Содержание

1 Постановка задачи

1.1 Основные уравнения

Для решения уравнений гидродинамики придумано множество численных методов с различными степенью аппроксимации, областью устойчивости и подходам к граничным условиям. Однако каждая задача уникальна и требует особого исследования для подбора соответствующей схемы. Незначительная модификация уравнений способна сделать работающий для исходной системы метод абсолютно непригодным для решения новой.

В данной работе будет рассматриваться система уравнений мелкой воды, описывающая волны Россби в постоянном вертикальном магнитном поле в приближении β-плоскости [1]. Кроме того будет показано, какие сложности могут создавать даже незначительные значения (относительно остальных величин в системе) магнитного поля при решении данной задачи.

Запишем уравнения:

$$\begin{cases} \dfrac{\partial h}{\partial t} + \dfrac{\partial (hv_x)}{\partial x} + \dfrac{\partial (hv_y)}{\partial y} = 0 \\ \dfrac{\partial (hv_x)}{\partial t} + \dfrac{\partial \left(h\left(v_x^2 - B_x^2\right) + \frac{1}{2}gh^2\right)}{\partial x} + \dfrac{\partial \left(h(v_x v_y - B_x B_y)\right)}{\partial y} + \left(B_0 B_x - hv_y\left(f_0 + \dfrac{\partial f}{\partial y}y\right)\right) = 0 \\ \dfrac{\partial (hv_y)}{\partial t} + \dfrac{\partial \left(h(v_x v_y - B_x B_y)\right)}{\partial x} + \dfrac{\partial \left(h\left(v_y^2 - B_y^2\right) + \frac{1}{2}gh^2\right)}{\partial y} + \left(B_0 B_y + hv_x\left(f_0 + \dfrac{\partial f}{\partial y}y\right)\right) = 0 \\ \dfrac{\partial (hB_x)}{\partial t} + \dfrac{\partial \left(h(v_y B_x - v_x B_y)\right)}{\partial y} + B_0 v_x = 0 \\ \dfrac{\partial (hB_y)}{\partial t} + \dfrac{\partial \left(h(v_x B_y - v_y B_x)\right)}{\partial x} + B_0 v_y = 0 \end{cases} \tag{1}$$

Вывод данной системы производится посредством выбора тонкого слоя, внутри которого значения скорости и поля вдоль вертикали изменяются незначительно, и последующего интегрирования уравнений мелкой воды по его толщине. Подробности можно найти в статье [2].

Численные эксперименты показали, что для наблюдения волн Россби необходимо учитывать рельеф поверхности планеты либо динамику более плотных слоёв атмосферы, в противном случае даже значительные возмущения начальных условий не приводят к появлению волн и в конечном счёте затухают [3]. В связи с этим добавим в слагаемое S компоненты градиента поверхности, в качестве которой будет выступать двумерная функция Гаусса с одинаковой дисперсией по обеим осям:

$$H(x,y) = 4000 \cdot \frac{1}{2\pi\sigma^2} e^{-\frac{(x-\mu_1)^2 + (y-\mu_2)^2}{2\sigma^2}} \tag{2}$$

Тогда система примет вид

$$\frac{\partial U}{\partial t} + \frac{\partial X}{\partial x} + \frac{\partial Y}{\partial y} + S = 0 \tag{3}$$

$$\begin{cases} U = \begin{pmatrix} h & hv_x & hv_y & B_x & B_y \end{pmatrix}^T \\ X = \begin{pmatrix} hv_x & h\left(v_x^2 - B_x^2\right) + \dfrac{1}{2}gh^2 & h\left(v_x v_y - B_x B_y\right) & 0 & h\left(v_x B_y - v_y B_x\right) \end{pmatrix}^T \\ Y = \begin{pmatrix} hv_y & h\left(v_x v_y - B_x B_y\right) & h\left(v_y^2 - B_y^2\right) + \dfrac{1}{2}gh^2 & h\left(v_y B_x - v_x B_y\right) & 0 \end{pmatrix}^T \\ S = \begin{pmatrix} 0 & B_0 B_x - hv_y\left(f_0 + \dfrac{\partial f}{\partial y}y\right) + g\dfrac{\partial H}{\partial x} & B_0 B_y + hv_x\left(f_0 + \dfrac{\partial f}{\partial y}y\right) + g\dfrac{\partial H}{\partial y} & B_0 v_x & B_0 v_y \end{pmatrix}^T \end{cases} \tag{4}$$

1.2 Подходы к решению

Для решения систем дифференциальных уравнений применяется множество численных методов, среди которых метод конечных разностей (FDM), метод конечных объёмов (FVM) и метод конечных элементов (FEM), используемые для решения гидродинамических задач. Рассмотрим каждый из них подробнее.

- Метод конечных разностей заключается в разложении исследуемых функций по Тейлору для аппроксимации их производных, фактически сводя систему дифференциальных уравнений к системе (или серии систем) алгебраических уравнений.

$$f(x_0 + \Delta x) = f(x_0) + \frac{1}{1!}\Delta x f'(x_0) + \frac{1}{2!}\Delta x^2 f''(x_0) + ...$$

$$f'(x_0) = \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x} - \frac{1}{2!}\Delta x f''(x_0) - ... = \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x} + o(\Delta x)$$

Таким образом вычисление значений функции происходит в узлах расчётной сетки [4]. Преимуществами такого подхода являются простота реализации, возможность выбора различных независимых методов для интегрирования по времени и пространству и увеличение порядка аппроксимации путём использования большего числа узлов сетки. Недостаток метода в затруднительном описании сложных геометрий на всей исследуемой области и в области резких перепадов производных функций.

- Метод конечных объёмов же рассматривается в некоторых ограниченных "секциях"расчётной области, соприкасающиеся друг с другом. Возьмём некоторое гиперболическое уравнение и проинтегрируем его по произвольному объёму:

$$\frac{\partial u}{\partial t} + div(\overline{f}) = S(u)$$

$$\int\limits_V \frac{\partial u}{\partial t} dV + \int\limits_V div(\overline{f}) dV = \int\limits_V S(u) dV$$

Пользуясь теоремой Остроградского-Гаусса, получим закон сохранения:

$$\int\limits_V div(\overline{f}) dV = \oint\limits_{\partial V} (\overline{f} \cdot \overline{n}) d\sigma$$

$$\int\limits_V \frac{\partial u}{\partial t} dV + \oint\limits_{\partial V} (\overline{f} \cdot \overline{n}) d\sigma = \int\limits_V S(u) dV$$

Теперь для численного решения необходимо дискретизировать данное уравнение. Искомые величины усредняются по конечным объёмам:

$$< u_i >= \frac{1}{V_i} \int\limits_{V_i} u dV$$

$$\frac{1}{V_i} \int\limits_{V_i} S(u) dV = S(< u_i >) + O(h^2)$$

$$\oint\limits_{\sigma_k} (\overline{f} \cdot \overline{n}) d\sigma = \overline{f}_k \cdot \overline{n}_k \sigma_k + O(h^p),$$

где σ - площадь грани (длина отрезка) k, $\overline{n}_k$ - внешняя единичная нормаль к грани (отрезку) k, $\overline{f}_k$ - значение потока в центре k, а h - расстояние между центрами масс конечных объёмов. Тогда схема приобретает вид

$$\frac{\partial < u_i >}{\partial t} + \frac{1}{V_i} \sum_{k=1}^{N_i} \overline{f}_k \cdot \overline{n}_k \sigma_k = S(< u_i >).$$

Для окончательного построения какой-либо схемы необходимо определить подход к вычислению $\overline{f}_k$ и аппроксимации по времени [5].

Существуют ещё метод конечных элементов (FEM) и его комбинация с методом конечных объёмов (DGFEM), однако из-за сложности описания и реализации оставим их за пределами данной работы [6].

Поскольку мы будем рассматривать кольцо атмосферы, заключённое между широтами 20° и 70°, что исключает возможность наличия сложной геометрии расчётной области, для простоты реализации воспользуемся методами класса конечных разностей.

1.3 Начальные и граничные условия

Чтобы задать начальные условия, необходимо определиться, для какой планеты мы решаем задачу. За основу возьмём Землю, поскольку для неё посчитано и измерено больше параметров. За основу возьмём уравнения геострофического баланса, поскольку нам важно убрать лишние колебания на ранних расчётных шагах:

$$fv_y = \frac{1}{\rho}\frac{\partial p}{\partial x} \tag{5}$$

$$fv_x = -\frac{1}{\rho}\frac{\partial p}{\partial y} \tag{6}$$

$$p = \rho g h \tag{7}$$

Так как в нашей модели мы не учитываем сжимаемость газа, то плотность можно вынести из-под знака дифференциала. Для расчёта высоты возьмём среднюю приблизительную скорость ветра (20) и подставим её в уравнение 23, чтобы получить градиент вдоль долготы (градиент вдоль широт примем нулём). В качестве константы при решении дифференциального уравнения возьмём высоту в 10км, которая будет соответствовать значению высоты на широте 45°. Пользуясь приближением β-плоскости, разложим параметр Кориолиса относительно той же широты. Тогда выражение для высоты примет вид

$$f(y) = f_0 + \beta(y - y_{45^\circ}) \tag{8}$$

$$\begin{aligned} h(z) &= -\frac{< v >}{g}\int_{y_{45^\circ}}^{z}(f_0 + \beta(y - y_{45^\circ}))dy = \\ &= \frac{< v >}{g} y_{45^\circ}\left(f_0 + \beta\left(\frac{y_{45^\circ}^2}{2} - y_{45^\circ}^2\right)\right) - \frac{< v >}{g} z\left(f_0 + \beta\left(\frac{z^2}{2} - z y_{45^\circ}\right)\right) = \\ &= \frac{< v >}{2g}(f(z) + f_0)(y_{45^\circ} - z) \approx \\ &\approx \frac{f(z) < v >}{g}(y_{45^\circ} - z) \end{aligned} \tag{9}$$

Для распределения скоростей по расчётной области подставим полученную высоту в уравнения 21 и 23. Таким образом мы получаем начальные условия:

$$v_y = \frac{g}{f}\frac{\partial h}{\partial x} = 0 \tag{10}$$

$$v_x = \frac{g}{f}\frac{\partial h}{\partial y}, \tag{11}$$

где $\frac{\partial h}{\partial y}$ можно посчитать простой линейной аппроксимацией

$$\frac{\partial h_{i,j}}{\partial y} = \frac{h_{i,j+1} - h_{i,j-1}}{2\Delta y} \tag{12}$$

Распределение же компонент магнитного поля получим с помощью калькулятора [7], основанного на WMM модели.

При решении данной задачи мы получим не только искомые волны Россби, но и им симметричные, однако направление движения этих волн строго определяется направлением вращения планеты, поэтому необходимо задать граничные условия, которые будут способствовать затуханию "лишних"волн. Для этого на левой границе занулим производные скоростей вдоль широты (outlet condition), это будет означать, что возмущения, пересекающие данную границу уходят за пределы расчётной области. В то же время на правой границе зададим периодические условия для скоростей, поскольку мы ведём расчёт в кольце. Для остальных величин системы периодическими условия будут на обеих границах. На южной и северной границах зафиксируем начальные условия. Важно заметить, что процесс зануления производной на границе усложняется с повышением точности схемы, поскольку количество точек, участвующих в расчётах, увеличивается. Если для расчёта в данном конкретном узле не хватает значений, а дополнительные точки (ghost points) зарезервированы под другие узлы, то можно воспользоваться экстраполяцией, получаемой разложением функции по Тейлору. Например, экстраполяция по четырём точкам будет иметь вид:

$$f(x_j) = 4f(x_{j+1}) - 6f(x_{j+2}) + 4f(x_{j+3}) - f(x_{j+4}) \tag{13}$$

$$f(x_j) = 4f(x_{j-1}) - 6f(x_{j-2}) + 4f(x_{j-3}) - f(x_{j-4}) \tag{14}$$

1.4 Теорема Лакса-Филиппова-Рябенького

Прежде чем применять какую-либо схему, необходимо понять условия её применимости в контексте данной задачи, то есть исследовать схему на сходимость. Теорема Лакса-Филиппова-Рябенького утверждает, что конечно-разностная схема, аппроксимирующая решение дифференциального уравнения в частных производных с корректно поставленной задачей Коши, сходится к этому решению тогда и только тогда, когда эта схема устойчива.

Введём ошибку схемы как разность между точным и численным решениями в некоторый момент времени n после одного расчётного шага Δx:

$$\delta^n_{\Delta x} = y^n_{\Delta x} - y(n\Delta t)$$

Тогда условие аппроксимации будет записано в таком виде:

$$\lim_{\Delta x \to 0} \frac{\delta^n_{\Delta x}}{\Delta x} = 0$$

Схема будет иметь порядок аппроксимации m, если

$$\delta^n_{\Delta x} = O(\Delta x^{m+1}), \Delta x \to 0$$

Учитывая то, что все схемы, приведённые в работе, удовлетворяют условию аппроксимации, имеет смысл исследовать их только на устойчивость для определения области их сходимости, следуя теореме Лакса-Филиппова-Рябенького.

1.5 Исследование схем на устойчивость

Введём понятие устойчивости. Пусть есть некоторая дифференциальная краевая задача

$$Lu = f$$

и составленная для неё разностная схема

$$L_{\Delta x} u^{(\Delta x)} = f^{(\Delta x)}$$

Разностная схема () является устойчивой, если существуют такие числа $\Delta > 0$ и $\delta > 0$, что для любых $\Delta x < \Delta$ и $\left|\epsilon^{\Delta x}\right| < \delta$ разностная задача

$$L_{\Delta x} z^{(\Delta x)} = f^{(\Delta x)} + \epsilon^{(\Delta x)},$$

полученная добавлением возмущения $\epsilon^{(\Delta x)}$ к правой части, имеет одно единственное решение $z^{(\Delta x)}$, отклоняющееся от решения $u^{(\Delta x)}$ на величину, удовлетворяющую оценке

$$\left|z^{(\Delta x)} - u^{(\Delta x)}\right| \leq M \left|\epsilon^{(\Delta x)}\right|,$$

где M - константа, не зависящая от Δx.

Для исследования схем на устойчивость мы будем использовать метод Фурье или, как его ещё называют, признак фон Неймана. Рассмотрим простейшую разностную схему и запишем её в каноническом виде:

$$\frac{u_{n+1} - u_n}{\Delta x} + Au_n = f_n$$

$$y_{n+1} = R_{\Delta x} y_n + \Delta x \psi_n,$$

где $y_n = u_n$, $\psi_n = f_n$ $R_{\Delta x} = 1 - A\Delta x$. Важно заметить, что для многоступенчатых схем ψ_n будет отличаться от f_n. Для оценки $|R^n_{\Delta x}|$ воспользуемся собственными значениями оператора $R_{\Delta x}$, полученными из уравнения

$$det|R_{\Delta x} - \lambda E| = 0,$$

тогда получим

$$R^n_{\Delta x} y = \lambda^n y$$
$$|R^n_{\Delta x} y| = |\lambda|^n |y|$$
$$|R^n_{\Delta x}| \geq |\lambda|^n$$

Получается, что необходимое условие ограниченности $|R^n_{\Delta x}|$ заключается в том, что все собственные значения должны лежать в круге на комплексной плоскости:

$$|\lambda| \leq 1 + C\Delta x,$$

где C - некоторая константа, не зависящая от шага Δx.

Устойчивость называется абсолютной, если условие устойчивости выполняется независимо от соотношения шагов по времени и пространству. Важно заметить, что признак фон Неймана является необходимым условием, но не достаточным, поэтому мы можем говорить только об условной устойчивости, подразумевающую связь между шагами. Это требует от нас исследования схем для каждой конкретной решаемой системы для выяснения этой связи.

2 Схемы первого порядка

2.1 Виды схем

Схемы первого порядка имеют самую низкую точность, однако крайне просты в реализации и требуют достаточно мало ресурсов CPU/GPU. В силу склонности к сглаживанию подобные схемы будут оптимальны в случае, когда решение не имеет больших перепадов производных. Распишем некоторые варианты этих схем:

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0 \tag{15}$$

- Прямая схема Эйлера

$$u_j^{n+1} = u_j^n - \frac{a\Delta t}{2\Delta x}\left(u_{j+1}^n - u_{j-1}^n\right) \tag{16}$$

- Обратная схема Эйлера

$$u_j^{n+1} = u_j^n - \frac{a\Delta t}{2\Delta x}\left(u_{j+1}^{n+1} - u_{j-1}^{n+1}\right) \tag{17}$$

- Схема Upwind ("Уголок")

$$u_j^{n+1} = u_j^n - \frac{a\Delta t}{\Delta x}\left(u_{j+1}^n - u_j^n\right) \tag{18}$$

- Прямая схема Лакса-Фридрихса

$$u_j^{n+1} = \frac{1}{2}\left(u_{j+1}^n + u_{j-1}^n\right) - \frac{a\Delta t}{2\Delta x}\left(u_{j+1}^n - u_{j-1}^n\right) \tag{19}$$

Вышеперечисленные схемы подробно расписаны и сравнены между собой в [8], мы же в данной работе будем рассматривать только последнюю из них.

2.2 Схема Лакса-Фридрихса

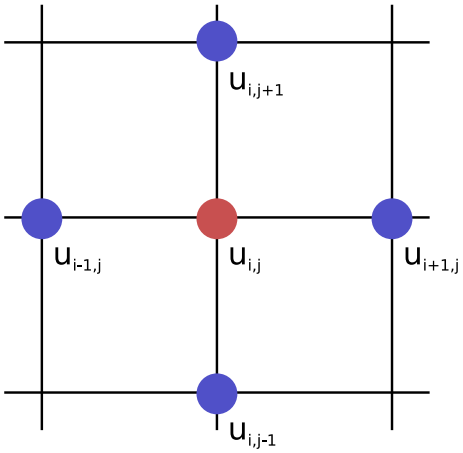

Рис. 1: Схема Лакса-Фридрихса

Для нашей системы в двумерном случае данная схема имеет вид

$$U_{i,j}^{n+1} = \frac{1}{4}\left(U_{i-1,j}^n + U_{i+1,j}^n + U_{i,j-1}^n + U_{i,j+1}^n\right) - \frac{\Delta t}{2}\left(\frac{X_{i+1,j}^n - X_{i-1,j}^n}{\Delta x} + \frac{Y_{i,j+1}^n - Y_{i,j-1}^n}{\Delta y} + 2S_{i,j}^n\right) \tag{20}$$

$$X_{i,j}^n = X(U_{i,j}^n)$$
$$Y_{i,j}^n = Y(U_{i,j}^n)$$
$$S_{i,j}^n = S(U_{i,j}^n)$$

На рисунке красным кружком отмечен искомый узел на следующем временном слое, синие кружки - опорные узлы схемы. Поскольку для получения искомого узла требуются ближайшие узлы, то для расчёта граничных условий вокруг сетки расчётной области необходимо создать "рамку" одинарной толщины из дополнительных узлов (ghost points). В случае outlet условия дополнительный узел будет равен предпоследнему на данном направлении.

2.3 Исследование схемы Лакса-Фридрихса на устойчивость

Рассмотрим линейное уравнение гиперболического типа первого порядка и представим его в виде разностного по схема Лакса-Фридрихса:

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

$$\frac{u_i^{n+1} - \frac{1}{2}(u_{i+1}^n + u_{i+1}^n)}{\Delta t} + \frac{a}{2}\frac{(u_{i+1}^n - u_{i+1}^n)}{\Delta x} = 0$$

Выразим решение через сумму точного решения разностного уравнения и погрешность округления и запишем уравнение для ошибки:

$$u_i^n = U_i^n + \epsilon_i^n$$

$$\frac{\epsilon_i^{n+1} - \frac{1}{2}(\epsilon_{i+1}^n + \epsilon_{i+1}^n)}{\Delta t} + \frac{a}{2}\frac{(\epsilon_{i+1}^n - \epsilon_{i+1}^n)}{\Delta x} = 0 \tag{21}$$

Разложим погрешность в ряд Фурье **(Поправить!!!!!!!!!)**:

$$\epsilon(x) = \sum_{m=1}^{M} A_m e^{ik_m x}, k_m = \frac{\pi m}{L}$$

Полагается, что амплитуда ошибки A_m является функцией времени, а поскольку изменение ошибки имеет экспоненциальный характер с течением времени, то функция погрешности будет иметь вид

$$\epsilon(x,t) = \sum_{m=1}^{M} e^{bt} e^{ik_m x}, k_m = \frac{\pi m}{L}$$

$$\epsilon_j^n = e^{bt_n} e^{ik_m x_j}$$

Тогда мы можем преобразовать уравнение 21:

$$\frac{e^{b(t_n+\Delta t)}e^{ik_m x_j} - \frac{1}{2}(e^{bt_n}e^{ik_m(x_j+\Delta x)} + e^{bt_n}e^{ik_m(x_j-\Delta x)})}{\Delta t} + \frac{a}{2}\frac{(e^{bt_n}e^{ik_m(x_j+\Delta x)} - e^{bt_n}e^{ik_m(x_j-\Delta x)})}{\Delta x} =$$

$$= \frac{e^{b\Delta t} - \frac{1}{2}(e^{ik_m\Delta x} + e^{-ik_m\Delta x})}{\Delta t} + \frac{a}{2}\frac{(e^{ik_m\Delta x} - e^{-ik_m\Delta x})}{\Delta x} =$$

$$= \frac{e^{b\Delta t} - cos(k_m\Delta x)}{\Delta t} + ia\frac{sin(k_m\Delta x)}{\Delta x} = 0$$

Введём коэффициент усиления (amplification factor), модуль которого должен быть меньше либо равен единице для выполнения условия устойчивости:

$$|G| = \left|\frac{\epsilon_j^{n+1}}{\epsilon_j^n}\right| = |e^{b\Delta t}| \leq 1$$

$$|e^b\Delta t| = \left|cos(k_m\Delta x) - i\frac{a\Delta t}{\Delta x}sin(k_m\Delta x)\right| \leq 1$$

$$cos^2(k_m\Delta x) + \frac{a^2\Delta t^2}{\Delta x^2}sin^2(k_m\Delta x) \leq 1$$

$$\frac{a^2\Delta t^2}{\Delta x^2}sin^2(k_m\Delta x) \leq sin^2(k_m\Delta x)$$

$$\frac{a^2\Delta t^2}{\Delta x^2} \leq 1$$

$$\Delta t \leq \frac{\Delta x}{|a|}$$

2.4 Исследование на устойчивость в случае нелинейной системы

В случае, когда система является нелинейной, метод фон Неймана не может быть напрямую применён, и требуется линеаризация. Для простоты рассмотрим одномерную систему уравнений мелкой воды:

$$\begin{cases} \dfrac{\partial h}{\partial t} + \dfrac{\partial (hv_x)}{\partial x} = 0 \\ \dfrac{\partial (hv_x)}{\partial t} + \dfrac{\partial \left(hv_x^2 + \frac{1}{2}gh^2\right)}{\partial x} = 0 \end{cases} \tag{22}$$

Преобразуем нелинейные слагаемые с помощью введения матрицы Якоби:

$$\begin{pmatrix} \dfrac{\partial (hv_x)}{\partial x} \\ \dfrac{\partial \left(hv_x^2 + \frac{1}{2}gh^2\right)}{\partial x} \end{pmatrix} = \begin{pmatrix} \dfrac{\partial F_1(h, hv_x)}{\partial x} \\ \dfrac{\partial F_2(h, hv_x)}{\partial x} \end{pmatrix} = A \cdot \begin{pmatrix} \dfrac{\partial h}{\partial x} \\ \dfrac{\partial (hv_x)}{\partial x} \end{pmatrix}$$

Поскольку h и hv_x неортогональны, сделаем замену переменных $q_1 = h, q_2 = hv_x$, тогда

$$\begin{pmatrix} F_1(q_1, q_2) \\ F_2(q_1, q_2) \end{pmatrix} = \begin{pmatrix} q_2 \\ \dfrac{q_2^2}{q_1} + \dfrac{1}{2}gq_1^2 \end{pmatrix}, A = \begin{pmatrix} \dfrac{\partial F_1}{\partial q_1} & \dfrac{\partial F_1}{\partial q_2} \\ \dfrac{\partial F_2}{\partial q_1} & \dfrac{\partial F_2}{\partial q_2} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -\dfrac{q_2^2}{q_1^2} + gq_1 & \dfrac{2q_2}{q_1} \end{pmatrix}$$

После обратной замены получим систему:

$$\begin{pmatrix} \partial_t h \\ \partial_t (hv_x) \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ -v_x^2 + gh & 2v_x \end{pmatrix} \begin{pmatrix} \partial_x h \\ \partial_x (hv_x) \end{pmatrix} = 0 \tag{23}$$

Для линеаризации необходимо зафиксировать значения матрицы Якоби для данного шага по времени. Каждое уравнение системы имеет свою ошибку, поэтому, ориентируясь на подход к линейному уравнению, запишем:

$$\begin{pmatrix} E^{nt+\Delta t} \\ D^{nt+\Delta t} \end{pmatrix} = \frac{1}{2} \begin{pmatrix} E^{nt}(e^{ik\Delta x} + e^{-ik\Delta x}) \\ V^{nt}(e^{ik\Delta x} + e^{-ik\Delta x}) \end{pmatrix} - \frac{\Delta t}{2\Delta x} \begin{pmatrix} 0 & 1 \\ -v_x^2 + gh & 2v_x \end{pmatrix} \begin{pmatrix} E^{nt}(e^{ik\Delta x} - e^{-ik\Delta x}) \\ V^{nt}(e^{ik\Delta x} - e^{-ik\Delta x}) \end{pmatrix} =$$

$$= \begin{pmatrix} E^{nt} cos(k\Delta x) \\ V^{nt} cos(k\Delta x) \end{pmatrix} - i\frac{\Delta t}{\Delta x} \begin{pmatrix} 0 & 1 \\ -v_x^2 + gh & 2v_x \end{pmatrix} \begin{pmatrix} E^{nt} sin(k\Delta x) \\ V^{nt} sin(k\Delta x) \end{pmatrix} =$$

$$= \begin{pmatrix} E^{nt} cos(k\Delta x) - iV^{nt} \dfrac{\Delta t}{\Delta x} sin(k\Delta x) \\ -iE^{nt} \dfrac{\Delta t}{\Delta x}(-v_x^2 + gh) sin(k\Delta x) + V^{nt}(cos(k\Delta x) - 2iv_x \dfrac{\Delta t}{\Delta x} sin(k\Delta x))) \end{pmatrix}$$

$$\begin{pmatrix} E^{nt+\Delta t} \\ D^{nt+\Delta t} \end{pmatrix} = G \begin{pmatrix} E^{nt} \\ D^{nt} \end{pmatrix} \tag{24}$$

$$G = \begin{pmatrix} cos(k\Delta x) & -i\dfrac{\Delta t}{\Delta x} sin(k\Delta x) \\ -i\dfrac{\Delta t}{\Delta x}(-v_x^2 + gh) sin(k\Delta x) & -2iv_x \dfrac{\Delta t}{\Delta x} sin(k\Delta x) + cos(k\Delta x) \end{pmatrix} \tag{25}$$

Матрица (25) называется матрицей усиления (amplification matrix). Модули её собственных значений должны быть меньше либо равны единице для всех узлов расчётной сетки на данном временном слое, что обеспечивает устойчивость схемы. В силу того, что мы решаем систему уравнений, собственные значения матрицы G будут зависеть от собственных значений матрицы A из уравнения 23.

$$\lambda_j = cos(k\Delta x) - iv_x \frac{\Delta t}{\Delta x} sin(k\Delta x) \pm i|sin(k\Delta x)|\sqrt{gh}\frac{\Delta t}{\Delta x} =$$

$$cos(k\Delta x) - i\frac{\Delta t}{\Delta x} sin(k\Delta x)(v_x \mp sgn(sin(k\Delta x))\sqrt{gh}) =$$

$$cos(k\Delta x) - i\frac{\Delta t}{\Delta x} sin(k\Delta x)(v_x \pm \sqrt{gh})$$

$$|\lambda_j| = \sqrt{cos^2(k\Delta x) + \frac{\Delta t^2}{\Delta x^2} sin^2(k\Delta x)(v_x \pm \sqrt{gh})^2} \leq 1$$

$$|\lambda_j| = \sqrt{1 + sin^2(k\Delta x)\left(\frac{\Delta t^2}{\Delta x^2}(v_x \pm \sqrt{gh})^2 - 1\right)} \leq 1$$

$$sin^2(k\Delta x)\left(\frac{\Delta t^2}{\Delta x^2}(v_x \pm \sqrt{gh})^2 - 1\right) \leq 0$$

$$\frac{\Delta t}{\Delta x}|v_x \pm \sqrt{gh}| \leq 1$$

$$\Delta t \leq \frac{\Delta x}{|v_x \pm \sqrt{gh}|} \tag{26}$$

Обратим внимание на то, что собственные значения матрицы A равны

$$\lambda_{12} = v_x \pm \sqrt{gh} \tag{27}$$

2.5 Реализация на языке C++

```
template <class T>
T solve(const T& tOffsetMinusX, const T& tOffsetPlusX,
        const T& tOffsetMinusY, const T& tOffsetPlusY) {
    return  (tOffsetMinusX + tOffsetPlusX + tOffsetMinusY + tOffsetPlusY) / 4.0 -
            ((funcX(tOffsetPlusX) - funcX(tOffsetMinusX)) * mRatioX +
            (funcY(tOffsetPlusY) - funcY(tOffsetMinusY)) * mRatioY) / 2.0;
}
```

Листинг 1: Lax-Friedrichs scheme 2D

2.6 Результаты

3 Схемы второго порядка

3.1 Виды схем

Схемами второго порядка пользуются, когда необходима точность при решении уравнений, они требуют больше опорных узлов и затрачивают больше ресурсов CPU/GPU, нежели схемы первого порядка. Недостаток данной, а также схем высших порядков, в неустойчивости в областях разрыва производных, о чём будет сказано в разделе 3.9. Распишем некоторые схемы второго порядка:

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0 \tag{28}$$

- Схема Лакса-Вендроффа

$$u_j^{n+1} = u_j^n - \frac{c\Delta t}{2\Delta x}\left(u_{j+1}^n - u_{j-1}^n\right) - \frac{c^2\Delta t^2}{2\Delta x^2}\left(u_{j+1}^n - 2u_j^n + u_{j-1}^n\right)$$

- Схема Leapfrog

$$u_j^{n+1} = u_j^{n-1} - \frac{c\Delta t}{\Delta x}\left(u_{j+1}^n - u_{j-1}^n\right)$$

- Схема Кранка-Николсона

$$u_j^{n+1} = u_j^n - \frac{c\Delta t}{4\Delta x}\left(u_{j+1}^n - u_{j-1}^n\right) - \frac{c\Delta t}{4\Delta x}\left(u_{j+1}^{n+1} - u_{j-1}^{n+1}\right)$$

- Схема Box

$$\left(1 - \frac{c\Delta t}{\Delta x}\right)u_j^{n+1} + \left(1 - \frac{c\Delta t}{\Delta x}\right)u_{j+1}^{n+1} = \left(1 - \frac{c\Delta t}{\Delta x}\right)u_j^n + \left(1 + \frac{c\Delta t}{\Delta x}\right)u_{j+1}^n$$

Вышеперечисленные схемы вместе с уже упоминавшимися схемами первого порядка подробно расписаны и сравнены между собой в [8]. В данной работе нас будет интересовать метод Рихтмайера [9], являющийся методом класса схем Лакса-Вендроффа, и его разновидность, схема Рихтмайера-Цваса.

3.2 Схема Рихтмайера

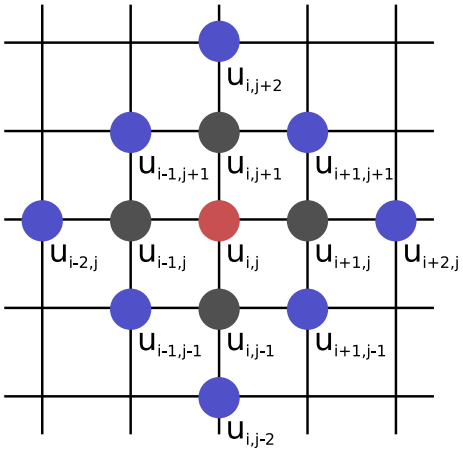

Рис. 2: Схема Рихтмайера

Применимо к нашей системе, в двумерном случае схема Рихтмайера в виде четырёх расчётов промежуточной ступени и финального расчёта второй ступени. Такие схемы называют двухшаговыми:

$$\begin{cases} U_{i-1,j}^{n+\frac{1}{2}} = \frac{1}{4}\left(U_{i,j}^n + U_{i-2,j}^n + U_{i-1,j+1}^n + U_{i-1,j-1}^n\right) - \frac{\Delta t}{2}\left(\frac{X_{i,j}^n - X_{i-2,j}^n}{\Delta x} + \frac{Y_{i-1,j+1}^n - Y_{i-1,j-1}^n}{\Delta y} + 2S_{i-1,j}^n\right) \\ U_{i+1,j}^{n+\frac{1}{2}} = \frac{1}{4}\left(U_{i+2,j}^n + U_{i,j}^n + U_{i+1,j+1}^n + U_{i+1,j-1}^n\right) - \frac{\Delta t}{2}\left(\frac{X_{i+2,j}^n - X_{i,j}^n}{\Delta x} + \frac{Y_{i+1,j+1}^n - Y_{i+1,j-1}^n}{\Delta y} + 2S_{i+1,j}^n\right) \\ U_{i,j-1}^{n+\frac{1}{2}} = \frac{1}{4}\left(U_{i+1,j-1}^n + U_{i-1,j-1}^n + U_{i,j}^n + U_{i,j-2}^n\right) - \frac{\Delta t}{2}\left(\frac{X_{i+1,j-1}^n - X_{i-1,j-1}^n}{\Delta x} + \frac{Y_{i,j}^n - Y_{i,j-2}^n}{\Delta y} + 2S_{i,j-1}^n\right) \\ U_{i,j+1}^{n+\frac{1}{2}} = \frac{1}{4}\left(U_{i+1,j+1}^n + U_{i-1,j+1}^n + U_{i,j+2}^n + U_{i,j}^n\right) - \frac{\Delta t}{2}\left(\frac{X_{i+1,j+1}^n - X_{i-1,j+1}^n}{\Delta x} + \frac{Y_{i,j+2}^n - Y_{i,j}^n}{\Delta y} + 2S_{i,j+1}^n\right) \end{cases} \tag{29}$$

$$U_{i,j}^{n+1} = U_{i,j}^n - \Delta t \left(\frac{X_{i+1,j}^{n+\frac{1}{2}} - X_{i-1,j}^{n+\frac{1}{2}}}{\Delta x} + \frac{Y_{i,j+1}^{n+\frac{1}{2}} - Y_{i,j-1}^{n+\frac{1}{2}}}{\Delta y} + S_{i,j}^n \right) \tag{30}$$

$$X_{i,j}^n = X(U_{i,j}^n), X_{i,j}^{n+\frac{1}{2}} = X(U_{i,j}^{n+\frac{1}{2}})$$

$$Y_{i,j}^n = Y(U_{i,j}^n), Y_{i,j}^{n+\frac{1}{2}} = Y(U_{i,j}^{n+\frac{1}{2}})$$

$$S_{i,j}^n = S(U_{i,j}^n)$$

На рисунке красным кружком является не только искомым узлом на следующем временном шаге, но и опорным для промежуточных расчётом первой ступени, синие кружки выступают опорными узлами для той же ступени. Серые кружки - опорные для финального шага. Заметим, что для данной схемы толщина "рамки" дополнительных узлов (ghost points) вокруг сетки расчётной области составляет уже два значения.

3.3 Исследование схемы Рихтмаейра на устойчивость

Рассмотрим схему Рихтмайера для линейного одномерного дифференциального уравнения:

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

$$\begin{cases} u_{j+1}^{n+\frac{1}{2}} = \frac{1}{2}(u_{j+2}^n + u_j^n) - \frac{a\Delta t}{2\Delta x}(u_{j+2}^n - u_j^n) \\ u_{j-1}^{n+\frac{1}{2}} = \frac{1}{2}(u_j^n + u_{j-2}^n) - \frac{a\Delta t}{2\Delta x}(u_j^n - u_{j-2}^n) \end{cases}$$

$$u_j^{n+1} = u_j^n - \frac{a\Delta t}{\Delta x}(u_{j+1}^{n+\frac{1}{2}} - u_{j-1}^{n+\frac{1}{2}})$$

Соберём обе ступени в одно уравнение:

$$u_j^{n+1} = u_j^n - \frac{a\Delta t}{\Delta x}\left(\frac{1}{2}(u_{j+2}^n - u_{j-2}^n) - \frac{a\Delta t}{2\Delta x}(u_{j+2}^n - 2u_j^n + u_{j-2}^n)\right)$$

Произведём преобразования Фурье величины u_j^n во всех узлах и разделим уравнение на $e^{at}e^{ik_m x}$:

$$e^{a\Delta t} = 1 - \frac{a\Delta t}{\Delta x}\left(\frac{1}{2}(e^{2ik_m\Delta x} - e^{-2ik_m\Delta x}) - \frac{a\Delta t}{2\Delta x}(e^{2ik_m\Delta x} - 2 + e^{-2ik_m\Delta x})\right)$$

$$e^{a\Delta t} = 1 - \frac{a\Delta t}{\Delta x}\left(i \cdot sin(2k_m\Delta x) + \frac{2a\Delta t}{\Delta x}sin^2(k_m\Delta x)\right)$$

$$|e^{a\Delta t}| = \sqrt{\left(1 - \frac{2a^2\Delta t^2}{\Delta x^2}sin^2(k_m\Delta x)\right)^2 + \frac{a^2\Delta t^2}{\Delta x^2}sin^2(2k_m\Delta x)} \leq 1$$

Снимем корень и избавимся от единицы:

$$\frac{4a^2\Delta t^2}{\Delta x^2}sin^4(k_m\Delta x) - 4sin^2(k_m\Delta x) + sin^2(2k_m\Delta x) \leq 0$$

$$\frac{4a^2\Delta t^2}{\Delta x^2}sin^4(k_m\Delta x) - 4sin^2(k_m\Delta x) + 4sin^2(k_m\Delta x)(1 - sin^2(k_m\Delta x)) \leq 0$$

$$\frac{4a^2\Delta t^2}{\Delta x^2}sin^4(k_m\Delta x) \leq 4sin^4(k_m\Delta x)$$

$$\frac{a^2\Delta t^2}{\Delta x^2} \leq 1$$

$$\Delta t \leq \frac{\Delta x}{|a|}$$

Однако такая вязкость соблюдается только в одномерном случае [10, 11]. С ростом размерности пространства p значение максимального шага по времени падает как

$$\Delta t \leq \frac{\Delta x}{|a|}\frac{1}{\sqrt{p}}$$

3.4 Реализация схемы Рихтмайера на языке C++

```
template <class T>
T firstStep(const T& tOffsetMinusX, const T& tOffsetPlusX,
            const T& tOffsetMinusY, const T& tOffsetPlusY) {
    return (tOffsetPlusX + tOffsetMinusX + tOffsetPlusY + tOffsetMinusY) / 4 -
           ((funcX(tOffsetPlusX) - funcX(tOffsetMinusX)) * mRatioX +
           (funcY(tOffsetPlusY) - funcY(tOffsetMinusY)) * mRatioY) / 2;
}

template <class T>
T solve(const T& tOffsetZero,
        const T& tX_min_2_Y,        const T& tX_pl_2_Y,
        const T& tX_Y_min_2,        const T& tX_Y_pl_2,
        const T& tX_pl_1_Y_pl_1,    const T& tX_min_1_Y_min_1,
        const T& tX_pl_1_Y_min_1,   const T& tX_min_1_Y_pl_1) {
    T HalfMinusX = firstStep(tX_min_2_Y, tOffsetZero, tX_min_1_Y_min_1, tX_min_1_Y_pl_1);
    T HalfPlusX = firstStep(tOffsetZero, tX_pl_2_Y, tX_pl_1_Y_min_1, tX_pl_1_Y_pl_1);
    T HalfMinusY = firstStep(tX_min_1_Y_min_1, tX_pl_1_Y_min_1, tX_Y_min_2, tOffsetZero);
    T HalfPlusY = firstStep(tX_min_1_Y_pl_1, tX_pl_1_Y_pl_1, tOffsetZero, tX_Y_pl_2);

    T Solution =    tOffsetZero -
                    (funcX(HalfPlusX) - funcX(HalfMinusX)) * mRatioX -
                    (funcY(HalfPlusY) - funcY(HalfMinusY)) * mRatioY;

    return Solution;
}
```

Листинг 2: Richmyer scheme 2D

3.5 Схема Рихтмайера-Цваса

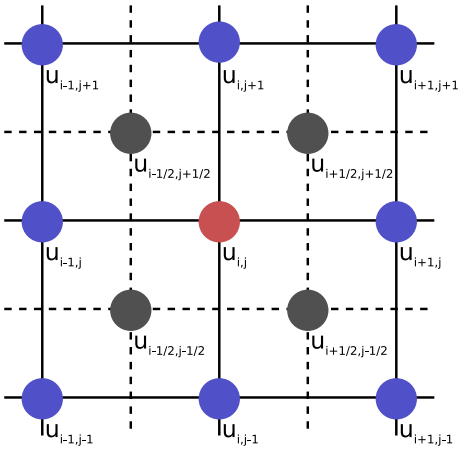

Рис. 3: Схема Рихтмайера-Цваса

Данная схема, как и предыдущая, использует 9 опорных узлов, однако более компактна с точки зрения расположения на расчётной области. Её главный недостаток в том, что приходится считать значения между узлами сетки, но это небольшая плата за бóльшую устойчивость, о чём будет сказано в следующем разделе. Для нашей системы схема примет вид

$$\begin{cases} U^{n+\frac{1}{2}}_{i-\frac{1}{2},j+\frac{1}{2}} = \dfrac{U^n_{i,j}+U^n_{i,j+1}+U^n_{i-1,j+1}+U^n_{i-1,j}}{4} - \dfrac{\Delta t}{2}\left(\dfrac{\tilde{X}^n_{i,j+\frac{1}{2}}-\tilde{X}^n_{i-1,j+\frac{1}{2}}}{\Delta x}+\dfrac{\tilde{Y}^n_{i-\frac{1}{2},j+1}-\tilde{Y}^n_{i-\frac{1}{2},j}}{\Delta y}+2\tilde{S}^n_{i-\frac{1}{2},j+\frac{1}{2}}\right) \\ U^{n+\frac{1}{2}}_{i+\frac{1}{2},j+\frac{1}{2}} = \dfrac{U^n_{i,j}+U^n_{i+1,j}+U^n_{i+1,j+1}+U^n_{i,j+1}}{4} - \dfrac{\Delta t}{2}\left(\dfrac{\tilde{X}^n_{i+1,j+\frac{1}{2}}-\tilde{X}^n_{i,j+\frac{1}{2}}}{\Delta x}+\dfrac{\tilde{Y}^n_{i+\frac{1}{2},j+1}-\tilde{Y}^n_{i+\frac{1}{2},j}}{\Delta y}+2\tilde{S}^n_{i+\frac{1}{2},j+\frac{1}{2}}\right) \\ U^{n+\frac{1}{2}}_{i-\frac{1}{2},j-\frac{1}{2}} = \dfrac{U^n_{i,j}+U^n_{i-1,j}+U^n_{i-1,j-1}+U^n_{i,j-1}}{4} - \dfrac{\Delta t}{2}\left(\dfrac{\tilde{X}^n_{i,j-\frac{1}{2}}-\tilde{X}^n_{i-1,j-\frac{1}{2}}}{\Delta x}+\dfrac{\tilde{Y}^n_{i-\frac{1}{2},j}-\tilde{Y}^n_{i-\frac{1}{2},j-1}}{\Delta y}+2\tilde{S}^n_{i-\frac{1}{2},j-\frac{1}{2}}\right) \\ U^{n+\frac{1}{2}}_{i+\frac{1}{2},j-\frac{1}{2}} = \dfrac{U^n_{i,j}+U^n_{i,j-1}+U^n_{i+1,j-1}+U^n_{i+1,j}}{4} - \dfrac{\Delta t}{2}\left(\dfrac{\tilde{X}^n_{i+1,j-\frac{1}{2}}-\tilde{X}^n_{i,j-\frac{1}{2}}}{\Delta x}+\dfrac{\tilde{Y}^n_{i+\frac{1}{2},j}-\tilde{Y}^n_{i+\frac{1}{2},j-1}}{\Delta y}+2\tilde{S}^n_{i+\frac{1}{2},j-\frac{1}{2}}\right) \end{cases} \tag{31}$$

$$U^{n+1}_{i,j} = U^n_{i,j} - \Delta t\left(\frac{\tilde{X}^{n+\frac{1}{2}}_{i+\frac{1}{2},j}-\tilde{X}^{n+\frac{1}{2}}_{i-\frac{1}{2},j}}{\Delta x}+\frac{\tilde{Y}^{n+\frac{1}{2}}_{i,j+\frac{1}{2}}-\tilde{Y}^{n+\frac{1}{2}}_{i,j-\frac{1}{2}}}{\Delta y}+2S^n_{i,j}\right) \tag{32}$$

$$\tilde{X}^n_{i,j+\frac{1}{2}} = X\left(\frac{U^n_{i,j+1}-U^n_{i,j}}{2}\right), \tilde{X}^{n+\frac{1}{2}}_{i+\frac{1}{2},j} = X\left(\frac{U^{n+\frac{1}{2}}_{i+\frac{1}{2},j+\frac{1}{2}}-U^{n+\frac{1}{2}}_{i+\frac{1}{2},j-\frac{1}{2}}}{2}\right)$$

$$\tilde{Y}^n_{i+\frac{1}{2},j} = Y\left(\frac{U^n_{i+1,j}-U^n_{i,j}}{2}\right), \tilde{Y}^{n+\frac{1}{2}}_{i,j+\frac{1}{2}} = \left(\frac{U^{n+\frac{1}{2}}_{i+\frac{1}{2},j+\frac{1}{2}}-U^{n+\frac{1}{2}}_{i-\frac{1}{2},j+\frac{1}{2}}}{2}\right)$$

$$\tilde{S}^n_{i+\frac{1}{2},j+\frac{1}{2}} = \frac{S^n_{i,j}+S^n_{i+1,j}+S^n_{i+1,j+1}+S^n_{i,j+1}}{4}$$

3.6 Исследование схемы Рихтмаейра-Цваса на устойчивость

Запишем схему Рихтмайера-Цваса для для уравнения

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

$$\begin{cases} u^{n+\frac{1}{2}}_{j+\frac{1}{2}} = \dfrac{1}{2}(u^n_{j+1}+u^n_j) - \dfrac{a\Delta t}{2\Delta x}(u^n_{j+1}-u^n_j) \\ u^{n+\frac{1}{2}}_{j-\frac{1}{2}} = \dfrac{1}{2}(u^n_j+u^n_{j-1}) - \dfrac{a\Delta t}{2\Delta x}(u^n_j-u^n_{j-1}) \end{cases}$$

$$u^{n+1}_j = u^n_j - \frac{a\Delta t}{\Delta x}(u^{n+\frac{1}{2}}_{j+\frac{1}{2}} - u^{n+\frac{1}{2}}_{j-\frac{1}{2}})$$

Как можно заметить, в одномерном случае данная схема отличается от схемы Рихтмайера только "размахом" опорных узлов, поэтому исследование на устойчивость становится очевидным:

$$u^{n+1}_j = u^n_j - \frac{a\Delta t}{\Delta x}\left(\frac{1}{2}(u^n_{j+1}-u^n_{j-1}) - \frac{a\Delta t}{2\Delta x}(u^n_{j+1}-2u^n_j+u^n_{j-1})\right)$$

$$e^{a\Delta t} = 1 - \frac{a\Delta t}{\Delta x}\left(\frac{1}{2}(e^{ik_m\Delta x}-e^{-ik_m\Delta x}) - \frac{a\Delta t}{2\Delta x}(e^{ik_m\Delta x}-2+e^{-ik_m\Delta x})\right)$$

$$e^{a\Delta t} = 1 - \frac{a\Delta t}{\Delta x}\left(i\cdot sin(k_m\Delta x) + \frac{2a\Delta t}{\Delta x}sin^2\left(\frac{k_m\Delta x}{2}\right)\right)$$

$$|e^{a\Delta t}| = \sqrt{\left(1-\frac{2a^2\Delta t^2}{\Delta x^2}sin^2\left(\frac{k_m\Delta x}{2}\right)\right)^2 + \frac{a^2\Delta t^2}{\Delta x^2}sin^2(k_m\Delta x)} \le 1$$

$$\frac{4a^2\Delta t^2}{\Delta x^2}sin^4\left(\frac{k_m\Delta x}{2}\right) - 4sin^2\left(\frac{k_m\Delta x}{2}\right) + sin^2(k_m\Delta x) \le 0$$

$$\frac{4a^2\Delta t^2}{\Delta x^2}sin^4\left(\frac{k_m\Delta x}{2}\right) - 4sin^2\left(\frac{k_m\Delta x}{2}\right) + 4sin^2\left(\frac{k_m\Delta x}{2}\right)\left(1-sin^2\left(\frac{k_m\Delta x}{2}\right)\right) \le 0$$

$$\frac{4a^2\Delta t^2}{\Delta x^2} sin^4\left(\frac{k_m \Delta x}{2}\right) \leq 4sin^4\left(\frac{k_m \Delta x}{2}\right)$$

$$\frac{a^2\Delta t^2}{\Delta x^2} \leq 1$$

$$\Delta t \leq \frac{\Delta x}{|a|}$$

Отличие этой схемы от предыдущей не только в "размахе" опорных узлов, но и в отсутствии зависимости необходимого условия устойчивости от размерности пространства p, то есть данный вариант схемы допускает максимальный шаг по времени в $\sqrt{p}$ больше, чем предыдущий [10, 12].

3.7 Реализация Рихтмайера-Цваса на языке C++ схемы

```
template <class T>
T firstStepZwas(const T& tBottomRight,  const T& tTopRight,
                const T& tTopLeft,      const T& tBottomLeft) {
    return  (tBottomRight + tTopRight + tTopLeft + tBottomLeft) / 4 -
            ((funcX((tBottomRight + tTopRight) / 2) - funcX((tBottomLeft + tTopLeft) / 2)) *
            mRatioX +
            (funcY((tTopRight + tTopLeft) / 2) - funcY((tBottomRight + tBottomLeft) / 2)) *
            mRatioY) / 2;
}

template <class T>
T solveZwas(const T& tOffsetZero,
            const T& tX_min_1_Y,        const T& tX_pl_1_Y,
            const T& tX_Y_min_1,        const T& tX_Y_pl_1,
            const T& tX_pl_1_Y_pl_1,    const T& tX_min_1_Y_min_1,
            const T& tX_pl_1_Y_min_1,   const T& tX_min_1_Y_pl_1) {
    T Half_X_min_Y_min = firstStepZwas(tX_Y_min_1, tOffsetZero, tX_min_1_Y, tX_min_1_Y_min_1)
;
    T Half_X_pl_Y_min = firstStepZwas(tX_pl_1_Y_min_1, tX_pl_1_Y, tOffsetZero, tX_Y_min_1);
    T Half_X_min_Y_pl = firstStepZwas(tOffsetZero, tX_Y_pl_1, tX_min_1_Y_pl_1, tX_min_1_Y);
    T Half_X_pl_Y_pl = firstStepZwas(tX_pl_1_Y, tX_pl_1_Y_pl_1, tX_Y_pl_1, tOffsetZero);

    T GradX =   (funcX(Half_X_pl_Y_pl) - funcX(Half_X_min_Y_pl) +
                funcX(Half_X_pl_Y_min) - funcX(Half_X_min_Y_min)) / 2;
    T GradY =   (funcY(Half_X_pl_Y_pl) - funcY(Half_X_pl_Y_min) +
                funcY(Half_X_min_Y_pl) - funcY(Half_X_min_Y_min)) / 2;

    return tOffsetZero - GradX * mRatioX - GradY * mRatioY;
}
```

Листинг 3: Richmyer-Zwas scheme 2D

3.8 Результаты

Temp

3.9 Искусственная вязкость

Как было описано раннее, в разделе 3.1, схемы второго и выше порядков имеют очень серьёзный недостаток - колебания в областях разрывов производных, что может приводить к неконтролируемому увеличению модуля собственных значений матрицы усиления и последующей расходимости схемы. Для решения этой проблемы существуют несколько подходов: использование неосциллирующих схем, про которые будет идти речь в разделе 4, и введение искусственной вязкости [13, 14, 15].

Искусственную вязкость можно ввести в виде добавки к потоку [16]:

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

$$X' = X - \nu \frac{\partial u}{\partial x}$$
$$\nu = c_\nu \Delta x^2 \left| \frac{\partial u}{\partial x} \right|,$$

где c_ν - коэффициент вязкости. Тогда уравнение с вязкостью приобретает вид

$$\frac{\partial u}{\partial t} + \frac{\partial X}{\partial x} = \frac{\partial}{\partial x} \left(c_\nu \Delta x^2 \left| \frac{\partial u}{\partial x} \right| \frac{\partial u}{\partial x} \right)$$

Аппроксимируем правую часть:

$$\frac{\partial}{\partial x} \left(c_\nu \Delta x^2 \left| \frac{\partial u}{\partial x} \right| \frac{\partial u}{\partial x} \right) \approx$$
$$\approx \frac{\Delta t}{\Delta x} \left(\left(c_\nu \Delta x^2 \left| \frac{\partial u}{\partial x} \right| \frac{\partial u}{\partial x} \right)_{j+\frac{1}{2}} - \left(c_\nu \Delta x^2 \left| \frac{\partial u}{\partial x} \right| \frac{\partial u}{\partial x} \right)_{j-\frac{1}{2}} \right) =$$
$$= c_\nu \frac{\Delta t}{\Delta x} \left(\left| u^n_{j+1} - u^n_j \right| \left(u^n_{j+1} - u^n_j \right) - \left| u^n_j - u^n_{j-1} \right| \left(u^n_j - u^n_{j-1} \right) \right)$$

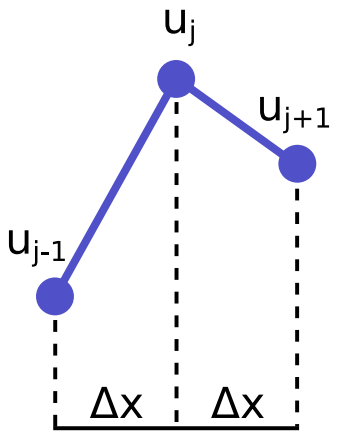

Рис. 4: Искусственная вязкость фон Неймана - Рихтмайера

Из конечной формулы и рисунка видно, что выражение в скобках выступает в роли параметра сглаживания: чем меньше разброс подряд идущих значений, тем меньше сглаживание и, следовательно, величина искусственной вязкости.

3.10 Результаты с вязкостью

Temp

4 Неосциллирующие схемы

4.1 TVD схемы

TVD (Total Variation Diminishing) схемы [17] - схемы, удовлетворяющие неравенству

$$TV(u^{n+1}) \leq TV(u^n),$$

где

$$TV(u(t)) = \int \left| \frac{\partial u}{\partial x} \right| dx$$

или в дискретном случае

$$TV(u^n) = \sum_j \left| u^n_{j+1} - u^n_j \right|$$

Рассмотрим явную схему:

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

$$u^{n+1}_j = u^n_j - \frac{\Delta t}{\Delta x} \left(X^n_{j+\frac{1}{2}} - X^n_{j-\frac{1}{2}} \right)$$

В общем случае для схем первого порядка поток, например, $X^n_{j+\frac{1}{2}}$, можно записать в виде

$$X^n_{j+\frac{1}{2}} = \frac{1}{2} \left(X^n_{j+1} + X^n_j - \psi \left(a^n_{j+\frac{1}{2}} \right) \left(u^n_{j+1} - u^n_j \right) \right),$$

где $a^n_{j+\frac{1}{2}}$ - аргумент, зависящий от собственных значений оператора X, а ψ - функция, отвечающая за численную вязкость, вводится в статье [18] Хартеном как

$$\psi(y) = \begin{cases} \dfrac{y^2}{4\epsilon} + \epsilon, & |y| < 2\epsilon \\ |y|, & |y| \geq 2\epsilon \end{cases},$$

где величина ϵ лежит в пределах $\left(0, \frac{1}{2}\right]$

Определив поток, для построения TVD схемы мы можем пойти двумя путями [19].

- Первый подход заключается в замене потока на его сумму с некоторой функцией $\tilde{X}$, чтобы получившийся поток был второго порядка аппроксимации по отношению к X. Тогда получим

$$X^n_{j+\frac{1}{2}} = \frac{1}{2} \left(X^n_{j+1} + X^n_j + \tilde{X}^n_{j+1} + \tilde{X}^n_j - \psi \left(a^n_{j+\frac{1}{2}} + \tilde{a}^n_{j+\frac{1}{2}} \right) \left(u^n_{j+1} - u^n_j \right) \right),$$

  где $a^n_{j+\frac{1}{2}}$ и $\tilde{a}^n_{j+\frac{1}{2}}$ принимают значения

$$a^n_{j+\frac{1}{2}} = \begin{cases} \dfrac{X^n_{j+1} - X^n_j}{u^n_{j+1} - u^n_j}, & u^n_{j+1} \neq u^n_j \\ \left(\dfrac{dX}{du} \right)_j, & u^n_{j+1} = u^n_j \end{cases}$$

$$\tilde{a}^n_{j+\frac{1}{2}} = \begin{cases} \dfrac{\tilde{X}^n_{j+1} - \tilde{X}^n_j}{u^n_{j+1} - u^n_j}, & u^n_{j+1} \neq u^n_j \\ 0, & u^n_{j+1} = u^n_j \end{cases}$$

- Второй, наиболее распространённый, путь состоит в добавлении к выражению для потока антидиффузионного члена, помноженного на некоторую функцию-ограничитель ϕ, что позволяет контролировать область устойчивости полученной схемы. Для простоты рассмотрим линейный поток:

$$X(u) = au,\ a = const,\ a > 0$$

$$X^n_{j+\frac{1}{2}} = \frac{1}{2} \left(X^n_{j+1} + X^n_j - a \left(u^n_{j+1} - u^n_j \right) \right) + \frac{a}{2} \left(u^n_j - u^n_{j-1} \right) \phi_j$$

Подставим полученный поток в исходную схему:

$$\theta_j = \frac{u_j^n - u_{j-1}^n}{u_{j+1}^n - u_j^n}, \quad \tilde{\theta}_j = \frac{u_{j+1}^n - u_j^n}{u_j^n - u_{j-1}^n}$$

$$u_j^{n+1} = u_j^n - a\frac{\Delta t}{\Delta x}\left(1 + \frac{1}{2}\phi_j - \frac{1}{2}\theta_{j-1}\phi_{j-1}\right)\left(u_j^n - u_{j-1}^n\right)$$

Существует множество различных ограничителей, среди которых наиболее известные

Minmod:

$$\phi_j = minmod\left(1, \tilde{\theta}_j\right)$$

Superbee:

$$\phi_j = max\left(0, min\left(1, 2\tilde{\theta}_j\right), min\left(2, \tilde{\theta}_j\right)\right)$$

Ван Лир:

$$\phi_j = \frac{\tilde{\theta}_j + \left|\tilde{\theta}_j\right|}{1 + \tilde{\theta}_j}$$

В данной работе мы не будем заниматься построением TVD схем, а возьмём на вооружение готовые TVD варианты схем класса Рунге-Кутта, которые обычно используют для интегрирования по времени. Нас интересует второй порядок аппроксимации. Пусть $L(u)$ - "пространственная часть" уравнения, которую мы аппроксимируем тем или иным способом, тогда "временную часть" можно аппроксимировать следующим образом:

$$\begin{cases} u_j^{(1)} = u_j^n - \Delta t L\left(u_j^n\right) \\ u_j^{n+1} = \frac{1}{2}u_j^n - \Delta t L\left(u_j^n\right) - \Delta t L\left(u_j^{(1)}\right) \end{cases}$$

Построение Рунге-Кутта схем разных порядков, а также их TVD-вариантов можно найти в книге [20]. Применение TVD схем к уравнениям мелкой воды описано в статье [21].

4.2 Схема ENO

Essentially Non-oscillatory (ENO) схема, наряду с TVD схемами, используется для предотвращения колебаний решения вблизи разрывов производных. В основе данной схемы лежит выбор серии опорных узлов, минимально отличающихся между собой по значениям, то есть поиск максимально гладкого участка.

Определим ячейки на расчётной сетке:

$$I_i = \left[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}\right]$$

Пусть $p_i(x)$ - степенная функция, аппроксимирующая искомую в ячейке I_i, тогда на границах ячейки

$$v_{i+\frac{1}{2}}^- = p_i\left(x_{i+\frac{1}{2}}\right), \qquad v_{i-\frac{1}{2}}^+ = p_i\left(x_{i-\frac{1}{2}}\right)$$

Если k - выбранный нами порядок схемы, тогда значение в узле i на следующем временном шаге будет аппроксимироваться узлом i на данном шаге, r узлами слева от i и s узлами справа, причём должно быть выполнено соотношение

$$r + s + 1 = k,$$

тогда на границах аппроксимации приобретают вид

$$v_{i+\frac{1}{2}}^- = \sum_{j=0}^{k-1} c_{rj} v_{i-r+j}, \qquad v_{i-\frac{1}{2}}^+ = \sum_{j=0}^{k-1} c_{r-1,j} v_{i-r+j}$$

$$c_{rj} = \sum_{m=j+1}^{k} \frac{\sum\limits_{\substack{l=0 \\ l\neq m}}^{k} \prod\limits_{\substack{q=0 \\ q\neq m,l}}^{k} (r-q+1)}{\prod\limits_{\substack{l=0 \\ l\neq m}}^{k} (m-l)}$$

Стоит отметить, что в общем случае под знаком суммы стоит усреднённое по ячейке значение функции на данном временном шаге $\overline{v}_{i-r+j}$, однако в случае равномерной сетки средние значения совпадают со значениями в узлах. Выражение для коэффициентов c_{rj} также записано специально для нашей задачи. Более общий подход и значения коэффициентов для разных порядков можно найти в статье [22].

Рассмотрим схему ENO 3-го порядка. Запишем все возможные коэффициенты c_{rj} для допустимых серий опорных узлов:

r	j = 0	j = 1	j = 2
-1	11/6	-7/6	1/3
0	1/3	5/6	-1/6
1	-1/6	5/6	1/3
2	1/3	-7/6	11/6

Теперь предположим, что мы решаем задачу с разрывом производной между узлами u_j и u_{j+1}:

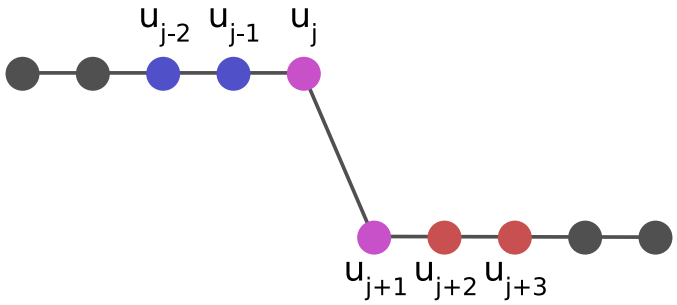

Рис. 5: Схема ENO

Подбор подходящей серии при расчёте узлов u_j^{n+1} и u_{j+1}^{n+1} заключается в том, чтобы из этой серии убрать разрыв, то есть синие узлы и u_j^n используются для расчёта u_j^{n+1}, а красные и u_{j+1}^n - для u_{j+1}^{n+1}. Чтобы автоматизировать данный процесс выбора, вводится разделённая разность Ньютона

$$V(x) = \int_{-\infty}^{x} v(\xi)d\xi$$

$$V\left[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}\right] = \frac{V\left(x_{i+\frac{1}{2}}\right) - V\left(x_{i-\frac{1}{2}}\right)}{x_{i+\frac{1}{2}} - x_{i-\frac{1}{2}}} = v_i,$$

обладающая на однородной сетке следующим свойством:

$$V\left[x_{i-\frac{1}{2}}, ..., x_{i+j+\frac{1}{2}}\right] = V\left[x_{i+\frac{1}{2}}, ..., x_{i+j+\frac{1}{2}}\right] - V\left[x_{i-\frac{1}{2}}, ..., x_{i+j-\frac{1}{2}}\right],\ j \geq 1$$

Например, для интересующего нас случая схемы 3-го порядка возможны три разные серии из трёх узлов, поэтому запишем

$$\begin{cases} V\left[x_{i-\frac{3}{2}}, x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}\right] = v_i - v_{i-1} \\ V\left[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}, x_{i+\frac{3}{2}}\right] = v_{i+1} - v_i \\ V\left[x_{i+\frac{1}{2}}, x_{i+\frac{3}{2}}, x_{i+\frac{5}{2}}\right] = v_{i+2} - v_{i+1} \end{cases}$$

Теперь опишем алгоритм выбора серии:

- Выберем ячейку, содержащую v_i, то есть $\left[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}\right]$.

- Произведём сравнение $\left|V\left[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}\right]\right|$ и $\left|V\left[x_{i+\frac{1}{2}}, x_{i+\frac{3}{2}}\right]\right|$.
- Если $\left|V\left[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}\right]\right| \leq \left|V\left[x_{i+\frac{1}{2}}, x_{i+\frac{3}{2}}\right]\right|$, то добавим точку слева и будем сравнивать $\left|V\left[x_{i-\frac{3}{2}}, x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}\right]\right|$ и $\left|V\left[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}, x_{i+\frac{3}{2}}\right]\right|$. В противном случае добавляем точку справа и сравниваем $\left|V\left[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}, x_{i+\frac{3}{2}}\right]\right|$ и $\left|V\left[x_{i+\frac{1}{2}}, x_{i+\frac{3}{2}}, x_{i+\frac{5}{2}}\right]\right|$.
- В итоге выбираем серию, для которой модуль функции V меньше.

Поскольку для неосциллирующих схем необходима монотонность потока, производят его разделение

$$X(u) = X^+(u) + X^-(u)$$

таким образом, чтобы выполнялись условия

$$\frac{dX^+(u)}{du} \geq 0, \ \frac{dX^-(u)}{du} \leq 0$$

Наиболее распространённым является разделение Лакса-Фридрихса:

$$X^{\pm}(u) = \frac{1}{2}(X(u) \pm \alpha u),$$

$$\alpha = \max_u \max_{1 \leq j \leq m} |\lambda_j(u)|$$

где $\lambda(u)$ - собственные значения оператора X. Тогда для аппроксимации потока примем

$$v_i = X^+(u_i), \qquad X^+_{i+\frac{1}{2}} = v^-_{i+\frac{1}{2}}$$

и посчитаем $X^+_{i+\frac{1}{2}}$, затем аналогично примем

$$v_i = X^-(u_i), \qquad X^-_{i+\frac{1}{2}} = v^+_{i+\frac{1}{2}}.$$

Посчитав $X^-_{i+\frac{1}{2}}$, окончательно запишем

$$X_{i+\frac{1}{2}} = X^+_{i+\frac{1}{2}} + X^-_{i+\frac{1}{2}}$$

Чтобы получить аппроксимацию производной по пространству, повторим действия для $X_{i+\frac{1}{2}}$ и возьмём разность

$$\frac{\partial X}{\partial x} \approx \frac{X_{i+\frac{1}{2}} - X_{i-\frac{1}{2}}}{\Delta x}$$

4.3 Схема WENO

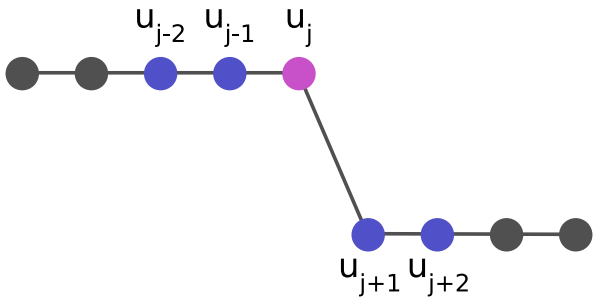

Рис. 6: Схема WENO

Общие принципы, параметр сглаживания, точность по сравнению со схемой ENO, использование в паре со схемой Рунге-Кутта

4.4 Результаты

5 Дивергенция магнитного поля

5.1 Проблема

5.2 Методы решения

5.3 GLM метод

5.4 Метод проекции

5.5 Методы решения уравнения Пуассона

5.5.1 Метод градиентного спуска

5.5.2 Метод сопряжённых градиентов

6 Оптимизация вычислений

6.1 Конечно-разностные схемы

6.1.1 Распараллеливание

6.1.2 Баланс вычислительных мощностей

6.2 Итеративные методы

6.2.1 Предобуславливание

6.2.2 Оптимизация операций с матрицами

6.2.3 Предсказание решения

Список литературы

[1] А. С. Петросян Д. А. Климачков. “Волны Россби в магнитной гидродинамике вращающейся плазмы в приближении мелкой воды”. В: *ЖЭТФ* том 152 (4 2017), с. 705—721. DOI: 10.7868/S004445101710008X.

[2] D.A. Klimachkov и A.S. Petrosyan. “Parametric instabilities in shallow water magnetohydrodynamics of astrophysical plasma in external magnetic field”. В: *Physics Letters A* 381.2 (2017), с. 106—113. ISSN: 0375-9601. DOI: https://doi.org/10.1016/j.physleta.2016.10.011.

[3] Robin Hogan. *Computer Practical: Shallow Water Model*. Февр. 2014. URL: http://www.met.reading.ac.uk/~swrhgnrj/shallow_water_model/swe_notes.pdf.

[4] Luciano Rezzolla. *Finite-difference Methods for the Solution of Partial Differential Equations*. Institute for Theoretical Physics, Frankfurt, Germany, окт. 2018.

[5] В. М. Ковеня и Д. В. Чирков. *Методы конечных разностей и конечных объёмов для решения задач математической физики. Введение в теорию*. НГУ, 2013.

[6] Jan Hesthaven и Tim Warburton. “Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications”. В: т. 54. Янв. 2007.

[7] *Magnetic Field Calculators*. National Oceanic и Atmospheric Administration. URL: https://www.ngdc.noaa.gov/geomag/calculators/magcalc.shtml#igrfgrid.

[8] Shuonan Dong. *Finite Difference Methods for the Hyperbolic Wave Partial Differential Equations*. 2006.

[9] Robert Richtmyer. “A Survey of Difference Methods for Non-Steady Fluid Dynamics”. В: *NCAR Technical Notes* (63 янв. 1962). DOI: 10.5065/D67P8WCQ.

[10] Joseph E. Flaherty. *Partial Differential Equations*. Rensselaer Polytechnic Institute, United States.

[11] J. C. WILSON. “Stability of Richtmyer Type Difference Schemes in any Finite Number of Space Variables and Their Comparison with Multistep Strang Schemes”. В: *IMA Journal of Applied Mathematics* 10.2 (окт. 1972), с. 238—257. DOI: 10.1093/imamat/10.2.238.

[12] Gideon Zwas. “On Two Step Lax-Wendroff Methods in Several Dimensions”. В: *Numer. Math.* 20.5 (окт. 1972), с. 350—355. DOI: 10.1007/BF01402557.

[13] R. D. Richtmyer J. Von Neumann. “A Method for the Numerical Calculation of Hydrodynamic Shocks”. В: *Journal of Applied Physics* 21 (3 март 1950), с. 232—237. DOI: 10.1063/1.1699639.

[14] James Campbell и Rade Vignjevic. “Artificial Viscosity Methods forModelling Shock Wave Propagation”. В: *Predictive Modeling of Dynamic Processes: A Tribute to Professor Klaus Thoma* (июнь 2009), с. 349—365. DOI: 10.1007/978-1-4419-0727-1_19.

[15] Jiequan Li и др. “Local oscillations in finite difference solutions of hyperbolic conservation laws”. В: *Math. Comput.* 78 (окт. 2009), с. 1997—2018. DOI: 10.1090/S0025-5718-09-02219-4.

[16] Gretar Tryggvason. *Computational Fluid Dynamics. Advection*. The University of Notre Dame. Февр. 2011. URL: https://www3.nd.edu/~gtryggva/CFD-Course/.

[17] Ju Y. Yi. “Definition and Construction of Entropy Satisfying Multiresolution Analysis (MRA)”. Дис. ... док. Utah State University, 2016. eprint: AllGraduateThesesandDissertations. URL: https://digitalcommons.usu.edu/etd/5057.

[18] Ami Harten. “High resolution schemes for hyperbolic conservation laws”. В: *Journal of Computational Physics* 49.3 (1983), с. 357—393. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(83)90136-5. URL: http://www.sciencedirect.com/science/article/pii/0021999183901365.

[19] Д.В. Чирков и С.Г. Черный. “Сравнение точности и сходимости некоторых TVD-схем”. В: *Вычислительные технологии* 5.5 (2000), с. 86—107.

[20] “Time Integration Methods”. В: *High-Resolution Methods for Incompressible and Low-Speed Flows*. Springer Berlin Heidelberg, 2005, с. 99—119. ISBN: 978-3-540-26454-5. DOI: 10.1007/3-540-26454-X_7.

[21] M. Louaked и L. Hanich. “TVD scheme for the shallow water equations”. В: *Journal of Hydraulic Research* 36.3 (1998), с. 363—378. DOI: 10.1080/00221689809498624.

[22] Chi-Wang Shu и др. “Essentially non-oscillatory and weighted essentially non-oscillatory schemes for hyperbolic conservation laws”. В: (нояб. 2006), с. 325—432. DOI: 10.1007/BFb0096355.

A C++ класс для решения задачи методом Лакса-Фридрихса

```
class dLaxFriedrichsSolver2D : public dLaxFriedrichs2D <dVector <double, 5>> {
public:
dLaxFriedrichsSolver2D(double TimeStepP, double xStepP, double yStepP) {
mStepTime = TimeStepP;
mStepX = xStepP;
mStepY = yStepP;

//        Homogeneous = false;
}
~dLaxFriedrichsSolver2D() = default;

void SetInitialData(unsigned long xSizeP, unsigned long ySizeP) {
xSize = xSizeP;
ySize = ySizeP;

DataFirst.resize(xSizeP);
DataSecond.resize(xSizeP);

for (unsigned long i = 0; i < xSizeP; i++) {
for (unsigned long j = 0; j < ySizeP; j++) {
DataFirst[i].emplace\_back(dVector <double, 5> (10.0, 0.0, 0.0, 0.0, 0.0));
DataSecond[i].emplace\_back(dVector <double, 5> (10.0, 0.0, 0.0, 0.0, 0.0));
}
}

SetExcitation();

for (unsigned long i = 0; i < ySizeP; i++) {
Gradient.emplace\_back(0.1 / ySizeP * i);
}

AmplitudeFile.open(SavePath + "Amplitude.dat");
xVelocityFile.open(SavePath + "xVelocity.dat");
yVelocityFile.open(SavePath + "yVelocity.dat");
xFieldFile.open(SavePath + "xField.dat");
yFieldFile.open(SavePath + "yField.dat");
}
void SetSavePath(const std::string\& PathP) {
SavePath = PathP;
}
void AppendData() {
for (int j = 0; j < ySize; j++) {
for (int i = 0; i < xSize; i++) {
AmplitudeFile << (*CurrentData)[i][j][0] << "\t";
xVelocityFile << (*CurrentData)[i][j][1] << "\t";
yVelocityFile << (*CurrentData)[i][j][2] << "\t";
xFieldFile << (*CurrentData)[i][j][3] << "\t";
yFieldFile << (*CurrentData)[i][j][4] << "\t";
}

AmplitudeFile << std::endl;
xVelocityFile << std::endl;
yVelocityFile << std::endl;
xFieldFile << std::endl;
yFieldFile << std::endl;
}
}
```

```
double GetFullEnergy() {
double EnergyL = 0.0;

for (const auto\& LineI : (*CurrentData)) {
for (const auto\& ValueI : LineI) {
EnergyL += (g * (ValueI[0] - 10.0) +
pow(ValueI[1] / ValueI[0], 2.0) +
pow(ValueI[2] / ValueI[0], 2.0) +
pow(ValueI[3] / ValueI[0], 2.0) +
pow(ValueI[4] / ValueI[0], 2.0));
}
}

return EnergyL;
}
double GetMaxAmplitude() {
double AbsMaxL = 0.0;

for (unsigned long i = 0; i < xSize; i++) {
for (unsigned long j = 0; j < ySize; j++) {
double Value =  fabs((*CurrentData)[i][j][0]) +
fabs((*CurrentData)[i][j][1]) +
fabs((*CurrentData)[i][j][2]) +
fabs((*CurrentData)[i][j][3]) +
fabs((*CurrentData)[i][j][4]);

if (AbsMaxL < Value) {
AbsMaxL = Value;
}
}
}

return AbsMaxL;
}

double getMaxSpeed() {
double VelMax = 0.0;

for (unsigned long i = 0; i < xSize; i++) {
for (unsigned long j = 0; j < ySize; j++) {
double CurVal = 0.0;

if ((*CurrentData)[i][j][0] != 0) {
CurVal = sqrt((*CurrentData)[i][j][1] * (*CurrentData)[i][j][1] +
(*CurrentData)[i][j][2] * (*CurrentData)[i][j][2]) / (*CurrentData)[i][j][0];
}

if (VelMax < CurVal) {
VelMax = CurVal;
}
}
}

return VelMax;
}

void SaveData() {
AmplitudeFile.close();
xVelocityFile.close();
yVelocityFile.close();
xFieldFile.close();
```

```
yFieldFile.close();
}

double getStepTime() {
return mStepTime;
}

double getStepX() {
return mStepX;
}
double getStepY() {
return mStepY;
}

//----------//

void solveGrid() {
long xIndex\_plus\_1;
long xIndex\_minus\_1;
long yIndex\_plus\_1;
long yIndex\_minus\_1;

for (size\_t i = 0; i < xSize; i++) {
for (size\_t  j = 0; j < ySize; j++) {
xIndex\_plus\_1 = (i + 1 == xSize ? 0 : i + 1);
xIndex\_minus\_1 = (i - 1 < 0 ? xSize - 1 : i - 1);
yIndex\_plus\_1 = (j + 1 == ySize ? 0 : j + 1);
yIndex\_minus\_1 = (j - 1 < 0 ? ySize - 1 : j - 1);

(*TempData)[i][j] = solve((*CurrentData)[xIndex\_minus\_1][j],
(*CurrentData)[xIndex\_plus\_1][j],
(*CurrentData)[i][yIndex\_minus\_1],
(*CurrentData)[i][yIndex\_plus\_1]);
}
}

std::swap(CurrentData, TempData);
}
private:
const double g = 9.81e-03;
const double B\_0 = 0.5;
const double f\_0 = 0.1;

unsigned long xSize = 1;
unsigned long ySize = 1;

std::vector <double> Gradient;

std::vector <std::vector <dVector <double, 5>>> DataFirst;
std::vector <std::vector <dVector <double, 5>>> DataSecond;

std::vector <std::vector <dVector <double, 5>>>* CurrentData = \&DataFirst;
std::vector <std::vector <dVector <double, 5>>>* TempData = \&DataSecond;

//----------//

dVector <double, 5> funcX(const dVector <double, 5>\& U) override {
return dVector <double, 5> (U[1],
(pow(U[1], 2.0) - pow(U[3], 2.0)) / U[0] + 0.5 * g * pow(U[0], 2.0),
(U[1] * U[2] - U[3] * U[4]) / U[0],
0,
```

```
(U[1] * U[4] - U[2] * U[3]) / U[0]);
}
dVector <double, 5> funcY(const dVector <double, 5>\& U) override {
return dVector <double, 5> (U[2],
(U[1] * U[2] - U[3] * U[4]) / U[0],
(pow(U[2], 2.0) - pow(U[4], 2.0)) / U[0] + 0.5 * g * pow(U[0], 2.0),
(U[2] * U[3] - U[1] * U[4]) / U[0],
0);
}


//----------//

std::ofstream AmplitudeFile;
std::ofstream xVelocityFile;
std::ofstream yVelocityFile;
std::ofstream xFieldFile;
std::ofstream yFieldFile;

std::string SavePath = "./";

//----------//

void SetPlotParameters(long xMinP, long xMaxP, long yMinP, long yMaxP) {
AmplitudeFile   << xSize        << "\t"     << ySize    << "\t"
<< xMinP        << "\t"     << xMaxP    << "\t"
<< yMinP        << "\t"     << yMaxP    << "\t"
<< "\\n";
xVelocityFile   << xSize        << "\t"     << ySize    << "\t"
<< xMinP        << "\t"     << xMaxP    << "\t"
<< yMinP        << "\t"     << yMaxP    << "\t"
<< "\\n";
yVelocityFile   << xSize        << "\t"     << ySize    << "\t"
<< xMinP        << "\t"     << xMaxP    << "\t"
<< yMinP        << "\t"     << yMaxP    << "\t"
<< "\\n";
xFieldFile      << xSize        << "\t"     << ySize    << "\t"
<< xMinP        << "\t"     << xMaxP    << "\t"
<< yMinP        << "\t"     << yMaxP    << "\t"
<< "\\n";
yFieldFile      << xSize        << "\t"     << ySize    << "\t"
<< xMinP        << "\t"     << xMaxP    << "\t"
<< yMinP        << "\t"     << yMaxP    << "\t"
<< "\\n";
}
void SetExcitation() {
double Fraction = 1.0 / (2.0 * M\_PI * 20.0 * mStepX * 20.0 * mStepY);

for (int i = -100; i < 100; i++) {
for (int j = -100; j < 100; j++) {
(*CurrentData)[xSize / 2 + i][ySize / 2 + j][0] = 10.0 + Fraction * exp(-0.5 * (pow(i / 20.0,
 2.0) + pow(j / 20.0, 2.0)));
}
}
}
};
```

Листинг 4: Lax-Fiendrichs solver

B C++ класс для решения задачи методом Рихтмайера-Цваса

```
class dRichtmyerSolver2D : public dRichtmyer2D <dVector <double, 5>> {
public:
dRichtmyerSolver2D(double TimeStepP, double xStepP, double yStepP) {
mStepTime = TimeStepP;
mStepX = xStepP;
mStepY = yStepP;

//        Homogeneous = false;
}
~dRichtmyerSolver2D() = default;

void SetInitialData(unsigned long xSizeP, unsigned long ySizeP) {
xSize = xSizeP;
ySize = ySizeP;

DataFirst.resize(xSizeP);
DataSecond.resize(xSizeP);

for (unsigned long i = 0; i < xSizeP; i++) {
for (unsigned long j = 0; j < ySizeP; j++) {
DataFirst[i].emplace_back(dVector <double, 5> (10.0, 0.0, 0.0, 0.0, 0.0));
DataSecond[i].emplace_back(dVector <double, 5> (10.0, 0.0, 0.0, 0.0, 0.0));
}
}

SetExcitation();
//        setSlope();

for (unsigned long i = 0; i < ySizeP; i++) {
//            Gradient.emplace_back(0.1 / ySizeP * i);
Gradient.emplace_back(14.58e-05 * i);
}

AmplitudeFile.open(SavePath + "Amplitude.dat");
xVelocityFile.open(SavePath + "xVelocity.dat");
yVelocityFile.open(SavePath + "yVelocity.dat");
xFieldFile.open(SavePath + "xField.dat");
yFieldFile.open(SavePath + "yField.dat");
}
void SetSavePath(const std::string& PathP) {
SavePath = PathP;
}
void AppendData() {
for (int j = 0; j < ySize; j++) {
for (int i = 0; i < xSize; i++) {
AmplitudeFile << (*CurrentData)[i][j][0] << "\t";
xVelocityFile << (*CurrentData)[i][j][1] << "\t";
yVelocityFile << (*CurrentData)[i][j][2] << "\t";
xFieldFile << (*CurrentData)[i][j][3] << "\t";
yFieldFile << (*CurrentData)[i][j][4] << "\t";
}

AmplitudeFile << std::endl;
xVelocityFile << std::endl;
yVelocityFile << std::endl;
xFieldFile << std::endl;
yFieldFile << std::endl;
}
```

```
}

double GetFullEnergy() {
double EnergyL = 0.0;

for (const auto& LineI : (*CurrentData)) {
for (const auto& ValueI : LineI) {
EnergyL += (g * (ValueI[0] - 10.0) +
pow(ValueI[1] / ValueI[0], 2.0) +
pow(ValueI[2] / ValueI[0], 2.0) +
pow(ValueI[3] / ValueI[0], 2.0) +
pow(ValueI[4] / ValueI[0], 2.0));
}
}

return EnergyL;
}
double GetMaxAmplitude() {
double AbsMaxL = 0.0;

for (unsigned long i = 0; i < xSize; i++) {
for (unsigned long j = 0; j < ySize; j++) {
double Value =  fabs((*CurrentData)[i][j][0]) +
fabs((*CurrentData)[i][j][1]) +
fabs((*CurrentData)[i][j][2]) +
fabs((*CurrentData)[i][j][3]) +
fabs((*CurrentData)[i][j][4]);

if (AbsMaxL < Value) {
AbsMaxL = Value;
}
}
}

return AbsMaxL;
}
double getMaxSpeed() {
double VelMax = 0.0;

for (unsigned long i = 0; i < xSize; i++) {
for (unsigned long j = 0; j < ySize; j++) {
double CurVal = 0.0;

if ((*CurrentData)[i][j][0] != 0) {
CurVal = sqrt((*CurrentData)[i][j][1] * (*CurrentData)[i][j][1] +
(*CurrentData)[i][j][2] * (*CurrentData)[i][j][2]) / (*CurrentData)[i][j][0];
}

if (VelMax < CurVal) {
VelMax = CurVal;
}
}
}

return VelMax;
}

void SaveData() {
AmplitudeFile.close();
xVelocityFile.close();
yVelocityFile.close();
```

```
xFieldFile.close();
yFieldFile.close();
}

double getStepTime() {
return mStepTime;
}

double getStepX() {
return mStepX;
}
double getStepY() {
return mStepY;
}

//----------//

void solveGridRichtmyer() {
long xIndex_plus_1;
long xIndex_plus_2;
long xIndex_minus_1;
long xIndex_minus_2;
long yIndex_plus_1;
long yIndex_plus_2;
long yIndex_minus_1;
long yIndex_minus_2;

for (int i = 0; i < xSize; i++) {
for (int j = 0; j < ySize; j++) {
xIndex_plus_1 = (i + 1 == xSize ? 0 : i + 1);
xIndex_plus_2 = (i + 2 >= xSize ? i + 2 - xSize : i + 2);
xIndex_minus_1 = (i - 1 < 0 ? xSize - 1 : i - 1);
xIndex_minus_2 = (i - 2 < 0 ? xSize + i - 2 : i - 2);
yIndex_plus_1 = (j + 1 == ySize ? 0 : j + 1);
yIndex_plus_2 = (j + 2 >= ySize ? j + 2 - ySize : j + 2);
yIndex_minus_1 = (j - 1 < 0 ? ySize - 1 : j - 1);
yIndex_minus_2 = (j - 2 < 0 ? ySize + j - 2 : j - 2);

(*TempData)[i][j] = solve((*CurrentData)[i][j],
(*CurrentData)[xIndex_minus_2][j],
(*CurrentData)[xIndex_plus_2][j],
(*CurrentData)[i][yIndex_minus_2],
(*CurrentData)[i][yIndex_plus_2],
(*CurrentData)[xIndex_plus_1][yIndex_plus_1],
(*CurrentData)[xIndex_minus_1][yIndex_minus_1],
(*CurrentData)[xIndex_plus_1][yIndex_minus_1],
(*CurrentData)[xIndex_minus_1][yIndex_plus_1]) -
viscosity(i, j) * 0.00007;
}
}

std::swap(CurrentData, TempData);
}
void solveGridRichtmyerZwas() {
long xIndex_plus_1;
long xIndex_minus_1;
long yIndex_plus_1;
long yIndex_minus_1;

dVector <double, 5> Extra(0, 0, 0, 0, 0);
```

```
for (int i = 0; i < xSize; i++) {
for (int j = 0; j < ySize; j++) {
xIndex_plus_1 = (i + 1 == xSize ? 0 : i + 1);
xIndex_minus_1 = (i - 1 < 0 ? xSize - 1 : i - 1);
yIndex_plus_1 = (j + 1 == ySize ? 0 : j + 1);
yIndex_minus_1 = (j - 1 < 0 ? ySize - 1 : j - 1);

Extra = nonhomogenPart(i, j) - viscosity(i, j) * 0.000025;

(*TempData)[i][j] = solveZwas((*CurrentData)[i][j],
(*CurrentData)[xIndex_minus_1][j],
(*CurrentData)[xIndex_plus_1][j],
(*CurrentData)[i][yIndex_minus_1],
(*CurrentData)[i][yIndex_plus_1],
(*CurrentData)[xIndex_plus_1][yIndex_plus_1],
(*CurrentData)[xIndex_minus_1][yIndex_minus_1],
(*CurrentData)[xIndex_plus_1][yIndex_minus_1],
(*CurrentData)[xIndex_minus_1][yIndex_plus_1],
Extra);
}
}

std::swap(CurrentData, TempData);
}
private:
const double g = 9.81e-03;
const double B_0 = 65.0e-02;
const double f_0 = 14.58e-05;

unsigned long xSize = 1;
unsigned long ySize = 1;

std::vector <double> Gradient;

std::vector <std::vector <dVector <double, 5>>> DataFirst;
std::vector <std::vector <dVector <double, 5>>> DataSecond;

std::vector <std::vector <dVector <double, 5>>>* CurrentData = &DataFirst;
std::vector <std::vector <dVector <double, 5>>>* TempData = &DataSecond;

//----------//

dVector <double, 5> funcX(const dVector <double, 5>& U) override {
return dVector <double, 5> (U[1],
(pow(U[1], 2.0) - pow(U[3], 2.0)) / U[0] + 0.5 * g * pow(U[0], 2.0),
(U[1] * U[2] - U[3] * U[4]) / U[0],
0,
(U[1] * U[4] - U[2] * U[3]) / U[0]);
}
dVector <double, 5> funcY(const dVector <double, 5>& U) override {
return dVector <double, 5> (U[2],
(U[1] * U[2] - U[3] * U[4]) / U[0],
(pow(U[2], 2.0) - pow(U[4], 2.0)) / U[0] + 0.5 * g * pow(U[0], 2.0),
(U[2] * U[3] - U[1] * U[4]) / U[0],
0);
}
dVector <double, 5> nonhomogenPart(int xPosP, int yPosP) {
return dVector <double, 5> (
0,
B_0 * (*CurrentData)[xPosP][yPosP][3] / (*CurrentData)[xPosP][yPosP][0] - (*CurrentData)[xPosP][
    yPosP][2] * (f_0 + Gradient[yPosP]),
```

```
B_0 * (*CurrentData)[xPosP][yPosP][4] / (*CurrentData)[xPosP][yPosP][0] - (*CurrentData)[xPosP][
    yPosP][1] * (f_0 + Gradient[yPosP]),
-B_0 * (*CurrentData)[xPosP][yPosP][1] / (*CurrentData)[xPosP][yPosP][0],
-B_0 * (*CurrentData)[xPosP][yPosP][2] / (*CurrentData)[xPosP][yPosP][0]);
}
dVector <double, 5> viscosity(int xPosP, int yPosP) {
long xIndex_plus_1;
long xIndex_minus_1;
long yIndex_plus_1;
long yIndex_minus_1;

xIndex_plus_1 = (xPosP + 1 == xSize ? 0 : xPosP + 1);
xIndex_minus_1 = (xPosP - 1 < 0 ? xSize - 1 : xPosP - 1);
yIndex_plus_1 = (yPosP + 1 == ySize ? 0 : yPosP + 1);
yIndex_minus_1 = (yPosP - 1 < 0 ? ySize - 1 : yPosP - 1);

double v_x_xx = ((*CurrentData)[xIndex_minus_1][yPosP][1] / (*CurrentData)[xIndex_minus_1][yPosP
    ][0] +
(*CurrentData)[xPosP][yPosP][1] / (*CurrentData)[xPosP][yPosP][0] * 2 +
(*CurrentData)[xIndex_plus_1][yPosP][1] / (*CurrentData)[xIndex_plus_1][yPosP][0]) /
(mStepX * mStepX);
double v_x_yy = ((*CurrentData)[xPosP][yIndex_minus_1][1] / (*CurrentData)[xPosP][yIndex_minus_1
    ][0] +
(*CurrentData)[xPosP][yPosP][1] / (*CurrentData)[xPosP][yPosP][0] * 2 +
(*CurrentData)[xPosP][yIndex_plus_1][1] / (*CurrentData)[xPosP][yIndex_plus_1][0]) /
(mStepY * mStepY);
double v_y_xx = ((*CurrentData)[xIndex_minus_1][yPosP][2] / (*CurrentData)[xIndex_minus_1][yPosP
    ][0] +
(*CurrentData)[xPosP][yPosP][2] / (*CurrentData)[xPosP][yPosP][0] * 2 +
(*CurrentData)[xIndex_plus_1][yPosP][2] / (*CurrentData)[xIndex_plus_1][yPosP][0]) /
(mStepX * mStepX);
double v_y_yy = ((*CurrentData)[xPosP][yIndex_minus_1][2] / (*CurrentData)[xPosP][yIndex_minus_1
    ][0] +
(*CurrentData)[xPosP][yPosP][2] / (*CurrentData)[xPosP][yPosP][0] * 2 +
(*CurrentData)[xPosP][yIndex_plus_1][2] / (*CurrentData)[xPosP][yIndex_plus_1][0]) /
(mStepY * mStepY);
return dVector <double, 5> (0,
(*CurrentData)[xPosP][yPosP][0] * (v_x_xx + v_x_yy),
(*CurrentData)[xPosP][yPosP][0] * (v_y_xx + v_y_yy),
0,
0);

}

//----------//

std::ofstream AmplitudeFile;
std::ofstream xVelocityFile;
std::ofstream yVelocityFile;
std::ofstream xFieldFile;
std::ofstream yFieldFile;

std::string SavePath = "./";

//----------//

void SetPlotParameters(long xMinP, long xMaxP, long yMinP, long yMaxP) {
AmplitudeFile   << xSize        << "\t"     << ySize    << "\t"
<< xMinP        << "\t"     << xMaxP    << "\t"
<< yMinP        << "\t"     << yMaxP    << "\t"
<< "\n";
```

```
xVelocityFile   << xSize        << "\t"     << ySize    << "\t"
<< xMinP        << "\t"     << xMaxP    << "\t"
<< yMinP        << "\t"     << yMaxP    << "\t"
<< "\n";
yVelocityFile   << xSize        << "\t"     << ySize    << "\t"
<< xMinP        << "\t"     << xMaxP    << "\t"
<< yMinP        << "\t"     << yMaxP    << "\t"
<< "\n";
xFieldFile      << xSize        << "\t"     << ySize    << "\t"
<< xMinP        << "\t"     << xMaxP    << "\t"
<< yMinP        << "\t"     << yMaxP    << "\t"
<< "\n";
yFieldFile      << xSize        << "\t"     << ySize    << "\t"
<< xMinP        << "\t"     << xMaxP    << "\t"
<< yMinP        << "\t"     << yMaxP    << "\t"
<< "\n";
}
void SetExcitation() {
double Fraction = 1.0 / (2.0 * M_PI * 20.0 * mStepX * 20.0 * mStepY);

for (int i = -100; i < 100; i++) {
for (int j = -100; j < 100; j++) {
(*CurrentData)[xSize / 2 + i][ySize / 2 + j][0] = 10.0 + Fraction * exp(-0.5 * (pow(i / 20.0,
    2.0) + pow(j / 20.0, 2.0))) * 10.0;
}
}
}
void setSlope() {
for (int i = 0; i < ySize; i++) {
for (int j = 0; j < xSize; j++) {
(*CurrentData)[j][i][0] = 10.0 + double(i) / ySize * 0.01;
}
}
}
};
```

Листинг 5: Richtmyer-Zwas solver

C C++ класс для решения задачи методом Рунге-Кутта-WENO

```
#include <iostream>
#include <cmath>
#include <utility>
#include <ctime>
#include <fstream>
#include <chrono>
#include <omp.h>

#include "Core/dVector.h"
#include "Core/dMatrix.h"

//-----------------------------//
#include <vulkan/vulkan.h>
#include <SDL2/SDL.h>
#include <SDL2/SDL_vulkan.h>
//-----------------------------//
#include "Renderer.h"
//-----------------------------//

using dGrid = std::vector <std::vector <dVector <double, 5>>>;
using dVec = dVector <double, 5>;

class Solver {
enum DisplayOutput {
DISPLAY_ELEVATION,
DISPLAY_FIELD,
DISPLAY_FIELD_DIVERGENCE
};
public:
Solver() {
GridCurrentData.resize(mGridX + mOffsetXL + mOffsetXR);
GridTempData.resize(mGridX + mOffsetXL + mOffsetXR);

Bottom.resize(mGridX + mOffsetXL + mOffsetXR);

for (size_t i = 0; i < mGridX + mOffsetXL + mOffsetXR; i++) {
GridCurrentData[i].resize(mGridY + mOffsetYU + mOffsetYD);
GridTempData[i].resize(mGridY + mOffsetYU + mOffsetYD);

Bottom[i].resize(mGridY + mOffsetYU + mOffsetYD);
}

AlphaX.resize(mGridY + mOffsetYU + mOffsetYD);
AlphaY.resize(mGridX + mOffsetXL + mOffsetXR);

mOutput.resize(mGridX);

//---Divergence---//
mDivergence.resize(mGridX * mGridY);
B.resize(mGridX * mGridY);
r.resize(mGridX * mGridY);
p.resize(mGridX * mGridY);
Ap.resize(mGridX * mGridY);
ResOpt.resize(mGridX * mGridY);

ResGrid.resize(mGridX + mOffsetXL + mOffsetXR);

for (auto& iRow : ResGrid) {
```

```
iRow.resize(mGridY + mOffsetYU + mOffsetYD);

for (auto& iVal : iRow) {
iVal = 0.0;
}
}

for (size_t i = 0; i < mGridX * mGridY; i++) {
mDivergence[i] = 0.0;
B[i] = 0.0;
r[i] = 0.0;
p[i] = 0.0;
Ap[i] = 0.0;
ResOpt[i] = 0.0;
}
//---Divergence---//

for (auto& iRow : mOutput) {
iRow.resize(mGridY);
}

initCoriolis();
initFields();
bottomFunc();
initConditions();

GridTempData = GridCurrentData;
GridTempData2 = GridCurrentData;

updateBoundaries();
std::swap(CurrentData, TempData);

EOutput.open("Amplitude.dat");
}
~Solver() {
CurrentData = nullptr;
TempData = nullptr;

EOutput.close();
}

void solve() {
findAlpha();

TimePassed += mStepTime;

if (int(TimePassed) % (60 * 15) == 0) {
std::cout << "Time:␣" << TimePassed / 3600 / 24 << "␣Step␣-␣" << mStepTime << "␣Alpha␣-␣" <<
    mMaxAlpha
<< std::endl;
}

#pragma omp parallel for default(none) firstprivate(mMaxAlpha) collapse(2) num_threads(4)
for (size_t i = mOffsetXL; i < mGridX + mOffsetXL; i++) {
for (size_t j = mOffsetYU; j < mGridY + mOffsetYU; j++) {
dVec Val = (*CurrentData)[i][j];

dVec Val_xpl_2 = (*CurrentData)[i + 2][j];
dVec Val_xpl_1 = (*CurrentData)[i + 1][j];
dVec Val_xmin_1 = (*CurrentData)[i - 1][j];
dVec Val_xmin_2 = (*CurrentData)[i - 2][j];
```

```
dVec Val_ypl_2 = (*CurrentData)[i][j + 2];
dVec Val_ypl_1 = (*CurrentData)[i][j + 1];
dVec Val_ymin_1 = (*CurrentData)[i][j - 1];
dVec Val_ymin_2 = (*CurrentData)[i][j - 2];

if (i == mOffsetXL) {
Val_xpl_2[1] = Val[1] / Val[0] * Val_xpl_2[0];
Val_xpl_1[1] = Val[1] / Val[0] * Val_xpl_1[0];
Val_xmin_1[1] = Val[1] / Val[0] * Val_xmin_1[0];
Val_xmin_2[1] = Val[1] / Val[0] * Val_xmin_2[0];

Val_xpl_2[2] = Val[2] / Val[0] * Val_xpl_2[0];
Val_xpl_1[2] = Val[2] / Val[0] * Val_xpl_1[0];
Val_xmin_1[2] = Val[2] / Val[0] * Val_xmin_1[0];
Val_xmin_2[2] = Val[2] / Val[0] * Val_xmin_2[0];
}

if (j == mOffsetYU || j == mGridY + mOffsetYU - 1) {
Val_ypl_2[1] = Val[1] / Val[0] * Val_ypl_2[0];
Val_ypl_1[1] = Val[1] / Val[0] * Val_ypl_1[0];
Val_ymin_1[1] = Val[1] / Val[0] * Val_ymin_1[0];
Val_ymin_2[1] = Val[1] / Val[0] * Val_ymin_2[0];
}

auto PlusX = WENO(
(funcX(Val_xmin_1) + mMaxAlpha * Val_xmin_1) / 2.0,
(funcX(Val) + mMaxAlpha * Val) / 2.0,
(funcX(Val_xpl_1) + mMaxAlpha * Val_xpl_1) / 2.0,
(funcX(Val) - mMaxAlpha * Val) / 2.0,
(funcX(Val_xpl_1) - mMaxAlpha * Val_xpl_1) / 2.0,
(funcX(Val_xpl_2) - mMaxAlpha * Val_xpl_2) / 2.0
);
auto MinusX = WENO(
(funcX(Val_xmin_2) + mMaxAlpha * Val_xmin_2) / 2.0,
(funcX(Val_xmin_1) + mMaxAlpha * Val_xmin_1) / 2.0,
(funcX(Val) + mMaxAlpha * Val) / 2.0,
(funcX(Val_xmin_1) - mMaxAlpha * Val_xmin_1) / 2.0,
(funcX(Val) - mMaxAlpha * Val) / 2.0,
(funcX(Val_xpl_1) - mMaxAlpha * Val_xpl_1) / 2.0
);

auto PlusY = WENO(
(funcY(Val_ymin_1) + mMaxAlpha * Val_ymin_1) / 2.0,
(funcY(Val) + mMaxAlpha * Val) / 2.0,
(funcY(Val_ypl_1) + mMaxAlpha * Val_ypl_1) / 2.0,
(funcY(Val) - mMaxAlpha * Val) / 2.0,
(funcY(Val_ypl_1) - mMaxAlpha * Val_ypl_1) / 2.0,
(funcY(Val_ypl_2) - mMaxAlpha * Val_ypl_2) / 2.0
);
auto MinusY = WENO(
(funcY(Val_ymin_2) + mMaxAlpha * Val_ymin_2) / 2.0,
(funcY(Val_ymin_1) + mMaxAlpha * Val_ymin_1) / 2.0,
(funcY(Val) + mMaxAlpha * Val) / 2.0,
(funcY(Val_ymin_1) - mMaxAlpha * Val_ymin_1) / 2.0,
(funcY(Val) - mMaxAlpha * Val) / 2.0,
(funcY(Val_ypl_1) - mMaxAlpha * Val_ypl_1) / 2.0
);

(*TempData)[i][j] =
(*CurrentData)[i][j] -
```

```
mStepTime *
((PlusX - MinusX) / mStepX + (PlusY - MinusY) / mStepY -
source(i, j));
}
}

updateBoundaries();
std::swap(TempData2, TempData);

#pragma omp parallel for default(none) firstprivate(mMaxAlpha) collapse(2) num_threads(4)
for (size_t i = mOffsetXL; i < mGridX + mOffsetXL; i++) {
for (size_t j = mOffsetYU; j < mGridY + mOffsetYU; j++) {
dVec Val = (*TempData2)[i][j];

dVec Val_xpl_2 = (*TempData2)[i + 2][j];
dVec Val_xpl_1 = (*TempData2)[i + 1][j];
dVec Val_xmin_1 = (*TempData2)[i - 1][j];
dVec Val_xmin_2 = (*TempData2)[i - 2][j];

dVec Val_ypl_2 = (*TempData2)[i][j + 2];
dVec Val_ypl_1 = (*TempData2)[i][j + 1];
dVec Val_ymin_1 = (*TempData2)[i][j - 1];
dVec Val_ymin_2 = (*TempData2)[i][j - 2];

if (i == mOffsetXL) {
Val_xpl_2[1] = Val[1] / Val[0] * Val_xpl_2[0];
Val_xpl_1[1] = Val[1] / Val[0] * Val_xpl_1[0];
Val_xmin_1[1] = Val[1] / Val[0] * Val_xmin_1[0];
Val_xmin_2[1] = Val[1] / Val[0] * Val_xmin_2[0];

Val_xpl_2[2] = Val[2] / Val[0] * Val_xpl_2[0];
Val_xpl_1[2] = Val[2] / Val[0] * Val_xpl_1[0];
Val_xmin_1[2] = Val[2] / Val[0] * Val_xmin_1[0];
Val_xmin_2[2] = Val[2] / Val[0] * Val_xmin_2[0];
}

if (j == mOffsetYU || j == mGridY + mOffsetYU - 1) {
Val_ypl_2[1] = Val[1] / Val[0] * Val_ypl_2[0];
Val_ypl_1[1] = Val[1] / Val[0] * Val_ypl_1[0];
Val_ymin_1[1] = Val[1] / Val[0] * Val_ymin_1[0];
Val_ymin_2[1] = Val[1] / Val[0] * Val_ymin_2[0];
}

auto PlusX = WENO(
(funcX(Val_xmin_1) + mMaxAlpha * Val_xmin_1) / 2.0,
(funcX(Val) + mMaxAlpha * Val) / 2.0,
(funcX(Val_xpl_1) + mMaxAlpha * Val_xpl_1) / 2.0,
(funcX(Val) - mMaxAlpha * Val) / 2.0,
(funcX(Val_xpl_1) - mMaxAlpha * Val_xpl_1) / 2.0,
(funcX(Val_xpl_2) - mMaxAlpha * Val_xpl_2) / 2.0
);
auto MinusX = WENO(
(funcX(Val_xmin_2) + mMaxAlpha * Val_xmin_2) / 2.0,
(funcX(Val_xmin_1) + mMaxAlpha * Val_xmin_1) / 2.0,
(funcX(Val) + mMaxAlpha * Val) / 2.0,
(funcX(Val_xmin_1) - mMaxAlpha * Val_xmin_1) / 2.0,
(funcX(Val) - mMaxAlpha * Val) / 2.0,
(funcX(Val_xpl_1) - mMaxAlpha * Val_xpl_1) / 2.0
);

auto PlusY = WENO(
```

```
(funcY(Val_ymin_1) + mMaxAlpha * Val_ymin_1) / 2.0,
(funcY(Val) + mMaxAlpha * Val) / 2.0,
(funcY(Val_ypl_1) + mMaxAlpha * Val_ypl_1) / 2.0,
(funcY(Val) - mMaxAlpha * Val) / 2.0,
(funcY(Val_ypl_1) - mMaxAlpha * Val_ypl_1) / 2.0,
(funcY(Val_ypl_2) - mMaxAlpha * Val_ypl_2) / 2.0
);
auto MinusY = WENO(
(funcY(Val_ymin_2) + mMaxAlpha * Val_ymin_2) / 2.0,
(funcY(Val_ymin_1) + mMaxAlpha * Val_ymin_1) / 2.0,
(funcY(Val) + mMaxAlpha * Val) / 2.0,
(funcY(Val_ymin_1) - mMaxAlpha * Val_ymin_1) / 2.0,
(funcY(Val) - mMaxAlpha * Val) / 2.0,
(funcY(Val_ypl_1) - mMaxAlpha * Val_ypl_1) / 2.0
);

(*TempData)[i][j] =
0.5 * (*CurrentData)[i][j] +
0.5 * (*TempData2)[i][j] -
0.5 * mStepTime *
((PlusX - MinusX) / mStepX + (PlusY - MinusY) / mStepY -
source(i, j));
}
}

updateBoundaries();

std::swap(CurrentData, TempData);

fixFieldDivergence();
fillOutput(DISPLAY_FIELD_DIVERGENCE);
}
void save() {
for (size_t j = mOffsetYU; j < mGridY + mOffsetYU; j++) {
for (size_t i = mOffsetXL; i < mGridX + mOffsetXL; i++) {
EOutput << (*CurrentData)[i][j][0] + Bottom[i][j] << "␣";
}

EOutput << std::endl;
}
}
void fillOutput(DisplayOutput tType) {
switch (tType) {
case DISPLAY_ELEVATION:
for (size_t j = mOffsetYU; j < mGridY + mOffsetYU; j++) {
for (size_t i = mOffsetXL; i < mGridX + mOffsetXL; i++) {
mOutput[i - mOffsetXL][j - mOffsetYU] = (*CurrentData)[i][j][0] + Bottom[i][j];
}
}

break;
case DISPLAY_FIELD:
for (size_t j = mOffsetYU; j < mGridY + mOffsetYU; j++) {
for (size_t i = mOffsetXL; i < mGridX + mOffsetXL; i++) {
double Bx2 = pow((*CurrentData)[i][j][3] / (*CurrentData)[i][j][0], 2.0);
double By2 = pow((*CurrentData)[i][j][4] / (*CurrentData)[i][j][0], 2.0);

mOutput[i - mOffsetXL][j - mOffsetYU] = sqrt(By2);
}
}
```

```
break;
case DISPLAY_FIELD_DIVERGENCE:
for (size_t j = mOffsetYU; j < mGridY + mOffsetYU; j++) {
for (size_t i = mOffsetXL; i < mGridX + mOffsetXL; i++) {
double dBx_dx = (
(*CurrentData)[i + 1][j][3] / (*CurrentData)[i + 1][j][0] -
(*CurrentData)[i - 1][j][3] / (*CurrentData)[i - 1][j][0]
) / (mStepX * 2.0);
double dBy_dy = (
(*CurrentData)[i][j + 1][4] / (*CurrentData)[i][j + 1][0] -
(*CurrentData)[i][j - 1][4] / (*CurrentData)[i][j - 1][0]
) / (mStepY * 2.0);

mOutput[i - mOffsetXL][j - mOffsetYU] = dBx_dx + dBy_dy;
}
}

break;
default:
break;
}
}
const std::vector <std::vector <double>>& getElevation() const {
return mOutput;
}
private:
size_t mGridX = 254;
size_t mGridY = 50;

size_t mOffsetXL = 2;
size_t mOffsetXR = 2;
size_t mOffsetYU = 2;
size_t mOffsetYD = 2;

dGrid GridCurrentData;
dGrid GridTempData;
dGrid GridTempData2;

dGrid *CurrentData = &GridCurrentData;
dGrid *TempData = &GridTempData;
dGrid *TempData2 = &GridTempData2;

double mStepX = 100.0e+03;
double mStepY = 100.0e+03;

double mStepTime = 60.0;

double mTimeLimit = 10000;

std::vector <std::vector <double>> Bottom;
std::vector <double> mCorParam;
std::vector <double> mHorizFieldY;
std::vector <double> mVertField;

double mMaxAlpha = 0.0;

std::vector <double> AlphaX;
std::vector <double> AlphaY;

double TimePassed = 0.0;
```

```
std::vector <std::vector <double>> mOutput;

//---Divergence---//
std::vector <double> mDivergence;
std::vector <double> B;
std::vector <double> r;
std::vector <double> p;
std::vector <double> Ap;
std::vector <double> ResOpt;

std::vector <std::vector <double>> ResGrid;
//---Divergence---//

//----------//

const double mGrav  = 9.81;
const double mCorParam_0 = 1.0e-04;
//    const double mBetaParam = 1.6e-11;
const double mBetaParam = 0.0;

//----------//

std::ofstream EOutput;

//----------//

void initCoriolis() {
double MeanY = int((mGridY + mOffsetYU + mOffsetYD) / 2) * mStepY;

mCorParam.resize(mGridY + mOffsetYU + mOffsetYD);

for (size_t i = 0; i < mGridY + mOffsetYU + mOffsetYD; i++) {
mCorParam[i] = mCorParam_0 + mBetaParam * (i * mStepY - MeanY);
}
}
void initFields() {
mHorizFieldY.resize(mGridY + mOffsetYU + mOffsetYD);
mVertField.resize(mGridY + mOffsetYU + mOffsetYD);

for (size_t i = 0; i < mGridY + mOffsetYU + mOffsetYD; i++) {
//            mHorizFieldY[i] = 3.5e-05 - 4.87e-07 * i;
//            mVertField[i] = 1.27e-05 + 1.46e-06 * i - 1.38e-08 * pow(i, 2.0);
mHorizFieldY[i] = 0.0;
//            mVertField[i] = 0.0;
mVertField[i] = 21.0;
}
}
void bottomFunc() {
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution <> dis(0.0, 250.0);

double StdX = 5 * mStepX;
double StdY = 5 * mStepY;

double MeanX = int((mGridX + mOffsetXL + mOffsetXR) / 2) * mStepX;
double MeanY = int((mGridY + mOffsetYU + mOffsetYD) / 2) * mStepY;

for (size_t i = 0; i < mGridX + mOffsetXL + mOffsetXR; i++) {
for (size_t j = 0; j < mGridY + mOffsetYU + mOffsetYD; j++) {
Bottom[i][j] = 0.0;
```

```
//                Bottom[i][j] = dis(gen);
Bottom[i][j] = 4000 * exp(
-0.5 * pow((i * mStepX - MeanX) / StdX, 2.0)
-0.5 * pow((j * mStepY - MeanY) / StdY, 2.0));
//                Bottom[i][j] = 4000 * pow(sin(i / 10.0), 2.0) + pow(sin(j / 10.0), 2.0);
}
}
}
void updateBoundaries() {
//---Periodic East-West---//

//---h---//
for (size_t j = mOffsetYU; j < mGridY + mOffsetYU; j++) {
(*TempData)[0][j][0] = (*TempData)[mGridX + mOffsetXL - 2][j][0];
(*TempData)[1][j][0] = (*TempData)[mGridX + mOffsetXL - 1][j][0];

(*TempData)[mGridX + mOffsetXL + mOffsetXR - 2][j][0] = (*TempData)[mOffsetXL][j][0];
(*TempData)[mGridX + mOffsetXL + mOffsetXR - 1][j][0] = (*TempData)[mOffsetXL + 1][j][0];
}
//---h---//

//---vx-vy---//
for (int iComp = 1; iComp < 3; iComp++) {
for (size_t j = mOffsetYU; j < mGridY + mOffsetYU; j++) {
(*TempData)[mGridX + mOffsetXL + mOffsetXR - 2][j][iComp] =
(*TempData)[mOffsetXL][j][iComp] /
(*TempData)[mOffsetXL][j][0] *
(*TempData)[mGridX + mOffsetXL + mOffsetXR - 2][j][0];
(*TempData)[mGridX + mOffsetXL + mOffsetXR - 1][j][iComp] =
(*TempData)[mOffsetXL + 1][j][iComp] /
(*TempData)[mOffsetXL + 1][j][0] *
(*TempData)[mGridX + mOffsetXL + mOffsetXR - 1][j][0];
}
}
//---vx-vy---//

//---Bx-By-Psi---//
for (int iComp = 3; iComp < 5; iComp++) {
for (size_t j = mOffsetYU; j < mGridY + mOffsetYU; j++) {
(*TempData)[0][j][iComp] =
(*TempData)[mGridX + mOffsetXL - 2][j][iComp] /
(*TempData)[mGridX + mOffsetXL - 2][j][0] *
(*TempData)[0][j][0];
(*TempData)[1][j][iComp] =
(*TempData)[mGridX + mOffsetXL - 1][j][iComp] /
(*TempData)[mGridX + mOffsetXL - 1][j][0] *
(*TempData)[1][j][0];

(*TempData)[mGridX + mOffsetXL + mOffsetXR - 2][j][iComp] =
(*TempData)[mOffsetXL][j][iComp] /
(*TempData)[mOffsetXL][j][0] *
(*TempData)[mGridX + mOffsetXL + mOffsetXR - 2][j][0];
(*TempData)[mGridX + mOffsetXL + mOffsetXR - 1][j][iComp] =
(*TempData)[mOffsetXL + 1][j][iComp] /
(*TempData)[mOffsetXL + 1][j][0] *
(*TempData)[mGridX + mOffsetXL + mOffsetXR - 1][j][0];
}
}
//---Bx-By-Psi---//

//---Periodic East-West---//
```

```
//------------------------------//

//---Extrapolation East---//

//---vx-vy---//
for (int iComp = 1; iComp < 3; iComp++) {
for (size_t j = mOffsetYU; j < mGridY + mOffsetYU; j++) {
(*TempData)[1][j][iComp] = extrapolate(
(*TempData)[mOffsetXL][j][iComp] / (*TempData)[mOffsetXL][j][0],
(*TempData)[mOffsetXL + 1][j][iComp] / (*TempData)[mOffsetXL + 1][j][0],
(*TempData)[mOffsetXL + 2][j][iComp] / (*TempData)[mOffsetXL + 2][j][0],
(*TempData)[mOffsetXL + 3][j][iComp] / (*TempData)[mOffsetXL + 3][j][0]
) * (*TempData)[1][j][0];
}
}
//---vx-vy---//

//---Extrapolation East---//

//------------------------------//

//---Fixed North-South---//

for (size_t i = mOffsetXL; i < mGridX + mOffsetXL; i++) {
//---vy---//
(*TempData)[i][1][2] = 0.0;
(*TempData)[i][0][2] = 0.0;

(*TempData)[i][mGridY + mOffsetYU + mOffsetYD - 2][2] = 0.0;
(*TempData)[i][mGridY + mOffsetYU + mOffsetYD - 1][2] = 0.0;
//---vy---//

//---Bx---//
(*TempData)[i][0][3] = 0.0;
(*TempData)[i][1][3] = 0.0;

(*TempData)[i][mGridY + mOffsetYU + mOffsetYD - 2][3] = 0.0;
(*TempData)[i][mGridY + mOffsetYU + mOffsetYD - 1][3] = 0.0;
//---Bx---//

//---By---//
(*TempData)[i][0][4] = mHorizFieldY[0] * (*TempData)[i][0][0];
(*TempData)[i][1][4] = mHorizFieldY[1] * (*TempData)[i][1][0];

(*TempData)[i][mGridY + mOffsetYU + mOffsetYD - 1][4] =
mHorizFieldY[mGridY + mOffsetYU + mOffsetYD - 1] *
(*TempData)[i][mGridY + mOffsetYU + mOffsetYD - 1][0];
(*TempData)[i][mGridY + mOffsetYU + mOffsetYD - 2][4] =
mHorizFieldY[mGridY + mOffsetYU + mOffsetYD - 2] *
(*TempData)[i][mGridY + mOffsetYU + mOffsetYD - 2][0];
//---By---//

//---Psi---//
//            (*TempData)[i][1][5] = extrapolate(
//                    (*TempData)[i][mOffsetYU][5] / (*TempData)[i][mOffsetYU][0],
//                    (*TempData)[i][mOffsetYU + 1][5] / (*TempData)[i][mOffsetYU + 1][0],
//                    (*TempData)[i][mOffsetYU + 2][5] / (*TempData)[i][mOffsetYU + 2][0],
//                    (*TempData)[i][mOffsetYU + 3][5] / (*TempData)[i][mOffsetYU + 3][0]
//                    ) * (*TempData)[i][1][0];
//
```

```
//            (*TempData)[i][mGridY + mOffsetYU + mOffsetYD - 2][5] = extrapolate(
//                    (*TempData)[i][mGridY + mOffsetYU - 1][5] / (*TempData)[i][mGridY +
    mOffsetYU - 1][0],
//                    (*TempData)[i][mGridY + mOffsetYU - 2][5] / (*TempData)[i][mGridY +
    mOffsetYU - 2][0],
//                    (*TempData)[i][mGridY + mOffsetYU - 3][5] / (*TempData)[i][mGridY +
    mOffsetYU - 3][0],
//                    (*TempData)[i][mGridY + mOffsetYU - 4][5] / (*TempData)[i][mGridY +
    mOffsetYU - 4][0]
//                    ) * (*TempData)[i][mGridY + mOffsetYU + mOffsetYD - 2][0];
//---Psi---//
}

//---Fixed North-South---//

//-----------------------------//

//---Extrapolation North-South---//

//---vx---//
for (size_t i = mOffsetXL; i < mGridX + mOffsetXL; i++) {
(*TempData)[i][1][1] = extrapolate(
(*TempData)[i][mOffsetYU][1] / (*TempData)[i][mOffsetYU][0],
(*TempData)[i][mOffsetYU + 1][1] / (*TempData)[i][mOffsetYU + 1][0],
(*TempData)[i][mOffsetYU + 2][1] / (*TempData)[i][mOffsetYU + 2][0],
(*TempData)[i][mOffsetYU + 3][1] / (*TempData)[i][mOffsetYU + 3][0]
) * (*TempData)[i][1][0];

(*TempData)[i][mGridY + mOffsetYU + mOffsetYD - 2][1] = extrapolate(
(*TempData)[i][mGridY + mOffsetYU - 1][1] / (*TempData)[i][mGridY + mOffsetYU - 1][0],
(*TempData)[i][mGridY + mOffsetYU - 2][1] / (*TempData)[i][mGridY + mOffsetYU - 2][0],
(*TempData)[i][mGridY + mOffsetYU - 3][1] / (*TempData)[i][mGridY + mOffsetYU - 3][0],
(*TempData)[i][mGridY + mOffsetYU - 4][1] / (*TempData)[i][mGridY + mOffsetYU - 4][0]
) * (*TempData)[i][mGridY + mOffsetYU + mOffsetYD - 2][0];
}
//---vx---//

//---Extrapolation North-South---//
}
void initConditions() {
double MeanWind = 20.0;
double MeanY = int((mGridY + mOffsetYU + mOffsetYD) / 2) * mStepY;

for (size_t i = 0; i < mGridX + mOffsetXL + mOffsetXR; i++) {
for (size_t j = 0; j < mGridY + mOffsetYU + mOffsetYD; j++) {
double TempHeight = 10000.0 - (MeanWind * mCorParam_0 / mGrav) * (j * mStepY - MeanY);

GridCurrentData[i][j] = dVec(
TempHeight,
0.0,
0.0,
0.0,
TempHeight * mHorizFieldY[j]
//                          0.0
);
}
}

for (size_t i = 0; i < mGridX + mOffsetXL + mOffsetXR; i++) {
for (size_t j = 1; j < mGridY + mOffsetYU + mOffsetYD - 1; j++) {
GridCurrentData[i][j][1] =
```

```
GridCurrentData[i][j][0] *
(-0.5 * mGrav / (mCorParam[j] * mStepX) *
(GridCurrentData[i][j + 1][0] - GridCurrentData[i][j - 1][0]));
}
}

for (size_t i = 1; i < mGridX + mOffsetXL + mOffsetXR - 1; i++) {
for (size_t j = 0; j < mGridY + mOffsetYU + mOffsetYD; j++) {
GridCurrentData[i][j][2] =
GridCurrentData[i][j][0] *
(0.5 * mGrav / (mCorParam[j] * mStepX) *
(GridCurrentData[i + 1][j][0] - GridCurrentData[i - 1][j][0]));
}
}

for (size_t i = 0; i < mGridX + mOffsetXL + mOffsetXR; i++) {
GridCurrentData[i][0][2] = 0.0;
GridCurrentData[i][mGridY + mOffsetYU + mOffsetYD - 1][2] = 0.0;
}

for (size_t i = 0; i < mGridX + mOffsetXL + mOffsetXR; i++) {
for (size_t j = 0; j < mGridY + mOffsetYU + mOffsetYD; j++) {
GridCurrentData[i][j][0] -= Bottom[i][j];
}
}
}

dVec funcX(const dVec& tVal) {
return dVec(
tVal[1],
(tVal[1] * tVal[1] - tVal[3] * tVal[3]) / tVal[0] + 0.5 * mGrav * tVal[0] * tVal[0],
(tVal[1] * tVal[2] - tVal[3] * tVal[4]) / tVal[0],
0.0,
//                tVal[5],
-(tVal[1] * tVal[4] - tVal[2] * tVal[3]) / tVal[0]
//                mMaxAlpha * mMaxAlpha * tVal[3]
);
}
dVec funcY(const dVec& tVal) {
return dVec(
tVal[2],
(tVal[1] * tVal[2] - tVal[3] * tVal[4]) / tVal[0],
(tVal[2] * tVal[2] - tVal[4] * tVal[4]) / tVal[0] + 0.5 * mGrav * tVal[0] * tVal[0],
-(tVal[2] * tVal[3] - tVal[1] * tVal[4]) / tVal[0],
0.0
//                tVal[5],
//                mMaxAlpha * mMaxAlpha * tVal[4]
);
}
dVec source(int tPosX, int tPosY) {
double Bz = -mVertField[tPosY];
//        double Bz = -(((*CurrentData)[tPosX + 1][tPosY][3] - (*CurrentData)[tPosX - 1][tPosY
    ][3]) / mStepX / 2.0 +
//                 ((*CurrentData)[tPosX][tPosY + 1][4] - (*CurrentData)[tPosX][tPosY - 1][4]) /
    mStepY / 2.0);

//        double dhBx_dx = ((*CurrentData)[tPosX + 1][tPosY][3] - (*CurrentData)[tPosX - 1][tPosY
    ][3]) / mStepX / 2.0;
//        double dhBy_dy = ((*CurrentData)[tPosX][tPosY + 1][4] - (*CurrentData)[tPosX][tPosY -
    1][4]) / mStepY / 2.0;
```

```
//        double dvx_dx = (
//                (*CurrentData)[tPosX + 1][tPosY][1] / (*CurrentData)[tPosX + 1][tPosY][0] -
//                (*CurrentData)[tPosX - 1][tPosY][1] / (*CurrentData)[tPosX - 1][tPosY][0]
//                ) / mStepX / 2.0;
//        double dvy_dy = (
//                (*CurrentData)[tPosX][tPosY + 1][2] / (*CurrentData)[tPosX][tPosY + 1][0] -
//                (*CurrentData)[tPosX][tPosY - 1][2] / (*CurrentData)[tPosX][tPosY - 1][0]
//                ) / mStepY / 2.0;

double vx = (*CurrentData)[tPosX][tPosY][1] / (*CurrentData)[tPosX][tPosY][0];
double vy = (*CurrentData)[tPosX][tPosY][2] / (*CurrentData)[tPosX][tPosY][0];
double Bx = (*CurrentData)[tPosX][tPosY][3] / (*CurrentData)[tPosX][tPosY][0];
double By = (*CurrentData)[tPosX][tPosY][4] / (*CurrentData)[tPosX][tPosY][0];

double dh_dx = (Bottom[tPosX + 1][tPosY] - Bottom[tPosX - 1][tPosY]) / (2.0 * mStepX);
double dh_dy = (Bottom[tPosX][tPosY + 1] - Bottom[tPosX][tPosY - 1]) / (2.0 * mStepY);

return dVec(
0.0,
Bz * Bx +
mCorParam[tPosY] * (*CurrentData)[tPosX][tPosY][2] -
mGrav * (*CurrentData)[tPosX][tPosY][0] * dh_dx,
Bz * By +
-mCorParam[tPosY] * (*CurrentData)[tPosX][tPosY][1] -
mGrav * (*CurrentData)[tPosX][tPosY][0] * dh_dy,
Bz * vx,
Bz * vy
//                Bz * vy,
//                -mMaxAlpha * mMaxAlpha * Bz
);
}
dVec viscosity(int tPosX, int tPosY) {
double v_x_xx = (
(*CurrentData)[tPosX - 1][tPosY][1] / (*CurrentData)[tPosX - 1][tPosY][0] +
(*CurrentData)[tPosX][tPosY][1] / (*CurrentData)[tPosX][tPosY][0] * 2.0 +
(*CurrentData)[tPosX + 1][tPosY][1] / (*CurrentData)[tPosX + 1][tPosY][0]) /
pow(mStepX, 2.0);
double v_x_yy = (
(*CurrentData)[tPosX][tPosY - 1][1] / (*CurrentData)[tPosX][tPosY - 1][0] +
(*CurrentData)[tPosX][tPosY][1] / (*CurrentData)[tPosX][tPosY][0] * 2.0 +
(*CurrentData)[tPosX][tPosY + 1][1] / (*CurrentData)[tPosX][tPosY + 1][0]) /
pow(mStepY, 2.0);
double v_y_xx = (
(*CurrentData)[tPosX - 1][tPosY][2] / (*CurrentData)[tPosX - 1][tPosY][0] +
(*CurrentData)[tPosX][tPosY][2] / (*CurrentData)[tPosX][tPosY][0] * 2.0 +
(*CurrentData)[tPosX + 1][tPosY][2] / (*CurrentData)[tPosX + 1][tPosY][0]) /
pow(mStepX, 2.0);
double v_y_yy = (
(*CurrentData)[tPosX][tPosY - 1][2] / (*CurrentData)[tPosX][tPosY - 1][0] +
(*CurrentData)[tPosX][tPosY][2] / (*CurrentData)[tPosX][tPosY][0] * 2.0 +
(*CurrentData)[tPosX][tPosY + 1][2] / (*CurrentData)[tPosX][tPosY + 1][0]) /
pow(mStepY, 2.0);

return dVec (
0.0,
(*CurrentData)[tPosX][tPosY][0] * (v_x_xx + v_x_yy),
(*CurrentData)[tPosX][tPosY][0] * (v_y_xx + v_y_yy),
0.0,
//                0.0,
0.0);
}
```

```
//----------//

dVec WENO(
const dVec& tPosVal_minus_1,
const dVec& tPosVal,
const dVec& tPosVal_plus_1,
const dVec& tNegVal,
const dVec& tNegVal_plus_1,
const dVec& tNegVal_plus_2) {
dVec fPlus(0.0, 0.0, 0.0, 0.0, 0.0);
dVec fMinus(0.0, 0.0, 0.0, 0.0, 0.0);

for (int i = 0; i < 5; i++) {
double Beta0 = pow(tPosVal_minus_1[i] - tPosVal[i], 2.0);
double Beta1 = pow(tPosVal[i] - tPosVal_plus_1[i], 2.0);

double d0 = 1.0 / 3.0;
double d1 = 2.0 / 3.0;

double Alpha0 = d0 / pow(1.0e-06 + Beta0, 2.0);
double Alpha1 = d1 / pow(1.0e-06 + Beta1, 2.0);

double Omega0 = Alpha0 / (Alpha0 + Alpha1);
double Omega1 = Alpha1 / (Alpha0 + Alpha1);

fPlus[i] =
Omega0 * (-0.5 * tPosVal_minus_1[i] + 1.5 * tPosVal[i]) +
Omega1 * (0.5 * tPosVal[i] + 0.5 * tPosVal_plus_1[i]);
}

for (int i = 0; i < 5; i++) {
double Beta0 = pow(tNegVal[i] - tNegVal_plus_1[i], 2.0);
double Beta1 = pow(tNegVal_plus_1[i] - tNegVal_plus_2[i], 2.0);

double d0 = 2.0 / 3.0;
double d1 = 1.0 / 3.0;

double Alpha0 = d0 / pow(1.0e-06 + Beta0, 2.0);
double Alpha1 = d1 / pow(1.0e-06 + Beta1, 2.0);

double Omega0 = Alpha0 / (Alpha0 + Alpha1);
double Omega1 = Alpha1 / (Alpha0 + Alpha1);

fMinus[i] =
Omega0 * (0.5 * tNegVal[i] + 0.5 * tNegVal_plus_1[i]) +
Omega1 * (1.5 * tNegVal_plus_1[i] - 0.5 * tNegVal_plus_2[i]);
}

return fPlus + fMinus;
}

double extrapolate(double tOffset_1, double tOffset_2, double tOffset_3, double tOffset_4) {
return 4.0 * tOffset_1 - 6.0 * tOffset_2 + 4.0 * tOffset_3 - tOffset_4;
}
void AxOpt(const std::vector <double>& tMult, std::vector <double>& tRes, size_t tRows, size_t
    tColumns) {
for (size_t i = 0; i < tRows * tColumns; i++) {
tRes[i] = 4 * tMult[i];

if (i >= 1 && i % tRows != 0) {
```

```
tRes[i - 1] -= tMult[i];
tRes[i] -= tMult[i - 1];
}

if (i >= tRows) {
tRes[i - tRows] -= tMult[i];
tRes[i] -= tMult[i - tRows];
}
}
}
void conjugateGradient() {
double rsold = 0.0;
AxOpt(mDivergence, ResOpt, mGridY, mGridX);

for (size_t i = 0; i < mGridX * mGridY; i++) {
r[i] = B[i] - ResOpt[i];
p[i] = r[i];

rsold += r[i] * r[i];
}

for (size_t i = 0; i < mGridX * mGridY; i++) {
AxOpt(p, Ap, mGridY, mGridX);
double C = 0.0;

for (size_t j = 0; j < Ap.size(); j++) {
C += p[j] * Ap[j];
}

double Alpha = rsold / C;

for (size_t j = 0; j < Ap.size(); j++) {
mDivergence[j] += Alpha * p[j];
r[j] -= Alpha * Ap[j];
}

double rsnew = 0.0;

for (size_t j = 0; j < Ap.size(); j++) {
rsnew += r[j] * r[j];
}

if (sqrt(rsnew) < 1.0e-10) {
break;
}

double Ratio = rsnew / rsold;

for (size_t j = 0; j < Ap.size(); j++) {
p[j] = r[j] + Ratio * p[j];
}

rsold = rsnew;
}
}
void fixFieldDivergence() {
for (size_t i = mOffsetXL; i < mGridX + mOffsetXL; i++) {
for (size_t j = mOffsetYU; j < mGridY + mOffsetYU; j++) {
double dBx_dx = (
(*CurrentData)[i + 1][j][3] / (*CurrentData)[i + 1][j][0] -
(*CurrentData)[i - 1][j][3] / (*CurrentData)[i - 1][j][0]
```

```
) / (mStepX * 2.0);
double dBy_dy = (
(*CurrentData)[i][j + 1][4] / (*CurrentData)[i][j + 1][0] -
(*CurrentData)[i][j - 1][4] / (*CurrentData)[i][j - 1][0]
) / (mStepY * 2.0);

B[(i - mOffsetXL) * mGridY + j - mOffsetYU] = -(dBx_dx + dBy_dy) * mStepX * mStepX;
//                mDivergence[(i - mOffsetXL) * mGridY + j - mOffsetYU] = 0.0;
}
}

conjugateGradient();

for (size_t i = mOffsetXL; i < mGridX + mOffsetXL; i++) {
for (size_t j = mOffsetYU; j < mGridY + mOffsetYU; j++) {
ResGrid[i][j] = mDivergence[(i - mOffsetXL) * mGridY + j - mOffsetYU];
}
}

for (size_t i = mOffsetXL + 1; i < mGridX + mOffsetXL - 1; i++) {
for (size_t j = mOffsetYU + 1; j < mGridY + mOffsetYU - 1; j++) {
(*CurrentData)[i][j][3] =
(*CurrentData)[i][j][3] / (*CurrentData)[i][j][0] -
(ResGrid[i + 1][j] - ResGrid[i - 1][j]) / (2.0 * mStepX) *
(*CurrentData)[i][j][0];
(*CurrentData)[i][j][4] =
(*CurrentData)[i][j][4] / (*CurrentData)[i][j][0] -
(ResGrid[i][j + 1] - ResGrid[i][j - 1]) / (2.0 * mStepX) *
(*CurrentData)[i][j][0];
}
}

for (size_t j = mOffsetYU + 1; j < mGridY + mOffsetYU - 1; j++) {
(*CurrentData)[mOffsetXL][j][3] =
(*CurrentData)[mOffsetXL][j][3] / (*CurrentData)[mOffsetXL][j][0] -
ResGrid[mOffsetXL + 1][j] / (2.0 * mStepX) *
(*CurrentData)[mOffsetXL][j][0];
(*CurrentData)[mGridX + mOffsetXL - 1][j][3] =
(*CurrentData)[mGridX + mOffsetXL - 1][j][3] / (*CurrentData)[mGridX + mOffsetXL - 1][j][0] +
ResGrid[mGridX + mOffsetXL - 2][j] / (2.0 * mStepX) *
(*CurrentData)[mGridX + mOffsetXL - 1][j][0];

(*CurrentData)[mOffsetXL][j][4] =
(*CurrentData)[mOffsetXL][j][4] / (*CurrentData)[mOffsetXL][j][0] -
(ResGrid[mOffsetXL][j + 1] - ResGrid[mOffsetXL][j - 1]) / (2.0 * mStepY) *
(*CurrentData)[mOffsetXL][j][0];
(*CurrentData)[mGridX + mOffsetXL - 1][j][4] =
(*CurrentData)[mGridX + mOffsetXL - 1][j][4] / (*CurrentData)[mGridX + mOffsetXL - 1][j][0] -
(ResGrid[mGridX + mOffsetXL - 1][j + 1] - ResGrid[mGridX + mOffsetXL - 1][j - 1]) / (2.0 * mStepY
    ) *
(*CurrentData)[mGridX + mOffsetXL - 1][j][0];
}

for (size_t i = mOffsetXL + 1; i < mGridX + mOffsetXL - 1; i++) {
(*CurrentData)[i][mOffsetYU][3] =
(*CurrentData)[i][mOffsetYU][3] / (*CurrentData)[i][mOffsetYU][0] -
(ResGrid[i + 1][mOffsetYU] - ResGrid[i - 1][mOffsetYU]) / (2.0 * mStepX) *
(*CurrentData)[i][mOffsetYU][0];
(*CurrentData)[i][mGridY + mOffsetYU - 1][3] =
(*CurrentData)[i][mGridY + mOffsetYU - 1][3] / (*CurrentData)[i][mGridY + mOffsetYU - 1][0] -
```

```
(ResGrid[i + 1][mGridY + mOffsetYU - 1] - ResGrid[i - 1][mGridY + mOffsetYU - 1]) / (2.0 * mStepX
    ) *
(*CurrentData)[i][mGridY + mOffsetYU - 1][0];

(*CurrentData)[i][mOffsetYU][4] =
(*CurrentData)[i][mOffsetYU][4] / (*CurrentData)[i][mOffsetYU][0] -
ResGrid[i][mOffsetYU + 1] / (2.0 * mStepY) *
(*CurrentData)[i][mOffsetYU][0];
(*CurrentData)[i][mGridY + mOffsetYU - 1][4] =
(*CurrentData)[i][mGridY + mOffsetYU - 1][4] / (*CurrentData)[i][mGridY + mOffsetYU - 1][0] +
ResGrid[i][mGridY + mOffsetYU - 2] / (2.0 * mStepY) *
(*CurrentData)[i][mGridY + mOffsetYU - 1][0];
}

(*CurrentData)[mOffsetXL][mOffsetYU][3] =
(*CurrentData)[mOffsetXL][mOffsetYU][3] / (*CurrentData)[mOffsetXL][mOffsetYU][0] -
ResGrid[mOffsetXL + 1][mOffsetYU] / (2.0 * mStepX) *
(*CurrentData)[mOffsetXL][mOffsetYU][0];
(*CurrentData)[mOffsetXL][mOffsetYU][4] =
(*CurrentData)[mOffsetXL][mOffsetYU][4] / (*CurrentData)[mOffsetXL][mOffsetYU][0] -
ResGrid[mOffsetXL][mOffsetYU + 1] / (2.0 * mStepY) *
(*CurrentData)[mOffsetXL][mOffsetYU][0];

(*CurrentData)[mGridX + mOffsetXL - 1][mOffsetYU][3] =
(*CurrentData)[mGridX + mOffsetXL - 1][mOffsetYU][3] / (*CurrentData)[mGridX + mOffsetXL - 1][
    mOffsetYU][0] +
ResGrid[mGridX + mOffsetXL - 2][mOffsetYU] / (2.0 * mStepX) *
(*CurrentData)[mGridX + mOffsetXL - 1][mOffsetYU][0];
(*CurrentData)[mGridX + mOffsetXL - 1][mOffsetYU][4] =
(*CurrentData)[mGridX + mOffsetXL - 1][mOffsetYU][4] / (*CurrentData)[mGridX + mOffsetXL - 1][
    mOffsetYU][0] -
ResGrid[mGridX + mOffsetXL - 1][mOffsetYU + 1] / (2.0 * mStepY) *
(*CurrentData)[mGridX + mOffsetXL - 1][mOffsetYU][0];

(*CurrentData)[mOffsetXL][mGridY + mOffsetYU - 1][3] =
(*CurrentData)[mOffsetXL][mGridY + mOffsetYU - 1][3] / (*CurrentData)[mOffsetXL][mGridY +
    mOffsetYU - 1][0] -
ResGrid[mOffsetXL + 1][mGridY + mOffsetYU - 1] / (2.0 * mStepX) *
(*CurrentData)[mOffsetXL][mGridY + mOffsetYU - 1][0];
(*CurrentData)[mOffsetXL][mGridY + mOffsetYU - 1][4] =
(*CurrentData)[mOffsetXL][mGridY + mOffsetYU - 1][4] / (*CurrentData)[mOffsetXL][mGridY +
    mOffsetYU - 1][0] +
ResGrid[mOffsetXL][mGridY + mOffsetYU - 2] / (2.0 * mStepY) *
(*CurrentData)[mOffsetXL][mGridY + mOffsetYU - 1][0];

(*CurrentData)[mGridX + mOffsetXL - 1][mGridY + mOffsetYU - 1][3] =
(*CurrentData)[mGridX + mOffsetXL - 1][mGridY + mOffsetYU - 1][3] / (*CurrentData)[mGridX +
    mOffsetXL - 1][mGridY + mOffsetYU - 1][0] +
ResGrid[mGridX + mOffsetXL - 2][mGridY + mOffsetYU - 1] / (2.0 * mStepX) *
(*CurrentData)[mGridX + mOffsetXL - 1][mGridY + mOffsetYU - 1][0];
(*CurrentData)[mGridX + mOffsetXL - 1][mGridY + mOffsetYU - 1][4] =
(*CurrentData)[mGridX + mOffsetXL - 1][mGridY + mOffsetYU - 1][4] / (*CurrentData)[mGridX +
    mOffsetXL - 1][mGridY + mOffsetYU - 1][0] +
ResGrid[mGridX + mOffsetXL - 1][mGridY + mOffsetYU - 2] / (2.0 * mStepY) *
(*CurrentData)[mGridX + mOffsetXL - 1][mGridY + mOffsetYU - 1][0];
}
void findAlpha() {
mMaxAlpha = 0.0;

for (size_t iX = 0; iX < mGridX + mOffsetXL + mOffsetXR; iX++) {
for (size_t iY = 0; iY < mGridY + mOffsetYU + mOffsetYD; iY++) {
```

```
double vx = (*CurrentData)[iX][iY][1] / (*CurrentData)[iX][iY][0];
double vy = (*CurrentData)[iX][iY][2] / (*CurrentData)[iX][iY][0];
double Bx = (*CurrentData)[iX][iY][3] / (*CurrentData)[iX][iY][0];
double By = (*CurrentData)[iX][iY][4] / (*CurrentData)[iX][iY][0];
double SqrX = sqrt(pow(Bx, 2.0) + 9.81 * (*CurrentData)[iX][iY][0]);
double SqrY = sqrt(pow(By, 2.0) + 9.81 * (*CurrentData)[iX][iY][0]);

mMaxAlpha = std::max(fabs(vx + SqrX), mMaxAlpha);
mMaxAlpha = std::max(fabs(vy + SqrY), mMaxAlpha);
}
}

//            if (mMaxAlpha > 500) {
//                break;
//            }

//            mStepTime = 0.2 * mStepX / mMaxAlpha;
}
};
```

Листинг 6: RK2-WENO solver