Содержание

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 арифмитического 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

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

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

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

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

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

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

$$f_i = f(x_i). \tag{3.4}$$

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

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

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

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

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

Далее путём прямого вычисления определим параметры задачи f, u_a, u_b такие, для которых функция u^e является точным решением задачи (3.1), (3.2).

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

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

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

Упрощая до одномерного случая

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

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

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

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

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

такая что

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

Окончательно среднеквадратичная норма отклонения численного решения от точного запишется в виде

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

3.2.3 Программная реализация

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

В качестве аналитической тестовой функции используется

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

на отрезке $x \in [0, 1]$.

3.2.3.1 Функция верхнего уровня

объявлена как

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

В программе в цикле по набору разбиений `n_cells`

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

создаётся решатель для тестовой задачи, использующий заданное число ячеек

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

вычисляется среднеквадратичная норма отклонения численного решения от точного

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

полученное численное решение (вместе с точным) сохраняется в vtk файле `poisson1_n={10,20,...}.vtk`

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

а полученная норма печатается в консоль напротив количества ячеек

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

В результате работы программы в консоли должна отобразиться таблица вида

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

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

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

3.2.3.2 Детали реализации

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

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

	A	B	C	D
1	N	Norm	Log10(N)	Log10(Norm)
2	10	0.179124	1	-0.74684622
3	20	0.0407822	1.301029996	-1.38952935
4	50	0.00634718	1.698970004	-2.19741919
5	100	0.00158055	2	-2.80119176
6	200	0.000394747	2.301029996	-3.40368116
7	500	6.31E-05	2.698970004	-4.19968098
8	1000	1.58E-05	3	-4.80175817

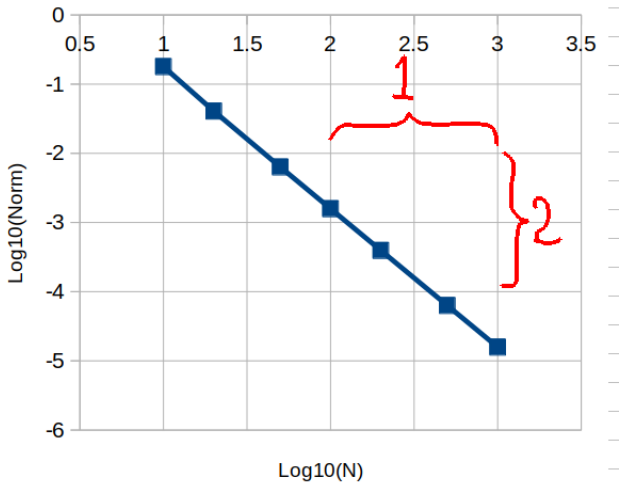


Рис. 1: Сходимость с уменьшением разбиения при решении одномерного уравнения Пуассона

```
15 class TestPoisson1Worker{
```

В методе `solve()` производится чиленное решения 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.

```
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();
42   }
```

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

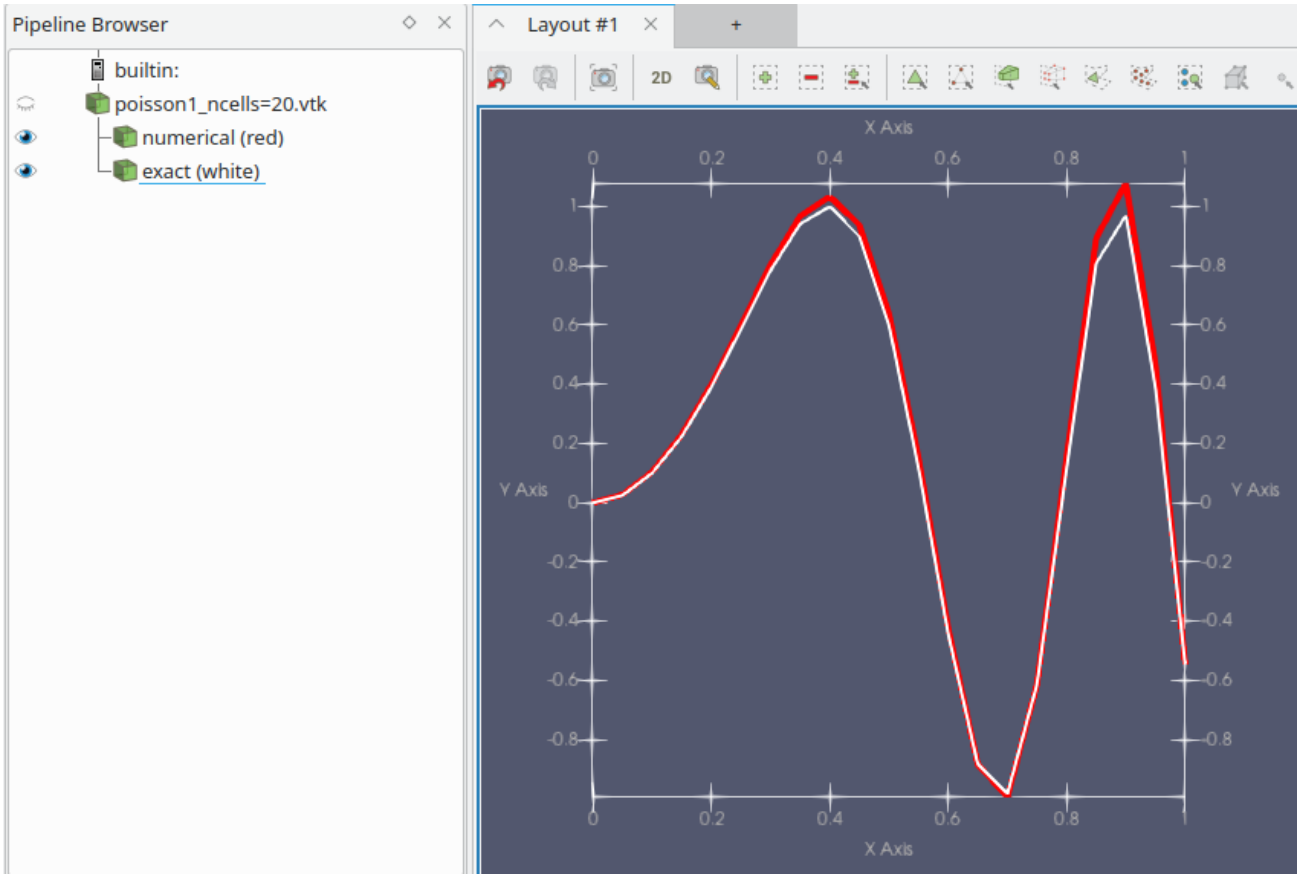


Рис. 2: Сравнение точного и численного решений уравнения Пуассона

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

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

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

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

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

- Вычисление нормы. Реализует формулу (3.9).

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

3.3 Задание для самостоятельной работы

3.3.1 Одномерное уравнение Пуассона

Скомпиллировать 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

$$k[i,j]=i+(n_x+1)j \tag{3.10}$$

– функция, переводящая парный (i,j) индекс узла регурярной 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:

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

Четрые угловые точки определяются как

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

Граничные неугловые точки:

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

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

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

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

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

В случае, если решатель системы линейных уравнений не решает построенную матрицу, использовать функцию `cfd::dbg::print` для отлаточной 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 алогритмы 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

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

Массивы, описывающие матрицу в формате CSR примут вид

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

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;
```

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

5.1.2 Метод Зейделя

Формулируется в виде

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

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

for $i = \overline{0, N-1}$

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

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 измениает решение по тому же принципу, что и метод Зейделя, но искуственно 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 $i = \overline{0, N-1}$

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

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 использоватть алгоритмы работы с CSR-матрицами из п. 4.1.1

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

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

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

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

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

...

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

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

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

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

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

По результатам профилировки нужно заполнить таблицу

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

Здесь Amg - исходный решатель.

6 Лекция 6 (23.03)

6.1 Метод конечных объёмов

6.1.1 Уравнение Пуассона

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

$$-\nabla^2 u = f, \tag{6.1}$$

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

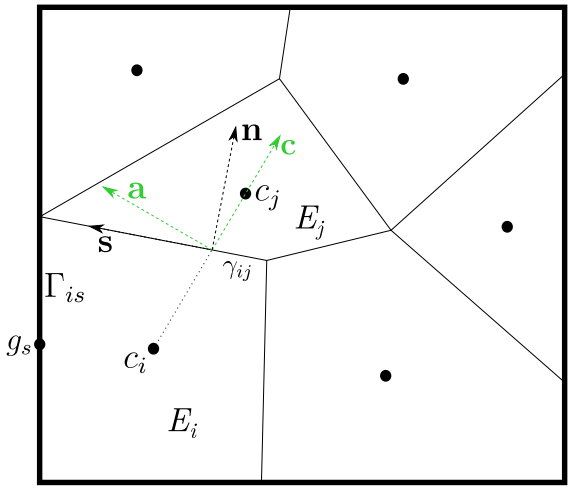


Рис. 3: Конечнообъёмная сетка

Проинтегрируем исходное уравнение по одной из подобластей E_i:

$$-\int\limits_{E_i} \nabla^2 u \, ds = \int\limits_{E_i} f \, d\mathbf{x}.$$

К интегралу в левой части применим формулу интегрирования по частям (A.13). Получим

$$-\int\limits_{\partial E_i} \frac{\partial u}{\partial n} \, ds = \int\limits_{E_i} f \, d\mathbf{x}. \tag{6.2}$$

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

$$\int\limits_{\partial E_i} \frac{\partial u}{\partial n} \, ds = \sum_j \int\limits_{\gamma_{ij}} \frac{\partial u}{\partial n} \, ds + \sum_s \int\limits_{\Gamma_{is}} \frac{\partial u}{\partial n} \, ds.$$

Аппроксимирум производную $\partial u/\partial n$ на каждой из граней константой. Тогда её можно вынести из под интегралов и предыдущее выражение записать в виде

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

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

$$\int\limits_{E_i} f\, d\mathbf{x} \approx f_i\, |E_i|\,. \tag{6.4}$$

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):

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

6.1.1.1 Обработка внутренних граней

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

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

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

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

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

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

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

6.1.1.2 Учёт граничных условий первого рода

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

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

Это делается с помощью граничных условий.

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

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

Аппроксимацию производных будем проводить из тех же соображений, которые использовали при анализе внутренних граней. Только вместо центра соседнего элемента c_j будем использовать центр грани g_s. В первом приближении, отбрасывая касательные производные, придём к формуле аналогичной (6.5):

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

6.1.2 Одномерный случай

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

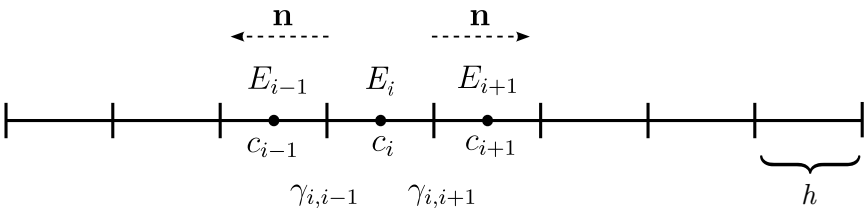


Рис. 4: Одномерная конечнообъёмная сетка

У внутренней ячейки i есть две 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 (6.7) примет вид

$$-\frac{u^\Gamma - u_0}{h/2} - \frac{u_1 - u_0}{h} = f_0 h.$$

В методе конечных разностей это условие выразилось бы в виде $u_0 = u^\Gamma$.

6.1.3 Сборка системы линейных уравнений

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

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

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:

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

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

$$Au = b.$$

6.1.3.1 Алгоритм сборки в цикле по ячейкам

Матрицу A и правую часть b системы (6.8) можно собирать в цикле по ячейкам: строчка за строчкой. Такой алгоритм выглядел бы следующим образом

for $i = \overline{0, N-1}$ – цикл по строкам СЛАУ
  $b_i = |E_i| f_i$
  for $j \in$ nei(i) – цикл по ячейкам, соседним с ячейкой i
    $v = |\gamma_{ij}|/h_{ij}$
    $A_{ii} \mathrel{+}= v$
    $A_{ij} \mathrel{-}= v$
  endfor
  for $s \in$ bnd1(i) – цикл по граням ячейки i с условиями первого рода
    $v = |\Gamma_{is}|/h_{is}$
    $A_{ii} \mathrel{+}= v$
    $b_i \mathrel{+}= u^\Gamma v$
  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

$$\begin{aligned}
&\textbf{for } s \in \text{internal} && \text{– цикл по внутренним граням}\\
&\quad i, j = \text{nei_cells(s)} && \text{– две ячейки, соседние с текущей гранью}\\
&\quad v = |\gamma_{ij}|/h_{ij}\\
&\quad A_{ii}{+}{=}v; \quad A_{jj}{+}{=}v && \text{– диагональные коэффициенты матрицы}\\
&\quad A_{ij}{-}{=}v; \quad A_{ji}{-}{=}v && \text{– внедиагональные коэффициенты матрицы}\\
&\textbf{endfor}
\end{aligned} \tag{6.9}$$

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

$$\begin{aligned}
&\textbf{for } s \in \text{bnd1} && \text{– грани с условиями первого рода}\\
&\quad i = \text{nei_cells(s)} && \text{– соседняя с граничной гранью ячейка}\\
&\quad v = |\Gamma_{is}|/h_{is}\\
&\quad A_{ii}{+}{=}v\\
&\quad b_i{+}{=}u^{\Gamma}v\\
&\textbf{endfor}
\end{aligned} \tag{6.10}$$

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

$$\begin{aligned}
&\textbf{for } i = \overline{0, N-1} && \text{– цикл по ячейкам}\\
&\quad b_i = |E_i| f_i\\
&\textbf{endfor}
\end{aligned} \tag{6.11}$$

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 расчитываются 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. B.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.

7 Лекция 7 (30.03)

7.1 Граничные условия второго рода

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

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

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

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

$$\begin{array}{ll} \textbf{for } s \in \text{bnd2} & \text{– грани с условиями второго рода} \\ \quad i = \text{nei\_cells(s)} & \text{– соседняя с граничной гранью ячейка} \\ \quad b_i \mathrel{+}= |\Gamma_{is}| q & \\ \textbf{endfor} & \end{array} \tag{7.2}$$

7.2 Граничные условия третьего рода

Теперь рассмотрим условия третьего рода

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

Распишем производную в форме (6.7):

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

откуда выразим u^Γ:

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

Подставляя это выражение в исходное граничное условие (7.3) получим

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

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

$$\begin{array}{ll} \textbf{for } s \in \text{bnd3} & \text{– грани с условиями третьего рода} \\ \quad i = \text{nei\_cells(s)} & \text{– соседняя с граничной гранью ячейка} \\ \quad v = |\Gamma_{is}| / (1 + \alpha h_{is}) & \\ \quad A_{ii} \mathrel{-}= \alpha v & \\ \quad b_i \mathrel{+}= \beta v & \\ \textbf{endfor} & \end{array} \tag{7.5}$$

7.2.1 Универсальность условий третьего рода

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

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

где ε – малое положительное число.

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

7.3 Задание для самостоятельной работы

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

По формуле

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

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

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

8 Лекция 8 (06.04)

8.1 Дополнительные точки коллокации на границах

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

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

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

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

for $s \in$ bnd – граничные грани
  $i =$ nei_cells(s) – соседняя с граничной гранью ячейка
  $j =$ bnd_col(s) – индекс точки коллокации, соответствующей грани
  $v = |\Gamma_{is}|/h_{is}$
  $A_{ii} \mathrel{+}= v$
  $A_{ij} \mathrel{-}= v$
endfor (8.1)

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

for $s \in$ bnd1 – грани с условиями первого рода
  $j =$ bnd_col(s) – индекс точки коллокации, соответствующей грани
  $A_{jj} = 1$
  $b_j = u^{\Gamma}$
endfor (8.2)

Для условий третьего рода (7.3) примем аппроксимацию

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

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

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

$$
\begin{aligned}
&\textbf{for } s \in \text{bnd3} && \text{– грани с условиями третьего рода}\\
&\quad i = \text{nei\_cells(s)} && \text{– соседняя с граничной гранью ячейка}\\
&\quad j = \text{bnd\_col(s)} && \text{– индекс точки коллокации, соответствующей грани}\\
&\quad A_{jj} = 1/h_{is} - \alpha\\
&\quad A_{ji} = -1/h_{is}\\
&\quad b_j = \beta\\
&\textbf{endfor}
\end{aligned}
\tag{8.3}
$$

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

Преимуществами такого подхода является:

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

8.1.1 Пример

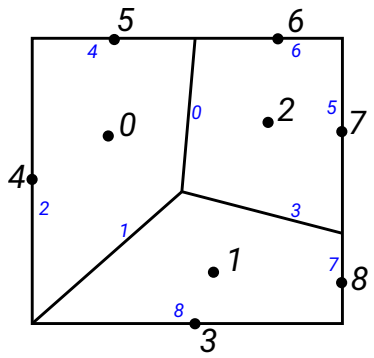


Рис. 5: Расширенный набор точек коллокации

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

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

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

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

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

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

Справа в 1-ой строке будет стоять

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

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

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

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

$$u_3 = C,$$

а уравнение 8 – условию третьего рода на грани γ_7:

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

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

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

для 3-ей строки

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

для 8-ой строки

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

8.2 Задание для самостоятельной работы

1. Решить задачу из п. 6.2, с истинными граничными условиями первого рода, и с граничными условиями первого рода, поставленными через условия третьего рода.

2. Убедится, что полученный ответ с точностью то ошибки решения СЛАУ (10^{-8}) совпадает с результатами, полученным в двух предыдущих заданиях.

3. Для постановки с условиями третьего рода построить график максимального отклонения граничного значения полученного сеточной функции $\{u\}$ от точного решения в зависимости от

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

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

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

Рекомендации к программированию

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

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

9 Лекция 9 (13.04)

9.1 Учёт скошенности сетки в двумерной МКО-аппроксимации

9.1.1 Уточнённая аппроксимация нормальной производной

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

Для двумерного случая распишем градиент u в системе координат, образованной еодиничными векторами нормали $\mathbf{n}$ и касательной $\mathbf{s} = (-n_y, n_x)$ к грани γ_{ij} (см. рисунок рис. 3):

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

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

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

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

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

Отсюда выразим искомую производную

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

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

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

Тогда получим аппроксимацию второго порядка искомой производной:

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

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

9.1.2 Интерполяция значения функции во вспомогательных точках

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

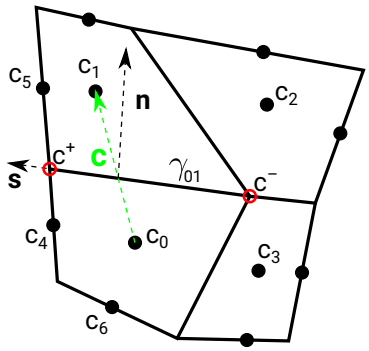


Рис. 6: Пример расположения вспомогательных точек (показаны красными кругами) для аппроксимации производной по грани

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

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

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

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

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

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

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

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

Коэффициенты которой равны

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

9.1.3 Производная по границе

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

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

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

тогда j-ое уравнение СЛАУ будет иметь вид

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

9.1.4 Сборка СЛАУ для уравнения Пуассона

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

ки, соответствующие центрам ячеек.

for $s \in$ faces — цикл по всем граням
  $i, j, k, s =$ dn_cells(s) — индексы точек коллокации для аппроксимации $\partial u/\partial n$
  $v_i, v_j, v_k, v_s =$ dn_coefs(s) — коэф-ты линейной формы для аппроксимации $\partial u/\partial n$
  $a =$ face_area(s) — площадь грани
  if i is cell center — если i – коллокация для центра ячейки
    $A_{ii} -= a\, v_i;$ — i-ая строка матрицы (против нормали к грани)
    $A_{ij} -= a\, v_j;$
    $A_{ik} -= a\, v_k;$
    $A_{is} -= a\, v_s;$ (9.5)
  endif
  if j is cell center — если j – коллокация для центра ячейки
    $A_{ji} += a\, v_i;$ — j-ая строка матрицы (по нормали к грани)
    $A_{jj} += a\, v_j;$
    $A_{jk} += a\, v_k;$
    $A_{js} += a\, v_s;$
  endif
endfor

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

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

9.2 Задание для самостоятельной работы

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

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

Объект этого класс необходимо собрать на этапе инициализации задачи. Далее линейные формы для заданной грани вычисляются вызовом метода `FvmLinformFacesDn::linear_combination(size_t iface)`. Этот метод возвращает упорядоченную линейную комбинацию из четырех (для двумерной задачи) слагаемых. Получение индексов и коэффициентов линейной формы согласно алгоритму (9.5) осуществляется следующим образом

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

10 Лекция 10 (20.04)

10.1 Учёт скошенности сетки в 3D

11 Лекция 11 (27.04)

11.1 Метод взвешенных невязок

11.2 Метод Бубнова–Галёркина

11.2.1 Степенные базисные функции

11.3 Метод конечных элементов

11.3.1 Узловые базисные функции

11.3.2 Одномерное уравнение Пуассона

11.3.2.1 Слабая интегральная постановка задачи

11.3.2.2 Линейный одномерный (пирамидальный) базис

12 Лекция 12 (04.05)

12.1 Поэлементная сборка конечноэлементных матриц

12.1.0.1 Элементные матрицы

Матрица масс

$$m_{ij}^k = \int\limits_{E^k} \phi_i(\mathbf{x})\phi_j(\mathbf{x})\, d\mathbf{x} \tag{12.1}$$

Вектор нагрузок

$$l_i^k = \int\limits_{E^k} \phi_i(\mathbf{x})\, d\mathbf{x} \tag{12.2}$$

Матрица жёсткости

$$s_{ij}^k = \int\limits_{E^k} \nabla\phi_i \cdot \nabla\phi_j\, d\mathbf{x} \tag{12.3}$$

13 Лекция 13 (11.05)

13.1 Вычисление элементных интегралов в параметрическом пространстве

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

$$m_{ij}^k = \int\limits_{\tilde{E}^k} \tilde{\phi}_i^{(k)}(\boldsymbol{\xi})\tilde{\phi}_j^{(k)}(\boldsymbol{\xi})|J^{(k)}(\boldsymbol{\xi})|\, d\boldsymbol{\xi} \tag{13.1}$$

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

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

Локальный вектор нагрузок

$$l_i^k = \int\limits_{\tilde{E}^k} \tilde{\phi}_i^{(k)}(\boldsymbol{\xi})|J^{(k)}(\boldsymbol{\xi})|\, d\boldsymbol{\xi} \tag{13.2}$$

Локальная матрица жёсткости

$$s_{ij}^k = \int\limits_{\tilde{E}^k} \nabla_{\mathbf{x}}\tilde{\phi}_i^{(k)} \cdot \nabla_{\mathbf{x}}\tilde{\phi}_j^{(k)}|J^{(k)}(\boldsymbol{\xi})|\, d\boldsymbol{\xi} \tag{13.3}$$

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

13.2 Двумерное уравнение Пуассона

13.2.0.1 Треугольный элемент. Линейный двумерный базис

Матрица масс (из (13.1)):

$$M_{ij}^E = \int\limits_0^1 \int\limits_0^{1-\xi} \phi_i(\xi,\eta)\phi_j(\xi,\eta)|J|\, d\eta d\xi = \frac{|J|}{24}\begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix} \tag{13.4}$$

13.3 Разбор программной реализации МКЭ

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

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

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

и формулируется уравнение Пуассона с граничными условиями первого рода, для которого эта функция является точным решением. Далее уравнение Пуассона решается численно и полученный численный результат сравнивается с точным ответом. Норма полученной ошибки печатается в консоль.

В функции верхнего уровня происходит построение одномерной сетки, создание рабочего объекта, вызов решения с возвращением нормы полученной ошибки и вывод данныех (сохранение решения в vtk-файл и печать нормы в консоль):

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

Основная работа происходит в классе `TestPoissonLinearSegmentWorker`.

13.3.1 Рабочий объект

Класс `TestPoissonLinearSegmentWorker` наследуется от `ITestPoisson1FemWorker`. В этом классе сформулированы аналитические функции, служащие правой частью, точным решением и условиями первого рода уравнения Пуассона. А этот класс в свою очередь наследуется от `ITestPoissonFemWorker`, в котором и происходит решение уравнения.

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

Для получения решения сначала собирается левая и правая часть системы линеных уравнений, потом происходит учёт граничных условий первого рода , вызывается решатель системы уравнений и вычислитель нормы ошибки.

Функция сборки матрицы левой части реализует сборку глобальной матрицы жёсткости через набор локальных матриц

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

Основой для сборки служит специальный объёкт `_fem` класса `FemAssembler` – сборщик. Этот объёкт сначала используется для задания шаблона итоговой матрицы, потом в цикле по элементам вычисляются локальные матрицы и с помощью метода этого класса `FemAssembler::add_to_global_matrix` локальные матрицы добавляются в глобальную.

По аналогичной процедуре работает и сборка правой части `approximate_rhs`.

13.3.2 Конечноэлементный сборщик

Конечноэлементный сборщик `FemAssembler` – основной класс, хранящий всю информацию о текущей конечноэлементной аппроксимации: массив конечных элементов и их связность. Эта информация подаётся ему при конструировании (реализация в файле `cfd/fem/fem_assembler.hpp`).

```
12   FemAssembler(size_t n_bases,
13                const std::vector<FemElement>& elements,
14                const std::vector<std::vector<size_t>>& tab_elem_basis);
```

Связность `tab_elem_basis` имеет формат “элемент-глобальный базис” и определяет глобальный индекс для каждого локального базисного индекса. В рассматренных нами узловых конечных элементах базис связан с узлом сетки. То есть эта таблица – это связность локальной и глобальной нумерации узлов сетки для каждой ячейки сетки.

Конечноэлементный сборщик создаётся в методе `TestPoissonLinearSegmentWorker::build_fem` итогового рабочего класса (то есть сборщик специфичен для конкретной сетки и конкретного выбора типов элементов). Далее он пробрасывается в конструктор базового рабочего класса.

13.3.3 Концепция конечного элемента

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

```
175 struct FemElement{
176   std::shared_ptr<const IElementGeometry> geometry;
177   std::shared_ptr<const IElementBasis> basis;
178   std::shared_ptr<const IElementIntegrals> integrals;
179 };
```

Главная задача объекта этого класса – вычисление элементных матриц, которые впоследствии используются сборщиком для создания глобальных матриц. Для расчёта элементных матриц в свою очередь требуется

- Геометрия элемента, включающая в себя правило отображения элемента из физической в параметрическую область,
- Набор локальных базисных функций, заданных в параметрическом пространстве на указанной геометрии,
- Непосредственно правило интегрирования в параметрической области.

Каждый из этих трёх алгоритмов определён через интерфейсы

- `IElementGeometry`
- `IElementBasis`
- `IElementIntegrals`

Определение конечного элемента заключается в задании конкретных реализаций этих интерфейсов.

13.3.3.1 Определение линейного одномерного элемента

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

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

Здесь последовательно определяются:

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

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

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

13.3.3.2 Геометрические свойства элемента

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

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

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

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

13.3.3.3 Элементный базис

Интерфейс для определения локального элементного базиса имеет вид

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

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

Этот интерфейс работает только с параметрическим пространстсвом и определяет следующие методы:

- `size` – количество базисных функций;
- `parametric_reference_points` – вектор из параметрических координанат точек, приписанных к соответствующим базисам;
- `value` – значение базисных функций в заданной точке;
- `grad` – градиент (в параметрическом пространстве) базисных функций по заданным точкам.
- `basis_type` – тип базисной функции. До сих пор мы имели дело только с узловыми (`BasisType::Nodal`) функциями.
- `upper_hessian` – верхняя часть матрицы Гессе (вторые производные базисных функций в заданной точке)

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

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

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

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

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

13.3.3.4 Калькулятор элементных матриц

Интерфейс `IElementIntegrals` предоставляет методы для вычисления элементных матриц. До сих пор были рассмотрены две элементные матрицы: матрица масс (13.1) и матрица жёсткости (13.3). Для вычисления этих матриц используются функции
`mass_matrix`, `stiff_matrix`. Возвращают эти функции локальные квадратные матрицы с числом строк, равным количеству базисов в элементе. Выходные матрицы развёрнуты в линейный массив. Так, для треугольного элемента с тремя базисными функциями на выходе будет массив из девяти элементов:

$$m_{00}, m_{01}, m_{02}, m_{10}, m_{11}, m_{12}, m_{20}, m_{21}, m_{22}.$$

Два подхода к вычислению элементных интегралов: точное и численное интегрирование, отражены в разных реализациях этого интерфейса.

До сих пор мы рассматривали только точное вычисление. Точные формулы интегрирования зависят от вида элемента. Так, для линейного одномерного элемента аналитическое интегрирование реализовано в классе `SegmentLinearIntegrals` в файле
`cfd/fem/elem1d/segment_linear.hpp`, а для линейного треугольного элемента –
`TriangleLinearIntegrals` в файле `cfd/fem/elem2d/triangle_linear.hpp` Все интегралы для этих симлексных элементов будут зависеть только от матрицы Якоби, которая и передётся этому классу в конструктор. Например, вычисление матрицы масс (по (13.4)) запрограммировано в виде

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

13.4 Задание для самостоятельной работы

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

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

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

Рекомендации к программированию Для реализации двумерного решения следует по аналогии с одномерным написать класс
`ITestPoisson2FemWorker`, в котором реализовать двумерные функции точного решения и правой части, и наследуемый от него рабочий класс `TestPoissonLinearSegmentWorker`, в котором реализовать статическую функцию `build_fem`. При реализации последней использовать алгоритмы для треугольников.

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

A Формулы и обозначения

A.1 Векторы

A.1.1 Обозначение

Геометрические вектора обозначаются жирным шрифтом $\mathbf{v}$. Скалярные координаты вектора – через нижний индекс с обозначением оси координат: (v_x, v_y, v_z). Если вектор $\mathbf{u}$ – вектор скорости, то его декартовые координаты имеют специальное обозначение $\mathbf{u} = (u, v, w)$. Единичные вектора, соответствующие осям координат, обозначаются знаком $\hat{\cdot}$: $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$, $\hat{\mathbf{z}}$. Координатные векторы обозначаются по символу первой оси. Например, $\mathbf{x} = (x, y, z)$ или $\boldsymbol{\xi} = (\xi, \eta, \zeta)$.

Операции в векторами имеют следующее обозначение (расписывая в декартовых координатах):

- Умножение на скалярную функцию

$$f\mathbf{u} = (fu_x)\hat{\mathbf{x}} + (fu_y)\hat{\mathbf{y}} + (fu_z)\hat{\mathbf{z}}; \tag{A.1}$$

- Скалярное произведение

$$\mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y + u_z v_z; \tag{A.2}$$

- Векторное произведение

$$\mathbf{u} \times \mathbf{v} = \begin{vmatrix} \hat{\mathbf{x}} & \hat{\mathbf{y}} & \hat{\mathbf{z}} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix} = (u_y v_z - u_z v_y)\,\hat{\mathbf{x}} - (u_x v_z - u_z v_x)\,\hat{\mathbf{y}} + (u_x v_y - u_y v_x)\,\hat{\mathbf{z}}. \tag{A.3}$$

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

$$2D: \ \mathbf{u} \times \mathbf{v} = u_x v_y - u_y v_x. \tag{A.4}$$

Геометрический смысл этого скаляра: площадь параллелограмма, построенного на векторах $\mathbf{u}$ и $\mathbf{v}$.

A.1.2 Набла–нотация

Символ ∇ – есть псевдовектор, который выражает покоординатные производные. Для декартовой системы координат (x, y, z) он запишется в виде

$$\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right).$$

В радиальной (r, ϕ, z):

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

В цилиндрической (r, θ, ϕ):

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

Удобство записи дифференциальных выражений с использованием ∇ заключается в независимости записи от вида системы координат. Но если требуется обозначить производную по конкретной координате, то, по аналогии с обычными векторами, это делается через нижний индекс:

$$\nabla_n f = \frac{\partial f}{\partial n}.$$

Для этого символа справедливы все векторные операции, описанные ранее. Так, применение ∇ к скалярной функции аналогично умножению вектора на скаляр (A.1) (здесь и далее приводятся покоординатные выражения для декартовой системы):

$$\nabla f = (\nabla_x f,\ \nabla_y f,\ \nabla_z f) = \frac{\partial f}{\partial x}\hat{\mathbf{x}} + \frac{\partial f}{\partial y}\hat{\mathbf{y}} + \frac{\partial f}{\partial z}\hat{\mathbf{z}}. \tag{A.5}$$

Результатом этой операции является вектор.

Скалярное умножение ∇ на вектор $\mathbf{v}$ по аналогии с (A.2) – есть дивергенция:

$$\nabla \cdot \mathbf{v} = \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} \tag{A.6}$$

результат которой – скалярная функция.

Двойное применение ∇ к скалярной функции – это оператор Лапласа:

$$\nabla \cdot \nabla f = \nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2} \tag{A.7}$$

Ротор – аналог векторного умножнения (A.3):

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

A.2 Интегрирование

A.2.1 Формула Гаусса–Остроградского

Формула Гаусса–Остроградского, связывающая интегрирование по объёму E с интегрированием по границе этого объёма Г, для векторного поля $\mathbf{v}$ имеет вид

$$\int\limits_E \nabla \cdot \mathbf{v}\, d\mathbf{x} = \int\limits_\Gamma v_n\, ds, \tag{A.9}$$

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. 7. Разобъем область на $N = 3$ равномерных подобласти

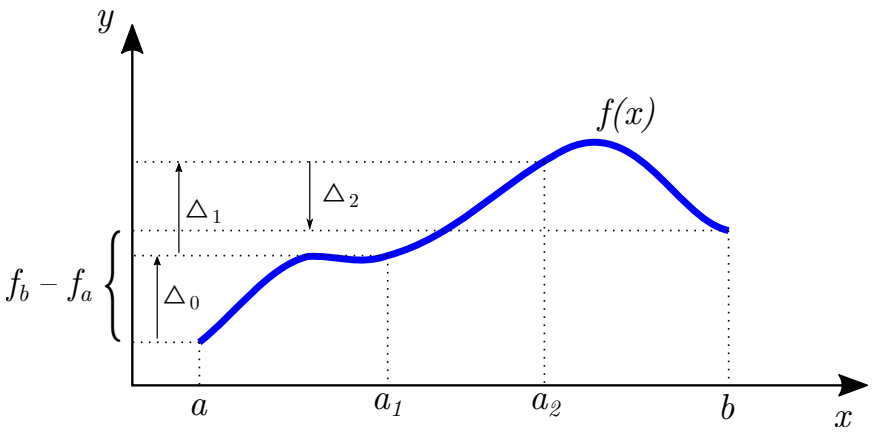


Рис. 7: Формула Гаусса-Остроградского в одномерном случае

длины h. Тогда расписывая интеграл как сумму, а производную через конечную разность, получим

$$\int\limits_E \frac{\partial f}{\partial x}\, dx \approx \sum_{i=0}^{2} h \left(\frac{\partial f}{\partial x}\right)_{i+\frac{1}{2}} \approx \sum_{i=0}^{2} (f_{i+1} - f_i) = \triangle_0 + \triangle_1 + \triangle_2 = f_b - f_a.$$

Очевидно что, при устремлении $N \to \infty$ правая часть предыдущего выражения не изменится. То есть, сумма всех изменений функции в области есть изменение функции по её границам:

$$\int\limits_a^b \frac{\partial f}{\partial x}\, dx = f(b) - f(a).$$

А формула (A.9) – есть многомерное обобщение этого выражения.

A.2.2 Интегрирование по частям

Подставив в (A.9) $\mathbf{v} = f\mathbf{u}$, где f – некоторая скалярная функция, и расписав дивергенцию в виде

$$\nabla \cdot (f\mathbf{u}) = f\nabla\mathbf{u} + \mathbf{u} \cdot \nabla f$$

получим формулу интегрирования по частям

$$\int\limits_E \mathbf{u} \cdot \nabla f \, d\mathbf{x} = \int\limits_\Gamma f u_n \, ds - \int\limits_E f \nabla \cdot \mathbf{u} \, d\mathbf{x} \tag{A.10}$$

Распишем некоторые частные случаи для формулы (A.10). Для $\mathbf{u} = (n_x, 0, 0)$ получим

$$\int\limits_E \frac{\partial f}{\partial x} \, d\mathbf{x} = \int\limits_\Gamma f \cos(\widehat{\mathbf{n}, \mathbf{x}}) \, ds \tag{A.11}$$

При $\mathbf{u} = \nabla g$

$$\int\limits_E f \left(\nabla^2 g\right) \, d\mathbf{x} = \int\limits_\Gamma f \frac{\partial g}{\partial n} \, ds - \int\limits_E \nabla f \cdot \nabla g \, d\mathbf{x} \tag{A.12}$$

При $f = 1$ и $\mathbf{u} = \nabla g$

$$\int\limits_E \nabla^2 g \, d\mathbf{x} = \int\limits_\Gamma \frac{\partial g}{\partial n} \, ds \tag{A.13}$$

A.2.3 Численное интегрирование в заданной области

Квадратурная формула

$$\int\limits_E f(\mathbf{x}) \, d\mathbf{x} = \sum_{i=0}^{N-1} w_i f(\mathbf{x}_i) \tag{A.14}$$

Она определяется заданием узлов интегрирования $\mathbf{x}_i$ и соответствующих весов w_i.

A.3 Интерполяционные полиномы

A.3.1 Многочлен Лагранжа

A.3.1.1 Узловые базисные функции

Рассмотрим функцию $f(\xi)$, заданную в области D. Внутри этой области зададим N узловых точек $\xi_i, i = \overline{0, N-1}$. Приближение функции f будем искать в виде

$$f(\xi) \approx \sum_{i=0}^{N-1} f_i \phi_i(\xi), \tag{A.15}$$

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

$$\phi_i(\xi_j) = \begin{cases} 1, & i = j, \\ 0, & i \neq j. \end{cases} \tag{A.16}$$

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

$$\sum_{i=0}^{N-1} \phi_i(\xi) = 1, \qquad \xi \in D. \tag{A.17}$$

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

$$\phi_i(\xi) = \sum_a A_i^{(a)} \xi^a = A_i^{(0)} + A_i^{(1)} \xi + A_i^{(2)} \xi^2 + \ldots, \qquad i = \overline{0, N-1}. \tag{A.18}$$

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

$$\begin{aligned}
&\phi_0(\xi_0) = A_0^{(0)} + A_0^{(1)} \xi_0 + A_0^{(2)} \xi_0^2 + A_0^{(3)} \xi_0^3 + \ldots = 1, \\
&\phi_0(\xi_1) = A_0^{(0)} + A_0^{(1)} \xi_1 + A_0^{(2)} \xi_1^2 + A_0^{(3)} \xi_1^3 + \ldots = 0, \\
&\phi_0(\xi_2) = A_0^{(0)} + A_0^{(1)} \xi_2 + A_0^{(2)} \xi_2^2 + A_0^{(3)} \xi_2^3 + \ldots = 0, \\
&\ldots
\end{aligned}$$

или в матричном виде

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

Записывая аналогичные выражения для остальных базисных функций, получим систему матричных уравнений вида $CA = E$:

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

Отсюда матрица неизвестных коэффициентов A определится как

$$A = C^{-1} = \begin{pmatrix} 1 & \xi_0 & \xi_0^2 & \xi_0^3 & \dots \\ 1 & \xi_1 & \xi_1^2 & \xi_1^3 & \dots \\ 1 & \xi_2 & \xi_2^2 & \xi_2^3 & \dots \\ 1 & \xi_3 & \xi_3^2 & \xi_3^3 & \dots \\ \dots & & & & \end{pmatrix}^{-1}. \tag{A.19}$$

Подставляя полином (A.18) в условие согласованности (A.17), получим требование

$$\sum_{i=0}^{N-1} A_i^{(a)} = \begin{cases} 1, & a = 0, \\ 0, & a = \overline{1, N-1}. \end{cases}$$

То есть сумма всех свободных членов в интерполяционных полиномах должна быть равна единице, а сумма коэффициентов при остальных степенях – нулю. Можно показать, что это свойство выполняется для любой матрицы $A = C^{-1}$, в случае, если первый столбец матрицы C состоит из единиц. То есть условие согласованности требует наличие свободного члена с интерполяционном полиноме.

A.3.1.2 Интерполяция в параметрическом отрезке

Будем рассматривать область интерполяции $D = [-1, 1]$. В качестве первых двух узлов интерполяции возьмем границы области: $\xi_0 = -1$, $\xi_1 = 1$.

Линейный базис Будем искать интерполяционный базис в виде

$$\phi_i(\xi) = A_i^{(0)} + A_i^{(1)}\xi.$$

на основе двух условий:

$$\phi_i(-1) = A_i^{(0)} - A_i^{(1)} = \delta_{0i}, \quad \phi_i(1) = A_i^{(0)} + A_i^{(1)}\delta_{1i}.$$

Составим матрицу C, записав эти условия в матричном виде

$$C = \left(\begin{array}{c|cc} & A^{(0)} & A^{(1)} \\ \hline \phi(-1) & 1 & -1 \\ \phi(1) & 1 & 1 \end{array}\right)$$

и, согласно (A.19), найдём матрицу коэффициентов

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

Отсюда узловые базисные функции примут вид (рис. 8)

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

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

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

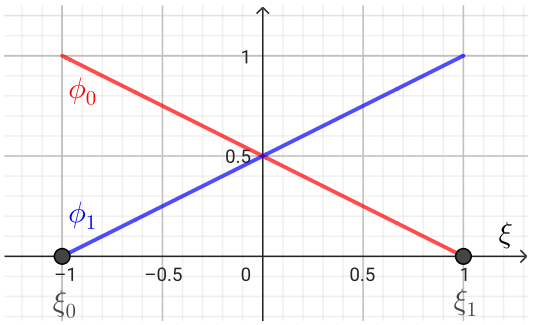


Рис. 8: Линейный базис в параметрическом отрезке

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 неизвестый коэффициент $A_i^{(2)}$, поэтому в набор условий (A.16) требуется ещё одно уравнение (ещё одна узловая точка). Поме-

стим её в центр параметрического сегмента $\xi_2 = 0$. Далее будем действовать по аналогии с линейным случаем:

$$C = \left(\begin{array}{c|ccc} & A^{(0)} & A^{(1)} & A^{(2)} \\ \hline \phi(-1) & 1 & -1 & 1 \\ \phi(1) & 1 & 1 & 1 \\ \phi(0) & 1 & 0 & 0 \end{array}\right) \quad \Rightarrow \quad A = C^{-1} = \left(\begin{array}{c|ccc} & \phi_0 & \phi_1 & \phi_2 \\ \hline 1 & 0 & 0 & 1 \\ \xi & -\frac{1}{2} & \frac{1}{2} & 0 \\ \xi^2 & \frac{1}{2} & \frac{1}{2} & -1 \end{array}\right).$$

Узловые базисные функции для квадратичной интерполяции примут вид (рис. 9)

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

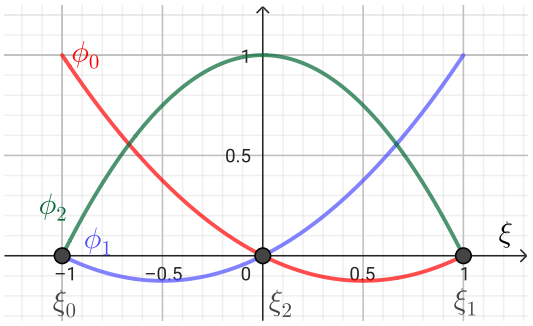


Рис. 9: Квадратичный базис в параметрическом отрезке

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:

$$C = \left(\begin{array}{c|cccc} & A^{(0)} & A^{(1)} & A^{(2)} & A^{(3)} \\ \hline \phi(-1) & 1 & -1 & 1 & -1 \\ \phi(1) & 1 & 1 & 1 & 1 \\ \phi(-\frac{1}{3}) & 1 & -\frac{1}{3} & \frac{1}{9} & -\frac{1}{27} \\ \phi(\frac{1}{3}) & 1 & \frac{1}{3} & \frac{1}{9} & \frac{1}{27} \end{array}\right) \quad \Rightarrow \quad A = C^{-1} = \frac{1}{16}\left(\begin{array}{c|cccc} & \phi_0 & \phi_1 & \phi_2 & \phi_3 \\ \hline 1 & -1 & -1 & 9 & 9 \\ \xi & 1 & -1 & -27 & 27 \\ \xi^2 & 9 & 9 & -9 & -9 \\ \xi^3 & -9 & 9 & 27 & -27 \end{array}\right)$$

Узловые базисные функции для квадратичной интерполяции примут вид (рис. 10)

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

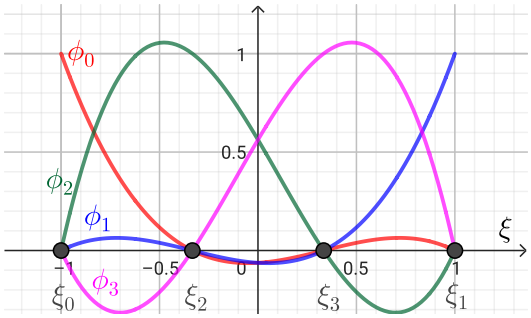


Рис. 10: Кубический базис в параметрическом отрезке

На рис. 11 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.

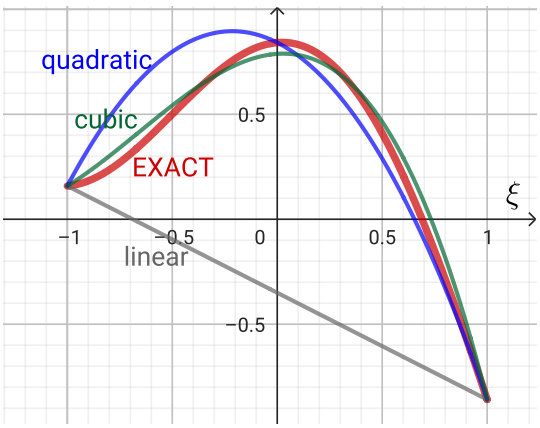


Рис. 11: Результат интерполяции

A.3.1.3 Интерполяция в параметрическом треугольнике

Теперь рассмотрим двумерное обобщение формулы

Линейный базис

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

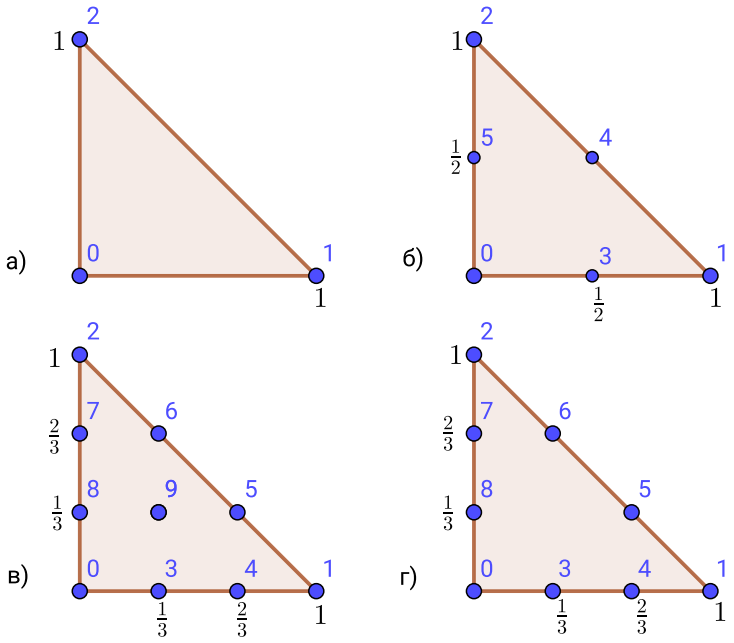


Рис. 12: Расположение узловых точек в параметрическом треугольнике. а) линейный базис, б) квадратичный базис, в) кубический базис, г) неполный кубический базис

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

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

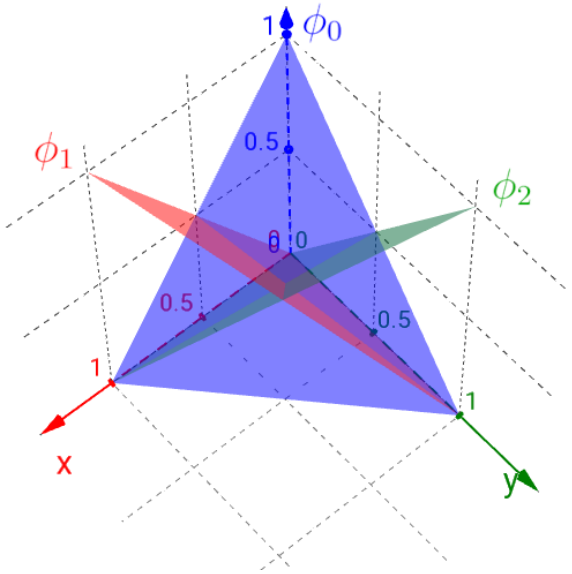


Рис. 13: Линейный базис в параметрическом треугольнике

Квадратичный базис

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

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

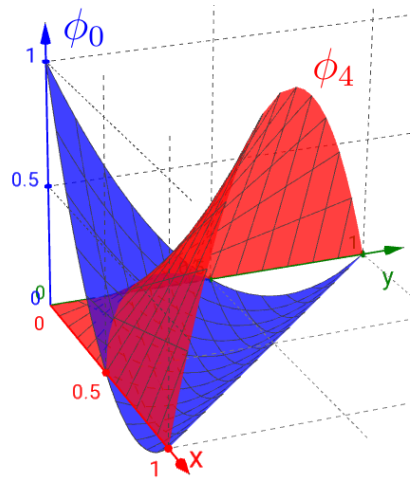


Рис. 14: Квадратичные функции ϕ_0, ϕ_4 в параметрическом треугольнике

Кубический базис TODO

Неполный кубический базис TODO

A.3.1.4 Интерполяция в параметрическом квадрате

Билинейный базис

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

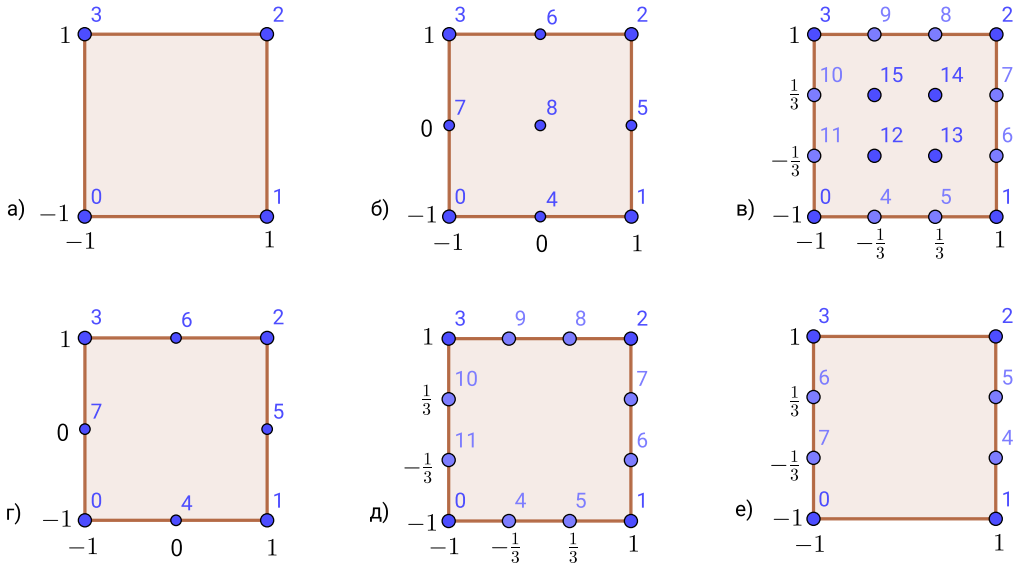


Рис. 15: Расположение узловых точек в параметрическом квадратре

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

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

Определение двумерных базисов через комбинацию одномерных Обратим внимание, что в искомые билинейные базисные функции линейны в каждом из направлений ξ, η, если брать их по отдельности. Значит можно представить эти функции как комбинацию одномерных линейных базисов (A.20) в каждом из направлений. Узлы двумерного параметрического квадрата можно выразить через узлы линейного базиса в параметрическом одномерном сегменте, рассмотренном в п. A.3.1.2:

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

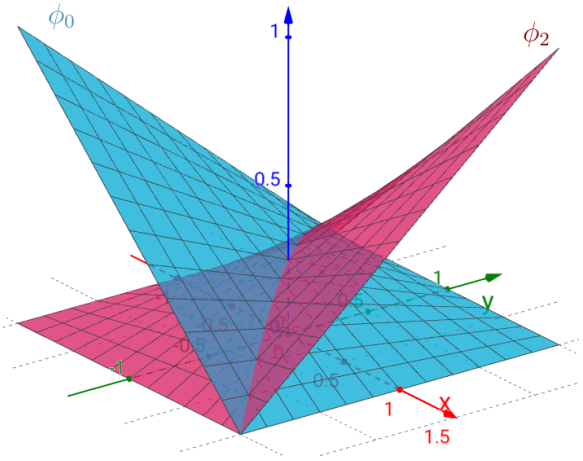


Рис. 16: Билинейные функции ϕ_0, ϕ_2 в параметрическом квадрате

Значит и соответствующие базисные функции можно выразить через линейный одномерный базис ϕ^{1D} из соотношений (A.20):

$$\phi_0(\xi,\eta) = \phi_0^{1D}(\xi)\phi_0^{1D}(\eta) = \frac{1-\xi}{2}\frac{1-\eta}{2},$$
$$\phi_1(\xi,\eta) = \phi_1^{1D}(\xi)\phi_0^{1D}(\eta) = \frac{1+\xi}{2}\frac{1-\eta}{2},$$
$$\phi_2(\xi,\eta) = \phi_1^{1D}(\xi)\phi_1^{1D}(\eta) = \frac{1+\xi}{2}\frac{1+\eta}{2},$$
$$\phi_3(\xi,\eta) = \phi_0^{1D}(\xi)\phi_1^{1D}(\eta) = \frac{1-\xi}{2}\frac{1+\eta}{2}.$$

Раскрыв скобки можно убедится, что мы получили тот же билинейный базис, что и ранее (A.24).

Биквадратичный базис Применим этот метод для вычисления биквадратичного базиса, определённого в точках на рис. 15б. В качестве основе возьмём квадратичный одномерный базис ϕ_i^{1D} из (A.21).

$$\begin{aligned}
&\phi_0(\xi,\eta) = \phi_0^{1D}(\xi)\phi_0^{1D}(\eta) = \frac{\xi^2-\xi}{2}\frac{\eta^2-\eta}{2}, && \phi_1(\xi,\eta) = \phi_1^{1D}(\xi)\phi_0^{1D}(\eta) = \frac{\xi^2+\xi}{2}\frac{\eta^2-\eta}{2},\\
&\phi_2(\xi,\eta) = \phi_1^{1D}(\xi)\phi_1^{1D}(\eta) = \frac{\xi^2+\xi}{2}\frac{\eta^2+\eta}{2}, && \phi_3(\xi,\eta) = \phi_0^{1D}(\xi)\phi_1^{1D}(\eta) = \frac{\xi^2-\xi}{2}\frac{\eta^2+\eta}{2},\\
&\phi_4(\xi,\eta) = \phi_2^{1D}(\xi)\phi_0^{1D}(\eta) = (1-\xi^2)\frac{\eta^2-\eta}{2}, && \phi_5(\xi,\eta) = \phi_1^{1D}(\xi)\phi_2^{1D}(\eta) = \frac{\xi^2+\xi}{2}(1-\eta^2),\\
&\phi_6(\xi,\eta) = \phi_2^{1D}(\xi)\phi_1^{1D}(\eta) = (1-\xi^2)\frac{\eta^2+\eta}{2}, && \phi_7(\xi,\eta) = \phi_0^{1D}(\xi)\phi_2^{1D}(\eta) = \frac{\xi^2-\xi}{2}(1-\eta^2),\\
&\phi_8(\xi,\eta) = \phi_2^{1D}(\xi)\phi_2^{1D}(\eta) = (1-\xi^2)(1-\eta^2).
\end{aligned} \tag{A.25}$$

Бикубический базис

Неполный биквадратичный базис

Неполный бикубический базис

A.4 Геометрические алгоритмы

A.4.1 Линейная интерполяция

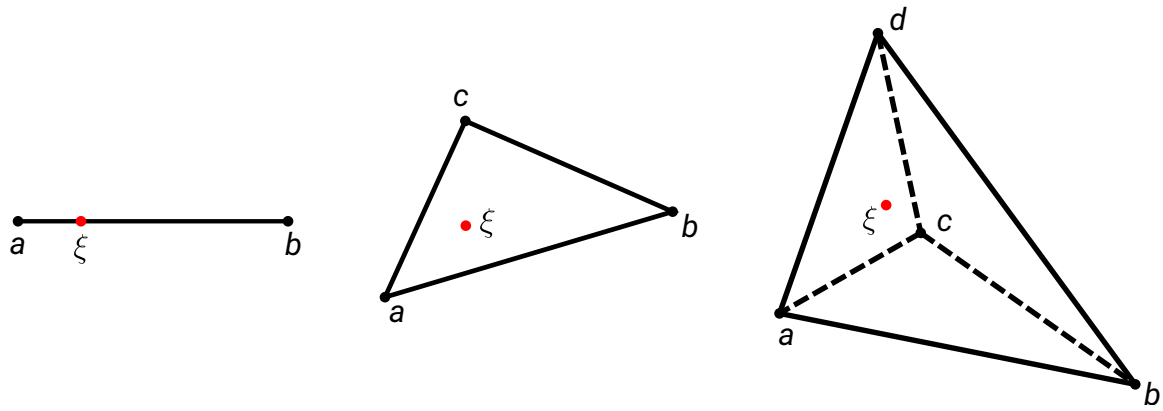


Рис. 17: Порядок нумерации точек одномерного, двумерного и трёхмерного симплекса при линейной интерполяции

Пусть функция u задана в узлах симплекса, имеющего нумерацию согласно рис. 17. Необходимо найти значение этой функции в точке $\boldsymbol{\xi}$ (эта точка вообще говоря не обязана лежать внутри симлекса).

Интерполяция в одномерном, двумерном и трёхмерном виде запишется как

$$u(\xi) = \frac{|\triangle_{\xi a}|u(b) + |\triangle_{b\xi}|u(a)}{|\triangle_{ba}|} \tag{A.26}$$

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

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

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

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

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

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

A.4.2 Преобразование координат

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

A.4.2.1 Матрица Якоби

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

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

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

$$\mathbf{x}(\boldsymbol{\xi}_0 + \mathbf{d\xi}) - \mathbf{x}_0 = J(\boldsymbol{\xi}_0)\, \mathbf{d\xi}. \tag{A.29}$$

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

$$J = \begin{pmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{pmatrix} = \begin{pmatrix} \dfrac{\partial x}{\partial \xi} & \dfrac{\partial x}{\partial \eta} \\ \dfrac{\partial y}{\partial \xi} & \dfrac{\partial y}{\partial \eta} \end{pmatrix} \tag{A.30}$$

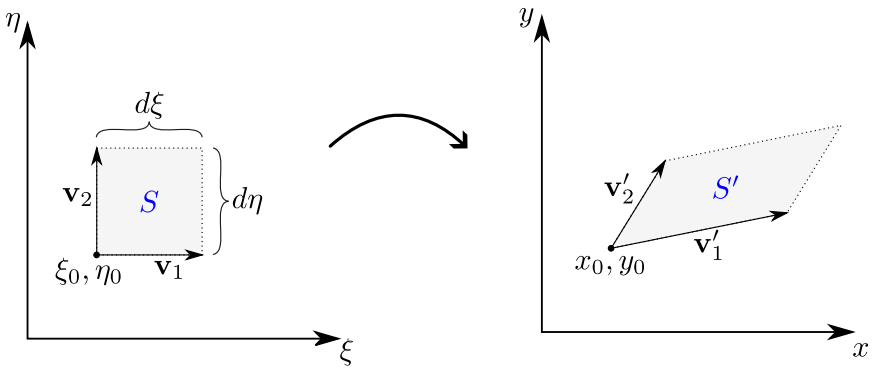


Рис. 18: Преобразование элементарного объёма

Якобиан Определитель матрицы Якоби (якобиан), взятый в конкретной точке параметрической плоскости $\boldsymbol{\xi}_0$, показывает, во сколько раз увеличился элементарный объём около этой точки в результате преобразования. Действительно, рассмотрим два перпендикулярных элементарных вектора в параметрической системе координат: $\mathbf{v}_1 = (d\xi, 0)$ и $\mathbf{v}_2 = (0, d\eta)$ отложенных от точки $\boldsymbol{\xi}_0$ (см. рис. 18). В результате преобразования по формуле (A.29) получим следующие преобразования концевых точек и векторов:

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

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

$$dx\,dy = |J|\,d\xi\,d\eta \tag{A.31}$$

Многомерным обобщением этой формулы будет

$$d\mathbf{x} = |J|\,d\boldsymbol{\xi} \tag{A.32}$$

A.4.2.2 Дифференцирование в параметрической плоскости

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

$$\frac{\partial f}{\partial \xi} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial \xi} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial \xi},$$
$$\frac{\partial f}{\partial \eta} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial \eta} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial \eta}.$$

Вспоминая определение (A.30), запишем

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

Обратная зависимость примет вид

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

В многомерном виде запишем

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

А.4.2.3 Интегрирование в параметрической плоскости

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

$$\int\limits_{D_x} f(\mathbf{x})\, d\mathbf{x} = \int\limits_{D_\xi} f(\boldsymbol{\xi})\, |J(\boldsymbol{\xi})| d\boldsymbol{\xi}, \tag{А.34}$$

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

А.4.2.4 Двумерное линейное преобразование. Параметрический треугольник

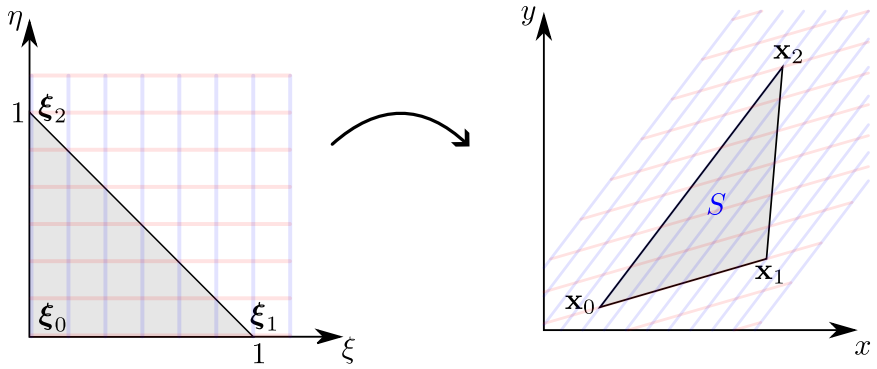


Рис. 19: Преобразование из параметрического треугольника

Рассмотрим двумерное преобразование, при котором определяющие функции являются линейными. То есть представимыми в виде

$$x(\xi, \eta) = A_x \xi + B_x \eta + C_x,$$
$$y(\xi, \eta) = A_y \xi + B_y \eta + C_y.$$

Для определения шести констант, определяющих это преобразование, достаточно выбрать три любые (не лежащие на одной прямой) точки: $(\xi_i, \eta_i) \to (x_i, y_i)$ 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. 19):

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

Тогда система линейных уравнений примет вид

$$x_0 = C_x, \quad y_0 = C_y,$$
$$x_1 = A_x + C_x, \quad y_1 = A_y + C_y,$$
$$y_2 = B_x + C_x, \quad y_2 = B_y + C_y.$$

Определив коэффициенты преобразования их этой системы, окончательно запишем преобразование

$$x(\xi, \eta) = (x_1 - x_0)\xi + (x_2 - x_0)\eta + x_0,$$
$$y(\xi, \eta) = (y_1 - y_0)\xi + (y_2 - y_0)\eta + y_0. \tag{А.35}$$

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

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

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

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

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

$$\int\limits_S f(x,y)\,dxdy = |J| \int\limits_0^1 \int\limits_0^{1-\xi} f(\xi,\eta) d\eta d\xi. \tag{A.38}$$

A.4.2.5 Двумерное билинейное преобразование. Параметрический квадрат

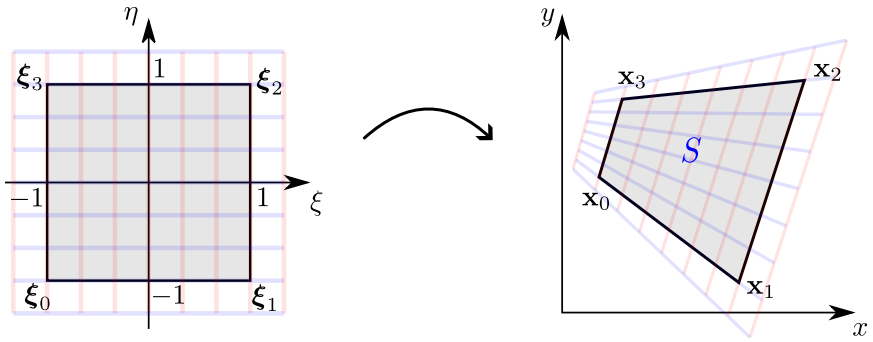


Рис. 20: Преобразование из параметрического квадрата

A.4.2.6 Трёхмерное линейное преобразование. Параметрический тетраэдр

TODO

A.4.3 Свойства многоугольника

A.4.3.1 Площадь многоугольника

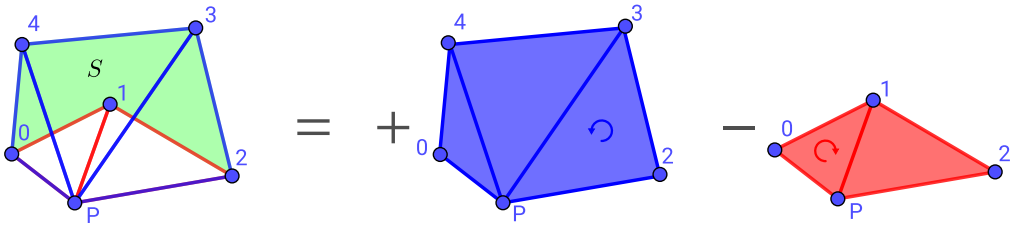


Рис. 21: Площадь произвольного многоугольника

Рассмотрим произвольный несамопересекающийся N-угольник S, заданный координатами своих узлов $\mathbf{x}_i$, $i = \overline{0, N-1}$, пронумерованных последовательно против часовой стрелки (рис. 21). 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 $\triangle_i^p$:

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

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

$$\triangle_i = (\mathbf{x}_0, \mathbf{x}_i, \mathbf{x}_{i+1}), \qquad i = \overline{1, N-2}. \tag{A.39}$$

Знаковая площадь ориентированного треугольника будет равна

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

Тогда окончательно формула определения площади примет вид

$$|S| = \sum_{i=1}^{N-2} |\triangle_i|. \tag{A.41}$$

Плоский полигон в пространтве Если плоский полигон S расположен в трёхмерном пространстве, то правая часть формулы (A.40) согласно определению векторного произведения в трёхмерном пространстве (A.3) – есть вектор. Чтобы получить скалярную площадь, нужно спроецировать этот вектор на единичную нормаль к плоскости многоугольника:

$$\mathbf{n} = \frac{\mathbf{k}}{|\mathbf{k}|}, \qquad \mathbf{k} = (\mathbf{x}_1 - \mathbf{x}_0) \times (\mathbf{x}_2 - \mathbf{x}_0).$$

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:

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

Формула для определения площади полигона (A.41) будет по прежнему верна. При этом итоговый знак величины S будет положительным, если закрутка полигона положительная (против часовой стрелки) при взгляде со стороны вычисленной нормали $\mathbf{n}$.

A.4.3.2 Интеграл по многоугольнику

Рассмотрим интеграл функции $f(x, y)$ по N-угольнику S, заданному последовательными координатами своих узлов $\mathbf{x}_i$. Введём последовательную триангуляцию согласно (A.39). Тогда интеграл по многоугольнику можно расписать как сумму интегралов по ориентированным треугольникам:

$$\int\limits_S f(x,y)\,dxdy = \sum_{i=1}^{N-2} \int\limits_{\triangle_i} f(x,y)\,dxdy. \tag{A.43}$$

Далее для вычисления интегралов в правой части воспользуемся преобразованием к параметрическому треугольнику (п. A.4.2.4). Следуя формуле интегрирования (A.38), распишем интеграл по i-ому треугольнику:

$$\int\limits_{\triangle_i} f(x,y)\,dxdy = |J_i| \int\limits_0^1 \int\limits_0^{1-\xi} f_i(\xi,\eta)\,d\eta d\xi,$$

где якобиан $|J_i|$ согласно (A.37) есть удвоенная площадь ориентированного треугольника $\triangle_i$ (положительная при закрутке против часовой стрелке и отрицательная иначе):

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

а функция $f_i(\xi, \eta)$ есть функция от преобразованных согласно (A.35) переменных:

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

Окончательно запишем

$$\int\limits_S f(x,y)\,dxdy = 2\sum_{i=1}^{N-2} |\triangle_i| \int\limits_0^1 \int\limits_0^{1-\xi} f_i(\xi,\eta)\,d\eta d\xi. \tag{A.44}$$

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

A.4.3.3 Центр масс многоугольника

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

$$c_x = \frac{1}{|S|}\int\limits_S x\,dxdy, \quad c_y = \frac{1}{|S|}\int\limits_S y\,dxdy.$$

Далее распишем интеграл в правой части через последовательную триангуляцию согласно (A.43) с учётом линейного преобразования (A.35):

$$\begin{aligned}\int\limits_S x\,dxdy &= \sum_{i=1}^{N-2}\int\limits_{\triangle_i} x\,dxdy \\ &= \sum_{i=1}^{N-2}|J_i|\int\limits_0^1\int\limits_0^{1-\xi}((x_i-x_0)\xi+(x_{i+1}-x_0)\eta+x_0)\,d\eta d\xi \\ &= \sum_{i=1}^{N-2}\frac{|J_i|}{2}\frac{x_0+x_i+x_{i+1}}{3} \\ &= \sum_{i=1}^{N-2}|\triangle_i|\frac{x_0+x_i+x_{i+1}}{3}.\end{aligned}$$

Итого, с учётом (A.41), координаты центра масс примут вид

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

Если полигон расположен в двумерном пространстве xy, то знаковая площадь треугольников вычисляется по формуле (A.40). В случае трёхмерного пространтва должна использоваться формула (A.42).

A.4.4 Свойства многогранника

A.4.4.1 Объём многогранника

TODO

A.4.4.2 Интеграл по многограннику

TODO

A.4.4.3 Центр масс многогранника

TODO

A.4.5 Поиск многоугольника, содержащего заданную точку

TODO

B Работа с инфраструктурой проекта CFDCourse

B.1 Сборка и запуск

B.1.1 Сборка проекта CFDCourse

Описанная ниже процедура собирает проект в отладочной конфигурации. Для проведения необходимых модификаций для сборки релизной версии смотри B.1.3.

B.1.1.1 Подготовка

1. Для сборки проекта необходимо установить `git` и `cmake>=3.0`

   В Windows необходимо скачать и установить диструбутивы:

   - 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 компиллятор `g++` и отладчик `gdb`.

2. Создайте папку в системе для репозиториев. Например `D:/git_repos/`

3. Возьмите необходимые заголовочные библиотеки boost из https://disk.yandex.ru/d/GwTZUvfAqPsZ
   и распакуйте архив в папку для репозиториев (D:/git_repos/boost). Проследите, чтобы внутри папки boost сразу шли папки с кодом (`accumulators`, `algorithm`, ...) и заголовочные файлы (`align.hpp`, `aligned_storage.hpp`, ...) без дополнительных уровней вложения.

4. Откройте терминал (git bash в Windows).

5. С помощью команды cd в терминале перейдите в папку для репозиториев

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

6. Клонируйте репозиторий

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

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

B.1.1.2 VisualStudio

1. Создайте папку build в корне проекта CFDCourse25

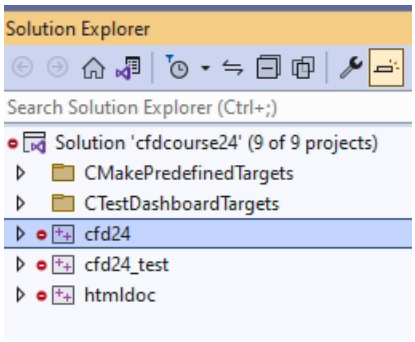
2. Скопируйте скрипт winbuild64.bat в папку build. Далее вносить изменения только в скопированном файле.

3. Скрипт написан для версии `Visual Studio 2019`. Если используется другая версия, измените в скрипте значение переменной `CMGenerator` на соответствующие вашей версии. Значения для разных версий Visual Studio написаны ниже

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

4. Запустите скрипт `winbuild64.bat` из папки `build`. Нужен доступ к интернету. В процессе будет скачано около 200Мб пакетов, поэтому первый запуск может занять время

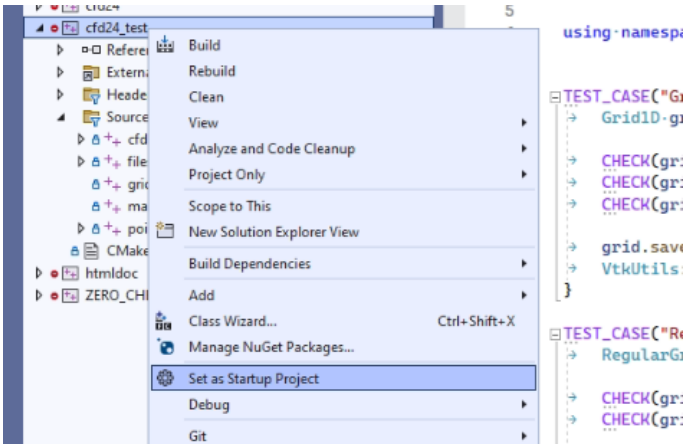
5. После сборки в папке `build` появится проект `VisualStudio` `cfdcourse25.sln`. Его нужно открыть в `VisualStudio`. Дерево решения должно иметь следующий вид:



Проекты:

- `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. Скомпиллировать решение. Несколько способов:

- `Ctrl+Shift+B`,

- `Build->Build Solution` в основном меню,
- `Build Solution` в меню решения в дереве решения,
- `Build` в меню проекта `cfd25_test`.

8. Запустить тесты (проект `cfd25_test`) нажав `F5` (или кнопку отладки в меню). После отработки должно высветиться сообщение об успешном прохождении всех тестов.

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

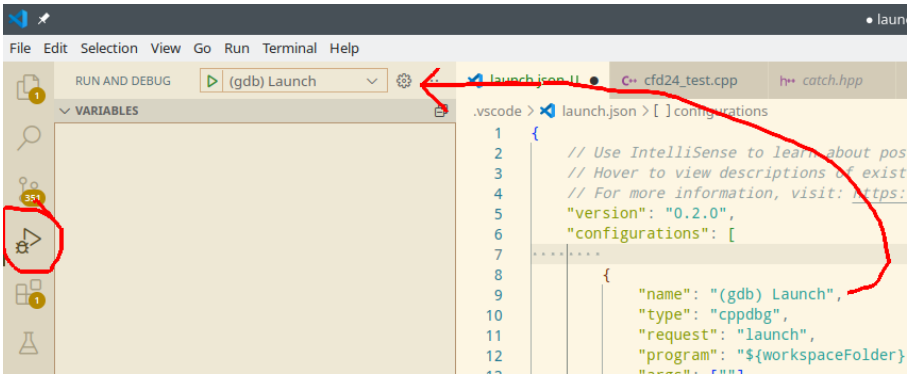
B.1.1.3 VSCode

1. Открыть корневую папку проека через `File->Open Folder`

2. Установить предлагаемые расширения cmake, c++

3. Для настройки отладки создайте конфигурацию launch.json следующего вида

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



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

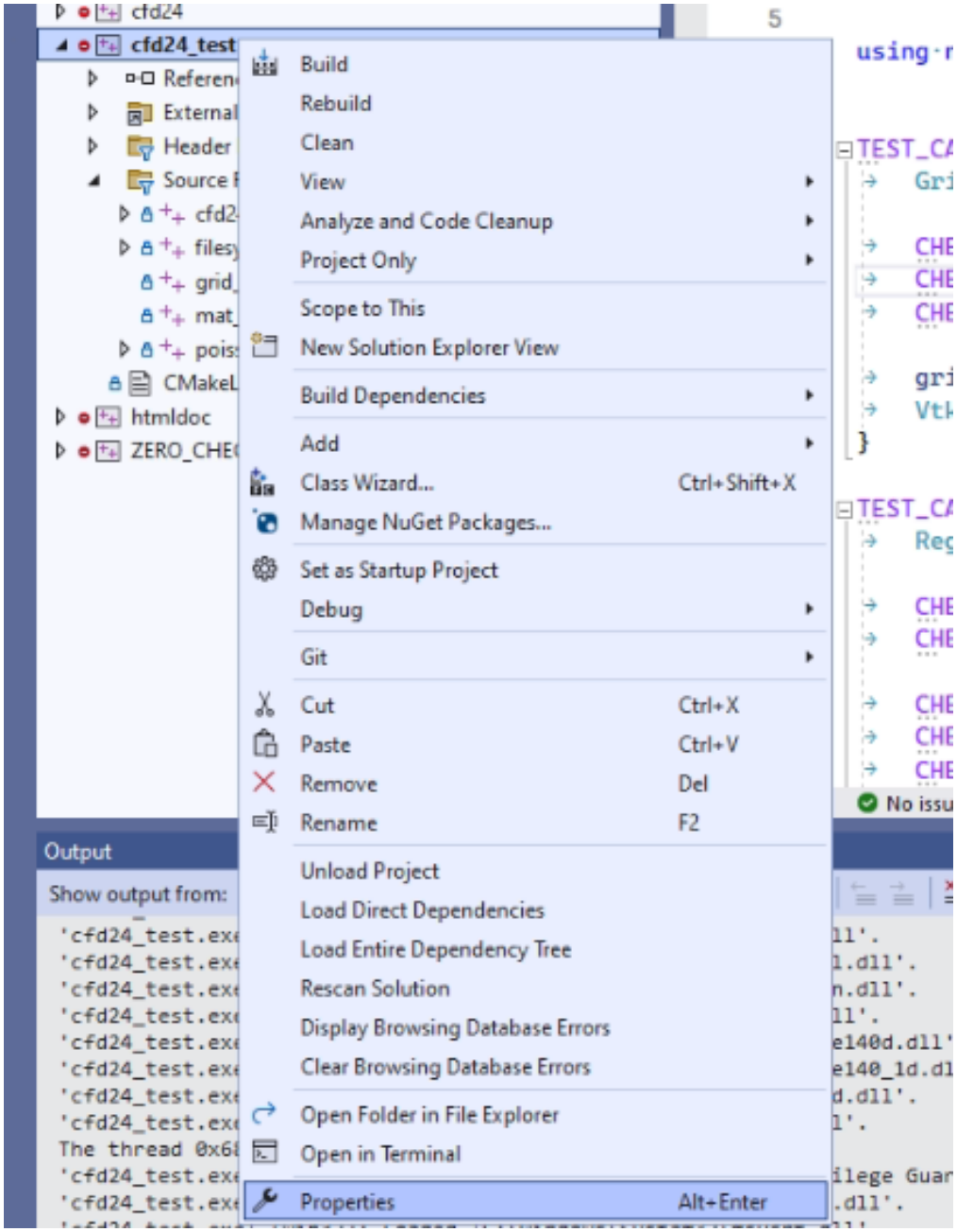
```
"name" : "(Windows) Launch",
"program": "${workspaceFolder}/build/bin/Debug/cfd25_test.exe"
```

B.1.2 Запуск конкретного теста

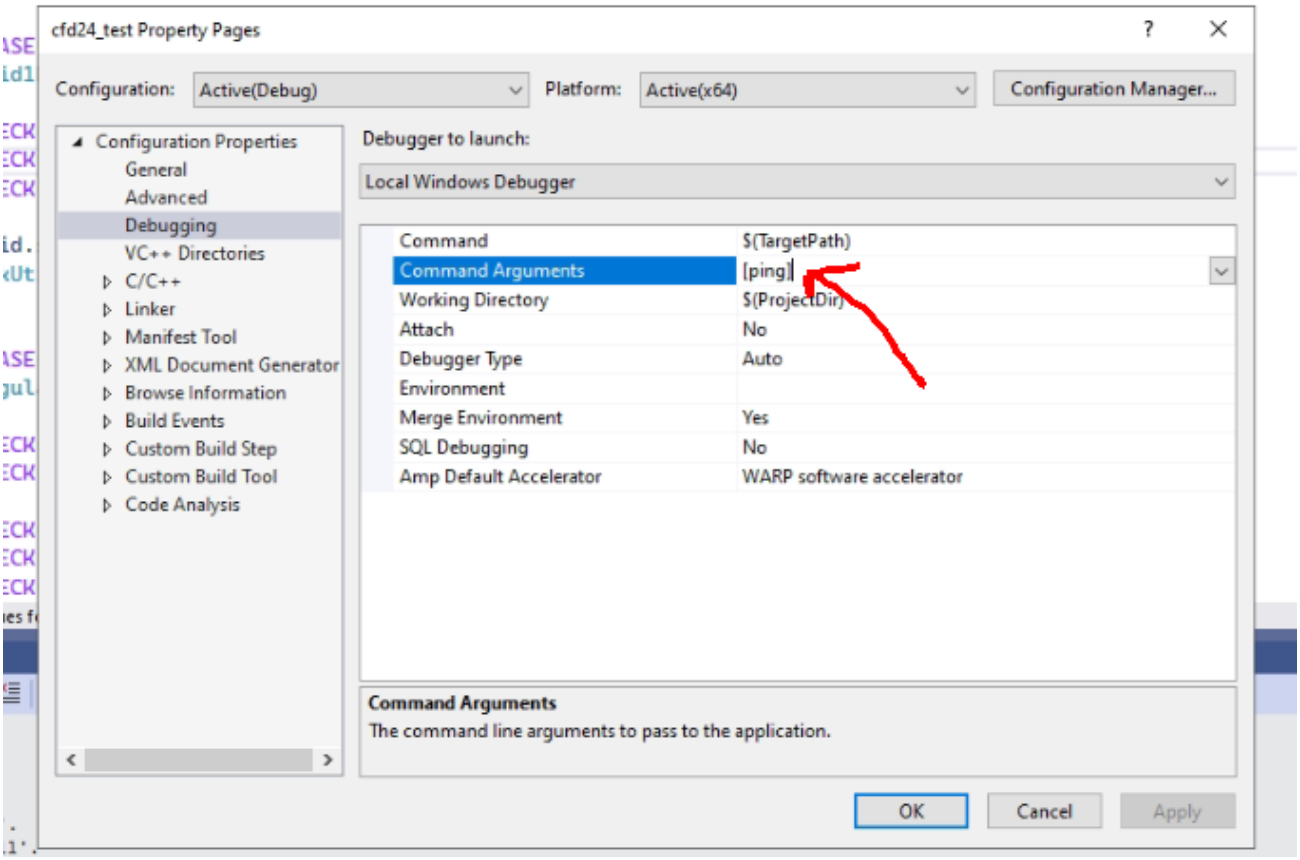
По умолчанию программа `cfd25_test` прогоняет все объявленные в проекте тесты. Иногда может возникнуть необходимость запустить только конкретный тест в целях отладки или проверки. Для этого нужно передать программе аргумент с тегом для этого теста.

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

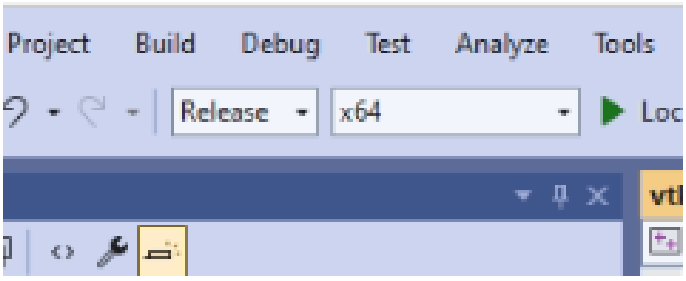
Релизная сборка программ даёт многократное увеличение производительности, но при этом отладка приложений в таком режиме невозможна.

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 запускускаемым проектом `cfd_test` и, если нужно, настроить аргументы отладки.

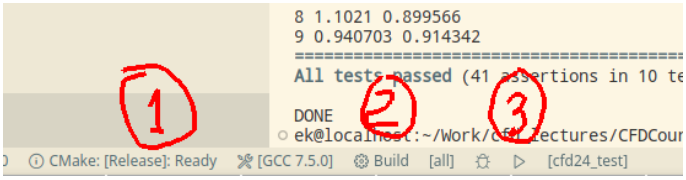
8. Бинарные файлы будут скомпиллированы в папку `CFDCourse25/build_release/bin/Release`. В случае работы через отладчик выходная директория – `CFDCourse25/build_release/src/test/`.

VSCode

1. Выбрать релизную сборку в `build variant`

2. Нажать `Build`

3. Нажать `Launch`



B.2 Git

B.2.1 Основные команды

Все команды выполнять в терминале (`git bash` для виндоус), находясь в корневой папке 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 комманда работает даже если есть нескоммиченные изменения. Если необходимо скоммитить изменеия в новую ветку, сразу за этой командой нужно вызвать

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

- **Сбросить** все нескоммиченные изменения. Вернуть файлы в состояние последнего коммита

```
> git reset --hard
```

Все изменения будут утеряны

- **Получить последние изменения** из удалённого хранилища с обновлением текущей ветки

```
> git pull
```

Работает только если статус текущей ветки 'Up to date'.
Если требуется получить изменения, но не обновлять локальную ветку:

```
> git fetch
```

Обновленная ветка будет доступна по имени origin/имя ветки.

- **Просмотр истории** коммитов в текущей ветке (последний коммит будет наверху)

```
> git log
```

- **Просмотр доступных веток** в текущем репозитории

```
> git branch
```

- **Просмотр** актуального состояние дерева репозитория в gui режиме

```
> git gui
```

Далее в меню
`Repository->Visualize all branch history`. В этом же окне можно посмотреть изменения файлов по сравнению с последним коммитом.

Альтернативно, при работе в виндоус можно установить программу GitExtensions и работать в ней.

B.2.2 Порядок работы с репозиторием CFDCourse

Основная ветка проекта –
`main`. После каждой лекции в эту ветку будет отправлен коммит с сообщением
`after-lect{index}`. Этот коммит будет содержать краткое содержание лекции, задание по итогам лекции и необходимые для этого задания изменения кода.

Таким образом, **после лекции**, после того, как изменение
`after-lect` 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

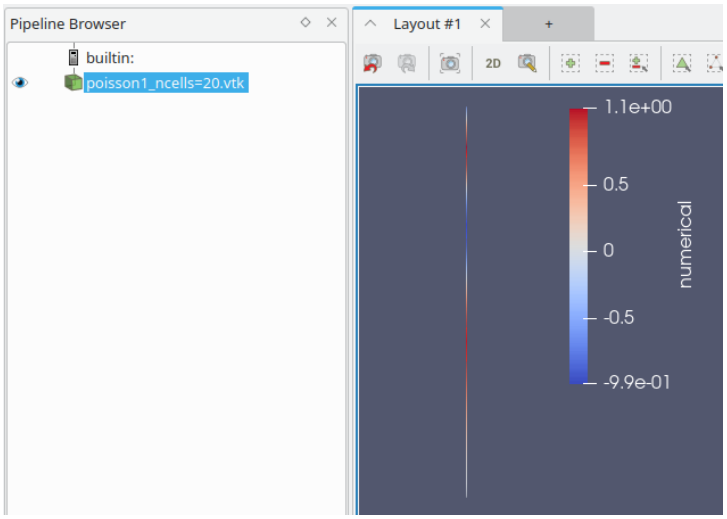
```
> git checkout work-lect{index}
```

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

B.3 Paraview

B.3.1 Данные на одномерных сетках

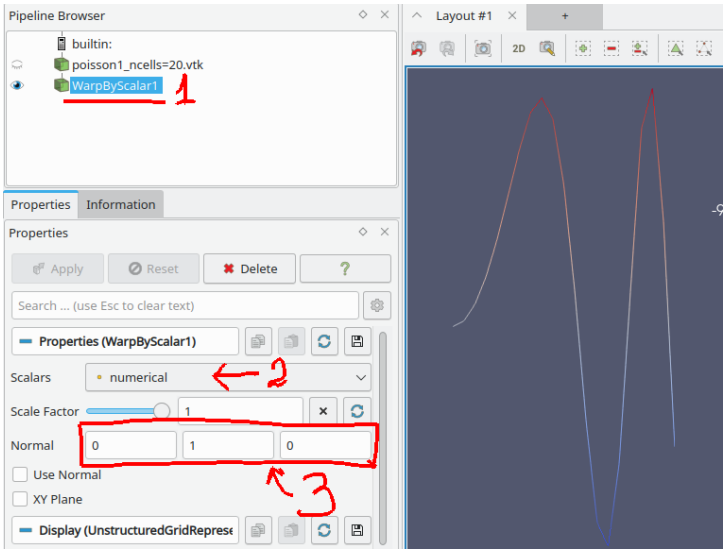
Заданные на сетке данные паравью показывает цветом. Поэтому при загрузке одномерных сеток можно видеть картинку типа



Развернуть изображение в плоскость xy

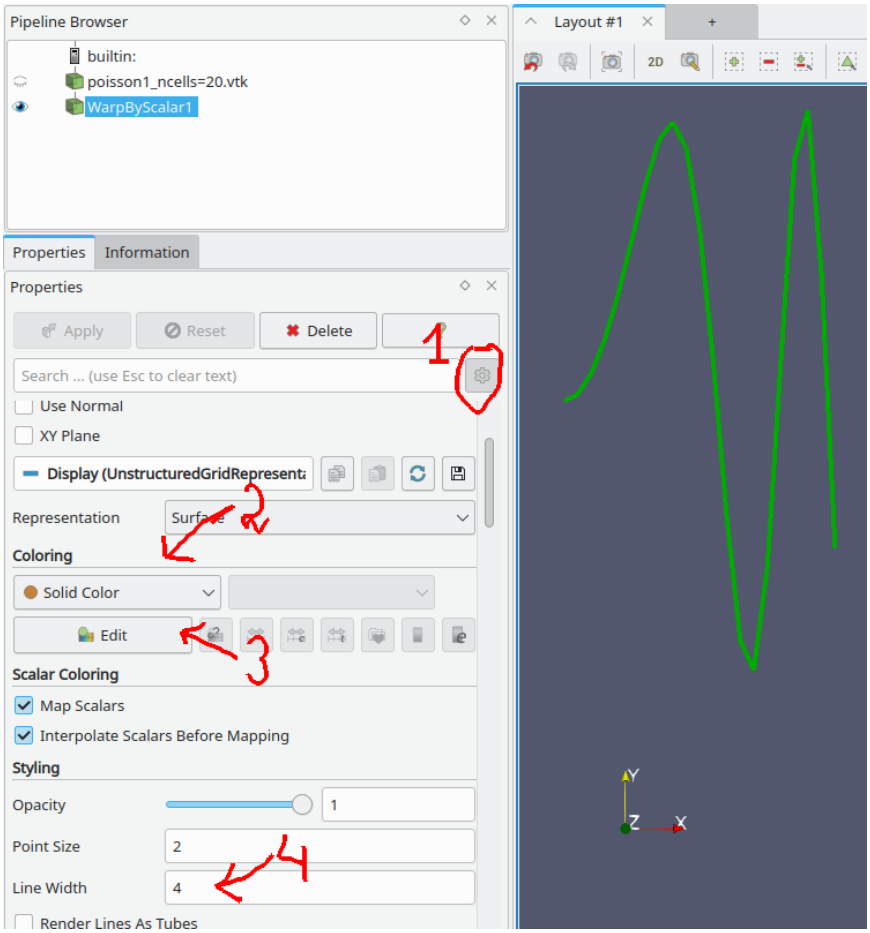
Отобразить данные в виде y-координаты Для того, что бы данные отображались в качестве значения по оси ординат, к загруженному файлу необходимо

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 y)



Цвет и толщина линии

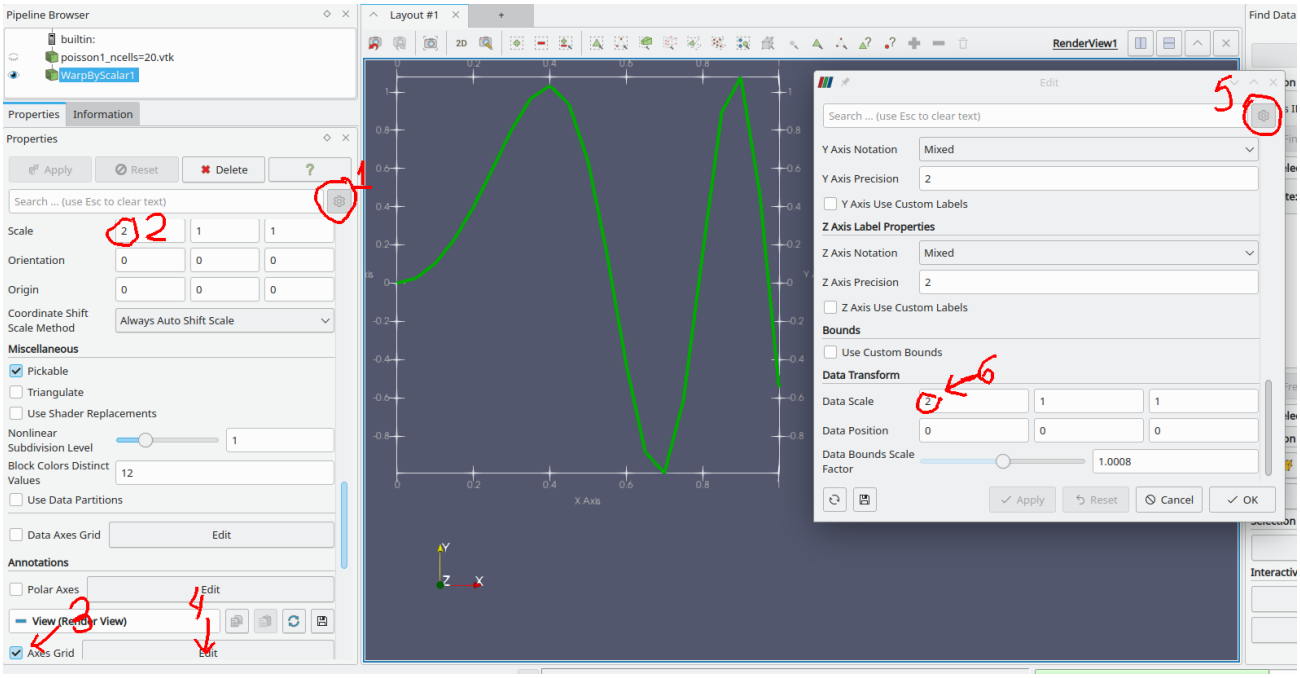
1. Включить подробные опции фильтра
2. Сменить стиль на `Solid Color`
3. В меню `Edit` выбрать желаемый цвет
4. В строке `Line Width` указать толщину линии



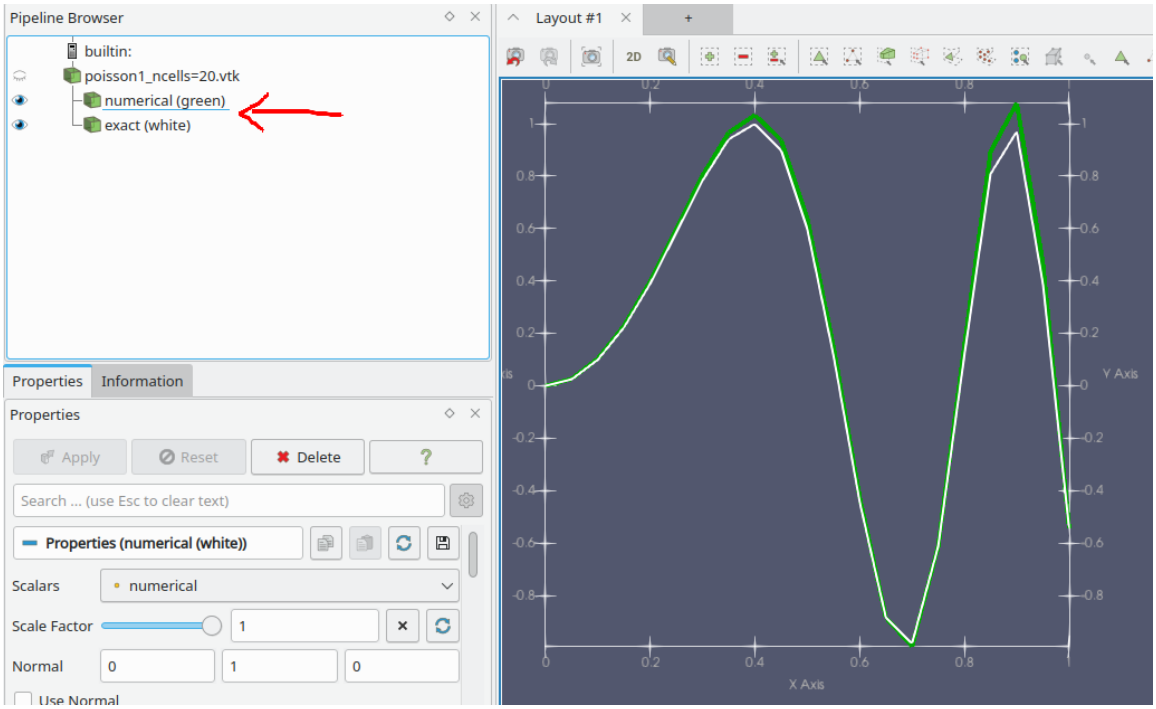
Настрока масштабов и отображение осей координат

1. Отметье 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 настроки
6. И также поставьте растяжение осей

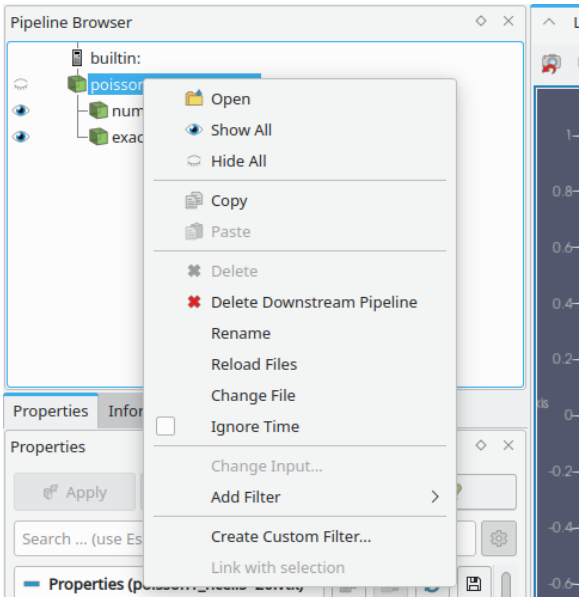
В случае, если масштабировать график не нужно, достаточно выполнить шаг 3.



Построение графиков для нескольких данных Если требуется нарисовать рядом несколько графиков для разных данных из одного файла, примените фильтр `Warp By Scalar` для этого файла ещё раз, изменив поле `Scalars` в настройке фильтра. Для наглядности измените имя узла в Pipeline Browser на осмысленные



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



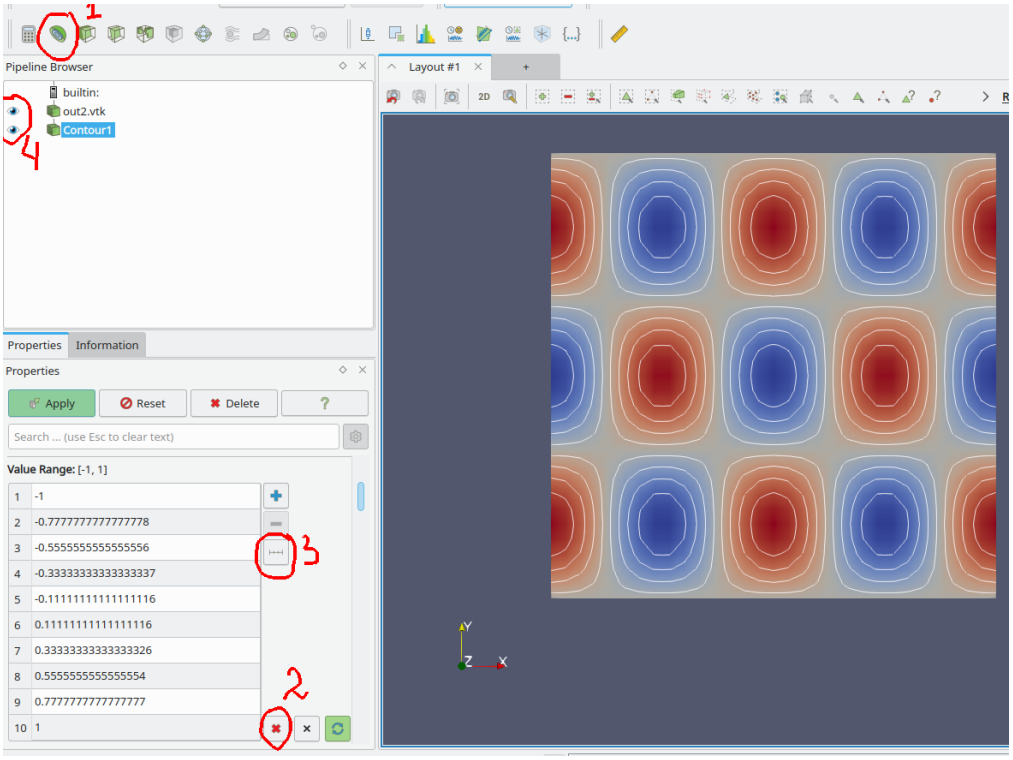
B.3.2 Изолинии для двумерного поля

1. Нажмите иконку `Contour` (или `Filters/Contour`) В настройках фильтра Contour by выберитее данные, по которым нужно строить изолинии.

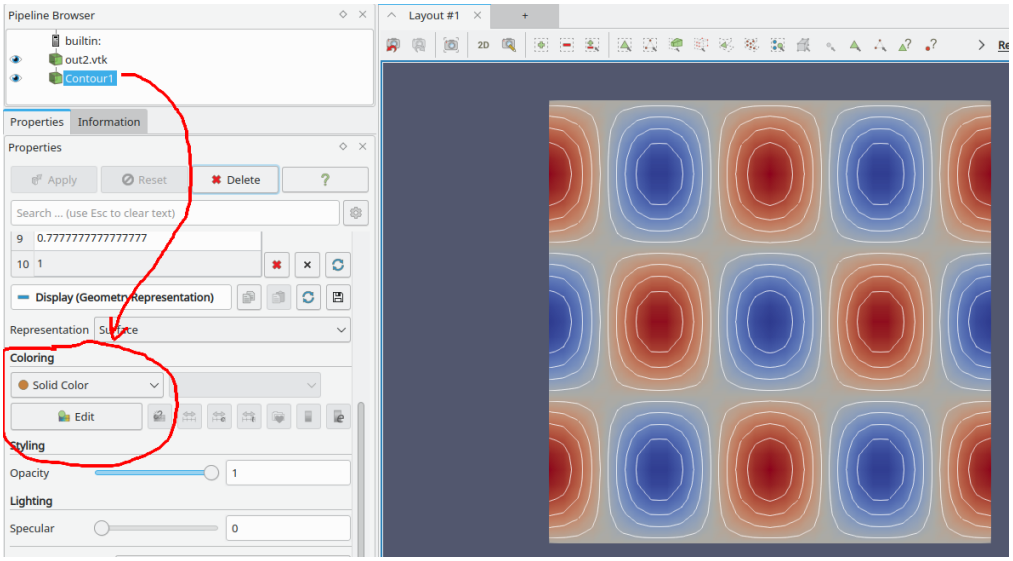
2. В настройках фильтра удалите все существующие записи о значениях для изолиний

3. Добавьте равномерные значения. В появившемся меню установите необходимое количество изолиний и их диапазон.

4. Если необходимо, включите одновременное отображения цветного поля и изолиний.



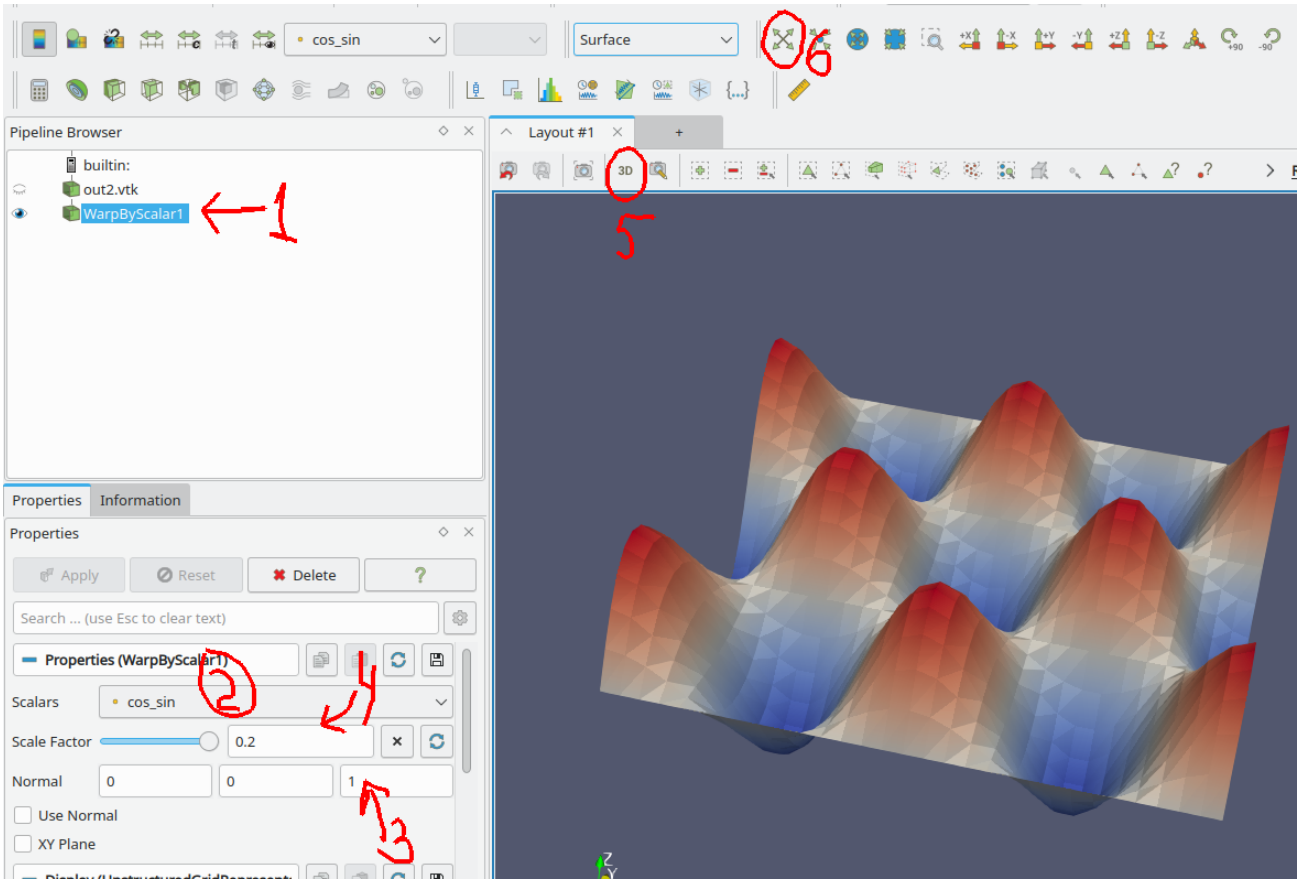
Задание цвета и толщины изолинии В случае, если нужно сделать изолинии одного цвета, установите поле `Coloring/Solid color` в настройках фильтра. Там же в меню `Edit` можно выбрать цвет. Для установления толщины линии включите подробные настройки и найдите там опцию `Styling/Line Width`.



B.3.3 Данные на двумерных сетках в виде поверхности

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

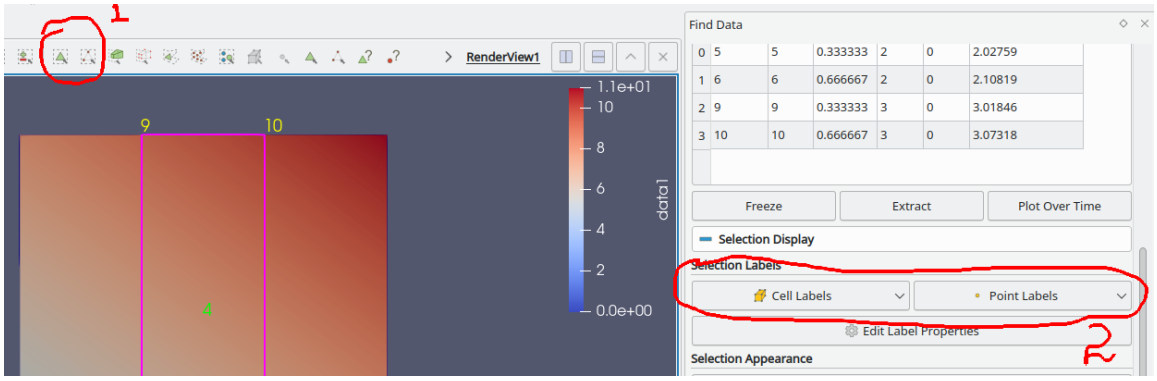
1. Включите фильтр `Filters/Warp By Scalar`
2. В настройках фильтра установите данные, которые будут проектироваться на координату z
3. Установите нормаль для проецирования (ось z)
4. Если нужно, выберите масштабирования для этой координаты
5. После нажатия `Apply` включите трёхмерное отображение
6.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



B.3.4 Числовых 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`.



B.3.5 Векторные поля

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

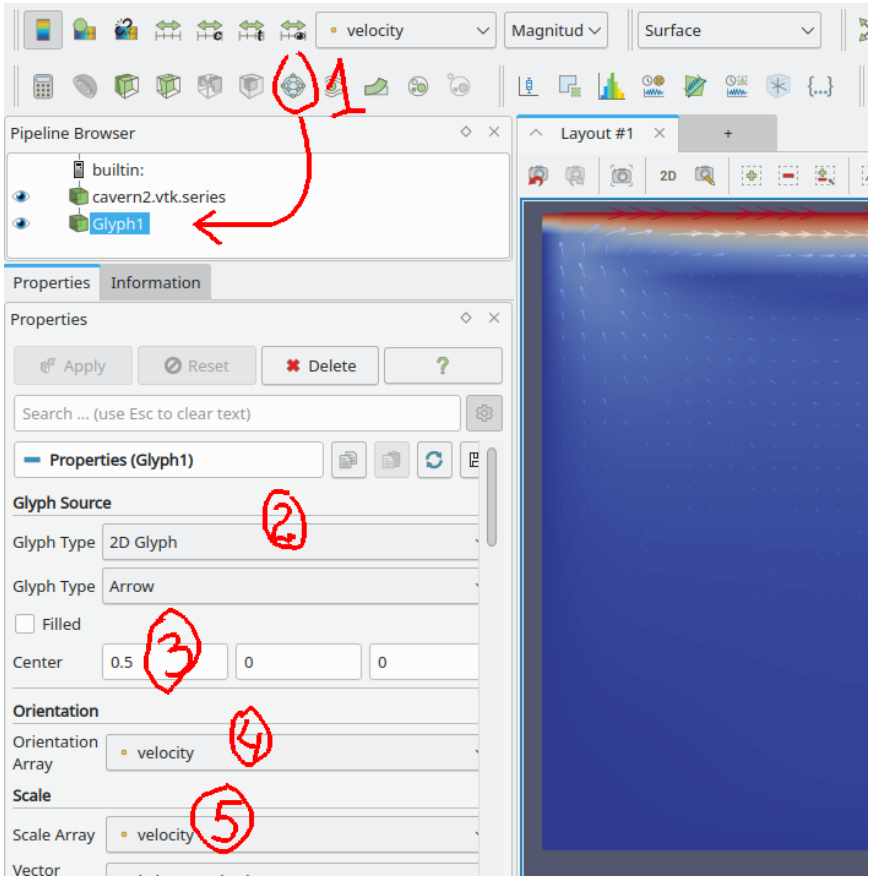
1. Создать фильтр `Glyph`

2. Задать двумерный тип стрелки

3. Сместить центр стрелки, чтобы она исходила из точки, к которой приписана

4. Отметить необходимое векторное поле в качестве ориентации

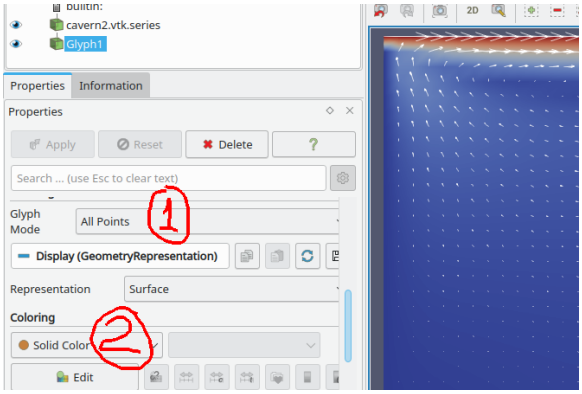
5. Отметить необходимое векторное поле для масштабирования Нажать `Apply`.



Настройка отображения стрелок

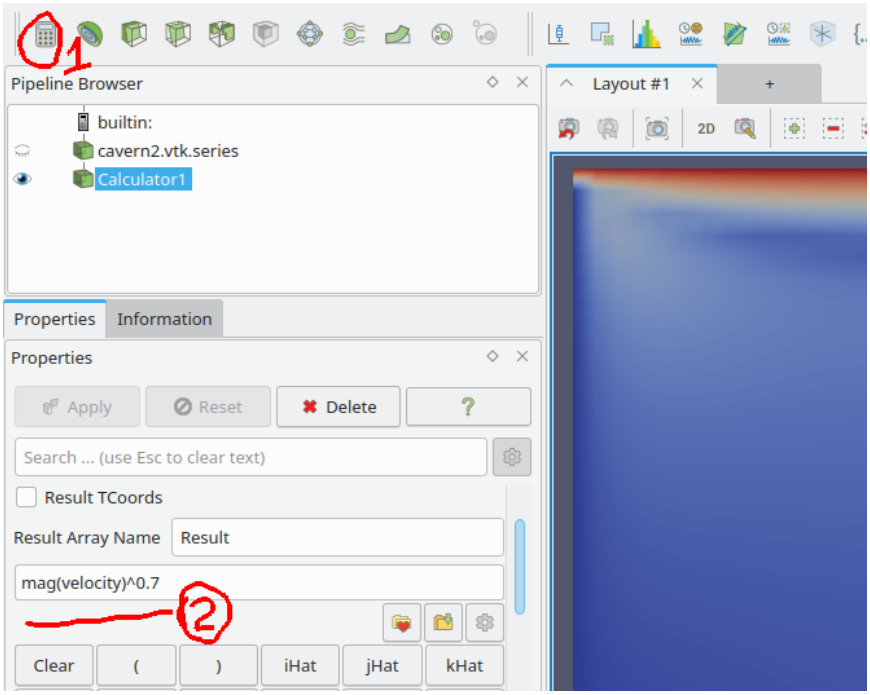
1. Выбрать необходимый `Glyph-mode`. Если сетка небольшая, то можно `All Points`.

2. Установить белый цвет для стрелок. Нажать Apply.



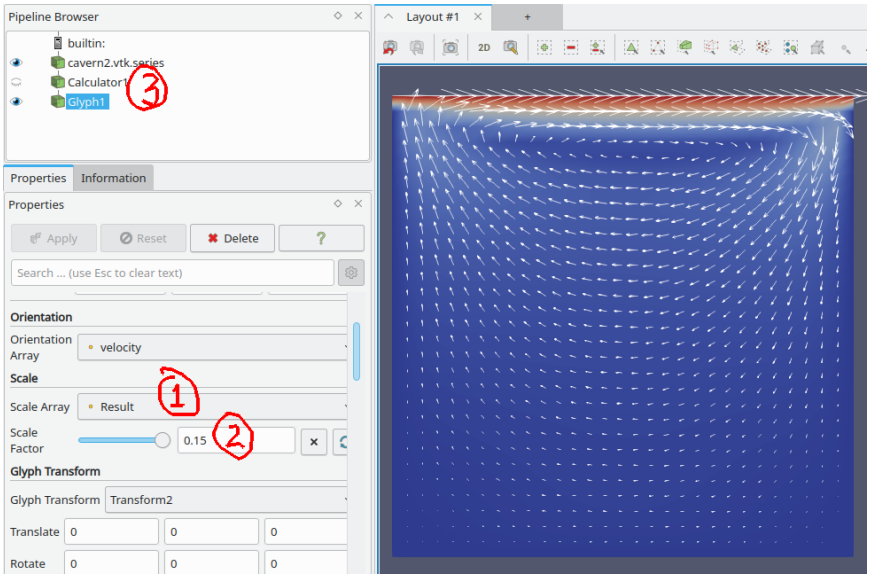
Уменьшения разброса по длине стрелок Если разброс по длинам стрелок слишком велик, его можно подравнять, введя новую функцию $|\mathbf{v}|^\alpha$ – длина вектора в степени меньше единицы (например, $\alpha = 0.7$). Такую функцию можно создать через калькулятор

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



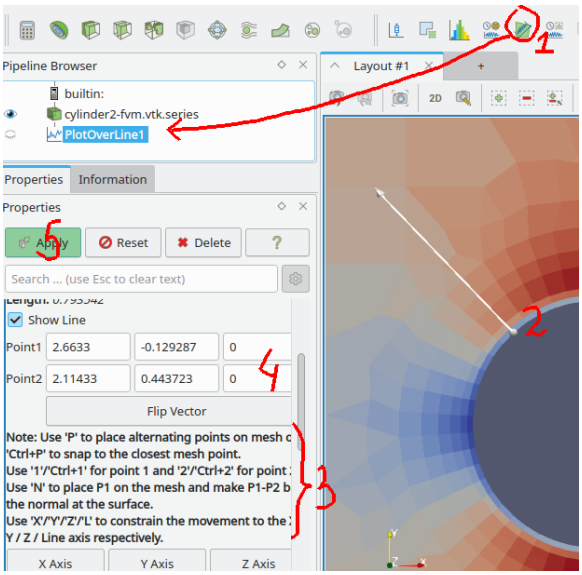
B.3.6 Значение функции вдоль линии

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

3. Можно использовать привязку к узлам сетки с помощью горячих клавиш (в подсказках написано)

4. Можно установить координаты руками в соответствующем поле. Для двумерных задач проследить, что координата Z равна нулю

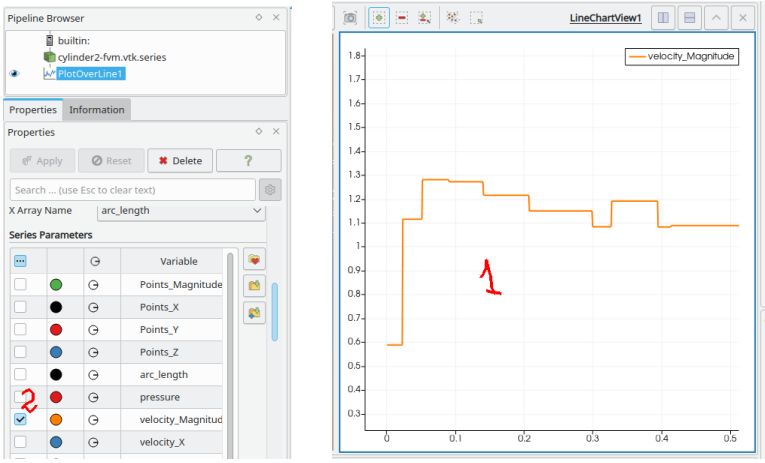
5. Нажать `Apply`



Настройка графика

1. После установок появится дополнительное окно типа `Line Chart View` с нарисованным графиком.

2. Сделав это окно активным в настройках фильтра `PlotOverLine` можно выбрать, какие поля рисовать (`Series Parameters`)

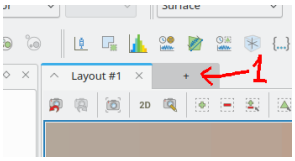


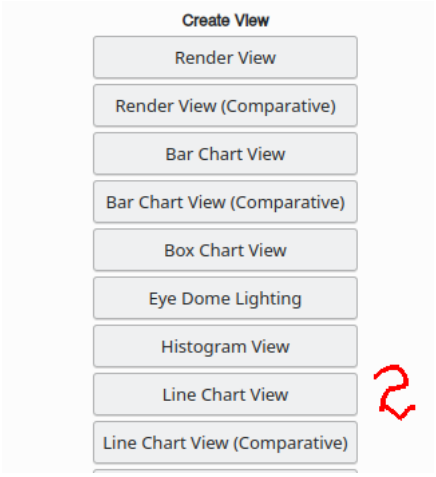
Отрисовка в отдельном окне

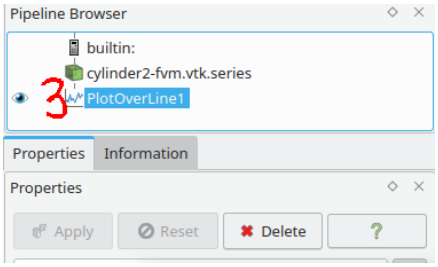
1. Открыть новую вкладку

2. Выбрать `Line Chart View`

3. Выбрать предварительно созданный фильтр с одномерным графиком


Layout #1
1


Create View
Render View
Render View (Comparative)
Bar Chart View
Bar Chart View (Comparative)
Box Chart View
Eye Dome Lighting
Histogram View
Line Chart View
Line Chart View (Comparative)
2


Pipeline Browser
builtin:
cylinder2-fvm.vtk.series
PlotOverLine1
3
Properties
Information
Properties
Apply
Reset
Delete

B.4 Hybmesh

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

B.4.1 Работа в Windows

Инсталлятор программы следует скачать по ссылке https://github.com/kalininei/HybMesh/releases и установить стандартным образом.

Для запуска скрипта построения `script.py` нужно открыть консоль, перейти в папку с нужным скриптом, оттуда выполнить (при условии, что программа была установлена в папку `C:\Program Files`):

```
> "C:\Program Files\HybMesh\bin\hybmesh.exe" -sx script.py
```

B.4.2 Работа в Linux

Версию для линукса нужно собирать из исходников. Либо, если собрать не получилось, можно строить сетки в Windows и переносить полученные vtk-файлы на рабочую систему.

Перед сборкой в систему необходимо установить dev-версии пакетов `suitesparse` и `libxml2`. Также должны быть доступны компилляторы
`gcc-c++` и `gcc-fortan` и `cmake`. Программа работает со скиптами python2. Лучше установить среду anaconda (https://docs.anaconda.com/free/anaconda/install/index.html) И в ней создать окружение с python-2.7:

```
> conda create -n py27 python=2.7   # создать среду с именем py27
> conda activate py27               # активировать среду py27
> pip install decorator             # установить пакет decorator
```

Сначала следует склонировать репозиторий в папку с репозиториями гита:

```
> cd D:/git_repos
> git clone https://github.com/kalininei/HybMesh
```

Поскольку программа не предназначена для запуска из под анаконды, в сборочные скрипты нужно внести некоторые изменения. В корневом сборочном файле `HybMesh/CMakeLists.txt` нужно закомментировать все строки в диапазоне

```
# ========================== Python check
....
# ========================== Windows installer options
```

а в файле `HybMesh/src/CMakeLists.txt` последнюю строку

```
#add_subdirectory(bindings)
```

Далее, находясь в корневой директории репозитория HybMesh, запустить сборку

```
> mkdir build
> cd build
> cmake .. -DCMAKE_BUILD_TYPE=Release
> make -j8
> sudo make install
```

Для запуска скриптов нужно создать скрипт-прокладку

```
import sys
sys.path.append("/path/to/HybMesh/src/py/")  # вставить полный путь к Hybmesh/src/py
execfile(sys.argv[1])
```

и сохранить его в любое место. Например в `path/to/HybMesh/hybmesh.py`.

Для запуска скрипта построения сетки следует перейти в папку, где находится нужный скрипт `script.py`, убедится, что анаконда работает в нужной среде (то есть `conda activate py27` был вызван), и запустить

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