DUNE PDELab Tutorial 07
Discontinuous Galerkin Method for Hyperbolic Conservation Laws

DUNE/PDELab Team

August 31, 2021

Contents

1 Introduction

In this tutorial we provide DG solver for hyperbolic conservation laws. As an example of hyperbolic system we consider: linear acoustics, shallow water equation, treatment of systems of hyperbolic partial differential equations in PDELab.

2 PDE Problem

We are interested in the numerical solution to the first-order hyperbolic partial differential equations (PDEs). The general conservative form of the hyperbolic problem, for the unknown $u \in \mathbb{R}^m$ reads as follows

$$\partial_t u(x,t) + \nabla \cdot F(u(x,t),x,t) = g(u(x,t),x,t) \quad \text{in } U = \Omega \times \Sigma, \tag{1}$$

where the matrix-valued function $F : \mathbb{R}^m \times \Omega \times \Sigma \to \mathbb{R}^{m \times d}$ with the columns $F(u,x,t) = [F_1(u,x,t), \ldots, F_d(u,x,t)]$ is called *flux function.* Note that the divergence is defined as $\nabla \cdot F(u(x,t),x,t) = \sum_{j=1}^d \partial_{x_j} F_j(u(x,t),x,t)$. Moreover let $\Omega = \mathbb{R}^d$, $d \in \mathbb{N}$ is the spatial domain, and $\Sigma = \mathbb{R}^+$ is the temporal domain. Equation (1) is supplemented with initial conditions

$$u(x,0) = u_0(x).$$

Equation (1) is said to be in *conservative form* as it arises naturally from the formulation of conservation of mass, momentum and energy. If the flux function is smooth enough, the PDE can be put in its *non-conservative* or *quasi-linear* form which reads

$$\partial_t u(x,t) + \sum_{j=1}^d B_j(u(x,t),x,t)\partial_{x_j} u(x,t) + \tilde{g}(u(x,t),x,t) = 0 \quad \text{in } \Omega \times \Sigma. \tag{2}$$

The reason is the chain rule

$$\partial_{x_j} F_{i,j}(u(x,t),x,t) = \sum_{k=1}^m \frac{\partial F_{i,j}}{\partial u_k}(u(x,t),x,t)\frac{\partial u_k}{\partial x_j}(x,t) + \frac{\partial F_{i,j}}{\partial x_j}(u(x,t),x,t)$$

which shows

$$(B_j(u,x,t))_{i,k} = \frac{\partial F_{i,j}}{\partial u_k}(u,x,t), \qquad \tilde{g}_i(u,x,t) = g_i(u,x,t) + \frac{\partial F_{i,j}}{\partial x_j}(u,x,t).$$

It turns out that many systems of the form (2) which are of practical interest satisfy an important property that is essential in the theoretical and numerical treatment.

Definition 1 (Hyperbolic First-Order PDE)**.** The system of equations (2) is called *hyperbolic* if for each feasible state $u \in \mathbb{R}^m$, $x \in \Omega$, $t \in \Sigma$ and $y \in \mathbb{R}^d$ the $m \times m$ matrix

$$B(u,x,t;y) = \sum_{j=1}^d y_j B_j(u,x,t) \tag{3}$$

is real diagonalizable, i.e. $B(u,x,t;y)$ has m real eigenvalues $\lambda_1(x,t;y), \ldots, \lambda_m(x,t;y)$ and its corresponding right eigenvectors $r_1(x,t;y), \ldots, r_m(x,t;y)$ form a basis of $\mathbb{R}^m$. In addition there are the special cases:

i) The system is called *symmetric hyperbolic* if $B_j(u,x,t)$ is symmetric for every feasible state $u \in \mathbb{R}^m$, $x \in \Omega$, $t \in \Sigma$ and $j = 1,\dots,m$.

ii) The system is called *strictly hyperbolic* if all m eigenvalues are distinct for every feasible state $u \in \mathbb{R}^m$, $x \in \Omega$, $t \in \Sigma$. $\square$

Note that the definition of hyperbolicity relies on the non-conservative form.

For the sake of brevity we omit theoretical discussion. Interested reader can find vast literature on hyperbolic PDEs, good start would be [2, Chapter 11].

In the subsequent Sections we will give three examples of hyperbolic systems: linear acoustics, shallow water equations, and Maxwell's equations. We formulate corresponding PDEs in conservative form (1) and provide properties necessary to develop numerical schemes.

2.1 Acoustic Wave Equation

The acoustic wave equation governs the propagation of acoustic waves through a material medium. Linearizing mass and momentum equations around the background state, dropping all higher-order terms in fluctuations and assuming *constant background pressure* results (without external sources) in

$$\partial_t\tilde{\rho} + \nabla\cdot(\bar{\rho}\tilde{v}) = 0, \qquad \text{(conservation of mass)} \tag{4a}$$
$$\partial_t(\bar{\rho}\tilde{v}) + \nabla\tilde{p} = 0, \qquad \text{(conservation of momentum).} \tag{4b}$$

It should be noted that it is the first order system that is derived from the physics and not the scalar second order wave equation, see also [4, § 2.7].

Conservative Form of Linear Acoustics We now consider the case that the speed of sound c is *piecewise constant* in fixed subdomains (e.g. due to temperature variations). Equation (4) is still valid in this case since only $\bar{p}$ being constant has been assumed. We conclude that pressure $\tilde{p}$ and normal momentum $\bar{\rho}\tilde{v}\cdot n$ are continuous at subdomain boundaries where c is discontinuous (this follows from integration by parts at enforcing continuity of mass and momentum at the subdomain boundaries).

Due to $\rho = p/c^2 = (\bar{p}+\tilde{p})/c^2 = \bar{p}/c^2+\tilde{p}/c^2 = \bar{\rho}+\tilde{\rho}$ also the background density $\bar{\rho}$ is piecewise constant. In case of varying speed of sound it is then more appropriate to use the conservative variables $(\tilde{\rho},\bar{\rho}\tilde{v}) = (\tilde{\rho},\tilde{q})$ resulting in the system

$$\partial_t\tilde{\rho} + \nabla\cdot\tilde{q} = 0, \tag{5a}$$
$$\partial_t\tilde{q} + \nabla(c^2\tilde{\rho}) = 0. \tag{5b}$$

At subdomain boundaries where c is discontinuous $c^2\tilde{\rho}$ (which is the pressure) and $\tilde{q}\cdot n$ are continuous.

Let us rewrite (5) at conservative hyperbolic system. Note that number of components $m = d+1$

$$\partial_t u(x,t) + \nabla\cdot F(u(x,t),x,t) = 0,$$

where

$$u = \begin{pmatrix} \varrho \\ q_1 \\ \vdots \\ q_d \end{pmatrix}, \quad F(u(x,t),x,t) = \begin{pmatrix} q_1 & q_2 & \dots & q_d \\ c^2\varrho & 0 & \dots & 0 \\ 0 & c^2\varrho & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & c^2\varrho \end{pmatrix} \in \mathbb{R}^{m\times d}.$$

Linear Acoustics problem has the two nonzero eigenvalues $\pm c$.

2.2 Shallow Water Equations

The Shallow Water model is the system of nonlinear hyperbolic PDEs, more precisely conservation law that describes the evolution of the height and the mean velocity of the fluid. It is widely used for predictions of flooding, dam-breaks, tsunamis or free oscillations of water. Another application of the Shallow Water Equations is long term simulations of the flow in rivers.

In 2d SWE reads as follows (number of components $m = d + 1 = 3$)

$$\partial_t \begin{pmatrix} h \\ u_1 h \\ u_2 h \end{pmatrix} + \nabla \cdot \begin{pmatrix} u_1 h & u_2 h \\ u_1^2 h + \frac{1}{2} g h^2 & u_1 u_2 h \\ u_1 u_2 h & u_2^2 h + \frac{1}{2} g h^2 \end{pmatrix} = 0, \tag{6}$$

Where $h > 0$ stands for water height, and $u = (u_1, u_2)$ is the velocity.

In conservative variables $q = (h, u_1 h, u_2 h)$ system (6) reads

$$\partial_t \begin{pmatrix} q_1 \\ q_2 \\ q_3 \end{pmatrix} + \nabla \cdot \begin{pmatrix} q_2 & q_3 \\ q_2^2/q_1 + \frac{1}{2} g q_1^2 & \frac{q_2 q_3}{q_1} \\ \frac{q_2 q_3}{q_1} & q_3^2/q_1 + \frac{1}{2} g q_1^2 \end{pmatrix} = 0. \tag{7}$$

Remarks:

- This modes have three distinct eigenvalues for $h \neq 0$ and therefore the two dimensional Shallow Water Equations are a strictly hyperbolic system for wet domains (i.e. $h > 0 \, \forall (x,t) \in \Omega \times \Sigma$).
- In this tutorial we also consider 1d SWE. One dimensional flux of SWE is 2x1 block of 2d flux matrix, however this analogy is not physical.
- SWE is a nonlinear system and in this tutorial works only with LLF flux.

2.3 Maxwell's Equations

Maxwell's equations are a set of PDS that underpin all electric, optical and radio technologies, including power generation, electric motors, wireless communication, cameras, televisions, computers etc. Maxwell's equations describe how electric and magnetic fields are generated by charges, currents, and changes of each other.

The Maxwell system is given by

$$\partial_t D - \nabla \times H = -J, \qquad \text{(Ampère)} \qquad (8a)$$
$$\partial_t B + \nabla \times E = 0, \qquad \text{(Faraday)} \qquad (8b)$$
$$\nabla \cdot D = \rho, \qquad \text{(Gauß)} \qquad (8c)$$
$$\nabla \cdot B = 0, \qquad \text{(Gauß for magnetism)} \qquad (8d)$$

together with the constitutive laws

$$D = \epsilon E, \qquad (\ \epsilon.\ \text{permittivity}) \qquad (9a)$$
$$B = \mu H, \qquad (\ \mu\text{: permeability}) \qquad (9b)$$
$$J = \sigma E + j, \qquad (\ \sigma\text{: conductivity}). \qquad (9c)$$

The following vector fields in $\mathbb{R}^3$ need to be determined:

Symbol	Name	Unit
B	magnetic flux density	$\frac{Vs}{m^2}$
H	magnetic field intensity	$\frac{A}{m}$
E	electric field intensity	$\frac{V}{m}$
D	displacement current density	$\frac{AS}{m^2}$

whereas the scalar charge density ρ and the current density j are prescribed.

The conditions (8c) and (8c) are needed only for the initial condition. The evolution in time is described by (8b) and (8a) only, see [3].

Since D and B are conserved quantities we formulate equations (8b) and (8a) in terms of D and B using the constitutive equations:

$$\partial_t D - \nabla \times \left(\frac{1}{\mu} B\right) + \frac{\sigma}{\epsilon} D = -j\,, \qquad (10a)$$
$$\partial_t B + \nabla \times \left(\frac{1}{\epsilon} D\right) = 0\,. \qquad (10b)$$

Writing out the curl operator $\nabla\times$ and defining the six component vector u, $(m = 2d)$, we obtain Maxwell system in conservative form

$$\partial_t u(x,t) + \nabla \cdot F(u(x,t),x,t) + g(u) = j,$$

where

$$u = \begin{pmatrix} D_1 \\ D_2 \\ D_3 \\ B_1 \\ B_2 \\ B_3 \end{pmatrix}, \quad F = \begin{pmatrix} 0 & -\mu^{-1}B_3 & \mu^{-1}B_2 \\ \mu^{-1}B_3 & 0 & -\mu^{-1}B_1 \\ -\mu^{-1}B_2 & \mu^{-1}B_1 & 0 \\ 0 & \epsilon^{-1}D_3 & -\epsilon^{-1}D_2 \\ -\epsilon^{-1}D_3 & 0 & \epsilon^{-1}D_1 \\ \epsilon^{-1}D_2 & -\epsilon^{-1}D_1 & 0 \end{pmatrix}, g = \begin{pmatrix} \sigma/\epsilon D_1 \\ \sigma/\epsilon D_2 \\ \sigma/\epsilon D_3 \\ 0 \\ 0 \\ 0 \end{pmatrix}, j = \begin{pmatrix} -j_1 \\ -j_2 \\ -j_3 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

It turns out that the eigenvalues of Maxwell system are 0, c and $-c$ each with multiplicity 2 and $c = 1/\sqrt{\epsilon\mu}$ the speed of light.

3 Discontinuous Galerkin Methods

In this section we present a numerical method to solve the original problem (1) which is repeated for convenience here. Let $u : \Omega \times \Sigma \to \mathbb{R}^m$ be the solution of the hyperbolic first-order system

$$\partial_t u(x,t) + \nabla \cdot F(u(x,t),x,t) = f(u(x,t),x,t), \qquad \text{in } U = \Omega \times \Sigma, \tag{11a}$$

$$u(x,t) = u_0(x), \qquad \text{at } t = 0, \tag{11b}$$

where $\Omega \subset \mathbb{R}^d$ is a bounded domain, $\Sigma = (t_0, t_0 + T)$ is a time interval of interest and $F(u,x,t) = [F_1(u,x,t), \ldots, F_n(u,x,t)]$ is the matrix valued flux function with columns $F_j(u,x,t)$.

3.1 Space Discretization with Discontinuous Galerkin

Finite element space

In order to achieve higher order we employ a finite element space with higher-order polynomials:

$$V_h^q = \left\{ v \in L^2(\Omega) \,:\, v|_e = p \circ \mu_e^{-1}, p \in \mathbb{P}^{q,d} \right\} \tag{12}$$

where the differentiable and invertible map

$$\mu_e : \hat{E} \to e$$

maps the reference element $\hat{E}$ to an element $e \in \mathcal{E}_h$ and the multivariate polynomials of degree q in d space dimensions are given by

$$\mathbb{P}^{q,d} = \begin{cases} \left\{ p \,:\, p(x_1, \ldots, x_d) = \sum\limits_{0 \le \|\alpha\|_1 \le q} c_\alpha x_1^{\alpha_1} \cdot \ldots \cdot x_d^{\alpha_d} \right\} & (\hat{E} \text{ simplex}), \\ \left\{ p \,:\, p(x_1, \ldots, x_d) = \sum\limits_{0 \le \|\alpha\|_\infty \le q} c_\alpha x_1^{\alpha_1} \cdot \ldots \cdot x_d^{\alpha_d} \right\} & (\hat{E} \text{ cube}), \end{cases}$$

depending on the type of element.

A function $v \in V_h^q$ is two-valued on an interior face $f \in \mathcal{F}_h^i$ and for $x \in f$ we denote by $v^-(x)$ the restriction from $e^-(f)$ and by $v^+(x)$ the restriction from $e^+(f)$. For any point $x \in f \in \mathcal{F}_h^i$ we define the jump

$$[\![v]\!](x) = v^-(x) - v^+(x) \tag{13}$$

and the average

$$\{v\}(x) = \frac{1}{2} v^-(x) - \frac{1}{2} v^+(x). \tag{14}$$

Discretization

For any test function v being piecewise smooth on the mesh $\mathcal{E}_h$ there holds

$$
\begin{aligned}
\int_\Omega \left[\partial_t u + \sum_{j=1}^{d} \partial_{x_j} F_j(u,x,t)\right] \cdot v \, dx = \\
&= d_t(u,v)_\Omega + \sum_{e\in\mathcal{E}_h} \sum_{j=1}^{d} \sum_{i=1}^{m} \int_e (\partial_{x_j} F_{i,j}(u,x,t))\, v_i \, dx \\
&= d_t(u,v)_\Omega + \sum_{e\in\mathcal{E}_h} \sum_{j=1}^{d} \sum_{i=1}^{m} \Big[- \int_e F_{i,j}(u,x,t)\, \partial_{x_j} v_i \, dx \\
&\qquad + \int_{\partial e} F_{i,j}(u,s,t) v_i n_j \, ds\Big] \\
&= d_t(u,v)_\Omega + \sum_{e\in\mathcal{E}_h} \left[- \int_e F(u,x,t) : \nabla v \, dx + \int_{\partial e} (F(u,s,t)n) \cdot v \, ds\right] \\
&= d_t(u,v)_\Omega - \sum_{e\in\mathcal{E}_h} \int_e F(u,x,t) : \nabla v \, dx \\
&\qquad + \sum_{f\in\mathcal{F}_h^i} \int_f [\![(F(u,s,t)n)\cdot v]\!] \, ds + \sum_{f\in\mathcal{F}_h^{\partial\Omega}} \int_f (F(u,s,t)n)\cdot v \, ds\,.
\end{aligned}
\tag{15}
$$

3.2 Numerical Fluxes

In general, a numerical flux is defined as a function

$$\Phi : \mathbb{R}^d \times U \times U \to \mathbb{R}, \tag{16}$$

which provides a single-valued approximation of $F \cdot n$ if we set the first argument to n, for detailed description we refer to [1].

In order to obtain physically correct approximations of the solution a numerical flux has to comply with the following properties:

Consistency of numerical fluxes

Definition 2. A numerical flux Φ is called consistent, if it is linear in its first argument, Lipschitz continuous with respect to the second and third argument and if for all $n \in \mathbb{R}^d, v \in U$ it holds

$$\Phi(n,v,v) = F(v)\cdot n. \tag{17}$$

That means it is exact if the solution is continuous between two neighboring elements.

Conservation of numerical fluxes

A numerical flux Φ needs to be conservative, i.e.

$$\Phi(n_1,u_1,u_2) + \Phi(n_2,u_2,u_1) = 0, \tag{18}$$

where u_1 and u_2 are the states on elements T_1, T_2 sharing an edge S and n_1 (resp. n_2) is the unit outer normal of T_1 (resp. T_2) on S, thus it holds $n_1 = -n_2$.

3.2.1 Local Lax-Friedrichs

For nonlinear problems a possible choice is the local Lax-Friedrichs flux,

$$\Phi(n, u^-, u^+) = \frac{1}{2}\left(F(u^-)\cdot n + F(u^+)\cdot n - \alpha(u^+ - u^-)\right), \tag{19}$$

where α is an estimate of the largest absolute value of the eigenvalues of the Jacobian $\partial_u F(u)\cdot n$ in a neighbourhood of the interface between u^+ and u^-.

For hyperbolic systems, we can calculate these eigenvalues. For example, the 1D SWE have the eigenvalues $\lambda_{1,2} = u \pm \sqrt{gh}$, depending on the physical variables u and h. Thus $|u| + \sqrt{gh}$ is the largest absolute value for one state and as we are interested in the maximum in a neighbourhood of S, we choose $\alpha = \max\limits_{u^-,u^+} |u| + \sqrt{gh}$.

3.2.2 Flux Vector Splitting

Here we only consider the linear constant coefficient case $F_j(u) = B_j u$. Then the normal flux is

$$F(u,x,t)n = \sum_{j=1}^{d} F_j(u)n_j = \sum_{j=1}^{d}(B_j u)n_j = \left(\sum_{j=1}^{d} n_j B_j\right) u = B_n u\,. \tag{20}$$

From Definition 1 the matrix $B_n = \left(\sum_{j=1}^{d} n_j B_j\right)$ is real diagonalizable for all $n \in \mathbb{R}^d$ and we recall that it implies that $B = RDR^{-1}$ with $D = \operatorname{diag}(\lambda_1, \ldots, \lambda_m)$ and regular R consisting columnwise of the eigenvectors $r_1, \ldots, r_m$. $w = R^{-1}u$ transforms a state u to characteristic variables in which the system is diagonal and where upwinding can be done in the usual way depending on the sign of the eigenvalues. Therefore we introduce the matrices

$$\begin{aligned} D^+ &= \operatorname{diag}(\max(0,\lambda_1), \ldots, \max(0,\lambda_m)),\\ D^- &= \operatorname{diag}(\min(0,\lambda_1), \ldots, \min(0,\lambda_m)), \end{aligned}$$

and

$$B^+ = RD^+R^{-1}, \qquad B^- = RD^-R^{-1}, \qquad B = B^+ + B^-\,. \tag{21}$$

With this we define the numerical flux at an interior point $x \in \mathcal{F}_h^i$ as

$$\Phi_U(u)(x) = B^+u^-(x) + B^-u^+(x)\,. \tag{22}$$

3.2.3 Variable Flux Vector Splitting

The coefficient matrix B may depend on position x. If this dependence is smooth one may put the hyperbolic system in nonconservative form an proceed as shown above. The case of discontinuous coefficient $B(x)$ deserves more thought. Consider the following one-dimensional Riemann problem

$$\partial_t u(x,t) + \partial_x(B(x)u(x,t)) = 0, \qquad (\text{in } \mathbb{R}\times\mathbb{R}^+) \tag{23a}$$

$$u(x,0) = \begin{cases} U_L & x \le 0 \\ U_R & x > 0 \end{cases}, \qquad (t=0), \tag{23b}$$

$$B(x) = \begin{cases} B_L & x \le 0 \\ B_R & x > 0 \end{cases}. \tag{23c}$$

Scalar Case For simplicity let us start with a single component $m = 1$. In order to determine what happens at the interface $x = 0$ we consider problem (23a) as two problems with an interface condition:

$$\partial_t u_L(x,t) + \partial_x(B_L u_L(x,t)) = 0, \qquad (\text{in } \mathbb{R}^-\times\mathbb{R}^+) \tag{24a}$$

$$u_L(x,0) = U_L, \tag{24b}$$

$$\partial_t u_R(x,t) + \partial_x(B_R u_R(x,t)) = 0, \qquad (\text{in } \mathbb{R}^+\times\mathbb{R}^+) \tag{24c}$$

$$u_R(x,0) = U_R, \tag{24d}$$

$$B_L u_L(0,t) = B_R u_R(0,t) \qquad (\text{flux continuity}). \tag{24e}$$

For arbitrary initial states flux continuity demands that B_L and B_R have the same sign: $B_L B_R > 0$. Then system (24a) can solved by the method of characteristics. Assume e.g. that $B_L, B_R > 0$, then

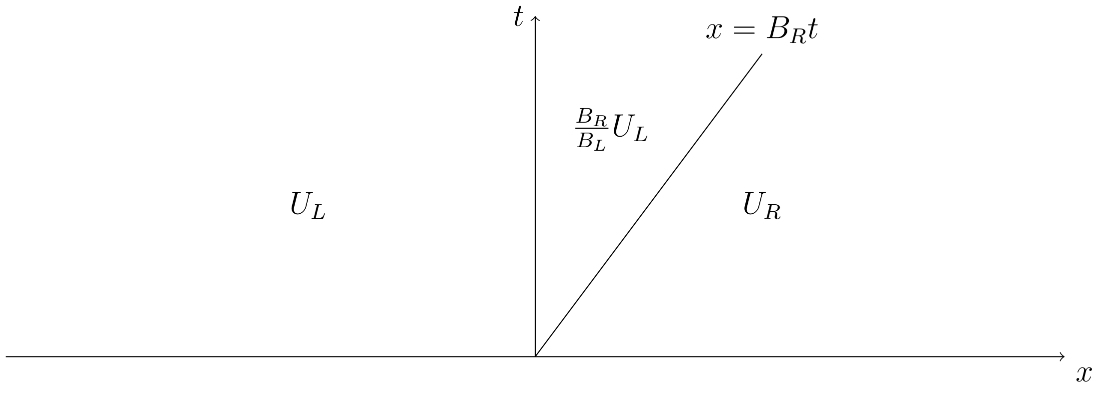
i) $x = 0$ is an outflow boundary for the left domain and $u_L(x,t) = U_L$ for $x \le 0$.

ii) $x = 0$ is an inflow boundary for the right domain. Flux continuity demands $B_L U_L = B_R u_R(0,t)$ and we get the boundary condition $u_R(0,t) = \frac{B_R}{B_L} U_L$.

iii) By the method of characteristic we obtain in the right domain:

$$u_R(x,t) = \begin{cases} \frac{B_R}{B_L} U_L & x - B_R t \le 0 \\ U_R & x - B_R t > 0 \end{cases}.$$

In the (x,t)-diagram this is:



System Case This is treated in the same way. However, since waves are going both ways across the interface the states left and right of the interface are determined by the solution of a linear system.

We define the states to the left and right of the interface

$$U_L^* = \lim_{x\to 0-} u_L(x,t), \qquad U_R^* = \lim_{x\to 0+} u_R(x,t), \qquad \text{(for any } t>0\text{)}.$$

Due to hyperbolicity, B_L and B_R are diagonalizable with eigenvalues λ_i^L, λ_i^R and eigenvectors r_i^L, r_i^R. The matrices R_L, R_R are formed by the eigenvectors and the diagonal matrices D_L, D_R contain the corresponding eigenvalues. As above we set $B_L^\pm = R_L D_L^\pm R_L^{-1}$, $B_R^\pm = R_R D_R^\pm R_R^{-1}$. By the transformation to characteristic variables we obtain the following representation of the interface states:

$$U_L^* = \sum_{\{i\,:\,\lambda_i^L \ge 0\}} r_i^L (R_L^{-1} U_L)_i + \sum_{\{i\,:\,\lambda_i^L < 0\}} r_i^L \alpha_i, \tag{25}$$

$$U_R^* = \sum_{\{i\,:\,\lambda_i^R \le 0\}} r_i^R (R_R^{-1} U_R)_i + \sum_{\{i\,:\,\lambda_i^R > 0\}} r_i^R \alpha_i. \tag{26}$$

The first sum takes into account the waves that reach the boundary from within in the respective domain. The second part describes the contribution coming from the boundary (the minus sign in the second line becomes obvious below). As a first assumption we put

$$\{i\,:\,\lambda_i^L < 0\} = \{i\,:\,\lambda_i^R < 0\} \quad \wedge \quad \{i\,:\,\lambda_i^L > 0\} = \{i\,:\,\lambda_i^R > 0\}, \tag{27}$$

i.e. the number of positive (negative) eigenvalues to the left and right coincides (and therefore also the number of zero eigenvalues) and positive and negative eigenvalues are numbered in the same way.

In order to determine the coefficients $\alpha \in \mathbb{R}^{I^*}$, $I^* = \{i\,:\,\lambda_i^L \neq 0\} \subseteq I = \{1,\dots,m\}$ we exploit flux continuity $B_L U_L^* = B_R U_R^*$. Further notation is needed to handle the case of zero eigenvalues when $m^* = |I^*| < m$. We introduce the "picking-out-matrix" $P \in \mathbb{R}^{I^* \times I}$ defined by

$$(Px)_j = (x)_j \qquad \forall j \in I^*.$$

Observing,

$$B_L U_L^* = \sum_{\{i\,:\,\lambda_i^L \ge 0\}} B_L r_i^L (R_L^{-1} U_L)_i + \sum_{\{i\,:\,\lambda_i^L < 0\}} B_L r_i^L \alpha_i = B_L^+ U_L + R_L D_L^- P^T \alpha,$$

$$B_R U_R^* = \sum_{\{i\,:\,\lambda_i^R \le 0\}} B_R r_i^R (R_R^{-1} U_R)_i + \sum_{\{i\,:\,\lambda_i^R > 0\}} B_R r_i^R \alpha_i = B_R^- U_R + R_R D_R^+ P^T \alpha.$$

we obtain

$$(R_R D_R^+ - R_L D_L^-) P^T \alpha = S\alpha = B_L^+ U_L - B_R^- U_R. \tag{28}$$

The linear system (28) has a unique solution if $S \in \mathbb{R}^{I \times I^*}$ has rank m^* and

$$\begin{aligned} &\operatorname{span}\left\{r_i^R : \lambda_i^R > 0\right\} + \operatorname{span}\left\{r_i^L : \lambda_i^R < 0\right\} = \\ &\qquad \operatorname{span}\left\{r_i^L : \lambda_i^R > 0\right\} + \operatorname{span}\left\{r_i^R : \lambda_i^R < 0\right\} \end{aligned} \tag{29}$$

and is then given by

$$\alpha = \left(S^T S\right)^{-1} S^T \left(B_L^+ U_L - B_R^- U_R\right). \tag{30}$$

The flux can then be computed from either side of the interface, e.g. from the left:

$$\begin{aligned} \hat{F}(U_L, U_R) &= B_L U_L^* = B_L^+ U_L + R_L D_L^- P^T \alpha \\ &= B_L^+ U_L + R_L D_L^- P^T \left(S^T S\right)^{-1} S^T \left(B_L^+ U_L - B_R^- U_R\right) \end{aligned} \tag{31}$$

For comparison consider the case of constant coefficients in this framework. Flux continuity then becomes

$$\begin{aligned} & BU_L^* = BU_R^* \\ \Leftrightarrow \sum_{\{i:\lambda_i>0\}} r_i\lambda_i(R^{-1}U_L)_i + \sum_{\{i:\lambda_i<0\}} r_i\lambda_i\alpha_i &= \sum_{\{i:\lambda_i<0\}} r_i\lambda_i(R^{-1}U_R)_i + \sum_{\{i:\lambda_i>0\}} r_i\lambda_i\alpha_i \end{aligned}$$

Since the r_i are linearly independent we must have

$$\alpha_i = (R^{-1}U_R)_i \text{ for } \lambda_i < 0, \quad \alpha_i = (R^{-1}U_L)_i \text{ for } \lambda_i > 0.$$

Inserting into one of both sides yields

$$\begin{aligned} \hat{F}(U_L, U_R) = BU_L^* &= \sum_{\{i:\lambda_i>0\}} r_i\lambda_i(R^{-1}U_L)_i + \sum_{\{i:\lambda_i<0\}} r_i\lambda_i\alpha_i \\ &= \sum_{\{i:\lambda_i>0\}} r_i\lambda_i(R^{-1}U_L)_i + \sum_{\{i:\lambda_i<0\}} r_i\lambda_i(R^{-1}U_R)_i \\ &= B^+U_L + B^-U_R. \end{aligned}$$

4 Realization in PDELab

The structure of the code is similar to previous tutorials. However we have separate files for different models, thus one must replace `[model]` with its name: `linearacoustics/maxwell/shallowwater`.

Source directory consists of the following files:

1) The ini-file `tutorial07-[model].ini` holds parameters read by various parts of the code which control the execution.

2) The problem file `[model]problem.hh` that describes the initial and boundary conditions for running the problem.

3) The model file `[model].hh` that provides which eigenvalues and matrix of the eigenvectors (rowwise) for problem.

4) Numerical flux `numericalflux.hh`

5) The main file `tutorial07-[model].cc` includes the necessary C++, DUNE and PDELab header files and contains the `main` function where the execution starts. The purpose of the `main` function is to instantiate DUNE grid objects and call the `driver` function.

6) File `driver.hh` instantiates the necessary PDELab classes for solving a instationary problem and finally solves the problem.

7) File `hyperbolicdg.hh` contains the local operator classes `DGHyperbolicSpatialOperator` and `DGHyperbolicTemporalOperator` realizing the spatial and temporal residual forms.

4.1 Ini-File

The ini-file contains the usual sections for `[grid]`. The `[fem]` section is the same as in tutorial 01 and allows to set the polynomial degree, temporal integration order and the time step size. In `[problem]` section we set final time and `[output]` defines filename, subsampling, and every (timestep count to save a solution).

4.2 Problem file `linearacousticsproblem.hh`

In the problem we define all the properties of the problem we want to solve. Here we explain what we mean by problem (not to confuse with model which is the system of equations). We can define following problem specific:

Material

```
//! material
// this function is used to decide if we work
// with discontinous coefficient case
template<typename E, typename X>
int material (const E& e, const X& x) const
{
  auto xglobal = e.geometry().center();
  if (xglobal[0]>1.625)
    return 1;
  else
    return 2;
}
```

Speed of sound (can be discontinuous)

```
//! speed of sound
template<typename E, typename X>
NUMBER c (const E& e, const X& x) const
{
  auto xglobal = e.geometry().center();
  if (xglobal[0]>1.625)
    return 0.33333;
  else
    return 1.0;
}
```

Boundary condition (reflective)

```
//! Boundary condition value - reflecting bc
template<typename I, typename X, typename R>
```

```
Range g (const I& is, const X& x, const R& s) const
{
  Range u(0.0);
  u[0] =  s[0];

  for (size_t i=0; i<dim; i++)
    u[i+1] = -s[i+1];

  return u;
}
```

Right hand side

```
//! right hand side
template<typename E, typename X>
Range q (const E& e, const X& x) const
{
  Range rhs(0.0);
  return rhs;
}
```

Initial value

```
//! initial value -> the same as tutorial04
template<typename E, typename X>
Range u0 (const E& e, const X& x) const
{
  X xglobal = e.geometry().global(x);
  Range u(0.0);
  for (int i=0; i<dim; i++)
    u[0] += (xglobal[i]-0.375)*(xglobal[i]-0.375);
  u[0] = std::max(0.0,1.0-8.0*sqrt(u[0]));

  return u;
}
```

Remark 3. The material property *speed of sound* is specific to linearacoustic problem. In the case of Maxwell system we define permabiliy. User must provide material function that is used to decide if we work in discontinuous coefficient case.

4.3 Model file `[model].hh`

Model contains all necessary information about your system that later will be used to determine numerical flux.

```
template<typename E, typename X, typename T2, typename T3>
void eigenvectors (const E& e, const X& x,
                   const Dune::FieldVector<T2,dim>& n,
                   Dune::FieldMatrix<T3,m,m>& RT) const
{
  auto c = problem.c(e,x);
```

```
    //TODO find a way to write eigenvectors independently of dim
    if (dim == 1) {
      RT[0][0] =  1; RT[1][0] = c*n[0];
      RT[0][1] = -1; RT[1][1] = c*n[0];
    }
    if (dim == 2) {
      RT[0][0] =  1.0/c;  RT[0][1] = -1.0/c;  RT[0][2] = 0.0;
      RT[1][0] =  n[0]; RT[1][1] = n[0];  RT[1][2] = -n[1];
      RT[2][0] =  n[1]; RT[2][1] = n[1];  RT[2][2] = n[0];
    }
  }
```

```
  template<typename E, typename X, typename RF>
  void diagonal (const E& e, const X& x,
                 Dune::FieldMatrix<RF,m,m>& D) const
  {
    auto c = problem.c(e,x);

    for (size_t i=0; i<m; i++)
      for (size_t j=0; j<m; j++)
        D[i][j] = 0.0;
    D[0][0] = c;
    D[1][1] = -c ;
  }
```

```
  //Flux function
  template<typename E, typename X, typename RF>
  void flux (const E& e, const X& x,
             const Dune::FieldVector<RF,m>& u,
             Dune::FieldMatrix<RF,m,dim>& F) const
  {
    auto c = problem.c(e,x);

    for (size_t i=0; i<dim; i++) {
      F[0][i] = u[i+1];
      F[i+1][i] = c*c*u[0];
    }

  }
```

Remark 4. The order of eigenvalues is important, the implementation of VariableFluxVectorSplitting flux requires consequently: positive, negative and zero eigenvalues. Implementation of the SWE example is based on class specialisation for one and two dimensions. Note, also that information contained in model must agree with the numerical flux you are willing to use.

4.4 Numerical flux `numericalflux.hh`

This class implements different numerical fluxes. Currently including: Local Lax-Friedrisch and Flux Vector Splitting (also in variable coefficient case).

The implementation of Flux Vector Splitting method reads as follows:

```
template<typename E, typename X>
void numericalFlux(const E& inside, const X& x_inside,
                   const E& outside, const X& x_outside,
                   const Dune::FieldVector<DF,dim> n_F,
                   const Dune::FieldVector<RF,m>& u_s,
                   const Dune::FieldVector<RF,m>& u_n,
                   Dune::FieldVector<RF,m>& f) const
{
  Dune::FieldMatrix<DF,m,m> D(0.0);
  // fetch eigenvalues
  model().diagonal(inside,x_inside,D);

  Dune::FieldMatrix<DF,m,m> Dplus(0.0);
  Dune::FieldMatrix<DF,m,m> Dminus(0.0);

  for (size_t i =0 ; i<m;i++)
    (D[i][i] > 0)
      ? Dplus[i][i] = D[i][i]
      : Dminus[i][i] = D[i][i];

  // fetch eigenvectors
  Dune::FieldMatrix<DF,m,m> Rot;
  model().eigenvectors(inside,x_inside,n_F,Rot);

  // compute B+ = RD+R^-1 and B- = RD-R^-1
  Dune::FieldMatrix<DF,m,m> Bplus(Rot);
  Dune::FieldMatrix<DF,m,m> Bminus(Rot);

  //multiply by D+-
  Bplus.rightmultiply(Dplus);
  Bminus.rightmultiply(Dminus);

  //multiply by R^-1
  Rot.invert();
  Bplus.rightmultiply(Rot);
  Bminus.rightmultiply(Rot);

  // Compute numerical flux at  the integration point
  f = 0.0;
  // f = Bplus*u_s + Bminus*u_n
  Bplus.umv(u_s,f);
  Bminus.umv(u_n,f);
}
```

4.5 Function `main`

The `main` function is very similar to the one in previous tutorials. However there are differences specific to hyperbolic solver.

We include our hyperbolic model and problem to solve

```
//Linear Acoustics
#include"linearacoustics.hh"//model
#include"linearacousticsproblem.hh"
#include"numericalflux.hh"
```

Calling Problem and Model constructors and choose proper numerical flux

```
//create problem (setting)
using PROBLEM = Problem<GV,GV::Grid::ctype>;
PROBLEM problem;

//create model on a given setting
using MODEL = Model<PROBLEM >;
MODEL model(problem);

//create numerical flux
using NUMFLUX = VariableFluxVectorSplitting<MODEL>;
NUMFLUX numflux(model);
```

Build FEM space and call driver

```
using FEM = Dune::PDELab::QkDGLocalFiniteElementMap
  <GV::Grid::ctype, double, 2, dim,
   Dune::PDELab::QkDGBasisPolynomial::legendre>;
FEM fem;
driver<GV,FEM, NUMFLUX>(gv,fem,numflux,ptree);
```

Not that we pass `numflux` to `driver` but indeed it contains both `model` and `problem`. Dependencies reads `numflux` <- `model` <- `problem`.

4.6 Function `driver`

The `driver` function gets a grid view, a finite element map and a parameter tree and its purpose is to solve the problem on the given mesh.

There are several changes now in the driver due to the system of PDEs.

At first we extract range field, dimension and number of components

```
  // Choose domain and range field type
  using RF = typename NUMFLUX::RF; // type for computations
  static constexpr int dim = NUMFLUX::dim;
  static constexpr int m = NUMFLUX::m; //number of components
```

Now we can set up the grid function space using the given finite element map and set up the product space containing two components. This is done by the following code section:

```
//block size for a component deducted by pdelab
using VBE0 = Dune::PDELab::ISTL::VectorBackend<>;

using VBE = Dune::PDELab::ISTL::VectorBackend<
  Dune::PDELab::ISTL::Blocking::fixed>;
using OrderingTag = Dune::PDELab::EntityBlockedOrderingTag;

using GFSDG = Dune::PDELab::GridFunctionSpace<GV,FEMDG,CON,VBE0>;
GFSDG gfsdg(gv,femdg);

using GFS = Dune::PDELab::PowerGridFunctionSpace<
  GFSDG,m,VBE,OrderingTag>;
GFS gfs(gfsdg);

typedef typename GFS::template ConstraintsContainer<RF>::Type C;
C cg;
gfs.update(); // initializing the gfs
std::cout << "degrees␣of␣freedom:␣" << gfs.globalSize() << std::endl;
```

Here we use PowerGridFunctionSpace which creates a product of a compile-time given number (m here) of *identical* function spaces (GFSDG here).

An important aspect of product spaces is the ordering of the corresponding degrees of freedom. Often the solvers need to exploit an underlying block structure of the matrices. This works in two stages: An ordering has first to be specified when creating product spaces which is then subsequently exploited in the backend. Here we use the EntityBlockedOrderingTag to specify that all degrees of freedom related to a geometric entity should be numbered consecutively in the coefficient vector.

Next, we create local operators

```
// Make instationary grid operator
using LOP = Dune::PDELab::DGHyperbolicSpatialOperator<NUMFLUX,FEMDG>;
LOP lop(numflux);
using TLOP = Dune::PDELab::DGHyperbolicTemporalOperator<NUMFLUX,FEMDG>;
TLOP tlop(numflux);
```

Pick time stepping scheme

```
// select and prepare time-stepping scheme
int torder = ptree.get("fem.torder",(int)1);

Dune::PDELab::ExplicitEulerParameter<RF> method1;
Dune::PDELab::HeunParameter<RF> method2;
Dune::PDELab::Shu3Parameter<RF> method3;
Dune::PDELab::RK4Parameter<RF> method4;
Dune::PDELab::TimeSteppingParameterInterface<RF> *method;

if (torder==1) {
  method=&method1;
  std::cout << "setting␣explicit␣Euler" << std::endl;}
if (torder==2) {
  method=&method2;
  std::cout << "setting␣Heun" << std::endl;}
if (torder==3) {
  method=&method3;
  std::cout << "setting␣Shu␣3" << std::endl;}
if (torder==4) {
  method=&method4;
  std::cout << "setting␣RK4" << std::endl;}
if (torder<1||torder>4)
  std::cout << "torder␣should␣be␣in␣[1,4]" << std::endl;
```

The rest of the driver is the same as for tutorial 03 except that a linear solver is used instead of Newton's method.

```
  //typedef Dune::PDELab::ISTLBackend_SEQ_UMFPack LS;
  //LS ls(false);

  // time-stepping
  Dune::PDELab::ExplicitOneStepMethod<RF,IGO,LS,V,V> osm(*method,igo,ls);
  osm.setVerbosityLevel(2);
```

4.7 Spatial Local Operator

`alpha_volume` method

The method `alpha_volume` has the *same* interface as in previous exercises, however the trial and test function spaces `LFSU` and `LFSV` now reflect the component structure of the global function space, i.e. they consist of $m-$components.

Important notice: Here we assume that trial and test space are identical (up to constraints) and that also both components are identical!

In the loop over the quadrature points we need to evaluate flux and compute residuum

```
Dune::FieldMatrix<RF,m,dim> F;
numflux.model().flux(eg,qp,u,F);

// integrate
auto factor = ip.weight() * geo.integrationElement(qp);
for (size_t k=0; k<dgspace.size(); k++)
  {
    // - F(u) \grad phi
    for (size_t i=0; i<m; i++)
      for (size_t j=0; j<dim; j++)
        r.accumulate(lfsv.child(i),k,-F[i][j]*gradphi[k][j]*factor);
  }
```

alpha_skeleton **method**

In the alpha_skeleton method we evaluate numerical flux and its jump on the internal boundary

```
for (const auto& ip : quadratureRule(geo,intorder))
  {
    const auto qp = ip.position();
    // Position of quadrature point in local coordinates of elements
    auto iplocal_s = geo_in_inside.global(qp);
    auto iplocal_n = geo_in_outside.global(qp);

    // Evaluate basis functions
    auto& phi_s = cache[order_s].evaluateFunction(
      iplocal_s,dgspace_s.finiteElement().localBasis());
    auto& phi_n = cache[order_n].evaluateFunction(
      iplocal_n,dgspace_n.finiteElement().localBasis());

    // Evaluate u from inside and outside
    u_s = 0.0;
    for (size_t i=0; i<m; i++) // for all components
      for (size_t k=0; k<dgspace_s.size(); k++)
        u_s[i] += x_s(lfsv_s.child(i),k)*phi_s[k];
    u_n = 0.0;
    for (size_t i=0; i<m; i++) // for all components
      for (size_t k=0; k<dgspace_n.size(); k++)
        u_n[i] += x_n(lfsv_n.child(i),k)*phi_n[k];

    // Compute numerical flux at  the integration point
    numflux.numericalFlux(cell_inside, iplocal_s,
                          cell_outside, iplocal_n,
                          ig.unitOuterNormal(qp),u_s,u_n,f);

    // Integrate
    auto factor = ip.weight() * geo.integrationElement(qp);
    // loop over all vector-valued basis functions
    for (size_t k=0; k<dgspace_s.size(); k++)
      for (size_t i=0; i<m; i++) // loop over all components
        r_s.accumulate(lfsv_s.child(i),k, f[i]*phi_s[k]*factor);
    // loop over all vector-valued basis functions
    for (size_t k=0; k<dgspace_n.size(); k++)
      for (size_t i=0; i<m; i++) // loop over all components
        r_n.accumulate(lfsv_n.child(i),k, - f[i]*phi_n[k]*factor);
  }
```

alpha_boundary **method**

The same is done in the alpha_boundary however the outside value is determined via problem setup

```
// Evaluate boundary condition
```

```
Dune::FieldVector<RF,m> u_n(
  numflux.model().problem.g(ig.intersection(),qp,u_s));

// Compute numerical flux at integration point
numflux.numericalFlux(cell_inside, iplocal_s,
                      cell_inside, iplocal_s,
                      ig.unitOuterNormal(qp),u_s,u_n,f);

// Integrate
auto factor = ip.weight() * geo.integrationElement(qp);
// loop over all vector-valued (!) basis functions
// (with identical components)
for (size_t k=0; k<dgspace_s.size(); k++)
  for (size_t i=0; i<m; i++) // loop over all components
    r_s.accumulate(lfsv_s.child(i),k, f[i]*phi_s[k]*factor);
```

4.8 Running the Example

Following models and fluxes are implemented:

Model		**Numerical Flux**	**Dimension**	**Components**
Linear Acounstics	linear	FVS, LLF	$d = 1, 2$	$m = d + 1$
Maxwall	linear	FVS, LLF	$d = 3$	$m = 6$
Shallow Water	non-linear	LLF	$d = 1, 2$	$m = d + 1$

References

[1] D.A. Di Pietro and A. Ern. *Mathematical Aspects of Discontinuous Galerkin Methods*. Mathématiques et Applications. Springer Berlin Heidelberg, 2011.

[2] L. C. Evans. *Partial Differential Equations*. American Mathematical Society, 2nd edition, 2010.

[3] J. Jin. *The Finite Element Method in Electromagnetics*. John Wiley & Sons, 2. edition, 2002.

[4] R. J. Leveque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press, 2002.