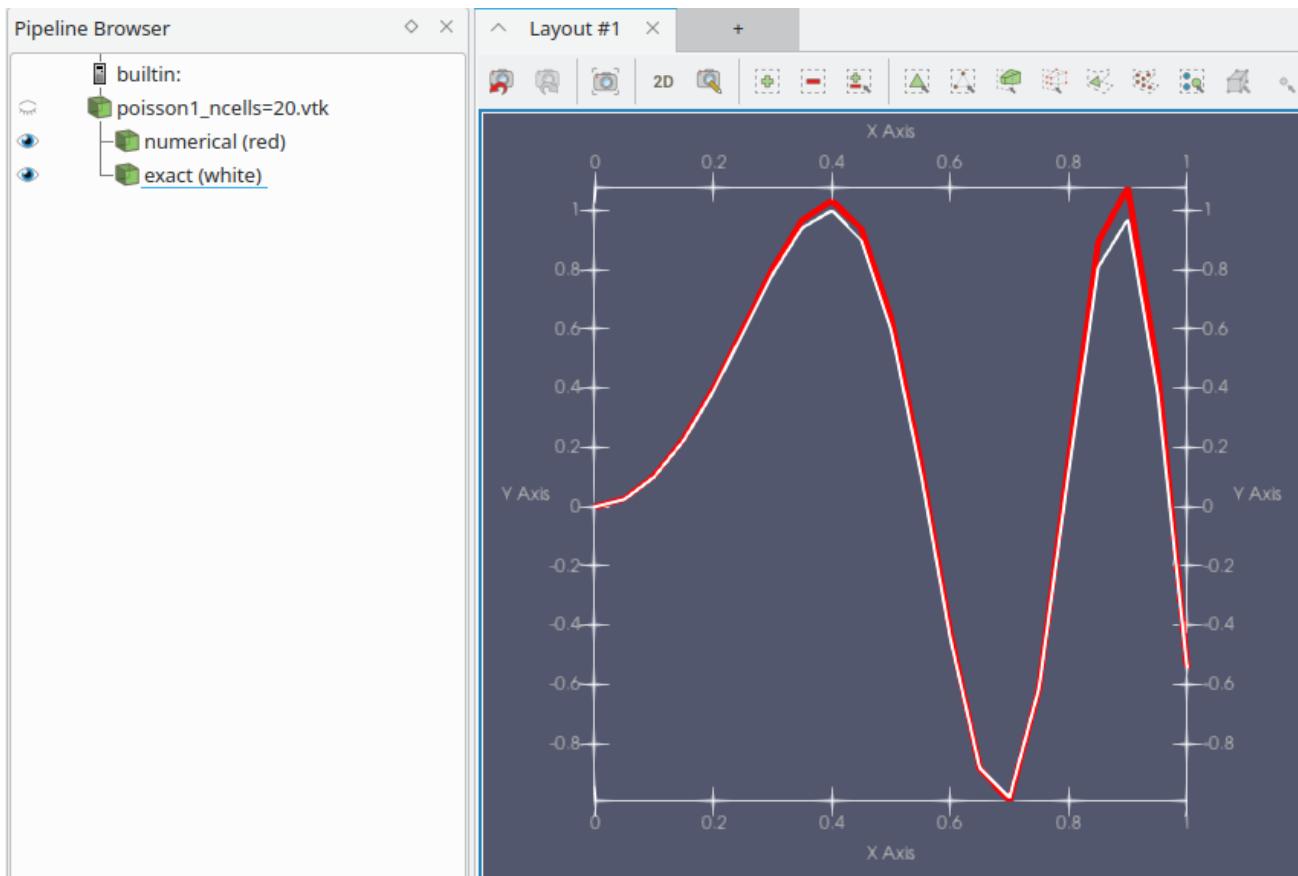


Рис. 2: Сравнение точного и численного решений уравнения Пуассона

- Сборка левой части СЛАУ. Реализует формулу (3.6). Для заполнения матрицы используется формат

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

```

64    CsrMatrix approximate_lhs() const{
65        // constant h = x[1] - x[0]
66        double h = grid.point(1).x() - grid.point(0).x();
67
68        // fill using 'easy-to-construct' sparse matrix format
69        LodMatrix mat(grid.n_points());
70        mat.add_value(0, 0, 1);
71        mat.add_value(grid.n_points()-1, grid.n_points()-1, 1);
72        double diag = 2.0/h/h;
73        double nondiag = -1.0/h/h;
74        for (size_t i=1; i<grid.n_points()-1; ++i){
75            mat.add_value(i, i-1, nondiag);
76            mat.add_value(i, i+1, nondiag);
77            mat.add_value(i, i, diag);
78        }
79

```

```

80     // return 'easy-to-use' sparse matrix format
81     return mat.to_csr();
82 }
```

- Сборка правой части СЛАУ. Реализует формулу (3.7).

```

84     std::vector<double> approximate_rhs() const{
85         std::vector<double> ret(grid.n_points());
86         ret[0] = exact_solution(grid.point(0).x());
87         ret[grid.n_points()-1] = exact_solution(grid.point(grid.n_points()-1).x());
88         for (size_t i=1; i<grid.n_points()-1; ++i){
89             ret[i] = exact_rhs(grid.point(i).x());
90         }
91         return ret;
92     }
```

- Вычисление нормы. Реализует формулу (3.9).

```

94     double compute_norm2() const{
95         // weights
96         double h = grid.point(1).x() - grid.point(0).x();
97         std::vector<double> w(grid.n_points(), h);
98         w[0] = w[grid.n_points()-1] = h/2;
99
100        // sum
101        double sum = 0;
102        for (size_t i=0; i<grid.n_points(); ++i){
103            double diff = u[i] - exact_solution(grid.point(i).x());
104            sum += w[i]*diff*diff;
105        }
106
107        double len = grid.point(grid.n_points()-1).x() - grid.point(0).x();
108        return std::sqrt(sum / len);
109    }
```

3.3 Задание для самостоятельной работы

3.3.1 Одномерное уравнение Пуассона

Скомпилировать и запустить программу, описанную в п. 3.2.3. Построить график полученного численного и точного решения, аналогичный рис. 2 (инструкцию по построению одномерного графика решения в Paraview см. в п B.3.1).

Построить график, подтверждающий второй порядок точности разностной схемы (3.3).

3.3.2 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

$$k[i, j] = i + (n_x + 1)j \quad (3.10)$$

– 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:

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

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

$$\begin{aligned} i[k] &= \text{mod}(k, (n_x + 1)), & // \text{ остаток от деления,} \\ j[k] &= \lfloor k / (n_x + 1) \rfloor, & // \text{ целая часть от деления.} \end{aligned} \quad (3.11)$$

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).

Для иллюстрации двумерного решения в Paraview использовать изолинии ([B.3.2](#)) и трёхмерные поверхности ([B.3.3](#)).

Построить график сходимости для двумерного случая. 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 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 $n = \sqrt{N}$.

4 Лекция 4 (02.03)

4.1 Форматы хранения разреженных матриц

4.1.1 CSR-формат

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

$$\begin{pmatrix} 2.0 & 0 & 0 & 1.0 \\ 0 & 3.0 & 5.0 & 4.0 \\ 0 & 0 & 6.0 & 0 \\ 0 & 7.0 & 0 & 8.0 \end{pmatrix} \quad (4.1)$$

Массивы, описывающие матрицу в формате CSR примут вид

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

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;
}

```

Формат CSR обеспечивает максимальную компактность хранения разреженной матрицы и при этом удобен для последовательной итерации по элементам матрицы (операции умножения матрицы на вектор), но его существенным недостатком является высокая сложность добавления нового элемента в шаблон.

4.1.2 Массив словарей

При реализации сеточных методов решения дифференциальных уравнений работу с матрицами можно разбить на два этапа: сборка матриц и их непосредственное использование. Сборка матрицы в свою очередь может быть разделена на этап вычисление шаблона матрицы (символьная сборка) и непосредственное вычисление коэффициентов матрицы (числовая сборка).

На этапе использования матрицы основной операцией является умножение матрицы на вектор, где наиболее эффективным является CSR-формат.

В случае использования неструктурированных сеток этап символьной сборки является нетривиальной операцией и сводится к неупорядоченному добавлению новых элементов в шаблон матрицы. Как было отмечено ранее, такая операция в случае использования CSR формата неэффективна.

Поэтому часто для этапов сборки и расчёта используют разные форматы хранения матриц, первый из которых оптимизирован для операции вставки, а второй – для операции умножения на вектор. В качестве формата, оптимизированного для вставки, можно представить формат массива словарей (List of dictionaries), где каждая строка матрицы описывается словарём, ключём которого является индекс колонки, а значением – величина соответствующего матричного коэффициента.

С использованием синтаксиса C++ такой формат может быть описан следующим образом:

```
std::vector<std::map<size_t, double>> data;
```

Матрица вида (4.1) в таком формате примет вид

```
data = {  
    {0: 2.0, 3: 1.0},  
    {1: 3.0, 2: 5.0, 3: 4.0},  
    {2: 6.0},  
    {1: 7.0, 3: 8.0}  
};
```

Добавление нового матричного коэффициента сведётся к вставке элемента в словарь:

```
data[i][j] = value;
```

А основной операцией для такого формата будет служить конверсия в CSR:

```
std::vector<size_t> addr{0};  
std::vector<size_t> cols;  
std::vector<double> vals;  
for (size_t irow=0; irow < data.size(); ++irow){  
    for (auto it: data[irow])  
        cols.push_back(it.first);  
        vals.push_back(it.second);  
    }  
    addr.push_back(addr.back() + data[irow].size());  
}
```

Поскольку данные в контейнере типа

`std::map` итерируются в отсортированном по ключам порядке, то полученный в результате массив `cols` также является локально отсортированным.

5 Лекция 5 (09.03)

5.1 Решение СЛАУ

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.

5.1.1 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

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

5.1.2 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

```

5.1.3 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 $\omega < 2$. В частности, для одномерных задач, заданных на единичном отрезке, для оптимальной сходимости можно использовать соотношение $\omega \approx 2 - 5h$, где h – шаг сетки.

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

```

5.2 Задание для самостоятельной работы

Вернемся к рассмотрению двумерного уравнения Пуассона (п. 3.3.2). Прошлая реализация этой задачи включала в себя решение СЛАУ алгебраическим многосеточным методом с помощью класса `AmgMatrixSolver`:

```

AmgMatrixSolver solver;
solver.set_matrix(mat);
solver.solve(rhs, u);

```

Необходимо реализовать рассмотренные ранее методы итерационного решения СЛАУ

- метод Якоби (5.1.1),

- метод Зейделя (5.1.2),
- метод SOR (5.1.3).

и использовать их вместо многосеточного решателя.

Реализовать означенные решатели нужно 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.

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
    ...
}
```

Дополнительно понадобится реализовать функцию, которая проверяет сходимость решения путём вычисления невязки вида

$$res = \max_i \left| \sum_j A_{ij} u_j - r_i \right|$$

и сравнения с заданным малым числом $\varepsilon = 10^{-8}$.

```
bool is_converged(const cfd::CsrMatrix& mat, const std::vector<double>& rhs, const
→ std::vector<double>& x){
    constexpr double EPS = 1e-8;
    double residual = 0;
    // ...
    return residual < EPS;
}
```

Для реализации вспомогательных функций необходимо использовать алгоритмы работы с CSR-матрицами из п. 4.1.1

При реализации метода SOR подобрать оптимальный параметр ω , при котором метод SOR сойдётся за минимальное число итераций.

После реализации всех методов необходимо сравнить время исполнения решателей. Замеры нужно проводить в Release-версии сборки (см. п. B.1.3). Для замера времени исполнения участка кода воспользоваться функциями

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

Код решения СЛАУ методом Якоби с вызовами профилировщика должен иметь примерно такой вид:

```
#include "dbg/tictoc.hpp"
using namespace cfd;

...
// реализация решения СЛАУ
dbg::Tic("total"); // запустить таймер total
for (size_t it=0; it < max_it; ++it){
    dbg::Tic("step"); // запустить таймер step
    jacobi_step(mat, rhs, u);
    dbg::Toc("step"); // остановить таймер step

    dbg::Tic("conv-check"); // запустить таймер conv-check
    bool is_conv = is_converged(mat, rhs, u);
    dbg::Toc("conv-check"); // остановить таймер conv-check

    if (is_conv) break;
}
dbg::Toc("total"); // остановить таймер total
```

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

```
total: 6.670 sec
step: 5.220 sec
conv-check: 1.210 sec
```

По результатам профилировки нужно заполнить таблицу

	total, s	step, s	conv-check, s	Кол-во итераций
Amg		—	—	—
Якоби				
Зейдель				
SOR($\omega = \dots$)				

Здесь Amg - исходный решатель.

6 Лекция 6 (23.03)

6.1 Метод конечных объёмов

6.1.1 Уравнение Пуассона

Пространственную аппроксимацию дифференциальных операторов методом конечных объёмов рассмотрим на примере многомерного уравнения Пуассона

$$-\nabla^2 u = f, \quad (6.1)$$

которое требуется решить в области D . Разобъём эту область на непересекающиеся подобласти E_i , $i = \overline{0, N - 1}$ (рис. 3). Центры ячеек обозначим как \mathbf{c}_i .

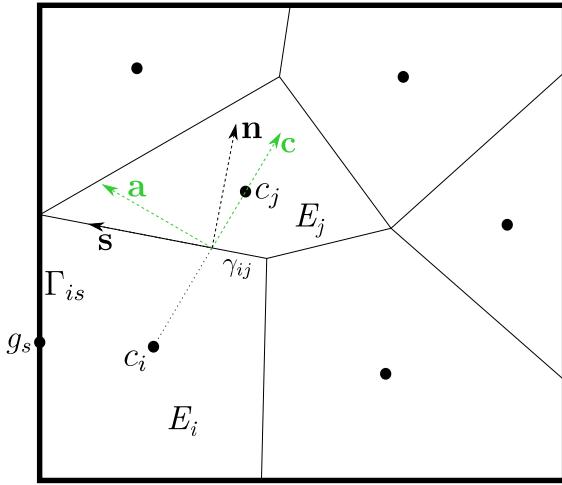


Рис. 3: Конечнообъёмная сетка

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

$$-\int_{\partial E_i} \frac{\partial u}{\partial n} \, ds = \int_{E_i} f \, d\mathbf{x}. \quad (6.2)$$

Здесь ∂E_i – совокупность всех границ подобласти E_i , а \mathbf{n} – внешняя к подобласти нормаль.

Граница ячейки E_i состоит из внутренних граней γ_{ij} (индекс j здесь соответствует индексу соседней ячейки) и граней Γ_{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

$$\int_{\partial E_i} \frac{\partial u}{\partial n} ds \approx \sum_j |\gamma_{ij}| \left(\frac{\partial u}{\partial n} \right)_{\gamma_{ij}} + \sum_s |\Gamma_{is}| \left(\frac{\partial u}{\partial n} \right)_{\Gamma_{is}} \quad (6.3)$$

Аналогично, анализируя интеграл правой части (6.2), приблизим значение функции правой части f внутри элемента E_i константой f_i , которую отнесём к центру элемента. Тогда

$$\int_{E_i} f d\mathbf{x} \approx f_i |E_i|. \quad (6.4)$$

Сеточный вектор $\{f_i\}$ – есть конечнообъёмная аппроксимация функции $f(\mathbf{x})$ на конечнообъёмную сетку. Значения f_i при аппроксимации чаще всего находятся как значения в центрах элементов

$$f_i = f(\mathbf{c}_i).$$

Хотя иногда может быть использовано и другое определение, следующее из (6.4):

$$f_i = \frac{1}{|E_i|} \int_{E_i} f(\mathbf{x}) d\mathbf{x}.$$

6.1.1.1 Обработка внутренних граней

Для начала будем рассматривать сетки, в которых вектора \mathbf{c} , соединяющие центры ячеек (зедёные вектора на рис. 3), коллинеарны (или почти коллинеарны) нормалям к граням \mathbf{n} . В этом случае производную искомой функции по нормали к грани можно записать в виде

$$\frac{\partial u}{\partial n} = \frac{\partial u}{\partial c}.$$

Далее определим значения функции u в точках c_i, c_j как u_i, u_j . Тогда значение производной $\partial u / \partial n$ на внутренней грани конечного объёма может быть приближена конечной разностью

$$\frac{\partial u}{\partial n} = \frac{\partial u}{\partial c} \approx \frac{u_j - u_i}{h_{ij}}, \quad h_{ij} = |\mathbf{c}_j - \mathbf{c}_i|. \quad (6.5)$$

Определим pebi (perpendicular-bisector) сетки как сетки, удовлетворяющие следующим свойствам

- линии, соединяющие центры двух соседних ячеек, перпендикулярны грани между этими ячейками;
- внутренние грани делят линии, соединяющие центры соседних ячеек, пополам.

Очевидно, что равномерная структурированная сетка удовлетворяет этим свойствам. Для построения неструктурных pebi-сеток используют алгоритмы построения ячеек Вороного. Для pebi-сеток разностная схема (6.5) является симметричной разностью и, поэтому, имеет второй порядок аппроксимации.

6.1.1.2 Учёт граничных условий первого рода

Для вычисления второго слагаемого в правой части (6.3) следует расписать значение нормальной к границе производной вида

$$\left(\frac{\partial u}{\partial n} \right)_{\Gamma_{is}}.$$

Это делается с помощью граничных условий.

Пусть на центре грани Γ_{is} задано значение искомой функции

$$\mathbf{x} \in \Gamma_{is} : \quad u(\mathbf{x}) = u^\Gamma. \quad (6.6)$$

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 аналогичной (6.5):

$$\frac{\partial u}{\partial n} \approx \frac{u^\Gamma - u_i}{h_{is}}, \quad h_{is} = |\mathbf{g}_s - \mathbf{c}_i|. \quad (6.7)$$

6.1.2 Одномерный случай

Рассмотрим результат конечнообъёмной аппроксимации задачи (6.1) в одномерном случае на равномерной сетке с шагом h (рис. 4).

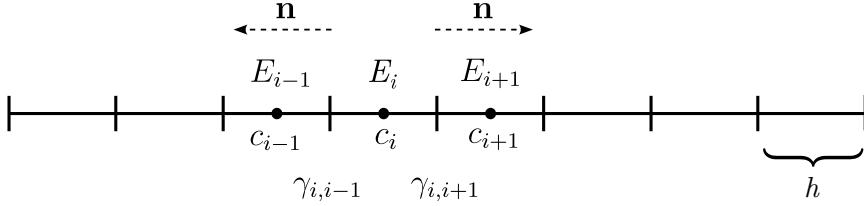


Рис. 4: Одномерная конечнообъёмная сетка

У внутренней ячейки i есть две грани: $\gamma_{i,i-1}$ и $\gamma_{i,i+1}$. Нормали по этим границам аппроксимируются по формулам ??:

$$\begin{aligned} \gamma_{i,i-1} : \quad & \frac{\partial u}{\partial n} = \frac{u_{i-1} - u_i}{h} \\ \gamma_{i,i+1} : \quad & \frac{\partial u}{\partial n} = \frac{u_{i+1} - u_i}{h} \end{aligned}$$

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 (6.2), 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 проявится в аппроксимации граничных условий. Так, если на левой границе задано условие первого рода, то соответствующее уравнение согласно (6.7) 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 $u_0 = u^\Gamma$.

6.1.3 Сборка системы линейных уравнений

Подставим все полученные аппроксимации (6.5), (6.7) в уравнение (6.2):

$$-\sum_j \frac{|\gamma_{ij}|}{h_{ij}} (u_j - u_i) - \sum_{s \in I} \frac{|\Gamma_{is}|}{h_{is}} (u^\Gamma - u_i) = f_i |E_i|.$$

Здесь первое слагаемое в левой части отвечает за потоки через внутренние границы, второе – граничные условия первого рода. Далее перенесём все известные значения в правую часть и окончательно получим линейное уравнение для i -го конечного объёма:

$$\sum_j \frac{|\gamma_{ij}|}{h_{ij}} (u_i - u_j) + \sum_{s \in I} \frac{|\Gamma_{is}|}{h_{is}} u_i = f_i |E_i| + \sum_{s \in I} \frac{|\Gamma_{is}|}{h_{is}} u^\Gamma \quad (6.8)$$

Таким образом мы получили систему из N (по количеству подобластей) линейных уравнений относительно неизвестного сеточного вектора $\{u_i\}$

$$Au = b.$$

6.1.3.1 Алгоритм сборки в цикле по ячейкам

Матрицу A и правую часть b системы (6.8) 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
endfor

```

Первым недостатком такого алгоритма является наличие вложенных циклов. Во-вторых, коэффициент, отвечающий за поток через внутреннюю грань γ_{ij} , равный $|\gamma_{ij}|/h_{ij}$ в таком алгоритме будет учитываться дважды: в строке i и в строке j .

6.1.3.2 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 удается 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

```

Первое слагаемое в правой части (6.8) учтём отдельным циклом:

```

for  $i = \overline{0, N - 1}$  – цикл по ячейкам
     $b_i = |E_i|f_i$ 
endfor

```

6.2 Задание для самостоятельной работы

В тесте `poisson1-fvm` из файла `poisson_fvm_solve_test.cpp` реализовано решение одномерного уравнения Пуассона с граничными условиями первого рода. Проводится расчёт на сгущающихся сетках с количеством ячеек от 10 до 1000 и рассчитываются среднеквадратичные нормы отклонения полученного численного решения от точного. Решения сохраняются в vtk-файлы `poisson1_fvm_n={}.vtk`.

Отталкиваясь от этой реализации необходимо:

1. написать аналогичный тест для двумерного уравнения и случая неструктурированных сеток,
2. провести серию расчётов на сгущающихся сетках разных типов (структурных, pebi и сконченных)
3. визуализировать решение, полученное на этих сетках
4. построить графики сходимости решения и определить порядок аппроксимации метода

Построение неструктурных 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 (сконченных) трех- и четырехугольников.

Инструкции по запуску этих скриптов смотри п. [B.4](#). Эти скрипты строят равномерную неструктурную 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.

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);
```

Рекомендации к программированию При написании новых тестов следует переиспользовать уже написанный код, избегая копирования. Для этого необходимо пользоваться механизмами наследования классов. В частности, следует обратить внимание, что все сетки наследуются от единого интерфейса `IGrid`. А уже написанный “одномерный” код в своей алгоритмической части использует только функции этого интерфейса.

7 Лекция 7 (30.03)

7.1 Граничные условия второго рода

Учёт условий второго рода тривиален. Если на центре грани Γ_{is} задано значение нормальной производной

$$\mathbf{x} \in \Gamma_{is} : \quad \frac{\partial u}{\partial n} = q(\mathbf{x}), \quad (7.1)$$

то это значение просто подставляется вместо соответствующей производной в (6.3).

По аналогии с (6.10) учёт граничных условий второго рода при сборке по граням будет иметь следующий вид

```

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

```

(7.2)

7.2 Граничные условия третьего рода

Теперь рассмотрим условия третьего рода

$$\mathbf{x} \in \Gamma_{is} : \quad \frac{\partial u}{\partial n} = \alpha(\mathbf{x})u + \beta(\mathbf{x}). \quad (7.3)$$

Распишем производную в форме (6.7):

$$\frac{u^\Gamma - u_i}{h_{is}} = \alpha u^\Gamma + \beta,$$

откуда выразим u^Γ :

$$u^\Gamma = \frac{u_i + \beta h_{is}}{1 - \alpha h_{is}}.$$

Подставляя это выражение в исходное граничное условие (7.3) получим

$$\frac{\partial u}{\partial n} = \frac{\alpha}{1 - \alpha h_{is}} u_i + \frac{\beta}{1 - \alpha h_{is}}. \quad (7.4)$$

Учёт граничных условий третьего рода при сборке по граням будет иметь вид

```

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

```

(7.5)

7.2.1 Универсальность условий третьего рода

Условие третьего рода (7.3) можно использовать для моделирования условий первого и второго рода. Так, условия второго рода (7.1) получаются, если положить $\alpha = 0$, $\beta = q$. А условия первого (6.6), – если

$$\alpha = \varepsilon^{-1}, \quad \beta = -\varepsilon^{-1}u^\Gamma, \quad (7.6)$$

где ε – малое положительное число.

Если подставить эти выражения в формулу (7.4), то можно убедится, что они дадут выражения (6.7) и (7.1) (в пределе при $\varepsilon \rightarrow 0$) соответственно.

7.3 Задание для самостоятельной работы

Сделать задачу, аналогичную п. 6.2, но использовать граничные условия 3-его рода. Необходимо имитировать условия первого рода через подход (7.6).

По формуле

$$n = \sqrt{\frac{\sum_i (u_i - u'_i)^2 V_i}{\sum_i V_i}}$$

подсчитать норму отклонения численного решения u_i задачи с использованием истинных граничных условий первого рода (6.7) от численного решения u' задачи, рас算анной с имитацией граничных условий первого рода через граничные условия третьего рода (7.4), (7.6).

Нарисовать график $n(\varepsilon)$ для расчётов на структурированной и скошенной сетках (в логарифмических координатах).

8 Лекция 8 (06.04)

8.1 Дополнительные точки коллокации на границах

До сих пор мы соотносили элементы сеточных векторов, которые получаются при аппроксимации функции на конечнообъёмную сетку, с центрами конечных объёмов. То есть точками коллокации служили центры объёмов, а длина сеточных векторов (количество точек коллокации) равнялась количеству ячеек сетки. Бывает удобно расширить набор точек коллокаций за счёт постановки точек на центры граничных граней.

Такой подход позволяет универсализировать подходы к аппроксимации перетоков через граничные грани. То есть для каждой граничной грани вместо использования одного из алгоритмов (6.10), (7.2), (7.5) в зависимости от типа граничного условия, нужно использовать универсальный алгоритм, основанный на внутреннем приближении типа (6.5):

$$\frac{\partial u}{\partial n} \approx \frac{u_j - u_i}{h_{ij}},$$

где i - индекс ячейки, соседней с граничной гранью, j - индекс точки коллокации, соответствующей граничной грани, h_{ij} – расстояние между двумя точками коллокации. С учётом этого соотношения обработка граничных граней для сборки строк матрицы, соответствующих центрам ячеек, примет вид

```
for s ∈ bnd      – граничные грани
    i = nei_cells(s) – соседняя с граничной гранью ячейка
    j = bnd_col(s)   – индекс точки коллокации, соответствующей грани
    v = |Γis| / his
    Aii += v
    Aij -= v
endfor
```

(8.1)

Строки матрицы, соответствующие граничным точкам коллокации, будут содержать аппроксимированные граничные условия. Так, для граней с условиями первого рода будет аппроксимироваться непосредственно выражение (6.6). Алгоритмическом виде это примет вид

```
for s ∈ bnd1     – грани с условиями первого рода
    j = bnd_col(s) – индекс точки коллокации, соответствующей грани
    Ajj = 1
    bj = uΓ
endfor
```

(8.2)

Для условий третьего рода (7.3) примем аппроксимацию

$$\frac{u_j - u_i}{h_{ij}} \approx \alpha u_j + \beta.$$

По прежнему будем считать, что i – точка коллокации, отнесённая к центру приграничной ячейки, j – точка коллокации отнесённая к центру граничной грани с условиями третьего рода. При сборки

СЛАУ просто запишем эту аппроксимацию в строки матриц, соответствующие граням с условиями третьего рода:

```

for  $s \in \text{bnd3}$            – грани с условиями третьего рода
     $i = \text{nei\_cells}(s)$    – соседняя с граничной гранью ячейка
     $j = \text{bnd\_col}(s)$      – индекс точки коллокации, соответствующей грани
     $A_{jj} = 1/h_{is} - \alpha$ 
     $A_{ji} = -1/h_{is}$ 
     $b_j = \beta$ 
endfor

```

(8.3)

Условия второго рода получаются из условий третьего упрощением $\alpha = 0$.

Преимуществами такого подхода является:

- Более очевидный учёт граничных условий в отдельной строке СЛАУ,
- Наличие явно выраженного граничного значения функции в сеточном векторе.

8.1.1 Пример

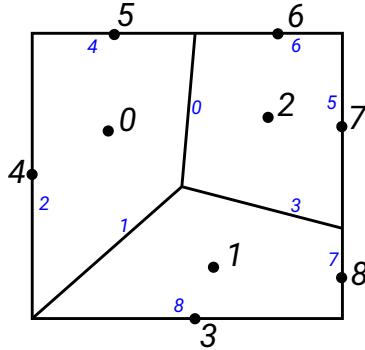


Рис. 5: Расширенный набор точек коллокации

На рис. 5. представлена конечнообъёмная сетка, содержащая три ячейки и девять граней. Индексация граней обозначена синими цифрами. Всего три внутренних грани и шесть граничных. Согласно стандартной методике конечных объёмов сеточная функция будет представлена массивом из трёх элементов. В расширенном наборе будет девять точек коллокации (обозначены чёрными кругами и проиндексированы чёрными цифрами): три соответствуют центрам ячеек и ещё шесть – центрам граничных граней.

Пусть в области с рис. 5 нужно решить уравнение Пуассона (6.1). Пусть на нижней грани задано условие первого рода: $u = C$, а на правой – условие третьего рода: $\partial u / \partial n = \alpha u + \beta$.

Старый подход Согласно ранее рассмотренному методу конечных объёмов аппроксимация задачи в ячейке с индексом 1 будет иметь следующий вид

$$\frac{u_1 - u_0}{h_{10}} |\gamma_1| + \frac{u_1 - u_2}{h_{12}} |\gamma_3| + \frac{u_1 - C}{h_{13}} |\gamma_8| + \frac{\alpha u_1 + \beta}{1 - \alpha h_{18}} |\gamma_7| = V_1 f_1.$$

Общая размерность матрицы СЛАУ при таком подходе будет равна 3×3 , а её элементы в 1-ой строке равны

$$a_{10} = -\frac{|\gamma_1|}{h_{10}}, \quad a_{12} = -\frac{|\gamma_3|}{h_{12}}, \quad a_{11} = \frac{|\gamma_1|}{h_{10}} + \frac{|\gamma_3|}{h_{12}} + \frac{|\gamma_8|}{h_{13}} + \frac{\alpha|\gamma_7|}{1 - \alpha h_{18}}.$$

Справа в 1-ой строке будет стоять

$$b_1 = V_1 f_1 + \frac{C|\gamma_8|}{h_{13}} - \frac{\beta|\gamma_7|}{1 - \alpha h_{18}}.$$

Новый подход В расширенным набором точек коллокаций матрица правой части будет иметь размерность 9×9 . Из них первые три будут собираться согласно классической процедуре метода конечных объёмов, но учитывая наличие дополнительных точек коллокации в центрах граничных граней. Так, 1-ое уравнение итоговой СЛАУ примет вид

$$\frac{u_1 - u_0}{h_{10}}|\gamma_1| + \frac{u_1 - u_2}{h_{12}}|\gamma_3| + \frac{u_1 - u_3}{h_{13}}|\gamma_8| + \frac{u_1 - u_8}{h_{18}}|\gamma_7| = V_1 f_1.$$

Остальные шесть уравнений будут представлять из себя аппроксимацию граничных условий для соответствующих граней. Так, 3-е уравнение будет соответствовать условию первого рода на грани γ_8 :

$$u_3 = C,$$

а уравнение 8 – условию третьего рода на грани γ_7 :

$$\frac{u_8 - u_1}{h_{18}} = \alpha u_8 + \beta$$

Переводя рассмотренные уравнения в матричные коэффициенты, получим следующие ненулевые коэффициенты итоговой матрицы $\{a_{ij}\}$ и вектора правой части $\{b_i\}$. Для 1-ой строки

$$a_{10} = -\frac{|\gamma_1|}{h_{10}}, \quad a_{12} = -\frac{|\gamma_3|}{h_{12}}, \quad a_{13} = -\frac{|\gamma_8|}{h_{13}}, \quad a_{18} = -\frac{|\gamma_7|}{h_{18}}, \quad a_{11} = -(a_{10} + a_{12} + a_{13} + a_{18}), \quad b_1 = V_1 f_1,$$

для 3-ей строки

$$a_{33} = 1, \quad b_3 = C,$$

для 8-ой строки

$$a_{81} = -\frac{1}{h_{18}}, \quad a_{88} = -a_{81} - \alpha, \quad b_8 = \beta$$

8.2 Задание для самостоятельной работы

1. Решить задачу из п. 6.2, с истинными граничными условиями первого рода, и с граничными условиями первого рода, поставленными через условия третьего рода.
2. Убедится, что полученный ответ с точностью до ошибки решения СЛАУ (10^{-8}) совпадает с результатами, полученным в двух предыдущих заданиях.
3. Для постановки с условиями третьего рода построить график максимального отклонения граничного значения полученного сеточной функции $\{u\}$ от точного решения в зависимости от

выбранного ε^{-1} из (7.6) от точного решения:

$$n_b = \max_{j > N_c} |u_j - u^e(\mathbf{x}_j)|,$$

N_c – количество ячеек сетки, j – индекс граничных точек коллокации, x_j – координата точки коллокации.

Рекомендации к программированию

- В структуру `DirichletFaces` необходимо добавить поле `icol` – индекс точки коллокации для текущей грани. Нумерацию граничных точек коллокации нужно вести начиная от `grid.n_cells()` согласно порядку вектора `_dirichlet_faces`.
- Следует учитывать, что вектор неизвестных `_u` будет иметь длину, большую чем количество ячеек. В частности, это может привести к ошибке сохранения в vtk. Чтобы ограничить количество сохраняемых элементов вектора, вызов сохранения необходимо осуществлять с дополнительным параметром:

```
VtkUtils::add_cell_data(_u, "numerical", filename, _grid.n_cells());
```

9 Лекция 9 (13.04)

9.1 Учёт скошенности сетки в двумерной МКО-аппроксимации

9.1.1 Уточнённая аппроксимация нормальной производной

Ключевым моментом для аппроксимации оператора Лапласа методом конечных объёмов является выражение нормально производной по грани конечного объёма. До сих пор мы пользовались соотношением (6.5), которая на скосенных сетках приводила к большим численным погрешностям и не давала желаемый второй порядок аппроксимации (см. задачу из п. 6.2). Для устранения этого недостатка распишем нормальную производную по грани более точно.

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 γ_{ij} (см. рисунок рис. 3):

$$\nabla u = \frac{\partial u}{\partial n} \mathbf{n} + \frac{\partial u}{\partial s} \mathbf{s}.$$

С помощью значений функции в точках коллокации i и j мы можем аппроксимировать производную

$$\left. \frac{\partial u}{\partial c} \right|_{\gamma_{ij}} = \frac{u_j - u_i}{h_{ij}} + O(h^2)$$

в центре грани γ_{ij} вторым порядком точности как симметричную разность. Эта производная есть проекция градиента функции u на вектор \mathbf{c} . Тогда можно записать:

$$\frac{\partial u}{\partial c} = \nabla u \cdot \mathbf{c} = \frac{\partial u}{\partial n} \mathbf{n} \cdot \mathbf{c} + \frac{\partial u}{\partial s} \mathbf{s} \cdot \mathbf{c} = \frac{\partial u}{\partial n} \cos(\widehat{\mathbf{n}, \mathbf{c}}) + \frac{\partial u}{\partial s} \sin(\widehat{\mathbf{n}, \mathbf{c}}).$$

Отсюда выразим искомую производную

$$\frac{\partial u}{\partial n} = \frac{\partial u}{\partial c} \frac{1}{\cos(\widehat{\mathbf{n}, \mathbf{c}})} - \frac{\partial u}{\partial s} \tan(\widehat{\mathbf{n}, \mathbf{c}}). \quad (9.1)$$

Для записи аппроксимации второго порядка нормальной производной необходимо с заданной точностью аппроксимировать производную по касательной к грани. В двумерном случае введём вспомогательные точки \mathbf{c}^+ и \mathbf{c}^- как показано на рисунке (6). Тогда в центре грани запишем искомую симметричную разность

$$\frac{\partial u}{\partial s} = \frac{u^+ - u^-}{|\mathbf{c}^+ - \mathbf{c}^-|} + O(h^2).$$

Тогда получим аппроксимацию второго порядка искомой производной:

$$\frac{\partial u}{\partial n} \approx \frac{u_j - u_i}{|\mathbf{c}_j - \mathbf{c}_i|} \frac{1}{\cos(\widehat{\mathbf{n}, \mathbf{c}})} - \frac{u^+ - u^-}{|\mathbf{c}^+ - \mathbf{c}^-|} \tan(\widehat{\mathbf{n}, \mathbf{c}}). \quad (9.2)$$

Точки c^\pm не являются точками коллокации, поэтому значения u^\pm явно присутствовать в аппроксимационной схеме не могут. Однако, их можно проинтерполировать через значения u в точках коллокации.

9.1.2 Интерполяция значения функции во вспомогательных точках

Для простой линейной интерполяции функции, заданной на двумерной плоскости, необходимо три узловые точки, не лежащие на одной прямой. Пусть двумя из этих трёх точек будут уже обозначенные \mathbf{c}_i и \mathbf{c}_j . Третью точку \mathbf{c}_k будем выбирать из коллокации, соседних с временной точкой: центров ячеек и центров граничных граней, которые содержат точку \mathbf{c}^- .

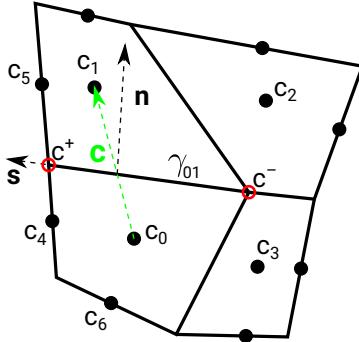


Рис. 6: Пример расположения вспомогательных точек (показаны красными кругами) для аппроксимации производной по грани

На примере с рис. 6 для точки \mathbf{c}^- соседними будут точки c_0, c_1, c_2, c_3 . Точки \mathbf{c}_0 и \mathbf{c}_1 – это первые две точки интерполяции (узлы i и j в предыдущих обозначениях). Третью точку выберем как ближайшую к \mathbf{c}^- из остальных, а именно c_3 . Для точки \mathbf{c}^+ третьей точкой будет c_4 , выбранная аналогичным образом из набора c_0, c_1, c_4, c_5 .

Отметим, что для написания надёжного кода необходимо удостовериться, что выбранная третья точка не лежит на одной прямой с двумя первыми. И если лежит, то необходимо выбрать следующего кандидата. А если кандидатов больше нет, то поиск необходимо продолжить среди несоседних точек коллокации.

После того, как три узловые точки интерполяции выбраны можно выразить вспомогательные значения

$$u^- = C_i^- u_i + C_j^- u_j + C_k^- u_k, \\ u^+ = C_i^+ u_i + C_j^+ u_j + C_s^+ u_s,$$

где коэффициенты интерполяции C вычисляются следуя формуле (A.27). Например,

$$C_i^- = \frac{|\Delta_{jk-}|}{|\Delta_{ijk}|} = \frac{(\mathbf{c}_k - \mathbf{c}_j) \times (\mathbf{c}^- - \mathbf{c}_j)}{(\mathbf{c}_j - \mathbf{c}_i) \times (\mathbf{c}_k - \mathbf{c}_i)}$$

Подставив полученные интерполяционные соотношения в (9.2), получим аппроксимацию нормальной производной в виде линейную формы

$$\frac{\partial u}{\partial n} \approx C_i u_i + C_j u_j + C_k u_k + C_s u_s. \quad (9.3)$$

Коэффициенты которой равны

$$\begin{aligned} C_i &= -w_1 - (C_i^+ - C_i^-)w_2, \\ C_j &= w_1 - (C_j^+ - C_j^-)w_2, \\ C_k &= C_k^- w_2, \\ C_s &= -C_s^+ w_2, \\ w_1 &= \frac{1}{|\mathbf{c}_j - \mathbf{c}_i| \cos(\widehat{\mathbf{n}, \mathbf{c}})}, \\ w_2 &= \frac{\tan(\widehat{\mathbf{n}, \mathbf{c}})}{|\mathbf{c}^+ - \mathbf{c}^-|} \end{aligned}$$

9.1.3 Производная по границе

Из-за использования расширенного набора точек коллокации, процедура аппроксимации нормальной производной по граничной грани ничем не отличается от аналогичной процедуры для внутренней грани. Так для точки коллокации \mathbf{c}_4 линейная форма, аппроксимирующая нормальную производную, будет содержать значения u_4, u_0, u_6, u_1 , последние два из которых используются для интерполяции вспомогательных точек.

Рассмотрим условие третьего рода (7.3) на стенке, соответствующей индексу коллокации j , соседней с ячейкой i . С учётом (9.3) получим:

$$\frac{\partial u}{\partial n} = C_i u_i + C_j u_j + C_k u_k + C_s u_s = \alpha u_j + \beta,$$

тогда j -ое уравнение СЛАУ будет иметь вид

$$C_i u_i + (C_j - \alpha) u_j + C_k u_k + C_s u_s = \beta. \quad (9.4)$$

9.1.4 Сборка СЛАУ для уравнения Пуассона

Рассмотрим процедуру сборки системы линейных уравнений для решения уравнения Пуассона. Будем использовать цикл по граням по аналогии с (6.9). Ключевым моментом алгоритма будет являться вычисление линейной формы вида (9.3). Отметим, что цикл по внутренним (6.9) и граничным (8.1) граням можно объединить в один цикл по всем граням. При этом следует иметь ввиду, что этот цикл собирает уравнение внутри конечного объема (6.2), и поэтому вносит изменения только в стро-

ки, соответствующие центрам ячеек.

```

for  $s \in \text{faces}$                                      – цикл по всем граням
     $i, j, k, s = \text{dn\_cells}(s)$                   – индексы точек коллокации для аппроксимации  $\partial u / \partial n$ 
     $v_i, v_j, v_k, v_s = \text{dn\_coefs}(s)$           – коэф-ты линейной формы для аппроксимации  $\partial u / \partial n$ 
     $a = \text{face\_area}(s)$                            – площадь грани
    if  $i$  is cell center                            – если  $i$  – коллокация для центра ячейки
         $A_{ii} -= a v_i;$                             –  $i$ -ая строка матрицы (против нормали к грани)
         $A_{ij} -= a v_j;$ 
         $A_{ik} -= a v_k;$ 
         $A_{is} -= a v_s;$ 
    endif
    if  $j$  is cell center                            – если  $j$  – коллокация для центра ячейки
         $A_{ji} += a v_i;$                             –  $j$ -ая строка матрицы (по нормали к грани)
         $A_{jj} += a v_j;$ 
         $A_{jk} += a v_k;$ 
         $A_{js} += a v_s;$ 
    endif
endfor

```

В случае, если используются граничные условия второго или третьего рода, та же процедура должна быть использована для выражения на границах (9.4) по аналогии с (8.3).

```

for  $s \in \text{bnd3}$                                      – грани с условиями третьего рода
     $i, j, k, s = \text{dn\_cells}(s)$                   – индексы точек коллокации для аппроксимации  $\partial u / \partial n$ 
     $v_i, v_j, v_k, v_s = \text{dn\_coefs}(s)$           – коэф-ты линейной формы для аппроксимации  $\partial u / \partial n$ 
    if  $j$  is cell center
         $\text{swap}(i, j)$                                 – переворачиваем нормаль. Теперь  $j$  – коллокация по грани
         $v_i = -v_i, \quad v_j = -v_j$ 
         $v_k = -v_k, \quad v_s = -v_s$ 
    endif
     $A_{ji} = C_i, \quad A_{jj} = C_j - \alpha$ 
     $A_{jk} = C_k, \quad A_{js} = C_s$ 
     $b_j = \beta$ 
endfor

```

9.2 Задание для самостоятельной работы

Решить двумерную задачу Пуассона с граничными условиями первого рода, аналогичную 6.2, но использовать поправку на скошенность. Провести расчёт на сгущающихся сетках и проиллюстрировать порядок аппроксимации для структурированной, pеби и скошенной сетки. Сравнить графики сходимости с аналогичными, полученными в п. 6.2.

Рекомендации к программированию Для вычисления линейной формы (9.3) использовать класс `FvmLinFormFacesDn` из файла `fvm/fvm_assembler.hpp`.

Объект этого класса необходимо собрать на этапе инициализации задачи. Далее линейные формы для заданной грани вычисляются вызовом метода

`FvmLinFormFacesDn::linear_combination(size_t iface)`. Этот метод возвращает упорядоченную линейную комбинацию из четырех (для двумерной задачи) слагаемых. Получение индексов и коэффициентов линейной формы согласно алгоритму (9.5) осуществляется следующим образом

```
FvmLinFormFacesDn faces_dn(grid);
for (size_t iface=0; iface < grid.n_faces(); ++iface){
    auto linform = faces_dn.linear_combination(iface);
    size_t i = linform[0].first;
    size_t j = linform[1].first;
    size_t k = linform[2].first;
    size_t s = linform[3].first;
    double vi = linform[0].second;
    double vj = linform[1].second;
    double vk = linform[2].second;
    double vs = linform[3].second;
}
```

10 Лекция 10 (20.04)

10.1 Учёт скошенности сетки в 3D

11 Лекция 11 (27.04)

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 (пирамидальный) базис

12 Лекция 12 (04.05)

12.1 Поэлементная сборка конечноэлементных матриц

12.1.0.1 Элементные матрицы

Матрица масс

$$m_{ij}^k = \int_{E^k} \phi_i(\mathbf{x}) \phi_j(\mathbf{x}) d\mathbf{x} \quad (12.1)$$

Вектор нагрузок

$$l_i^k = \int_{E^k} \phi_i(\mathbf{x}) d\mathbf{x} \quad (12.2)$$

Матрица жёсткости

$$s_{ij}^k = \int_{E^k} \nabla \phi_i \cdot \nabla \phi_j d\mathbf{x} \quad (12.3)$$

13 Лекция 13 (11.05)

13.1 Вычисление элементных интегралов в параметрическом пространстве

Будем вычислять элементные интегралы (12.1) – (12.3) в параметрическом пространстве ξ . Для этого введём преобразование координат $\mathbf{x} \rightarrow \xi$ согласно п.А.4.2. Интеграл для определения локальной матрицы масс (12.1) в параметрическом пространстве распишется согласно формуле (A.34)

$$m_{ij}^k = \int_{\tilde{E}^k} \tilde{\phi}_i^{(k)}(\xi) \tilde{\phi}_j^{(k)}(\xi) |J^{(k)}(\xi)| d\xi \quad (13.1)$$

Здесь \tilde{E}^k – параметрический образ конечного элемента E^k , $J^{(k)}$ - Якобиан преобразования для k -ого элемента, $\tilde{\phi}_i^{(k)}$ – часть базисной функции $\phi_{c_i^k}$, определённая в конечном элементе E^k и заданная в параметрических координатах, а c_i^k – глобальный индекс базисной функции, которая в k -ом элементе имеет локальный индекс i . Таким образом справедливо

$$\tilde{\phi}_i^{(k)}(\xi) = \phi_{c_i^k}(\mathbf{x}(\xi))$$

Локальный вектор нагрузок

$$l_i^k = \int_{\tilde{E}^k} \tilde{\phi}_i^{(k)}(\xi) |J^{(k)}(\xi)| d\xi \quad (13.2)$$

Локальная матрица жёсткости

$$s_{ij}^k = \int_{\tilde{E}^k} \nabla_{\mathbf{x}} \tilde{\phi}_i^{(k)} \cdot \nabla_{\mathbf{x}} \tilde{\phi}_j^{(k)} |J^{(k)}(\xi)| d\xi \quad (13.3)$$

Здесь $\nabla_{\mathbf{x}} \tilde{\phi}_i^{(k)}$ – градиент локального базиса (заданного в параметрическом пространстве) по физическим координатам. Для его вычисления следует воспользоваться формулами (A.33)

13.2 Двумерное уравнение Пуассона

13.2.0.1 Треугольный элемент. Линейный двумерный базис

Матрица масс (из (13.1)):

$$M_{ij}^E = \int_0^1 \int_0^{1-\xi} \phi_i(\xi, \eta) \phi_j(\xi, \eta) |J| d\eta d\xi = \frac{|J|}{24} \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix} \quad (13.4)$$

13.3 Разбор программной реализации МКЭ

Численное решение уравнения Пуассона с граничными условиями первого рода реализовано в файле `poisson_fem_solve_test.cpp`. Будем рассматривать решение одномерной задачи с использованием

пирамидальных базисов (тест [poisson1-fem-lintri]). В этом тесте определяется двумерная аналитическая функция

$$f(x) = \sin(10x^2),$$

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 построение одномерной сетки, создание рабочего объекта, вызов решения с возвращением нормы полученной ошибки и вывод данных (сохранение решения в vtk-файл и печать нормы в консоль):

```
170 Grid1D grid(0, 1, 10);
171 TestPoissonLinearSegmentWorker worker(grid);
172 double nrm = worker.solve();
173 worker.save_vtk("poisson1_fem.vtk");
174 std::cout << grid.n_cells() << " " << nrm << std::endl;
```

Основная работа происходит в классе `TestPoissonLinearSegmentWorker`.

13.3.1 Рабочий объект

Класс `TestPoissonLinearSegmentWorker` наследуется от `ITestPoisson1FemWorker`. В этом классе сформулированы 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.

```
42 double ITestPoissonFemWorker::solve(){
43     // 1. build SLAE
44     CsrMatrix mat = approximate_lhs();
45     std::vector<double> rhs = approximate_rhs();
46     // 2. Dirichlet bc
47     for (size_t ibas: dirichlet_bases()){
48         mat.set_unit_row(ibas);
49         Point p = _fem.reference_point(ibas);
50         rhs[ibas] = exact_solution(p);
51     }
52     // 3. solve SLAE
53     AmgcMatrixSolver solver({ {"precond.relax.type", "gauss_seidel"} });
54     solver.set_matrix(mat);
55     solver.solve(rhs, _u);
56     // 4. compute norm2
57     return compute_norm2();
58 }
```

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

```

73 CsrMatrix ITestPoissonFemWorker::approximate_lhs() const{
74     CsrMatrix ret(_fem.stencil());
75     for (size_t ielem=0; ielem < _fem.n_elements(); ++ielem){
76         const FemElement& elem = _fem.element(ielem);
77         std::vector<double> local_stiff = elem.integrals->stiff_matrix();
78         _fem.add_to_global_matrix(ielem, local_stiff, ret.vals());
79     }
80     return ret;
81 }
```

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`.

13.3.2 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 `cfd/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 `TestPoissonLinearSegmentWorker::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.

13.3.3 Концепция конечного элемента

Класс конечного элемента `FemElement` определён в файле `fem/fem_element.hpp` как

```
206 struct FemElement{  
207     std::shared_ptr<const IElementGeometry> geometry;  
208     std::shared_ptr<const IElementBasis> basis;  
209     std::shared_ptr<const IElementIntegrals> integrals;  
210 };
```

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.

13.3.3.1 Определение линейного одномерного элемента

. Так, в рассматриваемом нами teste `"[poisson1-fem-linsegm]"`, используются только линейные одномерные элементы. Используется следующее определение элемента:

```
152     auto geom = std::make_shared<SegmentLinearGeometry>(p0, p1);  
153     auto basis = std::make_shared<SegmentLinearBasis>();  
154     auto integrals = std::make_shared<SegmentLinearIntegrals>(geom->jacobi({}));  
155     FemElement elem{geom, basis, integrals};
```

Здесь последовательно определяются:

- геометрия отрезка `geom` – путём задания двух точек в физической плоскости `p0, p1`,
- линейный одномерный базис `basis`,

- правила интегрирования

`integrals` по параметрическому отрезку $x \in [-1, 1]$ с использованием точных формул. Эти формулы зависят только от матрицы Якоби `jac` (размерности 1×1), которая вычисляется с использованием геометрических свойств элемента (в данном случае матрица Якоби постоянная, поэтому её можно вычислять в любой точке параметрической плоскости)

Этих трёх алгоритмов достаточно для полного определения конечного элемента `elem`.

13.3.3.2 Геометрические свойства элемента

Интерфейс `IElementGeometry`, заданный в файле `cfd/fem/fem_element.hpp`, определяет геометрические свойства элемента:

```
14 class IElementGeometry{
15 public:
16     virtual ~IElementGeometry() = default;
17
18     virtual JacobiMatrix jacobi(Point xi) const = 0;
19     virtual Point to_physical(Point xi) const { _THROW_NOT_IMP_; }
20     virtual Point to_parametric(Point p) const { _THROW_NOT_IMP_; }
21     virtual Point parametric_center() const { _THROW_NOT_IMP_; }
22 };
```

Для вычисления элементных матриц главным геометрическим свойством элемента является функция для вычисления матрицы Якоби (`jacobi`). В простейших реализациях этого интерфейса для симплексных геометрий матрица Якоби постоянна для любой точки, то есть функция `jacobi` возвращает один и тот же ответ вне зависимости от переданного аргумента.

Кроме того, этот интерфейс предоставляет функции преобразования координат из физического пространства в параметрическое и обратно: `to_parametric`, `to_physical`. А также задает центральную точку в параметрическом пространстве `parametric_center`.

13.3.3.3 Элементный базис

Интерфейс для определения локального элементного базиса имеет вид

```
35 class IElementBasis{
36 public:
37     virtual ~IElementBasis() = default;
38
39     virtual size_t size() const = 0;
40     virtual std::vector<Point> parametric_reference_points() const = 0;
41     virtual std::vector<BasisType> basis_types() const = 0;
42     virtual std::vector<double> value(Point xi) const = 0;
```

```

43     virtual std::vector<Vector> grad(Point xi) const = 0;
44     virtual std::vector<std::array<double, 6>> upper_hessian(Point xi) const {
45         _THROW_NOT_IMP_;
    };

```

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.
- `basis_type` – тип базисной функции. До сих пор мы имели дело только с узловыми (`BasisType::Nodal`) функциями.
- `upper_hessian` – верхняя часть матрицы Гессе (вторые производные базисных функций в заданной точке)

Конкретная реализация для линейного треугольного элемента `TriangleLinearBasis` (в файле `fd/fem/elem2d/triangle_linear.cpp`) включает в себя линейный Лагранжев базис в двумерном пространстве согласно (A.23):

```

35 size_t TriangleLinearBasis::size() const {
36     return 3;
37 }

```

```

39 std::vector<Point> TriangleLinearBasis::parametric_reference_points() const {
40     return {Point(0, 0), Point(1, 0), Point(0, 1)};
41 }

```

```

47 std::vector<double> TriangleLinearBasis::value(Point xi_) const {
48     double xi = xi_.x();
49     double eta = xi_.y();
50     return {1 - xi - eta, xi, eta};
51 }

```

```

53 std::vector<Vector> TriangleLinearBasis::grad(Point xi) const {
54     return { Vector(-1, -1), Vector(1, 0), Vector(0, 1) };
55 }
```

13.3.3.4 Калькулятор элементных матриц

Интерфейс `IElementIntegrals` предоставляет методы для вычисления элементных матриц. До сих пор были рассмотрены две элементные матрицы: матрица масс (13.1) и матрица жёсткости (13.3). 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.

До сих пор мы рассматривали только точное вычисление. Точные формулы интегрирования зависят от вида элемента. Так, для линейного одномерного элемента аналитическое интегрирование реализовано в классе `SegmentLinearIntegrals` в файле

`cfd/fem/elem1d/segment_linear.hpp`, а для линейного треугольного элемента – `TriangleLinearIntegrals` в файле `cfd/fem/elem2d/triangle_linear.hpp`. Все интегралы для этих симплексных элементов будут зависеть только от матрицы Якоби, которая и передётся этому классу в конструктор. Например, вычисление матрицы масс (по (13.4)) запрограммировано в виде

```

62 std::vector<double> TriangleLinearIntegrals::mass_matrix() const {
63     double s0 = _jac.modj/12.0;
64     double s1 = _jac.modj/24.0;
65     return {s0, s1, s1,
66             s1, s0, s1,
67             s1, s1, s0};
68 }
```

13.4 Задание для самостоятельной работы

- Показать второй порядок аппроксимации решения одномерного уравнения Пуассона на линейных конечных элементах;
- Решить двумерное уравнение Пуассона с граничными условиями первого рода в квадратной области на треугольной сетке. Определить порядок аппроксимации двумерного уравнения

Пуассона на треугольных элементах. Для построения треугольных сеток различного разрешения использовать скрипт

`trigrid.py`.

- Показать второй порядок аппроксимации решения двумерного уравнения Пуассона на линейных треугольных конечных элементах;
- Сравнить сходимость на сгущающейся сетке конечноэлементного и конечнообъёмного решения двумерного уравнения Пуассона на одних и тех же треугольных сетках. Конечнообъёмное решение получать с использованием поправки на скошенность.

Рекомендации к программированию Для реализации двумерного решения следует по аналогии с одномерным написать класс

`ITestPoisson2FemWorker`, в котором реализовать двумерные функции точного решения и правой части, и наследуемый от него рабочий класс `TestPoissonLinearSegmentWorker`, в котором реализовать статическую функцию `build_fem`. При реализации последней использовать алгоритмы для треугольников.

```
auto geom = std::make_shared<TriangleLinearGeometry>(p0, p1, p2);
auto basis = std::make_shared<TriangleLinearBasis>();
auto integrals = std::make_shared<TriangleLinearIntegrals>(geom->jacobi({}));
```

14 Лекция 14 (02.11)

14.1 Двухслойные схемы для нестационарных уравнений

14.1.1 Определение

Рассмотрим дифференциальное уравнение вида

$$\frac{\partial u}{\partial t} = f, \quad f(x, t) = g(x, t) - Lu(x, y) \quad (14.1)$$

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

$$\begin{aligned} \frac{u(t + \tau) - u(t)}{\tau} &= \left. \frac{\partial u}{\partial t} \right|_t + o(\tau) \quad \text{– разность вперёд;} \\ \frac{u(t) - u(t + \tau)}{\tau} &= \left. \frac{\partial u}{\partial t} \right|_{t+\tau} + o(\tau) \quad \text{– разность назад;} \\ \frac{u(t + \tau) - u(t - \tau)}{2\tau} &= \left. \frac{\partial u}{\partial t} \right|_{t+\frac{\tau}{2}} + o(\tau^2) \quad \text{– симметричная разность.} \end{aligned} \quad (14.2)$$

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.

14.1.1.1 Явная схема

При использовании разности назад уравнение (14.1) в полуdiscретизованном (то есть дискретизованном только по времени, но не по пространству) виде запишется как

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

или, после переноса всех известных слагаемых вправо

$$u(x, t + \tau) = (E - \tau L) u(x, t) + \tau g(x, t). \quad (14.3)$$

Здесь E – единичный оператор. Схема (14.3) называется явной схемой и имеет первый порядок точности.

14.1.1.2 Неявная схема

Выбрав разность назад из выражения (14.2) полуdiscретизованная схема для уравнения (14.1) примет вид

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

В результате преобразования получим неявную схему первого порядка точности

$$(E + \tau L) u(x, t + \tau) = u(x, t) + \tau g(x, t + \tau). \quad (14.4)$$

14.1.1.3 Схема Кранка–Николсон

Подставим симметричную разность из (14.2) в уравнение (14.1). Формально получим

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

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:

$$u\left(x, t + \frac{\tau}{2}\right) = \frac{1}{2}u(x, t) + \frac{1}{2}u(x, t + \tau) + o(\tau^2) \quad (14.5)$$

Аналогичная запись справедлива и для свободного члена g . Если оператор L – нестационарный или нелинейный, то аппроксимацию (14.5) 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 (14.5) симметричная разностная схема запишется как

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

или

$$\left(E + \frac{\tau}{2}L\right)u(x, t + \tau) = \left(E - \frac{\tau}{2}L\right)u(x, t) + \frac{\tau}{2}(g(x, t) + g(x, t + \tau)). \quad (14.6)$$

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 (14.6) его нужно брать на следующем временном слое, а в правой – на текущем.

14.1.1.4 Обобщённая двухслойная схема

Выражения (14.3), (14.4), (14.6) можно записать в обобщённой форме

$$(E + \theta\tau L)u(x, t + \tau) = (E + (\theta - 1)\tau L)u(x, t) + (1 - \theta)g(x, t) + \theta g(x, t + \tau). \quad (14.7)$$

Коэффициент θ – степень неявности схемы:

- $\theta = 0$ – явная схема (14.3),
- $\theta = 1$ – полностью неявная схема (14.4),
- $\theta = 1/2$ – схема Кранка–Николсон (14.6).

Отметим, что только при $\theta = 1/2$ схема (14.7) имеет второй порядок точности по времени. Для других значений (в том числе промежуточных) схема будет иметь ошибку первого порядка $o(\tau)$.

14.2 Схемы высокого порядка точности

Существуют два подхода к построению схем дискретизации по времени произвольного высокого порядка: первый из них предполагает использование нескольких временных слоев: $t + \tau, t, t - \tau, t - 2\tau, \dots$, второй – использование большого количества промежуточных точек на единственном временном отрезке: $t + c_i \tau, c_i \in [0, 1]$. Первый подход используется для построения схем Адамса, второй – схем Рунге–Кутта.

14.2.1 Многослойные схемы. Схемы Адамса

В общем виде многослойную расчётную схему можно записать в виде

$$\sum_{i=1}^s a_i u_i = \tau \sum_{i=1}^s b_i f_i. \quad (14.8)$$

Здесь нижние индексы функций использованы для указания временного слоя:

$$\begin{aligned} u_i &= u(x, t + (i + 1 - s)\tau), \\ f_i &= g(x, t + (i + 1 - s)\tau) - Lu_i, \end{aligned}$$

а s – это общее количество используемых временных слоёв. Значение всех функций на слоях $i < s$ считаются известными, а значение u_s на последнем (текущем) слое подлежит определению. Коэффициенты a_i , b_i определяют конкретную схему. Для однозначности принято задавать коэффициент перед значением u на текущем слое равным единице: $a_s = 1$. Так, для двухслойной ($s = 2$) схемы Кранка–Николсон (14.6) эти коэффициенты примут вид:

$$\begin{aligned} a_1 &= -1, \quad a_2 = 1, \\ b_1 &= 1/2, \quad b_2 = 1/2. \end{aligned}$$

Для того чтобы схема, записанная в общем виде (14.8), была явной, необходимо, чтобы выполнялось условие $b_s = 0$.

14.2.1.1 Явные схемы Адамса–Башфорта

Запишем интерполяционный полином для правой части уравнения (14.1) по значениям на предыдущих временных слоях $f_i, i < s$:

$$p(t) = \sum_{i=1}^{s-1} \left(f_i \prod_{\substack{m=1 \\ m \neq i}}^{s-1} \frac{t - t_i}{t_m - t_i} \right).$$

Порядок аппроксимации этого полинома на единицу больше его степени. С учётом $t_{i+1} - t_i = \tau$ можно записать

$$f(t) = p(t) + o(\tau^{s-1}).$$

Далее проинтегрируем уравнение по текущему временному отрезку: $[t_{s-1}, t_s]$:

$$u_s - u_{s-1} = \sum_{i=1}^{s-1} \left(f_i \int_{t_{s-1}}^{t_s} \prod_{\substack{m=0 \\ m \neq i}}^{s-1} \frac{t - t_i}{t_m - t_i} dt \right).$$

Отсюда коэффициенты в общей форме (14.8) примут вид

$$a_i = \begin{cases} 1, & i = s \\ -1, & i = s-1 \\ 0, & i < s-1 \end{cases} \quad b_i = \begin{cases} 0, & i = s \\ \frac{1}{\tau} \int_{t_{s-1}}^{t_s} \prod_{\substack{m=0 \\ m \neq i}}^{s-1} \frac{t - t_i}{t_m - t_i} dt, & i < s. \end{cases}$$

Примеры трёх-, четырех- и пятислойной явных схем приведены ниже:

$$\begin{aligned} \frac{u_3 - u_2}{\tau} &= \frac{3f_2 - f_1}{2} + o(\tau^2), \\ \frac{u_4 - u_3}{\tau} &= \frac{23f_3 - 16f_2 + 5f_1}{12} + o(\tau^3), \\ \frac{u_5 - u_4}{\tau} &= \frac{55f_4 - 59f_3 + 37f_2 - 9f_1}{24} + o(\tau^4). \end{aligned}$$

14.2.1.2 Неявные схемы Адамса–Мулттона

Теперь снимем требование явности и запишем интерполяционный полином для правой части исходного уравнения (14.1) с использованием s точек:

$$p(t) = \sum_{i=1}^s \left(f_i \prod_{\substack{m=1 \\ m \neq i}}^s \frac{t - t_i}{t_m - t_i} \right).$$

Тогда порядок аппроксимации этого полинома будем на единицу больше, чем для полинома явного метода. Далее, проинтегрируем исходное уравнение по отрезку t_{s-1}, t_s и получим коэффициенты общей формы (14.8):

$$a_i = \begin{cases} 1, & i = s \\ -1, & i = s-1 \\ 0, & i < s-1 \end{cases} \quad b_i = \frac{1}{\tau} \int_{t_{s-1}}^{t_s} \prod_{\substack{m=0 \\ m \neq i}}^s \frac{t - t_i}{t_m - t_i} dt.$$

Двухслойная схема Адамса–Мулттона будет аналогична неявной двухслойной схеме (14.4), трёх-

слойная – схеме Кранка–Николсон (14.6). Примеры схемы высоких порядков представлены ниже:

$$\begin{aligned}\frac{u_3 - u_2}{\tau} &= \frac{5f_3 + 8f_2 - f_1}{12} + o(\tau^3), \\ \frac{u_4 - u_3}{\tau} &= \frac{9f_4 + 19f_3 - 5f_2 + f_1}{24} + o(\tau^4), \\ \frac{u_5 - u_4}{\tau} &= \frac{251f_5 + 646f_4 - 264f_3 + 106f_2 - 19f_1}{720} + o(\tau^5).\end{aligned}$$

14.2.2 Схемы Рунге–Кутта

TODO

14.3 Методы исследования устойчивости расчётных схем

14.3.1 Дискретизация по времени как итерационный процесс

14.3.1.1 Двухслойный итерационный процесс

Простой двухслойный итерационный процесс определяется как

$$u^{n+1} = Au^n + b, \quad (14.9)$$

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

Определение значения функции на следующий момент времени $u(t + \tau)$ по двухслойной схеме (14.7) можно представить как простой итерационный процесс (14.9), где

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

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

Итерационный процесс называется сходящимся, если

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

14.3.1.2 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^∞	∞	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 $|A| \leq 1$.

14.3.1.3 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.

14.3.2 Матричный метод

Итерационные процессы, возникающие при численном решении дифференциальных уравнений сечеточными методами, имеют матричную природу. То есть оператор преобразования A в выражении (14.9) – 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 $S(A)$:

$$S(A) = \max_j |\lambda_j| \leq 1, \quad (14.10)$$

где λ_j – собственные числа матрицы A .

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

14.3.2.1 Явная схема для нестационарного уравнения диффузии

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

$$\frac{\partial u}{\partial t} = k \frac{\partial^2 u}{\partial x^2},$$

$$u(x, 0) = u_0(x),$$

$$u(x_a, t) = u_a,$$

$$u(x_b, t) = u_b.$$

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

$$\hat{u}_i = u_i + \gamma(u_{i-1} - 2u_i + u_{i+1}), \quad i = \overline{1, N-1}, \quad (14.11)$$

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 (14.10), запишем условие устойчивости для явной схемы (14.11)

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

14.3.2.2 Неявная схема для нестационарного уравнения диффузии

Аналогично, рассмотрим неявную схему

$$\hat{u}_i - \gamma(\hat{u}_{i-1} - 2\hat{u}_i + \hat{u}_{i+1}) = u_i, \quad i = \overline{1, N-1}, \quad (14.12)$$

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 (14.12) безусловно устойчива.

14.3.3 Метод дискретных возмущений

Метод дискретных возмущений заключается в использовании в качестве начального приближения нулевого вектора, с возмущением ε 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 (14.3.1.2).

14.3.3.1 Явная схема против потока для уравнения переноса

Для иллюстрации рассмотрим одномерное уравнение переноса

$$\frac{\partial u}{\partial t} + V \frac{\partial u}{\partial x} = 0 \quad (14.13)$$

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

$$\hat{u}_i = u_i - C(u_i - u_{i-1}), \quad (14.14)$$

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 (14.14) будет выражение

$$C \leq 1.$$

14.3.4 Метод Неймана

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

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

κ – 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 κ_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

$$u_i = e^{\mathbf{i}i\theta}, \quad \theta = \kappa_0 h \quad (14.15)$$

На следующем временном шаге это возмущение примет вид:

$$\hat{u}_i = Ge^{\mathbf{i}i\theta}. \quad (14.16)$$

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

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

14.3.4.1 Неявная противопотоковая схема для уравнения переноса

Для примера анализа устойчивости методом Неймана опять рассмотрим задачу (14.13), но на этот раз рассмотрим чисто неявную аппроксимацию

$$\hat{u}_i + C(\hat{u}_i - \hat{u}_{i-1}) = u_i. \quad (14.17)$$

Подставим (14.15), (14.16)

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

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 (14.17) безусловно устойчива.

14.3.4.2 Противопотоковая схема Кранка-Николсон для уравнения переноса

Для того же самого уравнения (14.13) рассмотрим схему Кранка-Николсон (14.5):

$$\hat{u}_i + \frac{C}{2}(\hat{u}_i - \hat{u}_{i-1}) = u_i - \frac{C}{2}(u_i - u_{i-1}). \quad (14.18)$$

Так же подставим (14.15), (14.16) 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 θ .

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

Получаем, что схема видеа (14.18) безусловно устойчива.

14.3.4.3 Явная схема для уравнения нестационарной конвекции-диффузии

Рассмотрим уравнение конвекции-диффузии

$$\frac{\partial u}{\partial t} + V \frac{\partial u}{\partial x} = k \frac{\partial^2 u}{\partial x^2} \quad (14.19)$$

Сначала напишем чисто явную схему второго порядка по пространству:

$$\frac{\hat{u}_i - u_i}{\tau} + V \frac{u_{i+1} - u_{i-1}}{2h} = k \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} \quad (14.20)$$

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 (14.15), (14.16)

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

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 θ , 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 (14.20):

$$\begin{aligned}\cos \theta = 1 &\Rightarrow C \leq \sqrt{2\gamma}, \\ \cos \theta = -1 &\Rightarrow \gamma \leq 1/2.\end{aligned}\quad (14.21)$$

Обычно вместо первого из условий (14.21) 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 Re_c :

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

14.3.4.4 Неявная схема для уравнения нестационарной конвекции-диффузии

Аналогичным образом рассмотрим неявную диффузии схему для уравнения (14.19) вида

$$\frac{\hat{u}_i - u_i}{\tau} + V \frac{u_{i+1} - u_{i-1}}{2h} = k \frac{\hat{u}_{i+1} - 2\hat{u}_i + \hat{u}_{i-1}}{h^2} \quad (14.22)$$

Подставляя представление для возмущения с волновым числом θ , 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

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

14.3.5 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 (14.3.2.2) для уравнения диффузного переноса и для условно устойчивой явной схемы (14.3.2.1).
- Для уравнения переноса с разностью против потока явная (14.3.3.1) схема условно устойчива, в то время как неявная (14.3.4.1) – устойчива безусловно.
- Даже если только часть схемы неявная, это повышает устойчивость. Так, явная (14.3.4.3) схема для уравнения конвекции-диффузии имеет два условия устойчивости, в то время как схема, неявная по диффузии (14.3.4.4) – только одно.
- Аналогично, явная (14.3.3.1) схема против потока для уравнения переноса условно устойчива, а схема Кранка-Николсон (14.3.4.2) 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 (14.3.4.3), и полунеявная (14.3.4.4)), 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.

14.4 Программная реализация схемы для уравнения переноса

14.4.1 Постановка задачи

Рассматриваются три схемы по времени для противопотоковой аппроксимации уравнения переноса (14.13):

- явная схема (14.14) (тест называется `[transport1-explicit]`),
- неявная схема (14.17) (`[transport1-implicit]`),
- схема Кранка–Николсон (14.18) (`[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`.

14.4.2 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 τ , 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;
```

14.4.3 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

14.4.3.1 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 (14.14).

```
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 }
```

14.4.3.2 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 (14.17). 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 (14.17)

```
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 (14.17) 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;
}

```

14.4.3.3 Схема Кранка-Николсон

Её решатель реализован в классе `TestTransport1WorkerCN`.

По аналогии с предыдущей программой, здесь требуется решить СЛАУ, возникающую из схемы (14.18). 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 (14.18) с учётом граничных условий

```

204     CsrMatrix build_lhs(double tau) override{
205
206         LdMatrix 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 (14.18) 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     }

```

14.4.4 Анализ результатов работы

Сравнение полученных ответов (по явной и неявной схемам) с точным решением представлено на рис. 7.

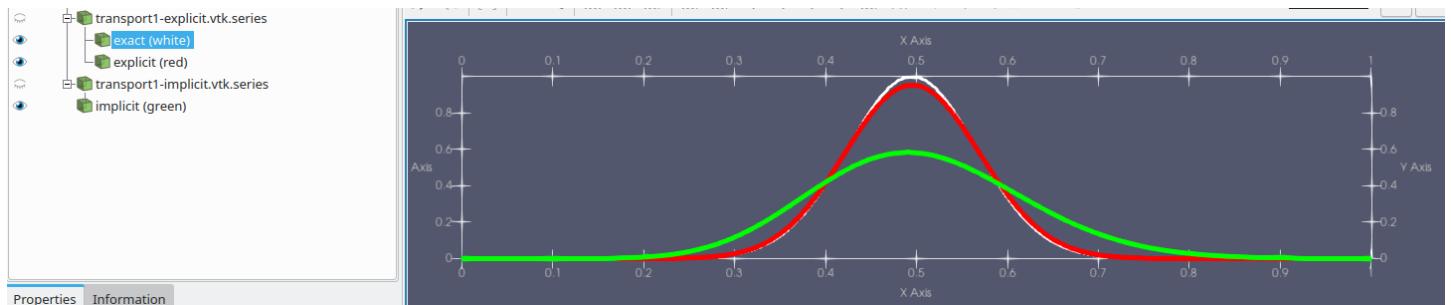


Рис. 7: Сравнение точного (белая линия) решения и численных решений по явной (красная) и неявной (зелёная) схемам

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. 8.

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).

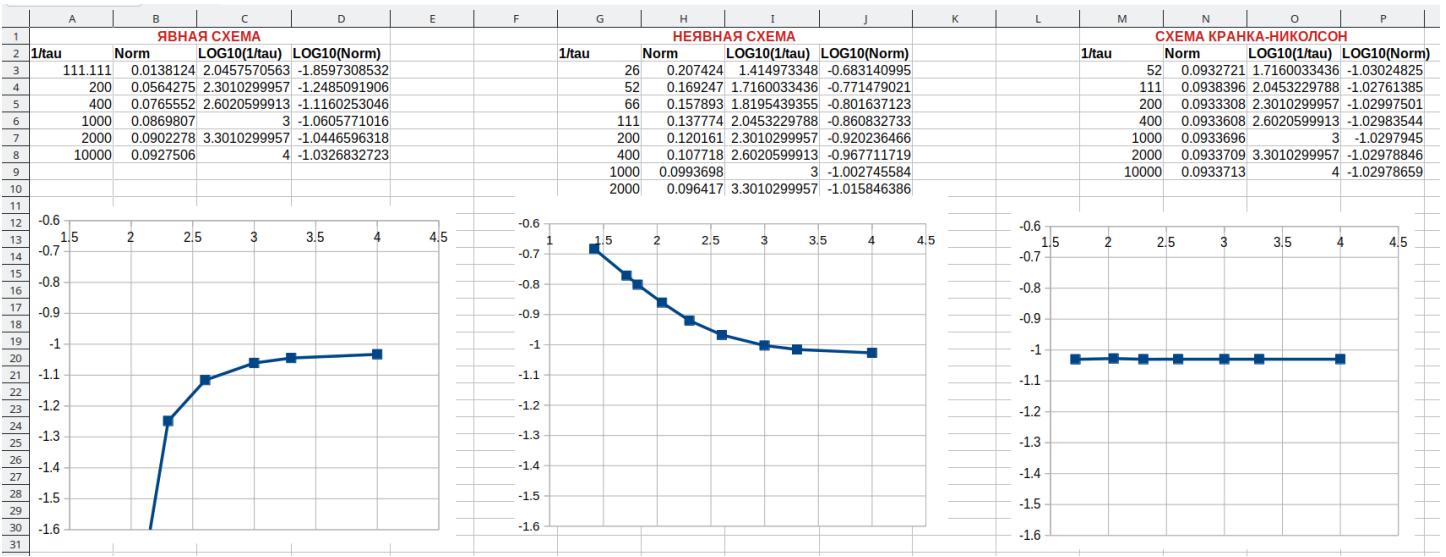


Рис. 8: Сходимость решения уравнения переноса с уменьшением τ

14.5 Задание для самостоятельной работы

14.5.1 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.

14.5.1.1 Тестовый пример 1

На этапе первичного тестирования использовать значения скорости

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

А в качестве начального решения брать простой "столбик" (рис. 9)

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

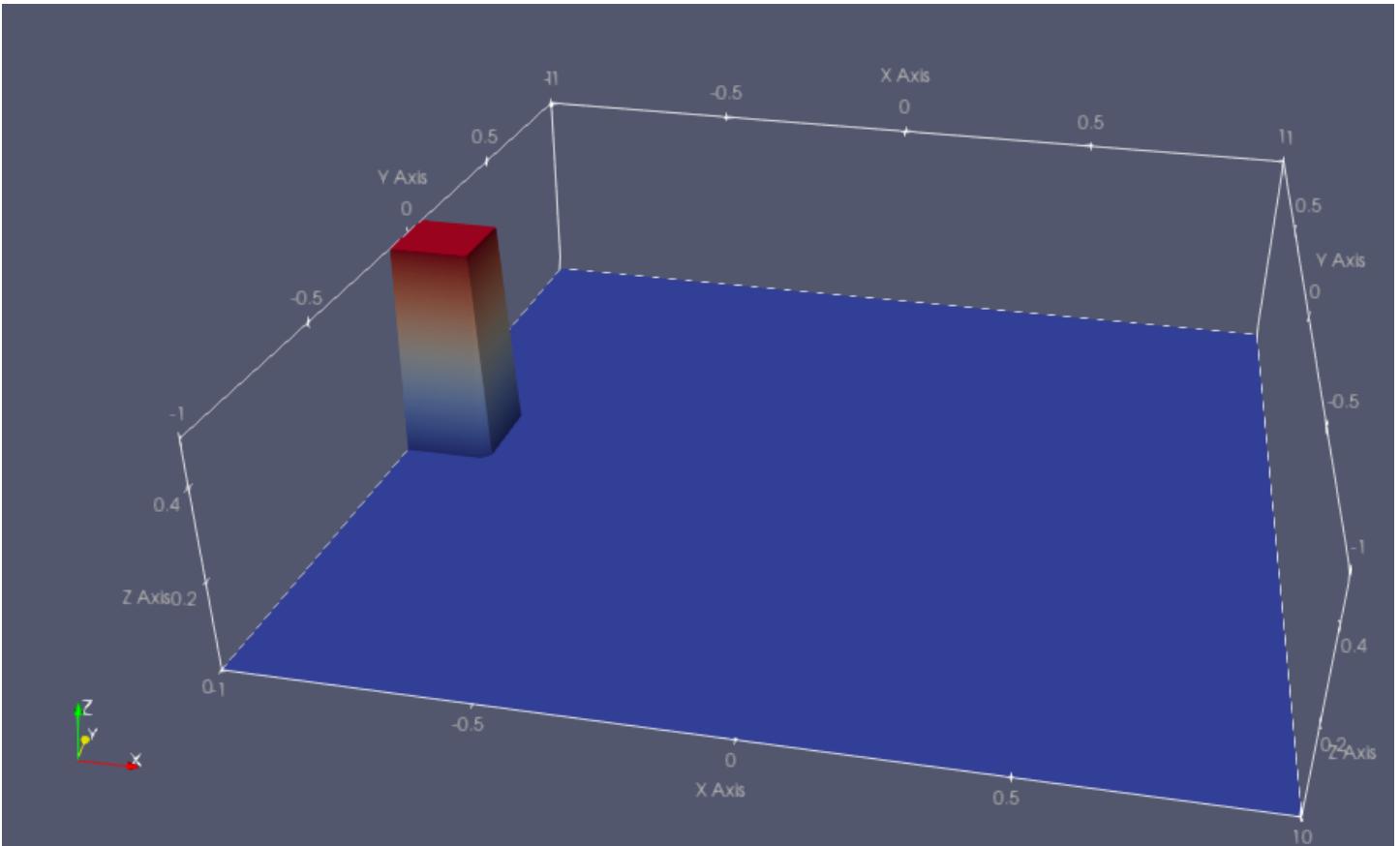


Рис. 9: Начальные условия для первого тестового примера

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.

14.5.1.2 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. 10)

$$\begin{aligned} r_0(x, y) &= \sqrt{(x - 0.5)^2 + y^2}; \quad \sigma = 0.05; \\ u(x, y, 0) &= u_0(x, y) = e^{-r_0^2(x, y)/\sigma^2} \end{aligned}$$

В процессе решения этот “холмик” будет двигаться по окружности с единичной скоростью, описывая полный оборот за время $t = \pi$.

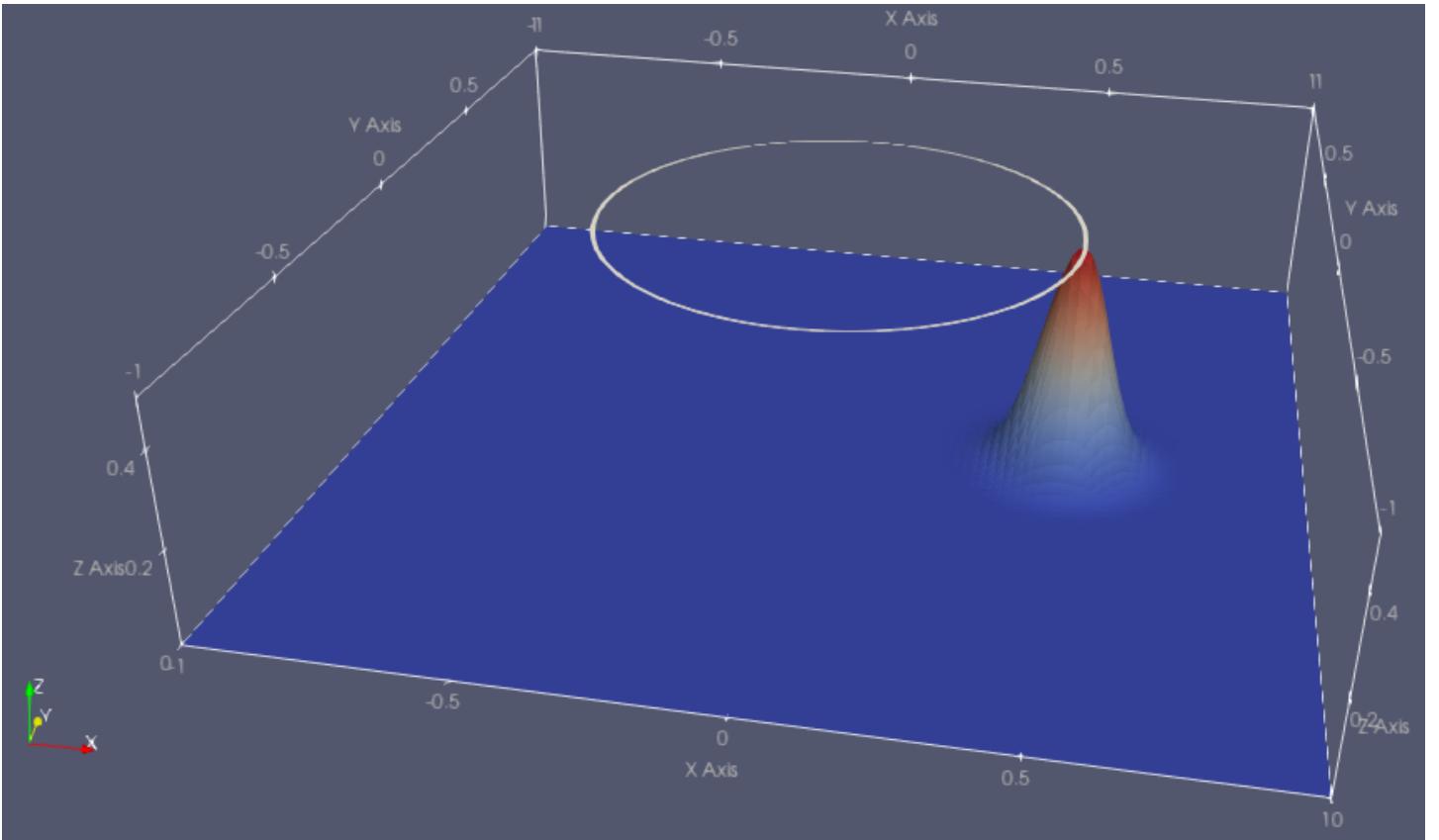


Рис. 10: Начальные условия для второго тестового примера

Точное решение на момент времени t будет иметь вид

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

14.5.2 Расчётная 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 (3.10) и обратно (3.11) 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.

15 Лекция 15 (09.11)

15.1 Аппроксимация уравнения переноса с ограничением потока

15.1.1 Схемы первого и второго порядка точности

Рассмотрим уравнение переноса в одномерной постановке

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

Все дальнейшие выкладки будем приводить исходя из условия положительности скорости переноса $U > 0$.

Рассмотрим два вида пространственной аппроксимации конвективного слагаемого на равномерной сетке: схемой против потока и симметричной схемой

$$\frac{\partial u_i}{\partial t} + U \frac{u_i - u_{i-1}}{h} = 0, \quad (15.1)$$

$$\frac{\partial u_i}{\partial t} + U \frac{u_{i+1} - u_{i-1}}{2h} = 0. \quad (15.2)$$

Первая схема является (условно) устойчивой но при этом обладает первым порядком аппроксимации. Вторая неустойчива, но имеет порядок $o(h^2)$. Идея методов аппроксимации с ограничением потока состоит в том, чтобы на основе комбинации первой и второй схем построить устойчивое решение, имеющее “почти везде” второй порядок аппроксимации.

Запишем эти аппроксимации в общем виде:

$$\frac{\partial u_i}{\partial t} = \frac{f_{i-1/2} - f_{i+1/2}}{h}. \quad (15.3)$$

Здесь $f_{i+1/2}$ – численный поток, который в зависимости от выбранной схемы будет равен

$$\begin{aligned} f_{i+1/2}^L &= U u_i && \text{– схема против потока} \\ f_{i+1/2}^H &= U \frac{u_i + u_{i+1}}{2} && \text{– симметричная схема.} \end{aligned} \quad (15.4)$$

Здесь f^L , f^H означают потоки низкого (Low) и высокого (High) порядка аппроксимации.

Аппроксимацию с ограничением потока запишем в виде

$$f_{i+1/2} = f_{i+1/2}^L + \Phi_{i+1/2} (f_{i+1/2}^H - f_{i+1/2}^L). \quad (15.5)$$

Φ в этой записи называется ограничителем, который служит переключателем: при $\Phi = 0$ мы получаем схему первого порядка, при $\Phi = 1$ – схему второго порядка.

Далее будем выбирать Φ таким образом, чтобы не допустить возникновения осцилляций в численном решении.

15.1.2 Условие TVD

В качестве критерия, характеризующего возникновение и развитие осцилляций, выберем полную вариацию:

$$\begin{aligned} TV(u) &= \int |\nabla u| dx = \\ &= \int \left| \frac{\partial u}{\partial x} \right| dx = \\ &= \sum_i |u_i - u_{i-1}|. \end{aligned} \quad \begin{array}{l} \text{в одномерном случае} \\ \text{для сеточной функции} \end{array} \quad (15.6)$$

Условие уменьшения осцилляций в решении на следующем временном слое примет вид

$$TV(\hat{u}) \leq TV(u).$$

Численные схемы, удовлетворяющие этому условию, называются TVD (Total variation diminishing) схемами.

Запишем численную схему в общем виде

$$\frac{\partial u_i}{\partial t} = c_{i-1/2}(u_{i-1} - u_i) + c_{i+1/2}(u_{i+1} - u_i). \quad (15.7)$$

Согласно теореме Хартена такая схема удовлетворяет свойству TVD, если $c_{i \pm 1/2} \geq 0$. Схема против потока (15.1) является TVD-схемой:

$$c_{i-1/2} = \frac{U}{h}, \quad c_{i+1/2} = 0,$$

а симметричная схема (15.2) – нет:

$$c_{i-1/2} = \frac{U}{2h}, \quad c_{i+1/2} = -\frac{U}{2h}.$$

Подставляя уравнение (15.5) в (15.3) и приводя к форме (15.7) получим

$$\frac{\partial u_i}{\partial t} = \frac{U}{2h} (2 - \Phi_{i-1/2})(u_{i-1} - u_i) + \frac{U}{2h} (-\Phi_{i+1/2})(u_{i+1} - u_i) \quad (15.8)$$

То есть для удовлетворения свойства TVD необходимо, чтобы $\Phi \leq 0$. Для второго порядка точности требуется $\Phi = 1$. То есть линейные схемы TVD не могут иметь высокий порядок точности.

15.1.3 Нелинейные TVD схемы

Для того, чтобы преодолеть это ограничение, будем строить нелинейные схемы. Общая идея построения таких схем состоит в том, чтобы выбрать такую Φ , при которой второе слагаемое равенства (15.8) можно было отнести к первому. То есть можно было записать

$$\Phi_{i+1/2}(u_{i+1} - u_i) = -\Phi'_{i+1/2}(u_{i-1} - u_i). \quad (15.9)$$

Тогда условием TVD станет выражение

$$2 - \Phi_{i-1/2} + \Phi'_{i+1/2} \geq 0. \quad (15.10)$$

Для характеристики поведения функции выберем соотношение наклонов (slope ratio), который в одномерном виде запишется в виде:

$$r_i = \frac{u_i - u_{i-1}}{u_{i+1} - u_i} \quad (15.11)$$

Нелинейность схемы будет выражаться в зависимости

$$\Phi_{i+1/2} = \Phi(r_i).$$

Из (15.9) следует

$$\Phi'_{i+1/2} = \frac{\Phi(r_i)}{r_i}$$

а неравенство перепишется в виде

$$2 - \Phi(r_i) + \frac{\Phi(r_i)}{r_i} \geq 0.$$

Чтобы из этого условия получить ограничение для Φ , явно не зависящее от r_i , потребуем

$$\Phi\left(\frac{1}{r_i}\right) = \frac{\Phi(r_i)}{r_i}. \quad (15.12)$$

Тогда неравенство (15.10) примет вид

$$2 - \Phi(r_i) + \Phi\left(\frac{1}{r_i}\right) \geq 0.$$

Отсюда получим условие для Φ :

$$0 \leq \Phi(r_i) \leq 2. \quad (15.13)$$

Дополнительно потребуем, чтобы в точках с гладким поведением функции использовать схему второго порядка точности:

$$\Phi(1) = 1 \quad (15.14)$$

а в точках локального экстремума (которые особенно подвержены появлению осцилляций) гарантировать переключение на схему первого порядка:

$$\Phi(r \leq 0) = 0. \quad (15.15)$$

Таким образом, для построения TVD схемы, функция ограничитель должна удовлетворять условиям (15.12) – (15.15). Ниже представлены некоторые часто используемые ограничители, удовлетво-

ряющие этим свойствам:

$$\Phi(r) = \begin{cases} \max(0, \min(r, 1)) & -\text{minmod;} \\ \frac{r + |r|}{1 + |r|} & -\text{Van Leer;} \\ \max(0, \min(2r, \frac{1+r}{2}, 2)) & -\text{monotonized central (MC);} \\ \max(0, \min(2, r), \min(1, 2r)) & -\text{superbee.} \end{cases} \quad (15.16)$$

15.2 TVD-схемы для неструктурированных конечнообъёмных сеток

Рассмотрим многомерное уравнение переноса

$$\frac{\partial u}{\partial t} + \mathbf{U} \cdot \nabla u = 0.$$

Применим конечнообъёмную процедуру для получения слабой интегральной постановки задачи. Для этого проинтегрируем это уравнение по конечному объёму E_i и применим формулу интегрирования по частям. Получим

$$|V_i| \frac{\partial u}{\partial t} + \sum_{j \in \text{nei}(i)} f_{ij} |\gamma_{ij}| = 0, \quad f_{ij} = u_{ij} U_{ij}. \quad (15.17)$$

Здесь $|V_i|$ – объём конечного элемента, $\text{nei}(i)$ – совокупность всех точек коллокации, инцидентных ячейке i (центров соседних ячеек и соседних граничных граней), f_{ij} – поток из точки коллокации i в точку коллокации j , $|\gamma_{ij}|$ – площадь грани конечного объёма i , через которую этот объём соединяется с точкой коллокации j , u_{ij} – значение функции u , отнесённое к этой грани, U_{ij} – скорость потока в направлении внешней по отношению к ячейке i нормали.

Для потока справедливо

$$f_{ij} = -f_{ji}. \quad (15.18)$$

То есть для вычисления потока на грани достаточно найти значение для одного направления. Выберем это направление \vec{ij} таким образом, чтобы $U_{ij} > 0$ (рис. 11).

Будем считать, что скорость переноса U – известная функция. Тогда запишем значения потоков высокого и низкого порядка согласно (15.4):

$$\begin{aligned} f_{ij}^L &= U_{ij} u_i && -\text{схема против потока} \\ f_{ij}^H &= U_{ij} \frac{u_i + u_j}{2} && -\text{симметричная схема.} \end{aligned} \quad (15.19)$$

Поток при этом запишется по аналогии с (15.5):

$$f_{ij} = f_{ij}^L + \Phi(r_{ij}) (f_{ij}^H - f_{ij}^L). \quad (15.20)$$

В одномерном случае для записи соотношения наклонов r_i (15.11) использовались три точки: текущий узел i , узел против потока $i - 1$, и узел по потоку $i + 1$. Для случаев неструктурированной сетки лишь две из этих трёх точек являются узлами коллокации: текущий узел i и узел по потоку

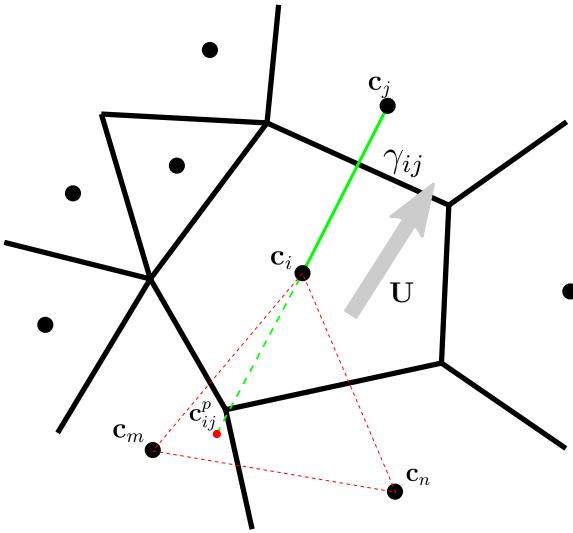


Рис. 11: Вспомогательный узел \mathbf{c}_{ij}^p на конечнообъёмной сетке

j . Определим точку против потока симметричным отражением: $\mathbf{c}_{ij}^p = 2\mathbf{c}_i - \mathbf{c}_j$ (см. рис. 11). Значение функции в этой точке обозначим как u_{ij}^p . Тогда соотношение наклонов запишется в виде

$$r_i = \frac{u_i - u_{ij}^p}{u_j - u_i} \quad (15.21)$$

Точка \mathbf{c}^p (в отличии от x_{i-1} из одномерного случая) не является точкой коллокации. То есть значение u^p нельзя достать из вектора столбца сеточной функции u . Однако, это значение можно интерполировать по значениям в ближайших точках коллокации.

15.2.1 Прямая интерполяция противопоточного значения

Так, в двумерном случае для определения u_{ij}^p необходимо найти три точки коллокации, ближайшие к точке \mathbf{c}_{ij}^p и не лежащие на одной прямой (или две точки коллокации, помимо c_i). На рис. 11 они помечены индексами \mathbf{c}_m , \mathbf{c}_n . И далее в треугольнике, образованном этими тремя точками (Δ_{imn}) провести интерполяцию по формуле (A.27). Специально отметим, что точка \mathbf{c}_{ij}^p не обязана содержаться внутри треугольника Δ_{imn} .

15.2.2 Интерполяция противопоточного значения через значение градиентов

. Другой подход к определению u_{ij}^p основан на записи симметричной конечной разности по направлению \mathbf{c}_{ij} :

$$\left. \frac{\partial u}{\partial c_{ij}} \right|_i = \frac{u_j - u_{ij}^p}{2|\mathbf{c}_{ij}|} \quad \Rightarrow \quad u_{ij}^p = u_j - 2|\mathbf{c}_{ij}| \left. \frac{\partial u}{\partial c_{ij}} \right|_i.$$

Производная по направлению c_{ij} находится как проекция градиента:

$$\left. |\mathbf{c}_{ij}| \frac{\partial u}{\partial c_{ij}} \right|_i = \mathbf{c}_{ij} \cdot (\nabla u)_i.$$

Таким образом, задача интерполяции сводится к задаче определения градиента функции u в узлах коллокации.

15.2.3 Определение градиентов в узлах коллокации. Метод наименьших квадратов

Будем рассматривать узел i , имеющий N_i соседних узлов j . Для каждого j можно записать линейное приближение

$$u_j = u_i + |\mathbf{c}_{ij}| \frac{\partial u}{\partial c_{ij}} = u_i + \mathbf{c}_{ij} \cdot \nabla u, \quad j = \overline{0, N_i - 1}.$$

Для двумерного случая можно записать:

$$(\mathbf{c}_{ij})_x \frac{\partial u}{\partial x} + (\mathbf{c}_{ij})_y \frac{\partial u}{\partial y} = u_j - u_i, \quad j = \overline{0, N_i - 1}.$$

Это выражение – есть система линейных уравнений с двумя неизвестными $\partial u / \partial x$, $\partial u / \partial y$ и N_i строками. Запишем её в матричном виде:

$$\begin{aligned} Ay = f, \quad \text{где} \quad A_{j0} &= (\mathbf{c}_{ij})_x & A_{j1} &= (\mathbf{c}_{ij})_y, \\ y_0 &= \partial u / \partial x & y_1 &= \partial u / \partial y, \\ f_j &= u_j - u_i . \end{aligned}$$

В двумерном случае размерность матрицы A есть $[N_i, 2]$ (для трёхмерной задачи следуя аналогичным рассуждениям получим матрицу с размерностью $[N_i, 3]$).

При этом в двумерном случае у конечного элемента будет минимум три грани (или четыре в трёхмерном случае). То есть $N_i \geq 3$ и полученная система имеет неизвестных больше, чем количество уравнений. Эта система в общем случае не имеет точного решения, но можно найти такие y , при котором невязка будет минимальной. Определим невязку как

$$r_i = \sum_{j=0}^{N_i} (A_{ij} y_j) - f_i, \quad i = 0, 1.$$

и будем минимизировать её квадрат

$$F = \sum_i r_i^2 \rightarrow \min$$

Запишем условие экстремума как

$$\frac{\partial F}{\partial y_i} = 2 \sum_j r_j \frac{\partial r_j}{\partial y_i} = 2 \sum_j \left(\sum_k (A_{jk} y_k) - f_j \right) A_{ji} = 0, \quad i = 0, 1.$$

Отсюда получим систему уравнений

$$\sum_j \left(A_{ji} \sum_k (A_{jk} y_k) \right) = \sum_j A_{ji} f_j = 0, \quad i = 0, 1.$$

Или, возвращаясь к матричной записи,

$$A^T A y = A^T f.$$

Полученная система имеет размерность 2×2 (или 3×3 в трёхмерном случае). Значение компонент

градиента в точке коллокации запишется как её прямое решение:

$$y = (A^T A)^{-1} A^T f.$$

Отметим, что матрица A зависит только от геометрии сетки. Поэтому в программной реализации матричное выражение $(A^T A)^{-1} A^T$ может быть расчитано один раз для каждого узла коллокации на этапе инициализации. Тогда определение градиента в центрах ячеек на этапе решения задачи сводится к сборке вектора f и умножении его на это выражение.

15.2.4 Реализация для явной схемы

Для примера рассмотрим написание TVD-схемы рассмотрим чисто явную схему для полудискретизованного уравнения (15.17):

$$|V_i| \frac{\hat{u} - u}{\tau} + \sum_{j \in \text{nei}(i)} f_{ij} |\gamma_{ij}| = 0. \quad (15.22)$$

Для того, чтобы избежать повторного вычисления потоков f_{ij} и f_{ji} , будем собирать эту схему в цикле по граням. Пусть через границу притока нет (то есть для граничные граней $f_{ij} = 0$). Тогда останется только цикл по внутренним граням:

$\hat{u} = u$	– инициализируем следующий шаг
for $s \in \text{internal}$	– цикл по внутренним граням
$i, j = \text{nei_cells}(s)$	– две ячейки, соседние с текущей гранью
\mathbf{U}_{ij}	– вектор скорости в центре грани
\mathbf{n}_{ij}	– вектор нормали к грани от ячейки i к j
$U_{ij} = \mathbf{U}_{ij} \cdot \mathbf{n}_{ij}$	– проекция скорости на нормаль
if $U_{ij} \leq 0$	– схема против потока
swap(i, j); $U_{ij} = -U_{ij}$	– гарантируем, что жидкость течет от i к j
endif	
$f_{ij}^L = U_{ij} u_i$	– поток 1-го порядка (15.19)
$f_{ij}^H = U_{ij} (u_i + u_j) / 2$	– поток 2-го порядка (15.19)
$\mathbf{c}_i, \mathbf{c}_j$	– центры ячеек
$\mathbf{c}^p = 2\mathbf{c}_i - \mathbf{c}_j$	– вспомогательная точка
$u^p = \text{interpolate}(i, j, u, \mathbf{c}^p)$	– интерполируем u в точке \mathbf{c}^p
$r = (u_i - u^p) / (u_j - u_i)$	– отношение наклонов (15.21)
$F = \text{limiter}(r)$	– ограничитель (15.16)
$f_{ij} = f_{ij}^L + F(f_{ij}^H - f_{ij}^L)$	– вычисление потока (15.20)
$\hat{u}_i = \hat{u} - \tau / V_i f_{ij} \gamma_{ij} $	– добавление в противопотоковую ячейку
$\hat{u}_j = \hat{u} + \tau / V_j f_{ij} \gamma_{ij} $	– добавление в попотоковую ячейку
endfor	

Отметим, что использование противоположенного знака при добавлении в правую от грани ячейку j связано с тождеством (15.18). То есть на самом деле в ячейку j должен был добавляться поток f_{ji} , но поскольку отдельной обработки этого направления не предусмотрено, мы добавляем f_{ij} с обратным знаком. При реализации функции interpolate должен использоваться один из методов, изложенных

в пп. 15.2.1, 15.2.2.

15.3 Задание для самостоятельной работы

В тесте `[transport2-fvm-upwind-explicit]` из файла `transport_fvm_solve_test.cpp` реализовано решение двумерного уравнения переноса по явной противопотоковой схеме. Реализация алгоритма в целом соответствует циклу (15.23) с упрощениями, следующими из отсутствия необходимости вычислять r в схеме против потока (где всегда $F = 0$).

Отталкиваясь от этого теста нужно решить двумерное уравнение переноса с помощью МКО аппроксимации на неструктурированной сетке. Использовать постановку из п. 14.5.1.2 с $\sigma = 0.1$. Рассмотреть схему против потока и МС TVD-схему пространственной аппроксимации и явную схему для дискретизации по времени. Для интерполяции противопотокового значения u^p использовать алгоритм 15.2.2.

Неструктурированную сетку строить с помощью скрипта `test_data/hexagrid.py`. Количество элементов сетки подобрать так, чтобы видеть эффект от применения схемы высокого порядка.

1. Проиллюстрировать динамику численного решения на неструктурной конечнообъемной сетке. Сравнить решение МС и Upwind.
2. Для выбранной сетки опытным путём определить максимально допустимый шаг по времени, при котором решение не разваливается;
3. Посчитать нормы отклонения полученного численного решения от известного точного (по максимуму и среднеквадратичную):

$$n_{max} = 1 - \max_i(u_i);$$
$$n_2 = \left(\frac{\sum_i (u_i - u^e(c_i))^2 |V_i|}{\sum_i |V_i|} \right)^{1/2}$$

Нарисовать графики этих норм от времени $n(t)$ для двух рассмотренных схем.

Рекомендации к программированию Для вычисления градиентов в центрах ячеек использовать класс `FvmCellGradient` из файла `fvm/fvm_assembler.hpp`, экземпляр которого нужно создать на этапе инициализации задачи. Тогда для вычисления градиентов от известной сеточной функции `u` достаточно вызывать метод `FvmCellGradient::compute`.

16 Лекция 16 (16.11)

16.1 Алгебраический подход к построению нелинейных TVD схем

TODO

16.2 Схемы с искусственной вязкостью

TODO

16.2.1 Направленная искусственная вязкость

TODO

16.2.2 SUPG

TODO

16.3 Задание для самостоятельной работы

В тестовом примере `[convdiff-fem-supg]` из файла `convdiff_fem_test.cpp` производится численное решение одномерного нестационарного уравнения конвекции-диффузии

$$\frac{\partial u}{\partial t} + v \frac{\partial u}{\partial x} - \varepsilon \frac{\partial^2 u}{\partial x^2} = 0$$

в области $x \in [0, 4]$ с точным решением вида

$$u^e(x, t) = \frac{1}{\sqrt{4\pi\varepsilon(t+t_0)}} \exp\left(-\frac{(x-vt)^2}{4\varepsilon(t+t_0)}\right)$$

Точное решение используется для формулировки начальных ($t = 0$) и граничных ($x = 0, 4$) условий первого рода. Результат расчёта сохраняется в файл `convdiff-supg.vtk.series`.

Задача полудискретизуется по схеме Кранка–Николсон с шагом по времени, вычисленным через число Куранта $C = 0.5$.

После дискретизации задача сводится к СЛАУ относительно неизвестного сеточного вектора u

$$\begin{aligned} Au &= B\ddot{u}, \\ A &= M + \tau\theta K + \varepsilon\tau\theta S, \\ B &= M - \tau(1-\theta)K - \varepsilon\tau(1-\theta)S. \end{aligned}$$

Здесь матрица масс M – результат коненочноэлементной аппроксимации единичного оператора, матрица переноса K – конвективного оператора, а матрица жёсткости S – оператора диффузии, τ – шаг по времени, и для схемы Кранка–Николсон $\theta = 0.5$.

Стабилизированные матрицы вычислялись по следующим формулам:

$$M = \int_{\Omega} \phi_j (\phi_i + s \mathbf{v} \cdot \nabla \phi_i) d\mathbf{x}$$

$$K = \int_{\Omega} \mathbf{v} \cdot \nabla \phi_j (\phi_i + s \mathbf{v} \cdot \nabla \phi_i) d\mathbf{x}$$

$$\begin{aligned} S &= - \int_{\Omega} \nabla^2 \phi_j (\phi_i + s \mathbf{v} \cdot \nabla \phi_i) d\mathbf{x} \\ &= \int_{\Omega} \nabla \phi_j \cdot \nabla (\phi_i + s \mathbf{v} \cdot \nabla \phi_i) d\mathbf{x} \\ &= \int_{\Omega} \nabla \phi_j \cdot \nabla \phi_i d\mathbf{x} \end{aligned} \quad \left. \begin{array}{l} \text{по формуле (A.12)} \\ \text{если } \phi_i \text{ линейны, то } \nabla^2 \phi_i = 0 \end{array} \right\}$$

где $s = \mu h / |\mathbf{v}|^2$, h – характерный линейный размер элемента, а μ – параметр SUPG-стабилизации.

Сборки необходимых матриц осуществляется в процедуре `assemble_solver`. Локальные матрицы для операторов M , K вычисляются с помощью численного интегрирования в процедуре `custom_matrix`, которой в качестве аргумента передаётся функция подинтегрального выражения.

Необходимо:

1. В одномерном тесте `[convdiff-supg]` с помощью анимированных графиков продемонстрировать наличие осцилляций при выбранных параметрах решения при отсутствии стабилизации `mu_supg = 0`, а так же эффект от SUPG-слагаемого (`mu_supg > 0`);
2. Нарисовать в сравнении результаты расчёта на конечный промежуток времени для различных μ ;
3. Подобрать оптимальную величину параметра μ , минимизирующую норму отклонения численного решения от точного;
4. Написать аналогичный тест для двумерного случая (решение в единичном квадрате) с точным решением

$$u^e(x, t) = \frac{1}{4\pi\varepsilon(t+t_0)} \exp\left(-\frac{(x-v_xt)^2 + (y-v_yt)^2}{4\varepsilon(t+t_0)}\right)$$

Использовать $\mathbf{v} = (1, 1)$. Взять треугольную сетку, построенную процедурой `trigrid.py`. Проанализировать результаты, полученные на грубой и подробной сетке.

В целом имеющийся решатель не зависит от геометрической размерности задачи. Для 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);

17 Лекция 17 (23.11)

17.1 Моделирование течения вязкой несжимаемой жидкости методом конечных разностей

17.1.1 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

$$\frac{\partial u^2}{\partial x} + \frac{\partial uv}{\partial y} = -\frac{\partial p}{\partial x} + \frac{1}{Re} \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right), \quad (17.1)$$

$$\frac{\partial uv}{\partial x} + \frac{\partial v^2}{\partial y} = -\frac{\partial p}{\partial y} + \frac{1}{Re} \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right), \quad (17.2)$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0. \quad (17.3)$$

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 ρ , [кг/м³] – постоянная (вследствии несжимаемости) плотность жидкости, а μ , [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 (17.1) – (17.3) можно выделить несколько проблем, которые необходимо решить при построении расчётной схемы:

- нелинейность конвективного оператора в (17.1), (17.2),
- 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).

17.1.2 Схема расчёта

Стационарную задачу (17.1)-(17.3) будем решать методом установления. Для этого в первые два уравнения введём фиктивную производную по времени, которую распишем по неявной двухслойной схеме с шагом τ . Тогда задача на одном итерационном слое примет вид

$$\frac{\hat{u} - u}{\tau} + \frac{\partial u \hat{u}}{\partial x} + \frac{\partial v \hat{u}}{\partial y} = -\frac{\partial \hat{p}}{\partial x} + \frac{1}{\text{Re}} \left(\frac{\partial^2 \hat{u}}{\partial x^2} + \frac{\partial^2 \hat{u}}{\partial y^2} \right), \quad (17.4)$$

$$\frac{\hat{v} - v}{\tau} + \frac{\partial u \hat{v}}{\partial x} + \frac{\partial v \hat{v}}{\partial y} = -\frac{\partial \hat{p}}{\partial y} + \frac{1}{\text{Re}} \left(\frac{\partial^2 \hat{v}}{\partial x^2} + \frac{\partial^2 \hat{v}}{\partial y^2} \right), \quad (17.5)$$

$$\frac{\partial \hat{u}}{\partial x} + \frac{\partial \hat{v}}{\partial y} = 0. \quad (17.6)$$

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:

$$\begin{aligned} r_u &= \frac{\partial \hat{u} \hat{u}}{\partial x} + \frac{\partial \hat{u} \hat{v}}{\partial y} + \frac{\partial \hat{p}}{\partial x} - \frac{1}{\text{Re}} \left(\frac{\partial^2 \hat{u}}{\partial x^2} + \frac{\partial^2 \hat{u}}{\partial y^2} \right), \\ r_v &= \frac{\partial \hat{u} \hat{v}}{\partial x} + \frac{\partial \hat{v} \hat{v}}{\partial y} + \frac{\partial \hat{p}}{\partial y} - \frac{1}{\text{Re}} \left(\frac{\partial^2 \hat{v}}{\partial x^2} + \frac{\partial^2 \hat{v}}{\partial y^2} \right), \\ \max(\|r_u\|, \|r_v\|) &< \varepsilon. \end{aligned} \quad (17.7)$$

17.1.2.1 Метод SIMPLE

Приведём алгоритм для явного выражения уравнения для давления из уравнения неразрывности (17.6).

Распишем искомые переносные в виде суммы

$$\begin{aligned} \hat{u} &= u^* + u', \\ \hat{v} &= v^* + v', \\ \hat{p} &= p + p'. \end{aligned} \quad (17.8)$$

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

$$u^* + \tau \frac{\partial u u^*}{\partial x} + \tau \frac{\partial v u^*}{\partial y} - \frac{\tau}{\text{Re}} \left(\frac{\partial^2 u^*}{\partial x^2} + \frac{\partial^2 u^*}{\partial y^2} \right) = -\tau \frac{\partial p}{\partial x} + u, \quad (17.9)$$

$$v^* + \tau \frac{\partial u v^*}{\partial x} + \tau \frac{\partial v v^*}{\partial y} - \frac{\tau}{\text{Re}} \left(\frac{\partial^2 v^*}{\partial x^2} + \frac{\partial^2 v^*}{\partial y^2} \right) = -\tau \frac{\partial p}{\partial y} + v. \quad (17.10)$$

Тогда уравнение для поправки u' запишем вычтя последнее выражение из уравнения (17.4), умноженного на τ :

$$u' + \tau \frac{\partial uu'}{\partial x} + \tau \frac{\partial vu'}{\partial y} - \frac{\tau}{\text{Re}} \left(\frac{\partial^2 u'}{\partial x^2} + \frac{\partial^2 \hat{u}'}{\partial y^2} \right) = -\tau \frac{\partial p'}{\partial x}. \quad (17.11)$$

Основная идея алгоритма SIMPLE заключается в приближённом представлении выражения (17.11) 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 :

$$u' \approx -\tau d^u(x, y) \frac{\partial p'}{\partial x}. \quad (17.12)$$

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

$$v' \approx -\tau d^v(x, y) \frac{\partial p'}{\partial y}. \quad (17.13)$$

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

Далее используем уравнение неразрывности (17.6). Подставим в него разложения (17.8) и используем (17.12), (17.13). Тогда получим уравнение Пуассона с непостоянным по пространству векторным коэффициентом диффузии (d^u, d^v) относительно поправки давления p' :

$$-\left[\frac{\partial}{\partial x} \left(d^u \frac{\partial p'}{\partial x} \right) + \frac{\partial}{\partial y} \left(d^v \frac{\partial p'}{\partial y} \right) \right] = -\frac{1}{\tau} \left(\frac{\partial u^*}{\partial x} + \frac{\partial v^*}{\partial y} \right). \quad (17.14)$$

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 (17.9), (17.10) вычисляются значения u^*, v^* ;
2. Они используются для вычисления правой части уравнения (17.14), в результате решения которого находится поправка давления p' ;
3. Дифференцируя найденную поправку давления найдём поправки скорости u', v' из выражений (17.12), (17.13);
4. Окончательно выразим значения переменных для текущего слоя из (17.8). Для улучшения стабильности алгоритма значение давления вычисляют с некоторым коэффициентом релаксации α_p :

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

5. Далее проводится вычисление невязки с использованием найденных значений $\hat{u}, \hat{v}, \hat{p}$ из выражения (17.7). 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 (17.6) в “чёрных” узлах сетки, но уравнения движения (17.4), (17.5) выполняются лишь приближённо.

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

17.1.3 Пространственная аппроксимация

Для численной реализации алгоритма решения необходимо провести пространственную аппроксимацию полудискретизованных выражений (17.9), (17.10), (17.12), (17.13), (17.14).

17.1.3.1 Разнесённая сетка

Будем использовать структурированную четырёхугольную сетку с постоянным шагом по пространству. При этом неизвестные параметры будем задавать по схеме, представленной на рис. 12.

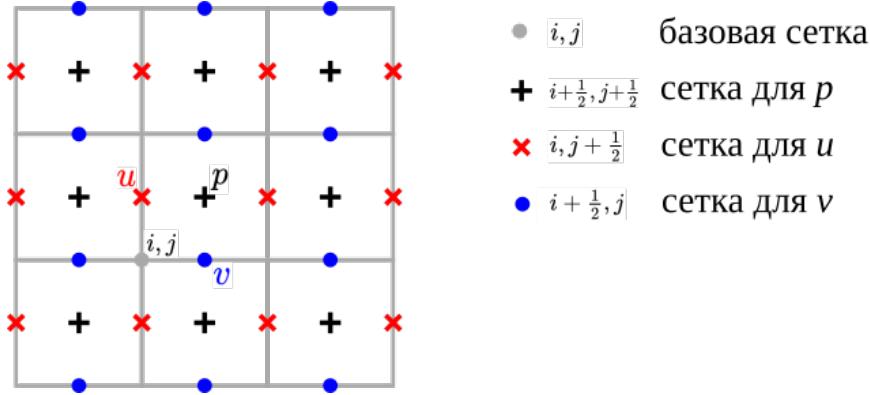


Рис. 12: Разнесённая сетка

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 (17.9), (17.12) аппроксимируются на сетке для u , выражения (17.10), (17.13) — на сетке для v , а (17.14) — 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

$$k(i, j) = j(n_x + 1) + i. \quad (17.15)$$

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

$$k(i + \frac{1}{2}, j + \frac{1}{2}) = jn_x + i, \quad - \text{сетка для давления } p \quad (17.16)$$

$$k(i, j + \frac{1}{2}) = j(n_x + 1) + i, \quad - \text{сетка для продольной скорости } u \quad (17.17)$$

$$k(i + \frac{1}{2}, j) = jn_x + i, \quad - \text{сетка для поперечной скорости } v \quad (17.18)$$

17.1.3.2 Уравнения движения

Запишем конечноразностную аппроксимацию уравнения (17.9) для пробной скорости u^* в “красных” узлах сетки $(i, j + \frac{1}{2})$:

$$\begin{aligned} & u_{i,j+\frac{1}{2}}^* + \frac{\tau}{h_x} \left((uu^*)_{i+\frac{1}{2},j+\frac{1}{2}} - (uu^*)_{i-\frac{1}{2},j+\frac{1}{2}} \right) \\ & + \frac{\tau}{h_y} \left((vu^*)_{i,j+1} - (vu^*)_{i,j} \right) \\ & - \frac{1}{\text{Re}} \frac{\tau}{h_x^2} \left(u_{i-1,j+\frac{1}{2}}^* - 2u_{i,j+\frac{1}{2}}^* + u_{i+1,j+\frac{1}{2}}^* \right) \\ & - \frac{1}{\text{Re}} \frac{\tau}{h_y^2} \left(u_{i,j-\frac{1}{2}}^* - 2u_{i,j+\frac{1}{2}}^* + u_{i,j+\frac{3}{2}}^* \right) \\ & = u_{i,j+\frac{1}{2}} - \frac{\tau}{h_x} \left(p_{i+\frac{1}{2},j+\frac{1}{2}} - p_{i-\frac{1}{2},j+\frac{1}{2}} \right). \end{aligned} \quad (17.19)$$

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 (17.19) можно записать в виде системы линейных уравнений вида

$$A^u u^* = b^u. \quad (17.20)$$

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 (17.19). Множитель при диагональном элементе $u_{i,j+\frac{1}{2}}^*$ будет равен:

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

Сумма первого и четвёртого слагаемых представляет собой разностный аналог уравнения неразрывности (17.6), 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:

$$A^u [k(i, j + \frac{1}{2}), k(i, j + \frac{1}{2})] = 1 + \frac{2\tau}{\text{Re}} \left(\frac{1}{h_x^2} + \frac{1}{h_y^2} \right) \quad - \text{ основная диагональ,} \quad (17.21)$$

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

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

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

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

Здесь $k(i, j)$ – функция перевода двумерного индекса в сквозной (17.17).

Аналогичные выкладки для второго из уравнений движения (17.10) дают систему уравнений

$$A^v v^* = b^v, \quad (17.22)$$

элементы пятидиагональной матрицы которой имеют вид

$$\begin{aligned} A^v [k(i + \frac{1}{2}, j), k(i + \frac{1}{2}, j)] &= 1 + \frac{2\tau}{\operatorname{Re}} \left(\frac{1}{h_x^2} + \frac{1}{h_y^2} \right) \quad - \text{ основная диагональ,} \\ A^v [k(i + \frac{1}{2}, j), k(i + \frac{3}{2}, j)] &= -\frac{\tau}{\operatorname{Re}} \frac{1}{h_x^2} + \frac{\tau}{4h_x} \left(u_{i+1,j+\frac{1}{2}} + u_{i+1,j-\frac{1}{2}} \right) \quad - \text{ первая верхняя диагональ,} \\ A^v [k(i + \frac{1}{2}, j), k(i - \frac{1}{2}, j)] &= -\frac{\tau}{\operatorname{Re}} \frac{1}{h_x^2} - \frac{\tau}{4h_x} \left(u_{i,j+\frac{1}{2}} + u_{i,j-\frac{1}{2}} \right) \quad - \text{ первая нижняя диагональ,} \\ A^v [k(i + \frac{1}{2}, j), k(i + \frac{1}{2}, j + 1)] &= -\frac{\tau}{\operatorname{Re}} \frac{1}{h_y^2} + \frac{\tau}{4h_y} \left(v_{i+\frac{1}{2},j} + v_{i+\frac{1}{2},j+1} \right) \quad - \text{ вторая верхняя диагональ,} \\ A^v [k(i + \frac{1}{2}, j), k(i + \frac{1}{2}, j - 1)] &= -\frac{\tau}{\operatorname{Re}} \frac{1}{h_y^2} - \frac{\tau}{4h_y} \left(v_{i+\frac{1}{2},j} + v_{i+\frac{1}{2},j-1} \right) \quad - \text{ вторая нижняя диагональ.} \end{aligned} \quad (17.23)$$

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 (17.18).

17.1.3.3 Уравнение для поправки давления

Распишем уравнение (17.14) на “чёрной” сетке методом конечных разностей. Для первого слагаемого получим

$$\begin{aligned} \frac{\partial}{\partial x} \left(d^u \frac{\partial p'}{\partial x} \right) \Big|_{i+\frac{1}{2},j+\frac{1}{2}} &\approx \frac{1}{h_x} \left(d^u_{i+1,j+\frac{1}{2}} \frac{\partial p'}{\partial x} \Big|_{i+1,j+\frac{1}{2}} - d^u_{i,j+\frac{1}{2}} \frac{\partial p'}{\partial x} \Big|_{i,j+\frac{1}{2}} \right) \\ &= \frac{1}{h_x} \left(d^u_{i+1,j+\frac{1}{2}} \frac{p'_{i+\frac{3}{2},j+\frac{1}{2}} - p'_{i+\frac{1}{2},j+\frac{1}{2}}}{h_x} - d^u_{i,j+\frac{1}{2}} \frac{p'_{i+\frac{1}{2},j+\frac{1}{2}} - p'_{i-\frac{1}{2},j+\frac{1}{2}}}{h_x} \right). \end{aligned} \quad (17.24)$$

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

$$A^p p' = b^p, \quad (17.25)$$

где ненулевые коэффициенты пятидиагональной матрицы примут вид

$$\begin{aligned}
A^p[k(i + \frac{1}{2}, j + \frac{1}{2}), k(i + \frac{1}{2}, j + \frac{1}{2})] &= \frac{1}{h_x^2} \left(d_{i+1,j+\frac{1}{2}}^u + d_{i,j+\frac{1}{2}}^u \right) + \frac{1}{h_y^2} \left(d_{i+\frac{1}{2},j}^v + d_{i+\frac{1}{2},j+1}^v \right), \\
A^p[k(i + \frac{1}{2}, j + \frac{1}{2}), k(i + \frac{3}{2}, j + \frac{1}{2})] &= -\frac{1}{h_x^2} d_{i+1,j+\frac{1}{2}}^u, \\
A^p[k(i + \frac{1}{2}, j + \frac{1}{2}), k(i - \frac{1}{2}, j + \frac{1}{2})] &= -\frac{1}{h_x^2} d_{i,j+\frac{1}{2}}^u, \\
A^p[k(i + \frac{1}{2}, j + \frac{1}{2}), k(i + \frac{1}{2}, j + \frac{3}{2})] &= -\frac{1}{h_y^2} d_{i+\frac{1}{2},j+1}^v, \\
A^p[k(i + \frac{1}{2}, j + \frac{1}{2}), k(i + \frac{1}{2}, j - \frac{1}{2})] &= -\frac{1}{h_y^2} d_{i+\frac{1}{2},j}^v.
\end{aligned} \tag{17.26}$$

Столбец свободных членов аппроксимируется в виде

$$b^p[k(i + \frac{1}{2}, j + \frac{1}{2})] = -\frac{1}{\tau} \left(\frac{u_{i+1,j+\frac{1}{2}}^* - u_{i,j+\frac{1}{2}}^*}{h_x} + \frac{v_{i+\frac{1}{2},j+1}^* - v_{i+\frac{1}{2},j}^*}{h_y} \right). \tag{17.28}$$

Здесь используется функция перевода двумерного индекса в сквозной из (17.16).

Далее определим значения d^u, d^v . Согласно идее алгоритма SIMPLE d^u должна быть такой функцией, которая максимально приближает выражение (17.11) к (17.12).

Пространственная аппроксимация выражения (17.11) 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 (17.19).

Сравнивая предыдущее выражение с (17.12) 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 (17.21) можно положить

$$d^u = (\text{diag}(A^u))^{-1} = \left(1 + \frac{2\tau}{\text{Re}} \left(\frac{1}{h_x^2} + \frac{1}{h_y^2} \right) \right)^{-1} \tag{17.29}$$

и аналогично из (17.23)

$$d^v = (\text{diag}(A^v))^{-1} = \left(1 + \frac{2\tau}{\text{Re}} \left(\frac{1}{h_x^2} + \frac{1}{h_y^2} \right) \right)^{-1}. \tag{17.30}$$

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 (17.29), (17.30) левая часть системы уравнений (17.25) 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 (17.20), (17.22), 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 (17.25).

17.1.3.4 Уравнение для поправки скорости

И наконец рассмотрим аппроксимацию выражений (17.12), (17.13), которые примут явный вид

$$u'_{i,j+\frac{1}{2}} = -\tau d^u_{i,j+\frac{1}{2}} \frac{p'_{i+\frac{1}{2},j+\frac{1}{2}} - p'_{i-\frac{1}{2},j+\frac{1}{2}}}{h_x}, \quad (17.31)$$

$$v'_{i+\frac{1}{2},j} = -\tau d^v_{i+\frac{1}{2},j} \frac{p'_{i+\frac{1}{2},j+\frac{1}{2}} - p'_{i+\frac{1}{2},j-\frac{1}{2}}}{h_x}. \quad (17.32)$$

17.1.3.5 Учёт граничных условий

Для уравнений Навье-Стокса на каждой границе расчётной области требуется столько условий, сколько есть уравнений движения. Для двумерной задачи (17.1) – (17.3) 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):

$$\begin{aligned} u^*(x, y)|_{x,y \in \Gamma} &= u^\Gamma(x, y), \\ v^*(x, y)|_{x,y \in \Gamma} &= v^\Gamma(x, y), \\ u'(x, y)|_{x,y \in \Gamma} &= 0, \\ v'(x, y)|_{x,y \in \Gamma} &= 0. \end{aligned} \tag{17.33}$$

Для учёта граничных условий по скорости требуется модифицировать системы линейных уравнений (17.20), (17.22).

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 (17.22):

$$A^v[k(i + \frac{1}{2}, 0), s] = \delta_{ks}, \quad \forall i, \forall s \tag{17.34}$$

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

Такая модификация просто заменяет $k(i + \frac{1}{2}, 0)$ -ое уравнение системы (17.22) 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 (17.20) по выведенным выше формулам (17.21) для узла, расположенного в этом ряду, мы бы столкнулись с необходимостью установки значения в фиктивную колонку: последнее из уравнений (17.21) 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 (17.21) имеет вид

$$Du^*_{i, \frac{1}{2}} + U^1 u^*_{i+1, \frac{1}{2}} + L^1 u^*_{i-1, \frac{1}{2}} + U^2 u^*_{i, \frac{3}{2}} + L^2 u^*_{i, -\frac{1}{2}} = b^u_{i, \frac{1}{2}}, \tag{17.35}$$

где D – коэффициент с основной диагональю, $U^{1,2}, L^{1,2}$ – коэффициенты с двух верхних и двух нижних диагоналях, вычисляемые по формулам (17.21). 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 (17.35) 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

$$A^u[k(i, \frac{1}{2}), k(i, \frac{1}{2})] = A^u[k(i, -\frac{1}{2})], \quad (17.36)$$

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

Приёмы (17.34), (17.36) используются и на остальных границах для постановки граничных условий для скорости.

При сборке системы линейных уравнений для поправки давления (17.25) так же возникает проблема с обращением к фиктивным узлам. Например, при рассмотрении левой стенки ($i = 0$ третье из уравнений (17.26) описывает несуществующий столбец $k(-\frac{1}{2}, j + \frac{1}{2})$). Если обратиться к выражению (17.24), то будет видно, что это слагаемое пришло в результате расписывания граничной производной p' , которая, исходя из выражения (17.12) пропорциональна граничному значению u' , то есть, вспоминая (17.33), 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 (17.26), просто модифицируем значения коэффициентов d^u, d^v :

$$d^u_{0,j+\frac{1}{2}} = d^u_{n_x+1,j+\frac{1}{2}} = 0, \quad (17.37)$$

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

В исходных уравнениях (17.1)-(17.3) 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 нулью в любом узле. Например, в случае нулевого узла, по аналогии с (17.34) запишем:

$$A^p[k(\frac{1}{2}, \frac{1}{2}), s] = \delta_{ks}, \quad (17.38)$$

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

17.1.4 Оптимальные значения параметров алгоритма SIMPLE

Введем обозначение

$$E = \frac{4\tau}{\text{Re } \bar{h}^2}$$

где \bar{h}^2 – среднее гармоническое значение шага, определяемое как

$$\bar{h}^2 = \frac{2 h_x^2 h_y^2}{h_x^2 + h_y^2}.$$

Значение E – есть диагональное компонента матрицы диффузии в аппроксимированных уравнениях движения (второе слагаемое в правой части первой формулы (17.21)).

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

$$E = 1 \quad \Rightarrow \quad \tau = \frac{\text{Re } \bar{h}^2}{4}, \\ \alpha_p = 0.8.$$

Ещё более эффективной сходимости соответствуют параметры

$$E \approx 4, \quad \alpha_p = \frac{1}{1+E}. \quad (17.39)$$

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

17.2 Программа для расчёта течения в каверне по схеме SIMPLE

17.2.1 Постановка задачи

Для иллюстрации работы алгоритма рассмотрим задачу о течении в каверне. Постановку задачи представлена на рис. 13.

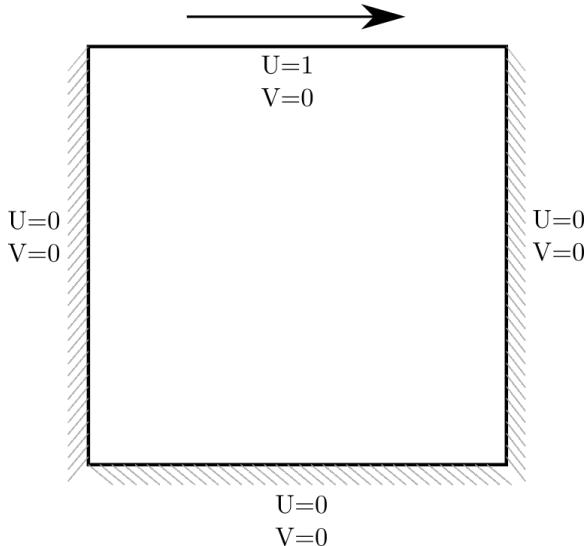


Рис. 13: Область расчёта задачи о каверне

Задача реализована в тесте `[cavity2-simple]` в файле `cavity_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 `cavity2.vtk.series`.

Итоговый результат (для $\varepsilon = 10^{-2}$) представлен на рис. 14.

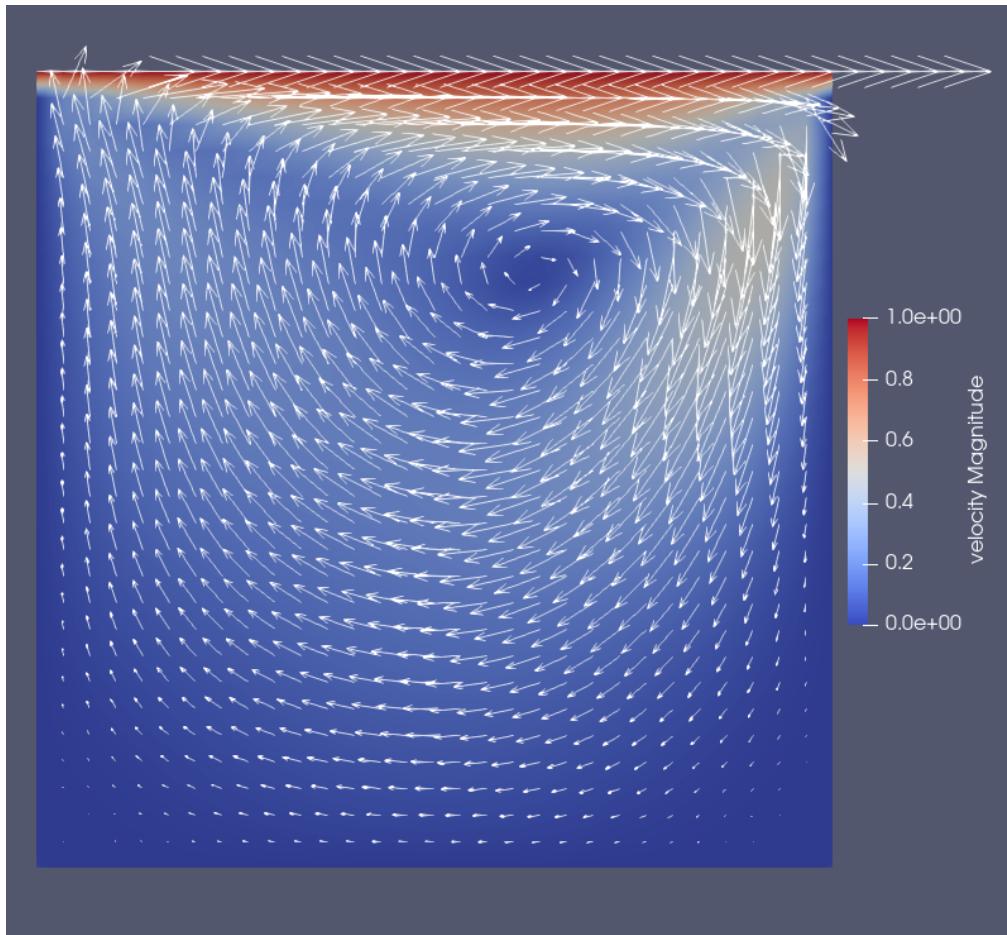


Рис. 14: Область расчёта задачи о каверне

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 ([17.16](#)),
- `cfd::RegularGrid2D::xface_grid_index_ip_j` – посчитать линейный индекс “синей” сетки ([17.18](#)),

- `cfd::RegularGrid2D::yface_grid_index_i_jp` – посчитать линейный индекс “красной” сетки (17.17).

17.2.2 Функция верхнего уровня

```
453 TEST_CASE("Cavity 2D, SIMPLE algorithm", "[cavity2-simple]"){

```

Сначала устанавливаются параметры задачи: число Рейнольдса,

```
457     double Re = 100;
```

параметры алгоритма SIMPLE,

```
458     double tau = 0.03;
459     double alpha = 0.8;
```

разбиение сетки,

```
460     size_t n_cells = 30;
```

максимальное количество итераций

```
461     size_t max_it = 10000;
```

и значение невязки, при котором итерации прекращаются

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

Затем происходит инициализация решателя, который определён в классе `Cavity2DSimpleWorker`

```
465     Cavity2DSimpleWorker 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 `cavity2.vtk.series`, будут создаваться также файлы `cavity2-u`, `cavity2-v`, `cavity2-p`.

```
466     worker.initialize_saver(false, "cavity2");
```

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

```
469 std::vector<double> u_init(worker.u_size(), 0.0);
470 std::vector<double> v_init(worker.v_size(), 0.0);
471 std::vector<double> p_init(worker.p_size(), 0.0);
472 worker.set_upv(u_init, v_init, p_init);
```

и начинается итерационный процесс.

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

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

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

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

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

Сохраняется состояние решателя на пройденную итерацию

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

и производится проверка на сходимость

```
487 if (nrm < eps){
488     break;
489 }
```

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

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

17.2.3 Поля класса решателя

Класс `Cavity2DSimpleWorker` 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. 12).

```
35 const RegularGrid2D _grid;
36 const RegularGrid2D _cc_grid;
37 const RegularGrid2D _xf_grid;
38 const RegularGrid2D _yf_grid;
```

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

```
39 const double _hx;
40 const double _hy;
41 const double _Re;
42 const double _tau;
43 const double _alpha_p;
```

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

```
45 std::vector<double> _p;
46 std::vector<double> _u;
47 std::vector<double> _v;
```

Также определяется данные для решения системы уравнений для нахождения p' (17.25): значения d^u, d^v , а так же инициализированный решатель системы уравнений. Поскольку используется постоянные шаги по времени, d^u, d^v являются скалярами.

```
49 double _du;
50 double _dv;
51 AmgMatrixSolver _p_prime_solver;
```

Хранятся левая и правая части систем уравнений (17.20), (17.22) для определения пробных значений скорости и расчета невязки.

```

53     CsrMatrix _mat_u;
54     CsrMatrix _mat_v;
55     std::vector<double> _rhs_u;
56     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.

```

58     std::shared_ptr<VtkUtils::TimeSeriesWriter> _writer_u;
59     std::shared_ptr<VtkUtils::TimeSeriesWriter> _writer_v;
60     std::shared_ptr<VtkUtils::TimeSeriesWriter> _writer_p;
61     std::shared_ptr<VtkUtils::TimeSeriesWriter> _writer_all;

```

17.2.4 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 (17.29), (17.30) 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.

```

80 Cavity2DSimpleWorker::Cavity2DSimpleWorker(double Re, size_t n_cells, double tau,
81     → double alpha_p):
82     _grid(0, 1, 0, 1, n_cells, n_cells),
83     _cc_grid(_grid.cell_centered_grid()),
84     _xf_grid(_grid.xface_centered_grid()),
85     _yf_grid(_grid.yface_centered_grid()),
86     _hx(1.0/n_cells),
87     _hy(1.0/n_cells),
88     _Re(Re),
89     _tau(tau),
90     _alpha_p(alpha_p)
91 {
92     _du = 1.0 / (1 + 2.0*_tau/_Re * (1.0/_hx/_hx + 1.0/_hy/_hy));
93     _dv = 1.0 / (1 + 2.0*_tau/_Re * (1.0/_hx/_hx + 1.0/_hy/_hy));
94     assemble_p_prime_solver();

```

Начальные значения устанавливаются через вызов функции `set_uvp`. 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.

```
105 double Cavity2DSimpleWorker::set_uvp(const std::vector<double>& u, const
106   std::vector<double>& v, const std::vector<double>& p){
107   _u = u;
108   _v = v;
109   _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 (17.20), (17.22).

```
109 assemble_u_slae();
110 assemble_v_slae();
```

Если посмотреть на выражение для невязки (17.7) 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:

```
112 // residuals
113 auto r_u = compute_residual_vec(_mat_u, _rhs_u, _u);
114 auto r_v = compute_residual_vec(_mat_v, _rhs_v, _v);
115 double nrm_u = (*std::max_element(r_u.begin(), r_u.end()))/_tau;
116 double nrm_v = (*std::max_element(r_v.begin(), r_v.end()))/_tau;
117
118 return std::max(nrm_u, nrm_v);
119 };
```

17.2.5 Шаг итерации SIMPLE

Осуществляется в процедуре

```
121 double Cavity2DSimpleWorker::step(){
122   // Predictor step: U-star
123   std::vector<double> u_star = compute_u_star();
124   std::vector<double> v_star = compute_v_star();
125   // Pressure correction
```

```

126 std::vector<double> p_prime = compute_p_prime(u_star, v_star);
127 // Velocity correction
128 std::vector<double> u_prime = compute_u_prime(p_prime);
129 std::vector<double> v_prime = compute_v_prime(p_prime);
130 // Set final values
131 std::vector<double> u_new = vector_sum(u_star, 1.0, u_prime);
132 std::vector<double> v_new = vector_sum(v_star, 1.0, v_prime);
133 std::vector<double> p_new = vector_sum(_p, _alpha_p, p_prime);
134
135 return set_uvp(u_new, v_new, p_new);
136 }
```

и представляет собой буквальное пошаговое следование алгоритму SIMPLE (17.1.2.1). В конце опять вызывается функция `set_uvp` для сборки матриц для следующей итерации и подсчёта невязки на текущей итерации.

17.2.6 Сборка системы уравнений для поправки давления

Сборка системы уравнений (17.25) осуществляется в процедуре

```
166 void Cavity2DSimpleWorker::assemble_p_prime_solver(){
```

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

```
167 LodMatrix mat(p_size());
```

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

```

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

Внутри цикла устанавливаются флаги, характеризующие граничный статус текущего узла

```

170 bool is_left = (i == 0);
171 bool is_right = (i == _cc_grid.nx());
172 bool is_bottom = (j == 0);
173 bool is_top = (j == _cc_grid.ny());
```

Вычисляется значение сквозного индекса по формуле (17.16)

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

и значения коэффициентов в формулах (17.26). Поскольку сетка равномерная, эти значения не меняются для разных узлов

```

176     double coef_x = _du/_hx/_hx;
177     double coef_y = _dv/_hy/_hy;

```

Далее формулы (17.26) применяются для заполнения матриц с учётом аппроксимированного граничного условия (17.37). Так, запись

```

178 // x
179 if (!is_right){
180     size_t ind1 = _grid.cell_centered_grid_index_ip_jp(i+1, j);
181     mat.add_value(ind0, ind0, coef_x);
182     mat.add_value(ind0, ind1, -coef_x);
183 }

```

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 (17.26)) и вычитает из недиагонального (четвёртое из уравнений (17.26)) элемента строки `ind0`. Для правых узлов работает граничное условие (17.26) и выполнять эту процедуру не нужно.

После заполнения в матрицу вводится граничное условие (17.38)

```

201 mat.set_unit_row(0);

```

И матрица передаётся в решатель СЛАУ предварительно сконвертированная в формат `cfd::CsrMatrix`

```

202 _p_prime_solver.set_matrix(mat.to_csr());

```

Правая часть собирается заново на каждой итерации по формуле (17.28). Её реализация представлена в функции

```

351 std::vector<double> Cavity2DSimpleWorker::compute_p_prime(const std::vector<double>&
→ u_star, const std::vector<double>& v_star){

```

Сначала собирается правая часть системы (17.25) по формуле (17.28):

```

353 for (size_t i = 0; i < _grid.nx(); ++i)
354     for (size_t j = 0; j < _grid.ny(); ++j){
355         size_t ind0 = _grid.cell_centered_grid_index_ip_jp(i, j);
356         size_t ind_left = _grid.yface_grid_index_i_jp(i, j);
357         size_t ind_right = _grid.yface_grid_index_i_jp(i+1, j);
358         size_t ind_bot = _grid.xface_grid_index_ip_j(i, j);

```

```

359     size_t ind_top = _grid.xface_grid_index_ip_j(i, j+1);
360     rhs[ind0] = -(u_star[ind_right] - u_star[ind_left])/_tau/_hx - (v_star[ind_top] -
361     v_star[ind_bot])/_tau/_hy;
361 }

```

потом осуществляется установка граничного условия (17.38)

```

362     rhs[0] = 0;

```

и вызывается решатель СЛАУ

```

363     std::vector<double> p_prime;
364     _p_prime_solver.solve(rhs, p_prime);
365     return p_prime;
366 }

```

17.2.7 Сборка системы уравнений для пробной скорости

Сборка системы (17.20) (как правой, так и левой частей) реализована в функции

```

205 void Cavity2DSimpleWorker::assemble_u_slae(){

```

Основной цикл идёт по негрничным узлам “красной” сетки, в котором реализуются формулы (17.21)

```

239     for (size_t j=0; j < _grid.ny(); ++j)
240         for (size_t i=1; i < _grid.nx(); ++i){
241             size_t row_index = _grid.yface_grid_index_i_jp(i, j); // [i, j+1/2]
242
243             double u0_plus = u_ip_jp(i, j); // u[i+1/2, j+1/2]
244             double u0_minus = u_ip_jp(i-1, j); // u[i-1/2, j+1/2]
245             double v0_plus = v_i_j(i, j+1); // v[i, j+1]
246             double v0_minus = v_i_j(i, j); // v[i, j]
247
248             // u_(i, j+1/2)
249             add_to_mat(row_index, {i, j}, 1.0);
250             // + tau * d(u0*u)/ dx
251             add_to_mat(row_index, {i+1, j}, _tau/2.0/_hx*u0_plus);
252             add_to_mat(row_index, {i-1, j}, -_tau/2.0/_hx*u0_minus);
253             // + tau * d(v0*u)/dy
254             add_to_mat(row_index, {i, j+1}, _tau/2.0/_hy*v0_plus);

```

```

255 add_to_mat(row_index, {i, j-1}, -_tau/2.0/_hy*v0_minus);
256 //      - tau / Re * d^2u/dx^2
257 add_to_mat(row_index, {i, j}, 2.0*_tau/_Re/_hx/_hx);
258 add_to_mat(row_index, {i+1, j}, -_tau/_Re/_hx/_hx);
259 add_to_mat(row_index, {i-1, j}, -_tau/_Re/_hx/_hx);
260 //      - tau / Re * d^2u/dy^2
261 add_to_mat(row_index, {i, j}, 2.0*_tau/_Re/_hy/_hy);
262 add_to_mat(row_index, {i, j+1}, -_tau/_Re/_hy/_hy);
263 add_to_mat(row_index, {i, j-1}, -_tau/_Re/_hy/_hy);
264 // = u0_(i, j+1/2)
265 _rhs_u[row_index] += _u[row_index];
266 //      - tau * dp/dx
267 _rhs_u[row_index] -= _tau/_hx*(p_ip_jp(i, j) - p_ip_jp(i-1, j));
268 }

```

Как было отмечено в пункте 17.1.3.5, граничные условия первого рода в этом уравнении учитываются двумя разными способами: узлы расположенные непосредственно на границе (нижней и верхней) учитываются по схеме (17.34), которая реализована в цикле

```

228 for (size_t j=0; j< _grid.ny(); ++j){
229     size_t index_left = _grid.yface_grid_index_i_jp(0, j);
230     add_to_mat(index_left, {0, j}, 1.0);
231     _rhs_u[index_left] = 0.0;
232
233     size_t index_right = _grid.yface_grid_index_i_jp(_grid.nx(), j);
234     add_to_mat(index_right, {_grid.nx(), j}, 1.0);
235     _rhs_u[index_right] = 0.0;
236 }

```

А фиктивные узлы, возникающие при обработке узлов расположенных в полу шаге от границ (левой и правой), обрабатываются по схеме (17.36). Эта схема реализована в виде препроцессинга алгоритма добавления элемента в матрицу в лямбда-функции

```

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

```

```

217 // ghost index => bottom boundary condition: u = u_bot
218 size_t ind1 = _grid.yface_grid_index_i_jp(ij_col[0], ij_col[1]+1);
219 mat.add_value(row_index, ind1, -value);
220 _rhs_u[row_index] -= 2.0*value * bottom_velocity()[0];
221 } else {
222 size_t ind1 = _grid.yface_grid_index_i_jp(ij_col[0], ij_col[1]);
223 mat.add_value(row_index, ind1, value);
224 }
225 };

```

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 (17.36).

17.3 Задание для самостоятельной работы

- Подобрать оптимальные параметры алгоритма SIMPLE τ, α_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.
- Нарисовать поле невязок r_u, r_v в динамике по каждой итерации. Отметить в каком из уравнений и в каких местах области расчёта наблюдаются наибольшие проблемы со сходимостью. Обратить внимание, что невязка r_u задана на “красной” сетке. При этом сохранение на этой сетке делается через объект `_writer_u`. Невязка r_v задается на “синей” сетке с объектом сохранения `_writer_v`.
- Решить аналогичную задачу, в которой скорость не только на верхней, но и на нижней стенке равна $U = 1$. Для этого завести новый тест `[cavity2-simple-sym]`.

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 ∇ – 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, ϕ, z) :

$$\nabla = \left(\frac{\partial}{\partial r}, \frac{1}{r} \frac{\partial}{\partial \phi}, \frac{\partial}{\partial z} \right).$$

В цилиндрической (r, θ, ϕ) :

$$\nabla = \left(\frac{\partial}{\partial r}, \frac{1}{r} \frac{\partial}{\partial \theta}, \frac{1}{r \sin \theta} \frac{\partial}{\partial \phi} \right).$$

Удобство записи дифференциальных выражений с использованием ∇ 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 ∇ 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 ∇ 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 ∇ 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 Γ , 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. 15. Разобьем область на $N = 3$ равномерных подобластей

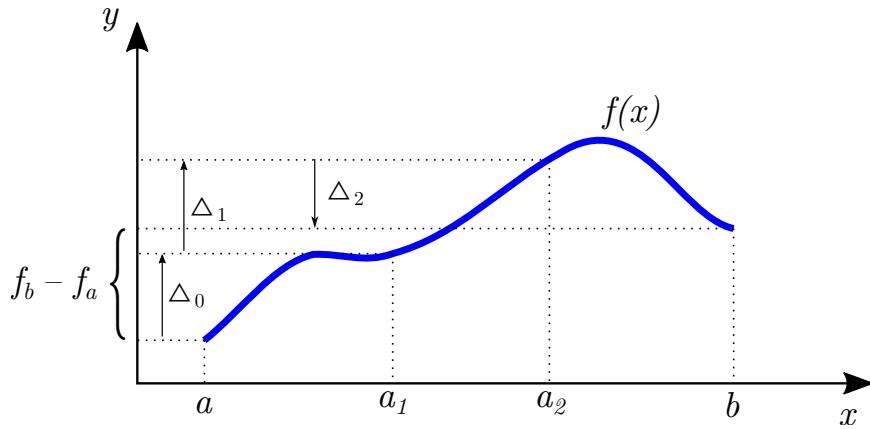


Рис. 15: Формула Гаусса–Остроградского в одномерном случае

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 $f_i = f(\xi_i)$, ϕ_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 ξ 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 ξ_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. 16)

$$\begin{aligned} \phi_0(\xi) &= \frac{1-\xi}{2}, \\ \phi_1(\xi) &= \frac{1+\xi}{2}. \end{aligned} \tag{A.20}$$

Окончательно интерполяционная функция из определения (A.15) примет вид

$$f(\xi) \approx \frac{1-\xi}{2}f(-1) + \frac{1+\xi}{2}f(1).$$

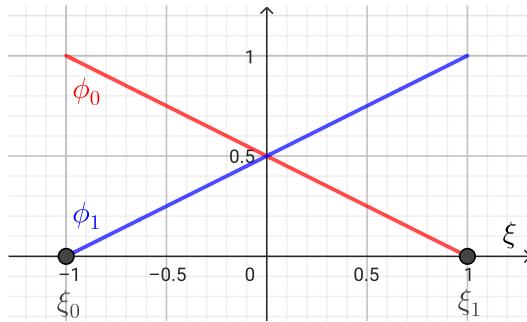


Рис. 16: Линейный базис в параметрическом отрезке

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. 17)

$$\begin{aligned} \phi_0(\xi) &= \frac{\xi^2 - \xi}{2}, \\ \phi_1(\xi) &= \frac{\xi^2 + \xi}{2}, \\ \phi_2(\xi) &= 1 - \xi^2. \end{aligned} \quad (\text{A.21})$$

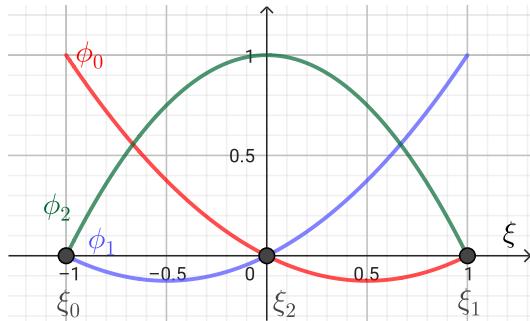


Рис. 17: Квадратичный базис в параметрическом отрезке

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. 18)

$$\begin{aligned}\phi_0(\xi) &= \frac{1}{16} (-1 + \xi + 9\xi^2 - 9\xi^3), \\ \phi_1(\xi) &= \frac{1}{16} (-1 - \xi + 9\xi^2 + 9\xi^3), \\ \phi_2(\xi) &= \frac{1}{16} (9 - 27\xi - 9\xi^2 + 27\xi^3), \\ \phi_3(\xi) &= \frac{1}{16} (9 + 27\xi - 9\xi^2 - 27\xi^3),\end{aligned}\tag{A.22}$$

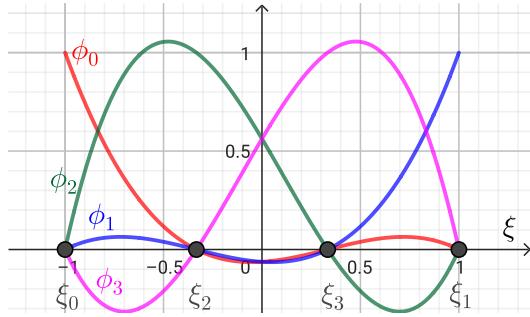


Рис. 18: Кубический базис в параметрическом отрезке

На рис. 19 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.

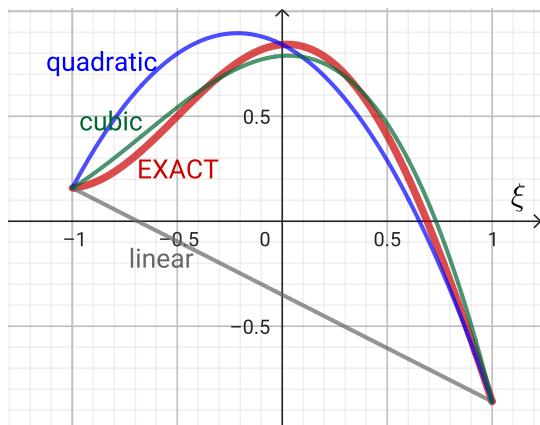


Рис. 19: Результат интерполяции

A.3.1.3 Интерполяция в параметрическом треугольнике

Теперь рассмотрим двумерное обобщение формулы

Линейный базис

$$\phi_i(\xi, \eta) = A_i^{(00)} + A_i^{(10)}\xi + A_i^{(01)}\eta.$$

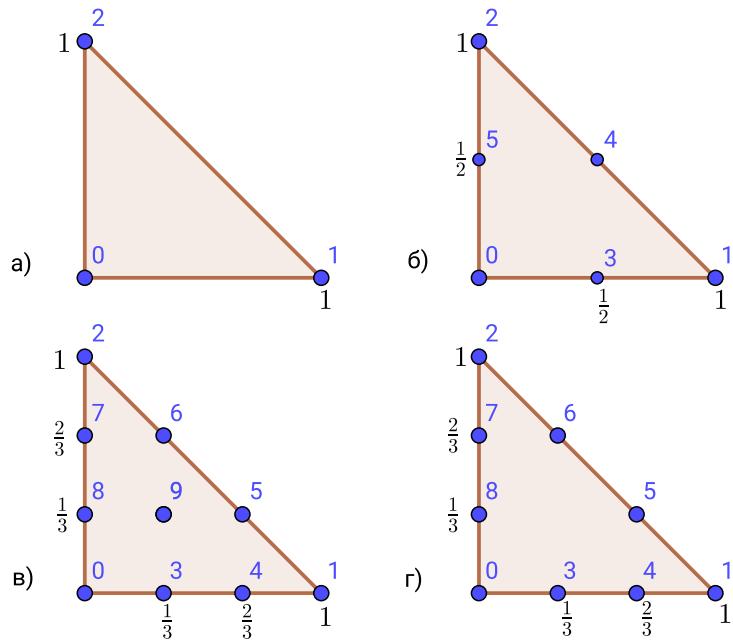


Рис. 20: Расположение узловых точек в параметрическом треугольнике. а) линейный базис, б) квадратичный базис, в) кубический базис, г) неполный кубический базис

$$C = \left(\begin{array}{c|ccc} & A^{(00)} & A^{(10)} & A^{(01)} \\ \hline \phi(0,0) & 1 & 0 & 0 \\ \phi(1,0) & 1 & 1 & 0 \\ \phi(0,1) & 1 & 0 & 1 \end{array} \right) \Rightarrow A = C^{-1} = \left(\begin{array}{c|ccc} & \phi_0 & \phi_1 & \phi_2 \\ \hline 1 & 1 & 0 & 0 \\ \xi & -1 & 1 & 0 \\ \eta & -1 & 0 & 1 \end{array} \right)$$

$$\begin{aligned} \phi_0(\xi, \eta) &= 1 - \xi - \eta, \\ \phi_1(\xi, \eta) &= \xi, \\ \phi_2(\xi, \eta) &= \eta, \end{aligned} \tag{A.23}$$

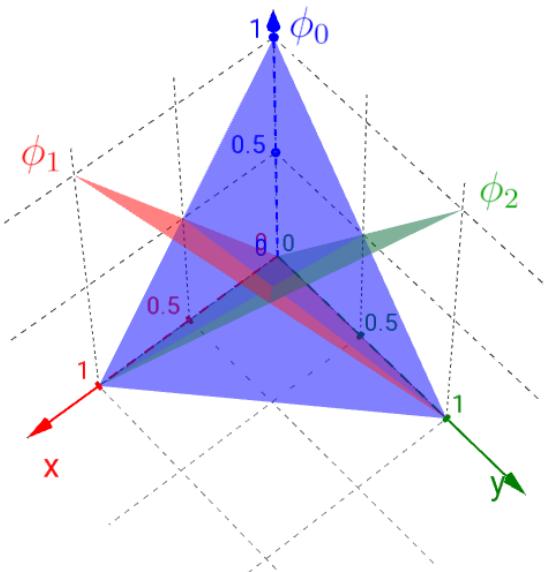


Рис. 21: Линейный базис в параметрическом треугольнике

Квадратичный базис

$$\phi_i(\xi, \eta) = A_i^{(00)} + A_i^{(10)}\xi + A_i^{(01)}\eta + A_i^{(11)}\xi\eta + A_i^{(20)}\xi^2 + A_i^{(02)}\eta^2.$$

$$C = \left(\begin{array}{c|cccccc} & A^{(00)} & A^{(10)} & A^{(01)} & A^{(11)} & A^{(20)} & A^{(02)} \\ \hline \phi(0,0) & 1 & 0 & 0 & 0 & 0 & 0 \\ \phi(1,0) & 1 & 1 & 0 & 0 & 1 & 0 \\ \phi(0,1) & 1 & 0 & 1 & 0 & 0 & 1 \\ \phi(\frac{1}{2},0) & 1 & \frac{1}{2} & 0 & 0 & \frac{1}{4} & 0 \\ \phi(\frac{1}{2},\frac{1}{2}) & 1 & \frac{1}{2} & \frac{1}{2} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \phi(0,\frac{1}{2}) & 1 & 0 & \frac{1}{2} & 0 & 0 & \frac{1}{4} \end{array} \right) \Rightarrow A = \left(\begin{array}{c|cccccc} & \phi_0 & \phi_1 & \phi_2 & \phi_3 & \phi_4 & \phi_5 \\ \hline 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ \xi & -3 & -1 & 0 & 4 & 0 & 0 \\ \eta & -3 & 0 & -1 & 0 & 0 & 4 \\ \xi\eta & 4 & 0 & 0 & -4 & 4 & -4 \\ \xi^2 & 2 & 2 & 0 & -4 & 0 & 0 \\ \eta^2 & 2 & 0 & 2 & 0 & 0 & -4 \end{array} \right)$$

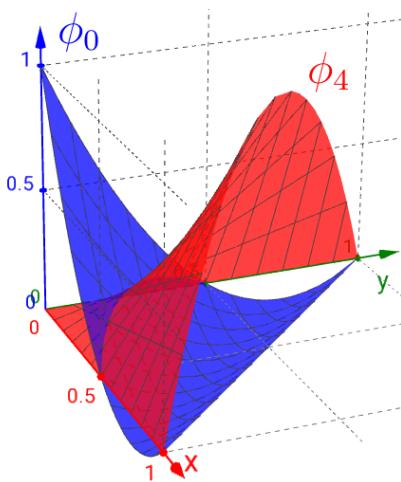


Рис. 22: Квадратичные функции ϕ_0, ϕ_4 в параметрическом треугольнике

Кубический базис TODO

Неполный кубический базис TODO

A.3.1.4 Интерполяция в параметрическом квадрате

Билинейный базис

$$\phi_i = A_i^{00} + A_i^{10}\xi + A_i^{01}\eta + A_i^{11}\xi\eta.$$

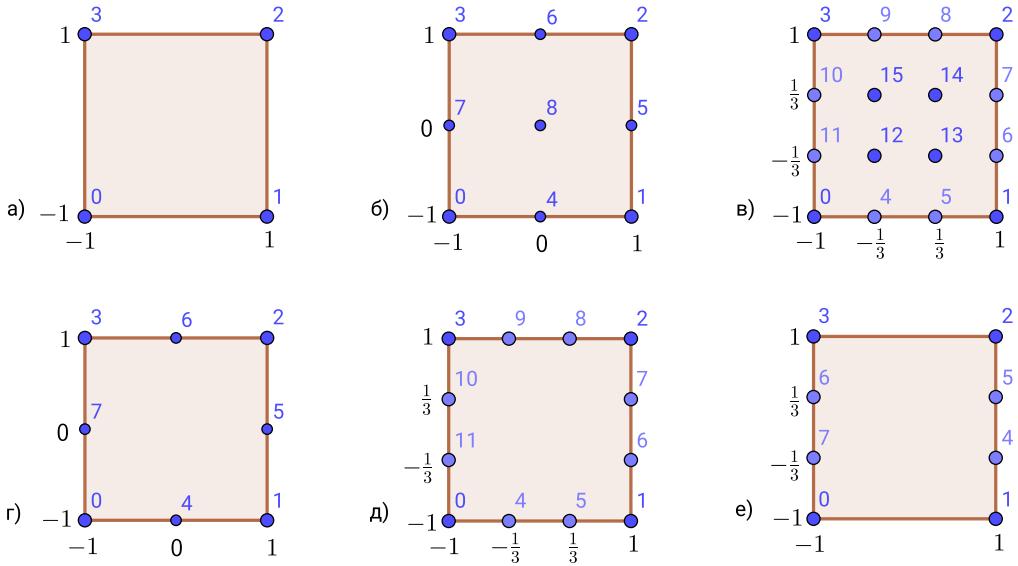


Рис. 23: Расположение узловых точек в параметрическом квадрате

$$C = \left(\begin{array}{c|cccc} & A^{(00)} & A^{(10)} & A^{(01)} & A^{(11)} \\ \hline \phi(-1, -1) & 1 & -1 & -1 & 1 \\ \phi(1, -1) & 1 & 1 & -1 & -1 \\ \phi(1, 1) & 1 & 1 & 1 & 1 \\ \phi(-1, 1) & 1 & -1 & 1 & -1 \end{array} \right) \Rightarrow A = C^{-1} = \frac{1}{4} \left(\begin{array}{c|ccccc} & \phi_0 & \phi_1 & \phi_2 & \phi_3 \\ \hline 1 & 1 & 1 & 1 & 1 \\ \xi & -1 & 1 & 1 & -1 \\ \eta & -1 & -1 & 1 & 1 \\ \xi\eta & 1 & -1 & 1 & -1 \end{array} \right)$$

$$\begin{aligned} \phi_0(\xi, \eta) &= \frac{1 - \xi - \eta + \xi\eta}{4} \\ \phi_1(\xi, \eta) &= \frac{1 + \xi - \eta - \xi\eta}{4} \\ \phi_2(\xi, \eta) &= \frac{1 + \xi + \eta + \xi\eta}{4} \\ \phi_3(\xi, \eta) &= \frac{1 - \xi + \eta - \xi\eta}{4} \end{aligned} \tag{A.24}$$

Определение двумерных базисов через комбинацию одномерных Обратим внимание, что в искомые билинейные базисные функции линейны в каждом из направлений ξ, η , 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:

$$\boldsymbol{\xi}_0 = (\xi_0^{1D}, \xi_0^{1D}), \quad \boldsymbol{\xi}_1 = (\xi_1^{1D}, \xi_0^{1D}), \quad \boldsymbol{\xi}_2 = (\xi_1^{1D}, \xi_1^{1D}), \quad \boldsymbol{\xi}_3 = (\xi_0^{1D}, \xi_1^{1D}).$$

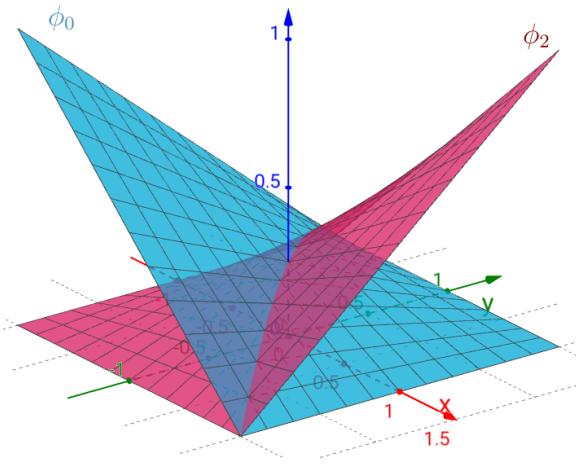


Рис. 24: Билинейные функции ϕ_0, ϕ_2 в параметрическом квадрате

Значит и соответствующие базисные функции можно выразить через линейный одномерный базис ϕ^{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. 23б. В качестве основе возьмём квадратичный одномерный базис ϕ_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 Линейная интерполяция

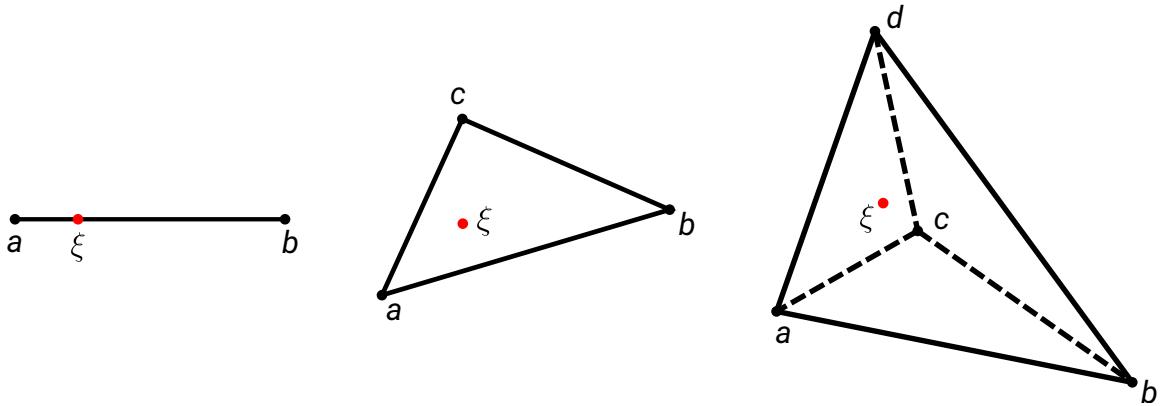


Рис. 25: Порядок нумерации точек одномерного, двумерного и трёхмерного симплекса при линейной интерполяции

Пусть функция u задана в узлах симплекса, имеющего нумерацию согласно рис. 25. Необходимо найти значение этой функции в точке ξ (эта точка вообще говоря не обязана лежать внутри симплекса).

Интерполяция в одномерном, двумерном и трёхмерном виде запишется как

$$u(\xi) = \frac{|\Delta_{\xi a}|u(b) + |\Delta_{b\xi}|u(a)}{|\Delta_{ba}|} \quad (\text{A.26})$$

$$u(\xi) = \frac{|\Delta_{ab\xi}|u(c) + |\Delta_{bc\xi}|u(a) + |\Delta_{ca\xi}|u(b)}{|\Delta_{abc}|} \quad (\text{A.27})$$

$$u(\xi) = \frac{|\Delta_{abc\xi}|u(d) + |\Delta_{cbd\xi}|u(a) + |\Delta_{cda\xi}|u(b) + |\Delta_{adb\xi}|u(c)}{|\Delta_{abcd}|}, \quad (\text{A.28})$$

где $|\Delta|$ – знаковый объём симплекса, вычисляемый как

$$|\Delta_{ab}| = b - a,$$

$$|\Delta_{abc}| = \left(\frac{(\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})}{2} \right)_z,$$

$$|\Delta_{abcd}| = \frac{(\mathbf{b} - \mathbf{a}) \cdot ((\mathbf{c} - \mathbf{a}) \times (\mathbf{d} - \mathbf{a}))}{6}.$$

A.4.2 Преобразование координат

Рассмотрим преобразование из двумерной параметрической системы координат ξ в физическую систему \mathbf{x} . Такое преобразование полностью определяется покоординатными функциями $\mathbf{x}(\xi)$. Далее получим соотношения, связывающие операции дифференцирования и интегрирования в физической и параметрической областях.

A.4.2.1 Матрица Якоби

Будем рассматривать двумерное преобразование $(\xi, \eta) \rightarrow (x, y)$. Линеаризуем это преобразование (разложим в ряд Фурье до линейного слагаемого)

$$x(\xi_0 + d\xi, \eta_0 + d\eta) \approx x_0 + \frac{\partial x}{\partial \xi} \Big|_{\xi_0, \eta_0} d\xi + \frac{\partial x}{\partial \eta} \Big|_{\xi_0, \eta_0} d\eta,$$

$$y(\xi_0 + d\xi, \eta_0 + d\eta) \approx y_0 + \frac{\partial y}{\partial \xi} \Big|_{\xi_0, \eta_0} d\xi + \frac{\partial y}{\partial \eta} \Big|_{\xi_0, \eta_0} d\eta,$$

где $x_0 = x(\xi_0, \eta_0)$, $y_0 = y(\xi_0, \eta_0)$. Переписывая это выражение в векторном виде, получим

$$\mathbf{x}(\xi_0 + d\xi) - \mathbf{x}_0 = J(\xi_0) d\xi. \quad (\text{A.29})$$

Матрица J (зависящая от точки приложения в параметрической плоскости) называется матрицей Якоби:

$$J = \begin{pmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{pmatrix} = \begin{pmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{pmatrix} \quad (\text{A.30})$$

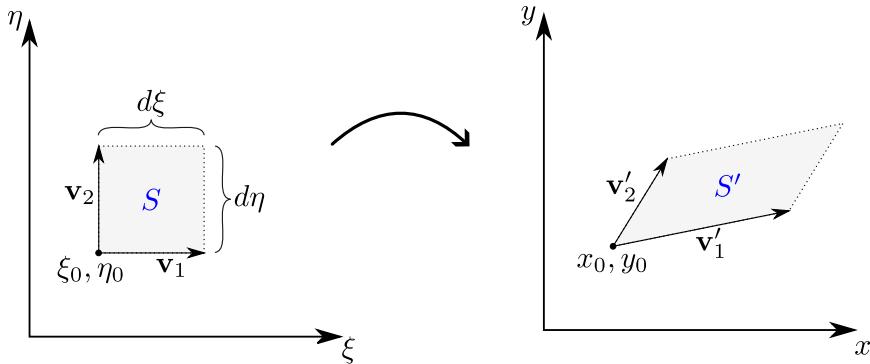


Рис. 26: Преобразование элементарного объёма

Якобиан Определитель матрицы Якоби (якобиан), взятый в конкретной точке параметрической плоскости ξ_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 ξ_0 (см. рис. 26). В результате преобразования по формуле (A.29) получим следующие преобразования концевых точек и векторов:

$$(\xi_0, \eta_0) \rightarrow (x_0, y_0),$$

$$(\xi_0 + d\xi, \eta_0) \rightarrow (x_0 + J_{11}d\xi, y_0 + J_{21}d\xi) \Rightarrow \mathbf{v}_1 \rightarrow \mathbf{v}'_1 = (J_{11}d\xi, J_{21}d\xi),$$

$$(\xi_0, \eta_0 + d\eta) \rightarrow (x_0 + J_{12}d\eta, y_0 + J_{22}d\eta) \Rightarrow \mathbf{v}_2 \rightarrow \mathbf{v}'_2 = (J_{12}d\eta, J_{22}d\eta).$$

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

$$dx dy = |J| d\xi d\eta \quad (\text{A.31})$$

Многомерным обобщением этой формулы будет

$$d\mathbf{x} = |J| d\boldsymbol{\xi} \quad (\text{A.32})$$

A.4.2.2 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.30), 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

$$\begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix} = (J^T)^{-1} \begin{pmatrix} \frac{\partial f}{\partial \xi} \\ \frac{\partial f}{\partial \eta} \end{pmatrix} = \frac{1}{|J|} \begin{pmatrix} J_{22} & -J_{21} \\ -J_{12} & J_{11} \end{pmatrix} \begin{pmatrix} \frac{\partial f}{\partial \xi} \\ \frac{\partial f}{\partial \eta} \end{pmatrix}$$

В многомерном виде запишем

$$\nabla_{\mathbf{x}} f = (J^T)^{-1} \nabla_{\boldsymbol{\xi}} f. \quad (\text{A.33})$$

A.4.2.3 Интегрирование в параметрической плоскости

Пусть в физической области \mathbf{x} задана область D_x . Интеграл функции $f(\mathbf{x})$ по этой области можно расписать, используя замену (A.32)

$$\int_{D_x} f(\mathbf{x}) d\mathbf{x} = \int_{D_\xi} f(\boldsymbol{\xi}) |J(\boldsymbol{\xi})| d\boldsymbol{\xi}, \quad (\text{A.34})$$

где $f(\boldsymbol{\xi}) = f(\mathbf{x}(\boldsymbol{\xi}))$, а D_ξ – образ области D_x в параметрической плоскости.

A.4.2.4 Двумерное линейное преобразование. Параметрический треугольник

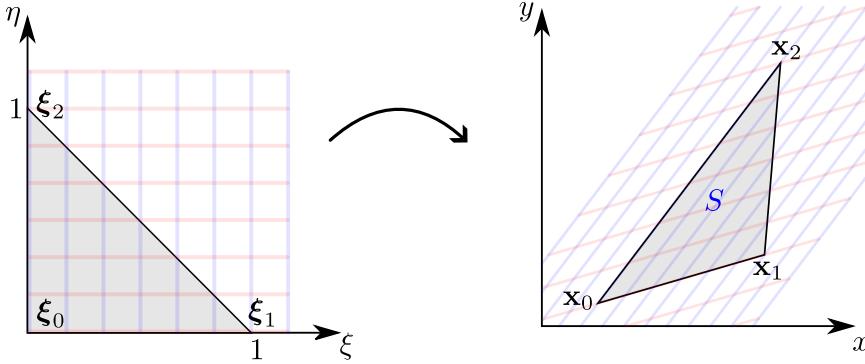


Рис. 27: Преобразование из параметрического треугольника

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. 27):

$$\xi_0, \eta_0 = (0, 0), \quad \xi_1, \eta_1 = (1, 0), \quad \xi_2, \eta_2 = (0, 1).$$

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

$$\begin{aligned} x(\xi, \eta) &= (x_1 - x_0)\xi + (x_2 - x_0)\eta + x_0, \\ y(\xi, \eta) &= (y_1 - y_0)\xi + (y_2 - y_0)\eta + y_0. \end{aligned} \quad (\text{A.35})$$

Матрица Якоби этого преобразования (A.30) не будет зависеть от параметрических координат ξ, η :

$$J = \begin{pmatrix} x_1 - x_0 & x_2 - x_0 \\ y_1 - y_0 & y_2 - y_0 \end{pmatrix}. \quad (\text{A.36})$$

Якобиан преобразования будет равен удвоенной площади треугольника S , составленного из определяющих точек в физической плоскости:

$$|J| = (x_1 - x_0)(y_2 - y_0) - (y_1 - y_0)(x_2 - x_0) = (\mathbf{x}_1 - \mathbf{x}_0) \times (\mathbf{x}_2 - \mathbf{x}_0) = 2|S|. \quad (\text{A.37})$$

Распишем интеграл по треугольнику S по формуле (A.34). Вследствии линейности преобразования якобиан постоянен и, поэтому, его можно вынести его из-под интеграла:

$$\int_S f(x, y) dx dy = |J| \int_0^1 \int_0^{1-\xi} f(\xi, \eta) d\eta d\xi. \quad (\text{A.38})$$

A.4.2.5 Двумерное билинейное преобразование. Параметрический квадрат

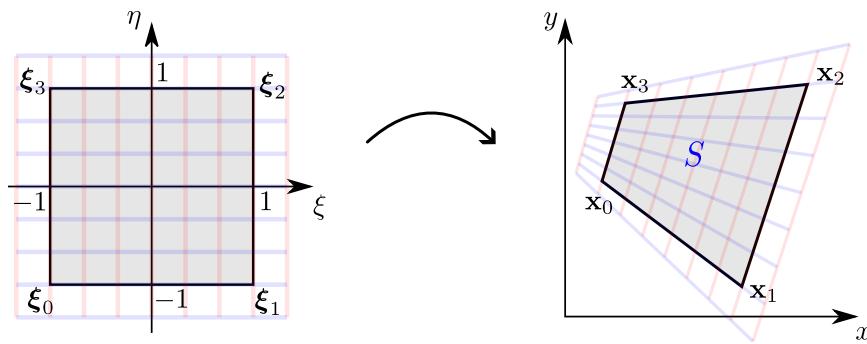


Рис. 28: Преобразование из параметрического квадрата

A.4.2.6 Трёхмерное линейное преобразование. Параметрический тетраэдр

TODO

A.4.3 Свойства многоугольника

A.4.3.1 Площадь многоугольника

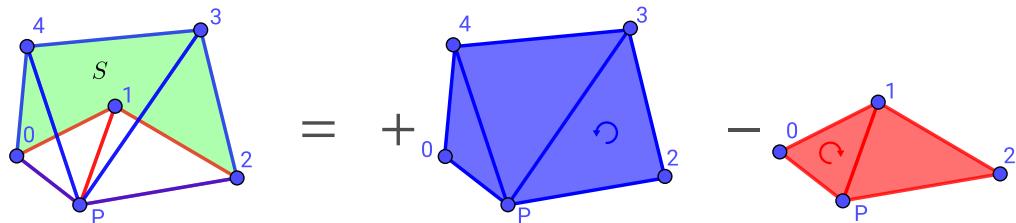


Рис. 29: Площадь произвольного многоугольника

Рассмотрим произвольный несамопересекающийся N -угольник S , заданный координатами своих узлов \mathbf{x}_i , $i = \overline{0, N-1}$, пронумерованных последовательно против часовой стрелки (рис. 29). 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 Δ_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 29 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

$$\Delta_i = (\mathbf{x}_0, \mathbf{x}_i, \mathbf{x}_{i+1}), \quad i = \overline{1, N-2}. \quad (\text{A.39})$$

Знаковая площадь ориентированного треугольника будет равна

$$|\Delta_i| = \frac{(\mathbf{x}_i - \mathbf{x}_0) \times (\mathbf{x}_{i+1} - \mathbf{x}_0)}{2}. \quad (\text{A.40})$$

Тогда окончательно формула определения площади примет вид

$$|S| = \sum_{i=1}^{N-2} |\Delta_i|. \quad (\text{A.41})$$

Плоский полигон в пространстве Если плоский полигон S расположен в трёхмерном пространстве, то правая часть формулы (A.40) 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:

$$|\Delta_i| = \frac{((\mathbf{x}_i - \mathbf{x}_0) \times (\mathbf{x}_{i+1} - \mathbf{x}_0)) \cdot \mathbf{n}}{2}. \quad (\text{A.42})$$

Формула для определения площади полигона (A.41) 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.4.3.2 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.39). Тогда интеграл по многоугольнику можно расписать как сумму интегралов по ориентированным треугольникам:

$$\int_S f(x, y) dx dy = \sum_{i=1}^{N-2} \int_{\Delta_i} f(x, y) dx dy. \quad (\text{A.43})$$

Далее для вычисления интегралов в правой части воспользуемся преобразованием к параметрическому треугольнику (п. A.4.2.4). Следуя формуле интегрирования (A.38), 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.37) есть удвоенная площадь ориентированного треугольника Δ_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.35) 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

$$\int_S f(x, y) dx dy = 2 \sum_{i=1}^{N-2} |\Delta_i| \int_0^1 \int_0^{1-\xi} f_i(\xi, \eta) d\eta d\xi. \quad (\text{A.44})$$

Отметим, что эта формула работает и в том случае, когда полигон расположен в трёхмерном пространстве (знаковую площадь при этом следует вычислять по (A.42)).

A.4.3.3 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.43) с учётом линейного преобразования (A.35):

$$\begin{aligned}
 \int_S x \, dx dy &= \sum_{i=1}^{N-2} \int_{\Delta_i} x \, dx dy \\
 &= \sum_{i=1}^{N-2} |J_i| \int_0^1 \int_0^{1-\xi} ((x_i - x_0)\xi + (x_{i+1} - x_0)\eta + x_0) d\eta d\xi \\
 &= \sum_{i=1}^{N-2} \frac{|J_i|}{2} \frac{x_0 + x_i + x_{i+1}}{3} \\
 &= \sum_{i=1}^{N-2} |\Delta_i| \frac{x_0 + x_i + x_{i+1}}{3}.
 \end{aligned}$$

Итого, с учётом (A.41), 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.40). В случае трёхмерного пространства должна использоваться формула (A.42).

A.4.4 Свойства многогранника

A.4.4.1 Объём многогранника

TODO

A.4.4.2 Интеграл по многограннику

TODO

A.4.4.3 Центр масс многогранника

TODO

A.4.5 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/v2.43.0.windows.1/Git-2.43.0-64-bit.exe>
- https://github.com/Kitware/CMake/releases/download/v3.28.3/cmake-3.28.3-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/CFDCourse25
```

В директории (`D:/git_repos` в примере) появится папка `CFDCourse25`, которая является корневой папкой проекта

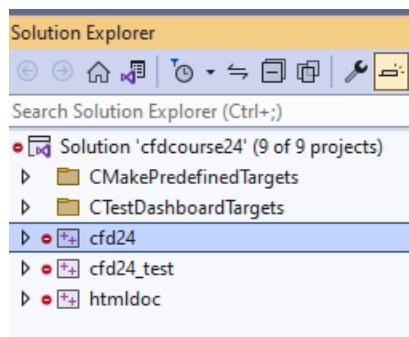
B.1.1.2 VisualStudio

1. Создайте папку `build` в корне проекта `CFDCourse25`
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"
```

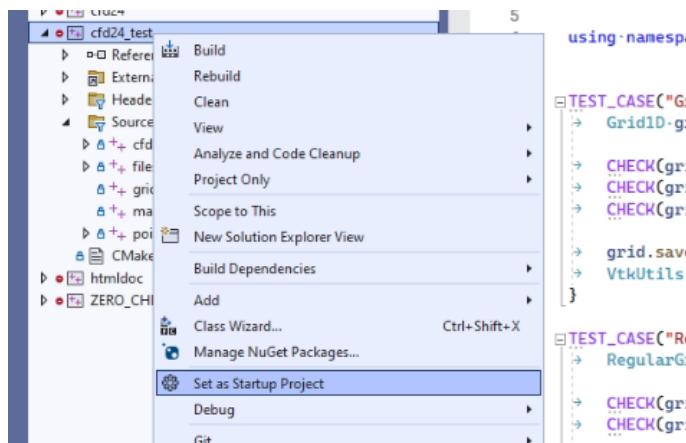
4. Запустите скрипт **winbuild64.bat** из папки **build**. Нужен доступ к интернету. В процессе будет скачано около 200Мб пакетов, поэтому первый запуск может занять время
5. После сборки в папке **build** появится проект **VisualStudio cfdcourse25.sln**. Его нужно открыть в **VisualStudio**. Дерево решения должно иметь следующий вид;



Проекты:

- **cfdbuild24** – расчётная библиотека
- **cfdbuild24_test** – модульные тесты для расчётных функций

6. Проект **cfdbuild24_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 **cf25_test**.

8. Запустить тесты (проект

cf25_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 **CFDCourse25/build/bin/Debug**. В случае работы через отладчик выходная директория, куда будут скидываться все файлы (в частности, vtk), должна быть **CFDCourse25/build/src/test/**.

B.1.1.3 VSCode

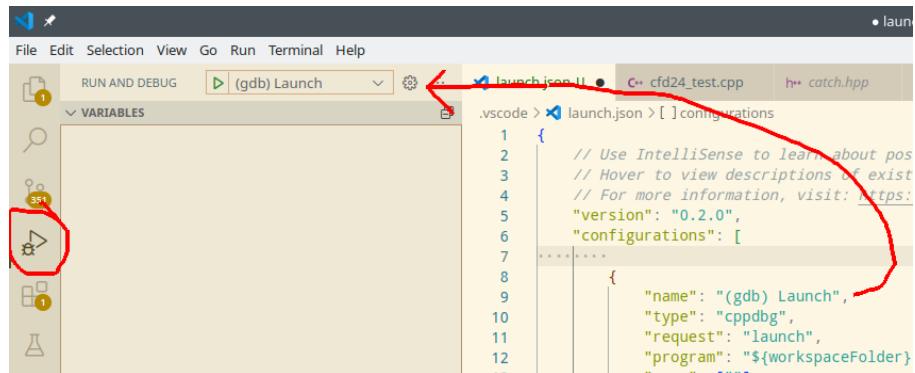
1. Открыть корневую папку проекта через **File->Open Folder**
2. Установить предлагаемые расширения cmake, c++
3. Для настройки отладки создайте конфигурацию launch.json следующего вида

```

5   "version": "0.2.0",
6   "configurations": [
7     {
8       "name": "(gdb) Launch",
9       "type": "cppdbg",
10      "request": "launch",
11      "program": "${workspaceFolder}/build/bin/cfd25_test",
12      "args": [""], ←
13      "stopAtEntry": false,
14      "cwd": "${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

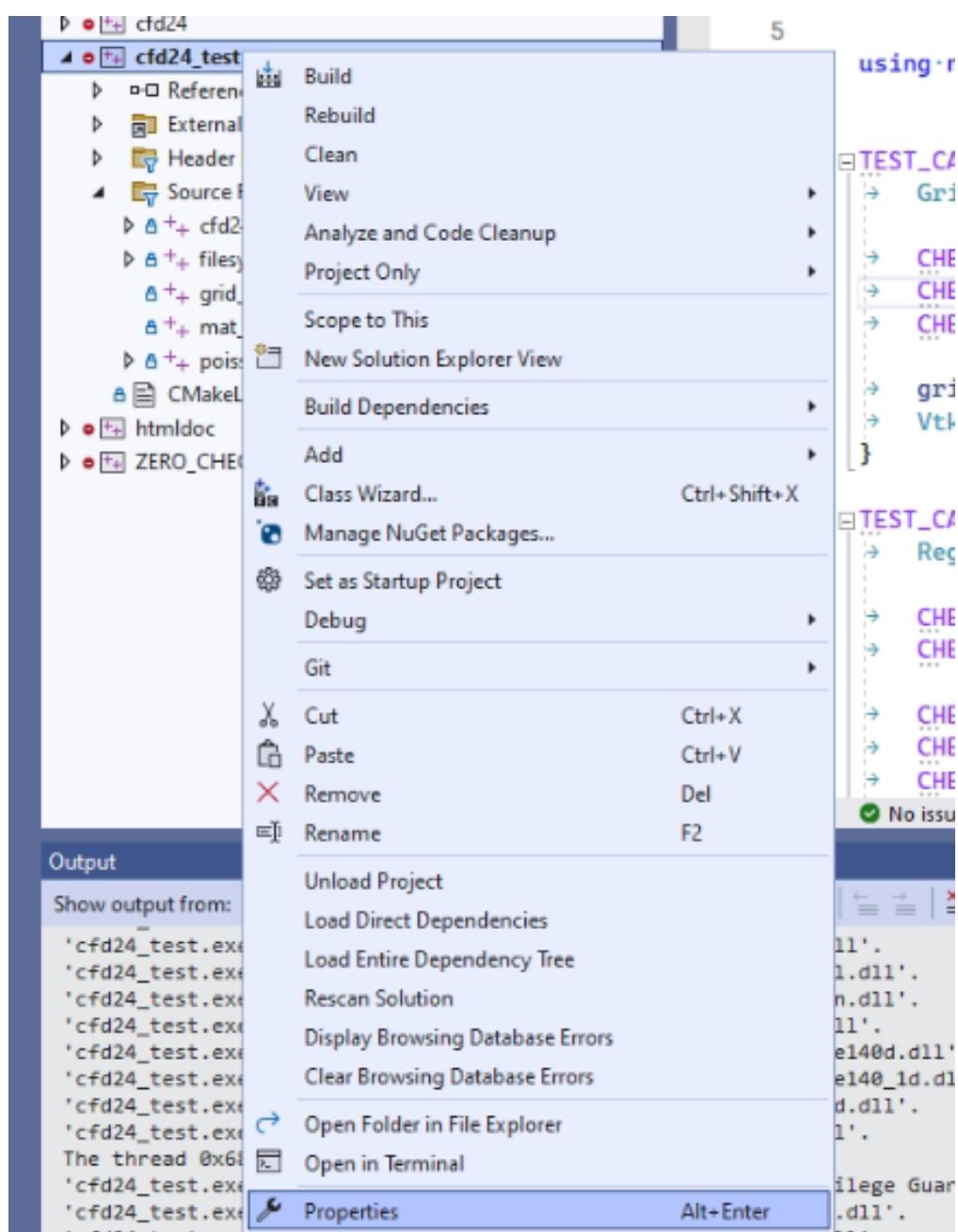
```
"name" : "(Windows) Launch",
"program": "${workspaceFolder}/build/bin/Debug/cfd25_test.exe"
```

B.1.2 Запуск конкретного теста

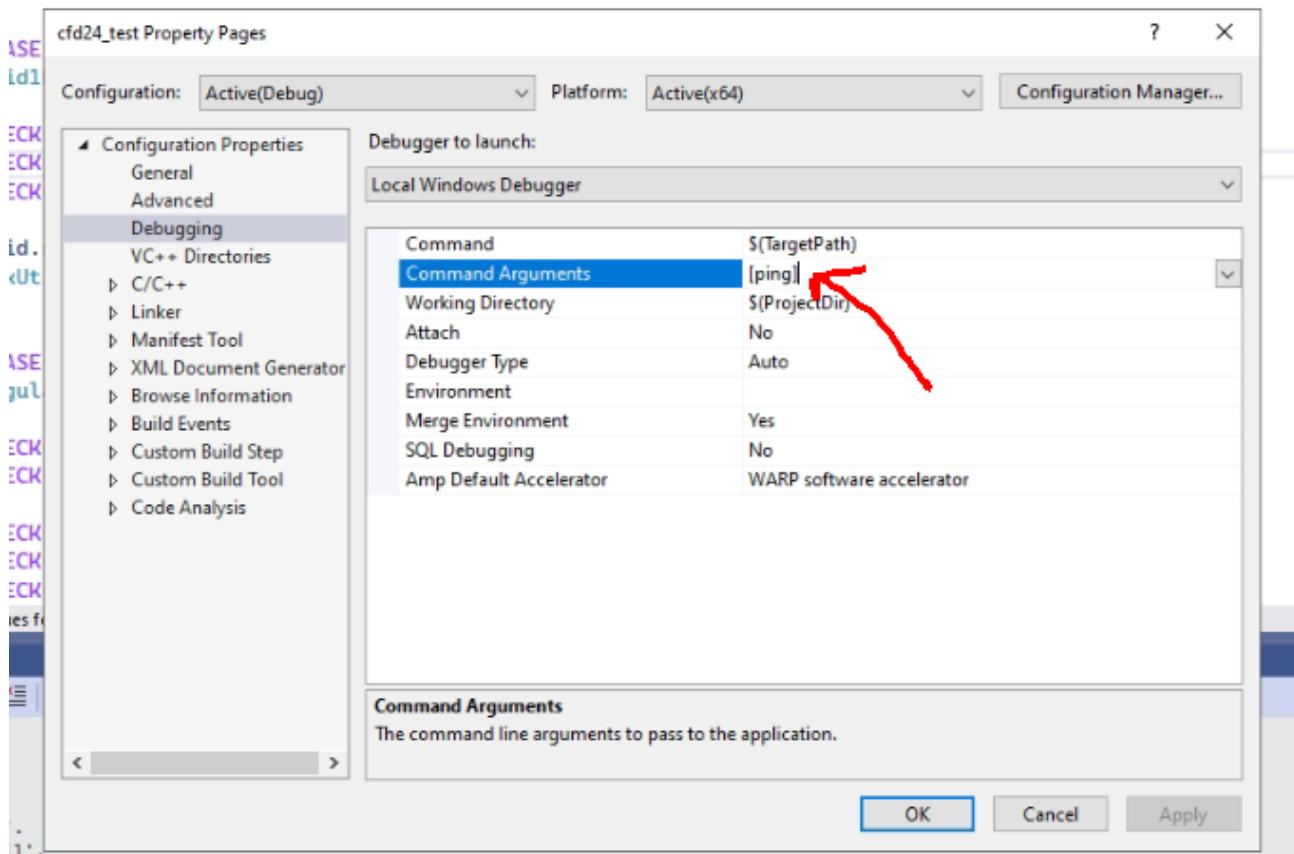
По умолчанию программа `cf25_test` прогоняет все объявленные в проекте тесты. Иногда может возникнуть необходимость запустить только конкретный тест в целях отладки или проверки. Для этого нужно передать программе аргумент с тегом для этого теста.

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 `cf25_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

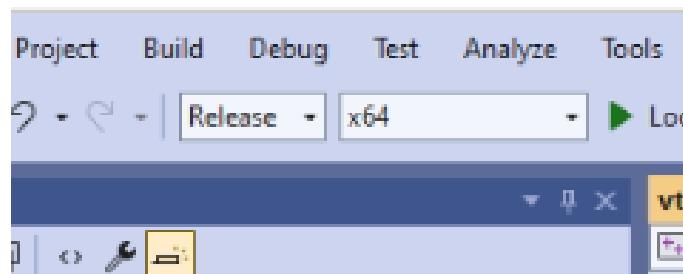
Релизная сборка программ даёт многократное увеличение производительности, но при этом отладка приложений в таком режиме невозможна.

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/cfdcourse25.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 `CFDCourse25/build_release/bin/Release`. В случае работы через отладчик выходная директория – `CFDCourse25/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 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.

Таким образом, **после лекции**, после того, как изменение

`after-lect` придет на сервер, 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

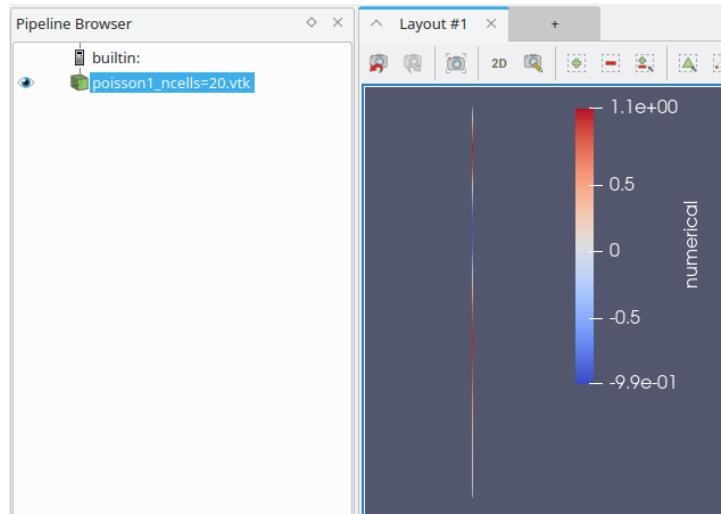
```
> git checkout work-lect{index}
```

Если ничего не было сделано (или все изменения не представляют ценности), можно повторить алгоритм “после лекции”.

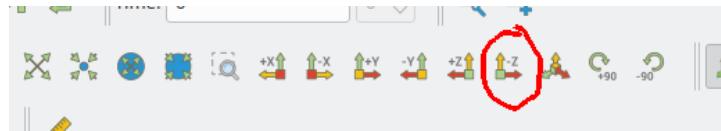
B.3 Paraview

B.3.1 Данные на одномерных сетках

Заданные на сетке данные паравью показывает цветом. Поэтому при загрузке одномерных сеток можно видеть картинку типа

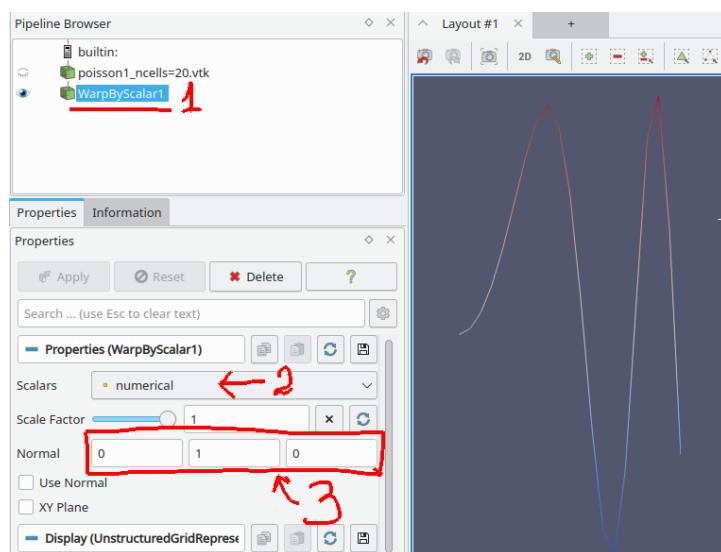


Развернуть изображение в плоскость xy



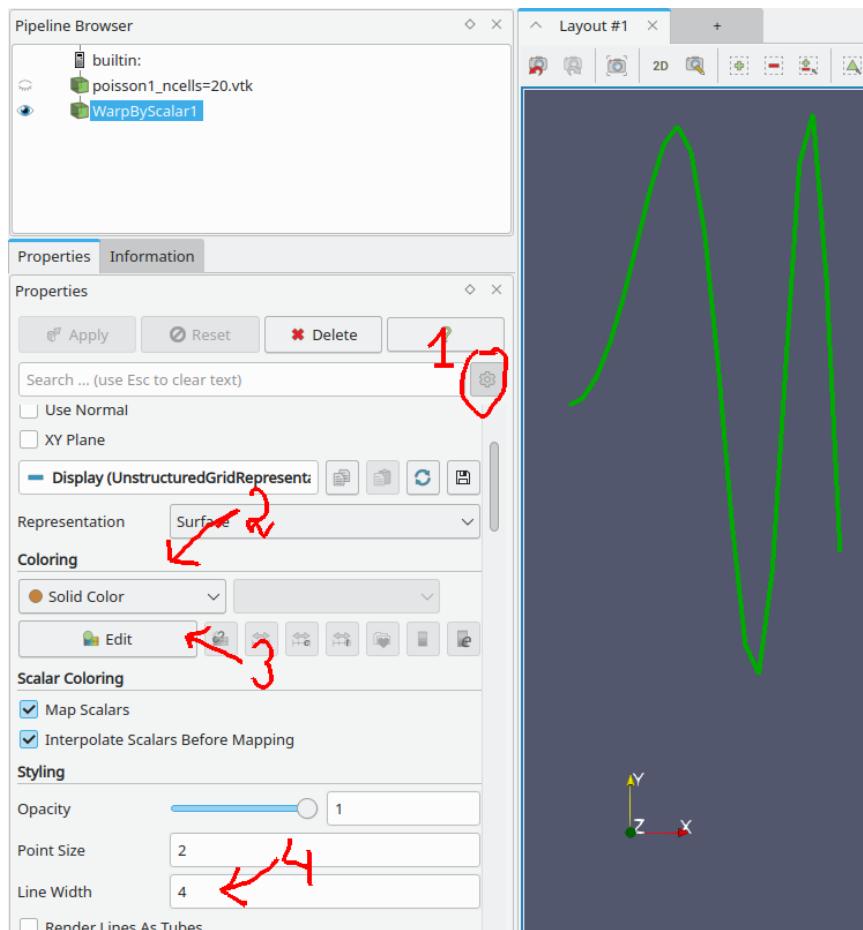
Отобразить данные в виде у-координаты Для того, что бы данные отображались в качестве значения по оси ординат, к загруженному файлу необходимо

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

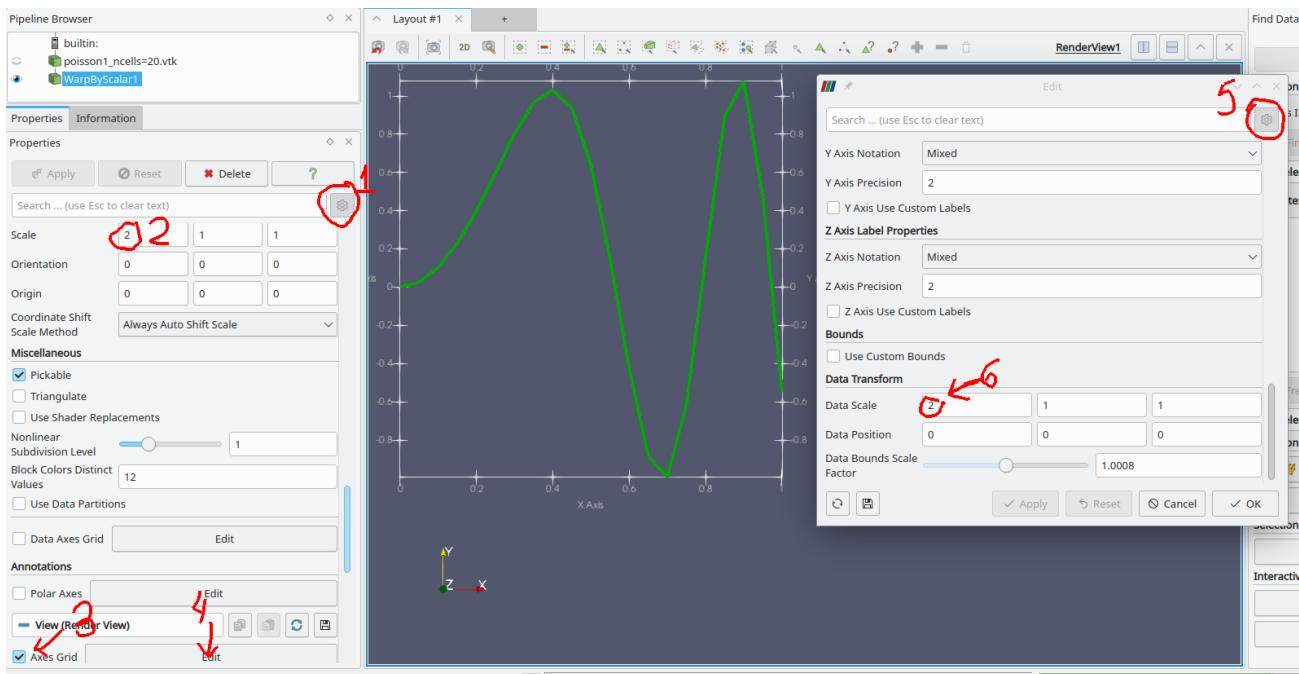
1. Включить подробные опции фильтра
2. Сменить стиль на **Solid Color**
3. В меню **Edit** выбрать желаемый цвет
4. В строке **Line Width** указать толщину линии



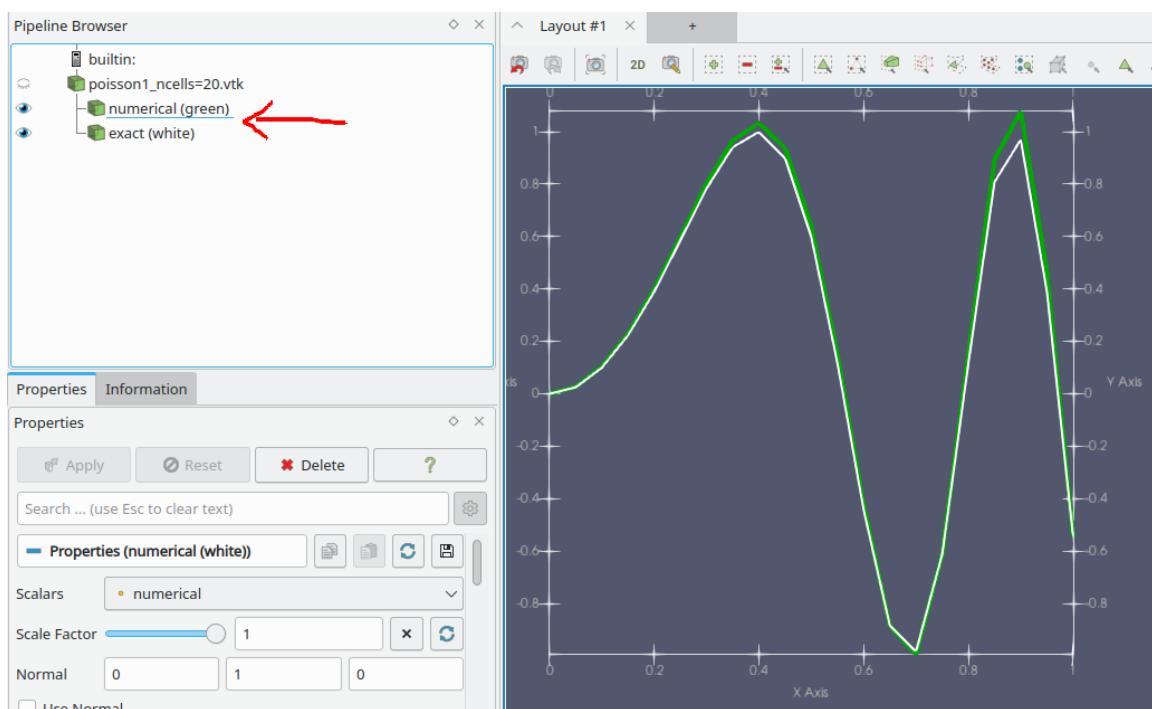
Настройка масштабов и отображение осей координат

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

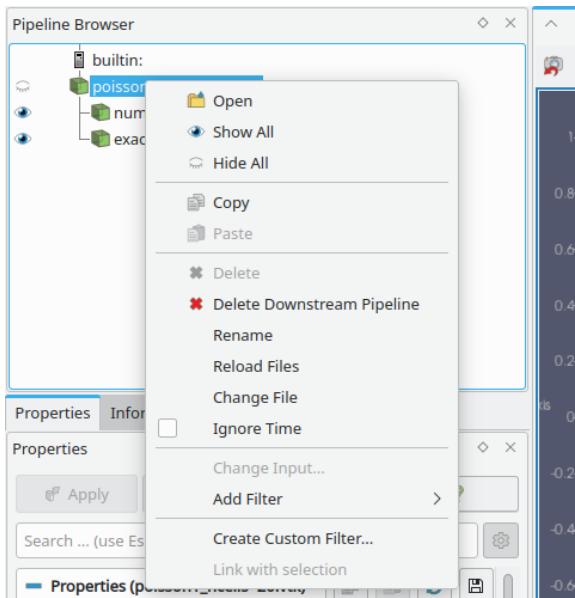
В случае, если масштабировать график не нужно, достаточно выполнить шаг 3.



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

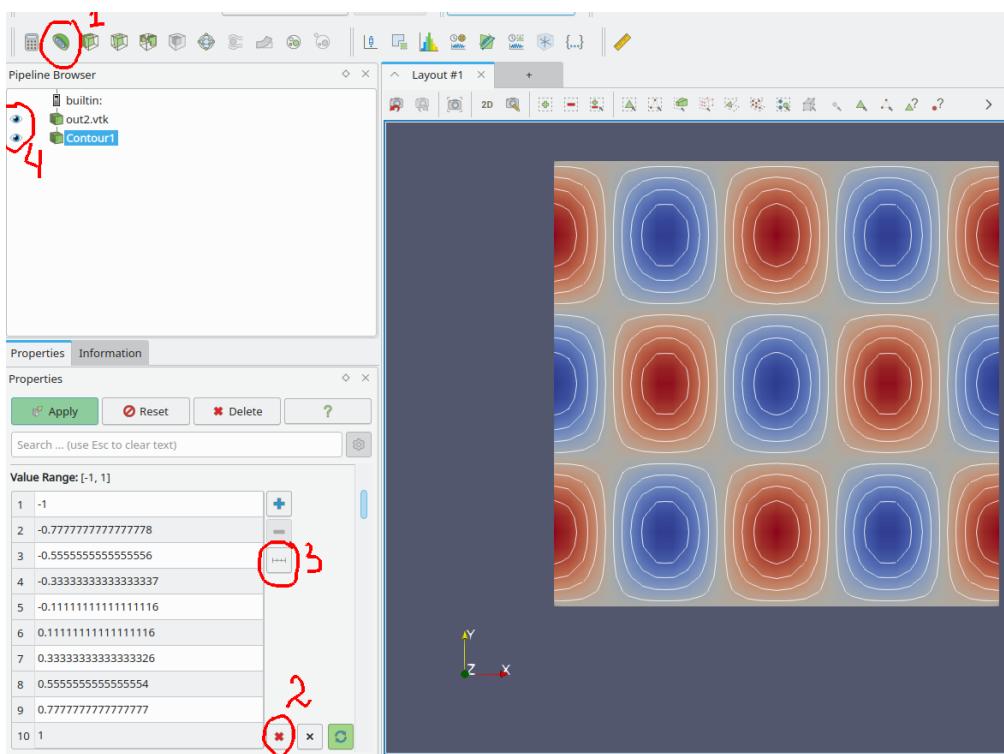


Обновление данных при изменении исходного файла В случае, если исходный файл был изменён, нужно в контекстном меню узла соответствующего файла выбрать **Reload Files** (или нажать F5). Если те же самые фильтры нужно применить для просмотра другого файла нужно в этом меню нажать **Change File**.

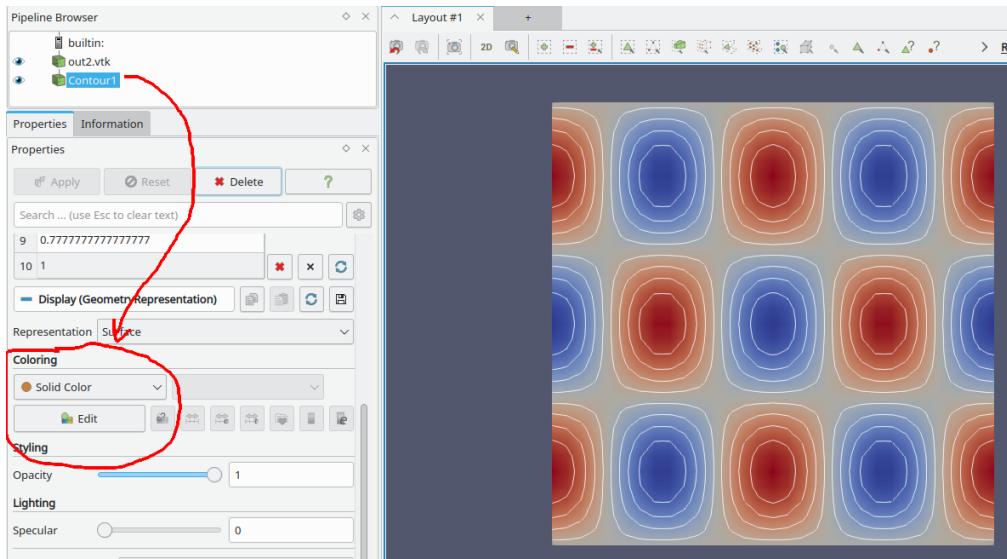


B.3.2 Изолинии для двумерного поля

- 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.



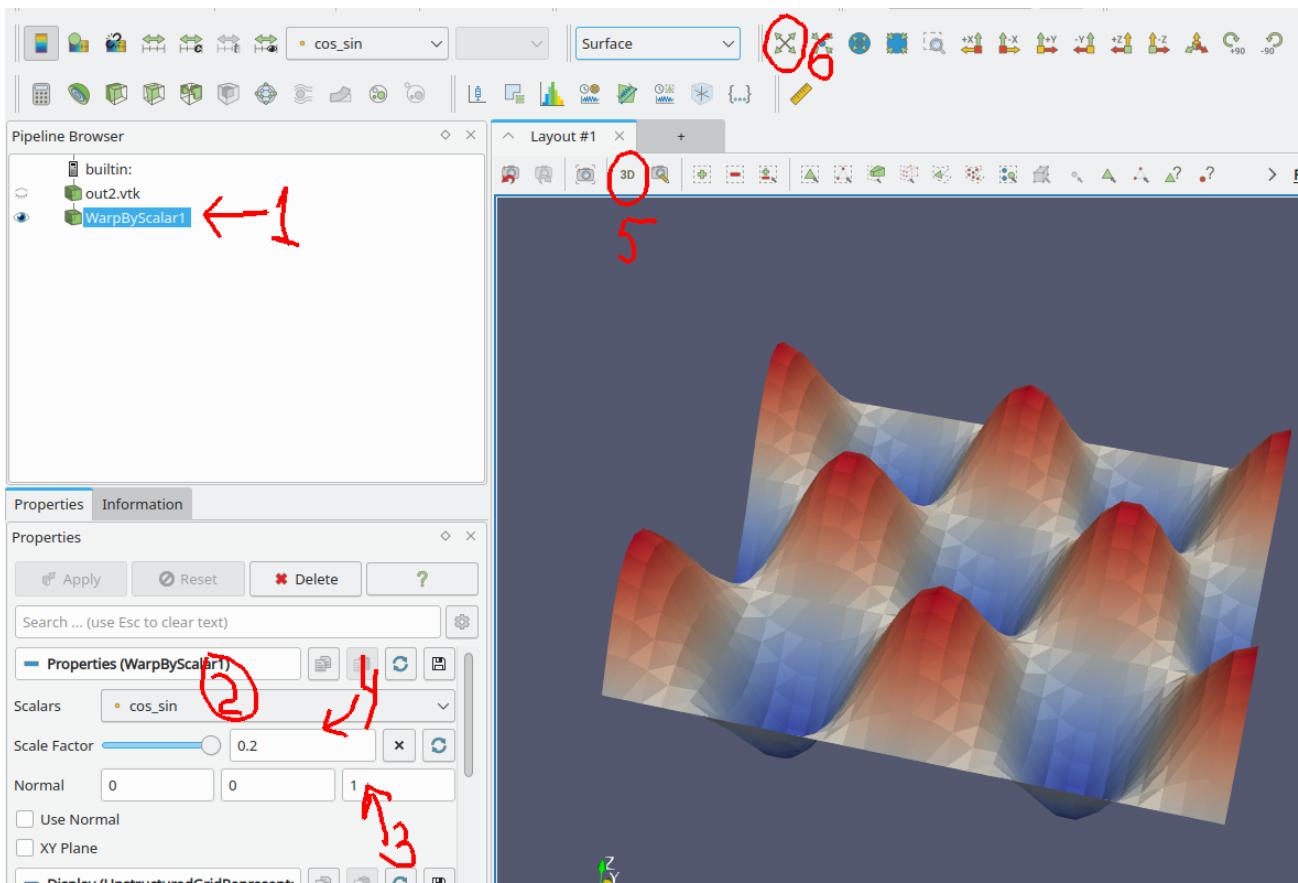
Задание цвета и толщины изолинии В случае, если нужно сделать изолинии одного цвета, установите поле **Coloring/Solid color** в настройках фильтра. Там же в меню **Edit** можно выбрать цвет. Для установления толщины линии включите подробные настройки и найдите там опцию **Styling/Line Width**.



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

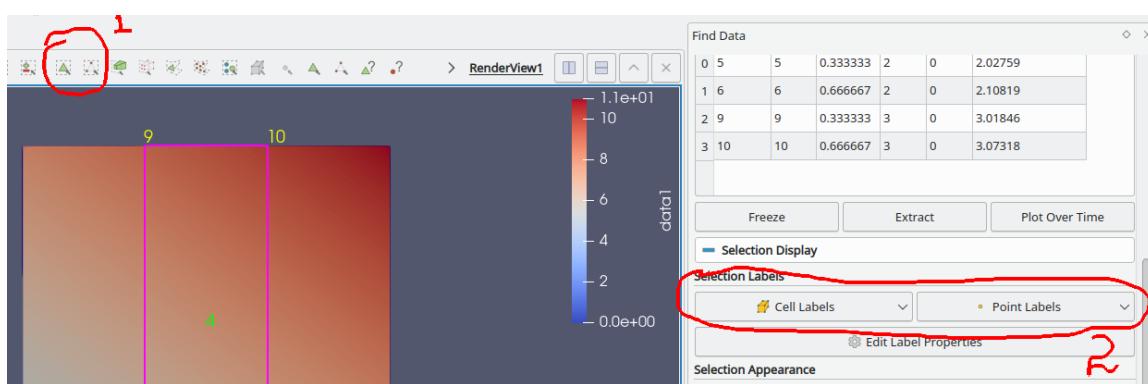
1. Включите фильтр **Filters/Warp By Scalar**
2. В настройках фильтра установите данные, которые будут проектироваться на координату z
3. Установите нормаль для проецирования (ось z)
4. Если нужно, выберите масштабирования для этой координаты
5. После нажатия **Apply** включите трёхмерное отображение
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**.

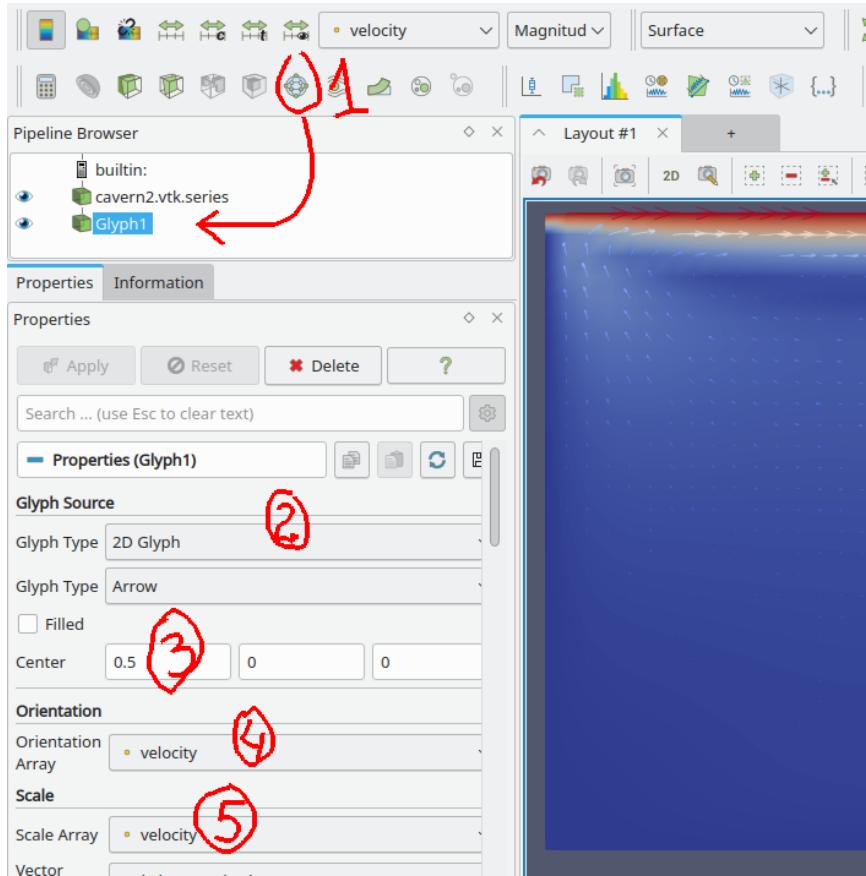


B.3.5 Векторные поля

Открыть файл vtk или vtk.series, который содержит векторное поле. Далее

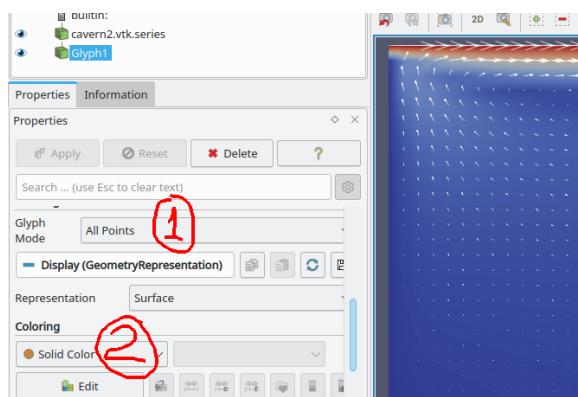
1. Создать фильтр **Glyph**

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

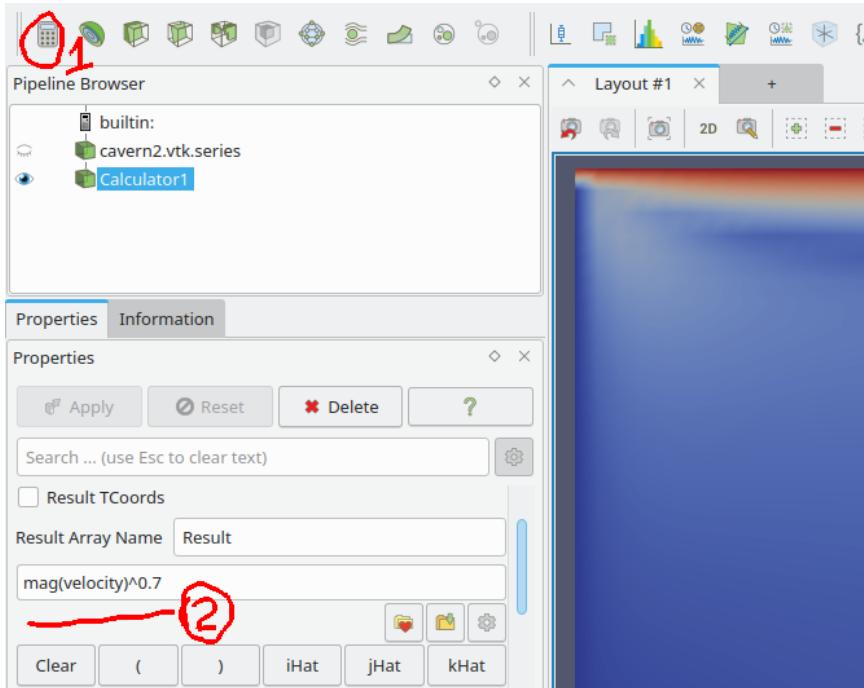
1. Выбрать необходимый **Glyph-mode**. Если сетка небольшая, то можно **All Points**.
2. Установить белый цвет для стрелок. Нажать **Apply**.



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

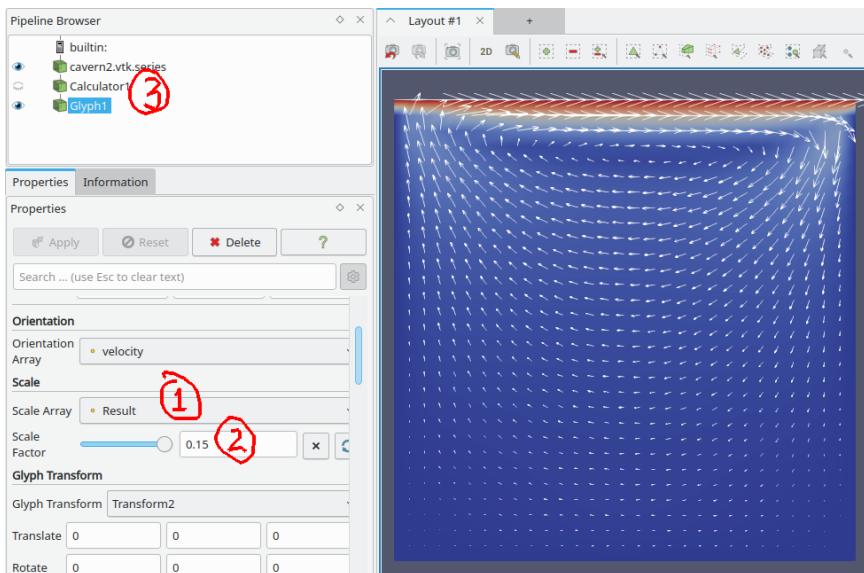
1. Начиная от загруженного файла создать фильтр **Calculator**

2. Там вбить необходимую формулу



Созданную функцию нужно прокинуть в **Glyph** в качестве коэффициента масштабирования

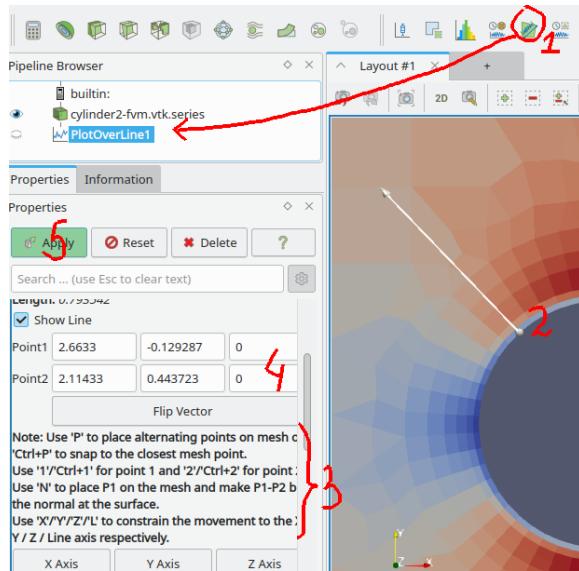
1. В **Scale Array** фильтра **Glyph** указать уже результат работы **Calculator**-а (**Result** по умолчанию),
2. Подтянуть значение **Scale Factor** до приемлимого
3. Не забыть отключить вспомогательное поле **Calculator** из отображения



B.3.6 Значение функции вдоль линии

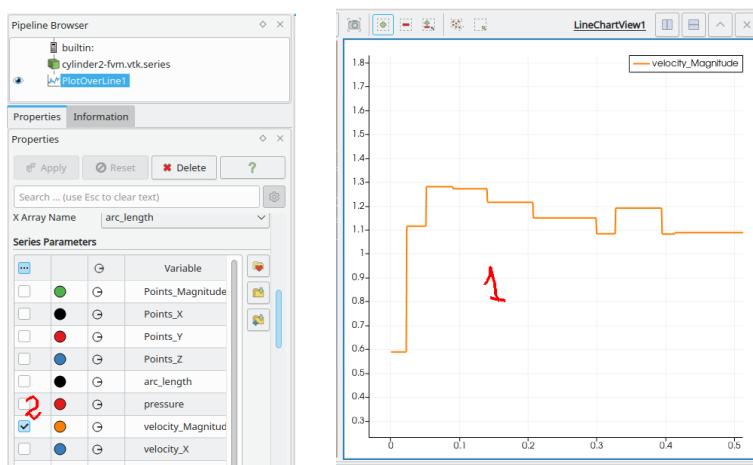
1. Выбрать фильтр **Plot Over Line** иконкой или в меню **Filters**
2. Установить начальную и конечную точку сечения

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**



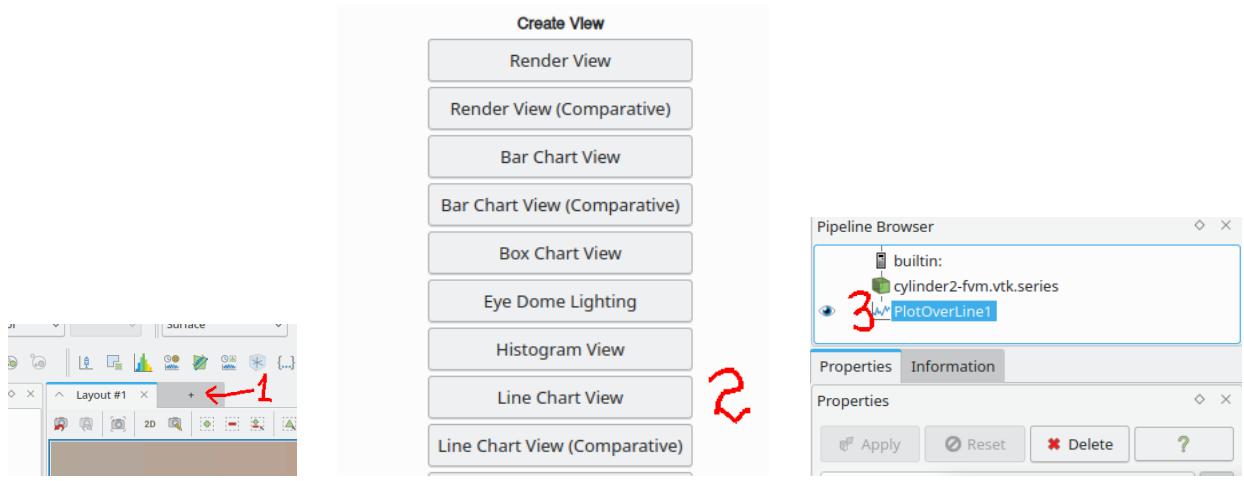
Настройка графика

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
```