Содержание

1 Сеточное решение уравнения Пуассона

1.1 Постановка задачи

Будем рассматривать многомерное дифференциальное уравнение в области Ω:

$$-\nabla \cdot (\lambda(\mathbf{x})\nabla u) = f(\mathbf{x}), \quad \mathbf{x} \in \Omega \tag{1.1}$$

Оператор в левой части (оператор Лапласа) описывает физический процесс диффузии с коэффициентом диффузии λ. Это уравнение (с нулевой правой частью) используют в частности для расчёта распределения температуры в однородном твёрдом теле. В этом случае коэффиент λ называют коэффициентом теплопроводности, а за счёт ненулевой f можно задавать дополнительные внутренние источники тепла.

В простом случае постоянного коэффициент диффузии ($\lambda = \text{const}$) его можно вынести из под дивергеции и отнести в правую часть. Тогда уравнение упростится до однородного вида:

$$-\nabla^2 u = f(\mathbf{x}), \quad \mathbf{x} \in \Omega \tag{1.2}$$

Далее на границе области расчёта $\partial\Omega$ рассмотрим несколько типов граничных условий.

1.1.1 Граничные условия первого рода

Также известны как граничные условия Дирихле. На границе $\partial\Omega_I$ задано точное значение искомой функции u^Γ:

$$u = u^\Gamma(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega_I \tag{1.3}$$

В аналогии задачи теплопроводности это условие можно трактовать как условие заданной на стенке температуры.

1.1.2 Граничные условия второго рода

Также известны как граничные условия Неймана. На границе $\partial\Omega_{II}$ задано значение нормальной производной искомой функции q:

$$-\lambda(\mathbf{x})\frac{\partial u}{\partial n} = q(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega_{II} \tag{1.4}$$

В аналогии задачи теплопроводности это условие можно трактовать как условие заданного на стенке теплового потока.

1.1.3 Граничные условия третьего рода

Также известны как граничные условия Робэна. На границе $\partial\Omega_{III}$ задано линейное соотношение значений функции и нормальной производной:

$$-\lambda(\mathbf{x})\frac{\partial u}{\partial n} = \alpha(\mathbf{x})u + \beta(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega_{III}. \tag{1.5}$$

При постановке задачи теплопроводности это условие часто записывают в виде

$$-\lambda(\mathbf{x})\frac{\partial u}{\partial n} = \alpha(\mathbf{x})\left(u - u^0\right).$$

где известное значение u^0 называют температурой окружающей стреды. Такое условие называют условием конвективной теплопроводности или условием Ньютона–Рихмана. Для приведения этого условия к исходному виду (1.5) достаточно положить $\beta = -\alpha u^0$.

1.1.3.1 Об универсальности условий третьего рода

Условия (1.3), (1.4) можно свести к условиям (1.5) при правильном подборе коэффициентов α и β. Так для условий второго рода нужно положить $\alpha = 0$, $\beta = q$. А для условий первого рода: $\alpha = \varepsilon^{-1}$, $\beta = -\alpha u^\Gamma$, где $\varepsilon \to 0$ – некоторое очень малое положительное число.

1.1.4 Периодические граничные условия

Необходимость в таких условиях возникает при расчёте физических процессов около периодических структур: решёток, лопастей, рядов скважин, оребрения нагревателя и т.п. В этом случае из исходную большую область расчёта представляют как бесконечную последовательность однотипных ячеек периодичности, в каждой из которых решения полностью идентичны (в более сложных вариантах – сдвинуты на константу).

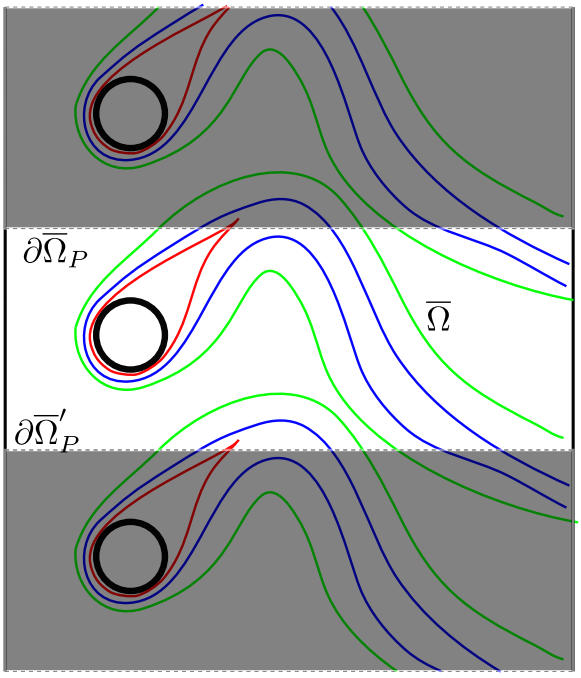


Рис. 1: Ячейка периодичности в задаче обтекания бесконечной решётки

На рис. 1 представлен пример области с выделенной ячейкой периодичности $\overline{\Omega}$ (незатенённая область). Изолинии можно трактовать как изотермы решения задачи о нестационарном обтекании решётки нагревателя (поле температур в этом случае описывается более сложным уравнением, чем (1.1) и приведено тут только для иллюстрации периодичности).

Пара периодических границ обозначена через $\partial\overline{\Omega}_P$ и $\partial\overline{\Omega}'_P$. Пусть эти границы топологически эквавалентны, то есть для любой точки $\mathbf{x} \in \partial\overline{\Omega}_P$ существует взаимнооднозначная точка $\mathbf{x}' \in$

$\partial\overline{\Omega}'_P$. Для того, чтобы решение за этими границами точно соответствовало решению внутри ячейки периодичности необходимо задать равенство значений и производных любого порядка:

$$\begin{cases} u(\mathbf{x}) = u(\mathbf{x}'), \\ \left.\dfrac{\partial^k u}{\partial n^k}\right|_{\mathbf{x}} = -\left.\dfrac{\partial^k u}{\partial n^k}\right|_{\mathbf{x}'} \end{cases} \quad \mathbf{x} \in \partial\overline{\Omega}_P, \quad \mathbf{x}' \in \partial\overline{\Omega}'_P, \quad \forall k. \tag{1.6}$$

Здесь под n подразумевается внешняя к ячейке периодичности нормаль, поэтому в правой части условия для производных стоит минус.

1.2 Метод конечных разностей

Рассмотрим задачу (1.2), (1.3) в упрощённой одномерной постановке:

$$-\frac{\partial^2 u}{\partial x^2} = f(x) \tag{1.7}$$

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

$$\begin{cases} u(a) = u_a, \\ u(b) = u_b. \end{cases} \tag{1.8}$$

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.

1.2.1 Метод решения

1.2.1.1 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 (1.7) имеет вид

$$\frac{-u_{i-1} + 2u_i - u_{i+1}}{h^2} = f_i, \qquad i = \overline{1, N-1}. \tag{1.9}$$

Здесь $\{f_i\}$ – известный сеточный вектор, определяемый через известную аналитическую функцию $f(x)$ в правой части уравнения (1.7) как

$$f_i = f(x_i). \tag{1.10}$$

Аппроксимация граничных условий (1.8) первого рода даёт дополнительные сеточные уравнения для граничных узлов

$$\begin{aligned} u_0 &= u_a, \\ u_N &= u_b \end{aligned} \tag{1.11}$$

Линейные уравнения (1.9), (1.11) составляют систему вида

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

с матричными коэффициентами

$$A_{ij} = \begin{cases} 1, & i = 0,\ j = 0; \\ 2/h^2, & i = \overline{1, N-1},\ j = i; \\ -1/h^2, & i = \overline{1, N-1},\ j = i-1; \\ -1/h^2, & i = \overline{1, N-1},\ j = i+1; \\ 1, & i = N,\ j = N; \\ 0, & \text{иначе.} \end{cases} \tag{1.12}$$

и правой частью

$$b_i = \begin{cases} u_a, & i = 0; \\ u_b, & i = N; \\ f_i, & i = \overline{1, N-1}. \end{cases} \tag{1.13}$$

Искомый вектор находится путём решения этой системы.

1.2.1.2 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 (1.7), (1.8).

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

$$||y(\mathbf{x})||_2 = \sqrt{\frac{1}{|D|} \int_D y(\mathbf{x})^2 \, d\mathbf{x}}. \tag{1.14}$$

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

$$||\{u_i - u^e(x_i)\}||_2 = \sqrt{\frac{1}{b-a} \sum_{i=0}^{N} w_i \left(u_i - u_i^e\right)^2}. \tag{1.15}$$

Пример программы, реализующей этот алгоритм смотри в п. B.1.

1.3 Метод конечных объёмов

Будем рассматривать задачу в многомерной постановке (1.2), (1.3).

1.3.1 Конечнообъёмная сетка

Разобъём область численного решения на непересекающиеся подобласти E_i, $i = \overline{0, N-1}$, а её границу $\partial\Omega$ на грани Γ_s, $s = \overline{0, N^\Gamma - 1}$ (рис. 2). Введем следующие сеточные примитивы:

- E_i – ячейка сетки,
- Γ_s – граничная грань,
- $\mathbf{c}_i$ – центр (масс) ячейки,
- $\mathbf{g}_s$ – центр (масс) грани Γ_s,
- γ_{ij} – внутренняя грань между i-ой и j-ой ячейками,

Будем считать, что ячейки сетки выпуклые, а грани – плоские.

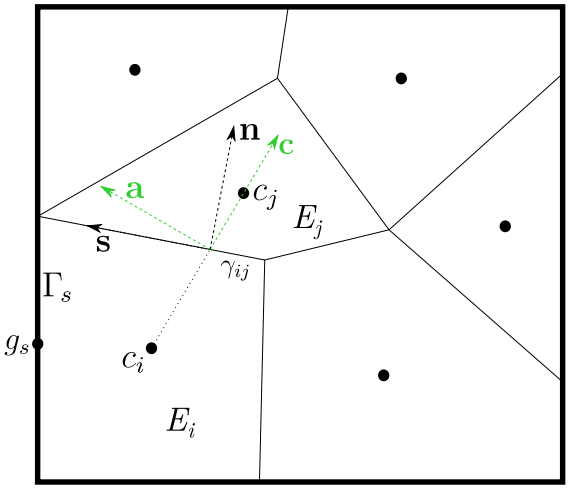


Рис. 2: Конечнообъёмная сетка

1.3.2 Конечнообъёмная аппроксимация

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 (C.13). Получим

$$-\int\limits_{\partial E_i} \frac{\partial u}{\partial n} \, d\mathbf{x} = \int\limits_{E_i} f \, d\mathbf{x}. \tag{1.16}$$

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 инцидентных ей граней Γ_s, лежащих на внешней границе расчётной области Ω. Тогда интеграл по общей границе ячейки распишется через сумму интегралов по плоским поверхностям

$$\int\limits_{\partial E_i} \frac{\partial u}{\partial n}\, ds = \sum_{j \in \mathrm{J}_i} \int\limits_{\gamma_{ij}} \frac{\partial u}{\partial n}\, ds + \sum_{s \in \mathrm{I}_i} \int\limits_{\Gamma_s} \frac{\partial u}{\partial n}\, ds.$$

Введены следующие обозначения множества индексов: J_i – индексы ячеек, соседних (имеющих общую грань) с текущей ячейкой i, I_i – индексы граничных граней первого рода, инцидентных ячейке E_i. Аппроксимирум производную $\partial u/\partial n$ на каждой из граней константой. Тогда её можно вынести из под интегралов и предыдущее выражение записать в виде

$$\int\limits_{\partial E_i} \frac{\partial u}{\partial n}\, ds \approx \sum_{j \in \mathrm{J}_i} |\gamma_{ij}| \left(\frac{\partial u}{\partial n}\right)_{\gamma_{ij}} + \sum_{s \in \mathrm{I}_i} |\Gamma_s| \left(\frac{\partial u}{\partial n}\right)_{\Gamma_s} \tag{1.17}$$

Аналогично, анализируя интеграл правой части (1.16), приблизим значение функции правой части f внутри элемента E_i константой f_i, которую отнесём к центру элемента. Тогда

$$\int\limits_{E_i} f\, d\mathbf{x} \approx f_i\, |E_i|\,. \tag{1.18}$$

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 (1.18):

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

1.3.2.1 Обработка внутренних граней

Для начала будем рассматривать сетки, в которых вектора $\mathbf{c}$, соединяющие центры ячеек (зедёные вектора на рис. 2), 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

$$\left(\frac{\partial u}{\partial n}\right)_{\gamma_{ij}} \approx \frac{\partial u}{\partial c} \approx \frac{u_j - u_i}{h_{ij}}, \quad h_{ij} = |\mathbf{c}_j - \mathbf{c}_i|. \tag{1.19}$$

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 (1.19) является симметричной разностью и, поэтому, имеет второй порядок аппроксимации.

1.3.2.2 Учёт граничных условий первого рода

Для вычисления второго слагаемого в правой части (1.17) следует расписать значение нормальной к границе производной вида

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

Это делается с помощью граничных условий.

Пусть в центре $\mathbf{g}_s$ грани Γ_s задано значение искомой функции (1.3):

$$u(\mathbf{g}_s) = u_s^{\Gamma}. \tag{1.20}$$

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 (1.19):

$$\left(\frac{\partial u}{\partial n}\right)_{\Gamma_s} \approx \frac{u_s^{\Gamma} - u_i}{h_{is}^{\Gamma}}, \quad h_{is}^{\Gamma} = |\mathbf{g}_s - \mathbf{c}_i|. \tag{1.21}$$

1.3.3 Одномерный случай

Рассмотрим результат конечнообъёмной аппроксимации задачи (1.2) в одномерном случае (1.7) на равномерной сетке с шагом h (рис. 3).

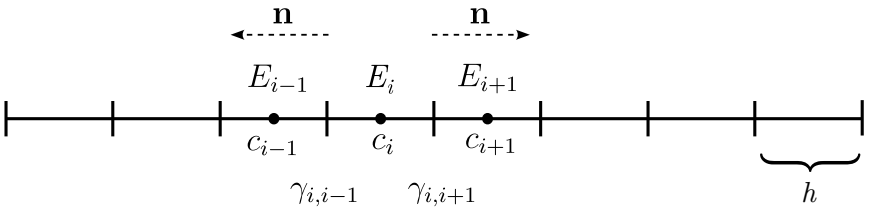


Рис. 3: Одномерная конечнообъёмная сетка

У внутренней ячейки i есть две границы: $\gamma_{i,i-1}$ и $\gamma_{i,i+1}$. Нормали по этим границам аппроксимируются по формулам (1.19):

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

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

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 (1.16), 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 $x = a$ задано условие первого рода, то соответствующее уравнение согласно (1.21) примет вид

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

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

1.3.4 Сборка системы линейных уравнений

Подставим все полученные аппроксимации (1.19), (1.21) в уравнение (1.16). Получим i-ое уравнение искомой системы уравнений относительно неизвестных u_i:

$$-\sum_{j \in \mathrm{J}_i} \frac{|\gamma_{ij}|}{h_{ij}} (u_j - u_i) - \sum_{s \in \mathrm{I}_i} \frac{|\Gamma_s|}{h_{is}^\Gamma} \left(u_s^\Gamma - u_i\right) = f_i |E_i|.$$

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:

$$\sum_{j \in \mathrm{J}_i} \frac{|\gamma_{ij}|}{h_{ij}} (u_i - u_j) + \sum_{s \in \mathrm{I}_i} \frac{|\Gamma_s|}{h_{is}^\Gamma} u_i = f_i |E_i| + \sum_{s \in \mathrm{I}_i} \frac{|\Gamma_s|}{h_{is}^\Gamma} u_s^\Gamma \tag{1.22}$$

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

$$Au = b.$$

Полученные в результате сборочных процедур матрицы являются разреженными – то есть большинство их элементов равно нулю. Полное хранение таких матриц в памяти невозможно, поэтому применяют специальные процедуры разреженного хранения (см. п. D.2).

Ниже приведён псеводкод для сборки СЛАУ. Перед началом процедур сборки левую правую часть нужно инициализировать нулями.

1.3.4.1 Алгоритм сборки в цикле по ячейкам

Матрицу A и правую часть b системы (1.22) 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} += v
    A_{ij} −= v
  endfor
  for $s \in$ bnd1(i) – цикл по граням ячейки i с условиями первого рода
    $v = |\Gamma_s| / h^{\Gamma}_{is}$
    A_{ii} += v
    b_i += $u^{\Gamma}_s v$
  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.

1.3.4.2 Алгоритм сборки в цикле по граням

Вместо общего цикла по ячейкам, будем использовать цикл по граням. В таком цикле коэффициенты потоков будут вычисляться один раз и вставляться сразу в две строки матрицы, соответствующие соседним с гранью ячейкам. Вложенных циклов в такой постановке удаётся избежать, потому что у грани есть только две соседние ячейки (в то время как у ячейки может быть произвольное количество соседних граней).

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

for $s \in$ internal – цикл по внутренним граням
  $i, j =$ nei_cells(s) – две ячейки, соседние с текущей гранью
  $v = |\gamma_{ij}|/h_{ij}$
  A_{ii} += v; A_{jj} += v – диагональные коэффициенты матрицы
  A_{ij} −= v; A_{ji} −= v – внедиагональные коэффициенты матрицы
endfor (1.23)

Граничные условия учитываются в отдельных циклах. Здесь будем учитывать, что у грани, принад-

лежащей границе области, есть только одна соседняя ячейка. Условия первого рода:

$$\begin{aligned}
&\textbf{for } s \in \text{bnd1} && \text{– грани с условиями первого рода}\\
&\quad i = \text{nei_cell(s)} && \text{– соседняя с граничной гранью ячейка}\\
&\quad v = |\Gamma_s| / h^{\Gamma}_{is}\\
&\quad A_{ii} \mathrel{+}= v\\
&\quad b_i \mathrel{+}= u^{\Gamma}_s v\\
&\textbf{endfor}
\end{aligned} \tag{1.24}$$

Первое слагаемое в правой части (1.22) учтём отдельным циклом:

$$\begin{aligned}
&\textbf{for } i = \overline{0, N-1} && \text{– цикл по ячейкам}\\
&\quad b_i \mathrel{+}= |E_i| f_i\\
&\textbf{endfor}
\end{aligned} \tag{1.25}$$

1.3.5 Расширенный набор точек коллокаций

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. Для написания аппроксимационных соотношений около границ будет удобно расширить набор точек коллокаций за счёт центров граничных граней.

Такой подход позволяет универсализировать подходы к аппроксимации перетоков через грани. То есть для каждой грани вместо использования разных алгоритмов для внутренних (1.23) и граничных (1.24) граней, нужно использовать универсальный алгоритм

$$\begin{aligned}
&\textbf{for } s \in \overline{0, N_f - 1} && \text{– цикл по всем граням}\\
&\quad i, j = \text{nei_colloc(s)} && \text{– инцидентные точки коллокаций}\\
&\quad v = |\gamma_{ij}| / h_{ij}\\
&\quad A_{ii} \mathrel{+}= v, \quad A_{ij} \mathrel{-}= v && \text{– } i\text{-ая строка}\\
&\quad A_{jj} \mathrel{+}= v, \quad A_{ji} \mathrel{-}= v && \text{– } j\text{-ая строка}\\
&\textbf{endfor}
\end{aligned} \tag{1.26}$$

Отметим, что эта процедура заполняет не только строки, соответствующие внутренним коллокациям, но и строки для граничных точек. Последние заполняются выражениями, соответствующие интегралам от нормальных производных

$$-\int\limits_{\Gamma_s} \frac{\partial u}{\partial n}\, ds \approx -\left.\frac{\partial u}{\partial n}\right|_{\mathbf{g}_s} |\Gamma_s| \;\approx \frac{u^{\Gamma}_s - u_j}{h^{\Gamma}_{is}} |\Gamma_s|,$$

где i – индекс ячейки, соседний с гранью Γ_s. Для задач с граничными условиями первого рода эти строки излишни и будут переписаны, но они окажутся полезными позднее, при учёте других типов граничных условий.

Строки матрицы, соответствующие граничным точкам коллокации, будут содержать аппроксими-

рованные граничные условия. Так, для граней с условиями первого рода будет аппроксимироваться непосредственно выражение (1.20). Алгоритмическом виде это примет вид

$$\begin{aligned}
&\textbf{for } s \in \text{bnd1} && \text{– грани с условиями первого рода}\\
&\quad j = \text{bnd_col(s)} && \text{– индекс точки коллокации, соответствующей грани}\\
&\quad A_{ij} = \delta_{ij} && \text{– единичная диагональ}\\
&\quad b_j = u_s^\Gamma\\
&\textbf{endfor}
\end{aligned} \tag{1.27}$$

Преимуществами такого подхода является:

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

1.3.5.1 Пример

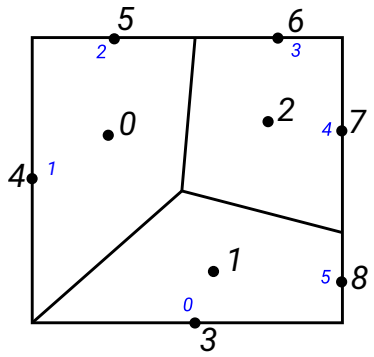


Рис. 4: Расширенный набор точек коллокации

На рис. 4. представлена конечнообъёмная сетка, содержащая три ячейки и девять граней. Индексация граничных граней обозначена синими цифрами. 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. 4 нужно решить уравнение Пуассона (1.2). Пусть на нижней и правой гранях задано условие первого рода: $u = C$.

Классический подход Согласно классическому методу конечных объёмов (п. 1.3.4.2) аппроксимация задачи в ячейке с индексом 1 будет иметь следующий вид

$$\frac{u_1 - u_0}{h_{10}}|\gamma_{10}| + \frac{u_1 - u_2}{h_{12}}|\gamma_{12}| + \frac{u_1 - C}{h_{10}^\Gamma}|\Gamma_0| + \frac{u_1 - C}{h_{15}^\Gamma}|\Gamma_5| = |E_1| f_1.$$

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

$$a_{10} = -\frac{|\gamma_{10}|}{h_{10}}, \quad a_{12} = -\frac{|\gamma_{12}|}{h_{12}}, \quad a_{11} = \frac{|\gamma_{10}|}{h_{10}} + \frac{|\gamma_{12}|}{h_{12}} + \frac{|\Gamma_0|}{h_{10}^\Gamma} + \frac{|\Gamma_5|}{h_{15}^\Gamma}.$$

Справа в 1-ой строке будет стоять

$$b_1 = |E_1| f_1 + \frac{C|\Gamma_0|}{h^{\Gamma}_{10}} + \frac{C|\Gamma_5|}{h^{\Gamma}_{15}}.$$

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, собранной согласно процедуре из п. 1.3.5, примет вид

$$\frac{u_1 - u_0}{h_{10}}|\gamma_{10}| + \frac{u_1 - u_2}{h_{12}}|\gamma_{12}| + \frac{u_1 - u_3}{h_{13}}|\gamma_{13}| + \frac{u_1 - u_8}{h_{18}}|\gamma_{18}| = |E_1| f_1.$$

Здесь введено соотвествие для граничных граней и расстояний:

$$\gamma_{13} = \Gamma_0, h_{13} = h^{\Gamma}_{10}, \gamma_{18} = \Gamma_5, h_{18} = h^{\Gamma}_{15}.$$

Остальные шесть уравнений будут представлять из себя аппроксимацию граничных условий для соответствующих граней. Так, 3-е и 8-е уравнение будет соответствовать условию первого рода:

$$u_3 = C, \qquad u_8 = C.$$

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

$$a_{10} = -\frac{|\gamma_{10}|}{h_{10}}, \quad a_{12} = -\frac{|\gamma_{12}|}{h_{12}}, \quad a_{13} = -\frac{|\gamma_{13}|}{h_{13}}, \quad a_{18} = -\frac{|\gamma_{18}|}{h_{18}}, \quad a_{11} = a_{10} + a_{12} + a_{13} + a_{18}, \quad b_1 = |E_1| f_1,$$

для 3-ей строки

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

для 8-ой строки

$$a_{88} = 1, \quad b_8 = C.$$

1.3.6 Граничные условия второго рода

Рассмотрим участок границы $\partial\Omega_{II}$ на котором заданы условия второго рода (1.4) при $\lambda = 1$. Проинтегрируем это условие по грани и получим уравнение для граничного узла коллокации:

$$-\int\limits_{\Gamma_s} \frac{\partial u}{\partial n}\, ds = \int\limits_{\Gamma_s} q(s)\, ds \approx |\Gamma_s| q(\mathbf{g}_s).$$

При сборке матрицы левой части согласно процедуре (1.26) левая часть этого уравнения уже содержит интеграл от нормальной производной. Тогда алгоритм сборки этого условия будет включать в

себя только подстановку q в правую часть:

$$\begin{array}{ll}\textbf{for } s \in \text{bnd2} & \text{– грани с условиями второго рода} \\ \quad j = \text{bnd\_col(s)} & \text{– индекс точки коллокации, соответствующей грани} \\ \quad b_j = |\Gamma_s| q(\mathbf{g}_s) & \\ \textbf{endfor} & \end{array} \tag{1.28}$$

1.3.7 Граничные условия третьего рода

Теперь рассмотрим участок границы $\partial\Omega_{III}$ с условиями (1.5) при $\lambda = 1$. Так же проинтегрируем его по s-ой грани

$$-\int\limits_{\Gamma_s} \frac{\partial u}{\partial n}\, ds = \int\limits_{\Gamma_s} \alpha(s)u + \beta(s)\, ds \approx |\Gamma_s| \left(\alpha(\mathbf{g}_s)u_s + \beta(\mathbf{g}_s)\right).$$

Перенесём слагаемое с неизвестной u_s в левую часть (то есть добавим коэффициент в диагональ матрицы), а β оставим справа. Тогда, после сборки матрицы левой части по процедуре (1.26), модифицируем матрицу и правую часть следующим образом:

$$\begin{array}{ll}\textbf{for } s \in \text{bnd3} & \text{– грани с условиями третьего рода} \\ \quad j = \text{bnd\_col(s)} & \text{– индекс точки коллокации, соответствующей грани} \\ \quad A_{jj} \mathrel{+}= |\Gamma_s| \alpha(\mathbf{g}_s) & \\ \quad b_j = |\Gamma_s| \beta(\mathbf{g}_s) & \\ \textbf{endfor} & \end{array} \tag{1.29}$$

1.3.8 Периодические граничные условия

Рассмотрим периодическую пару границ $\partial\Omega_P$, $\partial\Omega'_P$. Для того, чтобы такое условие можно было аппроксимировать сеточным методом, необходимо, чтобы сетка на границе $\partial\Omega_P$ в точности соотвествовала сетке на границе $\partial\Omega'_P$.

Естественный способ удовлетворить граничные уловия вида (1.6) – модифицировать таблицы связности сетки так, чтобы грани, лежащие на этих границах перестали быть граничными. То есть нужно убрать граничные точки коллокации, и добавить запись в таблицы связности “грань-ячейка”. Такая процедура требует специальной подстройки сеточных таблиц.

Чтобы этого избежать, можно работать без модификации сетки, но удовлетворить формальным математическим условиям (1.6). Поскольку конечнообъёмная аппроксимация уравнения Пуассона имеет не более чем второй порядок точности, достаточно записать это условие только для первой производной. Для периодической пары граничных граней с индексами s и s' это условие можно записать следующим образом:

$$u(\mathbf{g}_s) - u(\mathbf{g}_{s'}) = 0, \tag{1.30}$$

$$-\int\limits_{\Gamma_s} \frac{\partial u}{\partial n}\, ds - \int\limits_{\Gamma_{s'}} \frac{\partial u}{\partial n}\, ds = 0. \tag{1.31}$$

Первое из этих условий условий запишем в строке, соответствующей грани s, а второе - в строке для грани s'. При сборке (1.31) учтём, что предворительно проведённая процедура (1.26) собирает

входящие в него пару интегралов в строках для грани s и s' соответсвенно. Чтобы записать сумму этих интегралов, нужно просто суммировать эти строки матрицы. Тогда процедура примет следующий вид

$$
\begin{array}{ll}
\textbf{for } s, s' \in \text{periodic\_pairs} & \text{– периодические пары граней} \\
\quad i, j = \text{bnd\_col(s, s')} & \text{– индексы граничных точек коллокации} \\
\quad \textbf{for } k \in \overline{0, N + N^{\Gamma} - 1} & \text{– цикл по столбцам} \\
\qquad A_{jk} \mathrel{+}= A_{ik} & \text{– складываем строки } i + j \text{ (1.30)} \\
\qquad A_{ik} = 0 & \text{– зануляем строку } i \\
\quad \textbf{endfor} & \\
\quad A_{jj} \mathrel{+}= A_{ji}, \quad A_{ji} = 0 & \text{– усилим диагональ } A_{jj} \text{ (т.к } u_i = u_j) \\
\quad A_{ii} = 1, \quad A_{ij} = -1 & \text{– удовлетворим (1.31)} \\
\textbf{endfor} &
\end{array}
\tag{1.32}
$$

1.3.9 Учёт неортогональности сетки

Конечнообъёмная схема, описываемая уравнениями (1.16), (1.17), не использует в своем выводе аппроксимационных соотношений, и поэтому является точной. Погрешность аппроксимации вносится при расписывании нормальных производных на грани по разностным формулам: (1.19), (1.21) через значения скалярной функции в точках коллокации. Эти формулы имеют второй порядок аппроксимации в случае ортогональных сеток. Поэтому для таких сеток вся конечнообъёмная схема имеет второй порядок аппроксимации. Но если в сетке присутствуют скошенные ячейки, такие разностные соотношения дают только порядок точности. Чтобы сохранить второй порядок для скошенных сеток необходимо дополнительно учитывать изменения функции поперёк нормали.

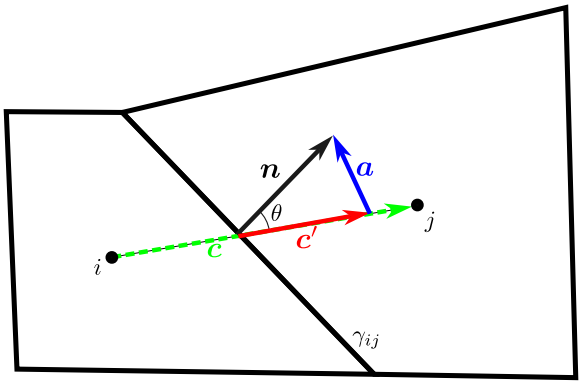


Рис. 5: Ячейка периодичности в задаче обтекания бесконечной решётки

Рассмотрим вычисление нормальной производной на грани γ_{ij}, разделяющие точки коллокации i и j (рис. 5). При использовании подхода с расширенным набором точек коллокации не имеет значения, являются ли эти точки граничными коллокациями или внутренними. Пусть вектор $\mathbf{c}$ соединяет точки коллокации. За меру ортогональности примем значение угла θ между вектором единичной нормали $\mathbf{n}$ и вектором $\mathbf{c}$:

$$\cos\theta = \frac{\mathbf{n} \cdot \mathbf{c}}{|\mathbf{c}|}.$$

Выделим некоторый вектор $\mathbf{c}'$, коллинеарный вектору $\mathbf{c}$. И распишем вектор нормали как

$$\mathbf{n} = \mathbf{c}' + \mathbf{a}. \tag{1.33}$$

Тогда

$$\frac{\partial u}{\partial n} = \nabla u \cdot \mathbf{n} = \nabla u \cdot \mathbf{c}' + \nabla u \cdot \mathbf{a} = |\mathbf{c}'| \nabla u \cdot \frac{\mathbf{c}}{|\mathbf{c}|} + \nabla u \cdot \mathbf{a}. \tag{1.34}$$

Первое слагаемое – ортогональное приближение, которое с точностью до множителя $|\mathbf{c}'|$ равно ранее вычисленным по разностным формулам (1.19), (1.21). Второе – поправка на скошенность.

Для реализации алгоритма с учётом этой поправки нужно решить следующие подзадачи:

- Задать длину $|\mathbf{c}'|$ (п. 1.3.9.1),
- Задать способ определения касательной производной $\nabla u \cdot \mathbf{a}$ (п. 1.3.9.2),
- Собрать полученные соотношения в результирующую систему уравнений (п. 1.3.9.3).

1.3.9.1 Методы разложения нормали

Рассмотрим различные варианты записи единичной нормали $\mathbf{n}$ в форме (1.33). Вектор $\mathbf{c}'$ сонаправлен заданному вектору $\mathbf{c}$, а вектор $\mathbf{a}$ может быть получен после определения $\mathbf{c}'$:

$$\mathbf{c}' = |\mathbf{c}'| \frac{\mathbf{c}}{|\mathbf{c}|},$$
$$\mathbf{a} = \mathbf{n} - \mathbf{c}'.$$

То есть для конкретизации разложения (1.33) нужно задать длину вектора $\mathbf{c}'$. Рассмотрим три варианта её определения.

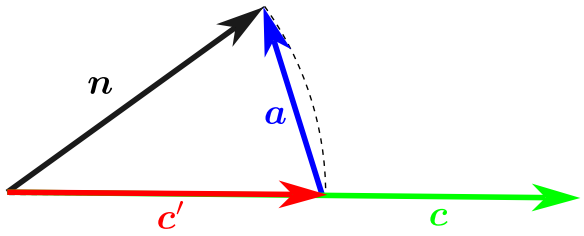


Рис. 6: Определение $\mathbf{c}'$ методом поворота

Поворот Положим $|\mathbf{c}'| = 1$. То есть положим длину искомого вектора равной длине единичной нормали $\mathbf{n}$ или повернём нормаль на угол θ (см. рис. 6).

$$\mathbf{c}' = \frac{\mathbf{c}}{|\mathbf{c}|}.$$

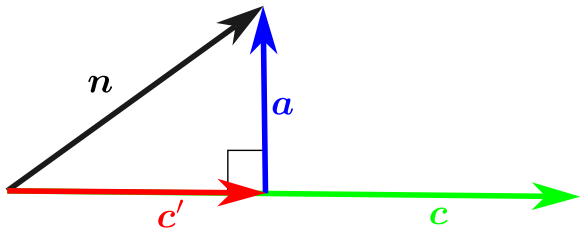


Рис. 7: Определение $\mathbf{c}'$ методом проекции

Проекция Определим вектор $\mathbf{c}'$ как проекцию вектора нормали на направление $\mathbf{c}$ (см. рис. 7). Тогда

$$|\mathbf{c}'| = \cos\theta = \frac{\mathbf{n}\cdot\mathbf{c}}{|\mathbf{c}|},$$
$$\mathbf{c}' = \frac{\mathbf{n}\cdot\mathbf{c}}{|\mathbf{c}|^2}\mathbf{c}$$

В этом случае $|\mathbf{c}'| \leq 1$. Таким образом, при записи нормальной производной (1.34) слагаемое с ортогональным приближением используется с коэффициентом, меньшим единицы. Поэтому этот метод можно назвать методом нижней релаксации.

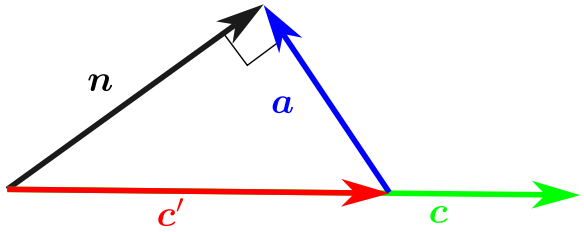


Рис. 8: Определение $\mathbf{c}'$ через обратную проекцию

Обратная проекция Наоборот, опустим перпендикуляр с направления $\mathbf{c}$ на нормаль (см. рис. 8):

$$|\mathbf{c}'| = \frac{1}{\cos\theta} = \frac{|\mathbf{c}|}{\mathbf{n}\cdot\mathbf{c}},$$
$$\mathbf{c}' = \frac{\mathbf{c}}{\mathbf{n}\cdot\mathbf{c}}.$$

Тогда, напротив $|\mathbf{c}'| \geq 1$, поэтому этот метод можно назвать методом верхней релаксации. Отметим, что в этом случае вектор $\mathbf{a}$ будет параллелен грани γ_{ij}.

1.3.9.2 Методы вычисления касательной производной

Рассмотрим способы получить второе слагаемое из разложения (1.34). Напомним, что это разложение записывается для значения производной на грани конечноэлементной сетки:

$$(\nabla u \cdot \mathbf{a})_{\gamma_{ij}}$$

При этом функция u задана своими значениями в точках коллокации, а вектор $\mathbf{a}$ известен (см. п. 1.3.9.1)

Определение через значение градиента в точках коллокации Пусть градиент ∇u также задан в точках коллокации. Тогда значение на грани γ_{ij} можно записать через линейную коминацию этих значений:

$$(\nabla u)_{\gamma_{ij}} \approx w_i (\nabla u)_i + w_j (\nabla u)_j, \qquad w_i + w_j = 1.$$

Пусть i-ая точка коллокации граничная, тогда $w_i = 1, w_j = 0$. Если же обе точки внутренние, то в простейшем случае можно взять $w_i = w_j = 0.5$. В более сложных случаях можно подобрать весовые коэффициенты в зависимости от расстояние точки коллокации до грани.

Для определения градиента в точках коллокации применим алгорим опрелеления градинетов в центрах ячеек (см. п. 1.3.10). К граничным точкам коллокации припишем значение градиента из инцидентной с ней ячейкой.

Таким образом, пусть известны значение $(\nabla u)_i$ во внутренних точках коллокации. Тогда значение градиента на границе будет равно

$$(\nabla u)_{\gamma_{ij}} = \begin{cases} \frac{1}{2}(\nabla u)_i + \frac{1}{2}(\nabla u)_j, & i \text{ и } j \text{ – внутренние точки коллокации} \\ (\nabla u)_i, & j \text{ – граничная точка коллокации} \\ (\nabla u)_j & i \text{ – граничная точка коллокации.} \end{cases}$$

Прямая интерполяция TODO

1.3.9.3 Учёт поправки при сборке СЛАУ

Подставим в левую часть выражения (1.16) разложение для нормальной с учётом поправки на скошенность (1.34)

$$-\int\limits_{\partial E_i} \left(|\mathbf{c}'| \frac{\partial u}{\partial c} + \nabla u \cdot \mathbf{a} \right) d\mathbf{x} = \int\limits_{E_i} f \, d\mathbf{x}.$$

Первое слагаемое в левой части – тоже самое слагаемое, которое использовалось в ортогональном приближении. Оно вычисляется по формулам (1.19), (1.21). Второе слагаемое – поправка на ортогональность, вычисляется по процедурам, описанным в п. 1.3.9.2.

Явный учёт поправки Пусть нам известно некоторое приближение решения $\check{u}$. Тогда мы можем вычислить скалярный сеточный вектор градиетов для каждой грани конечнообъёмной сетки γ_s:

$$corr_s = (\nabla \cdot \check{u})_s \cdot \mathbf{a}_s \tag{1.35}$$

Используем это решение для вычисление поправки на скошеннсть и перенесём её вправо. Получим

$$-\int\limits_{\partial E_i} |\mathbf{c}'| \frac{\partial u}{\partial c} \, d\mathbf{x} = \int\limits_{E_i} f \, d\mathbf{x} + \sum_{s \in \mathrm{S_i}} corr_s |\gamma_s|.$$

Здесь $\mathrm{S_i}$ – индексы граней, инцидентных ячейке i.

При сборке матрицы левой части нужно внести изменения в процедуру (1.26), которая учтёт

множитель $|\mathbf{c}'|$:

$$
\begin{aligned}
&\textbf{for } s \in \overline{0, N_f - 1} && \text{– цикл по всем граням} \\
&\quad i, j = \text{nei\_colloc(s)} && \text{– инцидентные точки коллокаций} \\
&\quad c = |\mathbf{c}'|_s && \text{– поправка} \\
&\quad v = c\,|\gamma_{ij}|/h_{ij} && \\
&\quad A_{ii} \mathrel{+}= v, \quad A_{ij} \mathrel{-}= v && \text{– } i\text{-ая строка} \\
&\quad A_{jj} \mathrel{+}= v, \quad A_{ji} \mathrel{-}= v && \text{– } j\text{-ая строка} \\
&\textbf{endfor}
\end{aligned}
\tag{1.36}
$$

Слагаемое в правой части будет учитано в аналогичной процедуре:

$$
\begin{aligned}
&\textbf{for } s \in \overline{0, N_f - 1} && \text{– цикл по всем граням} \\
&\quad i, j = \text{nei\_colloc(s)} && \text{– инцидентные точки коллокаций} \\
&\quad v = \text{corr}_s|\gamma_{ij}| && \\
&\quad b_i \mathrel{+}= v && \\
&\quad b_j \mathrel{+}= v && \\
&\textbf{endfor}
\end{aligned}
\tag{1.37}
$$

Эта процедура так же правит значения для граничных точек коллокаций, поэтому дополнительная модификация процедур для граничных условий второго и третьего рода не требуется. Для периодических условий потребуется учесть суммирование строк после (1.32)

$$
\begin{aligned}
&\textbf{for } s, s' \in \text{periodic\_pairs} && \text{– периодические пары граней} \\
&\quad i, j = \text{bnd\_col(s, s')} && \text{– индексы граничных точек коллокации} \\
&\quad b_j \mathrel{+}= b_i && \\
&\quad b_i = 0 && \\
&\textbf{endfor}
\end{aligned}
\tag{1.38}
$$

Тогда итоговый алгоритм сборки будет иметь следующий вид:
Этап инициализации

1. По алгоритмам п. 1.3.9.1 расчитать значения $|\mathbf{c}'|$ и $\mathbf{a}$ для каждой грани конечного объёма,
2. Собрать матрицу левой части по процедуре (1.36)
3. Собрать вектор b^0 – базовую часть правого столбца СЛАУ. Для этого инициализировать его нулями, потом применить алгоритм (1.25)
4. Применить процедуры для граничных условий (1.27) – (1.29), (1.32)
5. Задать начальное приближение $\check{u}$

Итерация на этапе расчёта

1. По процедурам п. 1.3.9.2 посчитать значение градиента $\nabla\check{u}$ для каждой грани конечнообъёмной сетки,

2. Найти вектор $corr$ по формуле (1.35)

3. Инициализировать вектор правой части $b = b^0$ и далее добавить в него поправку согласно (1.37)

4. При наличии периодических условий применить процедуру (1.38)

5. Посчитать невязку $r = ||b - A\check{u}||$. Если она мала, выйти из цикла

6. Решить СЛАУ $Au = b$

7. Перейти на следующую итерацию $\check{u} = u$.

Неявный учёт поправки TODO

1.3.10 Вычисление градиентов в центрах ячеек

1.3.10.1 Метод Гаусса

TODO

1.3.10.2 Метод наименьших квадратов

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

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

Для двумерного случая можно записать:

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

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

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

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

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

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

и будем минимизировать её квадрат

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

Запишем условие экстремума как

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

Отсюда получим систему уравнений

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

Или, возвращаясь к матричной записи,

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

Полученная система имеет размерность 2×2 (или 3×3 в трёхмерном случае). Значение компонент градиента в точке коллокации запишется как её прямое решение:

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

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

1.3.11 Неоднородный коэффициент диффузии

Рассмотрим задачу в постановке (1.1). Применение конечнообъёмных проеобразований по аналогии с п. 1.3.2 даст следующие уравнения для конечного объёма $|E_i|$:

$$-\sum_j \lambda_{ij} \left(\frac{\partial u}{\partial n} \right)_{ij} |\gamma_{ij}| = |E_i| f_i \tag{1.39}$$

Здесь λ_{ij} – значение коэффициента диффузии на грани между точками коллокации i и j. При этом в конечнообъёмной схеме все неизвестные скалярные поля, в том числе коэффициент диффузии, заданы только в точках коллокаций. То есть задача состоит в том, чтобы зная λ_i, λ_j выразить λ_{ij}.

Простейшим выходом будет использовать среднее арифметическое:

$$\lambda_{ij} = \frac{\lambda_i + \lambda_j}{2} \tag{1.40}$$

Однако, это выражение не сохраняет порядок аппроксимации схемы. Для записи более точного выражения, запишем выражение потоков слева и справа от грани γ_{ij}. Для этого введём значение u_{ij} на

грани. В ортогональном приближении получим:

$$\lambda_i \left(\frac{\partial u}{\partial n}\right)_i \approx \lambda_i \frac{u_{ij} - u_i}{h_{ij}^-},$$
$$\lambda_j \left(\frac{\partial u}{\partial n}\right)_j \approx \lambda_j \frac{u_j - u_{ij}}{h_{ij}^+}.$$

Здесь h_{ij}^+, h_{ij}^- – доли расстояния h_{ij}, находящиеся в i-ой и j-ой ячейках, такие что

$$h_{ij}^+ + h_{ij}^- = h_{ij}.$$

Эти выражения равны друг другу и равны суммарному потоку, вычисляемому через λ_{ij}:

$$\lambda_{ij} \left(\frac{\partial u}{\partial n}\right)_{ij} = \lambda_{ij} \frac{u_j - u_i}{h_{ij}}. \tag{1.41}$$

Таким образом, мы получили систему из двух линейных уравнений относительно неизвестных λ_{ij}, u_{ij}. Выразим из это системы искомый коэффициент диффузии как среднее взвешенное гармоническое выражение

$$\lambda_{ij} = \frac{(h_{ij}^+ + h_{ij}^-)\lambda_i \lambda_j}{\lambda_i h_{ij}^+ + \lambda_j h_{ij}^-}. \tag{1.42}$$

которое в приближении $h_{ij}^+ \approx h_{ij}^-$ может быть упрощено до обычного среднегармонического

$$\lambda_{ij} = \frac{2\lambda_i \lambda_j}{\lambda_i + \lambda_j}. \tag{1.43}$$

Нетрудно показать, что вычисление нормальной производной с учётом поправки на ортогональность сохраняет эти выражения. Распишем поток слева и в центре с поправкой:

$$\lambda_i \left(\frac{\partial u}{\partial n}\right)_i = \lambda_i \left(\frac{u_{ij} - u_i}{h_{ij}^-} + (\nabla u)_i \cdot \mathbf{a}\right),$$
$$\lambda_{ij} \left(\frac{\partial u}{\partial n}\right)_{ij} = \lambda_{ij} \left(\frac{u_j - u_i}{h_{ij}} + (\nabla u)_{ij} \cdot \mathbf{a}\right).$$

Если вычислять градиент на грани как среднее взвешенное арифметическое:

$$(\nabla u)_{ij} = \frac{h_{ij}^- (\nabla u)_i + h_{ij}^+ (\nabla u)_j}{h_{ij}^+ + h_{ij}^-}$$

то формула (1.42) выполнится точно.

1.3.12 Аппроксимация нормальной производной с учётом логарифмической особенности

Рассмотрим уравнение Лапласа ($f = 0$) в двусвязной области, образованной внешним контуром и внутренним кругом с центром в точке $\mathbf{C}$ (рис. 9). Будем считать, что значение искомой функции на внутренней границе постоянно.

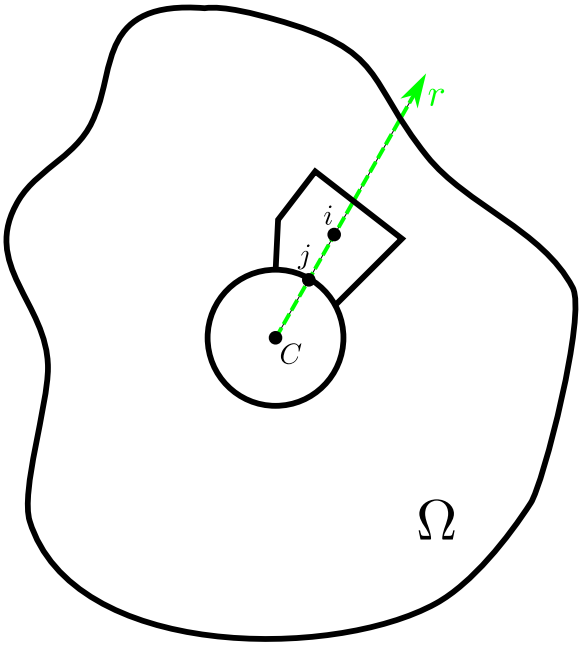


Рис. 9: Двусвязная область решения

Возьмём приграничную ячейку i и граничную точку коллокации j. Пусть точки $\mathbf{C}$, $\mathbf{c}_i$, $\mathbf{c}_j$ лежат на одной прямой. Координату вдоль этой прямой будем называть r. Будем считать, что граничное значение на внутреннем круге сильно отличается (в большую или меньшую сторону) от характерного значения u в области расчёта. Тогда в некотором приближении решении в окрестности внутреннего круга можно считать радиально симметричным: зависящим только от r, но не от угла поворота радиус-вектора $\mathbf{r}$. В приближении постоянного коэффициента диффузии такое решение будет удовлетворять уравнению

$$\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u}{\partial r}\right) = 0.$$

Общим решением этого уравнение будет выражение

$$u = A\ln r + B.$$

Коэффициенты A, B выразим через значения функции в точках коллокации:

$$u(r_i) = u_i, \quad u(r_j) = u_j.$$

Тогда решение примет вид

$$u(r) = \frac{\ln r - \ln r_i}{\ln r_j - \ln r_i}\left(u_j - u_i\right) + u_i.$$

Отсюда выразим нормальную производную на грани γ_{ij}:

$$\left.\frac{\partial u}{\partial n}\right|_{\gamma_{ij}} = -\left.\frac{\partial u}{\partial r}\right|_{r=r_j} = \frac{1}{r_j}\frac{1}{\ln r_i - \ln r_j}\left(u_j - u_i\right). \tag{1.44}$$

Это выражение будем использовать вместо (1.19) для вычисления производной на грани с логарифмической особенностью.

Учёт этой особенности можно реализовать за счёт модификации коэффициента диффузии λ_{ij} на

граничной грани. Выражение (1.44) можно свести к (1.41), если считать диффузию в виде

$$\lambda'_{ij} = \frac{\lambda_{ij}}{r_j} \frac{h_{ij}}{\ln r_i - \ln r_j} \tag{1.45}$$

1.3.13 Радиально-симметричная постановка

Теперь рассмотрим уравнение (1.1) в цилиндрических координатах (r, θ, z) и радиально-симметричной постановке. В частных производных определяющее уравнение запишется как

$$\frac{1}{r}\frac{\partial}{\partial r}\left(\lambda r \frac{\partial u}{\partial r}\right) + \frac{\partial}{\partial z}\left(\lambda \frac{\partial u}{\partial z}\right) = f, \quad \frac{\partial u}{\partial \theta} = 0.$$

Заметим, что общий вывод конечнообъёмной схемы (1.39) использует только формулу Гаусса–Остроградско (C.9) в операторном виде. Поэтому она будет справедлива в любой системе координат.

Особенность выбора радиально-симметричной постановки проявится только в вычислении площади грани $|\gamma|$ и объёма ячейки $|E|$, которые в этой постановке являются телами вращения вокруг оси Oz. Вычислим объём как

$$|E| = \int\limits_E d\mathbf{x} = \int\limits_0^{2\pi} \left(\iint\limits_{|E|_{rz}} r dr dz \right) d\phi = 2\pi \underbrace{\frac{1}{|E|_{rz}} \iint\limits_{|E|_{rz}} r \, dr dz}_{r_E} |E|_{rz}$$

где $|E|_{rz}$ и r_E – объём и r–координата центра масс ячейки в декартовой двумерной системе координат (r, z). Аналогичные рассуждения справедливы и для определения площади грани через её двумерную площадь $|\gamma|_{rz}$ и центр масс r_γ. Тогда окончательно запишем

$$|E| = 2\pi r_E |E|_{rz}, \qquad |\gamma| = 2\pi r_\gamma |\gamma|_{rz} \tag{1.46}$$

1.4 Метод конечных элементов

1.4.1 Формулировка

Формулировка классического метода конечных элементов состоит из последовательного применения трёх методик:

- слабая (интегральная) постановка задачи с помощью метода взвешенных невязок,
- использование метода Бубнова–Галёркина для записи интегральных соотношений для коэффициентов СЛАУ,
- построение базисных функций с локальным носителем на основе конечноэлементной сетки.

1.4.1.1 Метод взвешенных невязок

TODO

1.4.1.2 Метод Бубнова–Галёркина

TODO

Пример с применением степенных базисных функций TODO

1.4.1.3 Конечноэлементные базисные функции

TODO

1.4.2 Вывод СЛАУ для аппроксимации уравнения Пуассона

Метод взвешенных невязок Исходное уравнение (1.2) домножим на пробную функцию q и проинтегрируем по области решения

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

Чтобы сравнять порядок производной перед искомой функций u и пробной функцией q применим формулу интегрирования по частям (C.12):

$$-\int\limits_{\partial\Omega} \frac{\partial u}{\partial n} \, q \, ds + \int\limits_{\Omega} \nabla u \cdot \nabla q \, d\mathbf{x} = \int\limits_{\Omega} f \, q \, d\mathbf{x}.$$

Полученное выражение – есть окончательная слабая постановка задачи.

Метод Бубнова–Галёркина В качестве пробной функции q будем испльзовать набор базисных функций ϕ_i, $i \in \overline{0, N-1}$. По этому же базису разложим входящие в постановку скалярные функции u и f. Получим систему из N линейных уравнений

$$-\int\limits_{\partial\Omega} \frac{\partial u}{\partial n} \, \phi_i \, ds + \sum_{j=0}^{N-1} \left(\int\limits_{\Omega} \nabla \phi_j \cdot \nabla \phi_i \, d\mathbf{x} \right) u_j = \sum_{j=0}^{N-1} \left(\int\limits_{\Omega} \phi_j \, \phi_i \, d\mathbf{x} \right) f_j.$$

Первый из интегралов будет не равным нулю только для строк i, соответсвтующих граничным узлам (базисам). Во всех остальных базисная функция согласно свойству согласованности будет равна нулю. Тогда строки СЛАУ, соответсвующие внутренним узлам, примут вид

$$\mathrm{S}u = \mathrm{b}, \quad \mathrm{b} = \mathrm{M}f,$$

где элементы матрицы масс M и матрицы жёсткости S будут равны

$$M_{ij} = \int\limits_{\Omega} \phi_j \phi_i \, d\mathbf{x}, \tag{1.47}$$

$$S_{ij} = \int\limits_{\Omega} \nabla\phi_j \cdot \nabla\phi_i \, d\mathbf{x}. \tag{1.48}$$

Отдельно рассмотрим вычисление интеграла от некоторой функции g в рамках аппроксимации Бубнова–Галёркина:

$$\int\limits_{\Omega} g \, d\mathbf{x} = \sum_{j=0}^{N-1} V_i g_i,$$

где элементы ветора нагрузок V будут равны

$$V_i = \int\limits_{\Omega} \phi_i \, d\mathbf{x}. \tag{1.49}$$

1.4.2.1 Одномерный пример

TODO

1.4.3 Техника сборки конечноэлементных векторов и матриц

1.4.3.1 Элементные вектора

Распишем интеграл (1.49) по области Ω как сумму интегралов по отдельным элементам E_m. При вследствии свойства локальности конечноэелементных базисов в сумме можно оставить лишь элементы, инцидентные базису i:

$$V_i = \int\limits_{\Omega} \phi_i \, d\mathbf{x} = \sum_{m \in \mathrm{J^i}} \int\limits_{E_m} \phi_i^{(m)} \, d\mathbf{x}.$$

Определим элементный вектор нагрузок $V^{(m)}$ как

$$V_j^{(m)} = \int\limits_{E_m} \phi_{g(j)}^{(m)} \, d\mathbf{x}. \tag{1.50}$$

Его размерность равна количеству степеней свободы элемента $N^{(m)}$. Здесь запись $g(j)$ – это перевод локального внутриэлементного индекса $j \in \overline{0, N^{(m)} - 1}$ в глобальный индекс $i \in \overline{0, N}$.

1.4.3.2 Элементные матрицы

Аналогично можно расписать интегралы из (1.47), (1.48) через элементные интегралы. В частности для матрицы масс получим

$$M_{ij} = \int\limits_{\Omega} \phi_i \phi_j \, d\mathbf{x} = \sum_{m \in \mathrm{J^i}} \int\limits_{E_m} \phi_i^{(m)} \, \phi_j^{(m)} \, d\mathbf{x}.$$

а выражение для локальной матрицы масс размерности $N^{(m)} \times N^{(m)}$ примет вид

$$M_{kl}^{(m)} = \int\limits_{E_m} \phi_{g(k)}^{(m)} \, \phi_{g(l)}^{(m)} \, d\mathbf{x}. \tag{1.51}$$

По аналогии элементная матрица жёсткости запишется как

$$S_{kl}^{(m)} = \int\limits_{E_m} \nabla\phi_{g(k)}^{(m)} \cdot \nabla\phi_{g(l)}^{(m)} \, d\mathbf{x}. \tag{1.52}$$

1.4.3.3 Алгоритм сборки

После того, как элементные вектора/матрицы для искомого интеграла определены, следует применить алгоритм элементной сборки. Для этого понадобится таблица связность “элемент-индексы базисов”. В простейшем случае линейных лагранжевых базисов эта таблица эквивалентна геометрической таблице “ячейка-узлы”. Назовём эту таблицу *glob*. Тогда псеводкод для сборки вектора примет вид

$V_i = 0$ – инициализируем глобальный вектор нулями
for $m = \overline{0, N-1}$ – цикл по конечным элементам
  $N^m = dof(m)$ – количество степеней свободы у элемента
  for $i = \overline{0, N^m - 1}$ – цикл по базисам внутри элемента
    $g = glob(m, i)$ – глобальный индекс локального базиса
    $V_g \mathrel{+}= V_i^m$
  endfor
endfor

(1.53)

В аналогичном алгоритме для матрицы добавится ещё один цикл:

$M_{ij} = 0$ – инициализируем глобальную матрицу нулями
for $m = \overline{0, N-1}$ – цикл по конечным элементам
  $N^m = dof(m)$ – количество степеней свободы у элемента
  for $i = \overline{0, N^m - 1}$ – цикл по базисам внутри элемента
    for $j = \overline{0, N^m - 1}$ – цикл по базисам внутри элемента
      $g = glob(m, i)$ – глобальный индекс локального базиса
      $h = glob(m, j)$
      M_{gh} += M^m_{ij}
    endfor
  endfor
endfor

(1.54)

1.4.4 Вычисление элементных интегралов в модельном пространстве

Будем вычислять элементные интегралы (1.50) – (1.52) в модельном пространстве $\boldsymbol{\xi}$. Для этого введём преобразование координат $\mathbf{x} \to \boldsymbol{\xi}$ согласно п.D.1.2. Интеграл для определения локальной матрицы масс (1.51) в модельных координатах распишется согласно формуле (D.9)

$$M^{(m)}_{ij} = \int\limits_{\tilde{E}_m} \mathcal{N}^{(m)}_i \mathcal{N}^{(m)}_j |J^{(m)}| \, d\boldsymbol{\xi} \tag{1.55}$$

Здесь $\tilde{E}_m$ – параметрический образ конечного элемента E^k, $|J^{(m)}(\boldsymbol{\xi})|$ - Якобиан преобразования для m-ого элемента, $\mathcal{N}^{(m)}_i(\boldsymbol{\xi})$ – функция формы, или часть базисной фукнции, заданная в m-ом элементе в модельных координатах. То есть

$$\mathcal{N}^{(m)}_i(\boldsymbol{\xi}) = \phi^{(m)}_{g(i)}(\mathbf{x}(\boldsymbol{\xi})).$$

Локальный вектор нагрузок записывается из соотношения (1.50):

$$V^{(m)}_i = \int\limits_{\tilde{E}_m} \mathcal{N}^{(m)}_i |J^{(m)}| \, d\boldsymbol{\xi} \tag{1.56}$$

Локальная матрица жёсткости из (1.52):

$$S^{(m)}_{ij} = \int\limits_{\tilde{E}_m} \nabla_{\mathbf{x}} \mathcal{N}^{(m)}_i \cdot \nabla_{\mathbf{x}} \mathcal{N}^{(m)}_j |J^{(m)}| \, d\boldsymbol{\xi} \tag{1.57}$$

Здесь $\nabla_{\mathbf{x}} \mathcal{N}^m_i$ – градиент shape-функции (заданного в модельном пространстве) по физическим координатам. Для его вычисления следует воспользоваться формулами (D.8)

A Задания для самостоятельной работы

A.1 Лекция 2 (20.09.25) МКО для решения уравнения Пуассона

Теория: п. 1.3

В тесте `poisson1-fvm` из файла `poisson_fvm_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,
2. провести серию расчётов на сгущающихся сетках разных типов (структурированных, pebi и скошенных),
3. визуализировать в Paraview полученное на этих сетках решение,
4. построить графики сеточной сходимости решения и определить порядок аппроксимации метода,
5. в тестовой программе производится сборка в цикле по ячейкам (п. 1.3.4.1). Следует переписать процедуру сборки в циклах по граням (п. 1.3.4.2) и убедиться в идентичности результатов.

Работа с сетками Все сетки в программе наследуются от абстрактного класса `IGrid`. Необходимые для работы с сетками таблицы узлов, свойств и связности доступны как виртуальные методы этого класса и объявлены в заголовочном файле `grid/i_grid.hpp`. Например

- `Point IGrid::point(size_t ipoint)` – получить координату i-ой точки,
- `double IGrid::face_area(size_t iface)` – площадь i-ой грани,
- `std::vector<int> IGrid::tab_cell_face(size_t icell)` – получить список индексов граней для i-ой ячейки.

Грани (внутренние и граничные) пронумерованы сквозным образом. Координаты точек всегда трёхмерные. Для двумерных и одномерных задач “лишние” координаты приравниваются нулю.

С помощью метода `IGrid::save_vtk` сетка может быть экспортирована в vtk формат и просмотрена в Paraview. В п. E.6 приведены некоторые приёмы визуализации численного решения.

Для построения двумерных сеток необходимо использовать класс `Grid2`:

```
// сетка 10x10 в единичном квадрате
auto grid = std::make_shared<RegularGrid2D>(0, 0, 1, 1, 10, 10);
```

Неструктурированные сетки должны быть прочитаны из файла:

```
// Читаем сетку из файла /app/test_data/pebigrid.vtk
std::string fn = test_directory_file("pebigrid.vtk");
UnstructuredGrid2D grid = UnstructuredGrid2D::vtk_read(fn);
```

Строить неструктурированные сетки следует с помощью утилиты `hybmesh`. В папке `test_data` корневой директории репозитория лежат скрипты построения сеток:

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

Инструкции по запуску этих скриптов смотри п. E.7. 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`.

Тестовая задача Для тестирования двумерной задачи следует использовать двумерное точное решение. Например,

```
double exact_solution(Point p) const override{
    double x = p.x;
    double y = p.y;
    return cos(10*x*x)*sin(10*y) + sin(10*x*x)*cos(10*x);
}
```

Для вычисления правой части (функции `exact_rhs`) нужно подставить точное решение в исходное уравнение (1.2).

График сходимости График сеточной сходимости следует строить по аналогии с графиком на рис. 10. в логарифмических осях, где по оси абсцисс отложено разбиение, а по оси ординат – норма. Разбиение – это характерный линейный размер области расчёта, делённый на характерный линейный размер ячейки. Для получения корректного порядка аппроксимации в двух- и трёхмерных задачах следует внимательно отнестись к вычислению этого параметра. Для двумерной/трёхмерной области характерный размер можно определить как квадратный/кубический корень от объёма.

Рекомендации к программированию Свои программы следует оформлять в виде отдельных тестов (вместо того, чтобы модифицировать существующие). Желательно, после того как программа заработает, сразу оставить несколько базовых `CHECK` проверок и сделать локальный коммит, чтобы впоследствии легко распознавать и исправлять внесённые в дальнейшей работе ошибки. К тому же наличие готовых тестов значительно облегчает рефакторинг кода.

При написании новых тестов следует переиспользовать уже написанный код, избегая копирования. Для этого необходимо оформлять повторяющийся код в виде отдельных процедур и пользоваться механизмами наследования классов.

A.2 Лекция 3 (27.09.25) Поправка на скошенные сетки и периодические г.у.

Теория: п. 1.3.9

В тесте `poisson2-fvm` из файла `poisson_fvm_test.cpp` реализовано решение двумерного уравнения Пуассона с граничными условиями первого рода. Используется явный итерационный алгоритм поправки на скошенность. Определение вектора $\mathbf{c}'$ осуществляется методом поворота. Отталкиваясь от этой реализации необходимо:

1. Задаться тестовой функцией $u^{ex}(x, y)$, периодичной в направлении x с единичным периодом. Пересчитать для неё вектор правой части и повторить тест.

2. Проиллюстрировать решение с помощью изолиний (п. E.6.2). Сравнить решения на грубой сетке без поправки и с поправкой.

3. Построить серию сгущающихся сеток с помощью алгоритма `tetragrid.py`. Показать второй порядок аппроксимации схемы с поправкой

4. Реализовать вычисление вектора $\mathbf{c}'$ с помощью методов прямой и обратной проекции из п. 1.3.9.1. Сравнить скорость сходимости этих методов, нарисовав зависимость нормы от номера итерации для трёх методов

5. Реализовать периодические условия по граням $x = 0, 1$. Сравнить графики сеточной сходимости решения для этой задачи с задачей с условиями первого рода

Работа с расширенным набором точек коллокации в тестовой программе осуществляется в объекте `ecol_` класса `FvmExtendedCollocations`. Из него, в частности, берутся:

- `points` – координаты точек коллокации,
- `size()` – количество точек коллокации,
- `tab_face_colloc()` – таблица связности “грань – точка коллокации”
- и т.д. Полный список методов смотри в файле `fvm/fvm_extended_collocations.hpp`.

Нумерация точек коллокаций устроена так, что первые N (по количеству ячеек) индексов соответствуют внутренним точкам, оставшиеся N^Γ – граничные точки коллокации.

Подсчёт градиентов в центрах граней осущесвляется методом наименьших квадратов (п. 1.3.10.2) и реализован в объекте `grad_computer_` класса `IFvmFaceGradient`.

В тестовой задаче использовался алгоритм, в котором $|\mathbf{c}'| = 1$. Поэтому эта поправка не вносилась в левую часть. То есть использовался алгоритм (1.23) вместо (1.36). В случае использования алгоритма поворота эти процедуры идентичны. Следует обратить на это внимание при программировании алгоритмов с проекциями, где $|\mathbf{c}'| \neq 1$.

В представленном коде не производится разделения на шаг инициализации и шаг итерации, как описано в алгоритме явного учёта поправки в п. 1.3.9.3. Вместо это и правая и левая часть целиком пересобираются на каждой итерации. За счёт реализации такого разделения код может быть оптимизирован.

Рекомендации к программированию периодических условий По аналогии с классом `DirichletFace` следует создать класс `PeriodicFacePair` куда следует положить индексы соответствующих друг другу периодических граней и соотвествующие им точки коллокации. Сборку массива периодических пар следует проводить в процедуре `initialize`, которую, вероятно, следует сделать виртуальной. Отличить граничную грань первого рода от периодической граничной грани можно по координате её центра `grid_->face_center(iface)`.

Для отладки процедуры сборки можно пользоваться процедурами печати полной матрицы (если матрица совсем маленькая) – `dbg::print(mat)` и печати одной выбранной строки `dbg::print(irow, mat)`. Эти процедуры определены в файле `dbg/printer.hpp`.

Если не получается сразу решить задачу для неструктурированной сетки, имеет смысл попробовать рассмотреть задачу на регулярной сетке. Чтобы отсечь возможные ошибки, связанные с учётом неортогональности.

A.3 Лекция 4 (04.10.25) Непостоянный коэффициент диффузии, учёт особенностей

Теория п. 1.3.11, п. 1.3.12, п. 1.3.13.

В тесте `poisson1-fvm-radial` из файла `poisson_fvm_test.cpp` реализовано решение одномерного уравнения Пуассона с граничными условиями первого рода и неоднородным коэффициентом диффузии в радиально-симметричной постановке

$$\frac{1}{r}\frac{\partial}{\partial r}\left(\lambda(r) r \frac{\partial u}{\partial r}\right) = 0,$$
$$u(r_0) = a, \quad u(r_1) = b,$$
$$\lambda(r) = \begin{cases} \lambda_0, & r < r_k, \\ \lambda_1, & r \geq r_k. \end{cases}$$

Общий вид решения этого уравнения примет вид

$$u(r) = \begin{cases} u_0 = A_0 \ln(r) + B_0, & r < r_k \\ u_1 = A_1 \ln(r) + B_1, & r \geq r_k. \end{cases}$$

Четыре неизвестные константы находятся из двух граничных условий плюс двух условий на границе скачка коэффициента диффузии:

$$\begin{aligned} r = r_0 : &\quad u_0 = a; \\ r = r_1 : &\quad u_1 = b; \\ r = r_k : &\quad u_0 = u_1, \quad \partial u_0/\partial r \, = \, \partial u_1/\partial r \, . \end{aligned} \tag{A.1}$$

Использовались параметры

$$r_0 = 0.05, \quad r_1 = r_0 + 1.0, \quad r_k = r_0 + 0.5, \quad a = 1, \quad b = 0, \quad \lambda_0 = 10, \quad \lambda_1 = 1.$$

Для определения коэффициента диффузии на границе использовалось среднее арфифметическое (1.40). Учёт логарифмической особенности не производился. Необходимо:

1. Изменить способ вычисления коэффициента диффузии λ_{ij} на границе γ_{ij} на формулу (1.43),
2. Добавить учёт логарифмической особенности при вычислении нормальной производной около внутренней границы
3. Сравнить полученные решения на графиках для грубых сеток. Построить графики сеточной сходимости и сравнить порядки аппроксимации: первоначальной схемы, схемы с улучшением из п.1, схемы с улучшением из п.2 и схемы с обоими улучшениями.
4. Решить ту же задачу в декартовой 2D постановке на неструктурированной сетке с итерационной поправкой на неортогональность. Использовать учёт логарифмической особенности. Сравнить сеточную сходимость в случае использования формул (1.42) и (1.43).

5. Решить аналогичную сферически-симметричную задачу с учётом сферической особенности. Показать второй порядок аппроксимации решения.

Как было показано в п. 1.3.13, решение в конечнообъёмная схема в радиально-симметричном случае отличается от расчётной схемы в декартовых координатах только формулой вычисления мер площади и объёмов. Поэтому для этого случая был создан отдельный класс радиальных сеток `RadialGrid1D`, в котором формулы вычисления площадей были переписаны согласно формулам (1.46).

Коэффициент диффузии в точках коллокации хранится в поле `lambda_`. Вычисление его значения на грани λ_{ij} происходит в методе `face_lambda`.

Для выполнения пункта 5 нужно будет получить точное решение для задачи

$$\frac{1}{r^2}\frac{\partial}{\partial r}\left(\lambda(r) r^2 \frac{\partial u}{\partial r}\right) = 0,$$
$$u(r_0) = a, \quad u(r_1) = b,$$
$$\lambda(r) = \begin{cases} \lambda_0, & r < r_k, \\ \lambda_1, & r \geq r_k. \end{cases}$$

Для этого нужно использовать те же условия сращивания (A.1), но с общим решением вида

$$u(r) = \begin{cases} u_0 = \dfrac{A_0}{r} + B_0, & r < r_k \\ u_1 = \dfrac{A_1}{r} + B_1, & r \geq r_k. \end{cases}$$

Рекомендации к программированию Для учёта особенности (п. 2) нужно модифицировать коэффициент диффузии для граничных граней на границе r_0 по формуле (1.45) в методе `face_lambda`.

Для выполнени пункта 4 понадобится построить сетку в плоскости (x, y). Для этого следует использовать сетку из построителя `radial.py`,

Для пункта 5 будет удобно по аналогии с классом `RadialGrid1D` создать класс сеток `SphericalGrid1D` с соответствующими правилами вычисления площадей и объёмов.

A.4 Лекция 6 (18.10.25) Метод линейных конечных элементов

В тесте `poisson1-fem-linsegm` из файла `poisson_fem_test.cpp` реализовано решение одномерного уравнения Пуассона. Разбор этой программы смотри в п. B.2. На основе этого теста необходимо

1. Показать второй порядок аппроксимации решения одномерного уравнения Пуассона на линейных конечных элементах;

2. Вместо используемых точных формул вычисления элементых матриц использовать квадратурные формулы. Нарисовать графики сеточной сходимости при использовании квадратурных формул, точных для полиномов 1-ой и 2-ой степеней.

3. Решить двумерное уравнение Пуассона с граничными условиями первого рода в квадратной области на треугольной сетке. Для построения треугольных сеток различного разрешения использовать скрипт
   `trigrid.py`. Показать сеточную сходимость при сипользовании квадратурных формул, точных для полиномов 1-ой, 2-ой и 3-ей степеней.

4. Реализовать квадратурную формулу, с узлами, расположенными в узлах модельного треугольного элемента: $\xi_0(0,0)$, $\xi_1(1,0)$, $\xi_2(0,1)$ и равными весами $w_i = 1/6$. Построить график сеточной сходимости и сравнить с результатом из предыдущего пункта.

Рекомендации к программированию Программировать вычисления локального вектора нагрузок, матрицы масс и матрицы жёсткости (пункт 2) с помощью общих квадратурных формул лучше всего путём определения функций `element_load_vector`, `element_mass_matrix` и `element_stiffness_matrix` на уровне базового класса `ITestPoissonFemWorker`. Тогда переключать программу в режим работы по квадратурным функциям можно с помощью вызова родительского метода из частного. Например для вектора нагрузок:

```
std::vector<double> TestPoissonLinearSegmentWorker::element_load_vector(size_t ielem)
↪  const{
    return ITestPoissonFemWorker::element_load_vector(ielem);
}
```

Более того, реализация базового метода с помощью универсального алгоритма позволит не делать частную реализацию при программировании двумерной задачи (пункт 4).

Для самой реализации необходимо задать набор квадратур для конечного элемента `FemElement` путём определения поля `quadrature` через одну из квадратур, заданных в `quadrature.hpp`. Для этого в методе `build_fem` (где осуществляется сборка элементов) вместо нулевого указателя задать конкретную формулу:

```
auto quad = quadrature_segment_gauss2(); // полином 2-го порядка
```

Далее непосредственно в методе `element_...` получить конечный элемент (объект класса `FemElement`) по его индексу можно через объект сборщика

```
auto elem = fem_.element(ielem);
```

В свою очередь из этого объекта можно получить как квадратурные коэффициенты:

```
std::shared_ptr<const Quadrature> quad = elem->quadrature;  // коэффициенты
↪  квадратурной формулы
```

так и все необходимый функционал для вычисления подинтегральных выражений функций (1.55) – (1.57):

```
auto basis = elem->basis;   // shape-функции с параметрическом пространстве
auto geom = elem->geometry; // геом. свойства включая матрицу Якоби
```

Реализацию интегрирования проводить по примеру из п. B.2.3.4.

Для реализации двумерного решения (пункт 3) 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`. Треугольные конечные элементы собирать по

```
auto geom = std::make_shared<TriangleLinearGeometry>(p0, p1, p2);
auto basis = std::make_shared<TriangleLinearBasis>();
auto quad = quadrature_triangle_gauss2(); // 2-nd order polynom quadrature
```

Треугольную сетку следует положить в папку `test_grid`. После чего она может быть прочитаны из файла:

```
// Читаем сетку из файла /app/test_data/trigrid.vtk
std::string fn = test_directory_file("trigrid.vtk");
UnstructuredGrid2D grid = UnstructuredGrid2D::vtk_read(fn);
```

Для написания собственной квадтратурной формулы (пункт 4) необходимо создать свой объект класса `Quadrature`, передав в конструктор необходимые узлы и веса.

A.5 Лекция 8 (01.11.25) Конечные элементы высокого порядка

Тест `poisson2-fem-quadtri` из файла `poisson_fem_test.cpp` реализовано решение двумерного уравнения Пуассона с гранчиными условиями первого рода на квадратичных конечных элементов Лагранжа.

Также в тесте `poisson2-fem-radial` решена двумерная задача Лапласа со смешанными граничными условиями в кольце:

$$
\begin{aligned}
&-\nabla u = 0, \quad r_0 \le r \le r_1, \quad r = \sqrt{x^2 + y^2} \\
&r = r_0 : -\frac{\partial u}{\partial n} = q, \\
&r = r_1 : u = 0.
\end{aligned}
$$

Известное точное решение этой задачи:

$$
u^{ex} = q r_0 \ln\left(\frac{r}{r_1}\right).
$$

Здесь были использваны линейные конечные элементы, но использвалось неполное квадратичное преобразование геометрии на наборе шейп функций с узлами

$$
\boldsymbol{\xi}_i = (-1,-1), (1,-1), (1,1), (-1,1), (0,-1)
$$

при работе с конечными элементами около внутренней границы.

На основе этих тестов необходимо

1. Провести расчёт сеточной сходимости и показать порядок аппроксимации используемого метода для первой задачи. Для построения сеток разного разрешения использовать скрипт `trigrid.py`.
2. Решить ту же задачу с помощью линейных и кубических элементов. Нарисовать рядом три графика сходимости для элементов первого, второго и третьего порядка.
3. На основе имеющегося теста из второй задачи реализовать полностью линейное преобразование геометрии. Сравнить два графика сходимости для двух преобразований геометрии. Для построения сеток разного разрешения использовать скрипт `radial.py`.
4. Построить два таких же графика, но с использованием Лагранжевых Q и P элементов второго порядка.

Рекомендации к программированию Для понижения порядка используемых сеточных элементов до линейного в п.2 необходимо в функции `build_fem`

- правильно указать количество базисов в переменной `n_bases`,
- указать линейный базис `TriangleLinearBasis` при создании набора шейп-функций `basis`,

- при сборке таблицы связности
  `tab` убрать упоминание граневых базисов, оставив лишь узловые.

- так же следует убрать граневые базисы из списка индексов базисных функций, к которым приписаны условия первого рода в методе `dirichlet_bases`

Чтобы напротив, повысить порядок, использовав кубические элементы рис. 17в, следует ввести следующую нумерацию базисов: сначала удут узловые базисные функции, зачем граневые (по две на каждую грань) и завершают список элементные bubble-функции. Тогда нужно

- изменить `n_bases`,

- указать кубический базис `TriangleCubicBasis` при создании набора шейп-функций `basis`,

- при сборке таблицы связности `tab` добавить ещё одну граневую базисную функцию и одну bubble-функцию для каждого элемента:

```
std::vector<size_t> tab = info.ipoints;   // узловые
for (size_t i=0; i<info.n_points(); ++i){
    size_t iface = info.ifaces[i];
    if (!info.is_face_reverted[i]){  // в правом элементе грань "перевёрнута"
        tab.push_back(grid.n_points() + 2*iface + 0); // две граневые
        tab.push_back(grid.n_points() + 2*iface + 1);
    } else {
        tab.push_back(grid.n_points() + 2*iface + 1);
        tab.push_back(grid.n_points() + 2*iface + 0);
    }
}
tab.push_back(grid.n_points() + 2*grid.n_faces + icell); // bubble
```

- так же следует добавить ещё один граневый базис в списка индексов базисных функций в методе `dirichlet_bases`

Для понижения порядка геометрии до линейной в п.3 достаточно для приграничного Q-элемента указать тот же геометрический класс, что и для всех остальных:

```
el.geometry = std::make_shared<QuadrangleLinearGeometry>(p0, p1, p2, p3);
```

Чтобы повысить порядок элемента (п.4) до квадратного, следует учесть, что в настоящей сетке используются как четырёхугольные Q-элементы, так и треугольные P-элементы. Набор Квадратичных shape-функций для Q-элементов (см. рис. 20б) включает в себя bubble-функцию. Таким образом общее количество базисов будет равно количеству узлов плюс количество граней плюс количество Q-элементов. С учётом этого нужно изменить

- `n_bases`,

- `el.basis` – `TriangleQuadraticBasis` для P–элементов и `QuadrangleQuadraticBasis` для Q–элементов
- таблицу связности `tab`:

```
std::vector<size_t> tab = info.ipoints; // узловые
for (size_t iface: info.ifaces){
    tab.push_back(iface + grid.n_points()); // граневые
}
if (info.n_points() == 4){ // bubble для 4-х узловых
    tab.push_back(grid.n_points() + grid.n_faces() + q_element_index++);
}
```

  где счётчик `q_element_index` следует инициализировать нулём до начала цикла по ячейкам `icell`,
- не забыть добавить граневые базисы в метод `dirichlet_bases`.

B Детали программной реализации

B.1 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 $x \in [0, 1]$.

B.1.1 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. 10).

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

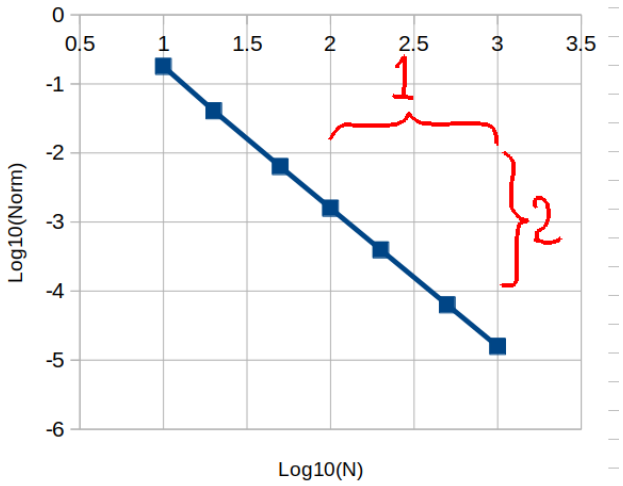


Рис. 10: Сходимость с уменьшением разбиения при решении одномерного уравнения Пуассона

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. 11).

B.1.2 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`:

```
15 class TestPoisson1Worker {
```

В методе `solve()` производится численное 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 (п. D.2.1), удобном для последовательного чтения.
2. Вызывается решатель СЛАУ. Решение записывается в приватное поле класса `u`.
3. Вызывается функция вычисления нормы.

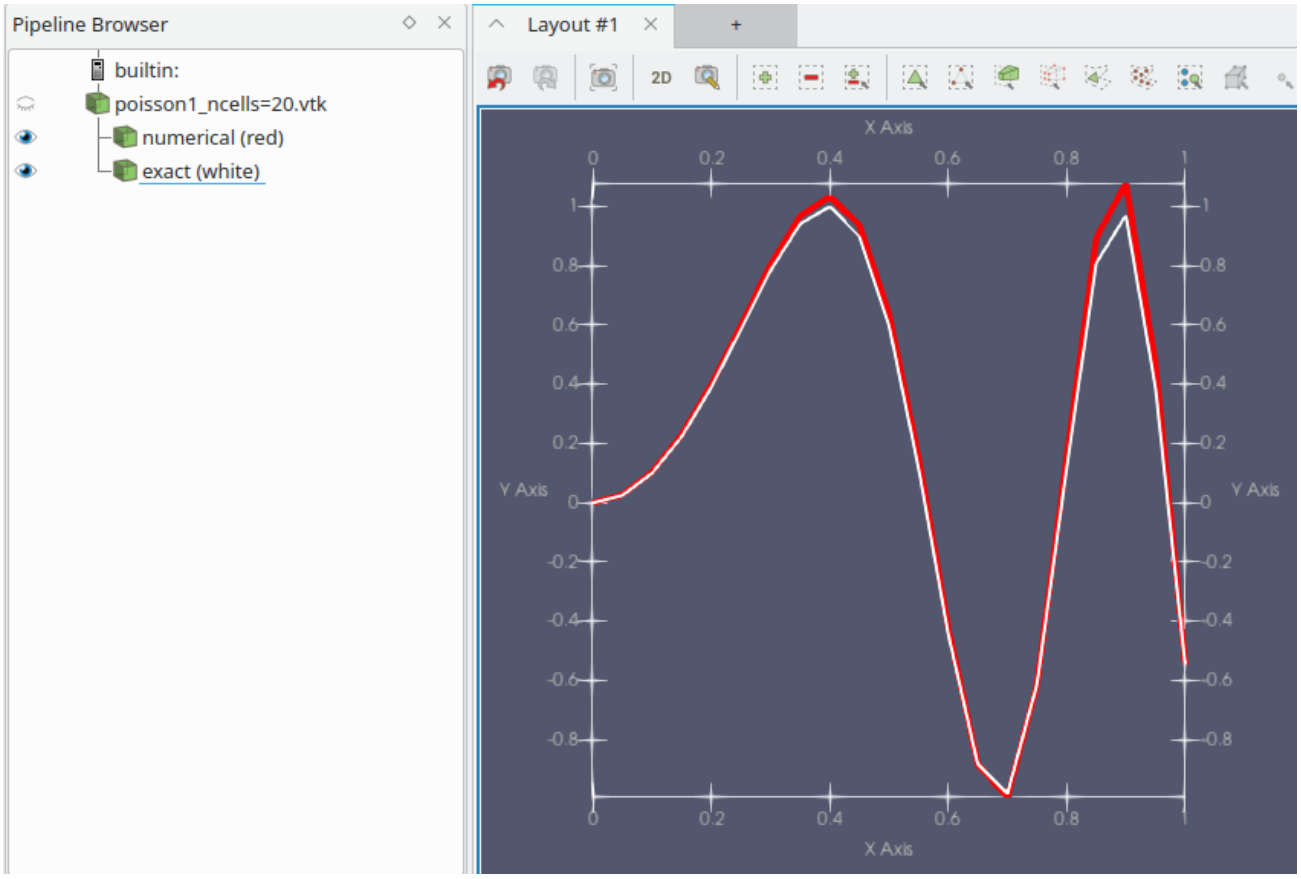


Рис. 11: Сравнение точного и численного решений уравнения Пуассона

```
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`):

- Сборка левой части СЛАУ. Реализует формулу (1.12). Для заполнения матрицы используется формат `cfd::LodMatrix` (п. D.2.2), 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 (1.13).

```
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 (1.15).

```
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     }
```

B.2 Разбор программной реализации МКЭ

Численное решение уравнения Пуассона с граничными условиями первого рода реализовано в файле `poisson_fem_test.cpp`. Будем рассматривать решение одномерной задачи с использованием пирамидальных базисов (тест `[poisson1-fem-lintri]`). В этом тесте определяется 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):

```
290     Grid1D grid(0, 1, 10);
291     TestPoissonLinearSegmentWorker worker(grid);
292     double nrm = worker.solve();
293     worker.save_vtk("poisson1_fem.vtk");
294     std::cout << grid.n_cells() << " " << nrm << std::endl;
```

Основная работа происходит в классе `TestPoissonLinearSegmentWorker`.

B.2.1 Рабочий объект

В конструкторе класса `TestPoissonLinearSegmentWorker` происходит вызов статической функции `build_fem`, в которой осуществляется сборка массива конечных элементов и таблицы связности “элемент–индексы базисов, определённых в элементе” (таблица `glob` из алгоритма `eq:fem_vector_assemble`). Эти данные используются для построения конечноэлементного сборщика (п. B.2.2).

В родительском класса рабочего объекта `ITestPoisson1FemWorker`, сформулированы аналитические функции, служащие правой частью, точным решением и условиями первого рода уравнения Пуассона.

А в базовом классе `ITestPoissonFemWorker` реализованы основные процедуры решения уравнения.

```
69 double ITestPoissonFemWorker::solve() {
70     // 1. build SLAE
71     CsrMatrix mat = approximate_lhs();
72     std::vector<double> rhs = approximate_rhs();
73     // 2. solve SLAE
74     AmgcMatrixSolver solver({{"precond.relax.type", "gauss_seidel"}, {"solver.tol",
   ↪  "1e-12"}});
```

```
    solver.set_matrix(mat);
    solver.solve(rhs, u_);
    // 3. compute norm2
    return compute_norm2();
}
```

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

В функции сборки левой части СЛАУ сначала происходит поэлементная сборка матрицы, соответствующая алгоритму (1.54).

```
CsrMatrix ITestPoissonFemWorker::approximate_lhs() const {
    CsrMatrix ret(fem_.stencil());
    for (size_t ielem = 0; ielem < fem_.n_elements(); ++ielem) {
        std::vector<double> local_stiff = element_stiffness_matrix(ielem);
        fem_.add_to_global_matrix(ielem, local_stiff, ret.vals());
    }
    // Dirichlet bc
    for (size_t ibas: dirichlet_bases()) {
        ret.set_unit_row(ibas);
    }
    return ret;
}
```

При этом вычисление элементной матрицы по формуле (1.57) осуществляется в абстрактном методе `element_stiffness_matrix`. В настоящей реализации используется точное вычисление этого интеграла для одномерного линейного элемента. в переопределённом методе `TestPoissonLinearSegmentWorker::element_stiffness_matrix`.

Вставка элементной матрицы в глобальную матрицу осуществляется с помощью специального сборщика `fem_` класса `FemAssembler`.

Далее происходит учёт граничных условий первого рода на матричном уровне, с помощью постановки единицы на диагональ в строку матрицы, соответсвующую граничному узлу (базису).

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

B.2.2 Конечноэлементный сборщик

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

```
12      FemAssembler(size_t n_bases, const std::vector<FemElement>& elements,
13                   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.

B.2.3 Концепция конечного элемента

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

```
53 struct FemElement {
54     std::shared_ptr<const IElementGeometry> geometry;
55     std::shared_ptr<const IElementBasis> basis;
56     std::shared_ptr<const Quadrature> quadrature;
57 };
```

Главная задача объекта этого класса – предоставлять всю информацию для вычисления элементных векторов и 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 и матрицу Якоби,
- Набор shape-функций, заданных в модельной геометрии,
- Непосредственно правило интегрирования в параметрической области (квадратурные формулы)

Каждый из этих трёх алгоритмов определён через объекты классов

- `IElementGeometry`
- `IElementBasis`
- `Quadrature`

Полное определение конечного элемента заключается в задании конкретных реализаций первых двух интерфейсов и объекта с квадратурой.

B.2.3.1 Определение линейного одномерного элемента

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

```
258         auto geom = std::make_shared<SegmentLinearGeometry>(p0, p1);
259         auto basis = std::make_shared<SegmentLinearBasis>();
260         std::shared_ptr<const Quadrature> quad = nullptr;
261         FemElement elem{geom, basis, quad};
```

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 $x \in [-1, 1]$ не задаются (используется `nullptr`), поскольку в дальнейшем интегрирование ведётся по точным формулам.

B.2.3.2 Геометрические свойства элемента

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

```
15 class IElementGeometry {
16 public:
17     virtual ~IElementGeometry() = default;
18 
19     virtual JacobiMatrix jacobi(Point) const = 0;
20     virtual Point to_physical(Point) const {
21         _THROW_NOT_IMP_;
22     }
23     virtual Point to_parametric(Point) const {
24         _THROW_NOT_IMP_;
25     }
26     virtual Point parametric_center() const {
27         _THROW_NOT_IMP_;
28     }
29 };
```

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`.

B.2.3.3 Элементный базис

Интерфейс для определения локального элементного базиса (набора shape-функций) 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<double> value(Point) const = 0;
41     virtual std::vector<Vector> grad(Point) const = 0;
42     virtual std::vector<std::array<double, 6>> upper_hessian(Point) const {
43         _THROW_NOT_IMP_;
44     }
45 };
```

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

- `size` – количество базисных функций;
- `parametric_reference_points` – вектор из параметрических координат точек, приписанных к соответствующим базисам;
- `value` – значение базисных функций в заданной точке;
- `grad` – градиент (в параметрическом пространстве) базисных функций по заданным точкам.
- `upper_hessian` – верхняя часть матрицы Гессе (вторые производные базисных функций в заданной точке)

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

```
39 size_t TriangleLinearBasis::size() const {
40     return 3;
41 }
```

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

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

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

B.2.3.4 Квадратурные формулы

Объект класса `Quadrature`, определённого в файле `quadrature.hpp`, предоставляет узловые точки и веса для численного вычисления интегралов. Гауссовы квадратуры доступны в файлах из папки `cfd/numeric_integration`:

- `segment_quadrature` – квадратуры для интегрирования по отрезку $[-1, 1]$,
- `triangle_quadrature` – квадратуры для интегрирования в треугольнике с узлами $(0, 0)$, $(1, 0)$, $(0, 1)$,
- `square_quadrature` – квадратуры для интегрирования в квадрате $[-1, 1] \times [-1, 1]$.

Последняя цифра в названии конкретной квадратуры – степень полинома, для которой она точна. Например, формула `quadrature_segment_gauss4` – квадратура для интегрирования в отрезке $[-1, 1]$, точная для полинома 4-ой степени.

Для того, чтобы проинтегрировать некоторую заданную координатную функцию f с помощью квадратуры, нужно написать цикл вида

```
double f(Point xi){
    ...
}
std::shared_ptr<Quadrature> quad = ...;
double integral = 0.0;
for (size_t k=0; k<quad->size(); ++k){
```

```
    Point p = quad->point(k);
    double w = quad->weight(k);
   integral += w*f(p);
}
```

Либо можно воспользоваться методом `integrate`:

```
double integral = quad->integrate(f);
```

Для упрощения вычисления специфичных для конечноэлементных интегралов выражений можно внутри подинтегральной функции использовать вспомогательный класс `FemElementValue`, позволяющий вычислять значение shape-функций (метод `phi`) и их производные по физическим координатам (методы `grad_phi`, `laplace`, `divirgence`).

C Формулы и обозначения

C.1 Векторы

C.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

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

- Скалярное произведение

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

- Векторное произведение

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

В двумерном случае можно считать, что $u_z = v_z = 0$. Тогда результатом векторного произведения согласно (C.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

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

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

C.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 (C.1) (здесь и далее приводятся покоординатные выражения для декартовой системы):

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

Результатом этой операции является вектор.

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

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

результат которой – скалярная функция.

Двойное применение ∇ к скалярной функции – это оператор Лапласа:

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

Ротор – аналог векторного умножнения (C.3):

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

C.2 Интегрирование

C.2.1 Формула Гаусса–Остроградского

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

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

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

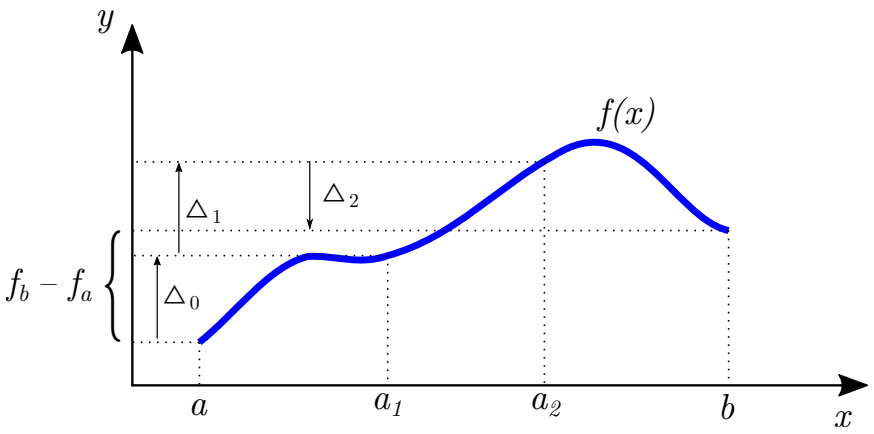


Рис. 12: Формула Гаусса-Остроградского в одномерном случае

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 (C.9) – есть многомерное обобщение этого выражения.

C.2.2 Интегрирование по частям

Подставив в (C.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

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

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

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

При $\mathbf{u} = \nabla g$

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

При $f = 1$ и $\mathbf{u} = \nabla g$

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

C.2.3 Численное интегрирование в заданной области

Квадратурная формула

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

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

C.3 Интерполяционные полиномы

C.3.1 Многочлен Лагранжа

C.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

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

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 (C.15), запишем условие на узловую базисную функцию

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

Дополнительно потребуем, чтобы формула (C.15) была точной для постоянных функций

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

Тогда для любого ξ должно выполняться условие

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

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

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

Определять коэффициенты $A_i^{(a)}$ будем из условий (C.16), которое даёт N линейных уравнений относительно неизвестных $A_i^{(a)}$ для каждого $i = \overline{0, N-1}$. Таким образом, в выражениях (C.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

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

Подставляя полином (C.18) в условие согласованности (C.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.

C.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 (C.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. 13)

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

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

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

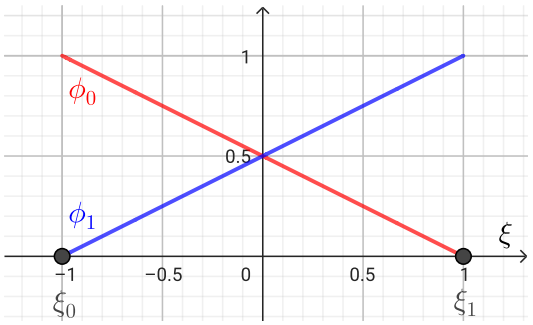


Рис. 13: Линейный базис в параметрическом отрезке

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 (C.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. 14)

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

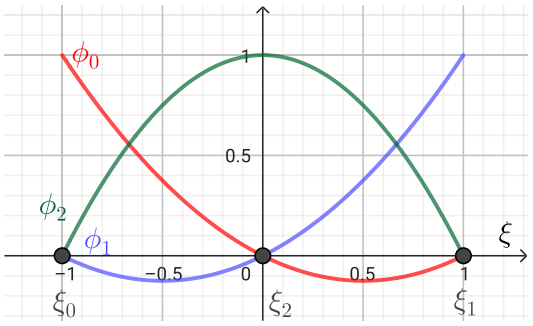


Рис. 14: Квадратичный базис в параметрическом отрезке

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. 15)

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

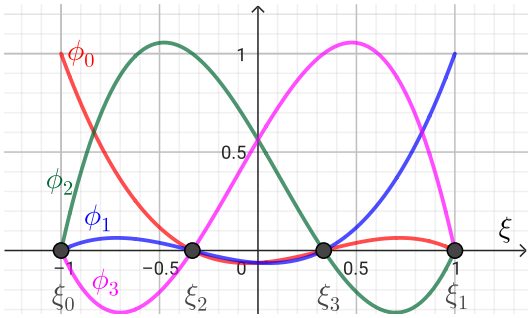


Рис. 15: Кубический базис в параметрическом отрезке

На рис. 16 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.

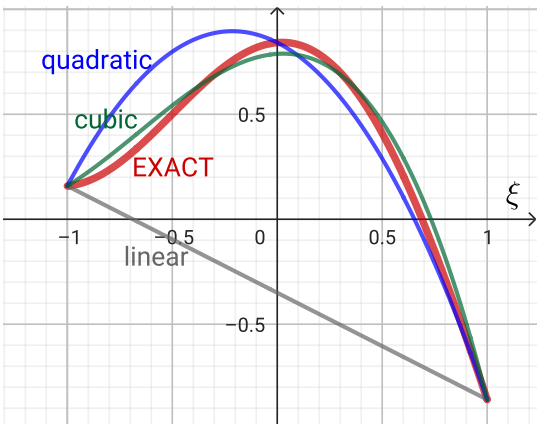


Рис. 16: Результат интерполяции

C.3.1.3 Интерполяция в параметрическом треугольнике

Теперь рассмотрим двумерное обобщение формулы

Линейный базис

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

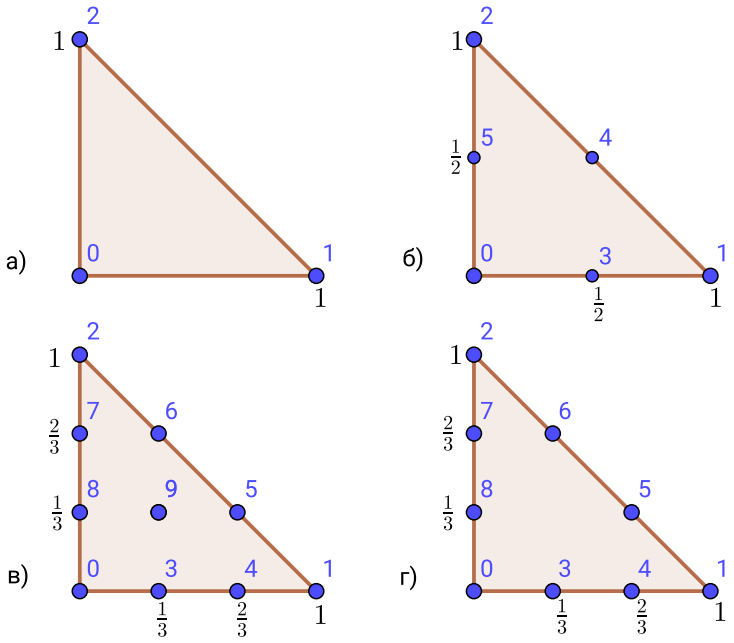


Рис. 17: Расположение узловых точек в параметрическом треугольнике. а) линейный базис, б) квадратичный базис, в) кубический базис, г) неполный кубический базис

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

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

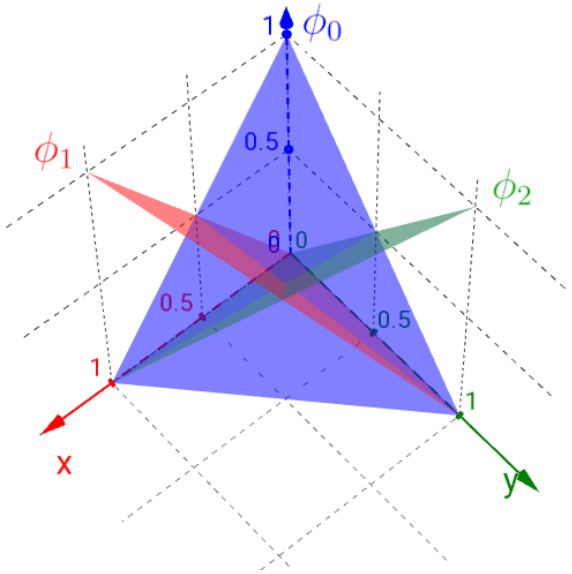


Рис. 18: Линейный базис в параметрическом треугольнике

Квадратичный базис

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

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

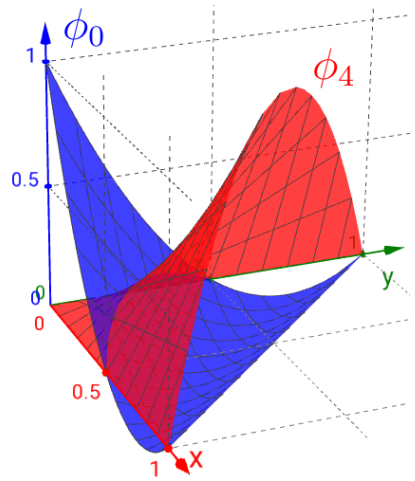


Рис. 19: Квадратичные функции ϕ_0, ϕ_4 в параметрическом треугольнике

Кубический базис TODO

Неполный кубический базис TODO

С.3.1.4 Интерполяция в параметрическом квадрате

Билинейный базис

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

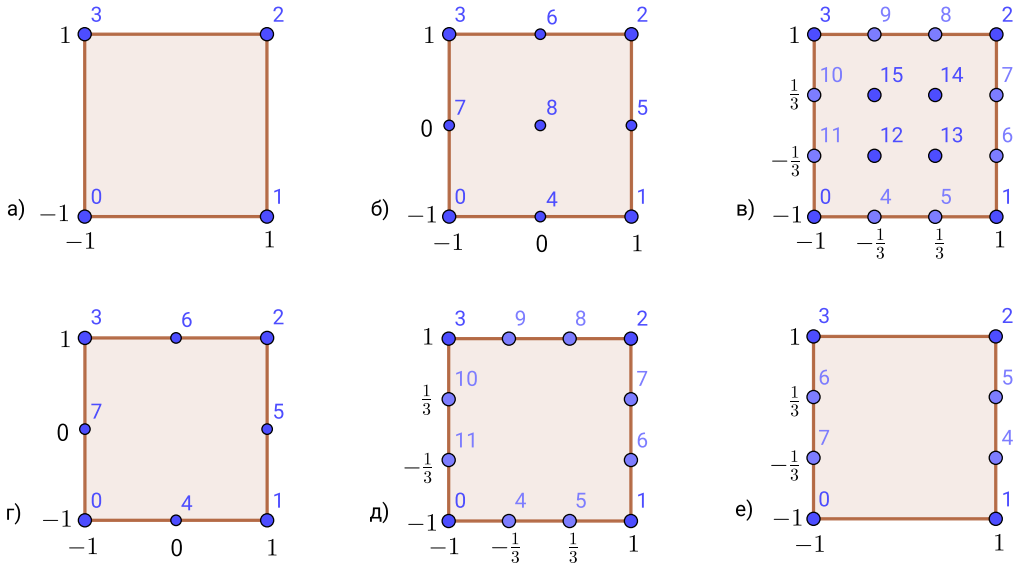


Рис. 20: Расположение узловых точек в параметрическом квадрате

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

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

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 (C.20) в каждом из направлений. Узлы двумерного параметрического квадрата можно выразить через узлы линейного базиса в параметрическом одномерном сегменте, рассмотренном в п. C.3.1.2:

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

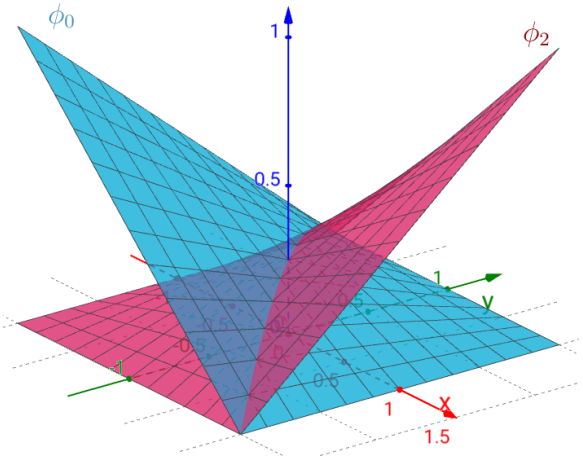


Рис. 21: Билинейные функции ϕ_0, ϕ_2 в параметрическом квадрате

Значит и соответствующие базисные функции можно выразить через линейный одномерный базис ϕ^{1D} из соотношений (C.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 (C.24).

Биквадратичный базис Применим этот метод для вычисления биквадратичного базиса, определённого в точках на рис. 20б. В качестве основе возьмём квадратичный одномерный базис ϕ_i^{1D} из (C.21).

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

Бикубический базис

Неполный биквадратичный базис

Неполный бикубический базис

D Алгоритмы

D.1 Геометрические алгоритмы

D.1.1 Линейная интерполяция

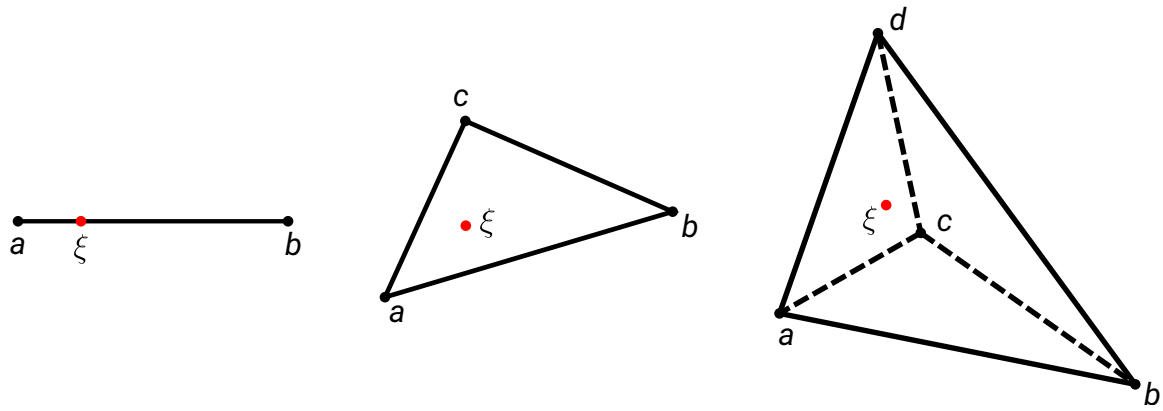


Рис. 22: Порядок нумерации точек одномерного, двумерного и трёхмерного симплекса при линейной интерполяции

Пусть функция u задана в узлах симплекса, имеющего нумерацию согласно рис. 22. 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

$$u(\xi) = \frac{|\triangle_{\xi a}|u(b) + |\triangle_{b\xi}|u(a)}{|\triangle_{ba}|} \tag{D.1}$$

$$u(\xi) = \frac{|\triangle_{ab\xi}|u(c) + |\triangle_{bc\xi}|u(a) + |\triangle_{ca\xi}|u(b)}{|\triangle_{abc}|} \tag{D.2}$$

$$u(\xi) = \frac{|\triangle_{abc\xi}|u(d) + |\triangle_{cbd\xi}|u(a) + |\triangle_{cda\xi}|u(b) + |\triangle_{adb\xi}|u(c)}{|\triangle_{abcd}|}, \tag{D.3}$$

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

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

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

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

D.1.2 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.

D.1.2.1 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

$$\mathbf{x}(\boldsymbol{\xi}_0 + \mathbf{d\xi}) - \mathbf{x}_0 = J(\boldsymbol{\xi}_0)\, \mathbf{d\xi}. \tag{D.4}$$

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

$$J = \begin{pmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{pmatrix} = \begin{pmatrix} \dfrac{\partial x}{\partial \xi} & \dfrac{\partial x}{\partial \eta} \\ \dfrac{\partial y}{\partial \xi} & \dfrac{\partial y}{\partial \eta} \end{pmatrix} \tag{D.5}$$

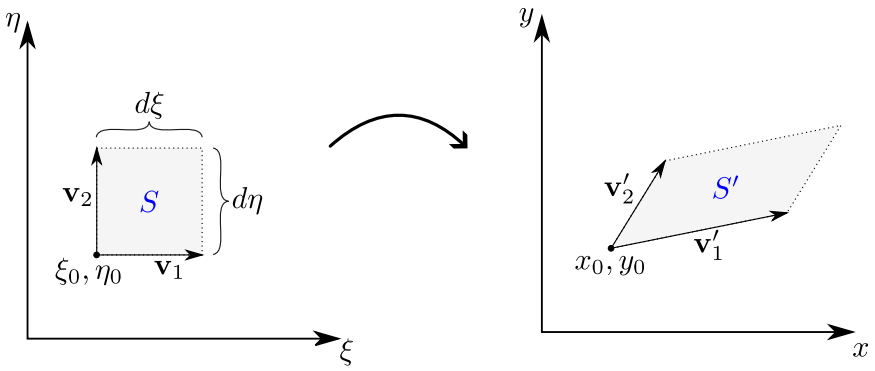


Рис. 23: Преобразование элементарного объёма

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. 23). В результате преобразования по формуле (D.4) 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 (C.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

$$dx\,dy = |J|\,d\xi\,d\eta \tag{D.6}$$

Многомерным обобщением этой формулы будет

$$d\mathbf{x} = |J|\,d\boldsymbol{\xi} \tag{D.7}$$

D.1.2.2 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 (D.5), 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

$$\nabla_{\mathbf{x}} f = \left(J^T\right)^{-1} \nabla_{\boldsymbol{\xi}} f. \tag{D.8}$$

D.1.2.3 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 (D.7)

$$\int\limits_{D_x} f(\mathbf{x})\, d\mathbf{x} = \int\limits_{D_\xi} f(\boldsymbol{\xi})\, |J(\boldsymbol{\xi})| d\boldsymbol{\xi}, \tag{D.9}$$

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

D.1.2.4 Двумерное линейное преобразование. Параметрический треугольник

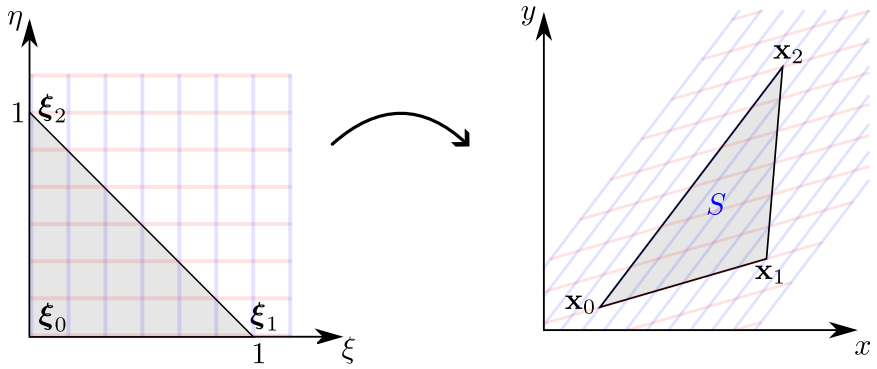


Рис. 24: Преобразование из параметрического треугольника

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

$$\begin{aligned} x(\xi, \eta) &= A_x \xi + B_x \eta + C_x, \\ y(\xi, \eta) &= A_y \xi + B_y \eta + C_y. \end{aligned}$$

Для определения шести констант, определяющих это преобразование, достаточно выбрать три любые (не лежащие на одной прямой) точки: $(\xi_i, \eta_i) \to (x_i, y_i)$ 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. 24):

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

Тогда система линейных уравнений примет вид

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

Определив коэффициенты преобразования их этой системы, окончательно запишем преобразование

$$\begin{aligned} x(\xi, \eta) &= (x_1 - x_0)\xi + (x_2 - x_0)\eta + x_0, \\ y(\xi, \eta) &= (y_1 - y_0)\xi + (y_2 - y_0)\eta + y_0. \end{aligned} \tag{D.10}$$

Матрица Якоби этого преобразования (D.5) не будет зависеть от параметрических координат ξ, η:

$$J = \begin{pmatrix} x_1 - x_0 & x_2 - x_0 \\ y_1 - y_0 & y_2 - y_0 \end{pmatrix}. \tag{D.11}$$

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

$$|J| = (x_1 - x_0)(y_2 - y_0) - (y_1 - y_0)(x_2 - x_0) = (\mathbf{x}_1 - \mathbf{x}_0) \times (\mathbf{x}_2 - \mathbf{x}_0) = 2|S|. \tag{D.12}$$

Распишем интеграл по треугольнику S по формуле (D.9). Вследствии линейности преобразования якобиан постоянен и, поэтому, его можно вынести его из-под интеграла:

$$\int\limits_S f(x,y)\,dxdy = |J| \int\limits_0^1 \int\limits_0^{1-\xi} f(\xi,\eta) d\eta d\xi. \tag{D.13}$$

D.1.2.5 Двумерное билинейное преобразование. Параметрический квадрат

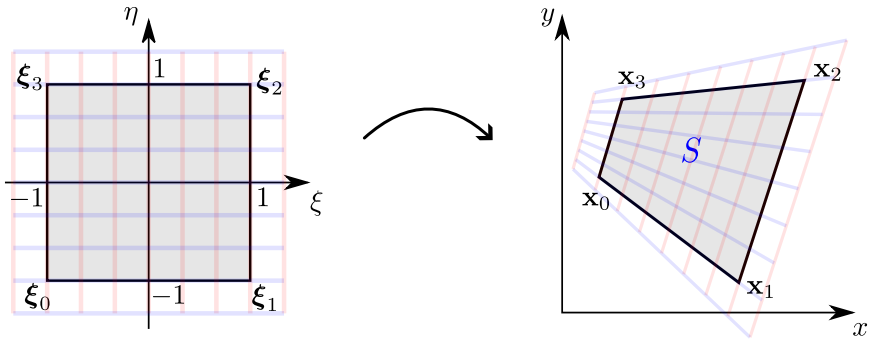


Рис. 25: Преобразование из параметрического квадрата

D.1.2.6 Трёхмерное линейное преобразование. Параметрический тетраэдр

TODO

D.1.3 Свойства многоугольника

D.1.3.1 Площадь многоугольника

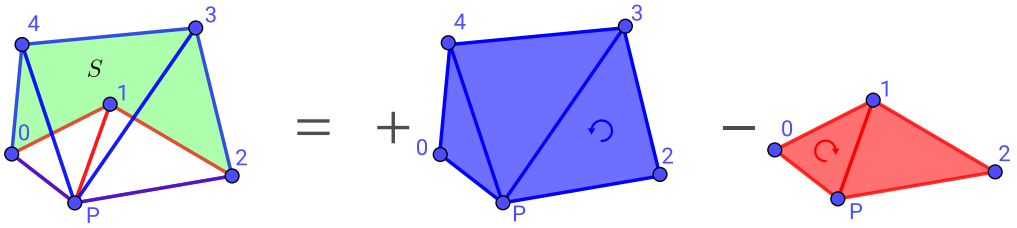


Рис. 26: Площадь произвольного многоугольника

Рассмотрим произвольный несамопересекающийся N-угольник S, заданный координатами своих узлов $\mathbf{x}_i$, $i = \overline{0, N-1}$, пронумерованных последовательно против часовой стрелки (рис. 26). 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 26 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

$$\triangle_i = (\mathbf{x}_0, \mathbf{x}_i, \mathbf{x}_{i+1}), \qquad i = \overline{1, N-2}. \tag{D.14}$$

Знаковая площадь ориентированного треугольника будет равна

$$|\triangle_i| = \frac{(\mathbf{x}_i - \mathbf{x}_0) \times (\mathbf{x}_{i+1} - \mathbf{x}_0)}{2}. \tag{D.15}$$

Тогда окончательно формула определения площади примет вид

$$|S| = \sum_{i=1}^{N-2} |\triangle_i|. \tag{D.16}$$

Плоский полигон в пространтве Если плоский полигон S расположен в трёхмерном пространстве, то правая часть формулы (D.15) согласно определению векторного произведения в трёхмерном пространстве (C.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:

$$|\triangle_i| = \frac{((\mathbf{x}_i - \mathbf{x}_0) \times (\mathbf{x}_{i+1} - \mathbf{x}_0)) \cdot \mathbf{n}}{2}. \tag{D.17}$$

Формула для определения площади полигона (D.16) 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 $\mathbf{n}$.

D.1.3.2 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 (D.14). Тогда интеграл по многоугольнику можно расписать как сумму интегралов по ориентированным треугольникам:

$$\int\limits_S f(x,y)\,dxdy = \sum_{i=1}^{N-2} \int\limits_{\triangle_i} f(x,y)\,dxdy. \tag{D.18}$$

Далее для вычисления интегралов в правой части воспользуемся преобразованием к параметрическому треугольнику (п. D.1.2.4). Следуя формуле интегрирования (D.13), 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 (D.12) 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 (D.10) 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

$$\int\limits_S f(x,y)\,dxdy = 2\sum_{i=1}^{N-2} |\triangle_i| \int\limits_0^1 \int\limits_0^{1-\xi} f_i(\xi,\eta)\,d\eta d\xi. \tag{D.19}$$

Отметим, что эта формула работает и в том случае, когда полигон расположен в трёхмерном пространстве (знаковую площадь при этом следует вычислять по (D.17)).

D.1.3.3 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 (D.18) с учётом линейного преобразования (D.10):

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

Итого, с учётом (D.16), 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 (D.15). В случае трёхмерного пространтва должна использоваться формула (D.17).

D.1.4 Свойства многогранника

D.1.4.1 Объём многогранника

TODO

D.1.4.2 Интеграл по многограннику

TODO

D.1.4.3 Центр масс многогранника

TODO

D.1.5 Поиск многоугольника, содержащего заданную точку

TODO

D.2 Форматы хранения разреженных матриц

D.2.1 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

$$\begin{pmatrix} 2.0 & 0 & 0 & 1.0 \\ 0 & 3.0 & 5.0 & 4.0 \\ 0 & 0 & 6.0 & 0 \\ 0 & 7.0 & 0 & 8.0 \end{pmatrix} \tag{D.20}$$

Массивы, описывающие матрицу в формате CSR примут вид

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

D.2.2 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 (D.20) 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.

E Работа с инфраструктурой проекта CFDCourse

В настоящем параграфе будут даны инструкции для разворачивания инфраструктуры сборки проекта для операционных систем Linux(Ubuntu), MacOS и Windows.

В процессе настройки будет необходимо установить систему контроля версий **Git**, систему контейнеризации **Docker** и (опционально) интегрированную среду разработки. Проект позволяет работать в любой среде разработки. Ниже будут приведены инструкции для настройки **vscode**.

Процесс сборки и запуска программ будет осуществляться в системе, развёрнутой в докере на основе Ubuntu 24.04. В дальнейшем систему, установленную непосредственно на компьютере, будем называть хост системой. А систему, развёрнутую в докере – контейнером.

Для успешной установке на хосте должно быть около 5Гб свободного места.

E.1 Клонирование

Для клонирования проекта на локальный компьютер необходимо установить систему контроля версий Git и открыть терминал на хост-системе в папке, в которой планируется хранить папку с репозиторием. В нижеследующих инструкциях в качестве такой папки будет использоваться домашняя папка пользователя.

Ubuntu

Откройте терминал и в нём

```
sudo apt install git  # установка гита
cd ~                  # переходим в папку,
                      # где будет хранится папка с репозиторием
```

Windows

В Windows необходимо скачать и установить дистрибутив https://github.com/git-for-windows/git/releases/download/v2.51.0.windows.1/Git-2.51.0-64-bit.exe. Далее откройте командную строку `cmd`. И в ней перейдите в целевую папку

```
cd %USERPROFILE%
```

MacOs

Откройте терминал и в нём

```
# Установите Homebrew, если у вас его ещё нет
/bin/bash -c \
"$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
# установка гита
brew install git
# переходим в папку где будет хранится папка с репозиторием
cd ~
```

Далее необходимо клонировать репозиторий

```
git clone https://github.com/kalininei/CFDCourse26
```

В результате в домашней папке должна появиться папка с `CFDCourse26`.

E.2 Разворачивание контейнера

Установим докер

Ubuntu+apt

Запустить скрипт, написанный на основе инструкций с официального сайта https://docs.docker.com/engine/install/ubuntu/#install-using-the-repository:

```
cd ~/CFDCourse26    # перейдём в репозиторий
./scripts/ubuntu_docker_install.sh
```

Windows/MacOs+DockerDesktop

Скачайте и установите дистрибутив с официального сайта

- https://docs.docker.com/desktop/setup/install/windows-install/ для Windows
- https://docs.docker.com/desktop/setup/install/mac-install/ для MacOs

Далее следуйте процедуре установки десктопного приложения. После установки запустите `DockerDesktop`. Этап регистрации при запуске опционален и может быть пропущен.

Далее в терминале находясь в директории `CFDCourse26` развернём контейнер:

```
docker compose up --build -d
```

Если вы работаете на Unix-системе с несколькими пользователями и ваш пользователь не является дефолтным, то лучше собрать контейнер под своим пользовательским id. Для этого нужно определить необходимые переменные окружения выполнив вместо последней команды из предыдущего листинга:

```
USER_ID=$(id -u) GROUP_ID=$(id -g) docker compose up --build -d
```

Альтернативно можно перестраивать контейнер, запуская с хост системы скрипт `scripts/rebuild_container.sh`.

На этом этапе будет скачаны необходимые образы и запущен контейнер. По окончании можно убедиться, что контейнер работает

```
docker ps
```

В случае работы с `DockerDesktop` запущенный контейнер будет виден в графическом интерфейсе во вкладке `Containers`.

E.3 Базовая разработка

Чтобы скомпилировать проект необходимо войти в терминал контейнера. Далее

```
docker ps                   # Убедимся, что контейнер cfd26 запущен
docker exec -it cfd26 bash  # Войдём в него
```

На Unix системе можно использовать скрипт `scripts/attach_to_container.sh`.

Папка хоста с репозиторием `CFDCourse` примонтирована к папке контейнера `/app`. Войдём туда

```
cd /app   # заходим в директорию
ls -alh   # смотрим список файлов
```

Для удобства в контейнере установлен консольный файловый менеджер `mc`, который можно использовать для операций с файлами. Так же есть его псевдоним `mcc`, который отличается от базового `mc` тем, что запоминает текущую директорию при выходе. Благодаря чему его можно использовать для навигации вместо `cd`.

E.3.1 Особенности проекта

- Проект состоит из статически линкуемой библиотеки `libcfd26.a` и исполняемого файла `cfd26_test` с тестовыми программами для этой библиотеки. Исходники библиотеки лежат в директории `src/cfd`, исходники тестов – `src/test`,
- Используется 20-ый стандарт C++,
- Сборка осуществляется в системе cmake,
- Для написания тестов используется фреймворк `Catch2`,
- В проекте установлены жёсткие правила работы с предупреждениями компилляции, из-за которых они как обрабатываются ошибки,
- В проекте установлены форматтеры для исходных кодов на C++, python, cmake. При сохранении любого исходного файла этих форматов в vscode форматтер будет вызван автоматически. Если исходники модифицировались иначе, то запустить форматтер для всех файлов проекта можно скриптом `scripts/formatall.sh` из папки `/app`.

E.3.2 Сборка в отладочном режиме

Из папки `/app` контейнера:

```
mkdir build                         # создаём папку, в которую будут строится программа
cd build                            # Заходим
cmake .. -DCMAKE_BUILD_TYPE=Debug   # Строим сборочные скрипты
make -j4                            # Компилируем на 4-х потоках
```

Сама папка
`build` является временной папкой для построения. Она не подключена к системе контроля версий. И может быть удалена и создана заново (например для полной гарантированной очистки кэша построения). Бинарные файлы будут построены в папку `/app/build/bin`. Для запуска тестов зайдём в эту папку:

```
cd bin
./cfd26_test                # запуск всех тестов
./cfd26_test [grid1]        # запуск единственного тест кейса
```

E.3.3 Сборка в релизном режиме

Отличается от сборки в отладочном режиме только флагом `cmake`. Будем строить в папку `build_release`. Также от папки `/app`

```
mkdir build_release                          # создаём папку
cd build_release                             # Заходим
cmake .. -DCMAKE_BUILD_TYPE=RelWithDebInfo   # Или просто Release,
                                             # если отладочная информация не нужна
make -j4                                     # Компилируем на 4-х потоках
cd bin                                       # заходим в папку с исполняемым файлом
./cfd26_test                                 # запуск всех тестов
./cfd26_test [grid1]                         # запуск единственного тест кейса
```

E.3.4 Работа с кодом

Работать с исходными файлами можно находясь на хосте и используя любой удобный текстовый редактор. (например
`notepad++` на Windows). Порядок работы будет выглядеть следующим образом

- Редактировать исходные файлы на хосте в папке `CFDCourse26/src`
- Для сборки переключиться на терминал и выполнить вышеописанную процедуру сборки
- Если сборка не прошла из-за ошибок в коде, эти ошибки будут распечатаны в терминал с указанием файла и номера строки
- Для отладки можно использовать консольный отладчик `gdb` из терминала

E.4 Разработка в vscode

E.4.1 Подключение к контейнеру

Необходимо установить vscode на хосте следуя инструкциям с официального сайта https://code.visualstudio.com/Download. В самом vscode нужно установить расширение
`Remote Explorer, Dev Containers` для удалённой разработки в контейнере. Далее нужно переключиться на вкладку RemoteExplorer (см. рис. 27).

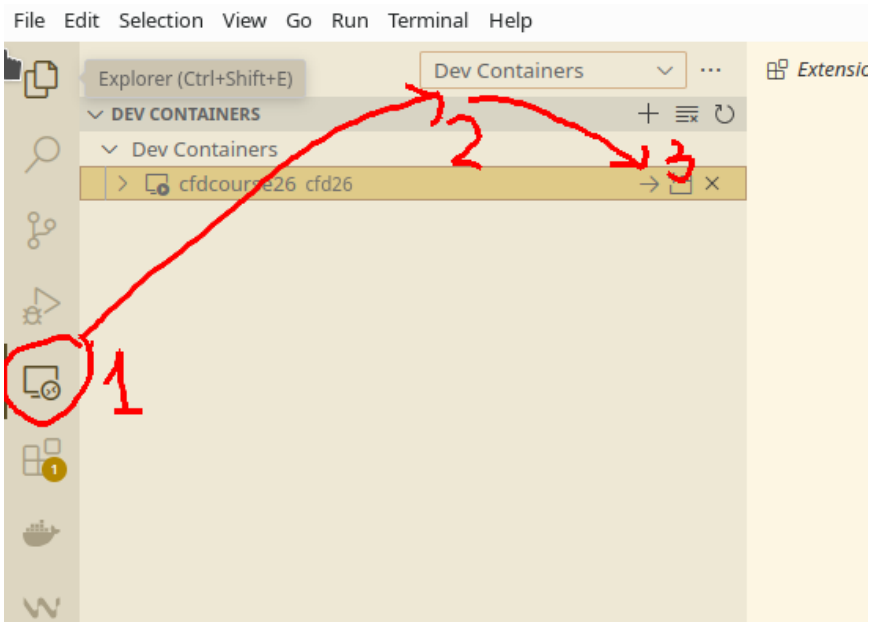


Рис. 27: Подключения vscode к контейнеру

E.4.2 Настройки vscode

Далее необходимо открыть папку `/app`: `File->Open Folder...`. Контейнер содержит в себе базовые настройки vscode, которые при сборке контейнера копируются в папки `.vscode, .vscode-server`.

Следует отметить, что удалённо подключённый vscode не может пользоваться расширениями, установленными на хосте. Все необходимые расширения нужно устанавливать в контейнер заново. Список базовых расширений, необходимых для работы, уже содержится в настроечных файлах. Когда вы откроете папку `app` в vscode, вам будет предложено эти расширения установить. Нужно согласиться.

Настроечные папки `.vscode, .vscode-server` игнорируются системой контроля версий и расположены на хосте. То есть вы можете дополнительно установить туда любые свои расширения и донастроить vscode как вам удобно. Эти настройки не будут зависеть от ветки гита и не будут затираться при пересборке контейнера.

Если всё же понадобится обнулить настройки vscode, нужно

1. удалить папки `.vscode, .vscode_server`
2. удалить все настройки из конфигурационного файла контейнера (`ctrl+shift+p, open container configuration file`)
3. выйти из контейнера на vscode: `File->Close Remote Connection`
4. остановить и пересобрать контейнер на хосте

```
docker stop cfd26
docker compose up --build -d
```

E.4.3 Сборка и отладка

В папке `.vscode` лежат базовые таски и лаунчи для компилляции и запуска программы. В случае необходимости можете дополнить базовый набор своими командами. Согласно настройкам по умолчанию по нажатию `F5` происходит сборка программы в отладочном режиме и запуск отладки тестовой программы с прогоном всех тестов. Посмотреть результаты можно на вкладке `TERMINAL`. В случае ошибок компилляции, список этих ошибок будет виден на вкладке `PROBLEMS`.

Для отладки конкретного теста необходимо запустить тестовую программу с аргументом (например `cfd26_test [grid1]`. Чтобы передать программе аргумент нужно этот аргумент прописать в файле `.vscode/launch.json` в поле `args`. Либо создать ещё одну конфигурацию запуска с вашими аргументами и указать эту конфигурацию в настройке `Run And Dubug`.

Чтобы собрать программу без запуска нужно выполнить таск (`ctrl+shift+b`):
`cmake: build debug, cmake: build release` для отладочного и релизного режима соответственно.

В целом сборка на vscode представляет из себя автоматизированный алгоритм, представленный в п. E.3. То есть исполняемая программа в дебаговой версии кладётся в папку `build`, в релизной – в `build_release` и её можно запустить из терминала. Удаление этих папок ведёт к полной очистке кэша построения.

E.5 Работа с системой контроля версий

Работать с гитом можно как с хоста (из папки `CFDCourse26`), так и из контейнера (из папки `\app`). Ниже будут даны инструкции для работы с гитом в консоли. Альтернативно, можно установить графический интерфейс (например `GitExtensions` для Windows) или командами vscode на вкладке `Source Control`.

Из системы контроля версий исключены следующие каталоги:

- build*/ – папки со сборками,
- .vscode, .vscode-server – настройки и рисширения vscode,
- local_data – папка для хранения любых пользовательских данных.

Изменения из этих папках не будут отслежены и скоммичены.

E.5.1 Порядок работы с репозиторием CFDCourse

Основная ветка проекта – `master`. После каждой лекции в эту ветку будет отправлен коммит с сообщением `lect{index}`. В этом коммите будет дополнен pdf документ с содержанием лекции, задание по итогам лекции и необходимые для этого задания изменения в коде.

E.5.1.1 Получение последнего коммита

Таким образом, **после лекции**, после того, как изменение
`lect{index}` придёт на сервер, необходимо выполнить следующие команды

```
git checkout master  # перейти на основную ветку
git pull             # получить изменения
```

Если изменения не содержали никаких изменений в настроечных файлах контейнера `Dockerfile, docker-compose.yaml`, то для сборки проекта рестарт контейнера не требуется. Иначе требуется пересобрать контейнер:

```
docker stop cfd26            # остановить текущий контейнер
docker compose up --build -d # пересобрать новый
```

E.5.1.2 Создание коммита с текущим дз

Перед началом лекции, если была сделана какая то работа по заданиям,

```
git checkout -b hw-lect{index}      # создать локальную ветку, содержащую задание
git add .
git commit -m "{свой комментарий}"  # скоммитить свои изменения в эту ветку
```

Даже если задание выполнено не до конца, вы в любой момент можете переключиться на ветку с заданием и его доделать

```
git checkout hw-lect{index}
```

E.5.1.3 Создание коммита с прошлым дз

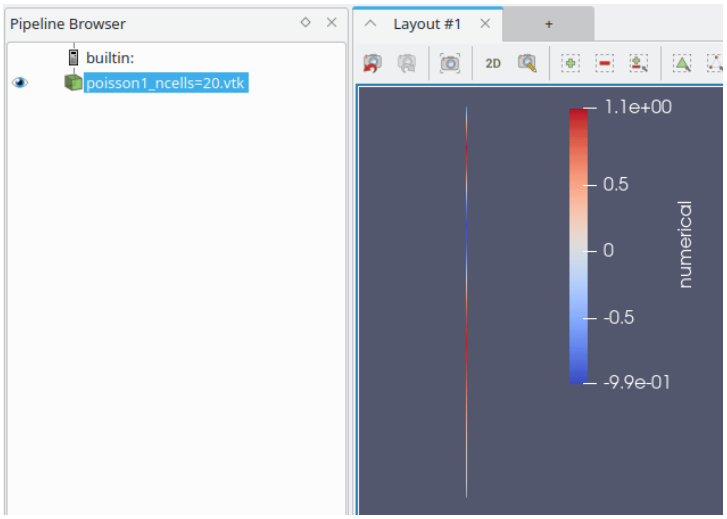
Если вы не сделали задание вовремя и решили вернутся к нему позже, то нужно

```
git checkout master            # перейти на основную ветку
git log --oneline              # в списке всех коммитов найти хэш коммита
                               # lect{index} той лекции которую нужно сделать
git checkout <...>             # переключиться на этот коммит по его хэшу
git checkout -b hw-lect{index} # создать ветку от этого коммита и работать в этой ветке
...                            # делаем работу
git add .
git commit -m "comment"        # по окончании работы скоммитить изменения
git checkout master            # и вернуться в основную ветку
```

E.6 Paraview

E.6.1 Данные на одномерных сетках

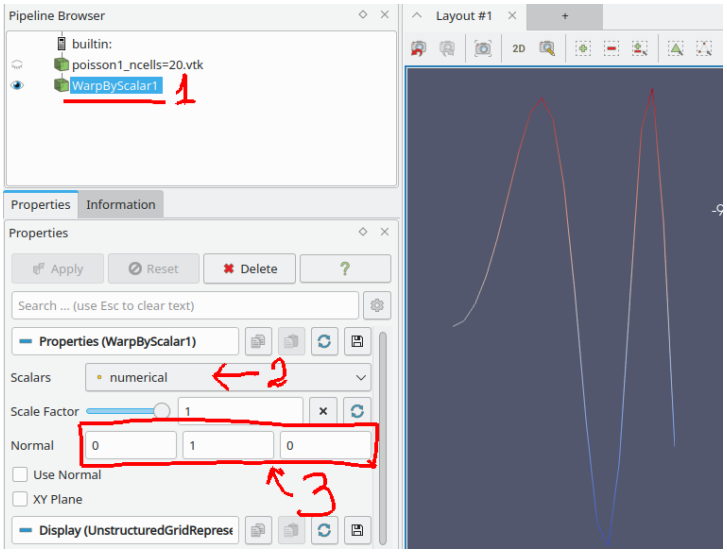
Заданные на сетке данные паравью показывает цветом. Поэтому при загрузке одномерных сеток можно видеть картинку типа



Развернуть изображение в плоскость xy

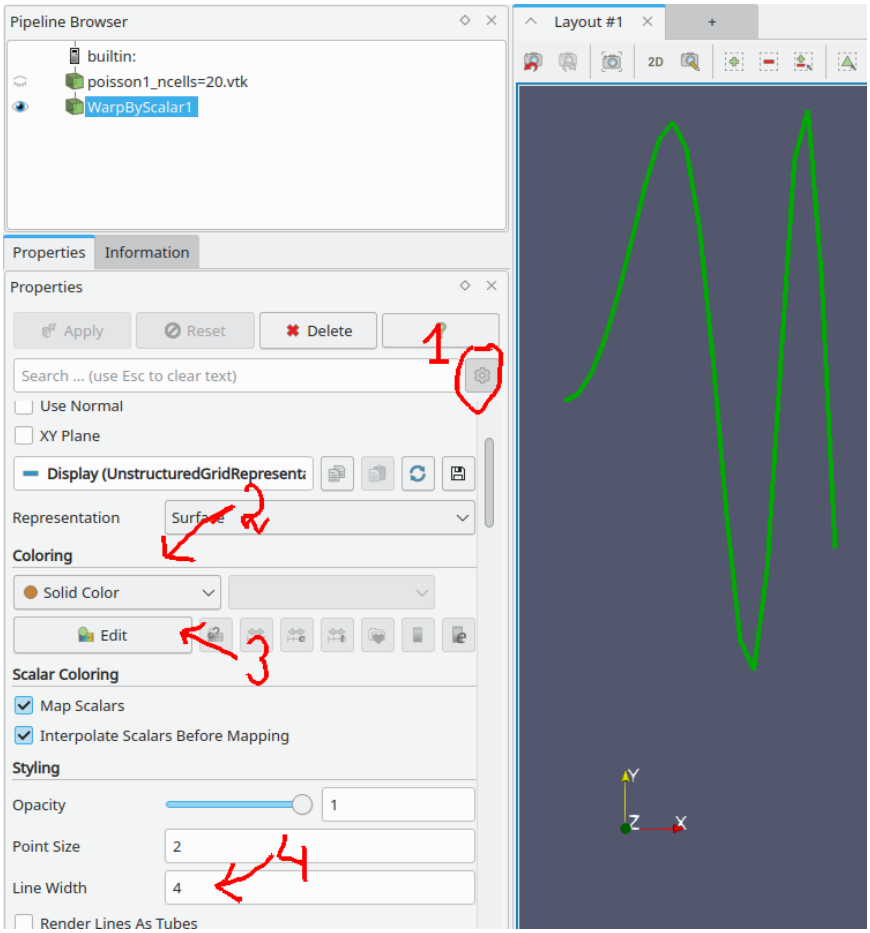
Отобразить данные в виде y-координаты Для того, что бы данные отображались в качестве значения по оси ординат, к загруженному файлу необходимо

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

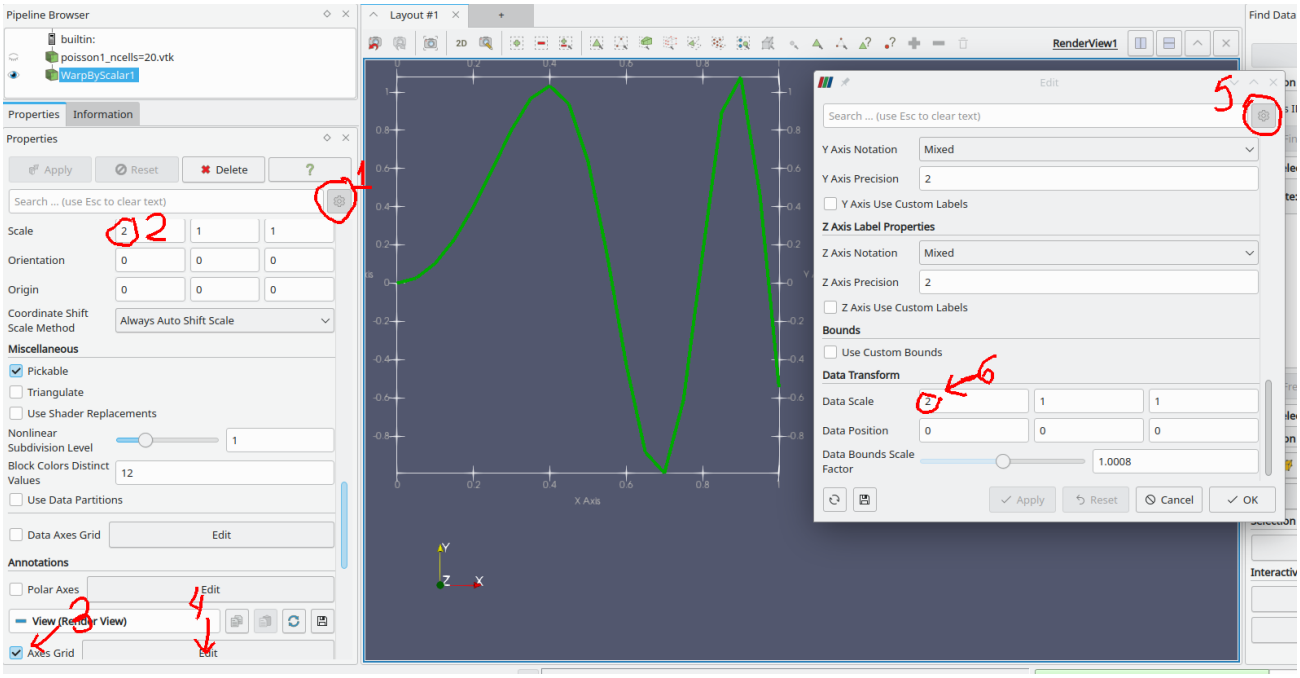
1. Включить подробные опции фильтра
2. Сменить стиль на `Solid Color`
3. В меню `Edit` выбрать желаемый цвет
4. В строке `Line Width` указать толщину линии



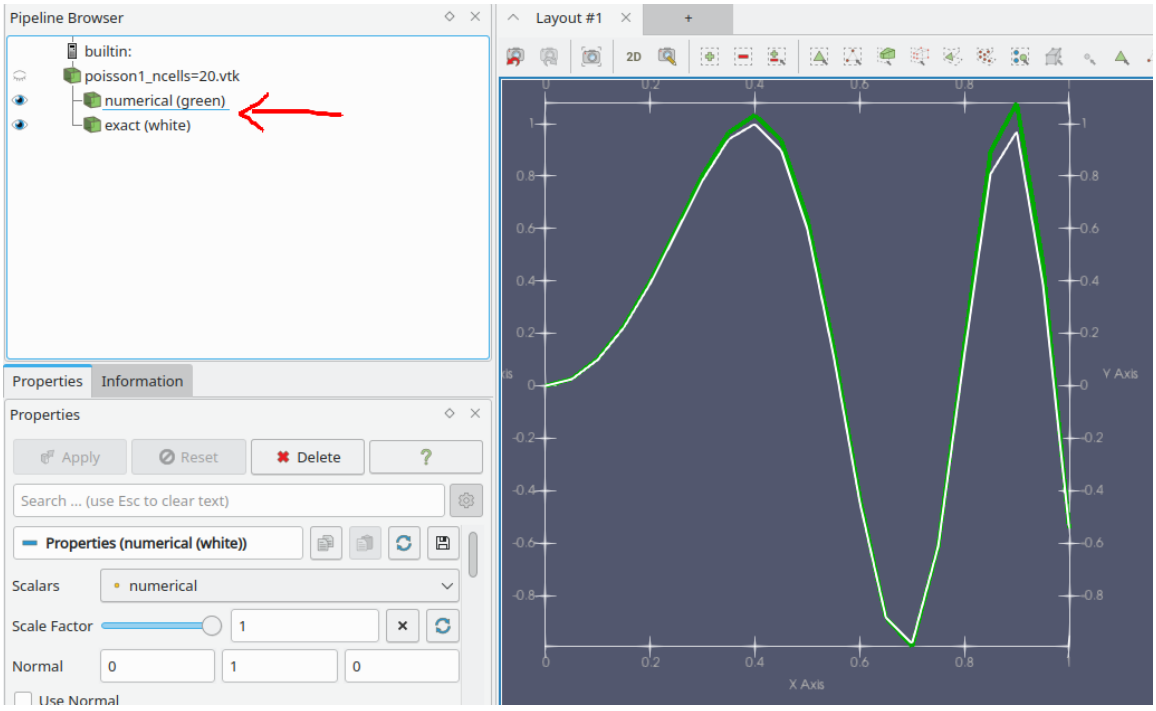
Настрока масштабов и отображение осей координат

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

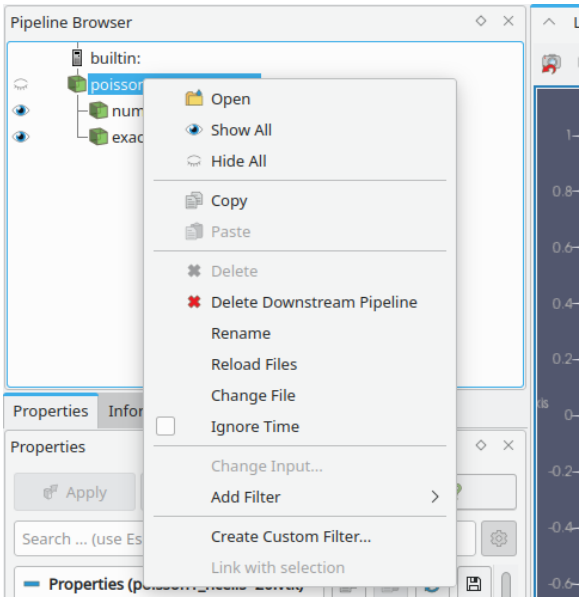
В случае, если масштабировать график не нужно, достаточно выполнить шаг 3.



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`.



E.6.2 Изолинии для двумерного поля

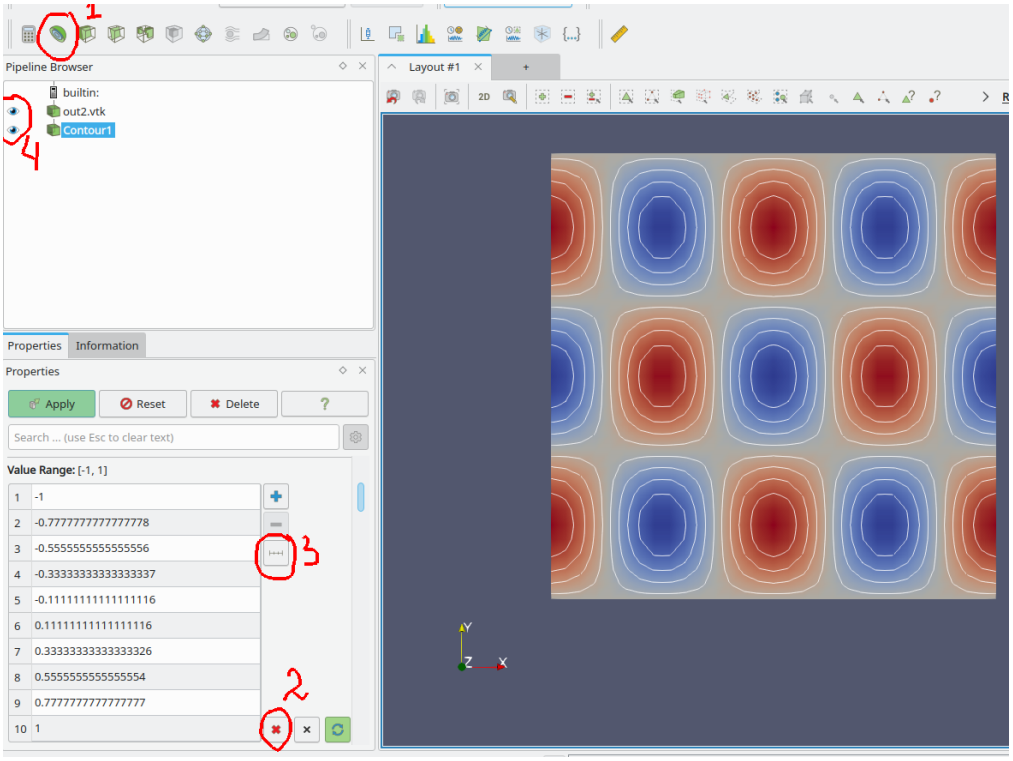
Ниже представлен алгоритм отрисовки изолиний для данных не сетке, на которой решение изветно в узлах. Если вы работаете с данными в ячейках (конечнообъёмная сетка), следует предварительно применить фильтр “Cell Data To Point Data”.

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

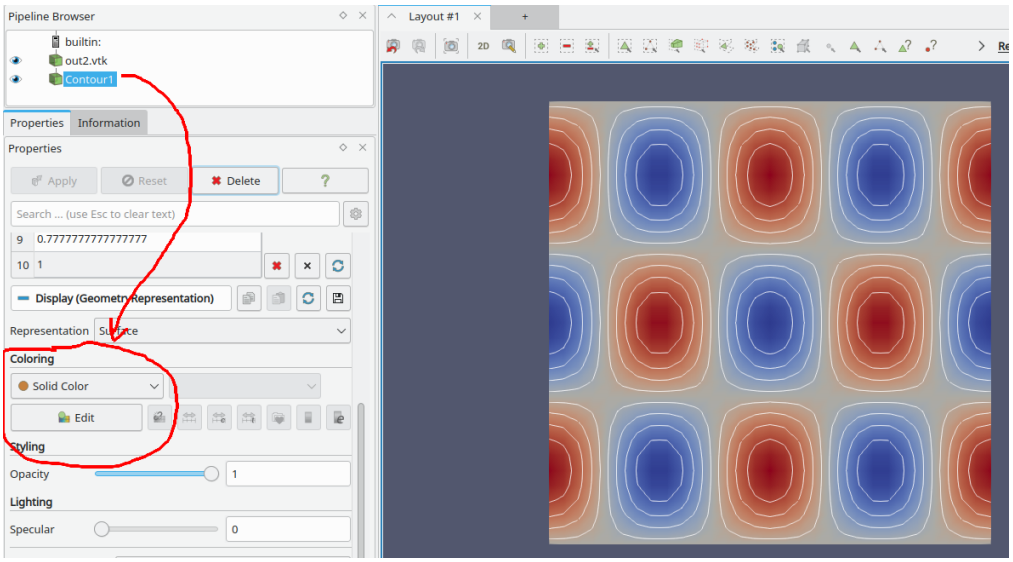
2. В настройках фильтра удалите все существующие записи о значениях для изолиний

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`.

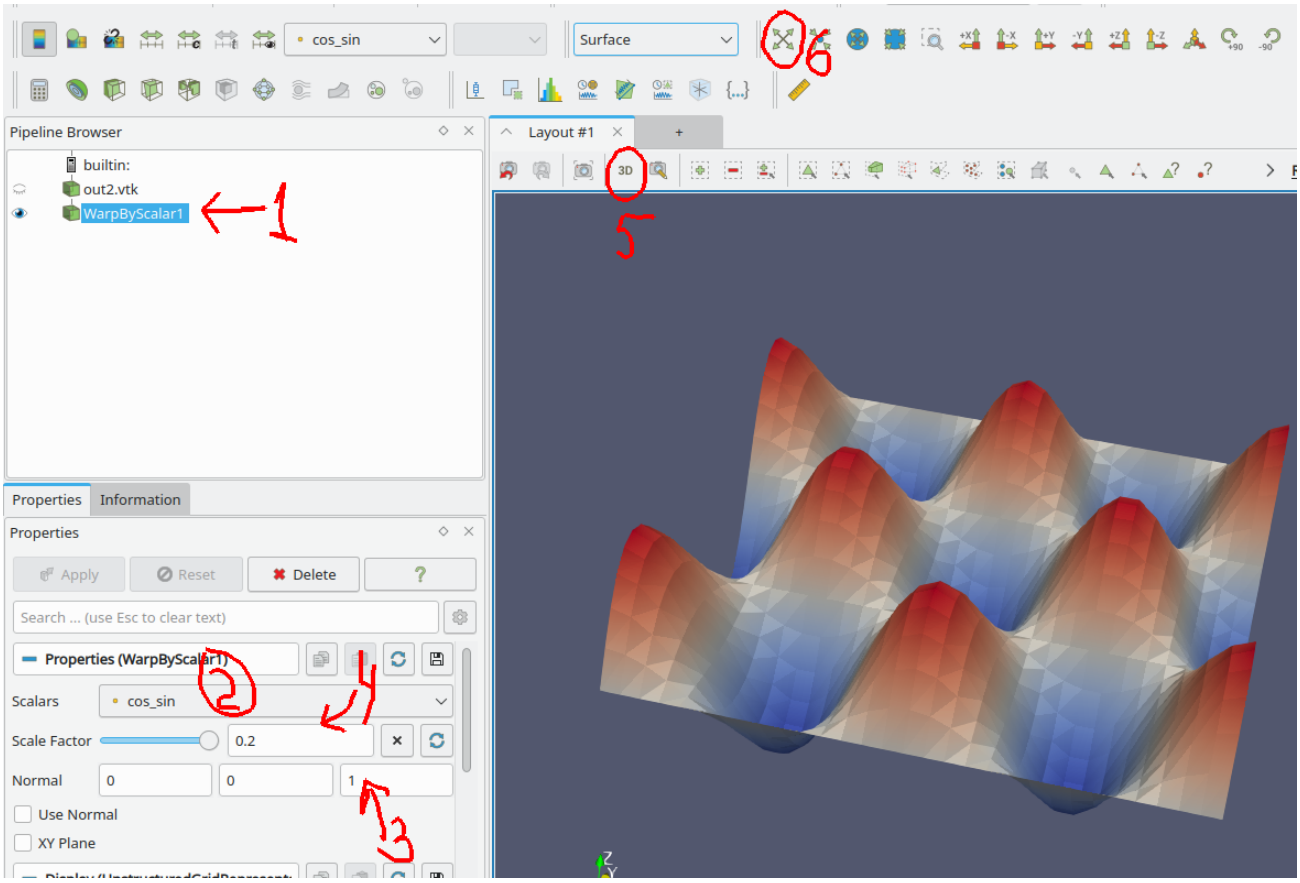


E.6.3 Данные на двумерных сетках в виде поверхности

По аналогии с одномерным графиком (п. E.6.1), 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)

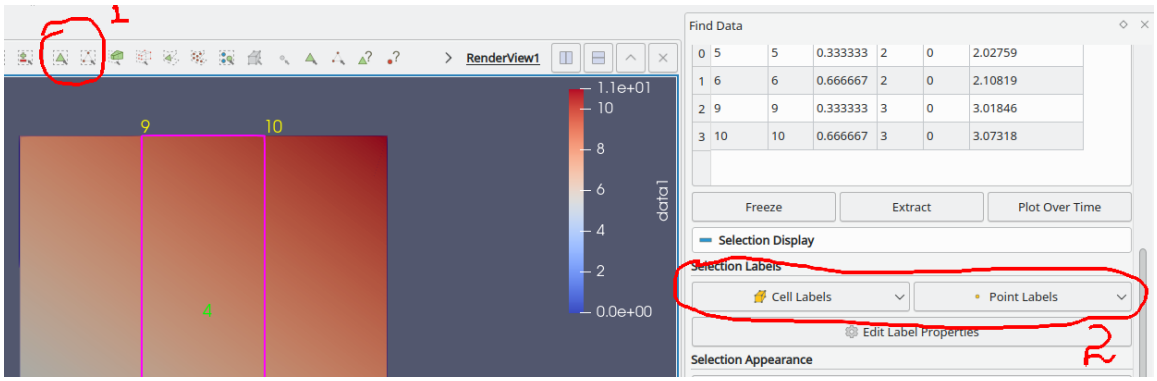
4. Если нужно, выберите масштабирования для этой координаты
5. После нажатия `Apply` включите трёхмерное отображение
6. Если данные не видно, обновите экран.



E.6.4 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

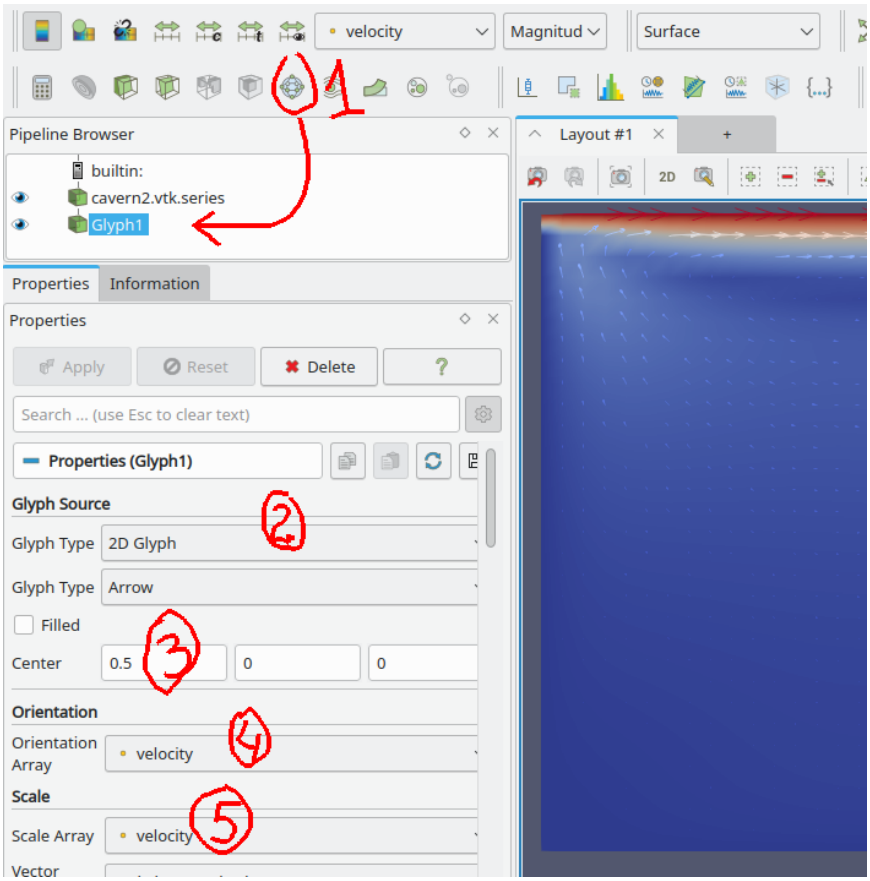
1. Включить режим выделения точек или ячеек (иконка (1 на рисунке) или горячие клавиши `s`, `d`). Выделить мышкой интересующую область
2. В окне `Find data` (или `Selection Inspector` для старых версий Paraview) отметить поле, которое должно отображаться в центрах ячеек и в точках (2 на рисунке). Если такого окна нет, включить его из основного меню `View`.



E.6.5 Векторные поля

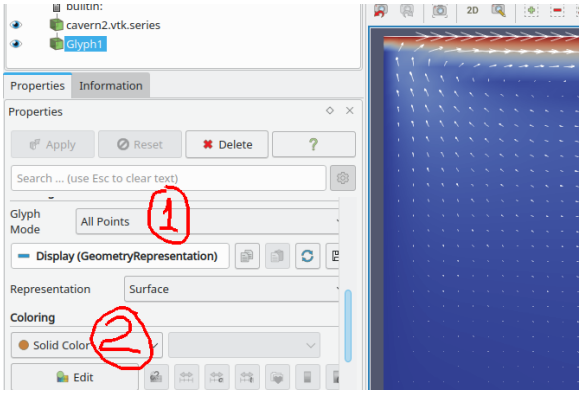
Открыть файл vtk или vtk.series, который содержит векторное поле. Далее

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`.



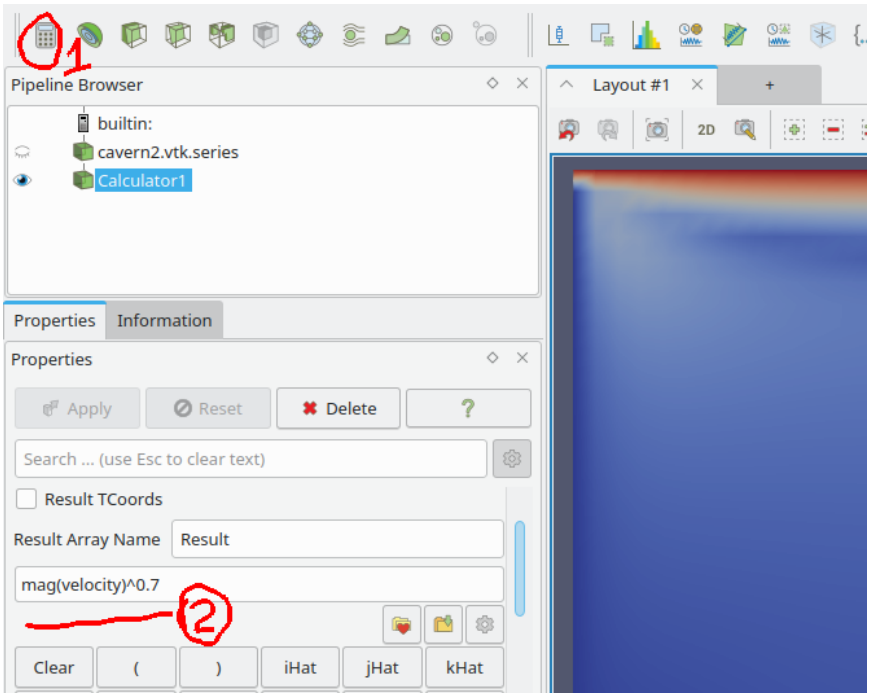
Настройка отображения стрелок

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



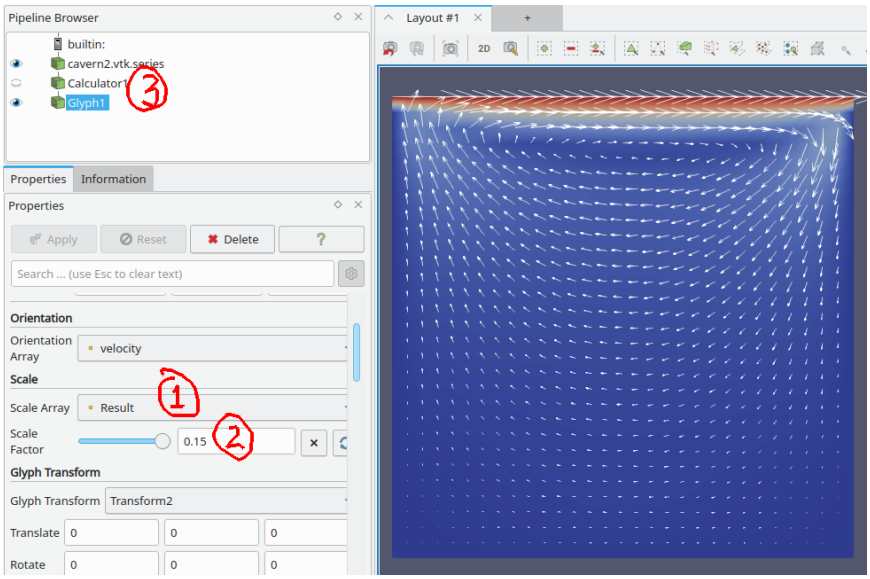
Уменьшения разброса по длине стрелок Если разброс по длинам стрелок слишком велик, его можно подравнять, введя новую функцию $|\mathbf{v}|^\alpha$ – длина вектора в степени меньше единицы (например, $\alpha = 0.7$). Такую функцию можно создать через калькулятор

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



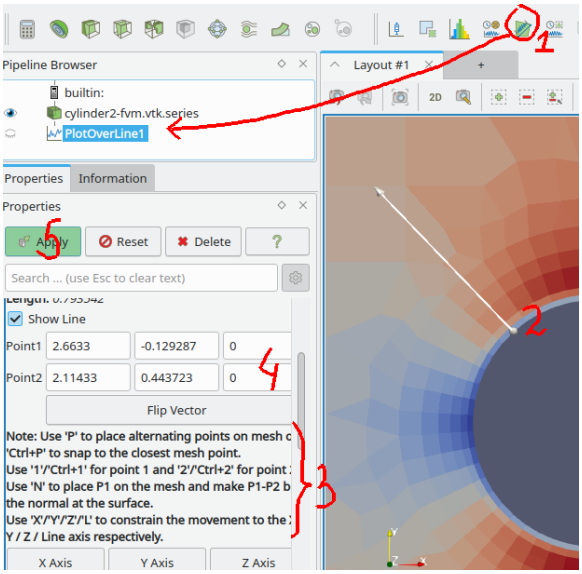
Созданную функцию нужно прокинуть в `Glyph` в качестве коэффициента масштабирования

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



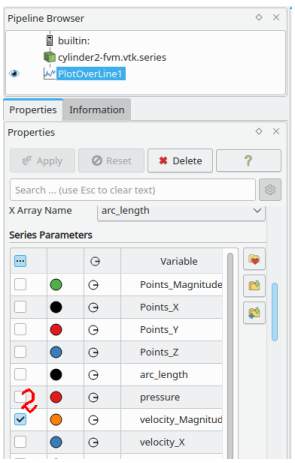
E.6.6 Значение функции вдоль линии

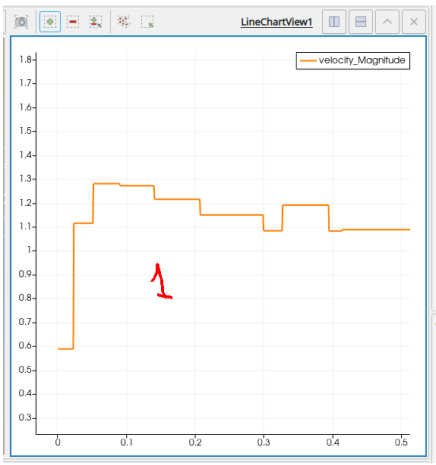
1. Выбрать фильтр `Plot Over Line` иконкой или в меню `Filters`
2. Установить начальную и конечную точку сечения
3. Можно использовать привязку к узлам сетки с помощью горячих клавиш (в подсказках написано)
4. Можно установить координаты руками в соответствующем поле. Для двумерных задач проследить, что координата Z равна нулю
5. Нажать `Apply`



Настройка графика

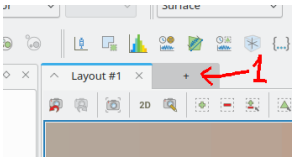
1. После установок появится дополнительное окно типа `Line Chart View` с нарисованным графиком.
2. Сделав это окно активным в настройках фильтра `PlotOverLine` можно выбрать, какие поля рисовать (`Series Parameters`)

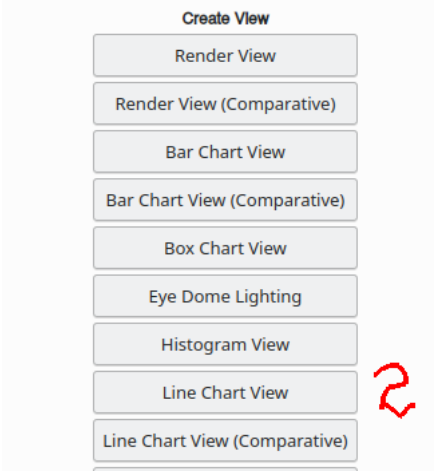


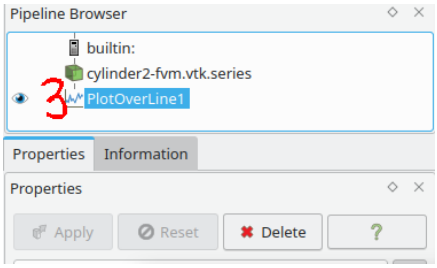


Отрисовка в отдельном окне

1. Открыть новую вкладку
2. Выбрать `Line Chart View`
3. Выбрать предварительно созданный фильтр с одномерным графиком







E.7 Hybmesh

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

E.7.1 Построение сеток

Скрипты построения сетки лежат в папке `/app/test-data`. Для запуска скрипта построения сетки следует находясь в контейнере перейти в эту папку и запустить (например для `trigrid.py`)

```
cd /app/test-data
hybmesh -sx trigrid.py
```

При первом запуске система предложит собрать и установить hybmesh из исходников. Следует согласиться, введя `y`. Первое построение может занять несколько минут.