DG-FEM for PDE’s
Lecture 2

Jan S Hesthaven
Brown University
Jan.Hesthaven@Brown.edu
A brief overview of what’s to come

- Lecture 1: Introduction and DG in 1D
- Lecture 2: Implementation and numerical aspects
- Lecture 3: Insight through theory
- Lecture 4: Nonlinear problems
- Lecture 5: Mesh generation and 2D DG-FEM
- Lecture 6: Implementation and applications
- Lecture 7: Higher order/Global problems
- Lecture 8: 3D and advanced topics
Let’s recall

We already know a lot about the basic DG-FEM

- **Stability** is provided by carefully choosing the numerical flux.
- **Accuracy** appear to be given by the local solution representation.
- We can utilize major advances on **monotone schemes** to design fluxes.
- The scheme generalizes with very few changes to very general **systems of conservation laws**.

At least in principle -- but how we do it in practice?
Recall the basic formulation

We introduce the local notation

\[ \mathcal{M}_i^k = \int_{D_k} \ell_i^k(x) \ell_j^k(x) \, dx, \quad \mathcal{S}_i^k = \int_{D_k} \ell_i^k(x) \frac{d\ell_j^k}{dx} \, dx. \]

to obtain the semi-discrete forms of DG-FEM

**Weak:**

\[ \mathcal{M}_k \frac{du_h^k}{dt} - (\mathcal{S}_k^T) f_h^k - \mathcal{M}_k g_h^k = -f^* (x^{k+1}) \ell(x^{k+1}) + f^* (x^k) \ell(x^k) \]

**Strong:**

\[ \mathcal{M}_k \frac{du_h^k}{dt} + \mathcal{S}_k f_h^k - \mathcal{M}_k g_h^k = (f_h^k (x^{k+1}) - f^* (x^{k+1})) \ell(x^{k+1}) - (f_h^k (x^k) - f^* (x^k)) \ell(x^k). \]
.. but how do we bring this to life?

We should seek

- A flexible and generic framework for solving 1D problems using DG-FEM
- Easy maintenance and setup
- Separation of grids, discretization, and physics
- Easy implementation

Let's see if we can achieve this
The local approximation

We recall that we have

$$\Omega \simeq \Omega_h = \bigcup_{k=1}^{K} D^k$$

where the computational domain is

![Diagram](attachment:image.png)

Consider a one-dimensional domain defined on $x \in [L, R]$. The shape of the local elements can in principle be of any shape, however, in practice we mostly consider $d$-dimensional simplexes (e.g. triangles in two dimensions).
The local approximation

On each element we have

\[ x \in D^k = [x^k_l, x^k_r] : u^k_h(x, t) = \sum_{n=1}^{N_p} \hat{u}^k_n(t) \psi_n(x) = \sum_{i=1}^{N_p} u^k_h(x^k_i, t) \ell^k_i(x). \]

using either a modal or a nodal representation

We introduce an affine mapping as

\[ x \in D^k : x(r) = x^k_l + \frac{1+r}{2} h^k, \quad h^k = x^k_r - x^k_l \]
The local approximation

**Modal** representation is based on Legendre polynomials

\[
\psi_n(r) = \tilde{P}_{n-1}(r) = \frac{P_{n-1}(r)}{\sqrt{\gamma_{n-1}}}, \quad \gamma_n = \frac{2}{2n+1} \quad \int_{-1}^{1} \tilde{P}_i(r)\tilde{P}_j(r)dr = \delta_{ij}
\]

\[
\psi_1(r) = 1, \quad \psi_2(r) = r, \quad \psi_3(r) = \sqrt{1-r^2}, \quad \psi_4(r) = -r\sqrt{1-r^2}, \quad \psi_5(r) = \sqrt{1-r^2}(1-r), \quad \psi_6(r) = r\sqrt{1-r^2}(1-r)
\]
The local approximation

Nodal representation based on Lagrange polynomials

$$l^k_i(x) = \delta_{ij}, \quad \delta_{ij} = \begin{cases} 0, & i \neq j \\ 1, & i = j \end{cases}$$
The local approximation

These two representations are connected as

\[ u(r) \simeq u_h(r) = \sum_{n=1}^{N_p} \hat{u}_n \tilde{P}_{n-1}(r) = \sum_{i=1}^{N_p} u(r_i) \ell_i(r), \]

\[ u = \mathbf{V}\hat{u}, \quad \mathbf{V}^T \ell(r) = \tilde{P}(r), \quad \mathbf{V}_{ij} = \tilde{P}_j(r_i). \]

**Conditioning of \( V \) is determined by choice of points**

**Optimal points:** Legendre Gauss Lobatto points

**Important for robustness:**
Orthogonal local basis and good points for interpolation
The local approximation

To implement a typical semi-discrete scheme

\[ \mathcal{M}_i^k \frac{d}{dt} \mathbf{u}_h^k + S_i^k(au_h^k) = \left[ \ell_i^k(x)(au_h^k - (au_h)^*) \right]_{x_r^k} \]

We need several local operators

\[ \mathcal{M}_{ij}^k = \int_{D^k} \ell_i^k(x) \ell_j^k(x) \, dx, \quad S_{ij}^k = \int_{D^k} \ell_i^k(x) \frac{d\ell_j^k}{dx} \, dx. \]

Consider first the local mass-matrix

\[ \mathcal{M}_{ij}^k = \int_{x_i^k}^{x_r^k} \ell_i^k(x) \ell_j^k(x) \, dx = \frac{h_i^k}{2} \int_{-1}^{1} \ell_i(r) \ell_j(r) \, dr = \frac{h_i^k}{2} (\ell_i, \ell_j)_I = \frac{h_i^k}{2} \mathcal{M}_{ij}, \]
The local approximation

Recall first

\[ u = \mathcal{V} \hat{u} \]

We also have

\[ u^T l(r) = \hat{u}^T \psi(r) \]

This directly yields

\[ u^T l(r) = \hat{u}^T \psi(r) \]
\[ \Rightarrow (\mathcal{V} \hat{u})^T l(r) = \hat{u}^T \psi(r) \]
\[ \Rightarrow \hat{u}^T \mathcal{V}^T l(r) = \hat{u}^T \psi(r) \]
\[ \Rightarrow \mathcal{V}^T l(r) = \psi(r) \]
The local approximation

Using this, we recover

\[
M_{ij} = \int_{-1}^{1} \sum_{n=1}^{N_p} (\mathbf{v}^T)^{-1}_{in} \tilde{P}_{n-1}(r) \sum_{m=1}^{N_p} (\mathbf{v}^T)^{-1}_{jm} \tilde{P}_{m-1}(r) \, dr
\]

\[
= \sum_{n=1}^{N_p} \sum_{m=1}^{N_p} (\mathbf{v}^T)^{-1}_{in} (\mathbf{v}^T)^{-1}_{jm} (\tilde{P}_{n-1}, \tilde{P}_{m-1}) = \sum_{n=1}^{N_p} (\mathbf{v}^T)^{-1}_{in} (\mathbf{v}^T)^{-1}_{jn},
\]

\[
M^k = \frac{h^k}{2} M = \frac{h^k}{2} (\mathbf{v} \mathbf{v}^T)^{-1}.
\]

Consider now the local stiffness matrix

\[
S^k_{ij} = \int_{x_l^k}^{x_r^k} \ell^k_i(x) \frac{d\ell^k_j(x)}{dx} \, dx = \int_{-1}^{1} \ell_i(r) \frac{d\ell_j(r)}{dr} \, dr = S_{ij}.
\]
The local approximation

Define first the nodal differentiation matrix

\[ D_{r,(i,j)} = \left. \frac{d\ell_j}{dr} \right|_{r_i} \]  

Consider

\[ (MD_r)_{(i,j)} = \sum_{n=1}^{N_p} M_{in} D_{r,(n,j)} = \sum_{n=1}^{N_p} \int_{-1}^{1} \ell_i(r) \ell_n(r) \left. \frac{d\ell_j}{dr} \right|_{r_n} dr \]

\[ = \int_{-1}^{1} \ell_i(r) \sum_{n=1}^{N_p} \left. \frac{d\ell_j}{dr} \right|_{r_n} \ell_n(r) dr = \int_{-1}^{1} \ell_i(r) \frac{d\ell_j(r)}{dr} dr = S_{ij}. \]

\[ MD_r = S. \]

\[ \nabla^T \ell(r) = \tilde{P}(r) \Rightarrow \nabla^T \frac{d}{dr} \ell(r) = \frac{d}{dr} \tilde{P}(r), \]

\[ \nabla^T D_r^T = (\nabla_r)^T, \quad \nabla_{r,(i,j)} = \left. \frac{d\tilde{P}_j}{dr} \right|_{r_i}. \]
The local approximation

We will also need a boundary operator like

\[ \int_{-1}^{1} \hat{n} \cdot (u_h - u^*) \ell_i(r) \, dr = (u_h - u^*)|_{r_{N_p}} e_{N_p} - (u_h - u^*)|_{r_{1}} e_{1}, \]

A list of essential functions to do local operations are

- JacobiP: Evaluate normalized Jacobi polynomial
- GradJacobiP: Evaluate derivative of Jacobi polynomial
- JacobiGQ: Compute Gauss points and weights for use in quadrature
- JacobiGL: Compute the Legendre-Gauss-Lobatto (LGL) nodes
- Vandermonde1D: Compute \( V \)
- GradVandermonde1D: Compute \( V_r \)
- Dmatrix1D: Compute \( D_r \)

These are entirely problem independent
Consider

\[ u(x) = \exp(\sin(\pi x)), \ x \in [-1, 1], \]

**Spectral convergence**

\[ u^{(0)}(x) = \begin{cases} -\cos(\pi x), & -1 \leq x < 0 \\ \cos(\pi x), & 0 \leq x \leq 1, \end{cases} \]

\[ \frac{du^{(i+1)}}{dx} = u^{(i)}, \ i = 0, 1, 2, 3 \ldots \]

\[ \left\| \frac{du^{(i)}}{dx} - D_r u^{(i)} \right\|_\Omega \propto N^{1/2-i}. \]

**Accuracy ~ regularity**
Getting the grid and metric together

From a grid generator we will get

- A numbered set of vertices - VX
- A set of elements, each of two vertices - EToV(K,2)

From this we need

- Grid
- Element sizes
- Element normals and Jacobians
- Element connectivity
  - element-to-element
  - edge-to-edge
Getting the grid and metric together

We first compute the grid

\[
\begin{align*}
\text{va} &= \text{EToV}(:,1)' ; \quad \text{vb} = \text{EToV}(:,2)' , \\
\text{x} &= \text{ones}(\text{Np},1)*\text{VX}(\text{va}) + 0.5*(r+1)*(\text{VX}(\text{vb})-\text{VX}(\text{va}));
\end{align*}
\]

and the local metric

\[
\begin{align*}
\mathbf{r}_x &= \frac{1}{x_r} = \frac{1}{J} = \frac{1}{\mathbf{D}_r \mathbf{x}},
\end{align*}
\]

The local normals are simple: \( \hat{n}_l = -1, \quad \hat{n}_r = 1 \)

Outward point face normals: \( \mathbf{nx} = [-1 \ 1; \ -1 \ 1]; \)
Getting the grid and metric together

We shall understand how to get the connectivity through an example - K=4, 5 vertices

\[ \mathbf{VX} = [0.0, 0.5, 1.5, 3.0, 2.5], \quad \mathbf{EToV} = \begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 5 \\ 5 & 4 \end{bmatrix}, \]

Form the companion matrix - FaceToVertex

\[ \mathbf{FToV} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \]

Element 1

Element 2

Element 3

Element 4

Faces
Getting the grid and metric together

Consider

\[(F_{ToV})(F_{ToV})^T = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1
\end{bmatrix}\]

From this all connectivity information can be found

- **Edge 1** only self connects -- it is a boundary!
- **Edge 2** connects to Edge 3
- **Edge 3** connects to Edge 2
- etc
Getting the grid and metric together

This is used to built index maps

- $vmapM$ -- such that $u(vmapM) = \text{interior } u$
- $vmapP$ -- such that $u(vmapP) = \text{exterior } u$
- $vmapB$ -- such that $u(vmapB) = \text{boundary } u$

Global node numbering defined:

$vmapM = [1 4 5 8];$
$vmapP = [1 5 4 8];$

Face node numbering defined:

$mapM = [1 2 3 4];$
$mapP = [1 3 2 4];$
$mapI = [1];$
$mapO = [4];$
The global scheme

To connect all elements and compute the metric we need routines like

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Globals1D</td>
<td>Define list of globals variables</td>
</tr>
<tr>
<td>MeshGen1D</td>
<td>Generates a simple equidistant grid with K elements</td>
</tr>
<tr>
<td>Startup1D</td>
<td>Main script for pre-processing</td>
</tr>
<tr>
<td>BuildMaps1D</td>
<td>Automatically create index maps from conn. and bc tables</td>
</tr>
<tr>
<td>Normals1D</td>
<td>Compute outward pointing normals at elements faces</td>
</tr>
<tr>
<td>Connect1D</td>
<td>Build global connectivity arrays for 1D grid</td>
</tr>
<tr>
<td>GeometricFactors1D</td>
<td>Compute metrics of local mappings</td>
</tr>
<tr>
<td>Lift1D</td>
<td>Compute surface integral term in DG formulation</td>
</tr>
</tbody>
</table>

These are also problem independent!
Putting it together - an example

Consider again the wave equation

\[
\frac{\partial u}{\partial t} + 2\pi \frac{\partial u}{\partial x} = 0, \quad x \in [0, 2\pi],
\]
\[
u(x, 0) = \sin(x),
\]
\[
u(0, t) = -\sin(2\pi t).
\]

Following the recipe we have developed, we get

\[
\mathcal{M}^k \frac{d u_h^k}{dt} + 2\pi S u_h^k = \left[ \ell^k(x)(2\pi u_h^k - (2\pi u)^*) \right]_{x_l^k},
\]
\[
\frac{d u_h^k}{dt} + 2\pi (\mathcal{M}^k)^{-1} S u_h^k = (\mathcal{M}^k)^{-1} \left[ \ell^k(x)(2\pi u_h^k - (2\pi u)^*) \right]_{x_l^k},
\]

and the stable flux

\[
(2\pi u)^* = \left\{ \{2\pi u\} \right\} + 2\pi \frac{1 - \alpha}{2} \llbracket u \rrbracket, \quad 0 \leq \alpha \leq 1
\]
Putting it together

We always formulate it through 3 parts

AdvecDriver1D Matlab main function for solving the 1D advection equation.
Advec1D(u,FinalTime) Matlab function for time integration of the semidiscrete PDE.
AdvecRHS1D(u,time,a) Matlab function defining right hand side of semidiscrete PDE
Putting it together - an example

3.5 Putting it all together

It is easy to see that the exact solution is

\[ u(x, t) = \sin(x - 2\pi t) \]

Following the developments in Chapter 2, we assume the local solution is well approximated as

\[ u_h(x, t) = \sum_{i=1}^{N_p} u_{k_i} \ell_{k_i}(x) \]

and the combination of these local solutions as the approximation to the global solution. This yields the local semidiscrete scheme

\[ M_k \frac{du_k}{dt} + 2\pi S u_k = [\ell_k(x)(2\pi u_k - (2\pi u^*)_{\text{ref}})]x_k r x_k \]

or

\[ \frac{du_k}{dt} + 2\pi (M_k - 1) S u_k = (M_k - 1)[\ell_k(x)(2\pi u_k - (2\pi u^*)_{\text{ref}})]x_k r x_k \]

The analysis in Example 2.3 shows that choosing the flux as

\[ (2\pi u)^* = \begin{cases} 2\pi u & 0 \leq \alpha \leq 1 \\ \end{cases} \]

results in a stable scheme.

An implementation of this is shown in AdvecRHS1D.m. The parameter \( \alpha \) can be used to adjust the flux; for example \( \alpha = 1 \) is a central flux and \( \alpha = 0 \) reflects an upwind flux.

---

### AdvecRHS1D.m

```matlab
function [rhsu] = AdvecRHS1D(u,time, a)

% function [rhsu] = AdvecRHS1D(u,time)
% Purpose : Evaluate RHS flux in 1D advection

Globals1D;

% form field differences at faces
alpha=1;
du = zeros(Nfp*Nfaces,K);
du(:) = (u(vmapM)-u(vmapP)).*(a*nx(:)-(1-alpha)*abs(a*nx(:)))/2;

% impose boundary condition at x=0
uin = -sin(a*time);
du (mapI) = (u(vmapI)- uin ).*(a*nx(mapI)-(1-alpha)*abs(a*nx(mapI)))/2;
du (mapO) = 0;

% compute right hand sides of the semi-discrete PDE
rhsu = -a*rx.*(Dr*u) + LIFT*(Fscale.*(du));
return
```

---

Friday, July 23, 2010
Temporal integration

As we have already discussed we use RK methods

\[ \frac{d\mathbf{u}_h}{dt} = \mathcal{L}_h (\mathbf{u}_h, t), \]

\[ p^{(0)} = u^n, \]

\[ i \in [1, \ldots, 5] : \begin{cases} k^{(i)} = a_i k^{(i-1)} + \Delta t \mathcal{L}_h (p^{(i-1)}, t^n + c_i \Delta t), \\ p^{(i)} = p^{(i-1)} + b_i k^{(i)}, \end{cases} \]

\[ u^{n+1}_h = p^{(5)}. \]

<table>
<thead>
<tr>
<th>i</th>
<th>(a_i)</th>
<th>(b_i)</th>
<th>(c_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1432997174477</td>
<td>9575080441755</td>
</tr>
<tr>
<td>2</td>
<td>-567301805773/1357537059087</td>
<td>5161836677717/13612068292357</td>
<td>1432997174477/9575080441755</td>
</tr>
<tr>
<td>3</td>
<td>-2404267990393/2016746695238</td>
<td>1720146321549/2090206949498</td>
<td>2526269341429/6820363962896</td>
</tr>
<tr>
<td>4</td>
<td>-3550918686646/2091501179385</td>
<td>3134564353537/4481467310338</td>
<td>3224310063776/2802321613138</td>
</tr>
<tr>
<td>5</td>
<td>-1275806237668/842570457699</td>
<td>2277821191437/14882151754819</td>
<td>2924317926251/2924317926251</td>
</tr>
</tbody>
</table>

There are many suitable alternatives
To complete the scheme for solving the advection equation, we need only integrate the system in time. As discussed in Section 3.4, we do this using a low-storage RK method. This is implemented in Advec1D.m.

```matlab
function [u] = Advec1D(u, FinalTime)
% function [u] = Advec1D(u, FinalTime)
% Purpose : Integrate 1D advection until FinalTime starting with
% initial the condition, u

Globals1D;
time = 0;

% Runge-Kutta residual storage
resu = zeros(Np,K);

% compute time step size
xmin = min(abs(x(1,:)-x(2,:)));
CFL=0.75; dt = CFL/(2*pi)*xmin; dt = .5*dt;
Nsteps = ceil(FinalTime/dt); dt = FinalTime/Nsteps;

% advection speed
a = 2*pi;

% outer time step loop
for tstep=1:Nsteps
    for INTRK = 1:5
        timelocal = time + rk4c(INTRK)*dt;
        [rhsu] = AdvecRHS1D(u, timelocal, a);
        resu = rk4a(INTRK)*resu + dt*rhsu;
        u = u+rk4b(INTRK)*resu;
    end;
    % Increment time
    time = time+dt;
end;
return
```

One issue we have not discussed yet is how to choose the timestep to ensure a discretely stable scheme. In Advec1D.m we use the guideline 

\[ \Delta t \leq C^2 \pi \min_{k,i} \Delta x_{k,i}, \]

where \( \Delta x_{k,i} = x_{k,i+1} - x_{k,i} \) and \( 2\pi \) is the maximum wave speed. The constant, \( C \), is a Courant-Friedrichs-Levy (CFL)-like constant that can be expected to be \( O(1) \). We will discuss this in more detail in Chapter 4 but the above expression is an excellent guideline.
Putting it together - an example

```matlab
% Driver script for solving the 1D advection equations
Globals1D;

% Order of polynomials used for approximation
N = 8;

% Generate simple mesh
[Nv, VX, K, EToV] = MeshGen1D(0.0,2.0,10);

% Initialize solver and construct grid and metric
StartUp1D;

% Set initial conditions
u = sin(x);

% Solve Problem
FinalTime = 10;
[u] = Advec1D(u,FinalTime);
```

Structure is the same for any time-dependent problem.

Often only the first routine needs any work.
Putting it together - an example

<table>
<thead>
<tr>
<th>N</th>
<th>K</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>Convergence rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>4.0E-01</td>
<td>9.1E-02</td>
<td>2.3E-02</td>
<td><strong>5.7E-03</strong></td>
<td>1.4E-03</td>
<td>2.0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2.0E-01</td>
<td>4.3E-02</td>
<td><strong>6.3E-03</strong></td>
<td>8.0E-04</td>
<td>1.0E-04</td>
<td>1.3E-05</td>
<td>3.0</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td><strong>3.3E-03</strong></td>
<td>3.1E-04</td>
<td>9.9E-06</td>
<td>3.2E-07</td>
<td>1.0E-08</td>
<td>3.3E-10</td>
<td>5.0</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>2.1E-07</td>
<td>2.5E-09</td>
<td>4.8E-12</td>
<td>2.2E-13</td>
<td>5.0E-13</td>
<td>6.6E-13</td>
<td>9.0</td>
</tr>
</tbody>
</table>

An important lesson to learn from this is that a visually smoother solution is not necessarily a more accurate solution, although in the case considered here, the global errors are comparable.

Many of the observations made in the above example regarding high-order methods discussed in this text.

Before we continue, it is instructive to consider an alternative derivation. The key ideas are problemdependent constants.

Fig. 2.2.

The rate by which the results converge are not, however, the same when changing the number of elements. However, the dissipative nature of the upwind flux is also apparent. One can increase the local order of approximation, which is a characteristic of the family of methods discussed here. To contrast, we show on the right of Fig. 2.2 the same solution computed using a pure upwind flux. This leads to a solution with smaller jumps between the element. In the one-dimensional case, this is achieved by adding two scaled Heaviside functions, defined as

$$f(x) = \frac{1}{2} \left( 1 + \frac{x}{h} \right) \quad \text{for} \quad -h < x < h,$$

where $h$ represents the exact solution.
A slightly more complicated example

Consider linear system

\[
\begin{bmatrix}
u \\
v
\end{bmatrix}_t + \begin{bmatrix}
0 & a \\
0 & b
\end{bmatrix}
\begin{bmatrix}
u \\
v
\end{bmatrix}_x = 0
\]

Clearly \(ab>0\) for hyperbolicity

How do we choose the flux in this case?

Recall that

\[
A = \begin{bmatrix}
0 & a \\
b & 0
\end{bmatrix}
\quad \quad S^{-1}AS = \Lambda
\]

With

\[
S = \begin{bmatrix}
\frac{1}{\sqrt{b}} & -\frac{1}{\sqrt{b}} \\
\frac{1}{\sqrt{a}} & \frac{1}{\sqrt{a}}
\end{bmatrix}
\quad S^{-1} = \frac{1}{2}
\begin{bmatrix}
\sqrt{b} & \sqrt{a} \\
-\sqrt{b} & \sqrt{a}
\end{bmatrix}
\quad \Lambda = \begin{bmatrix}
\sqrt{ab} & 0 \\
0 & -\sqrt{ab}
\end{bmatrix}
\]
A slightly more complicated example

This allows us to transform the equation as

\[ S^{-1} \begin{bmatrix} u \\ v \end{bmatrix}_t + \Lambda S^{-1} \begin{bmatrix} u \\ v \end{bmatrix}_x = 0 \]

So now we know that

\[ \frac{1}{2} (\sqrt{bu} + \sqrt{av}) \] propagates right

\[ \frac{1}{2} (-\sqrt{bu} + \sqrt{av}) \] propagates left

This suggests a flux like

\[ S \left( \Lambda^+ \frac{\sqrt{bu}^- + \sqrt{av}^-}{2} + \Lambda^- \frac{-\sqrt{bu}^+ + \sqrt{av}^+}{2} \right) \quad \Lambda = \Lambda^+ + \Lambda^- \]
A slightly more complicated example

Straightforward manipulations yield

\[
\begin{cases}
\frac{\sqrt{a}}{2} \left( \sqrt{a}(v^+ + v^-) + \sqrt{b}(u^- - u^+) \right) \\
\frac{\sqrt{b}}{2} \left( \sqrt{b}(u^+ + u^-) + \sqrt{a}(v^- - v^+) \right)
\end{cases}
\]

or the equivalent

\[
\begin{cases}
\{av\} + \frac{\sqrt{ab}}{2} \mathbf{n} \cdot \left[ [u] \right] \\
\{bu\} + \frac{\sqrt{ab}}{2} \mathbf{n} \cdot \left[ [v] \right]
\end{cases}
\]

Recall

\[
\begin{bmatrix}
u \\
v
\end{bmatrix}_t + \begin{bmatrix}
0 & a \\
b & 0
\end{bmatrix}
\begin{bmatrix}
u \\
v
\end{bmatrix}_x = 0
\]

we recognize this as the LF flux
Shallow water equations

Consider 1D shallow water equations in periodic domain

\[
\frac{\partial}{\partial t} \begin{bmatrix} \eta \\ u \end{bmatrix} = \begin{bmatrix} 0 & -h \\ -g & 0 \end{bmatrix} \frac{\partial}{\partial x} \begin{bmatrix} \eta \\ u \end{bmatrix}
\]

Tests of $h$- and $p$-refinement

\[
\|u - u_h\|_{\Omega,h} \leq Ch^{N+1}
\]
Putting it to use - an example

Consider Maxwell’s equations

\[
\varepsilon(x) \frac{\partial E}{\partial t} = -\frac{\partial H}{\partial x}, \quad \mu(x) \frac{\partial H}{\partial t} = -\frac{\partial E}{\partial x},
\]

BC: \(E(-1,t)=E(1,t)=0\) -- a PEC cavity

Interface conditions: Continuity of fields

The problem has a nice exact solution
Putting it to use - an example

Scheme becomes

\[
\frac{dE_h^k}{dt} + \frac{1}{J^k} \mathbf{D}_r \mathbf{H}_h^k = \frac{1}{J^k \varepsilon^k} \mathcal{M}^{-1} \left[ \ell^k(x)(H_h^k - H^*) \right]_{x_r^k}^{x_l^k} \\
= \frac{1}{J^k \varepsilon^k} \mathcal{M}^{-1} \int_{x_l^k}^{x_r^k} \mathbf{n} \cdot (H_h^k - H^*) \ell^k(x) \, dx,
\]

\[
\frac{dH_h^k}{dt} + \frac{1}{J^k \mu^k} \mathbf{D}_r \mathbf{E}_h^k = \frac{1}{J^k \mu^k} \mathcal{M}^{-1} \left[ \ell^k(x)(E_h^k - E^*) \right]_{x_r^k}^{x_l^k} \\
= \frac{1}{J^k \mu^k} \mathcal{M}^{-1} \int_{x_l^k}^{x_r^k} \mathbf{n} \cdot (E_h^k - E^*) \ell^k(x) \, dx.
\]

\[
H^- - H^* = \frac{1}{2 \{\{Z\}\}} \left( Z^+ [H] - [E] \right),
\]

\[
E^- - E^* = \frac{1}{2 \{\{Y\}\}} \left( Y^+ [E] - [H] \right),
\]

\[
Z^\pm = \sqrt{\frac{\mu^\pm}{\varepsilon^\pm}} = (Y^\pm)^{-1}.
\]
Putting it to use - an example

MaxwellRHS1D.m

function [rhsE, rhsH] = MaxwellRHS1D(E,H,eps,mu)

% function [rhsE, rhsH] = MaxwellRHS1D(E,H,eps,mu)
% Purpose : Evaluate RHS flux in 1D Maxwell

Globals1D;

% Compute impedance
Zimp = sqrt(mu./eps);

% Define field differences at faces
dE = zeros(Nfp*Nfaces,K); dE(:) = E(vmapM)-E(vmapP);
dH = zeros(Nfp*Nfaces,K); dH(:) = H(vmapM)-H(vmapP);
Zimpm = zeros(Nfp*Nfaces,K); Zimpm(:) = Zimp(vmapM);
Zimpp = zeros(Nfp*Nfaces,K); Zimpp(:) = Zimp(vmapP);
Yimpm = zeros(Nfp*Nfaces,K); Yimpm(:) = 1./Zimpm(:);
Yimpp = zeros(Nfp*Nfaces,K); Yimpp(:) = 1./Zimpp(:);

% Homogeneous boundary conditions, Ez=0
Ebc = -E(vmapB); dE (mapB) = E(vmapB) - Ebc;
Hbc = H(vmapB); dH (mapB) = H(vmapB) - Hbc;

% evaluate upwind fluxes
fluxE = 1./(Zimpm + Zimpp).*(nx.*Zimpp.*dH - dE);
fluxH = 1./(Yimpm + Yimpp).*(nx.*Yimpp.*dE - dH);

% compute right hand sides of the PDE's
rhsE = (-rx.*(Dr*H) + LIFT*(Fscale.*fluxE))./eps;
rhsH = (-rx.*(Dr*E) + LIFT*(Fscale.*fluxH))./mu;

return
Putting it to use - an example

---

**Maxwell1D.m**

```matlab
function [E,H] = Maxwell1D(E,H,eps,mu,FinalTime);

% function [E,H] = Maxwell1D(E,H,eps,mu,FinalTime)
% Purpose : Integrate 1D Maxwell's until FinalTime starting with
% conditions (E(t=0),H(t=0)) and materials (eps,mu).

Globals1D;

% Runge-Kutta residual storage
resE = zeros(Np,K); resH = zeros(Np,K);

% compute time step size
xmin = min(abs(x(1,:)-x(2,:)));
CFL=1.0; dt = CFL*xmin;
Nsteps = ceil(FinalTime/dt); dt = FinalTime/Nsteps;

% outer time step loop
for tstep=1:Nsteps
    for INTRK = 1:5
        [rhsE, rhsH] = MaxwellRHS1D(E,H,eps,mu);

        resE = rk4a(INTRK)*resE + dt*rhsE;
        resH = rk4b(INTRK)*resH + dt*rhsH;

        E = E+rk4b(INTRK)*resE;
        H = H+rk4b(INTRK)*resH;
    end

    % Increment time
    time = time+dt;
end
return
```

In the final routine, MaxwellDriver1D.m, the only significant difference from AdvecDriver1D.m is the need to also specify the spatial distribution of ε and µ. In this particular case, we have assumed that these are constant in each element but can jump between elements. Furthermore, we assume here that the jump is between the two middle elements. Clearly, this specification is problem dependent and can be changed as needed. Also, if ε and/or µ vary smoothly within the elements, this can be specified here with no further changes elsewhere.
Putting it to use - an example

```matlab
% Driver script for solving the 1D Maxwell's equations
Globals1D;

% Polynomial order used for approximation
N = 6;

% Generate simple mesh
[Nv, VX, K, EToV] = MeshGen1D(-2.0, 2.0, 80);

% Initialize solver and construct grid and metric
StartUp1D;

% Set up material parameters
eps1 = [ones(1,K/2), 2*ones(1,K/2)];
mu1 = ones(1,K);
epsilon = ones(Np,1)*eps1;
mu = ones(Np,1)*mu1;

% Set initial conditions
E = sin(pi*x).*(x<0); H = zeros(Np,K);

% Solve Problem
FinalTime = 10;
[E,H] = Maxwell1D(E,H,epsilon,mu,FinalTime);
```
Putting it to use - an example

Both time-stepping and driver are essentially unchanged from the first case.
Let’s summarize

We have some of the major pieces

- Some understanding of the key elements
- Evidence of good behavior
- Computationally well-conditioned operators
- Flexible and general implementations

However, many problems remain open

- What can we prove in terms of accuracy?
- How do we compute fluxes for systems?
- What about time-steps?