

Содержание

1	Лекция 1 (10.02)	4
1.1	Задание для самостоятельной работы	4
2	Лекция 2 (17.02)	5
2.1	Расчётная сетка	5
3	Лекция 3 (24.02)	6
3.1	Структурированная расчётная сетка	6
3.2	Метод конечных разностей. Уравнение Пуассона	6
3.2.1	Постановка задачи	6
3.2.2	Метод решения	6
3.2.2.1	Нахождение численного решения	6
3.2.2.2	Практическое определения порядка аппроксимации	7
3.2.3	Программная реализация	8
3.2.3.1	Функция верхнего уровня	8
3.2.3.2	Детали реализации	9
3.3	Задание для самостоятельной работы	13
3.3.1	Одномерное уравнение Пуассона	13
3.3.2	Двумерное уравнение Пуассона	13
4	Лекция 4 (02.03)	15
4.1	Форматы хранения разреженных матриц	15
4.1.1	CSR-формат	15
4.1.2	Массив словарей	17
5	Лекция 5 (09.03)	19
5.1	Решение СЛАУ	19
5.1.1	Метод Якоби	19
5.1.2	Метод Зейделя	19
5.1.3	Метод последовательных верхних релаксаций (SOR)	20
5.2	Задание для самостоятельной работы	20
6	Лекция 6 (23.03)	24
6.1	Метод конечных объёмов	24
6.1.1	Уравнение Пуассона	24
6.1.1.1	Обработка внутренних граней	25
6.1.1.2	Учёт граничных условий первого рода	26
6.1.2	Одномерный случай	26
6.1.3	Сборка системы линейных уравнений	27
6.1.3.1	Алгоритм сборки в цикле по ячейкам	27
6.1.3.2	Алгоритм сборки в цикле по граням	28

6.2	Задание для самостоятельной работы	28
A	Формулы и обозначения	30
A.1	Векторы	31
A.1.1	Обозначение	31
A.1.2	Набла-нотация	31
A.2	Интегрирование	33
A.2.1	Формула Гаусса–Остроградского	33
A.2.2	Интегрирование по частям	33
A.2.3	Численное интегрирование в заданной области	34
A.3	Интерполяционные полиномы	35
A.3.1	Многочлен Лагранжа	35
A.3.1.1	Узловые базисные функции	35
A.3.1.2	Интерполяция в параметрическом отрезке	36
A.3.1.3	Интерполяция в параметрическом треугольнике	39
A.3.1.4	Интерполяция в параметрическом квадрате	41
A.4	Геометрические алгоритмы	44
A.4.1	Преобразование координат	44
A.4.1.1	Матрица Якоби	44
A.4.1.2	Дифференцирование в параметрической плоскости	45
A.4.1.3	Интегрирование в параметрической плоскости	46
A.4.1.4	Двумерное линейное преобразование. Параметрический треугольник	46
A.4.1.5	Двумерное билинейное преобразование. Параметрический квадрат	47
A.4.1.6	Трёхмерное линейное преобразование. Параметрический тетраэдр	47
A.4.2	Свойства многоугольника	47
A.4.2.1	Площадь многоугольника	47
A.4.2.2	Интеграл по многоугольнику	49
A.4.2.3	Центр масс многоугольника	49
A.4.3	Свойства многогранника	50
A.4.3.1	Объём многогранника	50
A.4.3.2	Интеграл по многограннику	50
A.4.3.3	Центр масс многогранника	50
A.4.4	Поиск многоугольника, содержащего заданную точку	50
B	Работа с инфраструктурой проекта CFDCourse	51
B.1	Сборка и запуск	52
B.1.1	Сборка проекта CFDCourse	52
B.1.1.1	Подготовка	52
B.1.1.2	VisualStudio	52
B.1.1.3	VSCode	54
B.1.2	Запуск конкретного теста	55
B.1.3	Сборка релизной версии	57
B.2	Git	59

В.2.1	Основные команды	59
В.2.2	Порядок работы с репозиторием CFDCourse	60
В.3	Paraview	62
В.3.1	Данные на одномерных сетках	62
В.3.2	Изолинии для двумерного поля	65
В.3.3	Данные на двумерных сетках в виде поверхности	66
В.3.4	Числовых значения в точках и ячейках	67
В.3.5	Векторные поля	67
В.3.6	Значение функции вдоль линии	69
В.4	Hybmesh	72
В.4.1	Работа в Windows	72
В.4.2	Работа в Linux	72

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

$$\begin{aligned} u_0 &= u_a, \\ u_N &= u_b \end{aligned} \tag{3.5}$$

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

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

и правой частью

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

Искомый вектор находится путём решения этой системы.

3.2.2.2 Практическое определения 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 параметрами. 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:

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

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

111 }

В программе в цикле по набору разбиений `n_cells`

123

создаётся решатель для тестовой задачи, использующий заданное число ячеек

125

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

вычисляется среднеквадратичная норма отклонения численного решения от точного

128

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

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

131

а полученная норма печатается в консоль напротив количества ячеек

134

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`:

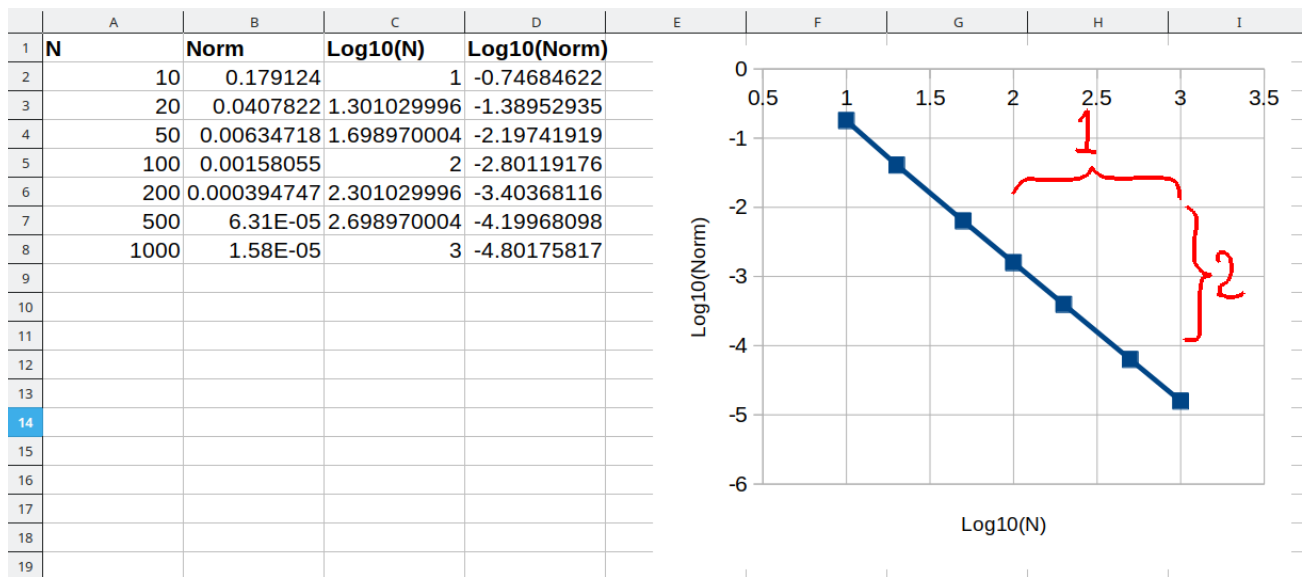


Рис. 1: Сходимость с уменьшением разбиения при решении одномерного уравнения Пуассона

```
14 namespace{
```

В методе

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

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

```
29 // returns norm2(u - u_exact)
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     AmgcMatrixSolver solver;
37     solver.set_matrix(mat);
38     solver.solve(rhs, u);
39
40     // 3. compute norm2
41     return compute_norm2();
```

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

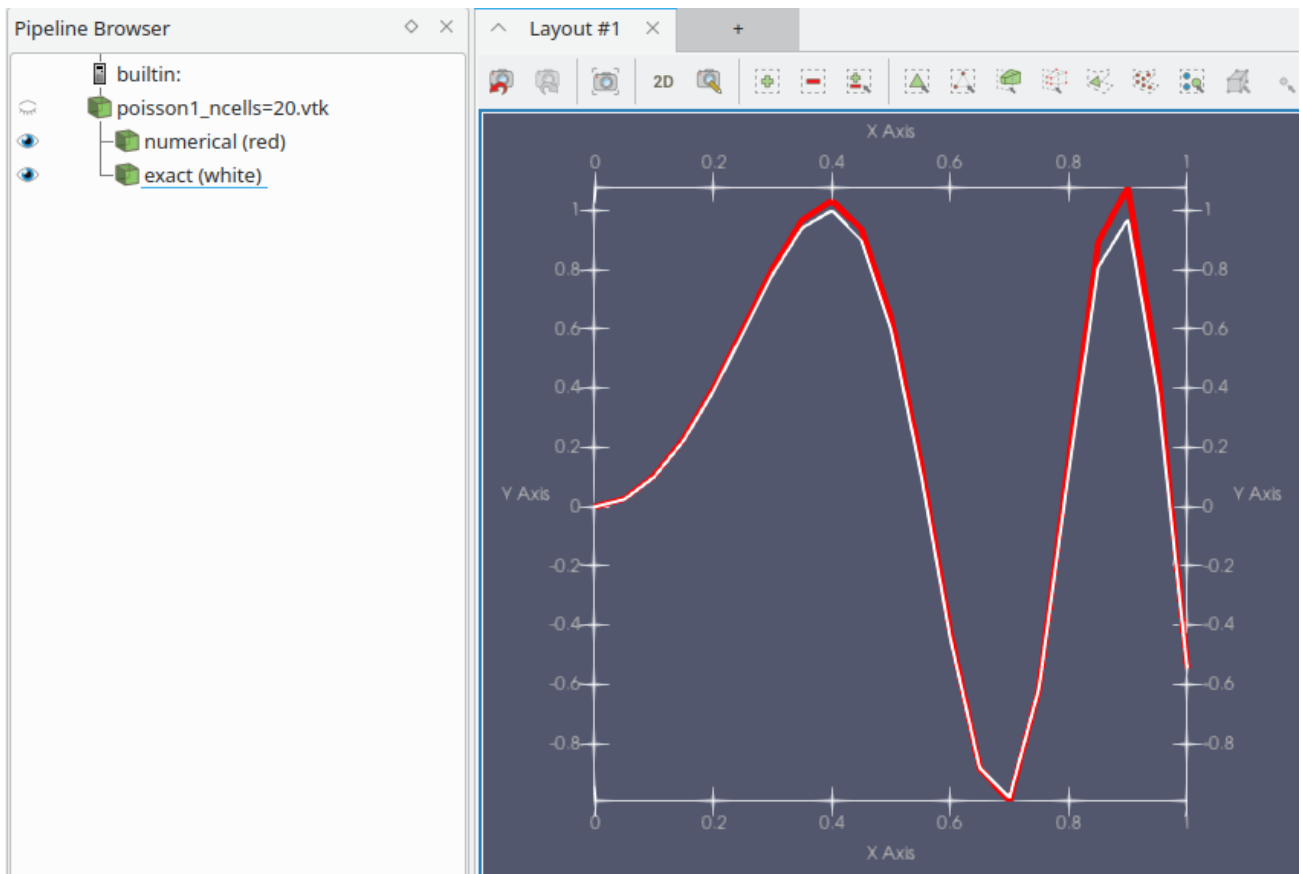


Рис. 2: Сравнение точного и численного решений уравнения Пуассона

- Сборка левой части СЛАУ. Реализует формулу (3.6). Для заполнения матрицы используется формат `cf::LodMatrix`, удобный для непоследовательной записи, который в конце конвертируется CSR.

```

63
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();

```

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

```

83
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;

```

- Вычисление нормы. Реализует формулу (3.9).

```

93
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);

```

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:

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

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

- `cfdd::RegularGrid2D::to_split_point_index`
- `cfdd::RegularGrid2D::to_linear_point_index`

В случае, если решатель системы линейных уравнений не решает построенную матрицу, использовать функцию `cfdd::dbg::print` для отладочной 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

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

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

Пусть матрица задана следующими массивами:

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

Число строк в матрице:

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

Число элементов в шаблоне (ненулевых элементов)

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

Число ненулевых элементов в заданной строке 'irow'

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

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

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

при этом избегаются лишние операции с нулями

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

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


```

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

```

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 отличие 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

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

```

AmgcMatrixSolver 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

- `cf::dbg::Tic` – вызвать до начала участка кода
- `cf::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 cf;

...

// 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 окончании 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 . Разобьём эту область на непересекающиеся подобласти E_i , $i = \overline{0, N-1}$ (рис. 3). Центры ячеек обозначим как \mathbf{c}_i .

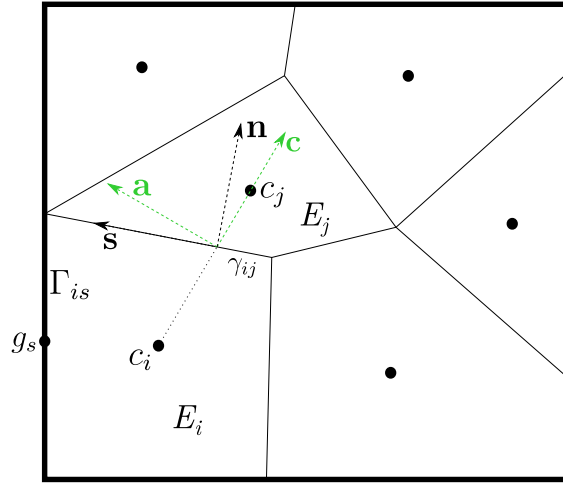


Рис. 3: Конечнoобъёмная сетка

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) нормальям 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 *ребі* (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 неструктурированных *ребі*-сеток используют алгоритмы построения ячеек Вороного. Для *ребі*-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

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

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):

$$\frac{\partial u}{\partial n} \approx \frac{u^\Gamma - u_i}{h_{is}}, \quad h_{is} = |\mathbf{g}_s - \mathbf{c}_i|. \quad (6.6)$$

6.1.2 Одномерный случай

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

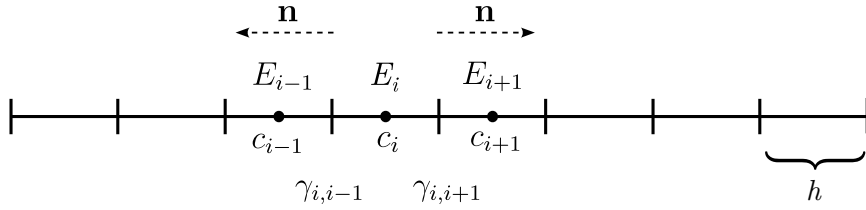


Рис. 4: Одномерная конечнообъёмная сетка

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

$$\gamma_{i,i-1} : \quad \frac{\partial u}{\partial n} = \frac{u_{i-1} - u_i}{h}$$

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

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 аппроксимацию уравнения Пуассона

$$\frac{-u_{i-1} + 2u_i - u_{i+1}}{h} = f_i h,$$

которая имеет второй порядок точности. Разница с методом конечных разностей здесь состоит в том, что значения сеточных векторов $\{u\}$, $\{f\}$ здесь приписаны к центрам ячеек, а не к их узлам.

Это отличие проявит себя в аппроксимации граничных условий. Так, если на левой границе задано условие первого рода, то соответствующее уравнение согласно (6.6) 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), (6.6) 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:

$$\sum_j \frac{|\gamma_{ij}|}{h_{ij}} (u_i - u_j) + \sum_{s \in I} \frac{|\Gamma_{is}|}{h_{is}} u_i = f_i |E_i| + \sum_{s \in I} \frac{|\Gamma_{is}|}{h_{is}} u^\Gamma \quad (6.7)$$

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 (6.7) можно собирать в цикле по ячейкам: строка за строкой. 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 удаётся избежать, потому что у грани есть только две соседние ячейки (в то время как у ячейки может быть произвольное количество соседних граней).

Разделим все грани на исходной сетки на внутренние и граничные (отдельный набор для каждого вида граничных условий). Тогда для внутренних граней можно записать

$$\begin{aligned}
 &\textbf{for } s \in \text{internal} && \text{-- цикл по внутренним граням} \\
 &\quad i, j = \text{nei_cells}(s) && \text{-- две ячейки, соседние с текущей гранью} \\
 &\quad v = |\gamma_{ij}|/h_{ij} \\
 &\quad A_{ii} += v; \quad A_{jj} += v && \text{-- диагональные коэффициенты матрицы} \\
 &\quad A_{ij} -= v; \quad A_{ji} -= v && \text{-- внедиагональные коэффициенты матрицы} \\
 &\textbf{endfor}
 \end{aligned} \tag{6.8}$$

Граничные условия учитываются в отдельных циклах. Здесь будем учитывать, что у грани, принадлежащей границе области, есть только одна соседняя ячейка. Условия первого рода:

$$\begin{aligned}
 &\textbf{for } s \in \text{bnd1} && \text{-- грани с условиями первого рода} \\
 &\quad i = \text{nei_cells}(s) && \text{-- соседняя с граничной гранью ячейка} \\
 &\quad v = |\Gamma_{is}|/h_{is} \\
 &\quad A_{ii} += v \\
 &\quad b_i += u^\Gamma v \\
 &\textbf{endfor}
 \end{aligned} \tag{6.9}$$

Первое слагаемое в правой части (6.7) учтём отдельным циклом:

$$\begin{aligned}
 &\textbf{for } i = \overline{0, N-1} && \text{-- цикл по ячейкам} \\
 &\quad b_i = |E_i| f_i \\
 &\textbf{endfor}
 \end{aligned} \tag{6.10}$$

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 (структурированных, pebi и скошенных)
3. визуализировать решение, полученное на этих сетках
4. построить графики сходимости решения и определить порядок аппроксимации метода

Построение неструктурированных сеток

В папке

`test_data` корневой директории репозитория лежат скрипты построения сеток в программе HybMesh :

- `pebigrid.py` – pebi-сетка,
- `tetragrid.py` – сетка, состоящая из произвольных (скошенных) трех- и четырехугольников.

Инструкции по запуску этих скриптов смотри п. В.4. Эти скрипты строят равномерную неструктурированную сетку в единичном квадрате и записывают её в файл `vtk`, который впоследствии можно загрузить в расчётную программу. В каждом из скриптов есть параметр `N`, означающий примерное количество ячеек в итоговой сетке. Меняя его значение можно строить сетки разного разрешения.

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/pebigrid.vtk` и создаст рабочий класс с использованием прочитанной сетки

```
std::string fn = test_directory_file("pebigrid.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.

А Формулы и обозначения

А.1 Векторы

А.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 декартовы 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 $\boldsymbol{\xi} = (\xi, \eta, \zeta)$.

Операции в векторах 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} .

А.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 покомпонентные 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):

$$\nabla \times \mathbf{v} = \begin{vmatrix} \hat{\mathbf{x}} & \hat{\mathbf{y}} & \hat{\mathbf{z}} \\ \nabla_x & \nabla_y & \nabla_z \\ v_x & v_y & v_z \end{vmatrix} = \left(\frac{\partial v_z}{\partial y} - \frac{\partial v_y}{\partial z} \right) \hat{\mathbf{x}} - \left(\frac{\partial v_z}{\partial x} - \frac{\partial v_x}{\partial z} \right) \hat{\mathbf{y}} + \left(\frac{\partial v_y}{\partial x} - \frac{\partial v_x}{\partial y} \right) \hat{\mathbf{z}}. \quad (\text{A.8})$$

А.2 Интегрирование

А.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. 5. Разобьём область на $N = 3$ равномерных подобласти

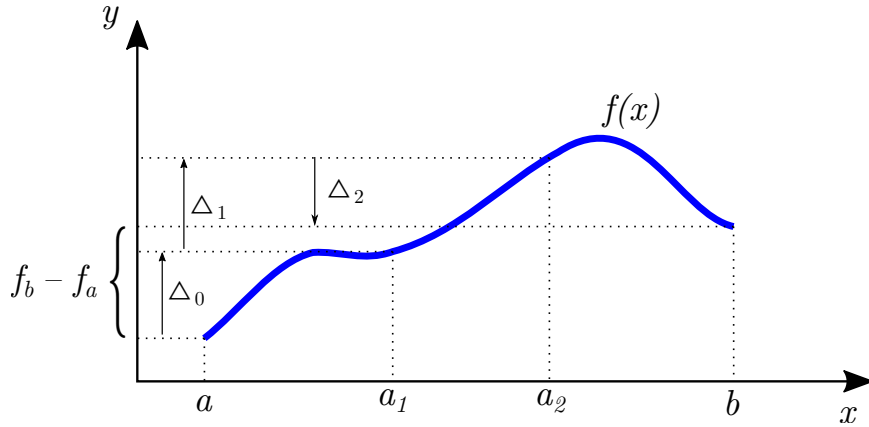


Рис. 5: Формула Гаусса–Остроградского в одномерном случае

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.

А.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

$$\int_E \frac{\partial f}{\partial x} \, d\mathbf{x} = \int_{\Gamma} f \cos(\widehat{\mathbf{n}}, \mathbf{x}) \, ds \quad (\text{A.11})$$

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 .

А.3 Интерполяционные полиномы

А.3.1 Многочлен Лагранжа

А.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

$$f(\xi) = \text{const} \quad \Rightarrow \quad f_i = \text{const}.$$

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 $CA = E$:

$$\begin{pmatrix} 1 & \xi_0 & \xi_0^2 & \xi_0^3 & \dots \\ 1 & \xi_1 & \xi_1^2 & \xi_1^3 & \dots \\ 1 & \xi_2 & \xi_2^2 & \xi_2^3 & \dots \\ 1 & \xi_3 & \xi_3^2 & \xi_3^3 & \dots \\ \dots & & & & \end{pmatrix} \begin{pmatrix} A_0^{(0)} & A_1^{(0)} & A_2^{(0)} & A_3^{(0)} & \dots \\ A_0^{(1)} & A_1^{(1)} & A_2^{(1)} & A_3^{(1)} & \dots \\ A_0^{(2)} & A_1^{(2)} & A_2^{(2)} & A_3^{(2)} & \dots \\ A_0^{(3)} & A_1^{(3)} & A_2^{(3)} & A_3^{(3)} & \dots \\ \vdots & & & & \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & \dots \\ 0 & 1 & 0 & 0 & \dots \\ 0 & 0 & 1 & 0 & \dots \\ 0 & 0 & 0 & 1 & \dots \\ \vdots & & & & \end{pmatrix}$$

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 интерполяционным полиномом.

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

$$A = \begin{pmatrix} A_0^{(0)} & A_1^{(0)} \\ A_0^{(1)} & A_1^{(1)} \end{pmatrix} = C^{-1} = \begin{pmatrix} & \phi_0 & \phi_1 \\ \hline 1 & \frac{1}{2} & \frac{1}{2} \\ \xi & -\frac{1}{2} & \frac{1}{2} \end{pmatrix}.$$

Отсюда узловые базисные функции примут вид (рис. 6)

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

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

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

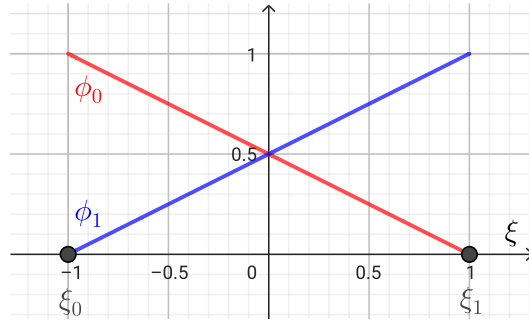


Рис. 6: Линейный базис в параметрическом отрезке

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). Поче-

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. 7)

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

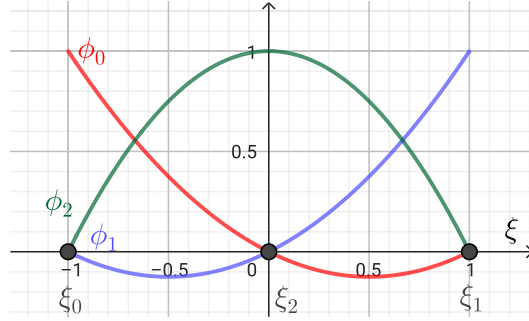


Рис. 7: Квадратичный базис в параметрическом отрезке

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. 8)

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

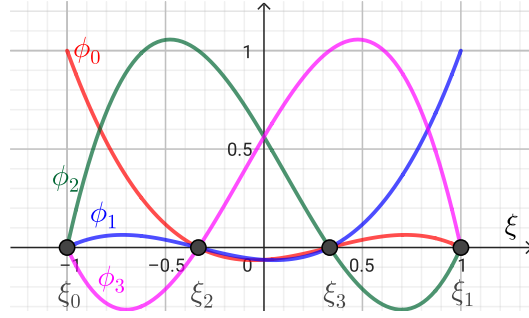


Рис. 8: Кубический базис в параметрическом отрезке

На рис. 9 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.

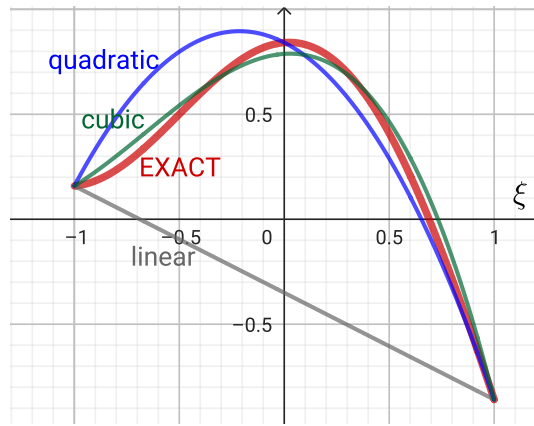


Рис. 9: Результат интерполяции

А.3.1.3 Интерполяция в параметрическом треугольнике

Теперь рассмотрим двумерное обобщение формулы

Линейный базис

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

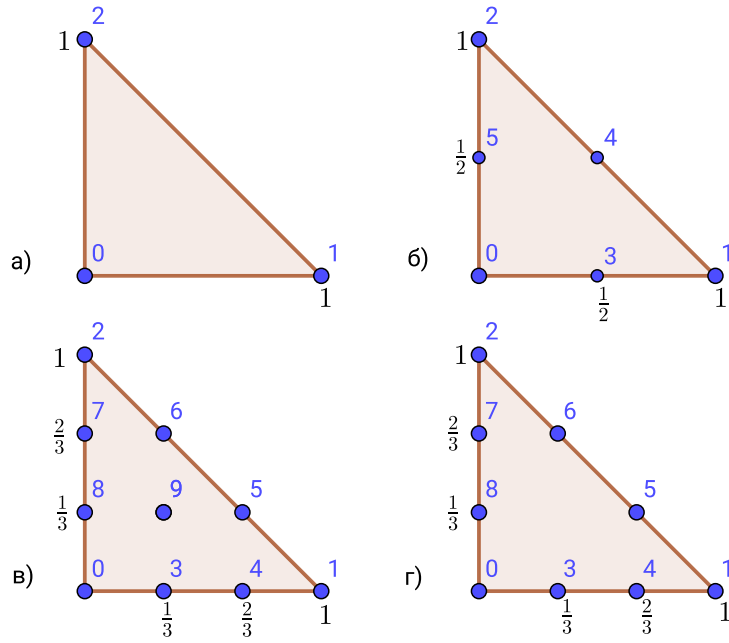


Рис. 10: Расположение узловых точек в параметрическом треугольнике. а) линейный базис, б) квадратичный базис, в) кубический базис, г) неполный кубический базис

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

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

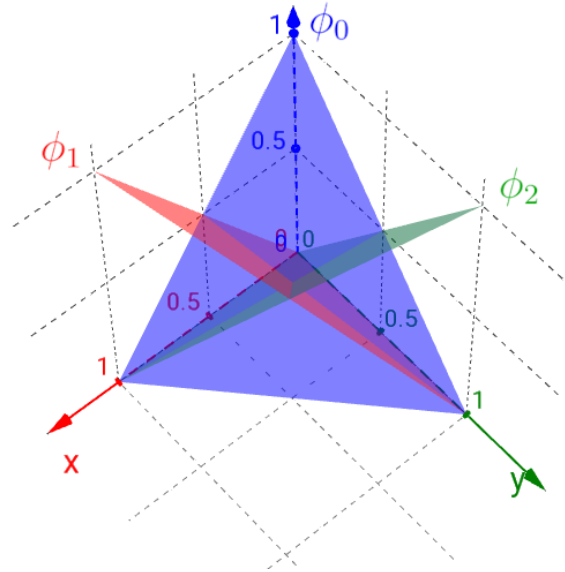


Рис. 11: Линейный базис в параметрическом треугольнике

Квадратичный базис

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

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

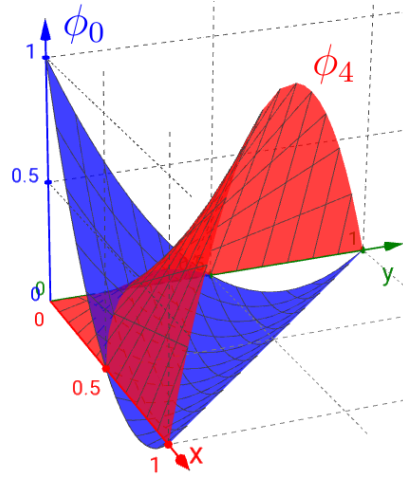


Рис. 12: Квадратичные функции ϕ_0, ϕ_4 в параметрическом треугольнике

Кубический базис TODO

Неполный кубический базис TODO

А.3.1.4 Интерполяция в параметрическом квадрате

Билинейный базис

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

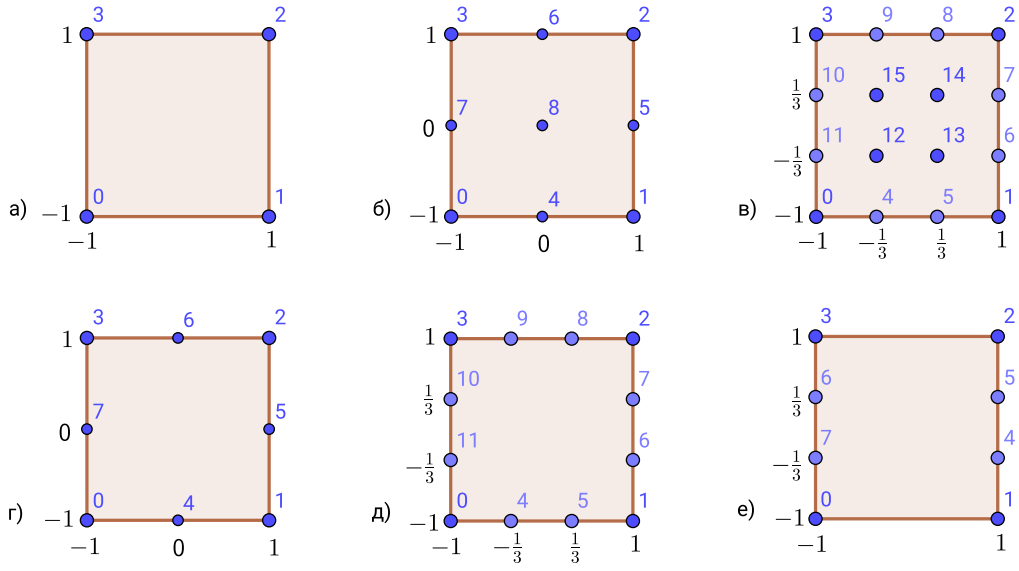


Рис. 13: Расположение узловых точек в параметрическом квадрате

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

$$\phi_0(\xi, \eta) = \frac{1 - \xi - \eta + \xi\eta}{4}$$

$$\phi_1(\xi, \eta) = \frac{1 + \xi - \eta - \xi\eta}{4}$$

$$\phi_2(\xi, \eta) = \frac{1 + \xi + \eta + \xi\eta}{4}$$

$$\phi_3(\xi, \eta) = \frac{1 - \xi + \eta - \xi\eta}{4}$$

(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:

$$\xi_0 = (\xi_0^{1D}, \xi_0^{1D}), \quad \xi_1 = (\xi_1^{1D}, \xi_0^{1D}), \quad \xi_2 = (\xi_1^{1D}, \xi_1^{1D}), \quad \xi_3 = (\xi_0^{1D}, \xi_1^{1D}).$$

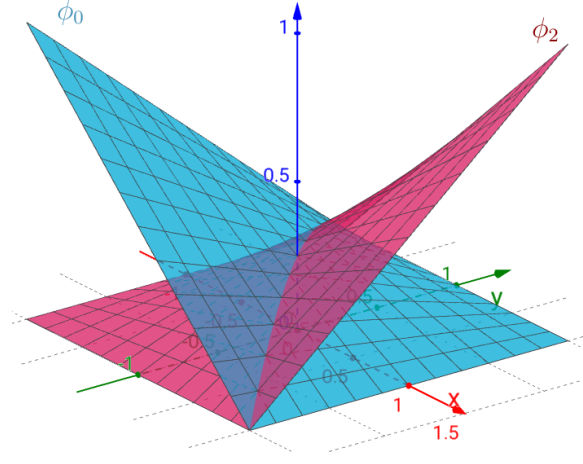


Рис. 14: Билинейные функции ϕ_0, ϕ_2 в параметрическом квадрате

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 убедиться, 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. 13б. 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

А.4 Геометрические алгоритмы

А.4.1 Преобразование координат

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

А.4.1.1 Матрица Якоби

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

$$\begin{aligned} x(\xi_0 + d\xi, \eta_0 + d\eta) &\approx x_0 + \left. \frac{\partial x}{\partial \xi} \right|_{\xi_0, \eta_0} d\xi + \left. \frac{\partial x}{\partial \eta} \right|_{\xi_0, \eta_0} d\eta, \\ y(\xi_0 + d\xi, \eta_0 + d\eta) &\approx y_0 + \left. \frac{\partial y}{\partial \xi} \right|_{\xi_0, \eta_0} d\xi + \left. \frac{\partial y}{\partial \eta} \right|_{\xi_0, \eta_0} d\eta, \end{aligned}$$

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

$$\mathbf{x}(\xi_0 + d\xi) - \mathbf{x}_0 = J(\xi_0) d\xi. \quad (\text{A.26})$$

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

$$J = \begin{pmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{pmatrix} = \begin{pmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{pmatrix} \quad (\text{A.27})$$

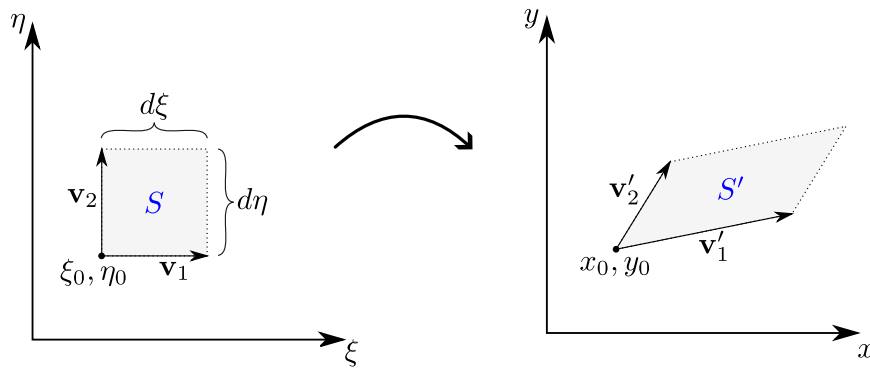


Рис. 15: Преобразование элементарного объёма

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. 15). В результате преобразования по формуле (A.26) получим следующие преобразования концевых точек

и векторов:

$$\begin{aligned} (\xi_0, \eta_0) &\rightarrow (x_0, y_0), \\ (\xi_0 + d\xi, \eta_0) &\rightarrow (x_0 + J_{11}d\xi, y_0 + J_{21}d\xi) \Rightarrow \mathbf{v}_1 \rightarrow \mathbf{v}'_1 = (J_{11}d\xi, J_{21}d\xi), \\ (\xi_0, \eta_0 + d\eta) &\rightarrow (x_0 + J_{12}d\eta, y_0 + J_{22}d\eta) \Rightarrow \mathbf{v}_2 \rightarrow \mathbf{v}'_2 = (J_{12}d\eta, J_{22}d\eta). \end{aligned}$$

Элементарный объём равен площади параллелограмма, построенного на элементарных векторах. В параметрической плоскости согласно (A.4) получим

$$|S| = \mathbf{v}_1 \times \mathbf{v}_2 = d\xi d\eta,$$

и аналогично для физической плоскости:

$$|S'| = \mathbf{v}'_1 \times \mathbf{v}'_2 = (J_{11}J_{22} - J_{12}J_{21})d\xi d\eta = |J|d\xi d\eta$$

Сравнивая два последних соотношения приходим к выводу, что элементарный объём в результате преобразования увеличился в $|J|$ раз. Тогда можно записать

$$dx dy = |J| d\xi d\eta \quad (\text{A.28})$$

A.4.1.2 Дифференцирование в параметрической плоскости

Пусть задана некоторая функция $f(x, y)$. Распишем её производную по параметрическим координатам:

$$\begin{aligned} \frac{\partial f}{\partial \xi} &= \frac{\partial f}{\partial x} \frac{\partial x}{\partial \xi} + \frac{\partial f}{\partial y} \frac{\partial y}{\partial \xi}, \\ \frac{\partial f}{\partial \eta} &= \frac{\partial f}{\partial x} \frac{\partial x}{\partial \eta} + \frac{\partial f}{\partial y} \frac{\partial y}{\partial \eta}. \end{aligned}$$

Вспоминая определение (A.27), запишем

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

Обратная зависимость примет вид

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

A.4.1.3 Интегрирование в параметрической плоскости

Пусть в физической области x, y задана область D_x . Интеграл функции $f(x, y)$ по этой области можно расписать, используя замену (A.28)

$$\int_{D_x} f(x, y) dx dy = \int_{D_\xi} f(\xi, \eta) |J(\xi, \eta)| d\xi d\eta, \quad (\text{A.29})$$

где $f(\xi, \eta) = f(x(\xi, \eta), y(\xi, \eta))$, а D_ξ – образ области D_x в параметрической плоскости.

A.4.1.4 Двумерное линейное преобразование. Параметрический треугольник

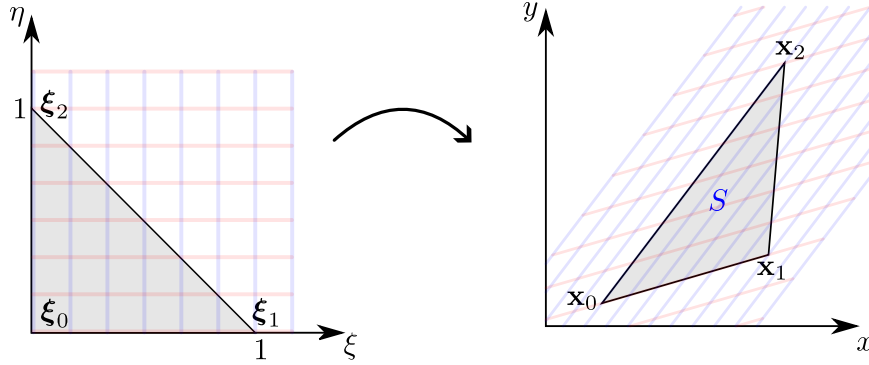


Рис. 16: Преобразование из параметрического треугольника

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. 16):

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

Тогда система линейных уравнений примет вид

$$\begin{aligned} x_0 &= C_x, & y_0 &= C_y, \\ x_1 &= A_x + C_x, & y_1 &= A_y + C_y, \\ y_2 &= B_x + C_x, & y_2 &= B_y + C_y. \end{aligned}$$

Определив коэффициенты преобразования из этой системы, окончательно запишем преобразование

$$\begin{aligned} x(\xi, \eta) &= (x_1 - x_0)\xi + (x_2 - x_0)\eta + x_0, \\ y(\xi, \eta) &= (y_1 - y_0)\xi + (y_2 - y_0)\eta + y_0. \end{aligned} \quad (\text{A.30})$$

Матрица Якоби этого преобразования (A.27) не будет зависеть от параметрических координат ξ, η :

$$J = \begin{pmatrix} x_1 - x_0 & x_2 - x_0 \\ y_1 - y_0 & y_2 - y_0 \end{pmatrix}. \quad (\text{A.31})$$

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

$$|J| = (x_1 - x_0)(y_2 - y_0) - (y_1 - y_0)(x_2 - x_0) = (\mathbf{x}_1 - \mathbf{x}_0) \times (\mathbf{x}_2 - \mathbf{x}_0) = 2|S|. \quad (\text{A.32})$$

Распишем интеграл по треугольнику S по формуле (A.29). Вследствии линейности преобразования якобиан постоянен и, поэтому, его можно вынести его из-под интеграла:

$$\int_S f(x, y) dx dy = |J| \int_0^1 \int_0^{1-\xi} f(\xi, \eta) d\eta d\xi. \quad (\text{A.33})$$

A.4.1.5 Двумерное билинейное преобразование. Параметрический квадрат

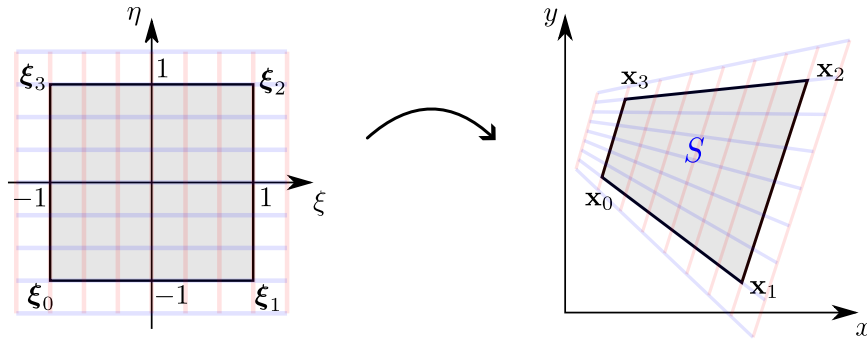


Рис. 17: Преобразование из параметрического квадрата

A.4.1.6 Трёхмерное линейное преобразование. Параметрический тетраэдр

TODO

A.4.2 Свойства многоугольника

A.4.2.1 Площадь многоугольника

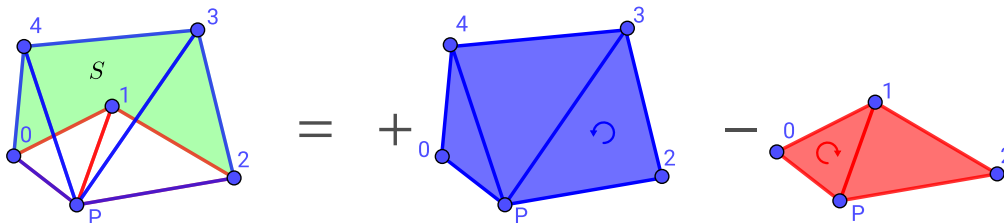


Рис. 18: Площадь произвольного многоугольника

Рассмотрим произвольный несамопересекающийся N -угольник S , заданный координатами своих узлов \mathbf{x}_i , $i = \overline{0, N-1}$, пронумерованных последовательно против часовой стрелки (рис. 18). Далее введём произвольную точку \mathbf{p} и от этой точки будем строить ориентированные треугольники до граней многоугольника:

$$\triangle_i^p = (\mathbf{p}, \mathbf{x}_i, \mathbf{x}_{i+1}), \quad i = \overline{0, N-1},$$

(для корректности записи будем считать, что $\mathbf{x}_N = \mathbf{x}_0$). Тогда площадь исходного многоугольника S будет равна сумме знаковых площадей треугольников \triangle_i^p :

$$|S| = \sum_{i=0}^{N-1} |\triangle_i^p|, \quad |\triangle_i^p| = \frac{(\mathbf{x}_i - \mathbf{p}) \times (\mathbf{x}_{i+1} - \mathbf{p})}{2}.$$

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 18 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 $\triangle_0^p, \triangle_{N-1}^p$ вырождаются (будут иметь нулевую площадь). Обозначим такую последовательную триангуляцию как

$$\triangle_i = (\mathbf{x}_0, \mathbf{x}_i, \mathbf{x}_{i+1}), \quad i = \overline{1, N-2}. \quad (\text{A.34})$$

Знаковая площадь ориентированного треугольника будет равна

$$|\triangle_i| = \frac{(\mathbf{x}_i - \mathbf{x}_0) \times (\mathbf{x}_{i+1} - \mathbf{x}_0)}{2}. \quad (\text{A.35})$$

Тогда окончательно формула определения площади примет вид

$$|S| = \sum_{i=1}^{N-2} |\triangle_i|. \quad (\text{A.36})$$

Плоский полигон в пространстве Если плоский полигон S расположен в трёхмерном пространстве, то правая часть формулы (A.35) согласно определению векторного произведения в трёхмерном пространстве (A.3) – есть вектор. Чтобы получить скалярную площадь, нужно спроецировать 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:

$$|\triangle_i| = \frac{((\mathbf{x}_i - \mathbf{x}_0) \times (\mathbf{x}_{i+1} - \mathbf{x}_0)) \cdot \mathbf{n}}{2}. \quad (\text{A.37})$$

Формула для определения площади полигона (A.36) будет по-прежнему верна. При этом итоговый знак величины S будет положительным, если закрутка полигона положительная (против часовой стрелки) при взгляде со стороны вычисленной нормали \mathbf{n} .

A.4.2.2 Интеграл по многоугольнику

Рассмотрим интеграл функции $f(x, y)$ по N -угольнику S , заданному последовательными координатами своих узлов \mathbf{x}_i . Введём последовательную триангуляцию согласно (A.34). Тогда интеграл по многоугольнику можно расписать как сумму интегралов по ориентированным треугольникам:

$$\int_S f(x, y) dx dy = \sum_{i=1}^{N-2} \int_{\Delta_i} f(x, y) dx dy. \quad (\text{A.38})$$

Далее для вычисления интегралов в правой части воспользуемся преобразованием к параметрическому треугольнику (п. A.4.1.4). Следуя формуле интегрирования (A.33), распишем интеграл по i -ому треугольнику:

$$\int_{\Delta_i} f(x, y) dx dy = |J_i| \int_0^1 \int_0^{1-\xi} f_i(\xi, \eta) d\eta d\xi,$$

где якобиан $|J_i|$ согласно (A.32) есть удвоенная площадь ориентированного треугольника Δ_i (положительная при закрутке против часовой стрелке и отрицательная иначе):

$$|J_i| = 2|\Delta_i| = (\mathbf{x}_i - \mathbf{x}_0) \times (\mathbf{x}_{i+1} - \mathbf{x}_0),$$

а функция $f_i(\xi, \eta)$ есть функция от преобразованных согласно (A.30) переменных:

$$f_i(\xi, \eta) = f((\mathbf{x}_i - \mathbf{x}_0)\xi + (\mathbf{x}_{i+1} - \mathbf{x}_0)\eta + \mathbf{x}_0).$$

Окончательно запишем

$$\int_S f(x, y) dx dy = 2 \sum_{i=1}^{N-2} |\Delta_i| \int_0^1 \int_0^{1-\xi} f_i(\xi, \eta) d\eta d\xi. \quad (\text{A.39})$$

Отметим, что эта формула работает и в том случае, когда полигон расположен в трёхмерном пространстве (знаковую площадь при этом следует вычислять по (A.37)).

A.4.2.3 Центр масс многоугольника

По определению, координаты центра масс \mathbf{s} области S равны среднеинтегральным значениям координатных функций. То есть

$$c_x = \frac{1}{|S|} \int_S x dx dy, \quad c_y = \frac{1}{|S|} \int_S y dx dy.$$

Далее распишем интеграл в правой части через последовательную триангуляцию согласно (A.38) с учётом линейного преобразования (A.30):

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

Итого, с учётом (A.36), координаты центра масс примут вид

$$\mathbf{c} = \frac{\sum_{i=1}^{N-2} \frac{\mathbf{x}_0 + \mathbf{x}_i + \mathbf{x}_{i+1}}{3} |\Delta_i|}{\sum_{i=1}^{N-2} |\Delta_i|}.$$

Если полигон расположен в двумерном пространстве xy , то знаковая площадь треугольников вычисляется по формуле (A.35). В случае трёхмерного пространства должна использоваться формула (A.37).

A.4.3 Свойства многогранника

A.4.3.1 Объём многогранника

TODO

A.4.3.2 Интеграл по многограннику

TODO

A.4.3.3 Центр масс многогранника

TODO

A.4.4 Поиск многоугольника, содержащего заданную точку

TODO

В Работа с инфраструктурой проекта CFDCourse

В.1 Сборка и запуск

В.1.1 Сборка проекта CFDCourse

Описанная ниже процедура собирает проект в отладочной конфигурации. Для проведения необходимых модификаций для сборки релизной версии смотри [В.1.3](#).

В.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 репозитория. Например `D:/git_repos/`
3. Возьмите необходимые заголовочные библиотеки boost из <https://disk.yandex.ru/d/GwTZUvfAqPsZ> и распакуйте архив в папку для репозитория (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 репозитория

```
> cd D:/git_repos
```

6. Клонировать репозиторий

```
> git clone https://github.com/kalininei/CFDCourse25
```

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

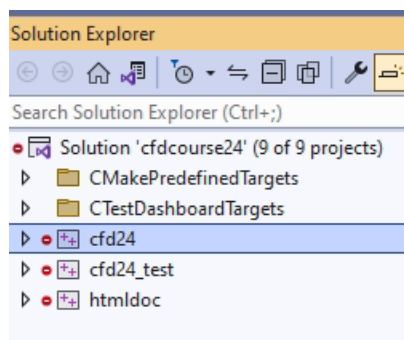
В.1.1.2 VisualStudio

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

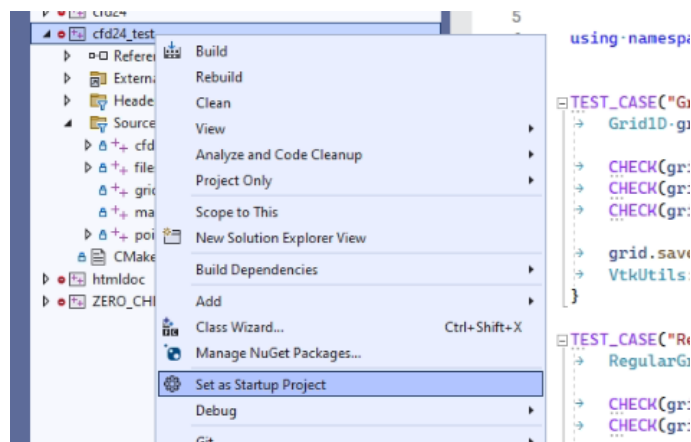
```
SET CMGenerator="Visual Studio 17 2022"
SET CMGenerator="Visual Studio 16 2019"
SET CMGenerator="Visual Studio 15 2017"
SET CMGenerator="Visual Studio 14 2015"
```

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:

- **cfd25** – расчётная библиотека
 - **cfd25_test** – модульные тесты для расчётных функций
6. Проект **cfd25_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 **cfid25_test**.

- Запустить тесты (проект **cfid25_test**) нажав **F5** (или кнопку отладки в меню). После отработки должно высветиться сообщение об успешном прохождении всех тестов.
- 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

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

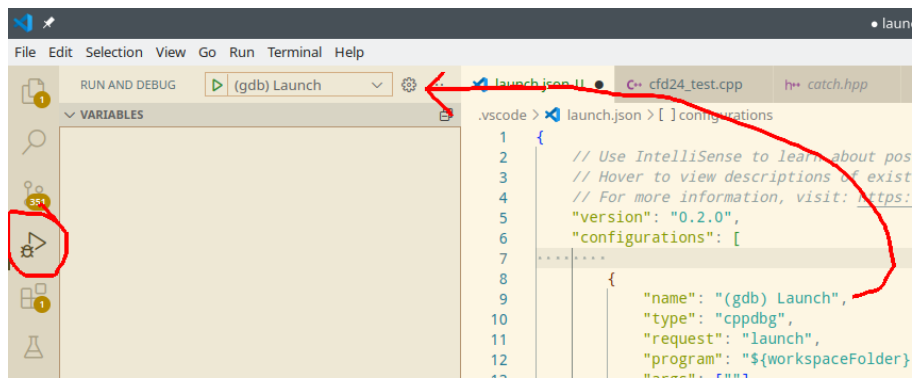


```

5  "version": "0.2.0",
6  "configurations": [
7    {
8      "name": "(gdb) Launch",
9      "type": "cppdbg",
10     "request": "launch",
11     "program": "${workspaceFolder}/build/bin/cfd24_test",
12     "args": [],
13     "stopAtEntry": false,
14     "cwd": "${fileDirname}",
15     "environment": [],
16     "externalConsole": false,
17     "MIMode": "gdb",
18     "setupCommands": [
19       {
20         "description": "Enable pretty-printing for gdb",
21         "text": "-enable-pretty-printing",
22         "ignoreFailures": true
23       },
24       {
25         "description": "Set Disassembly Flavor to Intel",
26         "text": "-gdb-set disassembly-flavor intel",
27         "ignoreFailures": true
28       }
29     ]
30   }
31 ]

```

- Для этого перейдите в меню **Run and Debug** (**Ctrl+Shift+D**), нажмите **create launch.json**, выберите пункт **Node.js**.
- После этого в корневой папке появится файл **.vscode/launch.json**.
- Откройте этот файл в **vscode**, нажмите **Add configuration**, **(gdb) Launch** или **(Windows) Launch** в зависимости от ОС.
- Далее напишите имя программы как показано на картинке.
- Используйте поле **args** для установки аргументов запуска.
- Выберите созданную конфигурацию для запуска отладчика по **F5**



На скриншотах представлены настройки в случае работы в линуксе. Для работы под виндоус

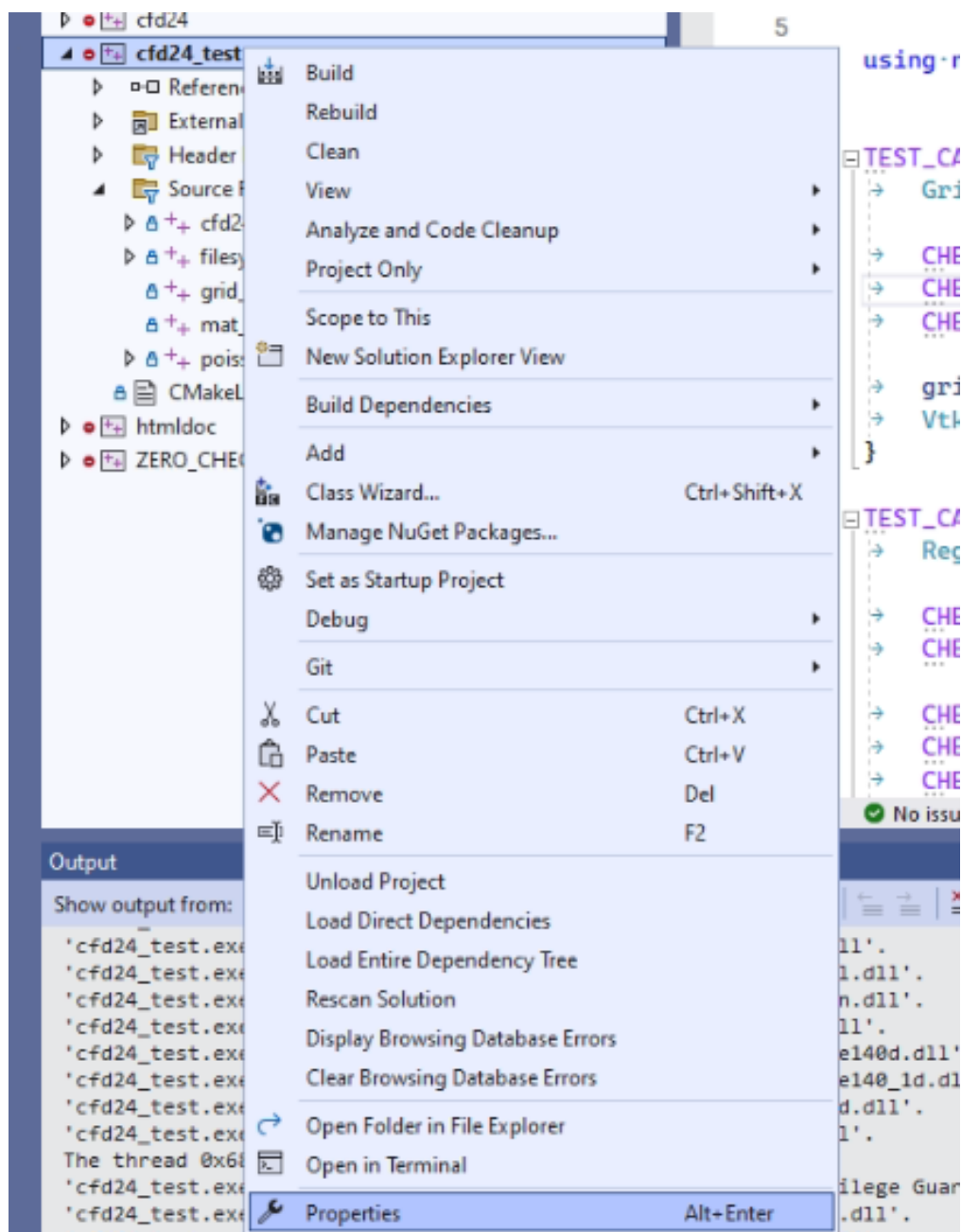
```
"name" : "(Windows) Launch",
"program": "${workspaceFolder}/build/bin/Debug/cfd25_test.exe"
```

В.1.2 Запуск конкретного теста

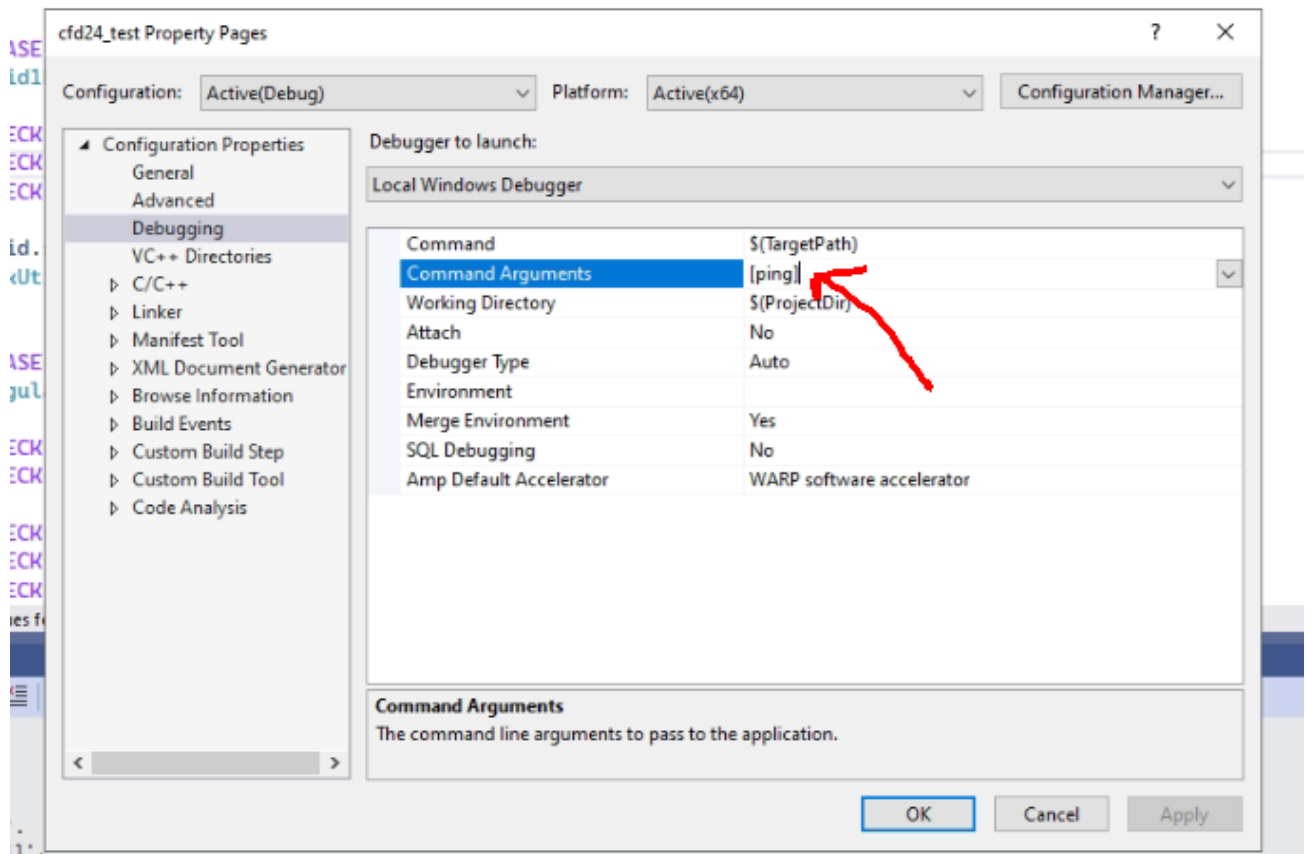
По умолчанию программа `cfd25_test` прогоняет все объявленные в проекте тесты. Иногда может возникнуть необходимость запустить только конкретный тест в целях отладки или проверки. Для этого нужно передать программе аргумент с тегом для этого теста.

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 `cfd25_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

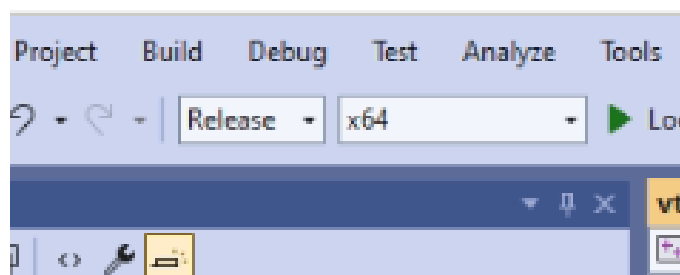
Релизная сборка программ даёт многократное увеличение производительности, но при этом отладка приложений в таком режиме невозможна.

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 ..
```

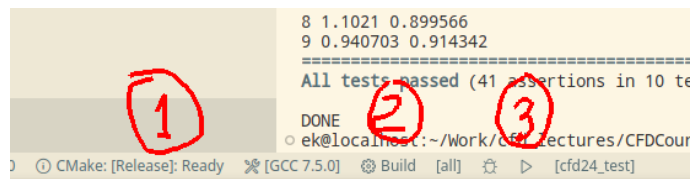
4. Запустить `winbuild64.bat` из новой папки
5. Открыть `build-release/cfdcourse25.sln` в **Visual Studio**
6. В проекте студии установить релизную сборку



7. Это новое решение, не связанное настройками с `debug`-версией. Поэтому нужно заново настроить запускаемый проектом `cfld_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`



В.2 Git

В.2.1 Основные команды

Все команды выполнять в терминале (`git bash` для виндоус), находясь в корневой папке проета 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 нескommиченные изменения. Если необходимо скоммитить изменения в новую ветку, сразу за этой командой нужно вызвать

```
> git add .  
> git commit -m "message"
```

- **Сбросить** все нескommиченные 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.

В.2.2 Порядок работы с репозитием 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** придёт 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

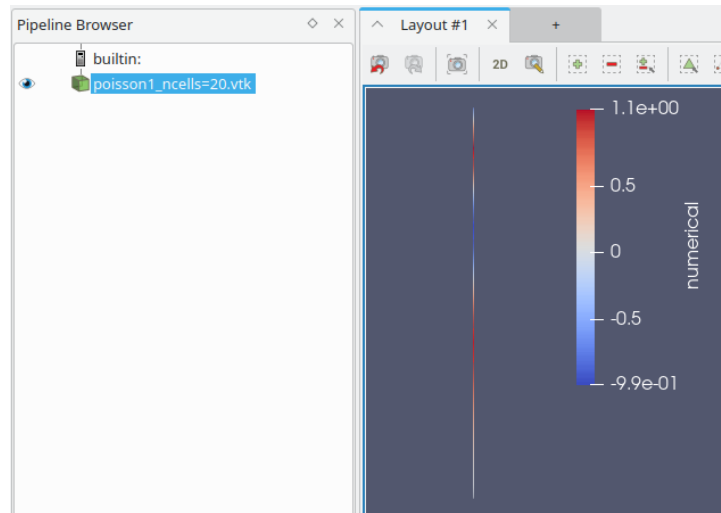
```
> git checkout work-lect{index}
```

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

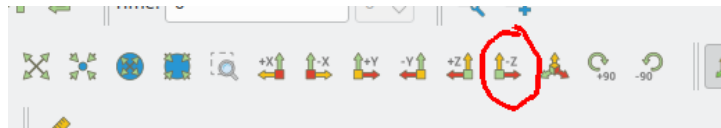
В.3 Paraview

В.3.1 Данные на одномерных сетках

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

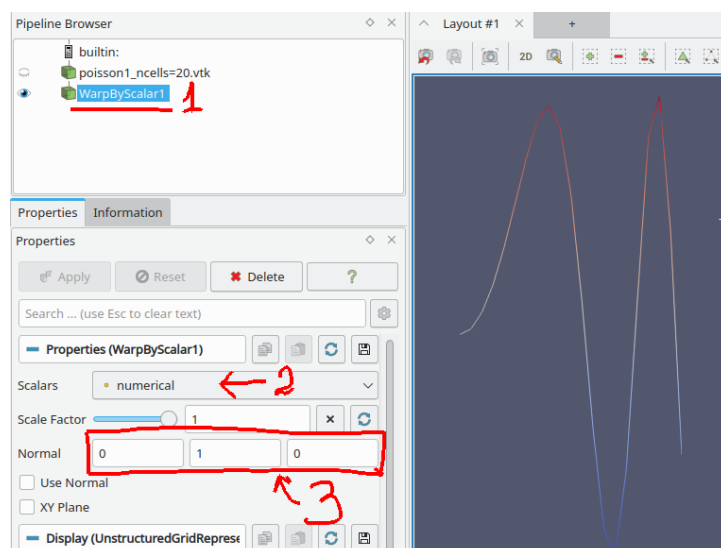


Развернуть изображение в плоскость ху



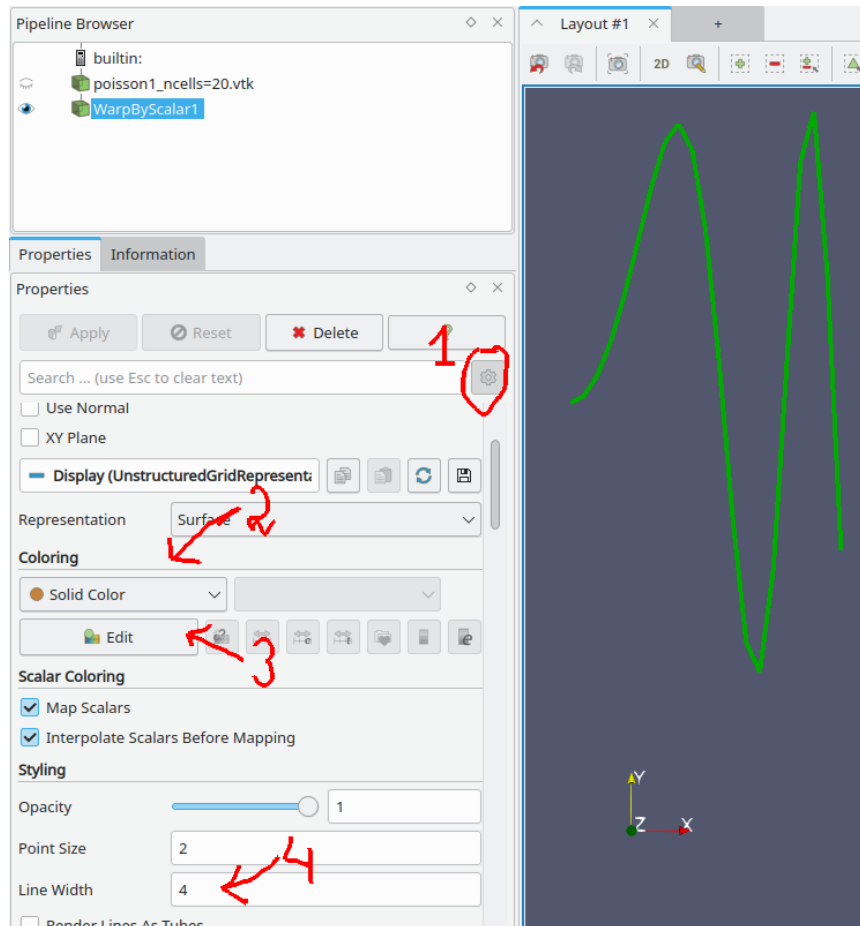
Отобразить данные в виде у-координаты Для того, что бы данные отображались в качестве значения по оси ординат, к загруженному файлу необходимо

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

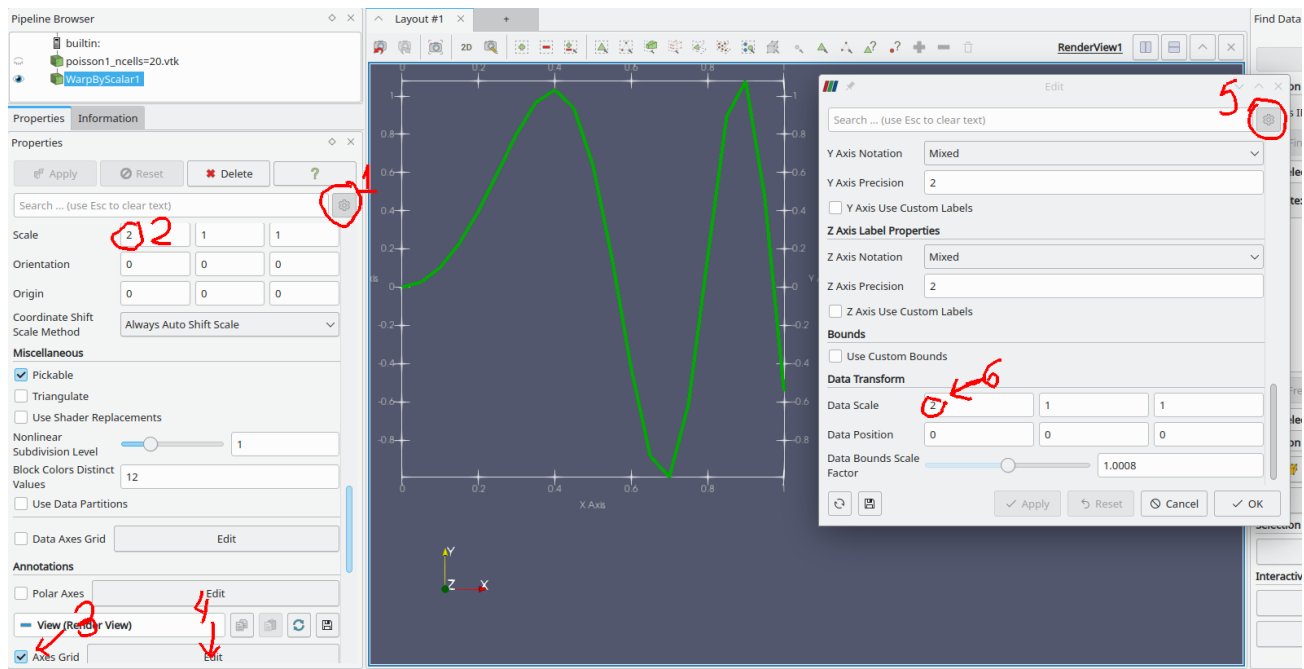
1. Включить подробные опции фильтра
2. Сменить стиль на **Solid Color**
3. В меню **Edit** выбрать желаемый цвет
4. В строке **Line Width** указать толщину линии



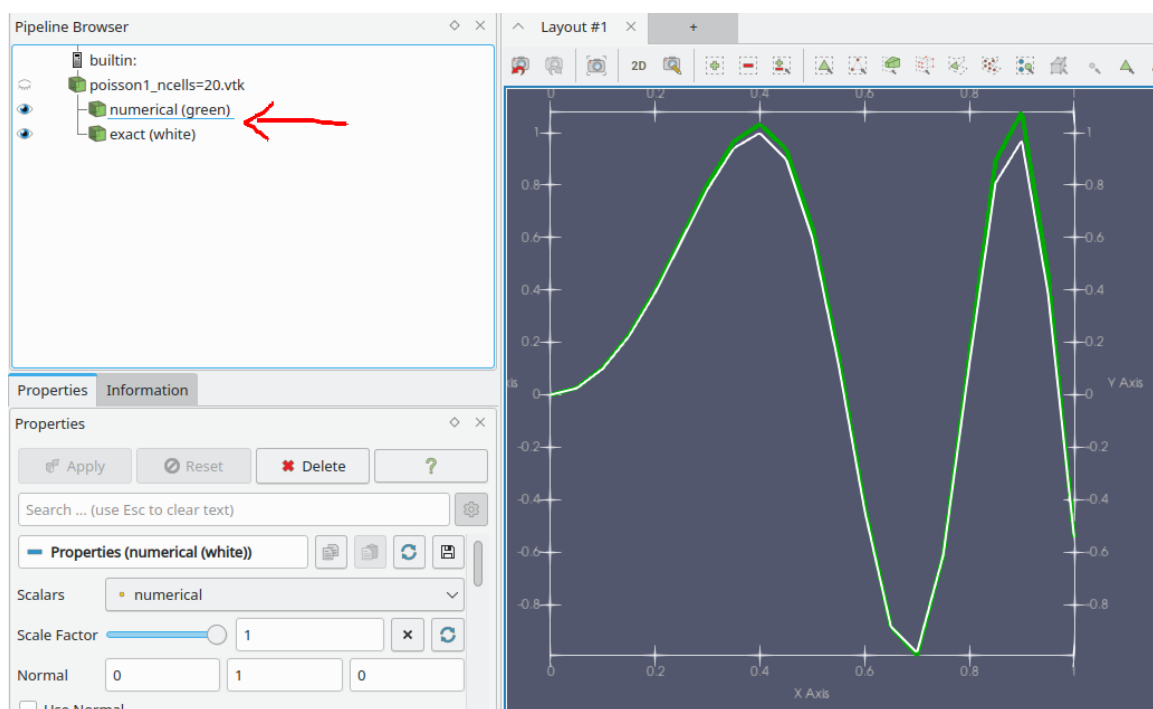
Настройка масштабов и отображение осей координат

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 натройки осей
5. В нём включите подробные настройки
6. И также поставьте растяжение осей

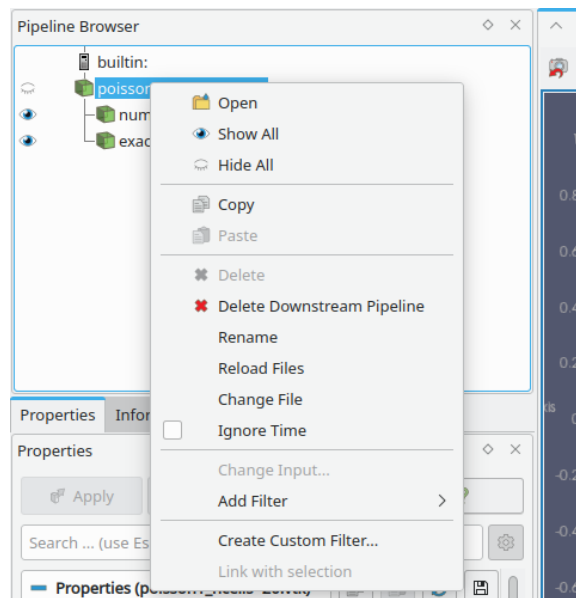
В случае, если масштабировать график не нужно, достаточно выполнить шаг 3.



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

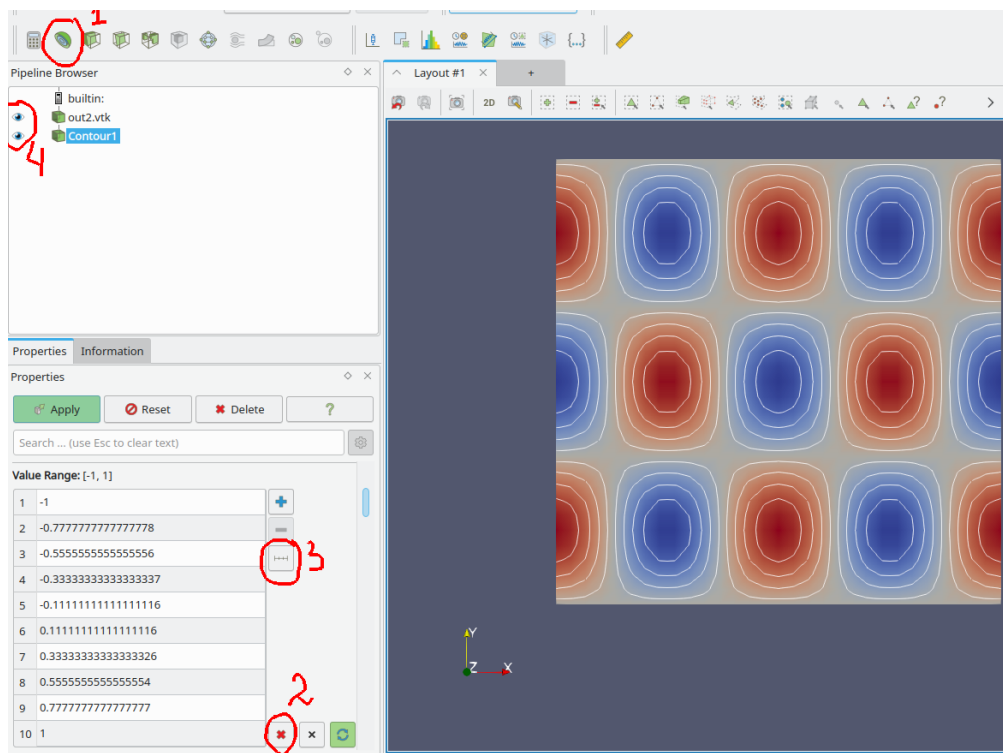


Обновление данных при изменении исходного файла В случае, если исходный файл был изменён, нужно в контекстном меню узла соответствующего файла выбрать **Reload Files** (или нажать F5). Если те же самые фильтры нужно применить для просмотра другого файла нужно в этом меню нажать **Change File**.

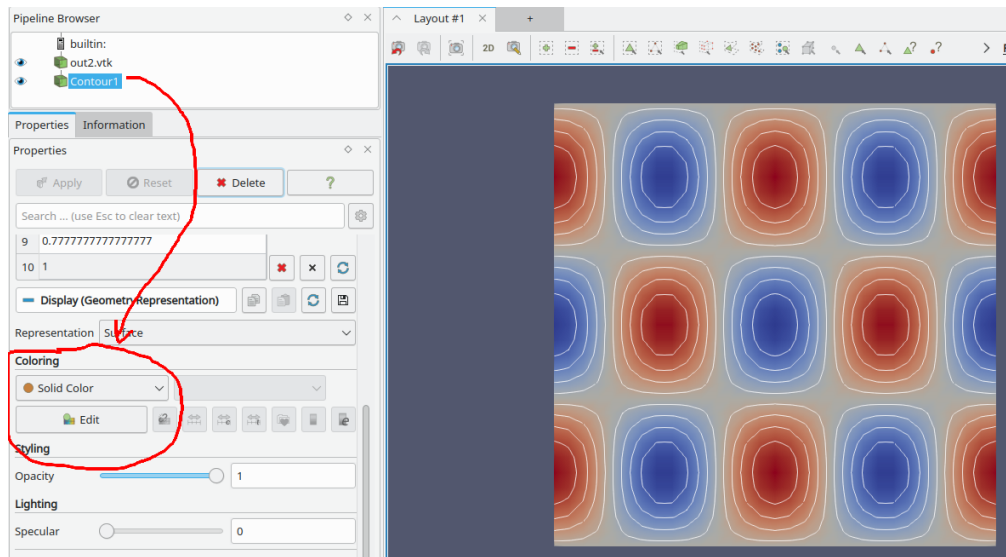


В.3.2 Изолинии для двумерного поля

1. Нажмите иконку **Contour** (или **Filters/Contour**) В настройках фильтра Contour by выберите данные, по которым нужно строить изолинии.
2. В настройках фильтра удалите все существующие записи о значениях для изолиний
3. Добавьте равномерные значения. В появившемся меню установите необходимое количество изолиний и их диапазон.
4. Если необходимо, включите одновременное отображения цветного поля и изолиний.



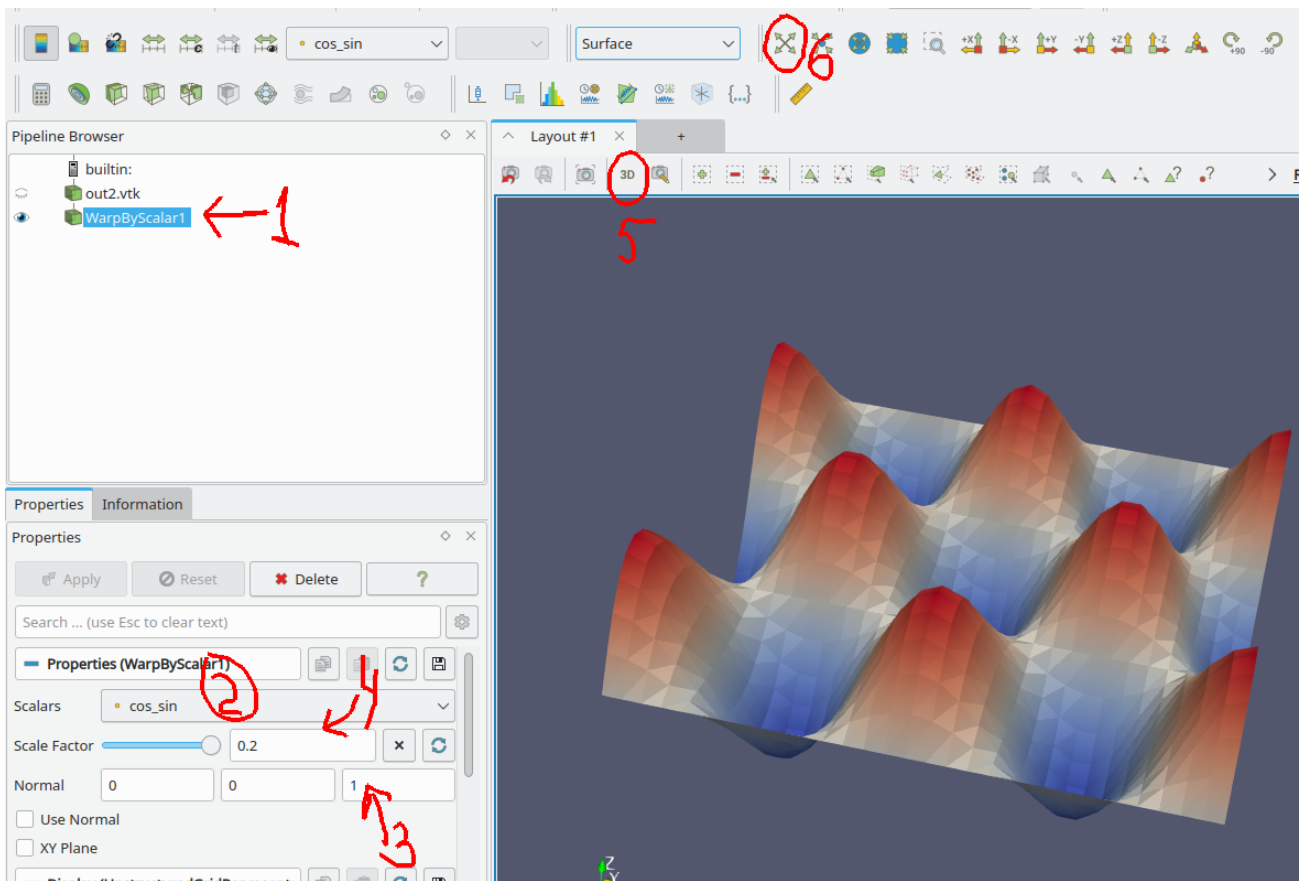
Задание цвета и толщины изолинии В случае, если нужно сделать изолинии одного цвета, установите поле **Coloring/Solid color** в настройках фильтра. Там же в меню **Edit** можно выбрать цвет. Для установления толщины линии включите подробные настройки и найдите там опцию **Styling/Line Width**.



В.3.3 Данные на двумерных сетках в виде поверхности

По аналогии с одномерным графиком (п. В.3.1), двумерные поля так же можно отобразить, проектируя данные на геометрическую координату для получения объёмного графика. Для этого

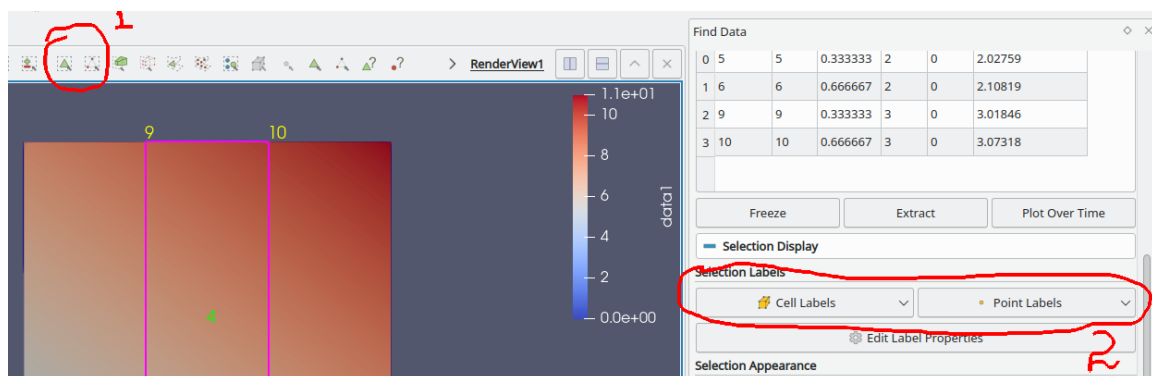
1. Включите фильтр **Filters/Warp By Scalar**
2. В настройках фильтра установите данные, которые будут проектироваться на координату z
3. Установите нормаль для проецирования (ось z)
4. Если нужно, выберите масштабирования для этой координаты
5. После нажатия **Apply** включите трёхмерное отображение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



В.3.4 Числовых значения 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**.

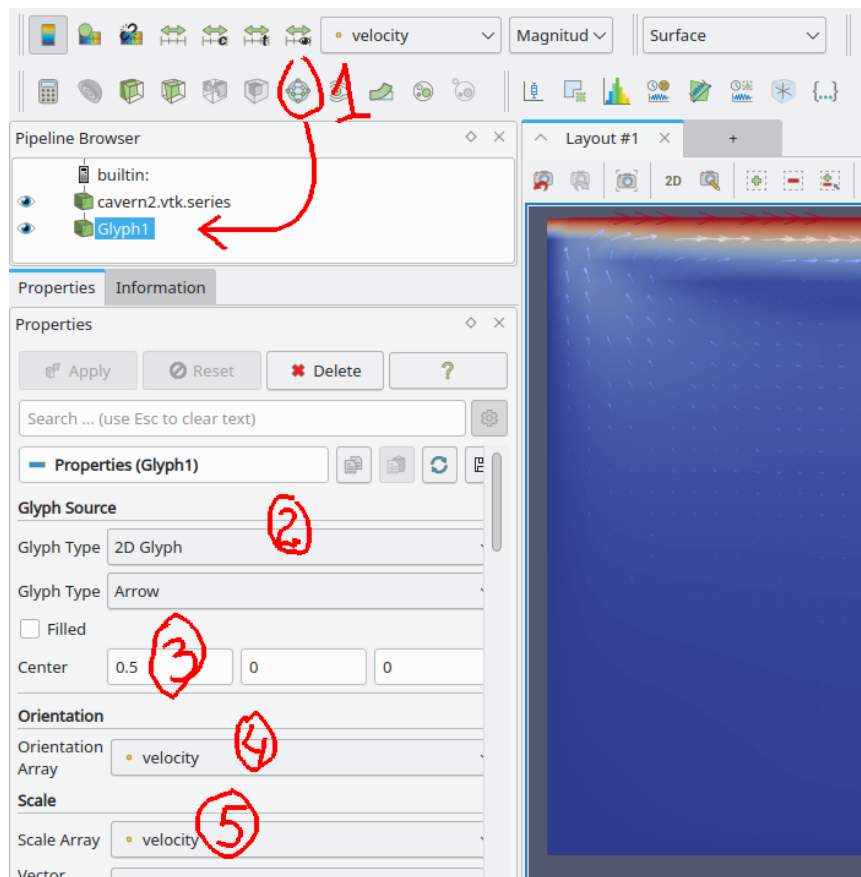


В.3.5 Векторные поля

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

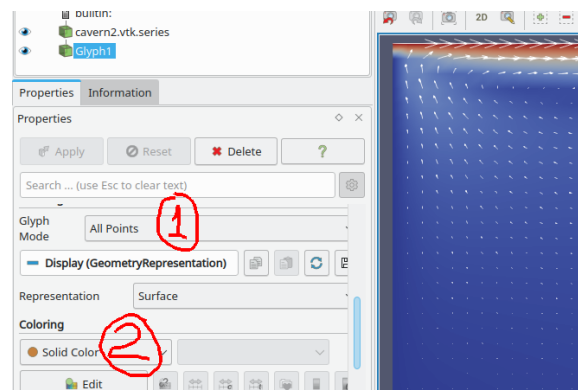
1. Создать фильтр **Glyph**

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**.



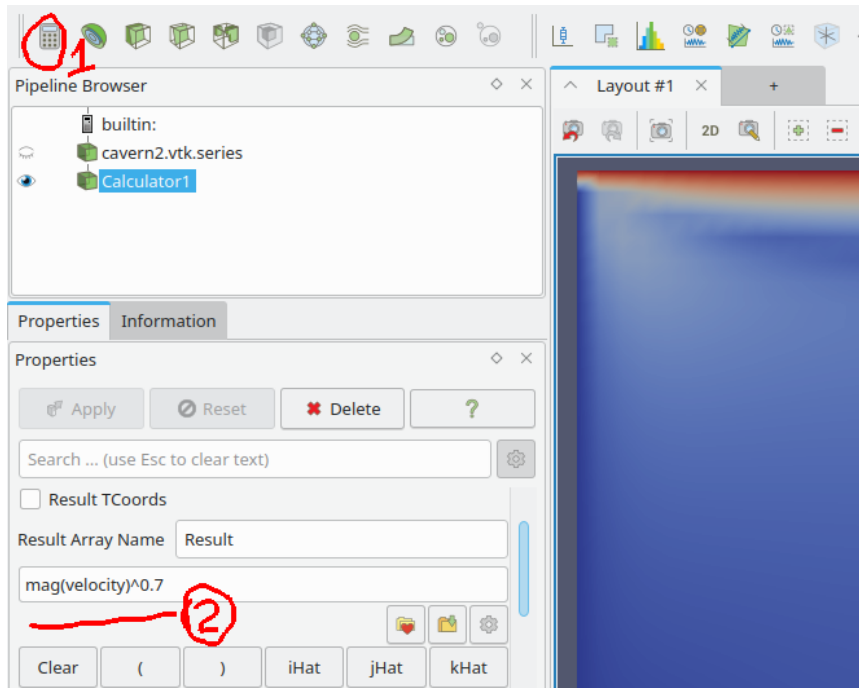
Настройка отображения стрелок

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



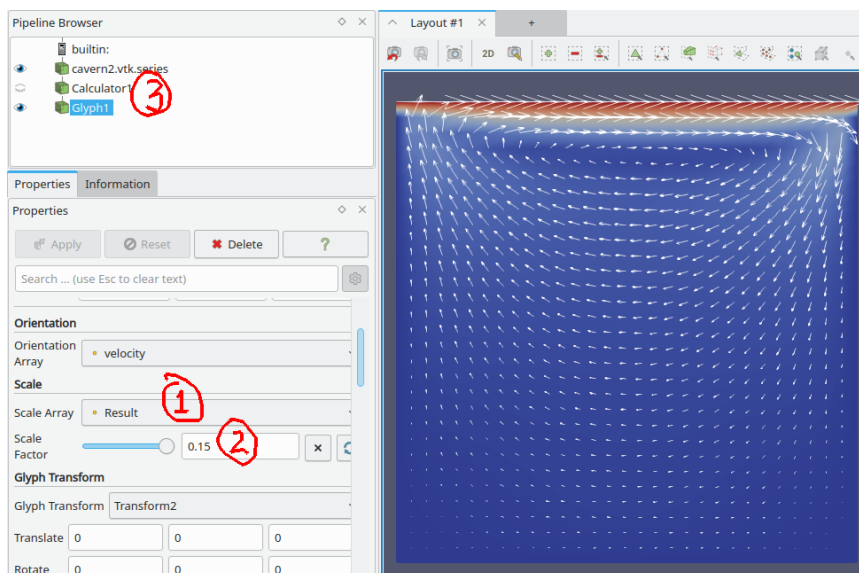
Уменьшения разброса по длине стрелок Если разброс по длинам стрелок слишком велик, его можно подравнять, введя новую функцию $|\mathbf{v}|^\alpha$ – длина вектора в степени меньше единицы (например, $\alpha = 0.7$). Такую функцию можно создать через калькулятор

1. Начиная от загруженного файла создать фильтр **Calculator**
2. Там вбить необходимую формулу



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

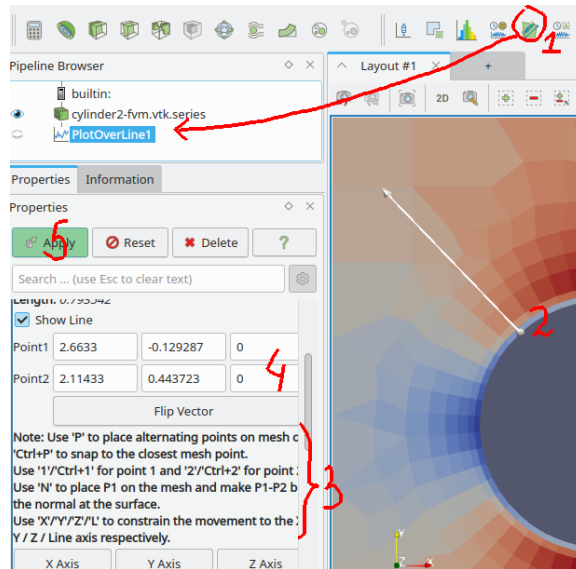
1. В **Scale Array** фильтра **Glyph** указать уже результат работы **Calculator**-а (**Result** по умолчанию),
2. Подтянуть значение **Scale Factor** до приемлимого
3. Не забыть отключить вспомогательное поле **Calculator** из отображения



В.3.6 Значение функции вдоль линии

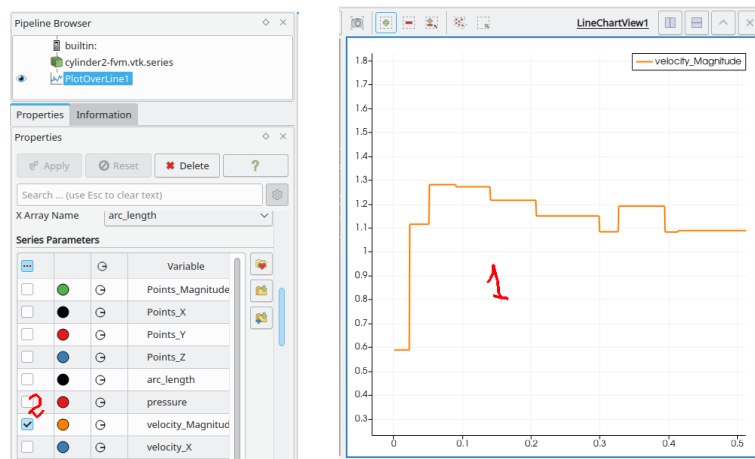
1. Выбрать фильтр **Plot Over Line** иконкой или в меню **Filters**
2. Установить начальную и конечную точку сечения

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**



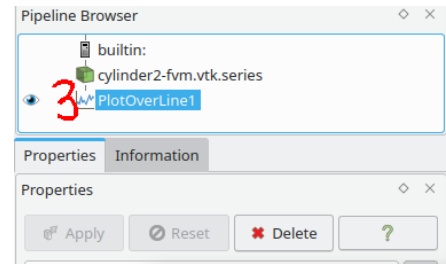
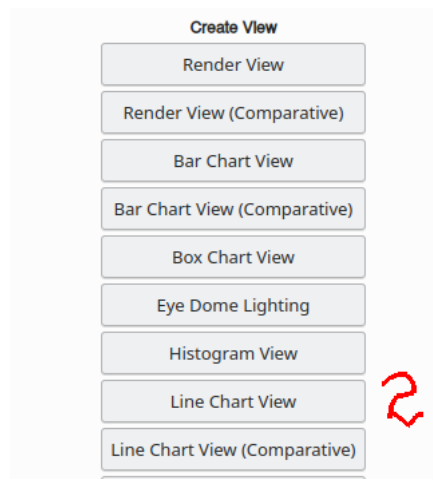
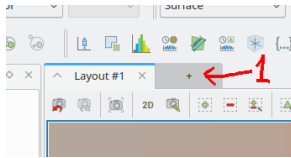
Настройка графика

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



В.4 Hybmesh

Генератор сеток на основе композитного подхода. Работает на основе python-скриптов. Полная документация <http://kalininei.github.io/HybMesh/index.html>

В.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
```

В.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 клонировать 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`, убедиться, что анаконда работает в нужной среде (то есть `conda activate py27` был вызван), и запустить

```
> python /path/to/HybMesh/hybmesh.py script.py
```