Содержание

1 Лекция 1 (02.09)

2 Лекция 2 (09.09)

2.1 Уравнение Пуассона

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

2.1.1 Постановка задачи

Рассматривается одномерное дифференциальное уравнение вида

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

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

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

Необходимо:

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

2.1.2 Метод решения

2.1.2.1 Нахождение численного решения

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

Разностная схема второго порядка для уравнения (2.1) имеет вид

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

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

$$f_i = f(x_i). \tag{2.4}$$

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

для граничных узлов

$$\begin{aligned} u_0 &= u_a, \\ u_N &= u_b \end{aligned} \tag{2.5}$$

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

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

с матричными коэффициентами

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

и правой частью

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

Искомый вектор находится путём решения этой системы.

2.1.2.2 Практическое определения порядка аппроксимации

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

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

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

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

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

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

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

Упрощая до одномерного случая

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

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

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

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

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

такая что

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

Окончательно среднеквадратичная норма отклонения численного решения от точного запишется в виде

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

2.1.3 Программная реализация

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

В качестве аналитической тестовой функции используется

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

на отрезке $x \in [0, 1]$.

2.1.3.1 Функция верхнего уровня

объявлена как

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

В программе в цикле по набору разбиений `n_cells`

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

создаётся решатель для тестовой задачи, использующий заданное число ячеек

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

вычисляется среднеквадратичная норма отклонения численного решения от точного

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

полученное численное решение (вместе с точным) сохраняется в vtk файле `poisson1_ncells={10,20,...}.vtk`

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

а полученная норма печатается в консоль напротив количества ячеек

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

В результате работы программы в консоли должна отобразиться таблица вида

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

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

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

2.1.3.2 Детали реализации

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

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

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

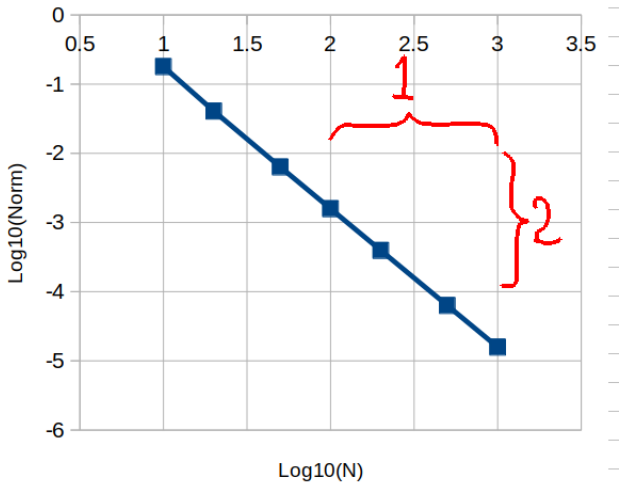


Рис. 1: Сходимость с уменьшением разбиения при решении одномерного уравнения Пуассона

```
14 class TestPoisson1Worker{
```

В методе `solve()` производится чиленное решения задачи и вычисления нормы. Для этого последовательно

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

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

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

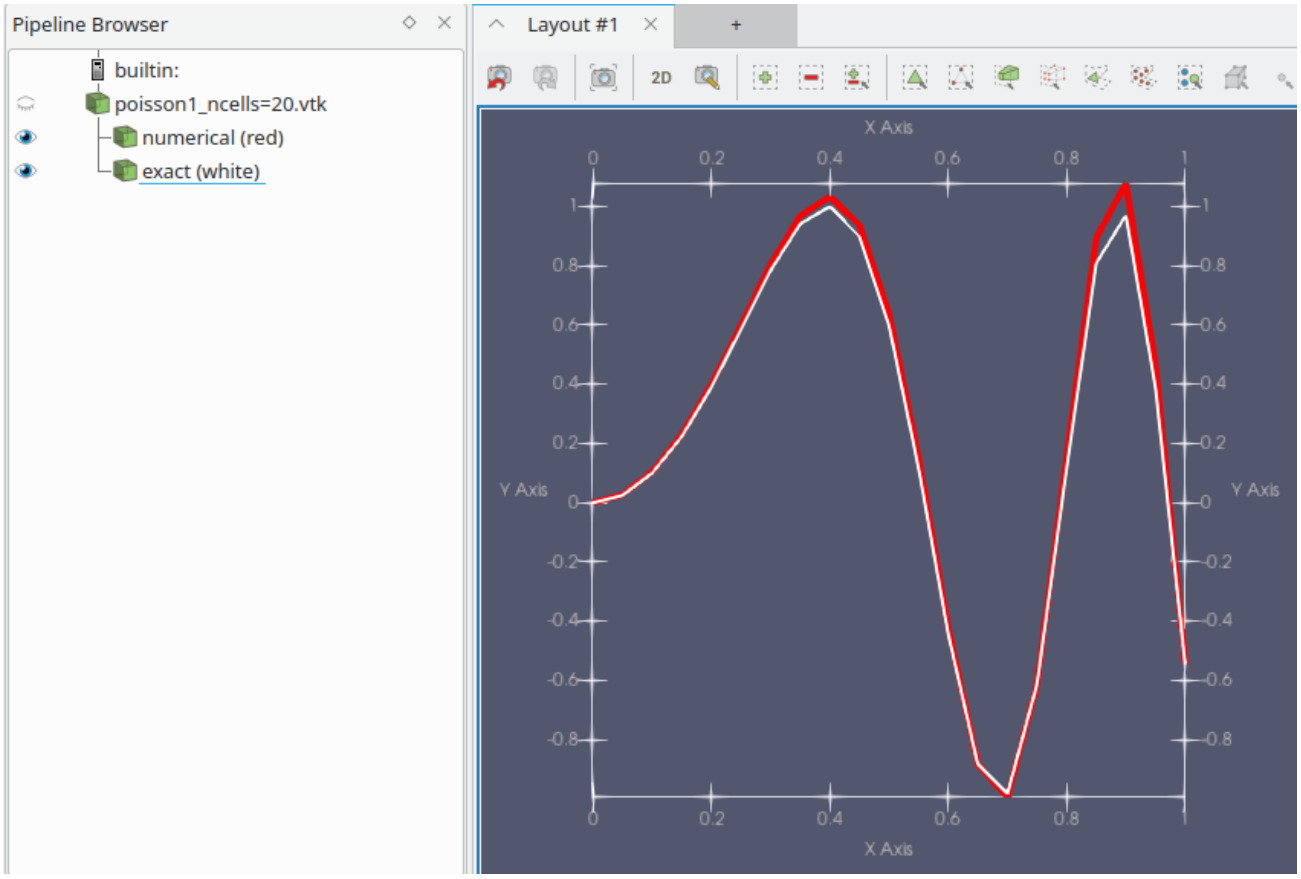


Рис. 2: Сравнение точного и численного решений уравнения Пуассона

- Сборка левой части СЛАУ. Реализует формулу (2.6). Для заполнения матрицы используется формат `cfd::LodMatrix`, удобный для непоследовательной записи, который в конце конвертируется CSR.

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

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

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

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

- Вычисление нормы. Реализует формулу (2.9).

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

2.2 Задание для самостоятельной работы

2.2.1 Порядок сходимости

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

2.2.2 Двумерное уравнение Пуассона

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

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

использовать разностную схему

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

где

$$k[i,j]=i+(n_x+1)j \tag{2.10}$$

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

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

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

Четрые угловые точки определяются как

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

Граничные неугловые точки:

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

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

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

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

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

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

3 Лекция 3 (16.09)

3.1 Двухслойные схемы для нестационарных уравнений

3.1.1 Определение

Рассмотрим дифференциальное уравнение вида

$$\frac{\partial u}{\partial t} + Lu = f, \tag{3.1}$$

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

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

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

3.1.1.1 Явная схема

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

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

или, после переноса всех известных слагаемых вправо

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

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

3.1.1.2 Неявная схема

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

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

В результате преобразования получим неявную схему первого порядка точности

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

3.1.1.3 Схема Кранка–Николсон

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

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

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

$$u\left(t\right)=u\left(t+\frac{\tau}{2}\right)-\frac{\tau}{2}\frac{\partial u}{\partial t}\bigg|_{t+\frac{\tau}{2}}+o(\tau^2)$$
$$u\left(t+\tau\right)=u\left(t+\frac{\tau}{2}\right)+\frac{\tau}{2}\frac{\partial u}{\partial t}\bigg|_{t+\frac{\tau}{2}}+o(\tau^2)$$

Взяв полусумму этих выражений получим аппроксимацию функции на промежуточном слое:

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

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

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

С учётом (3.5) симметричная разностная схема запишется как

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

или

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

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

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

3.1.1.4 Обобщённая двухслойная схема

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

$$\left(E+\theta\tau L\right)u(x,t+\tau)=\left(E+\left(\theta-1\right)\tau L\right)u(x,t)+\left(1-\theta\right)f(x,t)+\theta f(x,t+\tau). \tag{3.7}$$

Коэффициент θ – степень неявности схемы:

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

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

3.1.2 Дискретизация по времени как итерационный процесс

3.1.2.1 Двухслойный итерационный процесс

Простой двухслойный итерационный процесс определяется как

$$u^{n+1} = Au^n + b, \tag{3.8}$$

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

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

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

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

Итерационный процесс называется сходящимся, если

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

3.1.2.2 Устойчивость итерационного процесса

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

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

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

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

и проведём итерации.

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

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

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

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

3.1.2.3 Источники возмущений

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

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

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

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

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

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

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

3.2 Методы исследования устойчивости расчётных схем

3.2.1 Матричный метод

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

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

$$S(A) = \max_j |\lambda_j| \leq 1, \tag{3.9}$$

где λ_j – собственные числа матрицы A.

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

3.2.1.1 Явная схема для нестационарного уравнения диффузии

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

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

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

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

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

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

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

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

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

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

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

3.2.1.2 Неявная схема для нестационарного уравнения диффузии

Аналогично, рассмотрим неявную схему

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

В матричном виде

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

Собственные числа такой матрицы имеют вид

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

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

$$\lambda_j \geq 1.$$

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

3.2.2 Метод дискретных возмущений

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

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

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

3.2.2.1 Явная схема против потока для уравнения переноса

Для иллюстрации рассмотрим одномерное уравнение переноса

$$\frac{\partial u}{\partial t} + V\frac{\partial u}{\partial x} = 0 \tag{3.12}$$

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

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

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

Пусть $u_i = \varepsilon$. Тогда

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

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

$$C \le 1.$$

3.2.3 Метод Неймана

Запишем обратное преобразование Фурье для функции u(x):

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

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

Зададим такое начальное возмущение, которое имеет единичную амплитуду на одной частоте, соответствующей волновому числу κ_0:

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

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

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

На следующем временном шаге это возмущение примет вид:

$$\hat{u}_i = Ge^{\mathbf{i}i\theta}. \tag{3.15}$$

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

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

3.2.3.1 Неявная противопотоковая схема для уравнения переноса

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

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

Подставим (3.14), (3.15)

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

где для краткости введено обозначение

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

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

$$G + CG\left(1 - e(-1)\right) = 1 \qquad \Rightarrow$$

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

По определению комплексной экспоненты имеем

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

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

$$|1 + C(1 - \cos\theta + \mathbf{i}\sin\theta)| \geq 1 \qquad \Rightarrow$$

$$|1 + C(1 - \cos\theta) + C\mathbf{i}\sin\theta)|^2 \geq 1 \qquad \Rightarrow$$

$$1 + C^2(1 - \cos\theta)^2 + 2C(1 - \cos\theta) + C^2\sin^2\theta \geq 1 \qquad \Rightarrow$$

$$C^2(1 - \cos\theta) + 2C + C^2(1 + \cos\theta) \geq 0 \qquad \Rightarrow$$

$$C^2 + 2C \geq 0.$$

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

3.2.3.2 Противопотоковая схема Кранка-Николсон для уравнения переноса

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

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

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

$$G + \frac{CG}{2}(1 - e(-1)) = 1 - \frac{C}{2}(1 - e(-1)) \qquad \Rightarrow$$

$$G = \frac{1-p}{1+p}, \quad p = \frac{C}{2}(1 - e(-1)).$$

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

$$|1 - p|^2 \leq |1 + p|^2 \qquad \Rightarrow$$

$$(1 - \Re(p))^2 + \Im(p)^2 \leq (1 + \Re(p))^2 + \Im(p)^2 \qquad \Rightarrow$$

$$\Re(p) \geq 0$$

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

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

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

Получаем, что схема видеа (3.17) безусловно устойчива.

3.2.3.3 Явная схема для уравнения нестационарной конвекции-диффузии

Рассмотрим уравнение конвекции-диффузии

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

Сначала напишем чисто явную схему второго порядка по пространству:

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

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

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

Далее подставим (3.14), (3.15)

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

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

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

По определению комплексной экспоненты имеем

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

Отсюда

$$G = 2\gamma\cos\theta - \mathbf{i}C\sin\theta + (1 - 2\gamma)$$

Запишем квадрат модуля комплексного числа G:

$$|G|^2 = (1 - 2\gamma(1 - \cos\theta))^2 + C^2\sin^2\theta =$$
$$1 + 4\gamma^2(1 - \cos\theta)^2 - 4\gamma(1 - \cos\theta) + C^2(1 - \cos^2\theta).$$

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

$$1 + 4\gamma^2(1 - \cos\theta)^2 - 4\gamma(1 - \cos\theta) + C^2(1 - \cos^2\theta) \le 1 \qquad \Rightarrow$$
$$4\gamma^2(1 - \cos\theta)^2 - 4\gamma(1 - \cos\theta) + C^2(1 - \cos^2\theta) \le 0 \qquad \Rightarrow$$
$$4\gamma^2(1 - \cos\theta) - 4\gamma + C^2(1 + \cos\theta) \le 0 \qquad \Rightarrow$$
$$(C^2 - 4\gamma^2)\cos\theta + 4\gamma^2 - 4\gamma + C^2 \le 0$$

Поскольку неравенство должно выполняться для всех θ, а полученное выражение линейно за-

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

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

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

$$C \le 2\gamma,$$

которое с учётом определений сводится к условию на шаг по пространству, формулируемому в терминах сеточного числа Рейнольдса Re_c:

$$\frac{Vh}{k} \equiv \mathrm{Re}_c \le 2.$$

3.2.3.4 Неявная схема для уравнения нестационарной конвекции-диффузии

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

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

Подставляя представление для возмущения с волновым числом θ, получим

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

Для устойчивости необходимо

$$\begin{aligned} &|1 - \mathbf{i}C\sin\theta|^2 \le |1 - 2\gamma(\cos\theta - 1)|^2 \qquad \Rightarrow \\ &1 + C^2\sin^2\theta \le 1 + 4\gamma^2(1 - \cos\theta)^2 + 4\gamma(1 - \cos\theta) \qquad \Rightarrow \\ &C^2(1 + \cos\theta) \le 4\gamma^2(1 - \cos\theta) + 4\gamma \qquad \Rightarrow \\ &\cos\theta(C^2 + 4\gamma^2) + C^2 - 4\gamma - 4\gamma^2 \le 0 \end{aligned}$$

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

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

3.2.4 Общие рекомендации к выбору устойчивых расчётных схем

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

Неявные схемы более устойчивы, чем явные

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

Конвективное слагаемое провоцирует неусточивость, а диффузионное – напротив, добавляет устойчивость

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

3.3 Программная реализация схемы для уравнения переноса

3.3.1 Постановка задачи

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

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

Уравнение решается на отрезке $x \in [0, 1]$ с единичной скоростью $V = 1$ на сетке из 1000 ячеек. Временные итерации продолжаются до момента времени $t = 0.5$.

Начальным условием является функция вида

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

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

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

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

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

Программы реализованы в файле `transport_solve_test.cpp`.

3.3.2 Функция верхнего уровня

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

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

В начале происходит установка параметров численной схемы:

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

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

Далее вычисляются используемые шаги:

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

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

Потом устанавливается рабочий класс, в котором будет производится решение

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

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

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

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

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

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

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

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

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

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

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

Потом повторяется процедура сохранения текущего состояния решателя

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

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

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

3.3.3 Расчётные функции

Три класса-решателя для трёх заявленных задач: `TestTransport1WorkerExplicit`, `TestTransport1WorkerImplicit`, `TestTransport1WorkerCN` наследуются от одного абстрактного класса `ATestTransport1Worker`. В этом абстрактном классе реализованы все общие для всех решателей функции: создание сетки, сохранение в vtk, расчёт нормы, продвижение по времени.

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

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

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

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

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

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

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

3.3.3.1 Явная схема

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

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

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

3.3.3.2 Неявная схема

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3.3.3 Схема Кранка-Николсон

Её решатель реализован в классе `TestTransport1WorkerCN`.

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

```
200 class TestTransport1WorkerCN: public TestTransport1WorkerImplicit{
```

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

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

и правой части (3.17) с учётом граничных условий

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

3.3.4 Анализ результатов работы

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

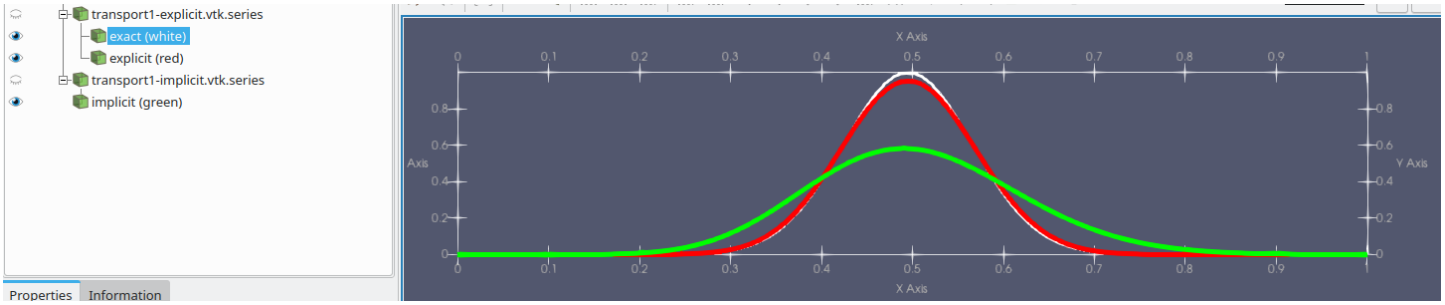


Рис. 3: Сравнение точного (белая линия) решения и численных решений по явной (красная) и неявной (зелёная) схемам

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

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

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

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

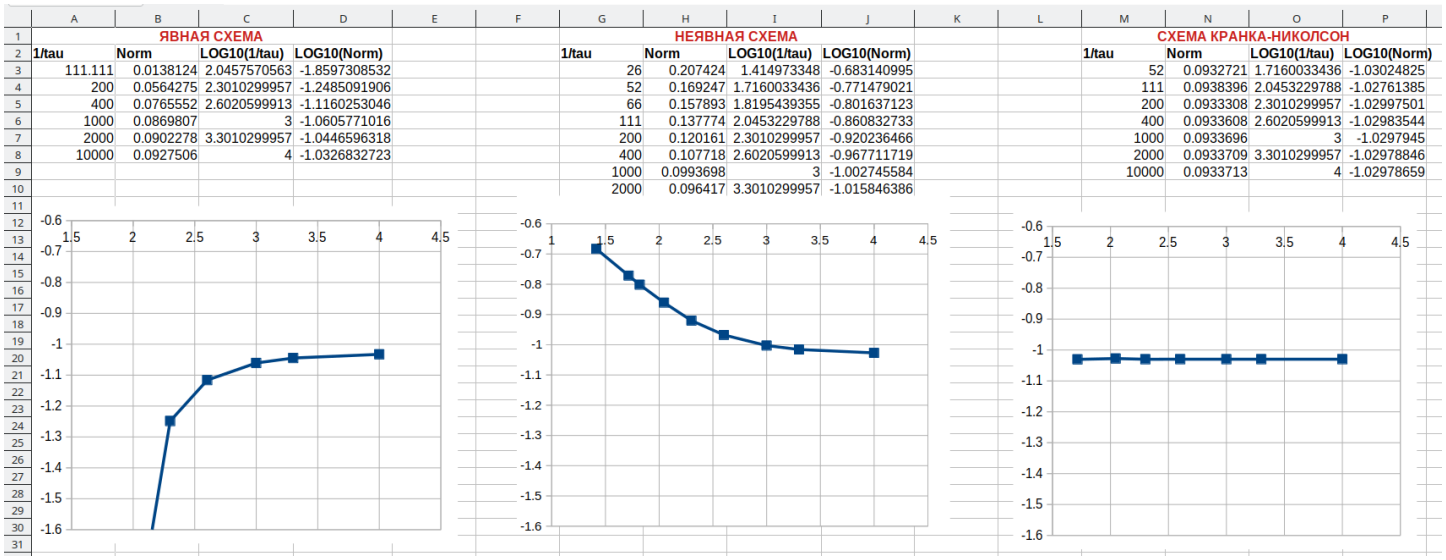
и построить график полученной таблицы в логарифмических осях. При задании диапазона изменений C следует учитывать, что явная схема устойчива только при $C \leq 1$, в то время как две другие схемы безусловно устойчивы.

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

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

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

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



ЯВНАЯ СХЕМА			
1/tau	Norm	LOG10(1/tau)	LOG10(Norm)
111.111	0.0138124	2.0457570563	-1.8597308532
200	0.0564275	2.3010299957	-1.2485091906
400	0.0765552	2.6020599913	-1.1160253046
1000	0.0869807	3	-1.0605771016
2000	0.0902278	3.3010299957	-1.0446596318
10000	0.0927506	4	-1.0326832723

НЕЯВНАЯ СХЕМА			
1/tau	Norm	LOG10(1/tau)	LOG10(Norm)
26	0.207424	1.414973348	-0.683140995
52	0.169247	1.7160033436	-0.771479021
66	0.157893	1.8195439355	-0.801637123
111	0.137774	2.0453229788	-0.860832733
200	0.120161	2.3010299957	-0.920236466
400	0.107718	2.6020599913	-0.967711719
1000	0.0993698	3	-1.002745584
2000	0.096417	3.3010299957	-1.015846386

СХЕМА КРАНКА-НИКОЛСОН			
1/tau	Norm	LOG10(1/tau)	LOG10(Norm)
52	0.0932721	1.7160033436	-1.03024825
111	0.0938396	2.0453229788	-1.02761385
200	0.0933308	2.3010299957	-1.02997501
400	0.0933608	2.6020599913	-1.02983544
1000	0.0933696	3	-1.0297945
2000	0.0933709	3.3010299957	-1.02978846
10000	0.0933713	4	-1.02978659

Рис. 4: Сходимость решения уравнения переноса с уменьшением τ

3.4 Задание для самостоятельной работы

3.4.1 Постановка задачи

Написать двумерный решатель для уравнения переноса

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

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

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

3.4.1.1 Тестовый пример 1

На этапе первичного тестирования использовать значения скорости

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

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

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

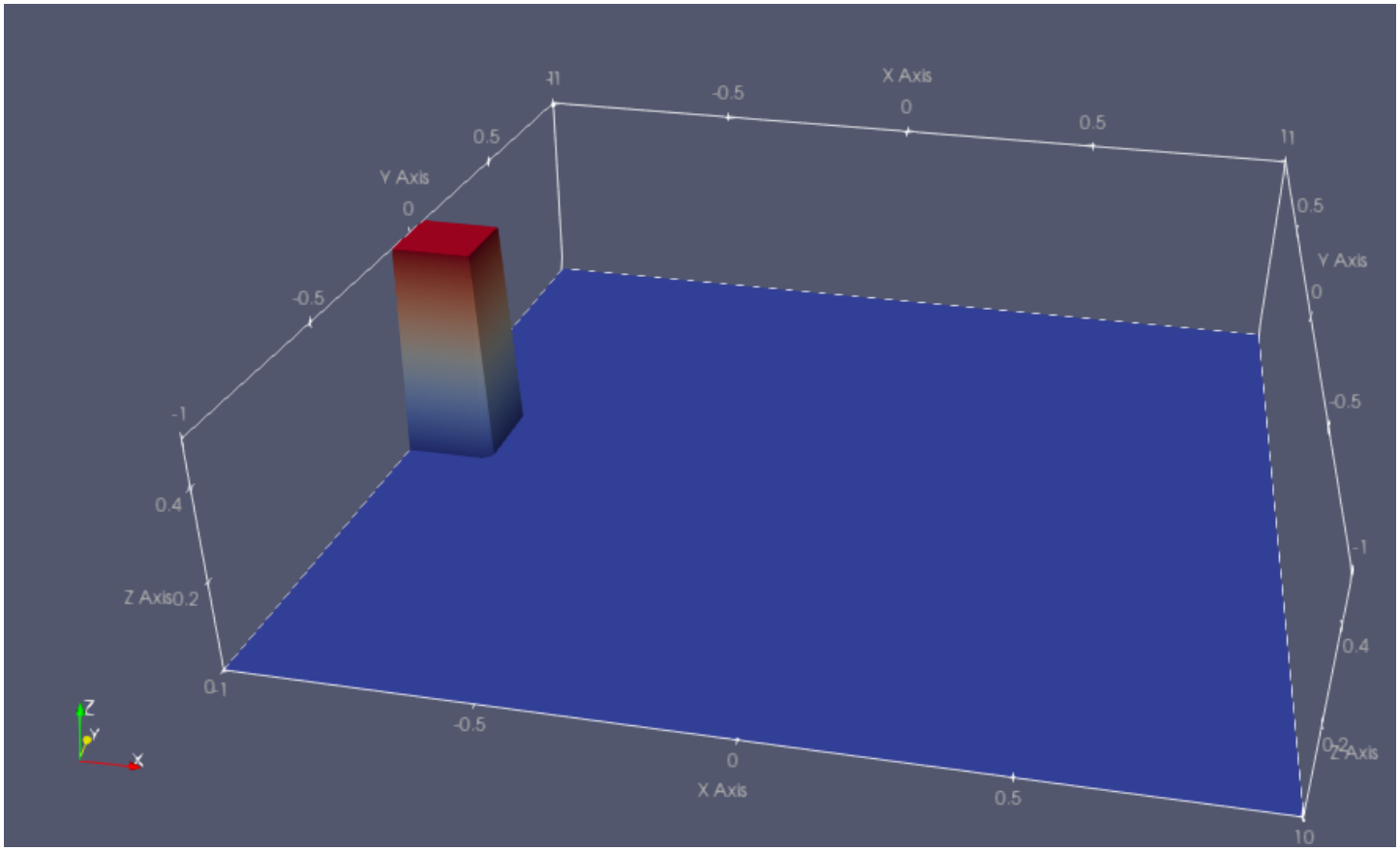


Рис. 5: Начальные условия для первого тестового примера

Точным решением будет функция

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

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

3.4.1.2 Тестовый пример 2

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

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

и начальным решением вида (рис. 6)

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

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

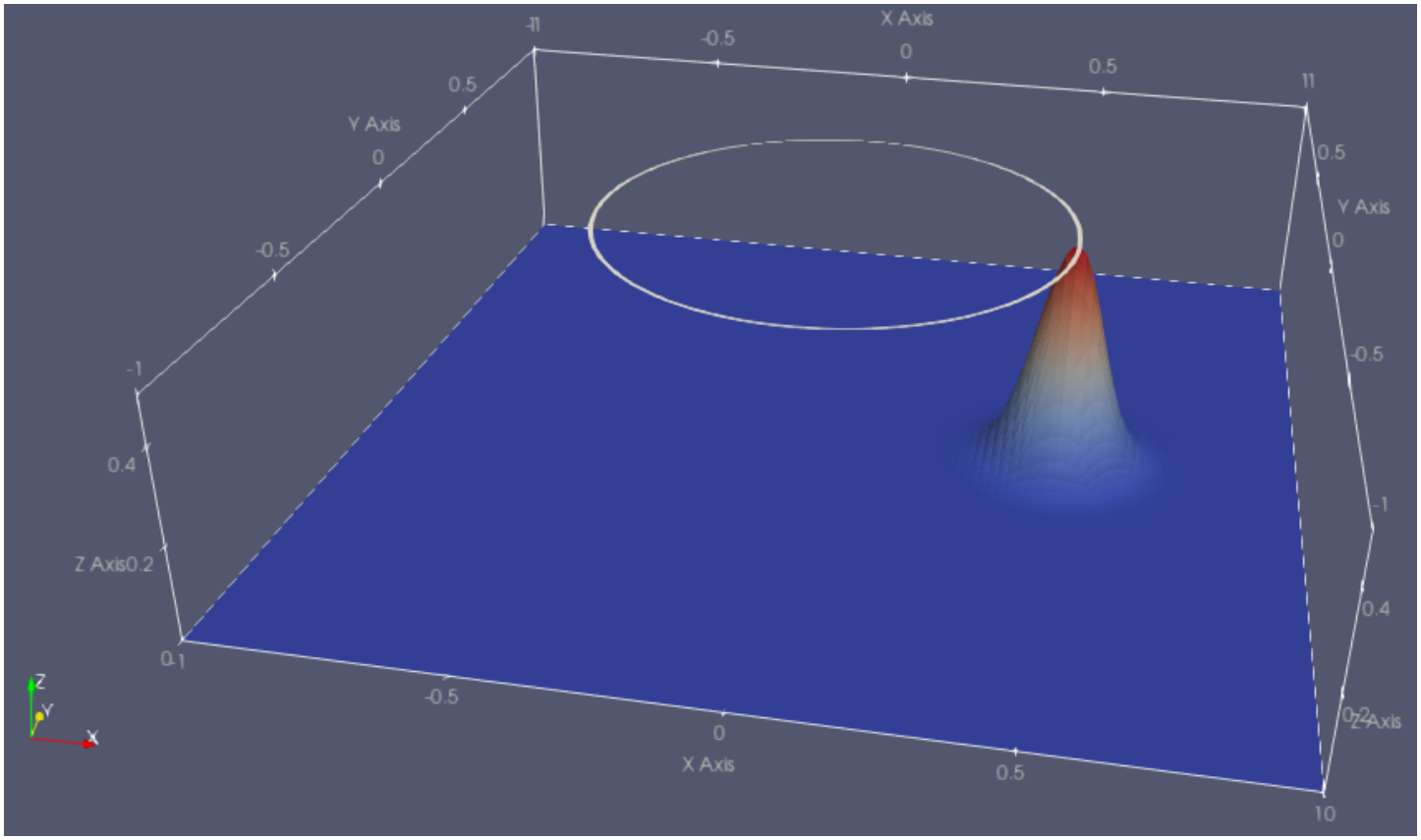


Рис. 6: Начальные условия для второго тестового примера

Точное решение на момент времени t будет иметь вид

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

3.4.2 Расчётная схема

Использовать противопотоковую явную схему:

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

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

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

индексу k (2.10) и обратно (2.11) запишем

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

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

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

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

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

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

4 Лекция 4 (30.09)

4.1 Моделирование течения вязкой несжимаемой жикости методом конечных разностей

4.1.1 Система уравнений Навье-Стокса

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

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

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

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0. \tag{4.3}$$

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

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

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

где ρ, [кг/м^3] – постоянная (вследствии несжимаемости) плотность жидкости, а μ, [Па·с] – динамическая вязкость жидокости.

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

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

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

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

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

4.1.2 Схема расчёта

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

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

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

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

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

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

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

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

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

$$\max(\|r_u\|,\|r_v\|)<\varepsilon. \tag{4.7}$$

4.1.2.1 Метод SIMPLE

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

Распишем искомые перенные в виде суммы

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

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

$$u^*+\tau\frac{\partial uu^*}{\partial x}+\tau\frac{\partial vu^*}{\partial y}-\frac{\tau}{\mathrm{Re}}\left(\frac{\partial^2u^*}{\partial x^2}+\frac{\partial^2u^*}{\partial y^2}\right)=-\tau\frac{\partial p}{\partial x}+u, \tag{4.9}$$

$$v^*+\tau\frac{\partial uv^*}{\partial x}+\tau\frac{\partial vv^*}{\partial y}-\frac{\tau}{\mathrm{Re}}\left(\frac{\partial^2v^*}{\partial x^2}+\frac{\partial^2v^*}{\partial y^2}\right)=-\tau\frac{\partial p}{\partial y}+v. \tag{4.10}$$

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

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

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

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

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

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

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

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

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

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

1. Из уравнений (4.9), (4.10) вычисляются значения u^*, v^*;
2. Они используются для вычисления правой части уравнения (4.14), в результате решения которого находится поправка давления p';
3. Дифференцируя найденную поправку давления найдём поправки скорости u', v' из выражений (4.12), (4.13);
4. Окончательно выразим значения переменных для текущего слоя из (4.8). Для улучшения стабильности алгоритма значение давления вычисляют с некоторым коэффициентом релаксации α_p:
$$\hat{p} = p + \alpha_p p';$$
5. Далее проводится вычисление невязки с ипользованием найденных значений $\hat{u}, \hat{v}, \hat{p}$ из выражения (4.7). Если она недостаточно мала, то выполняется присваивание $u = \hat{u},\ v = \hat{v},\ p = \hat{p}$ и возвращение на шаг 1.

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

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

4.1.3 Пространственная аппроксимация

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

4.1.3.1 Разнесённая сетка

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

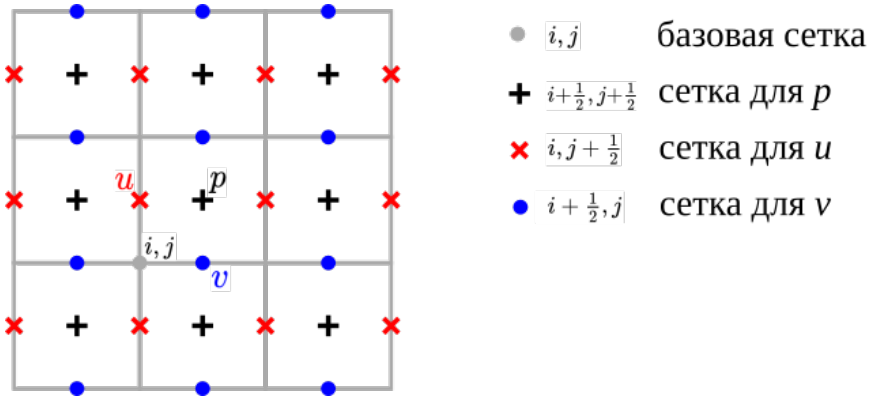


Рис. 7: Разнесённая сетка

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

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

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

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

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

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

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

$$k(i, j) = j(n_x + 1) + i. \tag{4.15}$$

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

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

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

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

4.1.3.2 Уравнения движения

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

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

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

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

Тогда

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

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

$$A^u u^* = b^u. \tag{4.20}$$

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

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

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

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

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

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

Окончательно запишем все пять ненулевых вхождений в строку матрицы:

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

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

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

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

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

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

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

$$A^v v^* = b^v, \tag{4.22}$$

элементы пятидиагональной матрицы которой имеют вид

$$A^v\left[k\left(i+\tfrac{1}{2},j\right),k\left(i+\tfrac{1}{2},j\right)\right] = 1+\frac{2\tau}{\mathrm{Re}}\left(\frac{1}{h_x^2}+\frac{1}{h_y^2}\right) \quad - \text{ основная диагональ,} \tag{4.23}$$

$$A^v\left[k\left(i+\tfrac{1}{2},j\right),k\left(i+\tfrac{3}{2},j\right)\right] = -\frac{\tau}{\mathrm{Re}}\frac{1}{h_x^2}+\frac{\tau}{4h_x}\left(u_{i+1,j+\frac{1}{2}}+u_{i+1,j-\frac{1}{2}}\right) \quad - \text{ первая верхняя диагональ,}$$

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

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

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

Правая часть аппроксимируется в виде

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

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

4.1.3.3 Уравнение для поправки давления

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

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

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

$$A^p p' = b^p, \tag{4.25}$$

где ненулевые коэффициенты пятидиагональной матрицы примут вид

$$A^p[k(i+\tfrac{1}{2},j+\tfrac{1}{2}),k(i+\tfrac{1}{2},j+\tfrac{1}{2})] = \frac{1}{h_x^2}\left(d^u_{i+1,j+\frac{1}{2}} + d^u_{i,j+\frac{1}{2}}\right) + \frac{1}{h_y^2}\left(d^v_{i+\frac{1}{2},j} + d^v_{i+\frac{1}{2},j+1}\right), \tag{4.26}$$

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

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

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

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

(4.27)

Столбец свободных членов аппроксимируется в виде

$$b^p[k(i+\tfrac{1}{2},j+\tfrac{1}{2})] = -\frac{1}{\tau}\left(\frac{u^*_{i+1,j+\frac{1}{2}} - u^*_{i,j+\frac{1}{2}}}{h_x} + \frac{v^*_{i+\frac{1}{2},j+1} - v^*_{i+\frac{1}{2},j}}{h_y}\right). \tag{4.28}$$

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

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

Пространственная аппроксимация выражения (4.11) приводит к системе уравнений

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

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

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

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

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

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

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

и аналогично из (4.23)

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

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

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

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

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

4.1.3.4 Уравнение для поправки скорости

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

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

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

$$\tag{4.33}$$

4.1.3.5 Учёт граничных условий

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

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

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

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

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

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

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

Рассмотрим нижнюю границу $j = 0$.

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

$$
\begin{aligned}
A^v[k(i+\tfrac{1}{2},0),s] &= \delta_{ks}, \quad \forall i,\ \forall s\\
b^v[k(i+\tfrac{1}{2},0)] &= v^\Gamma.
\end{aligned}
\tag{4.35}
$$

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

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

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

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

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

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

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

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

или

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

Подставляя это выражение в строку (4.36) получим

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

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

$$A^u[k(i,\tfrac{1}{2}), k(i,\tfrac{1}{2})] \mathrel{-}= A^u[k(i,-\tfrac{1}{2})], \tag{4.37}$$

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

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

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

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

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

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

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

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

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

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

4.2 Программа для расчёта течения в каверне по схеме SIMPLE

4.2.1 Постановка задачи

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

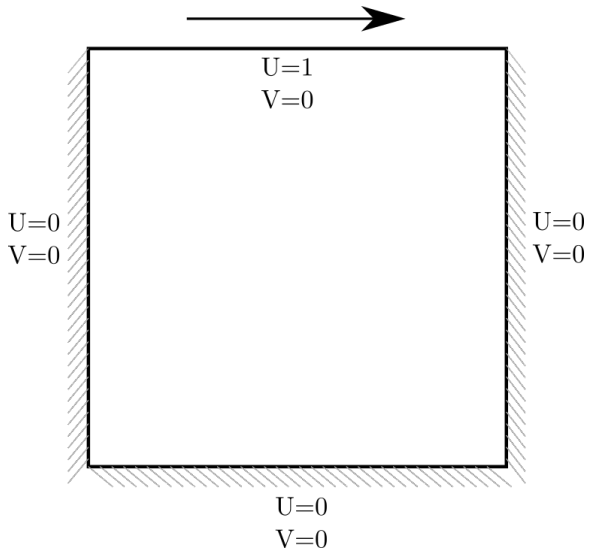


Рис. 8: Область расчёта задачи о каверне

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

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

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

Для отображения вектора поля скорости в Paraview см. справку в B.3.5.

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

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

и функции перевода индексов

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

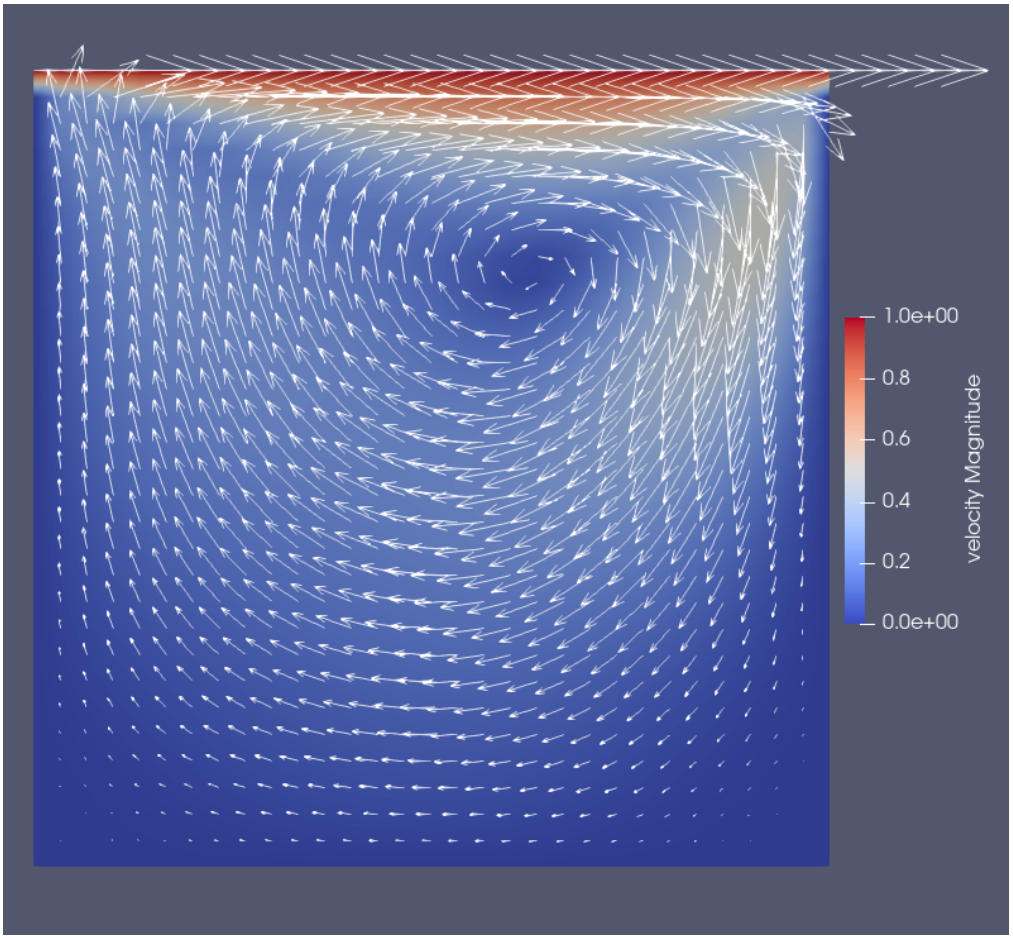


Рис. 9: Область расчёта задачи о каверне

4.2.2 Функция верхнего уровня

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

Сначала устанавливаются параметры задачи: число Рейнолдса,

```
449   double Re = 100;
```

параметры алгоритма SIMPLE,

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

разбиение сетки,

```
452   size_t n_cells = 30;
```

максимальное количество итераций

```
453   size_t max_it = 10000;
```

и значение невязки, при котором итерации прекращаются

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

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

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

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

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

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

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

и начинается итерационный процесс.

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

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

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

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

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

Сохраняется состояние решателся на пройденную итерацию

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

и производится проверка на сходимость

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

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

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

4.2.3 Поля класса решателя

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

Среди постоянных полей заданы 4 сетки: основная `_grid`, “чёрная” сетка `_cc_grid` (cell-centered) для давления, “красная” сетка `_yf_grid` (y-face) для u, “синяя” сетка `_xf_grid` (x-face) для v (рис. 7).

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

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

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

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

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

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

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

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

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

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

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

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

4.2.4 Инициализация решателя

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

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

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

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

В функции происходит переписывание переданных векторов в приватные поля класса.

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

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

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

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

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

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

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

4.2.5 Шаг итерации SIMPLE

Осуществляется в процедуре

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

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

4.2.6 Сборка системы уравнений для поправки давления

Сборка системы уравнений (4.25) осуществляется в процедуре

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

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

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

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

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

Внутри цикла устанавливаются флаги, характеризующие граничный статус текущего узла

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

Вычисляется значение сквозного индекса по формуле (4.16)

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

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

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

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

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

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

После заполнения в матрицу вводится граничное условие (4.39)

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

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

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

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

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

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

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

потом осуществляется установка граничного условия (4.39)

```
355   rhs[0] = 0;
```

и вызывается решатель СЛАУ

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

4.2.7 Сборка системы уравнений для пробной скорости

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

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

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

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

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

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

```
225
226     size_t index_right = _grid.yface_grid_index_i_jp(_grid.nx(), j);
227     add_to_mat(index_right, {_grid.nx(), j}, 1.0);
228     _rhs_u[index_right] = 0.0;
229   }
```

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

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

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

4.3 Задание для самостоятельной работы

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

2. Нарисовать поле невязок r_u, r_v в динамике. Отметить в каком из уравнений и в каких местах

области расчёта наблюдаются наибольшие проблемы со сходимостью.

3. Решить аналогичную задачу, в которой скорость не только на верхней, но и на нижней стенке равна $U = 1$. Для этого завести новый тест `[cavern2-simple-sym]` и новый тестовый рабочий класс `Cavern2DSimpleSymWorker`, который отнаследовать от существующего класса `Cavern2DSimpleWorker`. Для того, чтобы при реализации нового класса не повторять существующий код, а пользоваться механизмом перегрузок виртуальных функций, необходимо будет произвести небольшой рефакторинг: ввести новый приватный метод класса `Cavern2DSimpleWorker`

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

5 Лекция 5 (6.10)

5.1 Оптимальные значения параметров алгоритма SIMPLE

Введем обозначение

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

где $\mathrm{H}(a,b)$ – среднее гармоническое, определяемое как

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

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

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

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

Ещё более эффективной сходимости соответствуют параметры

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

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

5.2 Нестационарное уравнение Навье-Стокса

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

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

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

5.2.1 Схема расчёта по алгоритму SIMPLE

Производную по времени будет аппроксимировать по двухслойной неявной схеме.

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

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

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

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

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

$$\begin{aligned}
&\left(1+\frac{\tau}{\triangle t}\right)u^*+\tau\frac{\partial uu^*}{\partial x}+\tau\frac{\partial vu^*}{\partial y}-\frac{\tau}{\mathrm{Re}}\left(\frac{\partial^2u^*}{\partial x^2}+\frac{\partial^2u^*}{\partial y^2}\right)=-\tau\frac{\partial p}{\partial x}+u+\frac{\tau}{\triangle t}\check{u},\\
&\left(1+\frac{\tau}{\triangle t}\right)v^*+\tau\frac{\partial uv^*}{\partial x}+\tau\frac{\partial vv^*}{\partial y}-\frac{\tau}{\mathrm{Re}}\left(\frac{\partial^2v^*}{\partial x^2}+\frac{\partial^2v^*}{\partial y^2}\right)=-\tau\frac{\partial p}{\partial y}+v+\frac{\tau}{\triangle t}\check{v}.
\end{aligned} \tag{5.4}$$

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

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

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

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

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

5.3 Завихренность и функция тока

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

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

Также определим векторное поле завихренности как

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

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

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

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

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

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

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

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

и расписывая оператор Лапласа покоординатно

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

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

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

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

5.3.1 Определение завихренности и функции тока на разнесённой сетке

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

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

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

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

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

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

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

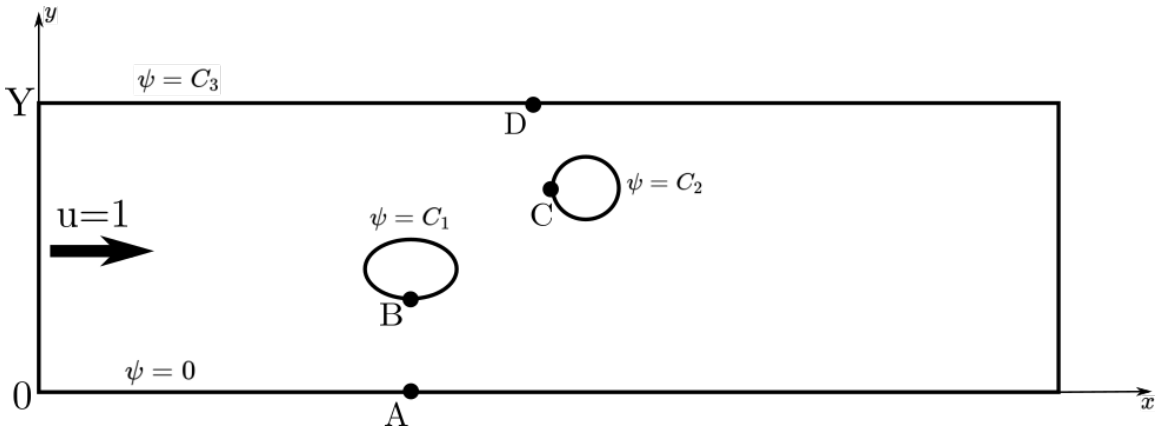


Рис. 10: Область течения в задаче обтекания

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

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

Исходя из значения ψ_{in} на верхней границе можно записать

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

Для определения ψ на первом из двух препятствий рассмотрим траекторию AB:

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

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

Аналогично для второго тела можно записать

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

5.4 Задание для самостоятельной работы

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

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

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

в функцию сохрания на основной сетке

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

6 Лекция 6 (13.10)

6.1 Оптимизация методов решения СЛАУ

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

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

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

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

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

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

6.1.1 Метод Якоби

Будем рассматривать систему уравнений вида

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

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

В классическом виде алгоритм Якоби формулируется в виде

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

Произведём некоторые преобразования

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

Таким образом, программировать итерацию этого алгоритма, обновляющую значения массива

$\{u\}$, можно в виде

$\hat{u} = u;$

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

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

endfor

$u = \hat{u};$

6.1.2 Метод Зейделя

Формулируется в виде

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

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

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

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

endfor

6.1.3 Метод последовательных верхних релаксаций (SOR)

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

Формулируется этот метод в виде

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

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

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

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

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

endfor

6.1.4 Формат хранения разреженных матриц CSR

При реализации решателей систем сеточных уравнений важно учитывать разреженный характрер 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

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

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;
}
```

6.2 Задача об обтекании препятствия

6.2.1 Расчётная сетка

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

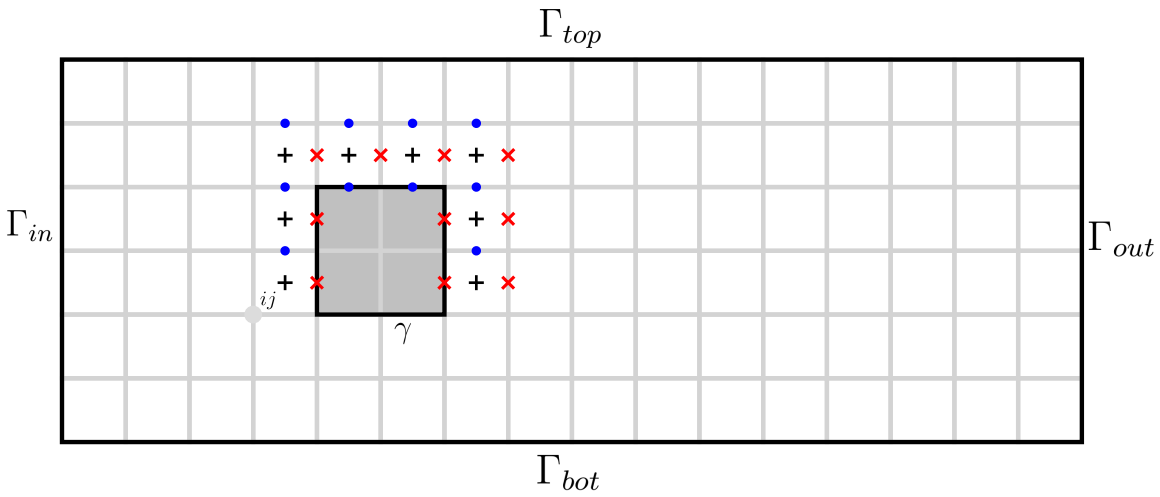


Рис. 11: Область расчёта и разнесённая сетка для задачи обтекания

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

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

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

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

6.2.2 Граничные условия

Рассмотрим постановку со следующими граничными условиями:

- во входном сечении зададим равномерный профиль скорости

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

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

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

- на самом обтекаемом деле – условия прилипания

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

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

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

6.2.2.1 Входное сечение

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

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

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

$$A^u_{km} = \delta_{km}, \quad b^u_k = 1. \tag{6.5}$$

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

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

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

$$A^v_{k_0,k_1} \mathrel{+}= a \quad \Rightarrow \quad A^v_{k_0,k_0} \mathrel{-}= a \tag{6.6}$$

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

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

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

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

Тогда добавление значения a в фиктивную колонку k_1 эквивалентно

$$A^p_{k_0,k_1} \mathrel{+}= a \quad \Rightarrow \quad A^p_{k_0,k_0} \mathrel{+}= a \tag{6.8}$$

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

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

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

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

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

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

6.2.2.2 Условия симметрии

Однородные условие для скорости (6.2) расписываются как

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

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

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

Условие на v^* можно использовать явно:

$$\begin{aligned} &k = k\left[i+\tfrac{1}{2}, \tfrac{1}{2}\right], \\ &A^u_{k,s} = \delta_{ks}, \quad b^u_k = 0. \end{aligned}$$

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

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

получим

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

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

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

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

6.2.2.3 Условия прилипания

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

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

6.2.2.4 Выходные граничные условия

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

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

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

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

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

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

$$\begin{gathered}(x,y) \in \Gamma_{out} : \frac{\hat{u} - u}{\tau} + u\frac{\partial \hat{u}}{\partial x} = 0, \\ \hat{v} = 0.\end{gathered} \tag{6.11}$$

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

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

$$v^* = 0. \tag{6.13}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

На матричном уровне получим:

$$A^u_{k0,s} = \begin{cases} 1 + \dfrac{\tau U_{k_0}}{h_x}, & s = k_0 \\ -\dfrac{\tau U_{k_0}}{h_x}, & s = k_1 \\ 0, & \text{иначе,} \end{cases}, \qquad b^u_{k_0} = u_{k_0} \tag{6.16}$$

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

$$k_0 = k\left[n_x - \tfrac{1}{2}, j\right],\ k_1 = k\left[n_x + \tfrac{1}{2}, j\right],$$
$$\frac{v^*_{k_0} + v^*_{k_1}}{2} = v^*_{\Gamma_{out}} = 0 \quad \Rightarrow \quad v^*_{k_1} = -v^*_{k_0}.$$

Отсюда добавление элемента a в фиктивную колонку будет осуществляться в виде

$$A^v_{k_0,k_1} \mathrel{+}= a \quad \Rightarrow \quad A^v_{k_0,k_0} \mathrel{-}= a.$$

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

$$k_0 = k\left[n_x - \tfrac{1}{2}, j + \tfrac{1}{2}\right],\ k_1 = k\left[n_x + \tfrac{1}{2}, j + \tfrac{1}{2}\right],$$

Из (6.15)

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

На матричном уровне добавление фиктивной колонки даёт

$$A^v_{k_0,k_1} \mathrel{+}= a \quad \Rightarrow \quad A^v_{k_0,k_0} \mathrel{+}= a,\ b^v_{k_0} \mathrel{+}= a\frac{h_x u'}{\tau d^u}. \tag{6.17}$$

6.2.3 Баланс сил. Коэффициенты сил

6.2.3.1 Сопротивление

Проинтегрируем уравнение движения (4.1) по области расчёта D:

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

Интегрирование по частям даёт:

$$\int\limits_D \frac{\partial f}{\partial x}\, d\mathbf{x} = \int\limits_{\Gamma_{out}} f\, ds - \int\limits_{\Gamma_{in}} f\, ds + \int\limits_{\gamma} f n_x\, ds$$

$$\int\limits_D \frac{\partial f}{\partial y}\, d\mathbf{x} = \int\limits_{\Gamma_{top}} f\, ds - \int\limits_{\Gamma_{bot}} f\, ds + \int\limits_{\gamma} f n_y\, ds,$$

$$\int\limits_D \nabla^2 f\, d\mathbf{x} = \int\limits_{\Gamma} \frac{\partial f}{\partial n}\, ds + \int\limits_{\gamma} \frac{\partial f}{\partial n}\, ds, \quad \Gamma = \Gamma_{in} \cap \Gamma_{out} \cap \Gamma_{bot} \cap \Gamma_{top}$$

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

$$\int\limits_{\Gamma_{in}} \left(u^2 + p\right)\, ds - \int\limits_{\Gamma_{out}} \left(u^2 + p\right)\, ds = \int\limits_{\gamma} p\, n_x\, ds - \frac{1}{\mathrm{Re}} \int\limits_{\gamma} \frac{\partial u}{\partial n}\, ds$$

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

Коэффициенты этих сил имеют следующее выражение:

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

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

6.2.3.2 Подъёмная сила

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

$$\int\limits_{\Gamma_{bot}} p\, ds - \int\limits_{\Gamma_{top}} p\, ds = \int\limits_{\gamma} p\, n_y\, ds - \frac{1}{\mathrm{Re}} \int\limits_{\gamma} \frac{\partial v}{\partial n}\, ds$$

и соответствующие коэффициенты

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

6.2.3.3 Вычисление коэффициентов сил на разнесённой сетке

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

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

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

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

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

отсюда

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

Аналогично для нижней границы

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

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

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

Тогда подинтегральное выражение равно

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

Аналогично на нижей границе получим:

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

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

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

Для коэффициента C_x^p напишем:

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

Для C_y^f:

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

6.3 Задание для самостоятельной работы

В SIMPLE-решателе для течения вязкой жидкости в каверне `[cavern2-simple]` рассмотреть простые итерационные подходы к решению систем уравнений для u^*, v^*:

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

Реализовать означенные решатели в виде функций вида:

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

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

Эти функции необходимо вызывать вместо

`AmgcMatrixSolver::solve_slae`

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

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

Все алгоритмы основаны на вычислении выражения вида

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

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

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

Использовать параметры решателя:

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

Сделать замеры времени исполнения:

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

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

Для замера времени исполнения участка кода воспользоваться функциями

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

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

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

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

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

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

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

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

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

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

Заполнить таблицу

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

Здесь Amg - исходный решатель.

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

7 Лекция 7 (20.10)

7.1 Инициализация решения

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

7.1.1 Задача о потенциале течения

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

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

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

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

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

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

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

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

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

7.1.2 Аппроксимация на разнесённой сетке

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

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

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

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

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

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

Отсюда получим

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

Поэтому уравнение (7.5) для левых узлов сетки примет вид

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

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

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

и уравнение (7.5) запишется как

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

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

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

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

$$u_{i,j+\frac{1}{2}} = \frac{\phi_{i+\frac{1}{2},j+\frac{1}{2}} - \phi_{i-\frac{1}{2},j+\frac{1}{2}}}{h_x}$$
$$u_{0,j+\frac{1}{2}} = v_n$$

7.2 Конвективный теплообмен

7.2.1 Уравнение теплопроводности

Дополним нестационарную систему уравнений течения (5.2) уравнением теплообмена

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

Здесь T – температура течения, К; ρ – плотность жидкости, кг/м^3; c_p – теплоёмкость, Дж/кг/К; λ – теплопроводность, Вт/м/К.

В безразмерном виде это уравнение примет вид

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

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

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

а число Пекле Pe есть

$$\mathrm{Pe} = \frac{UL}{a}, \quad a = \frac{\lambda}{\rho c_p}.$$

7.2.2 Дискретизация по времени

Пользуясь обозначениями из п. 5.2.1 запишем неявную дискретизацию по времени уравнения (7.6) в виде

$$\frac{\hat{T} - \check{T}}{\triangle t} + \hat{u}\frac{\partial \hat{T}}{\partial x} + \hat{v}\frac{\partial \hat{T}}{\partial y} = \frac{1}{\mathrm{Pe}}\left(\frac{\partial^2 \hat{T}}{\partial x^2} + \frac{\partial^2 \hat{T}}{\partial y^2}\right). \tag{7.7}$$

Полученное уравнение не содержит значений u, v с текущего итерационного слоя, и значит может быть решено один раз в конце шага по времени, когда сходимость уже достигнута.

7.2.3 Аппроксимация на разнесённой сетке

Пространственную аппроксимацию уравнения (7.7) будем проводить на разнесённой сетке в центральных (“чёрных”) узлах сетки (рис. 7). При этом конвективную производную будем приближать с помощью симметричной разности. Полученная конечная разность для узла $i+\frac{1}{2}, j+\frac{1}{2}$ примет вид

$$\begin{aligned}
\frac{1}{\triangle t}\hat{T}_{i+\frac{1}{2},j+\frac{1}{2}} &+ \hat{u}_{i+\frac{1}{2},j+\frac{1}{2}}\frac{\hat{T}_{i+\frac{3}{2},j+\frac{1}{2}} - \hat{T}_{i-\frac{1}{2},j+\frac{1}{2}}}{2h_x} \\
&+ \hat{v}_{i+\frac{1}{2},j+\frac{1}{2}}\frac{\hat{T}_{i+\frac{1}{2},j+\frac{3}{2}} - \hat{T}_{i+\frac{1}{2},j-\frac{1}{2}}}{2h_y} \\
&+ \frac{1}{\mathrm{Pe}}\frac{-\hat{T}_{i+\frac{3}{2},j+\frac{1}{2}} + 2\hat{T}_{i+\frac{1}{2},j+\frac{1}{2}} - \hat{T}_{i-\frac{1}{2},j+\frac{1}{2}}}{h_x^2} \\
&+ \frac{1}{\mathrm{Pe}}\frac{-\hat{T}_{i+\frac{1}{2},j+\frac{3}{2}} + 2\hat{T}_{i+\frac{1}{2},j+\frac{1}{2}} - \hat{T}_{i+\frac{1}{2},j=\frac{1}{2}}}{h_x^2} \\
&= \frac{1}{\triangle t}\check{T}_{i+\frac{1}{2},j+\frac{1}{2}}.
\end{aligned} \tag{7.8}$$

Значения компонент скорости в центрах ячеек вычисляются с помощью ближайшей полусуммы

$$\hat{u}_{i+\frac{1}{2},j+\frac{1}{2}} \approx \frac{\hat{u}_{i,j+\frac{1}{2}} + \hat{u}_{i+1,j+\frac{1}{2}}}{2},$$
$$\hat{v}_{i+\frac{1}{2},j+\frac{1}{2}} \approx \frac{\hat{v}_{i+\frac{1}{2},j} + \hat{v}_{i+\frac{1}{2},j+1}}{2}.$$

7.2.4 Граничные условия

Учёт граничных условий производится за счёт вычисления значений в фиктивных узлах около границ.

Пусть требуется учесть условие на левой стенке ($i = 0$). Тогда соответствующий фиктивный узел будет иметь индекс $-\frac{1}{2}, j$. Ниже приведём его выражения для трёх типов граничных условий.

7.2.4.1 Условия первого рода

Пусть

$$T|_{left} = T^{\Gamma} \tag{7.9}$$

Тогда

$$\frac{T_{-\frac{1}{2},j+\frac{1}{2}} + T_{\frac{1}{2},j+\frac{1}{2}}}{2} = T^{\Gamma}.$$

Отсюда

$$T_{-\frac{1}{2},j+\frac{1}{2}} = -T_{\frac{1}{2},j+\frac{1}{2}} + 2T^{\Gamma}.$$

Таким образом, если в матрицу A^T левой части выражения (7.8) в фиктивную колонку $k\left[-\frac{1}{2}, j+\frac{1}{2}\right]$ требуется добавить какое-то значение a, это равносильно добавлению этого выражнеия с обратным знаком в диагональ и удвоенного выражения, умноженного на граничное значение, в правую часть b^T:

$$\begin{aligned} &k_0 = k\left[\tfrac{1}{2}, j+\tfrac{1}{2}\right], \quad k_1 = k\left[-\tfrac{1}{2}, j+\tfrac{1}{2}\right], \\ &A^T_{k_0,k_1} \mathrel{+}= a \quad \Rightarrow \quad A^T_{k_0,k_0} \mathrel{-}= a, \quad b^T_{k_0} \mathrel{-}= 2aT^{\Gamma}. \end{aligned} \tag{7.10}$$

7.2.4.2 Условия второго рода

Если на левой границе задано условие второго рода

$$\left.\frac{\partial T}{\partial n}\right|_{left} = -\left.\frac{\partial T}{\partial x}\right|_{left} = q \tag{7.11}$$

То вычисление фиктивного узла производится из конечной разности вида

$$\frac{T_{-\frac{1}{2},j+\frac{1}{2}} - T_{\frac{1}{2},j+\frac{1}{2}}}{h_x} = q.$$

Отсюда

$$T_{-\frac{1}{2},j+\frac{1}{2}} = T_{\frac{1}{2},j+\frac{1}{2}} + h_x q$$

Тогда

$$A^T_{k_0,k_1} \mathrel{+}= a \quad \Rightarrow \quad A^T_{k_0,k_0} \mathrel{+}= a, \quad b^T_{k_0} \mathrel{-}= h_x q \tag{7.12}$$

7.2.4.3 Условия третьего рода

Пусть на левой границе задано условие второго рода

$$\left.\frac{\partial T}{\partial n}\right|_{left} = -\left.\frac{\partial T}{\partial x}\right|_{left} = \alpha T + \beta \tag{7.13}$$

Расписывая производную и вычисляя значение температуры на стенке через полусумму, получим

$$\frac{T_{-\frac{1}{2},j+\frac{1}{2}} - T_{\frac{1}{2},j+\frac{1}{2}}}{h_x} = \alpha \frac{T_{-\frac{1}{2},j+\frac{1}{2}} + T_{\frac{1}{2},j+\frac{1}{2}}}{2} + \beta.$$

Отсюда выразим значение в фиктивном узле

$$T_{-\frac{1}{2},j+\frac{1}{2}} = \frac{2+\alpha h_x}{2-\alpha h_x} T_{\frac{1}{2},j+\frac{1}{2}} + \frac{2\beta h_x}{2-\alpha h_x}$$

Тогда

$$A^T_{k_0,k_1} \mathrel{+}= a \quad \Rightarrow \quad A^T_{k_0,k_0} \mathrel{+}= \frac{2+\alpha h_x}{2-\alpha h_x} a, \quad b^T_{k_0} \mathrel{-}= \frac{2\beta h_x}{2-\alpha h_x} a. \tag{7.14}$$

7.2.4.4 Универсальность условий третьего рода

Условие третьего рода (7.13) можно использовать для моделирования условий первого и второго рода. Так, условия второго рода (7.11) получаются, если положить $\alpha = 0$, $\beta = q$. А условия первого (7.9), – если $\alpha = \varepsilon^{-1}, \beta = -\varepsilon^{-1} T^\Gamma$, где ε – малое положительное число.

Если подставить эти выражения в формулу (7.14), то можно убедится, что они дадут выражения (7.12) и (7.10) (в пределе при $\varepsilon \to 0$) соответственно.

7.2.5 Коэффициент теплообмена

На границах, где заданы условия первого рода (7.9) можно вычислить тепловой поток, тем самым определив, сколько тепловой энергии требуется для поддержания этой постоянной температуры.

Безразмерный интегральный коэффициент теплообмена (интегральное число Нуссельта) определяется как

$$\mathrm{Nu} = \int\limits_\gamma \frac{\partial T}{\partial n}\, ds. \tag{7.15}$$

Для получения размерной мощности из этого безразмерного коэффициента (измеряемой в Ваттах), необходимо умножить его на $\lambda \triangle T L$.

Вычисление интегрального числа Нуссельта из определения (7.15) происходит по той же схеме, что и вычисление коэффициентов сил (6.20). При этом нормальная производная на границе $\partial T / \partial n$

вычисляется в виде

$$(x,y) \in \gamma_i : \quad \frac{\partial T}{\partial n} \approx \frac{T^\Gamma - T_k}{h/2}, \tag{7.16}$$

где γ_i – отрезок границы, k – индекс ячейки, прилегающей к этому отрезку, h – шаг сетки, поперёк границы (h_x для вертикальных границ и h_y – для горизонтальных).

7.3 Тестовые примеры

7.3.1 Задача о равномерном течении

Рассмотрим задачу о стационарном прямолинейном течении с граничными условиями

$$\begin{aligned}(x,y) \in \Gamma_{in} : &\quad u = 1, v = 0,\\ (x,y) \in \Gamma_{top,bot} : &\quad \frac{\partial u}{\partial n} = 0, v = 0,\\ (x,y) \in \Gamma_{out} : &\quad u\frac{\partial u}{\partial x} = 0, v = 0.\end{aligned}$$

Очевидно, что точным решением этой задачи будут $u = 1, v = 0, p = 0$. В случае использования алгоритма инициализации (п. 7.1) мы бы сразу получили этот ответ. Но здесь в качестве теста будем начинать итерации из состояния покоя $u = v = p = 0$.

Задача решается в области $[0, 2] \times [-\frac{1}{2}, \frac{1}{2}]$ с использованием алгоритма SIMPLEC с $E = 4$ и разбиением на единицу длины $n_{un} = 20$.

Программа реализована в тесте `linear2-simple` в файле `linear_2d_simple_test.cpp`.

Программа по расчёту этой задачи отличается от рассмотренной ранее задачи в каверне (п. 4.2) только наличием условий входного и выходного сечений.

Шаг алгоритма SIMPLE В функции `step()`, описывающей основной шаг алгоритма SIMPLE, добавлено вычисление граничных значений поправки скорости u' из (6.14) необходимых для соблюдения баланса массы (`compute_u_stroke_outflow`).

```
116 double Linear2DSimpleWorker::step(){
117   // Predictor step: U-star
118   std::vector<double> u_star = compute_u_star();
119   std::vector<double> v_star = compute_v_star();
120   std::vector<double> u_stroke_outflow = compute_u_stroke_outflow(u_star);
121   // Pressure correction
122   std::vector<double> p_stroke = compute_p_stroke(u_star, v_star, u_stroke_outflow);
123   // Velocity correction
124   std::vector<double> u_stroke = compute_u_stroke(p_stroke, u_stroke_outflow);
125   std::vector<double> v_stroke = compute_v_stroke(p_stroke);
126   // Set final values
127   std::vector<double> u_new = vector_sum(u_star, 1.0, u_stroke);
128   std::vector<double> v_new = vector_sum(v_star, 1.0, v_stroke);
129   std::vector<double> p_new = vector_sum(_p, _alpha_p, p_stroke);
```

```
130
131   return set_uvp(u_new, v_new, p_new);
132 }
```

В дальнейшем эти условия используются для расчёта поправки давления и для расчёта самой поправки скорости.

7.3.1.1 Учёт граничных условий

Вычисление поправки скорости на выходной границе Функция `compute_u_stroke_outflow`, реализующая вычисление формулы (6.14), имеет вид

```
378 std::vector<double> Linear2DSimpleWorker::compute_u_stroke_outflow(const
    ↪  std::vector<double>& u_star) const{
379   double qin = 0;
380   double qout = 0;
381   for (size_t j=0; j<_grid.ny(); ++j){
382     size_t ind_left = _grid.yface_grid_index_i_jp(0, j);
383     size_t ind_right = _grid.yface_grid_index_i_jp(_grid.nx(), j);
384     qin += u_star[ind_left]*_hy;
385     qout += u_star[ind_right]*_hy;
386   }
387   double Lout = _grid.Ly();
388   double diff_u = (qin - qout)/Lout;
389   std::vector<double> ret(_grid.ny(), diff_u);
390   return ret;
391 }
```

. Здесь в цикле по вертикальным граням вычисляются расходы по входному и выходному сечениям (`qin`, `qout`), далее находится поправка скорости (`diff_u`), постоянная для всех выходных отрезков, и возвращается вектор, содержащий эту поправку для всех выходных отрезков.

Учёт граниных условий для u^* Для учёта граничных условий входа и выхода при сборке уравнения для u^* в функции `assemble_u_slae` используется цикл

```
222   for (size_t j=0; j< _grid.ny(); ++j){
223     // left boundary: u = 1
224     {
225       size_t index_left = _grid.yface_grid_index_i_jp(0, j);
226       mat.set_value(index_left, index_left, 1.0);
227       _rhs_u[index_left] = 1.0;
228     }
229     // right boundary: u*du/dx = 0
```

```
230     {
231       size_t index_right = _grid.yface_grid_index_i_jp(_grid.nx(), j);
232       size_t index_right_minus = _grid.yface_grid_index_i_jp(_grid.nx()-1, j);
233       double u0 = std::max(0.0, _u[index_right]);
234       double coef = _tau*u0/_hx;
235       mat.set_value(index_right, index_right, 1.0 + coef);
236       mat.set_value(index_right, index_right_minus, -coef);
237       _rhs_u[index_right] = u0;
238     }
239   }
```

Здесь для левой границы согласно (6.5) жёстко устанавливается единичное значение. А для правой границы используются соотношения (6.16).

Учёт граничных условий для p' Найденные поправки скорости на выходной границе должны быть учтены при решении задачи для p' согласно (6.17) Сборка матрицы левой части при этом останется неизменной. Действительно, распишем производную на правой (выходной) границе

$$d^u \frac{\partial p'}{\partial x}_{n_x, j+\frac{1}{2}} \approx d^u \frac{p'_{k_1} - p'_{k_0}}{h_x}$$

входящую в выражение (4.24). Где $k_0 = k[n_x - \frac{1}{2}, j + \frac{1}{2}]$ – реальный, а $k_1 = k[n_x + \frac{1}{2}, j + \frac{1}{2}]$ – находящийся правее него фиктивный узлы сетки. Наличие этой производной требует добавления выражения d^u/h_x^2 в реальный (диагональный) столбец k_0 и выражения $-d^u/h_x^2$ в фиктивный столбец k_1 матрицы A^p в строке k_0. Следуя алгоритму (6.17) добавление значения в фиктивный столбец равносильно добавлению этого же значения в диагональный столбец. То есть два этих значения взаимоуничтожаться. Останется только модифицировать столбец правых членов. Альтернативно можно просто подставить значение производной (6.15) в дискретизованное выражение (4.24), и, поскольку оно не содержит в себе p', унести его в правую часть с обратным знаком и делением на h_x.

Учёт граничных условий в правой части осуществляется в функции `assemble_p_stroke_solver` за счёт модификации правой части для узлов, расположенных около выходной границы:

```
367     // outflow compensation
368     if (i == _grid.nx()-1){
369       rhs[ind0] -= (u_stroke_outflow[j]) / _tau / _hx;
370     }
```

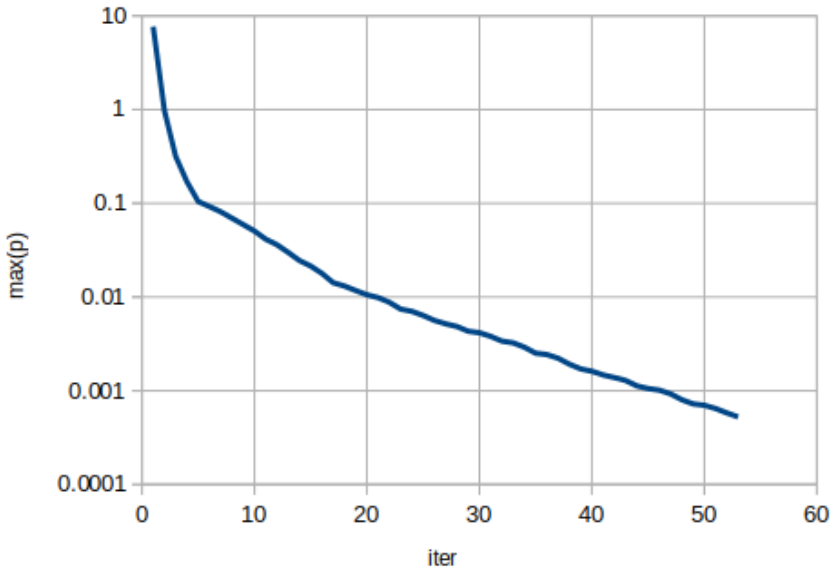
Учёт граничных условий для u' Явным образом предварительно найденные граничные значения для поправки скорости присваиваются в фукнции `"compute_u_stroke"`:

```
402   // outflow
403   for (size_t j=0; j<_grid.ny(); ++j){
404     size_t ind0 = _grid.yface_grid_index_i_jp(_grid.nx(), j);
405     u_stroke[ind0] = u_stroke_outflow[j];
406   }
```

7.3.1.2 Анализ результатов

До невязки $\varepsilon = 10^{-2}$ задача сходится за 53 итерации. При этом для скорости точный ответ получается уже на первой итерации, а всё остальное время происходит подстройка давления.

Максимальное значение давление в зависимости от итерации приведено на графике ниже



7.3.2 Течение Пуазейля

TODO

7.3.3 Стационарное обтекание квадратного препятствия

В тесте `obstacle2-simple` из файла `obstacle_2d_simple_test.cpp` рассматривается задача о стационарном обтекании квадратного препятствия (см. постановку в п. 6.2). По окончании расчёта в консоль печатаются коэффициенты сопротивления и подъёмной силы.

Используется естественная нумерация узлов с неактивными ячейками. Класс `RegularGrid2D` предлагает следующие методы, связанные с неактивными ячейками:

- `void RegularGrid2D::deactivate_cells(Point bot_left, Point top_right)` – установить область неактивных ячеек;

- `bool RegularGrid2D::is_active_cell(size_t icell)` – проверить, является ли ячейка активной.

Кроме того, в задаче появились внутренние границы. То есть для постаноки граничных условий уже не достаточно использовать крайние значение индексов i, j, а нужен механизм для получения граничных отрезков сетки. Для этого введены следующие функции

- `RegularGrid2d::boundary_yfaces()` – получить список всех вертикальных граничных фасок (возвращает парные индексы в соответствии `yface_centered_grid`.
- `RegularGrid2d::boundary_xfaces()` – получить список всех горизонтальных граничных фасок (возвращает парные индексы в соответствии `xface_centered_grid`.
- `RegularGrid2d::yface_type(size_t yface_index)` – узнать тип вертикальной грани по её глобальному индексу. Возвращает перечисление

```
enum struct FaceType{
  Internal,     // внутренняя
  Boundary,     // граничная
  Deactivated   // неактивная (находится внутри неактивной области)
};
```

- `RegularGrid2d::xface_type(size_t xface_index)` – узнать тип вертикальной грани по её глобальному индексу.

7.3.3.1 Функция верхнего уровня

Здесь сначала происходит установка параметров расчёта: числа Рейнольдса, параметра E, количества итераций, порога сходимости и разбиения единичного интервала.

```
729  double Re = 20;
730  double E = 4.0;
731  size_t max_it = 10000;
732  double eps = 1e-1;
733  size_t n_unit = 10;  // partition per unit length
```

Далее строится сетка

```
736  RegularGrid2D grid(0, 12, -2, 2, 12*n_unit, 4*n_unit);
```

В этом примере сетка строится в четырёхугольнике $[0, 12] \times [-2, 2]$. Потом для описания квадратного препятствия происходит деактивация ячеек, находящихся в единичном квадрате с нижней левой координатой $(2, -0.5)$ и верхней правой координатой $(3, 0.5)$.

```
737  grid.deactivate_cells({2, -0.5}, {3, 0.5});
```

Потом создаётся решатель, инициализируются функции сохранения и вызывается алгоритм потенциальной инициализации расчётных полей.

```
738  Obstacle2DSimpleWorker worker(Re, grid, E);
739  worker.initialize_saver(false, "obstacle2");
740
741  // initial condition
742  worker.initialize();
```

Затем идёт стандартный цикл по SIMPLE-итерациям

```
745  size_t it = 0;
746  for (it=1; it < max_it; ++it){
747    double nrm = worker.step();
748
749    // print norm
750    std::cout << it << " " << nrm << std::endl;
751
752    // break if residual is low enough
753    if (nrm < eps){
754      break;
755    }
756  }
```

По окочании цикла вызывается функция сохранения решения в vtk

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

В конце происходит расчёт коэффициентов сил и их печать в консоль

```
760  Obstacle2DSimpleWorker::Coefficients coefs = worker.coefficients();
761  std::cout << "=== Drag" << std::endl;
762  std::cout << "Cpx = " << coefs.cpx << std::endl;
763  std::cout << "Cfx = " << coefs.cfx << std::endl;
764  std::cout << "Cx  = " << coefs.cx  << std::endl;
765  std::cout << "=== Lift" << std::endl;
766  std::cout << "Cpy = " << coefs.cpy << std::endl;
```

```
767   std::cout << "Cfy = " << coefs.cfy << std::endl;
768   std::cout << "Cy  = " << coefs.cy  << std::endl;
```

Результирующее поле течения сохраняется в файл `obstacle2.vtk.series`.

7.3.3.2 Учёт неактивных ячеек

Неактивные ячейки учитываются во всех алгоритмах сборки систем линейных уравнений. Рассмотрим на примере сборки матрицы для пробной скорости, реализованной в функции `assemble_u_slae`.

```
326 void Obstacle2DSimpleWorker::assemble_u_slae(){
```

Рассмотрим цикл сборки внутренних узлов “красной” сетки для u (или, что тоже самое, цикл по всем вертикальным граням основной сетки)

```
371   for (size_t j=0; j < _grid.ny(); ++j)
372   for (size_t i=1; i < _grid.nx(); ++i){
```

Сначала вычисляется индекс строки (сквозной индекс текущей грани):

```
373     size_t row_index = _grid.yface_grid_index_i_jp(i, j);    //[i, j+1/2]
```

Эта грань может быть либо внутренней, либо граничной (принадлежать внутренней вертикальной границе), либо неактивной. Выполняется проверка, является ли эта грань внутренней

```
374     if (_grid.yface_type(row_index) == RegularGrid2D::FaceType::Internal){
```

Если да, то выполняется обычная процедура сборки

```
375       double u0_plus   = u_ip_jp(i, j);   // _u[i+1/2, j+1/2]
376       double u0_minus  = u_ip_jp(i-1, j); // _u[i-1/2, j+1/2]
377       double v0_plus   = v_i_j(i, j+1);   // _v[i,j+1]
378       double v0_minus  = v_i_j(i, j);     // _v[i,j]
379
380       // u_(i,j+1/2)
381       add_to_mat(row_index, {i, j}, 1.0);
382       //      + tau * d(u0*u)/ dx
383       add_to_mat(row_index, {i+1,j}, _tau/2.0/_hx*u0_plus);
384       add_to_mat(row_index, {i-1,j}, -_tau/2.0/_hx*u0_minus);
385       //      + tau * d(v0*u)/dy
386       add_to_mat(row_index, {i, j+1}, _tau/2.0/_hy*v0_plus);
387       add_to_mat(row_index, {i, j-1}, -_tau/2.0/_hy*v0_minus);
388       //      - tau / Re * d^2u/dx^2
```

```
389       add_to_mat(row_index, {i, j}, 2.0*_tau/_Re/_hx/_hx);
390       add_to_mat(row_index, {i+1, j}, -_tau/_Re/_hx/_hx);
391       add_to_mat(row_index, {i-1, j}, -_tau/_Re/_hy/_hy);
392       //      - tau / Re * d^2u/dy^2
393       add_to_mat(row_index, {i, j}, 2.0*_tau/_Re/_hy/_hy);
394       add_to_mat(row_index, {i, j+1}, -_tau/_Re/_hy/_hy);
395       add_to_mat(row_index, {i, j-1}, -_tau/_Re/_hy/_hy);
396       // = u0_(i,j+1/2)
397       _rhs_u[row_index] += _u[row_index];
398       //       - tau * dp/dx
399       _rhs_u[row_index] -= _tau/_hx*(p_ip_jp(i, j) - p_ip_jp(i-1, j));
```

Если нет (то есть грань либо неактивная, либо принадлежит внутренней границе), то в диагональ ставится единица, в правую часть 0.

```
400     } else {
401       mat.set_value(row_index, row_index, 1.0);
402       _rhs_u[row_index] = 0;
403     }
```

Это отражает тот факт, что на внутренних границах $u = 0$ из-за условий прилипания, а для неактивных мы пишем тривиальное уравнение, просто чтобы матрица не была вырождена.

Учёт условий прилипания на внутренних горизонтальных границах осуществляется через фиктивный узел в лямбда-функции `"add_to_mat"`, которая перехватывает все ситуации, когда алгоритм требует добавить что-либо в фиктивную колонку матрицы.

```
331   auto add_to_mat = [&](size_t row_index, std::array<size_t, 2> ij_col, double value){
```

Такие ситуации могут произойти либо при сборке около вертикальной грани, находящейся рядом с верхней границей:

```
332     if (ij_col[1] == _grid.ny()){
333       // ghost index => top boundary condition: du/dn = 0
334       size_t ind1 = _grid.yface_grid_index_i_jp(ij_col[0], ij_col[1]-1);
335       mat.add_value(row_index, ind1, value);
```

либо около вертикальной грани, находящейся рядом с нижней границей границей:

```
336     } else if (ij_col[1] == (size_t)-1){
337       // ghost index => bottom boundary condition: du/dn = 0
338       size_t ind1 = _grid.yface_grid_index_i_jp(ij_col[0], ij_col[1]+1);
339       mat.add_value(row_index, ind1, value);
```

либо около вертикальнй грани, находящейся непосредственно над или под препятствием. В этом случае индекс фиктивной колонки, в которую трубуется поставить будет соответствовать неакотвной вертикальной грани. Мы вычисляем этот индекс

```
340     } else {
341       size_t ind1 = _grid.yface_grid_index_i_jp(ij_col[0], ij_col[1]);
```

Если он неактивный, то следуем по процедуре добавления фиктивного узла около границы с нулевым значением.

```
342       if (_grid.yface_type(ind1) == RegularGrid2D::FaceType::Deactivated){
343         // ghost index => obstacle boundary u = 0
344         mat.add_value(row_index, row_index, -value);
345       } else {
```

Иначе – это нормальная колонка и мы добавляем туда значение по стандартной процедуре

```
346         mat.add_value(row_index, ind1, value);
```

7.3.3.3 Расчёт коэффициентов сопротивления

Расчёт коэффициентов сил по формулам (6.18), (6.19) осуществляется в процедуре `coefficients()`. Она возвращает структуру, куда входят все шесть искомых значений

```
33   struct Coefficients{
34     double cpx;
35     double cpy;
36     double cfx;
37     double cfy;
38     double cx;
39     double cy;
40   };
```

Процедура, объявленная как

```
661 Obstacle2DSimpleWorker::Coefficients Obstacle2DSimpleWorker::coefficients() const{
```

производит вычисления четырёх интегралов по простой квадратуре (6.20). Результаты аггрегируются в переменные

```
662   double sum_cpx = 0;
663   double sum_cpy = 0;
```

```
664   double sum_cfx = 0;
665   double sum_cfy = 0;
```

Операции проводятся в циклах по внутренним граничным отрезкам. Сначала рассматриваются вертикальные границы:

```
668   for (const RegularGrid2D::split_index_t& yface: _grid.boundary_yfaces()){
```

Здесь в переменную `yface` попадают все парные индексы вертикальных граней, лежащих на границах. Сначала нужно отфильтровать границы, лежащие во входном и выходном сечениях

```
669     if (yface[0] == 0){
670       // input => ignore
671     } else if (yface[0] == _grid.nx()){
```

На вертикальных границах актуально вычисление коэффициентов C_x^p, C_y^f (из пункта 6.2.3.3). Дли их определения на каждой сеточной грани мы должны определить $p\,n_x$, $\partial v/\partial n$ по формулам (6.21), (6.22).

```
674       double pnx, dvdn;
```

Для использования этих формул нужно определить, является ли это левой или правой границей обтекаемого тела. Мы вычисляем индексы ячеек, лежащих слева и справа. Если левая ячейка активна, значит это левая граница, если правая активна, значит это правая граница.

```
675       size_t left_cell = _grid.cell_centered_grid_index_ip_jp(yface[0]-1, yface[1]);
676       size_t right_cell = _grid.cell_centered_grid_index_ip_jp(yface[0], yface[1]);
```

Далее левой границы

```
677       if (_grid.is_active_cell(left_cell)){
678         pnx = _p[left_cell];
679         dvdn = -v_ip_jp(yface[0]-1, yface[1]) / (_hx/2.0);
```

для правой

```
680       } else if (_grid.is_active_cell(right_cell)){
681         pnx = -_p[right_cell];
682         dvdn = -v_ip_jp(yface[0], yface[1]) / (_hx/2.0);
```

иначе (если это и не правая и не левая граница) бросается исключение, потому что так быть не должно: у любой внутренней границы должна быть хоть одна соседняя активная ячейка

```
683       } else {
684         _THROW_UNREACHABLE_;
685       }
```

После вычисления $p\,n_x$, $\partial v/\partial n$ они добавляются в искомые интегралы согласно (6.20):

```
686       sum_cpx += pnx * _hy;
687       sum_cfy += dvdn * _hy;
```

Далее аналогичная процедура проводится для горизонтальных граней, в результате которой вычисляются интегралы `sum_cpy`, `sum_cfx`.

```
692   for (const RegularGrid2D::split_index_t& xface: _grid.boundary_xfaces()){
693     if (xface[1] == 0){
694       // bottom => ignore
695     } else if (xface[1] == _grid.ny()){
696       // top => ignore
697     } else {
698       double pny, dudn;
699       size_t bot_cell = _grid.cell_centered_grid_index_ip_jp(xface[0], xface[1]-1);
700       size_t top_cell = _grid.cell_centered_grid_index_ip_jp(xface[0], xface[1]);
701       if (_grid.is_active_cell(bot_cell)){
702         pny = _p[bot_cell];
703         dudn = -u_ip_jp(xface[0], xface[1]-1)/(_hy/2.0);
704       } else if (_grid.is_active_cell(top_cell)){
705         pny = -_p[top_cell];
706         dudn = -u_ip_jp(xface[0], xface[1])/(_hy/2.0);
707       } else {
708         _THROW_UNREACHABLE_;
709       }
710       sum_cpy += pny * _hx;
711       sum_cfx += dudn * _hx;
712     }
713   }
```

В конце функции искомые коэффициенты вычисляются через уже найденные интегралы согласно (6.18), (6.19):

```
715  Coefficients coefs;
716  coefs.cpx = 2.0*sum_cpx;
717  coefs.cpy = 2.0*sum_cpy;
718  coefs.cfx = -2.0/_Re*sum_cfx;
719  coefs.cfy = -2.0/_Re*sum_cfy;
720  coefs.cx = coefs.cpx + coefs.cfx;
721  coefs.cy = coefs.cpy + coefs.cfy;
722  return coefs;
```

7.3.3.4 Результаты расчёта

Картина течения, полученная для сетки
`n_part = 10` при $\mathrm{Re} = 20$, представлена на рис. 12 Полученные коэффициенты сопротивления:

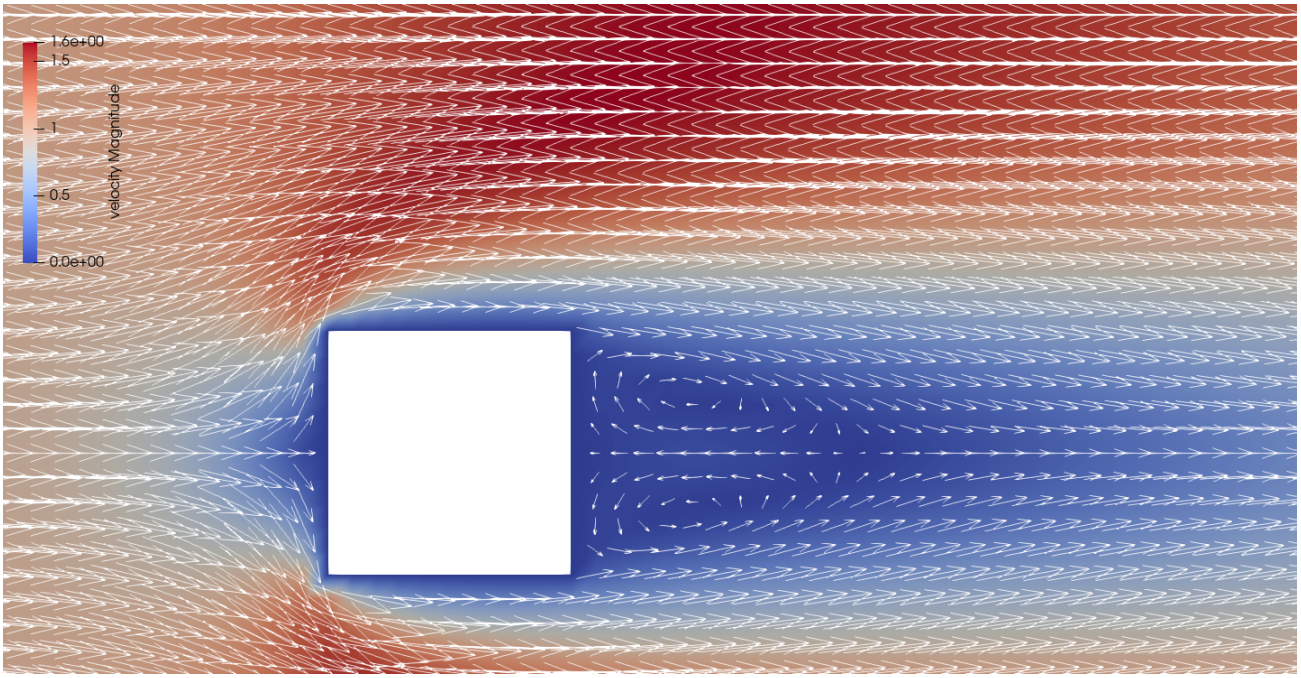


Рис. 12: Обтекание квадратного препятствия в стационарном режиме

```
=== Drag
Cpx = 2.97224
Cfx = 1.08639
Cx  = 4.05863
=== Lift
Cpy = -6.815e-10
Cfy = -1.64419e-10
Cy  = -8.45919e-10
```

Для $\varepsilon = 10^{-1}$ решение сошлось за 29 итераций.

7.3.4 Нестационарное обтекание квадратного препятствия с теплообменом

Эта задача реализована в файле `obstacle_nonstat_2d_simple_test.cpp` в тесте `[obstacle2-nonstat-simple]`.

Программа решает задачу в той же области, которая рассматривалась в предыдущем пункте, но в нестационарной постановке (5.2) и с добавлением температуры (7.6). Граничные условия для температуры имеют вид

$$\begin{aligned} (x,y) \in \Gamma_{in}: &\quad T = 0, \\ (x,y) \in \gamma: &\quad T = 1, \\ (x,y) \in \Gamma_{out,top,bot}: &\quad \frac{\partial T}{\partial n} = 0. \end{aligned}$$

Поля течения для разных моментов времени пишутся в файл `obstacle-nonstat.vtk.series`. Кроме того, в файл `c.txt` пишутся вычисленные на разные моменты времени коэффициенты сопротивления и интегральное число Нуссельта.

7.3.4.1 Функция верхнего уровня

В начале обозначим параметры задачи: числа Рейнольдса и Пекле, разбиение единичного отрезка, шаг по времени $\triangle t$ и конечное время, параметр внутреннего итерационного процесса E, максимальное количество итераций во внутреннем итерационном процессе и порог по невязке:

```
866  double Re = 100;
867  double Pe = 100;
868  size_t n_unit = 10;  // partition per unit length
869  double time_step = 0.25;
870  double end_time = 5;
871  double E = 4.0;
872  size_t max_it = 10000;
873  double eps = 1e-0;
```

Далее проводится создание сетки (так же, как и в предыдущем примере) и начальная инициализация решателя

```
876  RegularGrid2D grid(0, 12, -2, 2, 12*n_unit, 4*n_unit);
877  grid.deactivate_cells({2, -0.5}, {3, 0.5});
878  ObstacleNonstat2DSimpleWorker worker(Re, Pe, grid, E, time_step);
879  worker.initialize_saver(false, "obstacle2-nonstat");
880
881  // initial condition
882  worker.initialize();
883  worker.save_current_fields(0);
```

После всех инициализаций начинается цикл по времени

```
886  for (double time=time_step; time<end_time+1e-6; time+=time_step){
```

Отметим, что поскольку значению $t = 0$ соответствует начальное состояние решения, то цикл начинается сразу с первого шага $t = \triangle t$.

Внутри цикла по времени производится цикл внутренних итераций SIMPLE

```
887    size_t it = 0;
888    for (it=1; it < max_it; ++it){
889      double nrm = worker.step();
890
891      // break inner iterations if residual is low enough
892      if (nrm < eps){
893        break;
894      } else if (it == max_it -1) {
895        std::cout << "WARNING: internal SIMPLE interations not converged with nrm = "
896                  << nrm << std::endl;
897      }
898    }
```

Далее, если текущее время кратно 1.0, производится сохранение решения в файл vtk и запись коэффициентов сил в файл:

```
900    if (std::abs(time - round(time)) < 1e-6){
901      worker.save_current_fields(time);
902    }
```

Печатается информация о сходимости текущей итерации

```
903    std::cout << convergence_report(time, it);
```

и производится переход на следующий шаг по времени:

```
906    worker.to_next_time_step();
```

7.3.4.2 Учёт нестационарности

Согласно пункту 5.2 наличие производной по времени учитывается:

- При вычислении коэффициентов d^u, d^v (5.5):

```
116   _du = 1.0 / (1 + _tau/_time_step + 2.0*_tau/_Re * (1.0/_hx/_hx + 1.0/_hy/_hy));
117   _dv = 1.0 / (1 + _tau/_time_step + 2.0*_tau/_Re * (1.0/_hx/_hx + 1.0/_hy/_hy));
```

- При сборке систем уравнений для u^*, u^* (5.4) как прибавка к диагонали

```
423       add_to_mat(row_index, {i, j}, 1.0 + _tau/_time_step);
```

и правой части

```
441       _rhs_u[row_index] += (_tau/_time_step)*_u_old[row_index];
```

- А так же в граничных условиях на выходе. В этом случае условия (6.10) для u по аналогии с (6.11) аппроксимируются к виду

$$(x,y) \in \Gamma_{out}: \ \frac{\hat{u}-\check{u}}{\triangle t} + \frac{\hat{u}-u}{\tau} + u\frac{\partial \hat{u}}{\partial x} = 0.$$

Для упрощения по прежнему будем использовать "стационарное" условие для поперечной скорости $v = 0$. Тогда для u^* можно записать

$$(x,y) \in \Gamma_{out}: \ \left(1 + \frac{\tau}{\triangle t}\right) u^*_{n_x,j+\frac{1}{2}} + \tau U_{n_x,j+\frac{1}{2}} \frac{u^*_{n_x,j+\frac{1}{2}} - u^*_{n_x-1,j+\frac{1}{2}}}{h_x} = u_{n_x,j+\frac{1}{2}} + \frac{\tau}{\triangle t}\check{u}_{n_x,j+\frac{1}{2}}$$

Это выражение и добавляется в соответствующие строки матрицы и правой части

```
400     // right boundary: du/dt + u*du/dx = 0
401     {
402       size_t index_right = _grid.yface_grid_index_i_jp(_grid.nx(), j);
403       size_t index_right_prev = _grid.yface_grid_index_i_jp(_grid.nx()-1, j);
404       double u0 = std::max(0.0, _u[index_right]);
405       double coef = _tau*u0/_hx;
406       mat.set_value(index_right, index_right, 1.0 + _tau/_time_step + coef);
407       mat.set_value(index_right, index_right_prev, -coef);
408       _rhs_u[index_right] = u0 + _tau/_time_step * _u_old[index_right];
409     }
```

- При переходе на слудующий шаг по времени в функции происходит вычисление текущего значения температуры, и присваивание значений $\check{u}$, $\check{v}$.

```
247 double ObstacleNonstat2DSimpleWorker::to_next_time_step(){
248   _t = compute_temperature();
249   _u_old = _u;
```

```
250   _v_old = _v;
251   return set_uvp(_u, _v, _p);
252 }
```

Вызов `set_uvp` здесь осуществляется для пересборки актуальных матриц.

7.3.4.3 Расчёт температурного поля

Осуществляется в функции

```
624 std::vector<double> ObstacleNonstat2DSimpleWorker::compute_temperature() const{
```

Для сборки системы используется цикл по ячейкам сетки

```
651   for (size_t j=0; j < _grid.ny(); ++j)
652   for (size_t i=0; i < _grid.nx(); ++i){
653     size_t row_index = _grid.cell_centered_grid_index_ip_jp(i, j);
654     if (_grid.is_active_cell(row_index)){
655       double u_left = _u[_grid.yface_grid_index_i_jp(i, j)];
656       double u_right = _u[_grid.yface_grid_index_i_jp(i+1, j)];
657       double v_bot = _v[_grid.xface_grid_index_ip_j(i, j)];
658       double v_top = _v[_grid.xface_grid_index_ip_j(i, j+1)];
659 
660       // 1.0/time_step T(i+1/2,j+1/2)
661       add_to_mat(row_index, {i, j}, 1.0 / _time_step);
662       //     + d(u0*T)/ dx
663       add_to_mat(row_index, {i+1,j},  u_right/2.0/_hx);
664       add_to_mat(row_index, {i-1,j}, -u_left/2.0/_hx);
665       //     + d(v0*T)/dy
666       add_to_mat(row_index, {i, j+1},  v_top/2.0/_hy);
667       add_to_mat(row_index, {i, j-1}, -v_bot/2.0/_hy);
668       //     - 1 / Re * d^2u/dx^2
669       add_to_mat(row_index, {i, j},    2.0/_Pe/_hx/_hx);
670       add_to_mat(row_index, {i+1, j}, -1.0/_Pe/_hx/_hx);
671       add_to_mat(row_index, {i-1, j}, -1.0/_Pe/_hx/_hx);
672       //     - 1 / Re * d^2u/dy^2
673       add_to_mat(row_index, {i, j},     2.0/_Pe/_hy/_hy);
674       add_to_mat(row_index, {i, j+1},  -1.0/_Pe/_hy/_hy);
675       add_to_mat(row_index, {i, j-1},  -1.0/_Pe/_hy/_hy);
676       // = 1.0 / time_step*Told
677       rhs[row_index] += 1.0 / _time_step*_t[row_index];
678     } else {
679       mat.set_value(row_index, row_index, 1.0);
```

```
      rhs[row_index] = 0;
    }
  }
```

для активных ячеек используются формулы (7.8), а для неактивных – тривиальное уравнение $T = 0$.

Учёт граничных условий осуществляется за счёт фиктивных узлов в функции `add_to_mat`

```
  auto add_to_mat = [&](size_t row_index, std::array<size_t, 2> ij_col, double value){
```

Для левой границы условия первого рода (7.10) с $T^{\Gamma} = 0$

```
      // left boundary: T=0
      mat.add_value(row_index, row_index, -value);
```

для нижней, верхней и выходной – условия второго рода с $q = 0$ (7.12):

```
      // right boundary: dT/dn = 0
      mat.add_value(row_index, row_index, value);
```

Для границы обтекаемого тела – условия первого рода (7.10) с $T^{\Gamma} = 1$:

```
        // internal boundary: T = 1
        mat.add_value(row_index, row_index, -value);
        rhs[row_index] -= 2*value;
```

После сборки правой и левой частей происходит решение СЛАУ и возвращается ответ

```
  std::vector<double> temperature;
  AmgcMatrixSolver::solve_slae(mat.to_csr(), rhs, temperature);
  return temperature;
```

7.3.4.4 Вычисление коэффициента теплообмена

Производится в той же функции, где и другие коэффициенты (см. п.7.3.3.3)

```
ObstacleNonstat2DSimpleWorker::Coefficients
↪  ObstacleNonstat2DSimpleWorker::coefficients() const{
```

В цикле по вертикальным границам

```
787  for (const RegularGrid2D::split_index_t& yface: _grid.boundary_yfaces()){
```

на левой границе обтекаемого тела согласно формуле (7.16) имеем

```
799          dtdn = (1.0 - _t[left_cell]) / (_hx/2.0);
```

Здесь `left_cell` – индекс ячейки, лежащей слева от рассматриваемого участка границы, а $T^{\Gamma} = 1$ – значение температуры из граничного условия. Аналогичные выражения исользованы для правых

```
803          dtdn = (1.0 - _t[right_cell]) / (_hx/2.0);
```

нижних

```
826          dtdn = (1.0 - _t[bot_cell]) / (_hy/2.0);
```

и верхних фасок

```
830          dtdn = (1.0 - _t[top_cell]) / (_hy/2.0);
```

После определения нормальной производной по температуре она добавляется в интеграл согласно (6.20). Например, для горизонтальных границ

```
836       sum_nu += dtdn * _hx;
```

7.3.4.5 Результаты расчёта

Настоящую задачу будем решать с параметрами $\mathrm{Re} = 100$, $\mathrm{Pe} = 100$, $\varepsilon = 10^{-1}$, $\triangle t = 0.1$, $t_{end} = 200$, $n = 10$.

На рис. 13 представлено поле температуры на разные моменты времени. Видно, что сначала нагреваемая жидкость продвигается вниз по потоку, потом, на момент времени $t \approx 50$ поток устанавливается, но, в районе $t \approx 100$ решение теряет устойчивость и за препятствием образуется дорожка Кармана.

На рис. 14 представлено зависимость количество проведённых итераций от времени. На начальном этапе, пока течение развивается от состояния потенциального обтекания (начальное условие), количество итераций велико, затем, с достижением решения локального установления, решение начинает сходится за одну итерацию. Но после $t > 100$, когда начинает развиваться неустойчивость, количество итераций вновь возрастает. Количество итераций на слое характеризует степень изменения решения при продвижении к следующему шагу по времени.

Коэффициенты сопротивления, подъёмной силы и теплоотдачи нарисованы на рис. 15. Переход к нестационарному режиму течения характеризуется повышением теплоотдачи и сопротивлению и появлению заметных осцилляций в подъёмной силе.

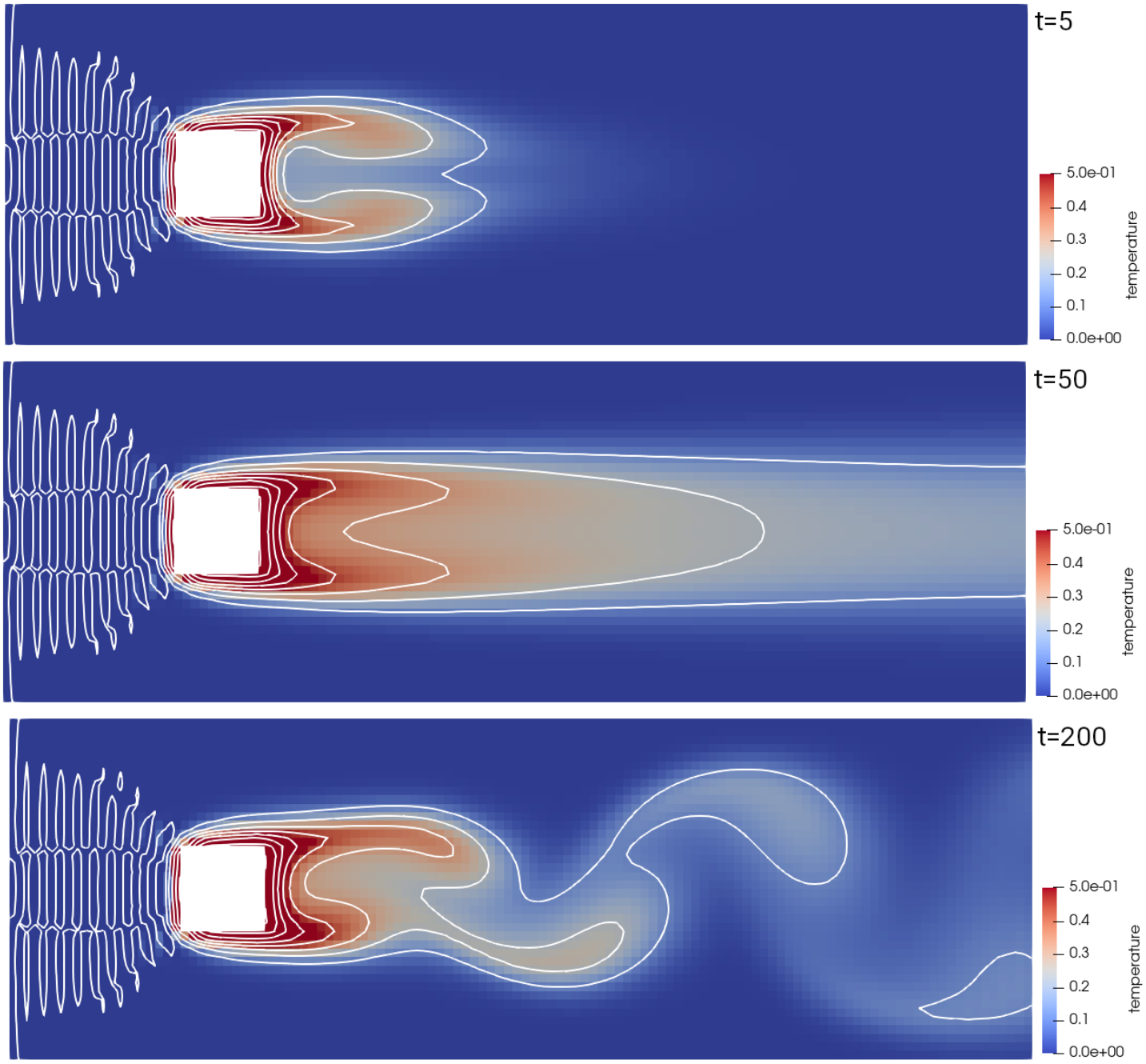


Рис. 13: Температурное поле при решении нестационарной задачи обтекания. Моменты времени $t = 5$, $t = 50$, $t = 200$

Следует обратить внимание на небольшую рябь в поле температур, заметную слева от препятствия на рис. 13. Её хорошо видно на трёхмерном отображении (см. рис. 16). Если обратить внимание на легенду, то можно заметить, что температура в этой области даже становится отрицательной, что физически невозможно в рамках поставленных граничных условий.

Причина этой ряби кроется в симметричной разности, которую мы использовали для аппроксимации конвективного слагаемого уравнения температуры (см. (7.10)). Известно, что симметричные разности склонны давать осциллирующее решение (даже если оно и устойчиво). Если бы мы использовали схему против потока, то этой нефизичности в решении бы не было, но при этом решение бы имело первый порядок точности по пространству.

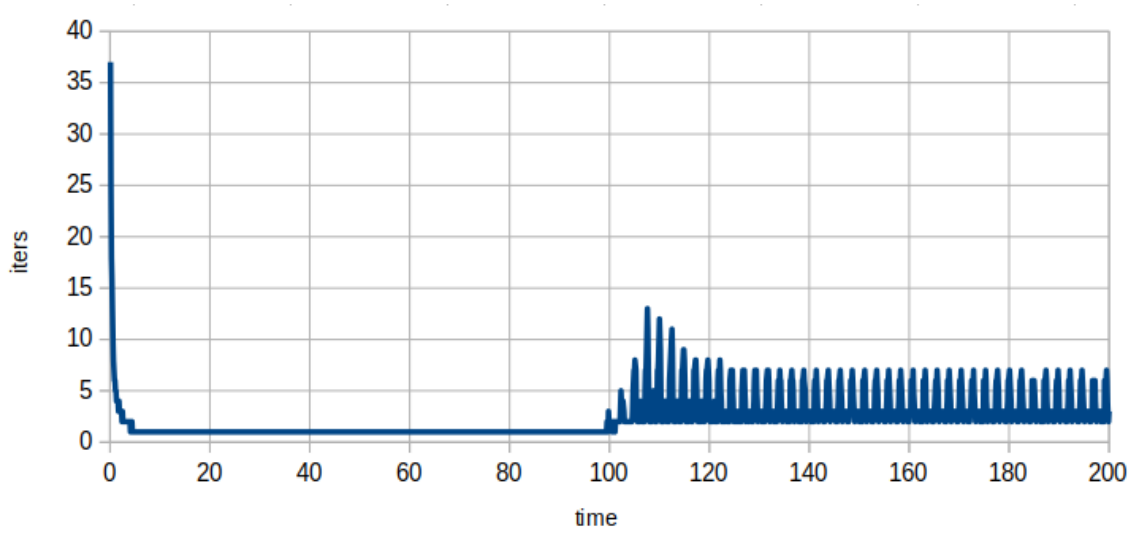


Рис. 14: Зависимость количества внутренних итераций SIMPLE от момента времени

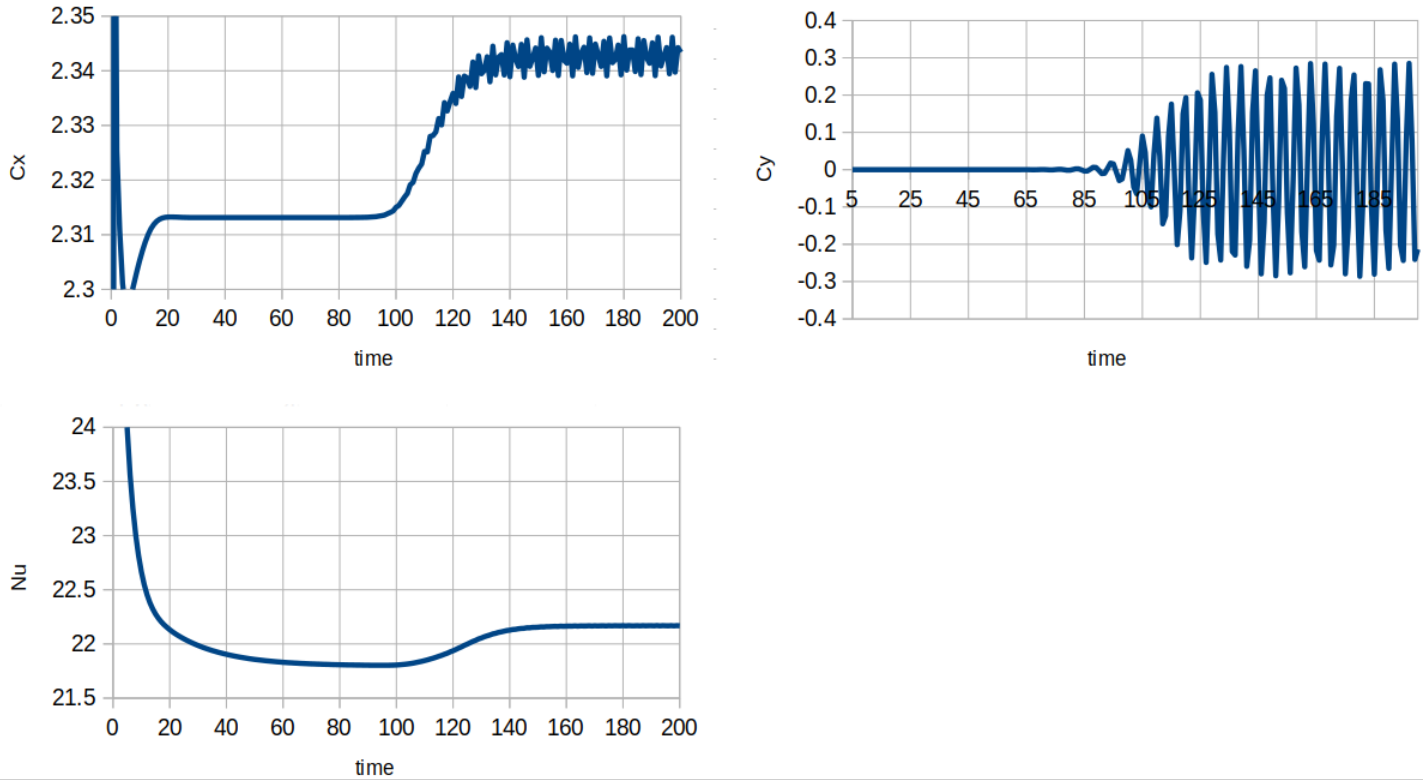


Рис. 15: Эволюция коэффициентов сопротивления C_x, подъёмной силы C_y и теплоотдачи Nu

7.4 Задание для самостоятельной работы

Решить задачу с двумя обтекаемыми телами: одно расположено в прямоугольнике

$$\gamma_1 : \begin{cases} 2.0 \le x \le 2.5, \\ -0.7 \le y \le 0.3, \end{cases}$$

второе –

$$\gamma_2 : \begin{cases} 4 \le x \le 4.5, \\ -0.3 \le y \le 0.7. \end{cases}$$

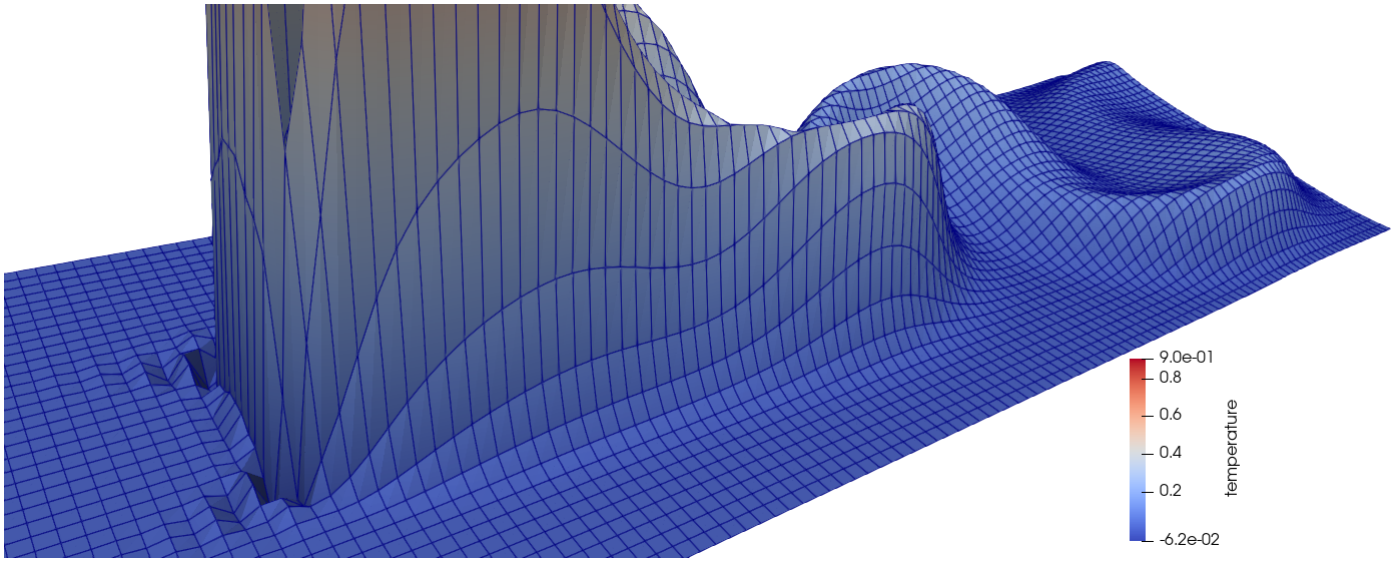


Рис. 16: Температура на момент $t = 150$ в трёхмерном отображении. Нефизичные осцилляции в области перед препятствием

Использовать условия для температуры:

$$(x, y) \in \gamma_1 : \quad T = 0.5,$$
$$(x, y) \in \gamma_2 : \quad T = 1.0.$$

Область расчёта $[0, 15] \times [-2, 2]$:

```
RegularGrid2D grid(0, 15, -2, 2, 15*n_unit, 4*n_unit);
```

Остальные граничные условия использовать те же, что и в рассмотренной в п. 7.3.4 задаче. Параметры задачи: $\mathrm{Re} = 100$, $\mathrm{Pe} = 100$, $\varepsilon = 10^{-1}$, $\triangle t = 0.1$, $t_{end} = 200$, $n = 10$.

Подсчитать коэффициенты сопротивления и теплоотдачи для каждого из двух тел в отдельности. Нарисовать графики из изменения со временем.

Делать на основе программы из файла `obstacle_nonstat_2d_simple_test.cpp`.

Задание сетки с неактивными ячейками Обтекаемые препятствия следует задавать при определении сетки. В рассмотренном примере из предыдущего пункта это делалось в строке

```
877  grid.deactivate_cells({2, -0.5}, {3, 0.5});
```

В настоящей задачи нужно эту функцию вызвать два раза, указав там по очереди обе нужные области.

Задание граничных условий на температуру Поскольку в задаче граничные условия на обтекаемых телах отличаются по своему значению, то следует модифицировать алгоритм их задания. Граничные условия на температуру задаются в функции `compute_temperature` в строке

```
645         // internal boundary: T = 1
646         mat.add_value(row_index, row_index, -value);
647         rhs[row_index] -= 2*value;
```

Согласно форме (7.10) изменения в левой части не зависит от величины граничной температуры, а в провой – пропорционально ей. Таким образом, для первого тела (где $T^{\Gamma} = 0.5$) добавка в правую часть будет иметь вид

```
rhs[row_index] -= 2*value*0.5,
```

а для второго – останется такой же, как и раньше.

При этом нужно уметь отличать грани, принадлежащие первому телу от граней, принадлежащих второму. Для этого в классе `ObstacleNonstat2DSimpleWorker` можно объявить функцию, которая определяет ближайшую к ячейке границу. Саму функцию можно реализовать просто используя координаты центра ячейки `_grid.cell_center(icell)`. Например:

```
// => 1 если ячейка icell близка к первому обтекаемому телу и 2 - если ко второму
int ObstacleNonstat2DSimpleWorker::gamma_closest_to_cell(size_t icell){
  double x = _grid.cell_center(icell).x();
  if (x < 3.25) {  // центр между первым и вторым телами
    return 1;
  } else {
    return 2;
  }
}
```

Тогда можно написать

```
double t_gamma = (gamma_closest_to_cell(row_index) == 1) ? 0.5 : 1.0;
rhs[row_index] -= 2*t_gamme*value;
```

Здесь запись `double a = (cond) ? 0.5 : 1.0;` есть сокращение от

```
double a;
if (cond){
  a = 0.5;
} else {
  a = 1.0
}
```

Вычисление коэффициентов Во-первых следует модифицировать структуру, хранящую коэффициенты, сделав там отдельные записи для каждого тела:

```
struct Coefficients{
  double cpx1, cpx2;
  double cpy1, cpy2;
  double cfx1, cfx2;
  double cfy1, cfy2;
  double cx1, cx2;
  double cy1, cy2;
  double nu1, nu2;
};
```

Во-вторых, в функции сохранения этих коэффициентов в файл в функции `ObstacleNonstat2DSimpleWorker::save_current_fields`

```
cx_writer << time << " ";
cx_writer << coefs.cx1 << " " << coefs.cy1 << " " << coefs.nu1 << " ";
cx_writer << coefs.cx2 << " " << coefs.cy2 << " " << coefs.nu2 << std::endl;
```

Соответственно можно поправить легенду в функции `initialize_saver()`.

Сами коэффициенты следует вычислять в функции `coefficients()`. Там нужно завести аггрегаторы на оба тела:

```
double sum_cpx1 = 0;
double sum_cpy1 = 0;
double sum_cfx1 = 0;
double sum_cfy1 = 0;
double sum_nu1  = 0;
double sum_cpx2 = 0;
double sum_cpy2 = 0;
double sum_cfx2 = 0;
double sum_cfy2 = 0;
double sum_nu2  = 0;
```

И далее заполнять их в зависимости от близости ячейки. Так же следует учесть значение граничной температуры при вычислении $\partial T/\partial n$ по формуле (7.16)

Например, для вертикальных граней после определения ячейки `left_cell`:

```
int gamma_i = gamma_closest_to_cell(left_cell);
double t_gamma = (gamma_i == 1) ? 0.5 : 1.0;
```

далее учесть при вычислении `dtdn` (два раза)

```
dtdn = (t_gamma - _t[left_cell]) / (_hx/2.0);
```

а также при выборе агрегатора

```
if (gamma_i == 1){
  sum_cpx1 += pnx * _hy;
  sum_cfy1 += dvdn * _hy;
  sum_nu1 += dtdn * _hy;
} else {
  sum_cpx2 += pnx * _hy;
  sum_cfy2 += dvdn * _hy;
  sum_nu2 += dtdn * _hy;
}
```

Аналогичную процедуру следует проделать и для горизонтальных граней.

В конце нужно правильным образом заполнить все поля переменной `coef`.

```
  coefs.cpx1 = 2.0*sum_cpx1;
  coefs.cpx2 = 2.0*sum_cpx2;
  ...
```

8 Лекция 8 (28.10)

8.1 Метод конечных объёмов

8.1.1 Уравнение Пуассона

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

$$-\nabla^2 u = f, \tag{8.1}$$

которое требуется решить в области D. Разобъём эту область на непересекающиеся подобласти E_i, $i = \overline{0, N-1}$ (рис. 17). Центры ячеек обозначим как $\mathbf{c}_i$.

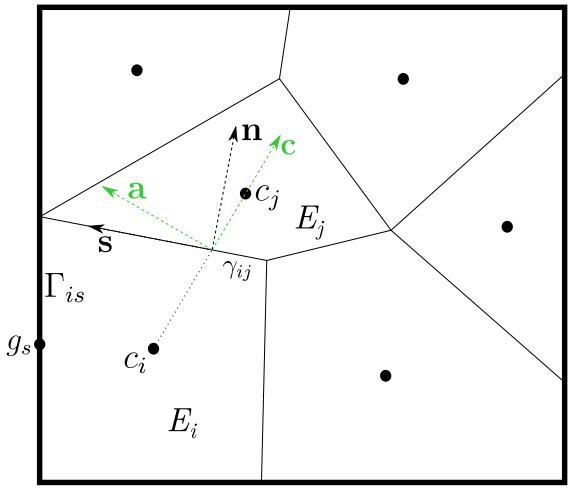


Рис. 17: Конечнообъёмная сетка

Проинтегрируем исходное уравнение по одной из подобластей. К интегралу в левой части применим формулу интегрирования по частям. Получим

$$-\int\limits_{\partial E_i} \frac{\partial u}{\partial n}\, ds = \int\limits_{E_i} f\, d\mathbf{x}. \tag{8.2}$$

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

Граница ячейки E_i состоит из внутренних граней γ_{ij} (индекс j здесь соответствует индексу соседней ячейки) и граней Γ_{is}, лежащих на внешней границе расчётной области D. Тогда интеграл по общей границе ячейки распишется через сумму интегралов по плоским поверхностям

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

Аппроксимирум производную $\partial u/\partial n$ на каждой из граней константой. Тогда её можно вынести из

под интегралов и предыдущее выражение записать в виде

$$\int\limits_{\partial E_i} \frac{\partial u}{\partial n}\, ds \approx \sum_j |\gamma_{ij}| \left(\frac{\partial u}{\partial n}\right)_{\gamma_{ij}} + \sum_s |\Gamma_{is}| \left(\frac{\partial u}{\partial n}\right)_{\Gamma_{is}} \tag{8.3}$$

Аналогично, анализируя интеграл правой части (8.2), приблизим значение функции правой части f внутри элемента E_i константой f_i, которую отнесём к центру элемента. Тогда

$$\int\limits_{E_i} f\, d\mathbf{x} \approx f_i\, |E_i|\,. \tag{8.4}$$

8.1.1.1 Обработка внутренних граней

Рассмотрим значение нормальной производной по грани γ_{ij}, входящее в первое слагаемое правой части (8.3). Для двумерного случая распишем градиент u в системе координат, образованной еодиничными векторами нормали $\mathbf{n}$ и касательной $\mathbf{s} = (-n_y, n_x)$ к грани γ_{ij}:

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

Теперь введём другую систему координат, которая образована векторами $\mathbf{c}$ – нормированный вектор $\mathbf{c}_j - \mathbf{c}_i$ и перпендикулярного к нему единичного вектора $\mathbf{c}^\perp = \mathbf{a} = (-c_y, c_x)$ (см. зелёные вектора на рис. 17). В этой системе

$$\nabla u = \frac{\partial u}{\partial c}\mathbf{c} + \frac{\partial u}{\partial a}\mathbf{a}.$$

Пользуясь формулами поворота систем координат $(\mathbf{c}, \mathbf{a}) \to (\mathbf{n}, \mathbf{s})$ искомую производную можно записать

$$\frac{\partial u}{\partial n} = \frac{\partial u}{\partial c}\cos(\widehat{\mathbf{c}, \mathbf{n}}) + \frac{\partial u}{\partial a}\sin(\widehat{\mathbf{c}, \mathbf{n}}). \tag{8.5}$$

Можно рассмотреть и обратный поворот $(\mathbf{n}, \mathbf{s}) \to (\mathbf{c}, \mathbf{a})$:

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

Тогда

$$\frac{\partial u}{\partial n} = \frac{\partial u}{\partial c}\frac{1}{\cos(\widehat{\mathbf{n}, \mathbf{c}})} - \frac{\partial u}{\partial s}\tan(\widehat{\mathbf{n}, \mathbf{c}}). \tag{8.6}$$

Таким образом, мы получили два соотношения для определения производной $\partial u/\partial n$: (8.5), (8.6).

Отметим, что в трёхмерном случае эти формулы так же остаются справедливыми. При этом векторы $\mathbf{s}$ и $\mathbf{a}$ следует строить в плоскости, образованной векторами $\mathbf{n}$ и $\mathbf{c}$:

$$\mathbf{s} = \frac{(\mathbf{n} \times \mathbf{c}) \times \mathbf{n}}{|(\mathbf{n} \times \mathbf{c}) \times \mathbf{n}|}, \quad \mathbf{a} = \frac{(\mathbf{n} \times \mathbf{c}) \times \mathbf{c}}{|(\mathbf{n} \times \mathbf{c}) \times \mathbf{c}|}.$$

При выводе этих формул используется тот факт, что результат векторного произведения перпендикулярено плоскости, образованной его аргументами.

Определим значения функции u в точках c_i, c_j как u_i, u_j. Тогда входящая в оба соотношения

(8.5), (8.6) производная $\partial u/\partial c$ может быть приближена конечной разностью

$$\frac{\partial u}{\partial c} \approx \frac{u_j - u_i}{|\mathbf{c}_j - \mathbf{c}_i|}.$$

Вторые слагаемые в правых частях (8.5), (8.6) можно в первом приближении отбросить, если считать, что угол между векторами $\mathbf{c}$ и $\mathbf{n}$ близок к нулю: $\widehat{\mathbf{n},\mathbf{c}} \approx 0$. Тогда искомую производную можно записать в виде:

$$\frac{\partial u}{\partial n} \approx \frac{u_j - u_i}{h_{ij}}, \tag{8.7}$$

где эффективное расстояние h_{ij} между узлами c_j и c_i для приближения (8.5) запишется как

$$h_{ij} = \frac{|\mathbf{c}_j - \mathbf{c}_i|}{\cos(\widehat{\mathbf{n},\mathbf{c}})} = \frac{|\mathbf{c}_j - \mathbf{c}_i|^2}{(\mathbf{c}_j - \mathbf{c}_i)\cdot\mathbf{n}}. \tag{8.8}$$

а для (8.6) –

$$h_{ij} = |\mathbf{c}_j - \mathbf{c}_i|\cos(\widehat{\mathbf{n},\mathbf{c}}) = (\mathbf{c}_j - \mathbf{c}_i)\cdot\mathbf{n} \tag{8.9}$$

Здесь для упрощений было использовано соотношение $\mathbf{a}\cdot\mathbf{b} = |\mathbf{a}||\mathbf{b}|\cos(\widehat{\mathbf{a},\mathbf{b}})$ и единичная длина вектора нормали: $|\mathbf{n}| = 1$.

Если вектора $\mathbf{c}$ и $\mathbf{n}$ сонаправлены, то $\cos(\widehat{\mathbf{n},\mathbf{c}}) = 0$ и тогда формулы (8.8), (8.9) идентичны. Если же при этом равны и расстояния от точек c_j, c_i до границы γ_{ij}, то конечная разность (8.7) является симметричной и поэтому имеет второй порядок аппроксимации. Сетки, которые сохраняют такие свойства, называются pebi-сетками (perpendicular bisector). Строятся такие сетки на основе ячеек Вороного.

Для сильно скошенных сеток кажется, что использование формулы (8.8) безопаснее чем (8.9). Потому что отброшенное из формулы (8.6) слагаемое

$$\frac{\partial u}{\partial s}\tan(\widehat{\mathbf{n},\mathbf{c}})$$

стремится к бесконечности в вырожденном случае $\widehat{\mathbf{n},\mathbf{c}} \to \frac{\pi}{2}$. Однако следует понимать, что обе эти формулы имеют одинаковый первый порядок точности (это следует из разложения синуса и тангенса вокруг нуля).

8.1.1.2 Учёт граничных условий

Для вычисления второго слагаемого в правой части (8.3) следует расписать значение нормальной к границе производной вида

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

Это делается с помощью граничных условий. Далее рассмотрим постановку трёх видов граничных условий.

Граничные условия первого рода Пусть на центре грани Γ_{is} задано значение искомой функции

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

Аппроксимацию производных будем проводить из тех же соображений, которые использовали при анализе внутренних граней. Только вместо центра соседнего элемента c_j будем использовать центр грани g_s. В первом приближении, отбрасывая касательные производные, придём к формуле аналигичной (8.7):

$$\frac{\partial u}{\partial n} \approx \frac{u^\Gamma - u_i}{h_{is}}, \tag{8.10}$$

где эффективное расстояние h_{is} зависимости от использованного подхода (8.5) или (8.6) вычисляется по одному из соотношений

$$h_{is} = \frac{|\mathbf{g}_s - \mathbf{c}_i|^2}{(\mathbf{g}_s - \mathbf{c}_i) \cdot \mathbf{n}}, \tag{8.11}$$

$$h_{is} = (\mathbf{g}_s - \mathbf{c}_i) \cdot \mathbf{n}. \tag{8.12}$$

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

$$\mathbf{x} \in \Gamma_{is}: \quad \frac{\partial u}{\partial n} = q, \tag{8.13}$$

то это значение просто подставляется вместо соответствующей производной в (8.3).

Граничные условия третьего рода Теперь рассмотрим условия третьего рода

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

Распишем производную в форме (8.10):

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

откуда выразим u^Γ:

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

Подставляя это выражение в исходное граничное условие получим

$$\frac{\partial u}{\partial n} \approx \frac{\alpha}{1 - \alpha h_{is}} u_i + \frac{\beta}{1 - \alpha h_{is}}. \tag{8.14}$$

8.1.2 Одномерный случай

Рассмотрим результат конечнообъёмной аппроксимации задачи (8.1) в одномерном случае на равномерной сетке с шагом h (рис. 18).

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

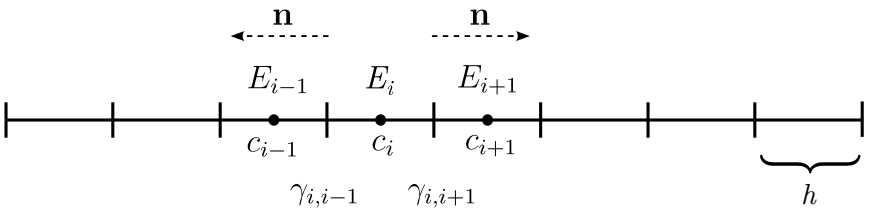


Рис. 18: Одномерная конечнообъёмная сетка

руются по формулам (8.7):

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

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

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 (8.2), получим знакомую конечноразностную схему аппроксимацию уравнения Пуассона

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

которая имеет второй порядок точности. Разница с методом конечных разностей здесь состоит в том, что значения сеточных векторов $\{u\}$, $\{f\}$ здесь приписаны к центрам ячеек, а не к их узлам. Это отличие проявит себя в аппроксимации граничных условий. Так, если на левой границе задано условие первого рода, то соответствующее уравнение согласно (8.10) примет вид

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

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

8.1.3 Сборка системы линейных уравнений

Подставим все полученные аппроксимации (8.7), (8.10), (8.13), (8.14) в уравнение (8.2):

$$-\sum_j \frac{|\gamma_{ij}|}{h_{ij}} (u_j - u_i) - \sum_{s \in \mathrm{I}} \frac{|\Gamma_{is}|}{h_{is}} \left(u^\Gamma - u_i\right) - \sum_{s \in \mathrm{II}} |\Gamma_{is}| q - \sum_{s \in \mathrm{III}} \frac{|\Gamma_{is}|}{1 - \alpha h_{is}} (\alpha u_i + \beta) = f_i |E_i|.$$

Здесь первое слагаемое в левой части отвечает за потоки через внутренние границы, второе – граничные условия первого рода, третье – граничные условия второго рода и четвёртое – граничные условия третьего рода. Далее перенесём все известные значения в правую часть и окончательно

получим линейное уравнение для i-го конечного объёма:

$$\begin{aligned}\sum_j \frac{|\gamma_{ij}|}{h_{ij}}(u_i - u_j) + \sum_{s\in \mathrm{I}} \frac{|\Gamma_{is}|}{h_{is}} u_i - \sum_{s\in \mathrm{III}} \frac{\alpha|\Gamma_{is}|}{1-\alpha h_{is}} u_i \\ = f_i|E_i| + \sum_{s\in \mathrm{I}} \frac{|\Gamma_{is}|}{h_{is}} u^\Gamma + \sum_{s\in \mathrm{II}} |\Gamma_{is}| q + \sum_{s\in \mathrm{III}} \frac{\beta|\Gamma_{is}|}{1-\alpha h_{is}}.\end{aligned} \tag{8.15}$$

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

$$Au = b.$$

8.1.3.1 Алгоритм сборки в цикле по ячейкам

Матрицу A и правую часть b системы (8.15) можно собирать в цикле по ячейкам: строчка за строчкой. Такой алгоритм выглядел бы следующим образом

for $i = \overline{0, N-1}$ – цикл по строкам СЛАУ
  $b_i = |E_i| f_i$
  for $j \in$ 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
    $v = |\Gamma_{is}|/h_{is}$
    A_{ii}+=v
    b_i+=$u^\Gamma v$
  endfor
  for $s \in$ bnd2(i) – цикл по граням ячейки i с условиями второго рода
    b_i+=$q|\Gamma_{is}|$
  endfor
  for $s \in$ bnd3(i) – цикл по граням ячейки i с условиями третьего рода
    $v = |\Gamma_{is}|/(1-\alpha h_{is})$
    A_{ii}−=αv
    b_i+=βv
  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.

8.1.3.2 Алгоритм сборки в цикле по граням

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

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

$$\begin{array}{ll}
\textbf{for } s \in \text{internal} & \text{– цикл по внутренним граням} \\
\quad i, j = \text{nei_cells(s)} & \text{– две ячейки, соседние с текущей гранью} \\
\quad v = |\gamma_{ij}|/h_{ij} & \\
\quad A_{ii} \mathrel{+}= v; \quad A_{jj} \mathrel{+}= v & \text{– диагональные коэффициенты матрицы} \\
\quad A_{ij} \mathrel{-}= v; \quad A_{ji} \mathrel{-}= v & \text{– внедиагональные коэффициенты матрицы} \\
\textbf{endfor} &
\end{array} \tag{8.16}$$

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

$$\begin{array}{ll}
\textbf{for } s \in \text{bnd1} & \text{– грани с условиями первого рода} \\
\quad i = \text{nei_cells(s)} & \text{– соседняя с граничной гранью ячейка} \\
\quad v = |\Gamma_{is}|/h_{is} & \\
\quad A_{ii} \mathrel{+}= v & \\
\quad b_i \mathrel{+}= u^{\Gamma} v & \\
\textbf{endfor} &
\end{array} \tag{8.17}$$

Условия второго рода:

$$\begin{array}{ll}
\textbf{for } s \in \text{bnd2} & \text{– грани с условиями второго рода} \\
\quad i = \text{nei_cells(s)} & \text{– соседняя с граничной гранью ячейка} \\
\quad b_i \mathrel{+}= |\Gamma_{is}| q & \\
\textbf{endfor} &
\end{array} \tag{8.18}$$

Условия третьего рода:

$$\begin{array}{ll}
\textbf{for } s \in \text{bnd3} & \text{– грани с условиями третьего рода} \\
\quad i = \text{nei_cells(s)} & \text{– соседняя с граничной гранью ячейка} \\
\quad v = |\Gamma_{is}|/(1 + \alpha h_{is}) & \\
\quad A_{ii} \mathrel{-}= \alpha v & \\
\quad b_i \mathrel{+}= \beta v & \\
\textbf{endfor} &
\end{array} \tag{8.19}$$

Первое слагаемое в правой части (8.15) учтём отдельным циклом:

$$\begin{aligned}&\textbf{for } i = \overline{0, N-1} \quad \text{– цикл по строкам}\\ &\quad b_i \mathrel{+}= |E_i| f_i\\ &\textbf{endfor}\end{aligned} \tag{8.20}$$

8.2 Конечнообъёмная сетка

Для реализации сборки системы линейных уравнений по алгоритму (8.16) – (8.20) необходимо уметь вычислять следующие параметры конечнообъёмной сетки:

- таблица связности грань-ячейка
- объём ячейки
- центр ячейки
- площадь грани
- центр грани
- нормаль к грани.

8.2.1 Определение конечнообъёмной сетки

TODO

8.2.2 Объём ячейки и площадь грани

TODO

8.2.3 Центры ячейки и грани

TODO

8.2.4 Аппроксимация значения в заданной точке

TODO

8.2.5 Интегрирование сеточной функции

Пусть задана сеточная функция $\{u_i\}$, которая аппроксимирует функцию u. Интеграл по области расчёта от этой функции можно расписать через сумму интегралов по каждой ячейке и далее воспользоваться тем фактом, что значение аппроксимированной функции внутри ячейки постоянно:

$$\int\limits_D u \, d\mathbf{x} = \sum_{i=0}^{N-1} \int\limits_{E_i} u \, d\mathbf{x} \approx \sum_{i=0}^{N-1} u_i |E_i|. \tag{8.21}$$

8.3 Пример расчётной программы

Рассмотрим пример решения двумерного уравнения (8.1) с граничными условиями первого рода. Для тестирования методики действовать будем по аналогии из п.2.1.2.2:

- Зададим точное решение в виде

$$u^e = \cos(10x^2)\sin(10y) + \sin(10x^2)\cos(10x);$$

- Расчитаем правую часть прямым дифференцированием

$$f = -\frac{\partial^2 u^e}{\partial x^2} - \frac{\partial^2 u^e}{\partial y^2};$$

- Используя подсчитанную f применим алгоритм метода конечных объёмов для получения численного решения u;

- Для вычисления отклонения численного решения от точного подсчитаем интеграл вида (2.8):

$$||u - u^e||_2 = \sqrt{\frac{1}{D}\int\limits_D (u - u^e)\, d\mathbf{x}}. \tag{8.22}$$

Пример программы лежит в файле `poisson_fvm_solve_test.cpp` в тесте `[poisson2-fvm]`. Программа использует регулярную двумерную сетку в единичном квадрате квадрате с разбиением в 20 ячеек по каждой оси. После расчёта файл с численным и точным решениями сохраняется в файл `poisson2_fvm.vtk`, а на печать выводится количество ячеек в сетке и полученная норма отклонения. Результат работы программы представлен на рис. 19. Поскольку вектор решений задан в центрах ячеек, то его отображение имеет мозаичный вид.

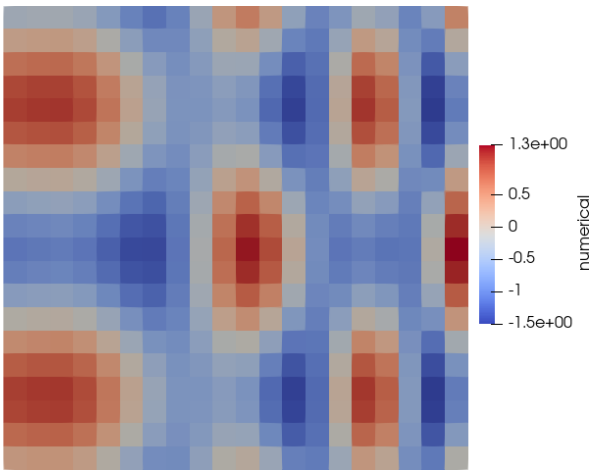


Рис. 19: Результат расчёта

8.3.1 Работа с сеткой

Несмотря на то, что на вход подаётся регулярная сетка, все алгоритмы используют сетку через абстракный интерфейс `IGrid`, который определён для сеток произвольной структуры и размерности.

Этот интерфейс (полностью объявленный в файле `grid/i_grid.hpp`) предоставляет следующие функции, используемые в алгоритмах (8.16) – (8.20):

- `IGrid::face_normal` – вектор нормали для заданной грани;
- `IGrid::tab_face_cell` – таблица связности грань–ячейка. Возвращает пару индексов ячеек. Первый из этих индексов соответствует ячейке, лежащей в направлении, противоположенном направлению нормали заданной грани. Для граничных граней один из этих индексов (в зависимости от направления нормали этой грани) равен глобальной константе `INVALID_INDEX`;
- `IGrid::face_center` – центр грани;
- `IGrid::face_area` – значение площади заданной грани;
- `IGrid::cell_center` – центр ячейки;
- `IGrid::cell_volume` – объём ячейки.

Конкретная реализация этих функций зависит от вида сетки. Так, для структурированной сетки `RegularGrid2D` их (тривиальная для таких сеток) реализация находится в файле `grid/regular_grid2.cpp`. Для произвольной двумерной неструктурированной сетки `UnstructuredGrid2D` общие алгоритмы, описанные в пункте 8.2 реализованы в файле `grid/unstructured_grid2d.cpp`

8.3.2 Функция верхнего уровня

На верхнем уровне создаётся сетка класса `RegularGrid2D`, которая затем используется при конструировании робочего класса `TestPoisson2FvmWorker`.

```
171 TEST_CASE("Poisson-fvm 2D solver", "[poisson2-fvm]"){
172   std::cout << std::endl << "--- cfd24_test [poisson2-fvm] --- " << std::endl;
173
174   size_t nx = 20;
175   RegularGrid2D grid(0.0, 1.0, 0.0, 1.0, nx, nx);
176   TestPoisson2FvmWorker worker(grid);
```

Далее вызывается решатель, который возвращает величину нормы:

```
177   double nrm = worker.solve();
```

результат сохраняется в файл

```
178   worker.save_vtk("poisson2_fvm.vtk");
```

печатается количество ячеек и полученная норма

```
179   std::cout << grid.n_cells() << " " << nrm << std::endl;
```

и полученная норма проверяется с предварительно расчитанным для заданных параметров значением

```
181   CHECK(nrm == Approx(0.04371).margin(1e-4));
```

8.3.3 Инициализация решения

Рассмотрим рабочий класс `TestPoisson2FvmWorker`. В его объявлении реализованы два статических метода: заданное точное решение

```
20    static double exact_solution(Point p){
21      double x = p.x();
22      double y = p.y();
23      return cos(10*x*x)*sin(10*y) + sin(10*x*x)*cos(10*x);
24    }
```

и подсчитанная правая часть, соответствующая этому решению

```
25    static double exact_rhs(Point p){
26      double x = p.x();
27      double y = p.y();
28      return (20*sin(10*x*x)+(400*x*x+100)*cos(10*x*x))*sin(10*y)
29          +(400*x*x+100)*cos(10*x)*sin(10*x*x)
30          +(400*x*sin(10*x)-20*cos(10*x))*cos(10*x*x);
31    }
```

В полях класса хранятся: ссылка на абстрактную сетку

```
37    const IGrid& _grid;
```

список внутренних граней

```
38    std::vector<size_t> _internal_faces;
```

и список граничных граней с условиями первого рода

```
45    std::vector<DirichletFace> _dirichlet_faces;
```

Каждая из этих граней описана структурой вида

```
39  struct DirichletFace{
40    size_t iface;
41    size_t icell;
42    double value;
43    Vector outer_normal;
44  };
```

в которой хранятся индекс грани, индекс ячейки, соседней с этой гранью, граничное значение функции в центре этой грани и направление внешней к области нормали.

Поле `_grid` передается в класс пользователем, в то время как списки `_internal_faces` и `_dirichlet_faces` собираются при конструировании рабочего класса.

```
53 TestPoisson2FvmWorker::TestPoisson2FvmWorker(const IGrid& grid): _grid(grid){
```

Чтобы отличить внутреннюю грань от граничной, необходимо проверить таблицу связности грань–ячейка. Если для грани одно из значений этой таблицы равно `INVALID_INDEX`, значит с соответствующей стороны нет ячейки, а грань является граничной. Эта проверка проводится в цикле по граням

```
55  for (size_t iface=0; iface<_grid.n_faces(); ++iface){
56    size_t icell_negative = _grid.tab_face_cell(iface)[0];
57    size_t icell_positive = _grid.tab_face_cell(iface)[1];
```

Если обе ячейки валидны, значит грань внутренняя и её индекс следует добавить в список `_internal_faces`

```
58    if (icell_positive != INVALID_INDEX && icell_negative != INVALID_INDEX){
59      // internal faces list
60      _internal_faces.push_back(iface);
```

Для граничных граней создаётся и заполняется структура `DirichletFace`. Сначала заполняются те поля, которые не зависят от направления: индекс грани и граничное значение (которое берётся из точного решения):

```
63      DirichletFace dir_face;
64      dir_face.iface = iface;
65      dir_face.value = exact_solution(_grid.face_center(iface));
```

Далее, если нормаль грани направлена вовне расчётной области (ячейка по направлению нормали невалида), то индекс соседней ячейки берётся с отрицательного направления, а внешняя к области нормаль совпадает с нормалью грани

```
66        if (icell_positive == INVALID_INDEX){
67          dir_face.icell = icell_negative;
68          dir_face.outer_normal = _grid.face_normal(iface);
```

иначе индекс ячейки берётся из положительного направления, а внешняя к области нормаль противоположна нормали грани

```
69        } else {
70          dir_face.icell = icell_positive;
71          dir_face.outer_normal = -_grid.face_normal(iface);
72        }
```

8.3.4 Реализация решения

Решение осуществляется вызовом функции

```
78 double TestPoisson2FvmWorker::solve(){
79   // 1. build SLAE
80   CsrMatrix mat = approximate_lhs();
81   std::vector<double> rhs = approximate_rhs();
82   // 2. solve SLAE
83   AmgcMatrixSolver solver;
84   solver.set_matrix(mat);
85   solver.solve(rhs, _u);
86   // 3. compute norm2
87   return compute_norm2();
88 }
```

в которой последовательно строятся левая и правая часть СЛАУ, вызывается решатель СЛАУ и производится сравнение с точным решением.

8.3.4.1 Сборка матрицы

В функции сборки левой части СЛАУ

```
107 CsrMatrix TestPoisson2FvmWorker::approximate_lhs() const{
```

реализуются алгоритмы (8.16), (8.17) в той их части, которая касается коэффициентов матрицы. Сначала согласно (8.16) проходит цикл по внутренним ячейкам

```
110   for (size_t iface: _internal_faces){
```

Для грани берётся нормаль

```
111     Vector normal = _grid.face_normal(iface);
```

Вычисляются соседние с гранью ячейки и их центры

```
112     size_t negative_side_cell = _grid.tab_face_cell(iface)[0];
113     size_t positive_side_cell = _grid.tab_face_cell(iface)[1];
114     Point ci = _grid.cell_center(negative_side_cell);
115     Point cj = _grid.cell_center(positive_side_cell);
```

находися эффективное расстояние между ними по модели (8.9)

```
116     double h = dot_product(normal, cj-ci);
```

и далее проводится заполнение матрицы найденным значением потока `coef`:

```
117     double coef = _grid.face_area(iface) / h;
118 
119     mat.add_value(negative_side_cell, negative_side_cell, coef);
120     mat.add_value(positive_side_cell, positive_side_cell, coef);
121     mat.add_value(negative_side_cell, positive_side_cell, -coef);
122     mat.add_value(positive_side_cell, negative_side_cell, -coef);
123   }
```

Учёт условий первого рода проводится в цикле по соответствующим граням согласно алгоритму (8.17) и модели вычисления эффективного расстояния (8.12)

```
125   for (const DirichletFace& dir_face: _dirichlet_faces){
126     size_t icell = dir_face.icell;
127     size_t iface = dir_face.iface;
128     Point gs = _grid.face_center(iface);
129     Point ci = _grid.cell_center(icell);
130     Vector normal = dir_face.outer_normal;
131     double h = dot_product(normal, gs-ci);
132     double coef = _grid.face_area(iface) / h;
133     mat.add_value(icell, icell, coef);
134   }
```

8.3.4.2 Сборка правой части

В сборке правой части

```
138 std::vector<double> TestPoisson2FvmWorker::approximate_rhs() const{
139   std::vector<double> rhs(_grid.n_cells(), 0.0);
```

сначала прогоняется алгоритм (8.20)

```
141   for (size_t icell=0; icell < _grid.n_cells(); ++icell){
142     double value = exact_rhs(_grid.cell_center(icell));
143     double volume = _grid.cell_volume(icell);
144     rhs[icell] = value * volume;
145   }
```

а затем учитываются граничные условия первого рода согласно (8.17)

```
147   for (const DirichletFace& dir_face: _dirichlet_faces){
148     size_t icell = dir_face.icell;
149     size_t iface = dir_face.iface;
150     Point gs = _grid.face_center(iface);
151     Point ci = _grid.cell_center(icell);
152     Vector normal = dir_face.outer_normal;
153     double h = dot_product(normal, gs-ci);
154     double coef = _grid.face_area(iface) / h;
155     rhs[icell] += dir_face.value * coef;
156   }
```

8.3.4.3 Вычисление нормы отклонения от точного решения

Вычисление выражения (8.22) с использованием алгоритма (8.21) производится в функции

```
160 double TestPoisson2FvmWorker::compute_norm2() const{
161   double norm2 = 0;
162   double full_area = 0;
163   for (size_t icell=0; icell<_grid.n_cells(); ++icell){
164     double diff = _u[icell] - exact_solution(_grid.cell_center(icell));
165     norm2 += _grid.cell_volume(icell) * diff * diff;
166     full_area += _grid.cell_volume(icell);
167   }
168   return std::sqrt(norm2/full_area);
169 }
```

8.4 Задание для самостоятельной работы

Получить решения для неструктурированных сеток В папке `test_data` корневой директории репозитория лежат скрипты построения сеток в программе `HybMesh`:

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

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

Необходимо с помощью этих скиптов построить сетки из ∼1000 ячеек. Далее на этих сетках решить задачу из п. 8.3 и сравнить поля решений и полученные нормы для этих сеток и равномерной структурированной сетки сходного разрешения.

Работать на основе теста `poisson2-fvm` в файле `poisson2_fvm_solve_test.cpp`. Можно создать отдельный тест, использующий те же классы для работы.

Для загрузки построенной сетки в решатель необходимо файл с сеткой поместить в каталог `test_data` и далее загрузить её в класс `UnstructuredGrid2D`. Нижеследующий код прочитает файл `test_data/pebigrid.vtk` и создаст рабочий класс с использованием прочитанной сетки

```
std::string fn = test_directory_file("pebigrid.vtk");
UnstructuredGrid2D grid = UnstructuredGrid2D::vtk_read(fn);
TestPoisson2FvmWorker worker(grid);
```

Получить порядок аппроксимации Для трёх видов сеток: структурированной, pebi и произвольной построить график сходимости аналогичный рис. 1. Для этого провести серию расчётов с различным количеством ячеек в диапазоне $N \in [500, 10^5]$.

Следует иметь ввиду, что на графике сходимости по оси абсцисс отложено линейное разбиение, вычисляемое как $n = 1/h$, где h – это характерный линейный размер ячейки. Для неструктурированных двумерных сеток этот линейный размер сетки можно вычислить через среднюю площадь ячейки A как $h = \sqrt{A}$, которую в свою очередь можно получить, разделив общую площадь на количество ячеек: $A = |D|/N$. Тогда, в случае единичного квадрата, линейное разбиение будет равно $n = \sqrt{N}$.

Реализовать граничные условия второго рода Решить ту же задачу используя граничные условия второго рода на pebi–сетке и сравнить полученные результы с решением задачи первого рода.

Для этого нужно в рабочий класс добавить функцию вычисления нормальной производной. Эта функция будет на вход принимать точку, лежащую на границе единичного квадрата, и возвращать

$\partial u^e/\partial n$, вычисленную прямым дифференцированием известного точного решения. Следует учитывать направление внешней нормали. Например, на вертикальной границе

$$x = 0: \quad \frac{\partial u}{\partial n} = -\frac{\partial u}{\partial x},$$
$$x = 1: \quad \frac{\partial u}{\partial n} = \frac{\partial u}{\partial x},$$

Для определения конкретной границы следует анализировать входную точку. Поскольку работа ведётся в числах с плавающей точкой, сравнение следует делать с некоторым допуском:

```
struct TestPoisson2FvmWorker{
  // точная производная по x
  static double exact_dudx(Point p){
    ...
  }
  // точная производная по y
  static double exact_dudy(Point p){
    ...
  }
  static double exact_dudn(Point p){
    double x = p.x();
    double y = p.y();
    if (std::abs(x) < 1e-6){
      // левая граница. Вернуть -du/dx
      return -exact_dudx(p);
    } else if (std::abs(x-1) < 1e-6){
      // правая граница. Вернуть du/dx
      return exact_dudx(p);
    } else if ...
  }
```

Для реализации алгоритма (8.18) предварительно нужно собрать информацию о гранях, которую следует хранить в массиве структур (по аналогии с `DirichletFace`)

```
struct NeumannFace{
  size_t iface;
  size_t icell;
  double value;
};
std::vector<NeumannFace> _neumann_faces;
```

В отличие от условий Дирихле, структура для условий Неймана не содержит нормалей, поскольку

в формулах (8.18) они не используются. Заполнять массив `_neumann_faces` следует в конструктуре `TestPoisson2FvmWorker` вместо `_dirichlet_faces`.

Алгоритм учёта условий второго рода не изменяет матрицу, поэтому обработку граничных граней следует убрать из функции `approximate_lhs`, а в функции сборки правой части `approximate_rhs` нужно реализовать формулы (8.18).

Задача в такой постановке имеет бесконечное множество решений, отличающихся на константу. Для получения однозначного ответа нужно задать точное решение в одной из ячеек. Выберем нулевую ячейку. Тогда нулевое уравнение СЛАУ нужно преобразовать к виду

$$u_0 = u^e(\mathbf{c}_0) \quad \Rightarrow \quad A_{0j} = \delta_{0j}, \quad b_0 = u^e(\mathbf{c}_0).$$

Для этого в функции `approximate_lhs` после окончания сборки следует вызвать метод

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

а в конце `approximate_rhs` –

```
rhs[0] = exact_solution(_grid.cell_center(0));
```

9 Лекция 9 (11.11)

9.1 Решение системы Уравнений Навье-Стокса методом конечных объёмов

TODO

9.1.1 Схема SIMPLE

TODO

9.1.2 Вычисление градиента давления методом наименьших квадратов

TODO

9.1.3 Интерполяция Rhie-Chow нормальной компоненты скорости

TODO

9.1.4 Порядок вычисления на итерации

TODO

9.2 Пример расчётной программы. Течение в каверне

Представлена в файле `cavern_2d_fvm_simple_test.cpp` в тесте `[cavern2-fvm-simple]`
TODO

9.3 Задание для самостоятельной работы

На основе теста `[cavern2-fvm-simple]` из файла `cavern_2d_fvm_simple_test.cpp` сравнить результат решения задачи о течении в каверне на структурированной, pebi и произвольной сетках. Использовать количество элементов $N \approx 2000$. Построение и чтение сеток проводить по аналогии с п.8.4.

1. Нарисовать результат решения (давление и векторы скорости) на различных итерациях для всех использованных сеток. Отметить количество требуемых итераций до сходимости с $\varepsilon = 10^{-2}$.
2. На основе полученного решения построить и сохранить в выходной vtk файл дивергенцию скорости и невязку решения.
3. Построить графики сходимости невязки (печатается в консоль при итерациях) от номера итерации для трёх сеток.

Вычисление дивиргенции скорости Распишем значения дивергенции скорости в ячейке E_i как среднеинтегральное и далее воспользуемся формулой Гаусса-Остроградского (A.8):

$$(\nabla \cdot \mathbf{u})_i \approx \frac{1}{|E_i|} \int\limits_{E_i} \nabla \cdot \mathbf{u}\, d\mathbf{x} = \frac{1}{|E_i|} \sum_j \int\limits_{\gamma_{ij}} u_n\, ds \approx \frac{1}{|E_i|} \sum_j (u_n)_{ij}\, |\gamma_{ij}|,$$

где γ_{ij} – все грани ячейки E_i, а $(u_n)_{ij}$ – значение нормальной (внешней нормали по отношению к ячейке E_i) скорости на этой грани. Расчёт по этой формуле нужно вести в цикле по граням, определяя для каждой грани пару соседних ячеек.

$d = \{0, ...\}$ – массив дивергенций. Длина равна количеству ячеек
for $s = \overline{0, N_f - 1}$ – цикл по всем граням
  $i, j =$ nei_cells(s) – ячейки, соседние с гранью: против и по нормали
  $c = (u_n)_s |\gamma_s|$
  if ($i \neq$ INVALID_INDEX) – если слева от грани есть ячейка
    $d_i \mathrel{+}= c/|E_i|$ – добавляем, так как нормаль внешняя для ячейки i
  endif
  if ($j \neq$ INVALID_INDEX) – если справа от грани есть ячейка
    $d_j \mathrel{-}= c/|E_j|$ – вычитаем, так как нормаль внутренняя для ячейки j
  endif
endfor

Массив нормальных скоростей к граням сетки доступен как поле рабочего класса `Cavern2DFvmSimpleWorker::_un_face`. Методы сетки `IGrid`, необходимые для программирования этой формулы:

- `IGrid::n_cells()`, `IGrid::n_faces()` – количество ячеек и граней сетки,
- `IGrid::tab_face_cell(iface)` – индексы пары соседних с гранью ячеек. Первая – против нормали, вторая – по. Для граничных граней один из возвращаемых индексов равен `INVALID_INDEX`.
- `IGrid::cell_volume(icell)` – объём ячейки E_i,
- `IGrid::face_area(iface)` – площадь грани γ_s.

Вычисление массива дивиргенций можно производить непосредственно в процедуре перед сохранением решения `Cavern2DFvmSimpleWorker::save_current_fields` с тем, чтобы сразу сохранить полученный массив дивергениций в файл вызовом `VtkUtils::add_cell_data(d, "div", filepath);`

Вычисление массива невязок происходит в функции установки текущих значение решения `set_uvp`:

```
141   std::vector<double> res_u = compute_residual_vec(_mat_uv, _rhs_u, _u);
142   std::vector<double> res_v = compute_residual_vec(_mat_uv, _rhs_v, _v);
143   res_u.resize(_grid.n_cells());
144   res_v.resize(_grid.n_cells());
145   for (size_t icell=0; icell < _grid.n_cells(); ++icell){
146     double coef = 1.0 / _tau / _grid.cell_volume(icell);
147     res_u[icell] *= coef;
148     res_v[icell] *= coef;
149   }
```

Для его сохранения в файл нужно либо повторить эту процедуру в функции сохранения, а лучше сделать `rhs_u`, `rhs_v` полями рабочего класса, чтобы иметь к ним доступ в методах класса. Для их сохранения нужно в процедуре сохранения `save_current_fields` вызвать

```
VtkUtils::add_cell_data(res_u, "res_u", filepath);
VtkUtils::add_cell_data(res_v, "res_v", filepath);
```

A Формулы и обозначения

A.1 Векторы

A.1.1 Обозначение

Геометрические вектора обозначаются жирным шрифтом $\mathbf{v}$. Скалярные координаты вектора – через нижний индекс с обозначением оси координат: (v_x, v_y, v_z). Если вектор $\mathbf{u}$ – вектор скорости, то его декартовые координаты имеют специальное обозначение $\mathbf{u} = (u, v, w)$. Операции в векторами имеют следующее обозначение (расписывая в декартовых координатах):

- Умножение на скалярную функцию

$$f\mathbf{u} = (fu_x)\mathbf{i} + (fu_y)\mathbf{j} + (fu_z)\mathbf{k}; \tag{A.1}$$

- Скалярное произведение

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

- Векторное произведение

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

A.1.2 Набла–нотация

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

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

В радиальной (r, ϕ, z):

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

В цилиндрической (r, θ, ϕ):

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

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

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

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

покоординатные выражения для декартовой системы):

$$\nabla f = (\nabla_x f,\ \nabla_y f,\ \nabla_z f) = \frac{\partial f}{\partial x}\mathbf{i} + \frac{\partial f}{\partial y}\mathbf{j} + \frac{\partial f}{\partial z}\mathbf{k}. \tag{A.4}$$

Результатом этой операции является вектор.

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

$$\nabla \cdot \mathbf{v} = \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} \tag{A.5}$$

результат которой – скалярная функция.

Двойное применение ∇ к скалярной функции – это оператор Лапласа:

$$\nabla \cdot \nabla f = \nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2} \tag{A.6}$$

Ротор – аналог векторного умножнения (A.3):

$$\nabla \times \mathbf{v} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ \nabla_x & \nabla_y & \nabla_z \\ v_x & v_y & v_z \end{vmatrix} = \left(\frac{\partial v_z}{\partial y} - \frac{\partial v_y}{\partial z}\right)\mathbf{i} - \left(\frac{\partial v_z}{\partial x} - \frac{\partial v_x}{\partial z}\right)\mathbf{j} + \left(\frac{\partial v_y}{\partial x} - \frac{\partial v_x}{\partial y}\right)\mathbf{k}. \tag{A.7}$$

A.2 Интегрирование

A.2.1 Формула Гаусса–Остроградского

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

$$\int\limits_E \nabla \cdot \mathbf{v}\, d\mathbf{x} = \int\limits_\Gamma v_n\, ds, \tag{A.8}$$

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

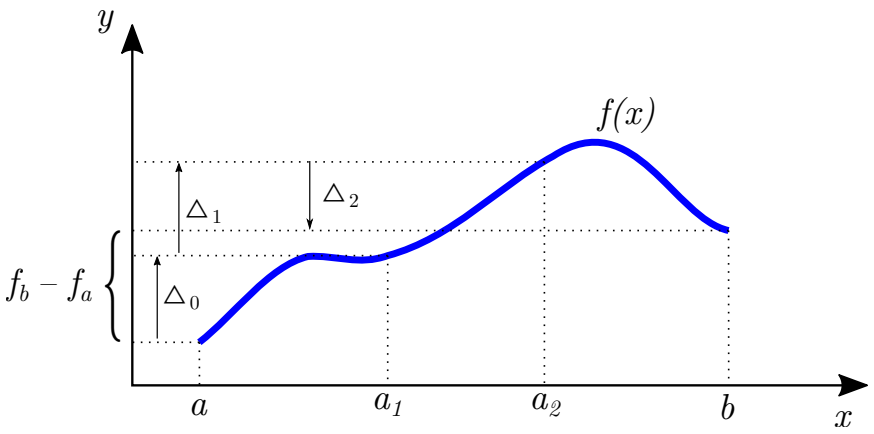


Рис. 20: Формула Гаусса-Остроградского в одномерном случае

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

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

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

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

А формула (A.8) – есть многомерное обобщение этого выражения.

A.2.2 Интегрирование по частям

Подставив в (A.8) $\mathbf{v} = f\mathbf{u}$, где f – некоторая скалярная функция, и расписав дивергенцию в виде

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

получим формулу интегрирования по частям

$$\int\limits_E \mathbf{u} \cdot \nabla f \, d\mathbf{x} = \int\limits_\Gamma f u_n \, ds - \int\limits_E f \nabla \cdot \mathbf{u} \, d\mathbf{x} \tag{A.9}$$

Распишем некоторые частные случаи для формулы (A.9). Для $\mathbf{u} = (n_x, 0, 0)$ получим

$$\int\limits_E \frac{\partial f}{\partial x} \, d\mathbf{x} = \int\limits_\Gamma f \cos(\widehat{\mathbf{n}, \mathbf{x}}) \, ds \tag{A.10}$$

При $\mathbf{u} = \nabla g$

$$\int\limits_E f \left(\nabla^2 g\right) \, d\mathbf{x} = \int\limits_\Gamma f \frac{\partial g}{\partial n} \, ds - \int\limits_E \nabla f \cdot \nabla g \, d\mathbf{x} \tag{A.11}$$

При $f = 1$ и $\mathbf{u} = \nabla g$

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

B Работа с инфраструктурой проекта CFDCourse

B.1 Сборка и запуск

B.1.1 Сборка проекта CFDCourse

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

B.1.1.1 Подготовка

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

   В Windows необходимо скачать и установить диструбутивы:

   - https://github.com/git-for-windows/git/releases/download/v2.38.1.windows.1/Git-2.38.1-64-bit.exe
   - https://github.com/Kitware/CMake/releases/download/v3.24.2/cmake-3.24.2-windows-x86_64.msi

   При установке cmake проследите, что бы путь к `cmake.exe` сохранился в системных путях. Msi установщик спросит об этом в диалоге.

   В **линуксе** используйте менеджеры пакетов, предоставляемые вашим дистрибутивом. Также проследите чтобы были доступны компиллятор `g++` и отладчик `gdb`.

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

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

4. Откройте терминал (git bash в Windows).

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

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

6. Клонируйте репозиторий

   ```
   > git clone https://github.com/kalininei/CFDCourse24
   ```

   В директории (`D:/git_repos` в примере) появится папка
   `CFDCourse24`, которая является корневой папкой проекта

B.1.1.2 VisualStudio

1. Создайте папку build в корне проекта CFDCourse24

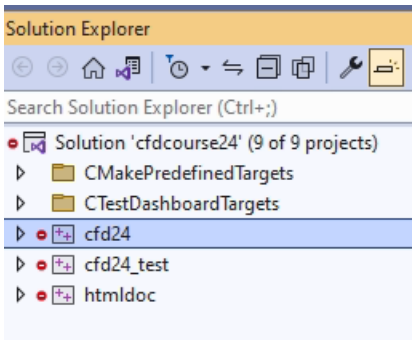
2. Скопируйте скрипт winbuild64.bat в папку build. Далее вносить изменения только в скопированном файле.

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

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

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

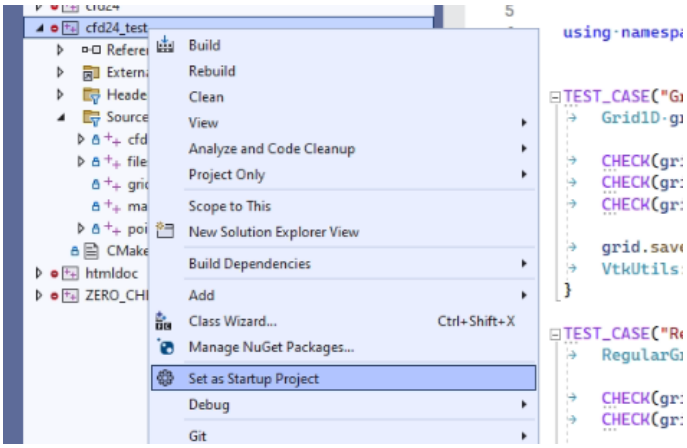
5. После сборки в папке `build` появится проект `VisualStudio` `cfdcourse24.sln`. Его нужно открыть в `VisualStudio`. Дерево решения должно иметь следующий вид:



Проекты:

- `cfd24` – расчётная библиотека
- `cfd24_test` – модульные тесты для расчётных функций

6. Проект `cfd24_test` необходимо назначить запускаемым проектом. Для этого нажать правой кнопкой мыши по проекту и в выпадающем меню выбрать соответствующий пункт. После этого заголовок проекта должен стать жирным.



7. Скомпиллировать решение. Несколько способов:

- `Ctrl+Shift+B`,

- `Build->Build Solution` в основном меню,
- `Build Solution` в меню решения в дереве решения,
- `Build` в меню проекта `cfd24_test`.

8. Запустить тесты (проект `cfd24_test`) нажав `F5` (или кнопку отладки в меню). После отработки должно высветиться сообщение об успешном прохождении всех тестов.

9. Бинарные файлы будут скомпиллированы в папку `CFDCourse24/build/bin/Debug`. В случае работы через отладчик выходная директория, куда будут скидываться все файлы (в частности, vtk), должна быть `CFDCourse24/build/src/test/`.

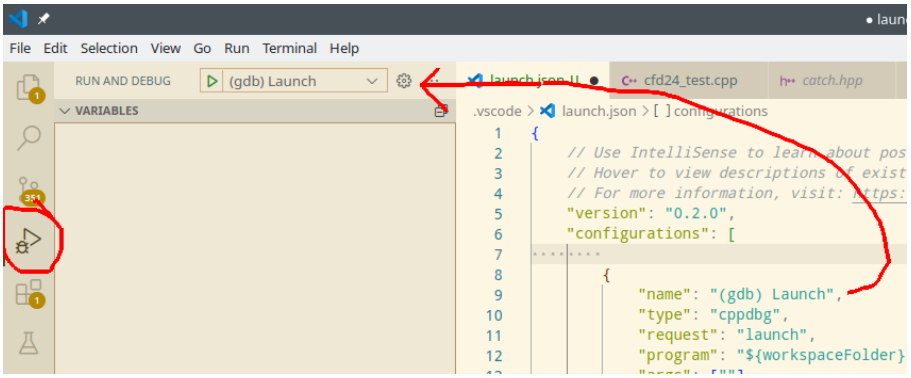
B.1.1.3 VSCode

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

2. Установить предлагаемые расширения cmake, c++

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

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



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

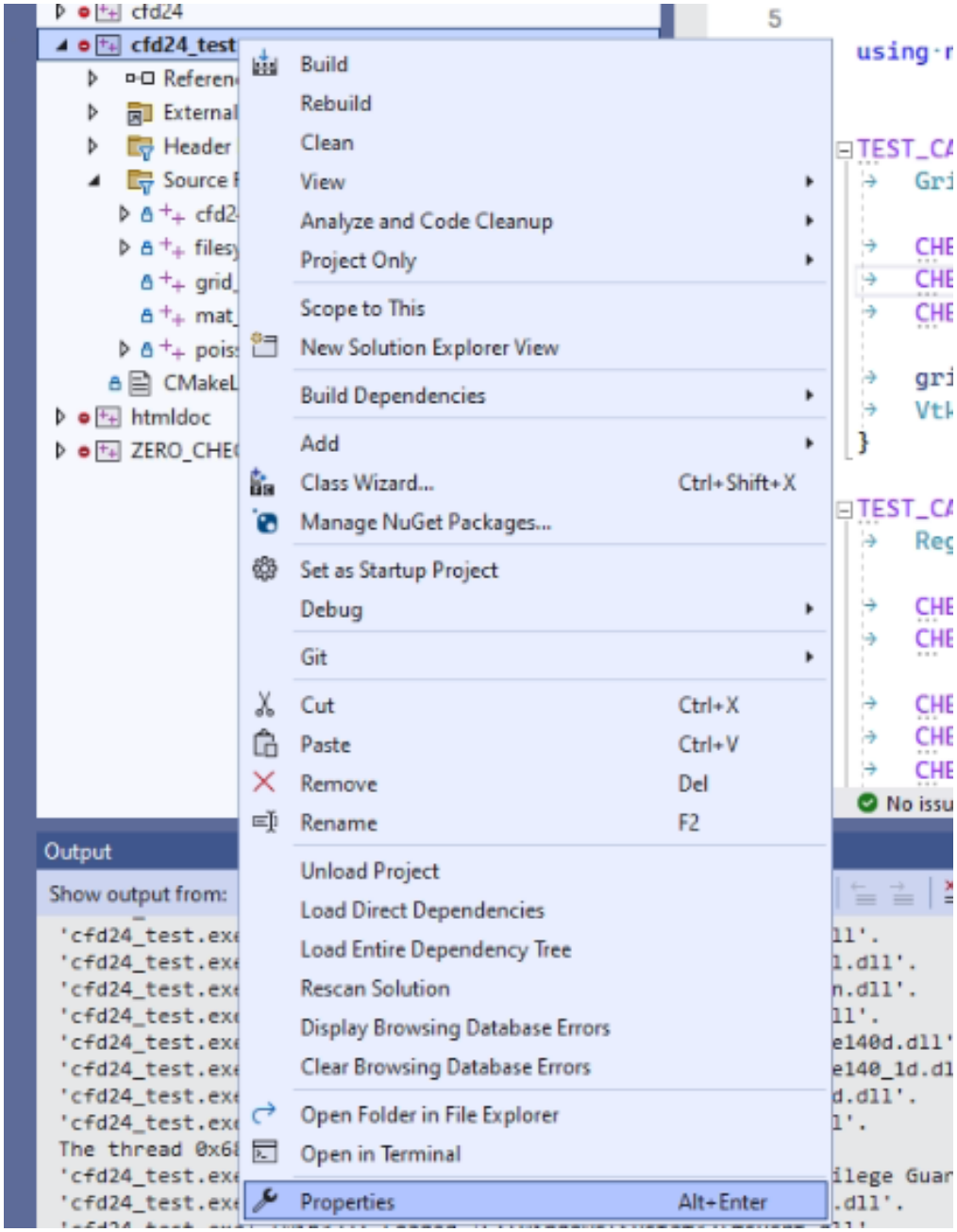
```
"name" : "(Windows) Launch",
"program": "${workspaceFolder}/build/bin/Debug/cfd24_test.exe"
```

B.1.2 Запуск конкретного теста

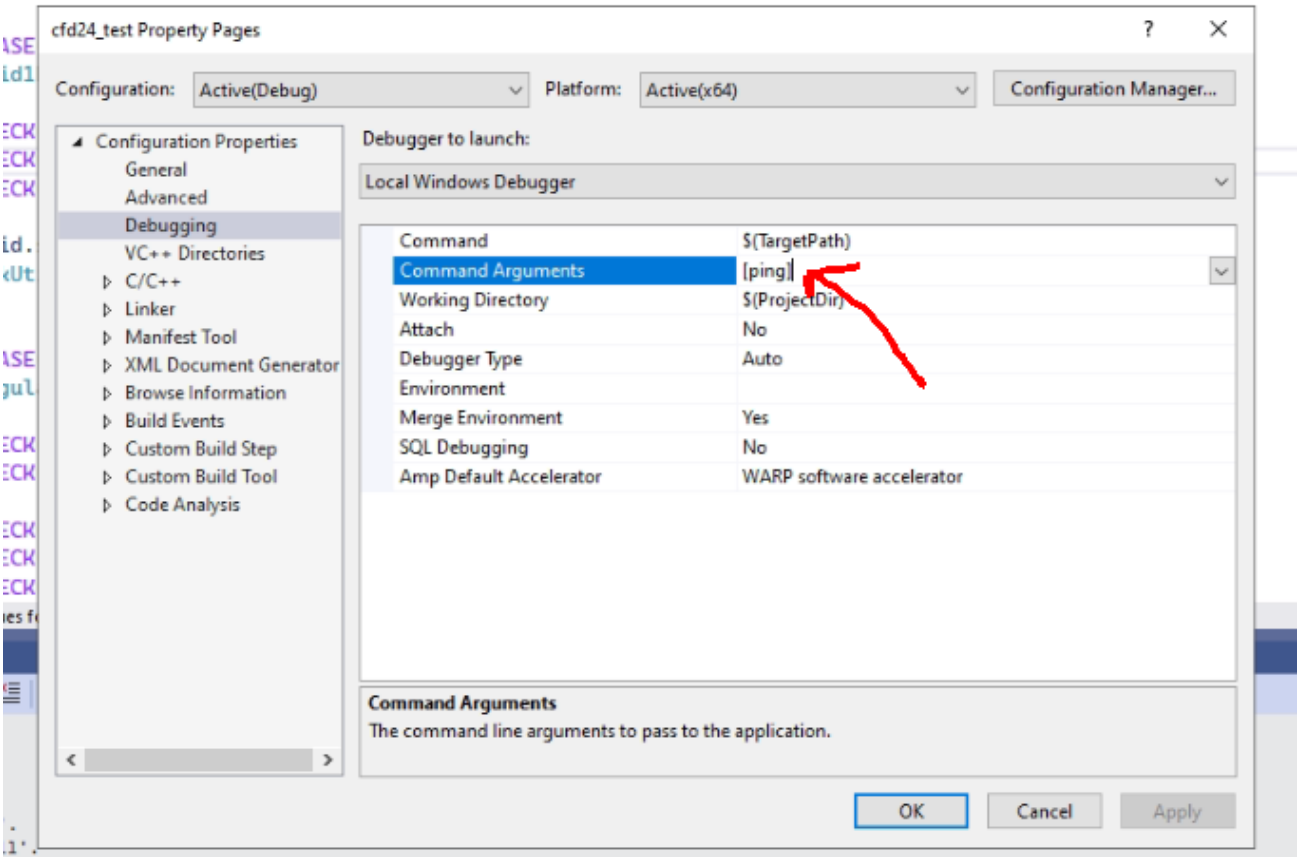
По умолчанию программа `cfd_test` прогоняет все объявленные в проекте тесты. Иногда может возникнуть необходимость запустить только конкретный тест в целях отладки или проверки. Для этого нужно передать программе аргумент с тегом для этого теста.

Тег для теста – это второй аргумент в макросе `TEST_CASE`, записанный в квадратных скобках. Добавлять нужно вместе со скобками. Например, `[ping]`.

Чтобы добавить аргумент в `VisualStudio`, необходимо в контекстном меню проекта `cfd_test` выбрать опции отладки



и там в поле Аргументы прописать нужный тэг.



В `VSCode` аргументы нужно добавлять в файле `.vscode/launch.json` в поле args в кавычках (см. картинку B.1.1.3 с настройками launch.json).

B.1.3 Сборка релизной версии

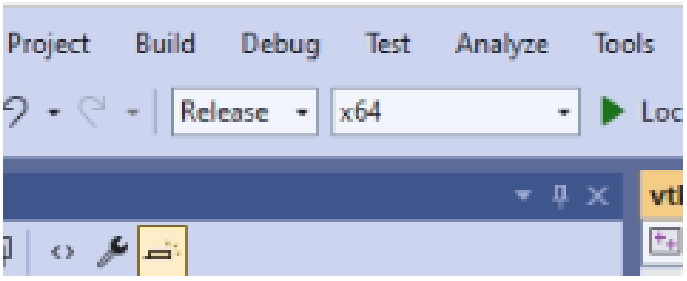
Релизная сборка программ даёт многократное увеличение производительности, но при этом отладка приложений в таком режиме невозможна.

Visual Studio

1. Создать папку `build-release` рядом с папкой `build`.
2. Скопировать в неё файл `winbuild64.bat` из папки `build`.
3. В скопированном файле произвести замену `Debug` на `Release`

```
-DCMAKE_BUILD_TYPE=Release ..
```

4. Запустить `winbuild64.bat` из новой папки
5. Открыть `build-release/cfdcourse24.sln` в `Visual Studio`
6. В проекте студии установить релизную сборку



7. Это новое решение, не связанное настройками с `debug`-версией. Поэтому нужно заново настроить запускускаемым проектом `cfd_test` и, если нужно, настроить аргументы отладки.

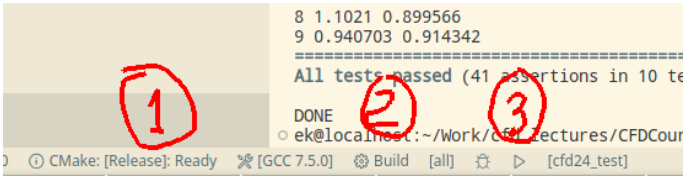
8. Бинарные файлы будут скомпиллированы в папку `CFDCourse24/build_release/bin/Release`. В случае работы через отладчик выходная директория – `CFDCourse24/build_release/src/test/`.

VSCode

1. Выбрать релизную сборку в `build variant`

2. Нажать `Build`

3. Нажать `Launch`



B.2 Git

B.2.1 Основные команды

Все команды выполнять в терминале (`git bash` для виндоус), находясь в корневой папке проета CFDCourse24.

- Для **смены директории** использовать команду `cd`. Например, находясь в папке `A` перейти в папку `A/B/C`

```
> cd B/C
```

- **Подняться** на директорию выше

```
> cd ..
```

- **Просмотр статуса** текущего репозитория: текущую ветку, все изменённые файлы и т.п.

```
> git status
```

- **Сохранить и скоммитить** изменения в текущую ветку

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

"message" – произвольная информация о текущем коммите, которая будет приписана к этому коммиту

- **Переключиться на ветку** main

```
> git checkout main
```

работает только в том случае, если все файлы скоммичены и статус ветки 'Up to date'

- **Создать новую ветку** ответвлённую от последнего коммита текущей ветки и переключиться на неё

```
> git checkout -b new-branch-name
```

new-branch-name – имя новой ветки. Пробелы не допускаются

Эта комманда работает даже если есть нескоммиченные изменения. Если необходимо скоммитить изменеия в новую ветку, сразу за этой командой нужно вызвать

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

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

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

Все изменения будут утеряны

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

```
> git pull
```

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

```
> git fetch
```

Обновленная ветка будет доступна по имени origin/имя ветки.

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

```
> git log
```

- **Просмотр доступных веток** в текущем репозитории

```
> git branch
```

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

```
> git gui
```

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

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

B.2.2 Порядок работы с репозиторием CFDCourse

Основная ветка проекта –
`main`. После каждой лекции (в течении 1-2 дней) в эту ветку будет отправлен коммит с сообщением `after-lect{index}`. Этот коммит будет содержать краткое содержание лекции, задание по итогам лекции и необходимые для этого задания изменения кода.

Если предполагается работа с кодом на лекции, то перед лекцией в эту ветку будет отправлен коммит с сообщением `before-lect{index}`. Этот коммит содержит изменения кода для работы на лекции.

Таким образом, **после лекции** необходимо выполнить следующие команды (находясь в ветке `main`)

```
> git reset --hard  # очистить локальную копию от изменений,
                    # сделанных на лекции (если они не представляют ценности)
> git pull          # получить изменения
```

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

```
> git checkout -b work-lect{index}    # создать локальную ветку, содержащую задание
> git add .
> git commit -m "{свой комментарий}"  # скоммитить свои изменения в эту ветку
> git checkout main                   # вернуться на ветку main
> git pull                            # получить изменения
```

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

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

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

B.3 Paraview

B.3.1 Отображение одномерных графиков

B.3.2 Отображение изолиний для двумерного поля

B.3.3 Отображение двумерного поля в 3D

B.3.4 Отображение числовых данных для точек и ячеек

B.3.5 Отображение векторов скорости

B.4 Hybmesh

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

B.4.1 Работа в Windows

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

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

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

B.4.2 Работа в Linux

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

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

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

Сначала следует склонировать репозиторий в папку с репозиториями гита:

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

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

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

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

```
#add_subdirectory(bindings)
```

Далее, находясь в корневой директории репозитория HybMesh, запустить сборку

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

Для запуска скриптов нужно создать скрипт-прокладку

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

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

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

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