Solution set 11:

Discontinuous Galerkin methods

Exercise 11.1 **(a)** The system

$$u_t + v_x = 0 \tag{1}$$
$$v_t + u_x = 0 \ , \tag{2}$$

has the form

$$w_t + A w_x = 0 \tag{3}$$

where A is a symmetric matrix. In the kth element the approximation w_h is given by

$$w_h^k(x,t) = \sum_{n=1}^{N_p} \psi_n^k(x) \hat{w}_n^k(t) \ . \tag{4}$$

We require that

$$\int_{D^k} \psi_j^k \left(\frac{\mathrm{d}w_h^k}{\mathrm{d}t} + A \frac{\mathrm{d}w_h^k}{\mathrm{d}x} \right) \mathrm{d}x = 0 \qquad k = 1, \dots, K \ . \tag{5}$$

We integrate by parts, and replace Aw_h^k in the boundary terms by a numerical flux f^* to get

$$\sum_{n=1}^{N_p} \left(\psi_j^k, \psi_n^k\right)_{D^k} \frac{\mathrm{d}\hat{w}_n^k}{\mathrm{d}t} - \sum_{n=1}^{N_p} \left(\frac{\mathrm{d}\psi_j^k}{\mathrm{d}x}, \ \psi_n^k \right)_{D^k} A\hat{w}_n^k = -\left[\psi_j^k f^*\right]_{x_l^k}^{x_r^k} \ . \tag{6}$$

That is

$$\hat{\mathcal{M}}^k \frac{\mathrm{d}}{\mathrm{d}t} \hat{\boldsymbol{u}}_n^k - \left(\hat{\mathcal{S}}^k\right)^T \hat{\boldsymbol{v}}_h^k = \left[\boldsymbol{\psi} g^*\right]_{x_l^k}^{x_r^k} \tag{7}$$
$$\hat{\mathcal{M}}^k \frac{\mathrm{d}}{\mathrm{d}t} \hat{\boldsymbol{v}}_n^k - \left(\hat{\mathcal{S}}^k\right)^T \hat{\boldsymbol{u}}_h^k = \left[\boldsymbol{\psi} h^*\right]_{x_l^k}^{x_r^k} \ , \tag{8}$$

where $f^* = (g^*, h^*)^T$.

(b) By taking the dot product of (6) and $\hat{w}_j^k$, and summing over $j = 1, \dots, N_p$, we get

$$\frac{1}{2} \frac{\mathrm{d}}{\mathrm{d}t} \left\|w_h^k\right\|_{D^k}^2 - \int_{D^k} \frac{\mathrm{d}w_h^k}{\mathrm{d}x} \cdot A w_h^k \ \mathrm{d}x = -\left[w_h^k \cdot f^*\right]_{x_l^k}^{x_r^k} \ , \tag{9}$$

which implies

$$\frac{\mathrm{d}}{\mathrm{d}t} \left\|w_h^k\right\|_{D^k}^2 = \left[w_h^k \cdot \left(Aw_h^k - 2f^*\right)\right]_{x_l^k}^{x_r^k} = w_r^k \cdot \left(Aw_r^k - 2f_{k,k+1}^*\right) - w_l^k \cdot \left(Aw_l^k - 2f_{k-1,k}^*\right) \ , \tag{10}$$

since A is symmetric. Here $w_r^k = w_h^k\left(x_r^k, \cdot\right)$, and $w_l^k = w_h^k\left(x_l^k, \cdot\right)$. Summing over $k = 1, \dots, K$ provides

$$\frac{\mathrm{d}}{\mathrm{d}t} \left\|w_h\right\|^2 = \sum_{k=1}^{K} w_r^k \cdot \left(Aw_r^k - 2f_{k,k+1}^*\right) - \sum_{k=0}^{K-1} w_l^{k+1} \cdot \left(Aw_l^{k+1} - 2f_{k,k+1}^*\right) \tag{11}$$
$$= -\sum_{k=1}^{K} \left[w_l^{k+1} \cdot Aw_l^{k+1} - w_r^k \cdot Aw_r^k - 2\left(w_l^{k+1} - w_r^k\right) \cdot f_{k,k+1}^* \right] \ , \tag{12}$$

where $w_l^{K+1} := w_l^1$ and $f_{K,K+1}^* := f_{K,1}^*$. Now we substitute the upwind flux

$$f_{k,k+1}^* = \frac{1}{2} \ A\left(w_l^{k+1} + w_r^k\right) - \frac{1}{2} \ |A| \left(w_l^{k+1} - w_r^k\right) \tag{13}$$

into (11) to get

$$\frac{\mathrm{d}}{\mathrm{d}t} \left\|w_h\right\|^2 = -\sum_{k=1}^{K} \left(w_l^{k+1} - w_r^k\right) \cdot |A| \left(w_l^{k+1} - w_r^k\right) \leq 0 \ . \tag{14}$$

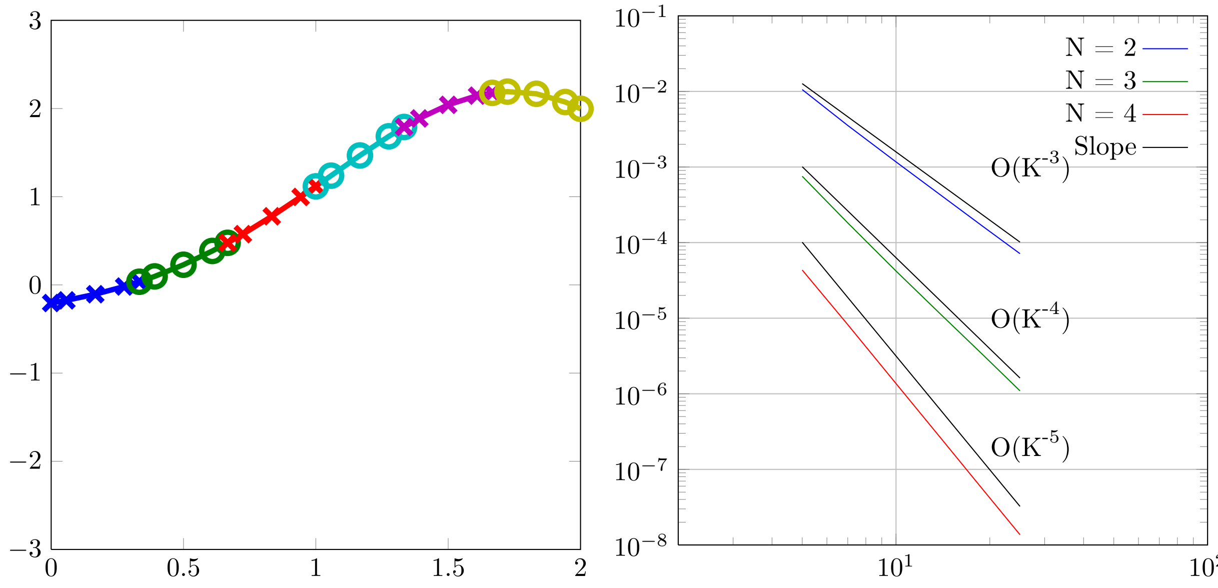
Exercise 11.2 We solve the equation

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = h\left(x,t\right),\ x \in [0,2\pi] \tag{15}$$

with $a \geq 0$ and

$$u\left(0,t\right) = g\left(t\right),\ u\left(x,0\right) = f\left(x\right) \tag{16}$$

(a) Find Matlab/Octave code attached, see figures created when running the code. In figurer 1 the numerical approximation to eq. 15 at $t = \frac{1}{2}\pi$ is presented with $a = 2\pi$, $h\left(x,t\right) = 10\cos\left(t\right)\mathrm{e}^{\sin(x)\cos(t)}$, $g\left(t\right) = \mathrm{e}^{\sin(2\pi t)^2}$ and $f\left(x\right) = \sin\left(2\pi x\right)$. The solution is generated using the Nodal Discontinous Garlerkin Matlab code provided.
(b) See code attached and figurer 1. Notice how a very small time-step is used to ensure that the error measured is dominated by the errror made in the aproximation of the spacial domain. In figurer 2, the accuracy is measured. As expected, the order of accuracy measured is $\mathcal{O}\left(\Delta x^{N+1}\right)$.



(a) (b)

Figure 1: (a) Solution structure. (b) Accuracy test.

```
clear all,clc
% In this DEMO_SCRIPT advection equation is solved
% g(t), f(x) and h(x,t) in advection equation.
% More explanation
%

%% Exercise 11.2a

Globals1D;

% Order N of polymomials used for approximation
N           = 4;
Elements    = 6;

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

% Initialize solver and construct grid and metric
StartUp1D;

% advection speed
a = 2*pi;

% Set initial condition and final time
u =  f(x);
time = 0;
FinalTime = pi/2;

% 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;

% Integrate and plot
figure(1);
shapestr = {'-o','-x'};
for tstep=1:Nsteps
    for INTRK = 1:5
        timelocal = time + rk4c(INTRK)*dt;
        [rhsu] = AdvecRHS1DMOD(u, timelocal, a);
        resu = rk4a(INTRK)*resu + dt*rhsu;
        u = u+rk4b(INTRK)*resu;
    end;
    % Increment time
    time = time+dt;
    if(rem(tstep,10)==0)
        for i=1:Elements
            plot(x(:,i),u(:,i),shapestr{1+rem(i,2)},'Markersize',8,'LineWidth',2);
            hold all
        end
        axis([0 2 -3 3]);
        drawnow;
        hold off
    end
end

matlab2tikz('Solution.tikz', 'height','\figureheight', 'width', '\figurewidth');

%% Exercise 11.2b

NPOLY = [2,3,4];
ELEME = ceil(5.^(1:0.2:2.0));
ERROR = zeros(numel(ELEME),numel(NPOLY));

tic
for i = 1:numel(NPOLY)
    for j = 1:numel(ELEME)
        i
        j
        % Order N of polymomials used for approximation
        N            = NPOLY(i);
        Elements     = ELEME(j);

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

        % Initialize solver and construct grid and metric
        StartUp1D;

        % advection speed
        a = 2*pi;

        % Set initial condition and final time
        u = sin(pi*x);
        time = 0;
        FinalTime = 0.25*pi;

        % 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 = .05*dt;
        Nsteps = ceil(FinalTime/dt); dt = FinalTime/Nsteps;

        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;
            time = time+dt;
        end

        ERROR(j,i) = norm(u(:) - sin(pi*(x(:)-a*time)),1)/numel(u(:));

    end
end
t = toc
%% Plot that shizzle
figure(2);
for i = 1:numel(NPOLY)
    loglog(ELEME,ERROR(:,i))
    hold all
end
grid on
hold off
loglog(ELEME,10^(+0.2)*ELEME.^(-3),'k')
loglog(ELEME,10^(-0.2)*ELEME.^(-4),'k')
loglog(ELEME,10^(-0.5)*ELEME.^(-5),'k')
legend('N = 2','N = 3','N = 4',' Slope');
axis([ 2*10^(0) 10^(2) 10^(-8) 10^(-1) ]);
text(2*10,10^(-3),'O(K^{-3})')
text(2*10,10^(-5),'O(K^{-4})')
text(2*10,2*10^(-7),'O(K^{-5})')

matlab2tikz('Slope.tikz', 'height','\figureheight', 'width', '\figurewidth');
```

```
function [ out ] = f( x )
    out = sin(2*pi.*x);
end
```

```
function [ out ] = g( t )
    out = exp(sin(2*pi.*t)).*sin(2*pi.*t);
end
```

```
function [ out ] = h( x , t )
    out = 10*cos(t)*exp(sin(x)*cos(t));
```

```
end
```

```
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;

    if (mod(tstep,10)==0)
       plot(x,u);
       axis([0 2 -4 4]);
       drawnow;
    end
end;
return
```

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

% Order of polymomials used for approximation
N = 4;

% Generate simple mesh
[Nv, VX, K, EToV] = MeshGen1D(0.0,2.0,16);

% Initialize solver and construct grid and metric
StartUp1D;

% Set initial conditions
u = zeros(size(x));%sin(pi*x);

% Solve Problem
FinalTime = 5*pi;
[u] = Advec1D(u,FinalTime);
```

```
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 = 0 - upwind flux
% alpha = 1 — central flux

alpha = 0;
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 = u(vmapO); uout = u(vmapI);
du (mapI) = (u(vmapI)— uin ).*(a*nx(mapI)—(1—alpha)*abs(a*nx(mapI)))/2;
du (mapO) = (u(vmapO)— uout).*(a*nx(mapO)—(1—alpha)*abs(a*nx(mapO)))/2;

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

return
```

```
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 = 0 — upwind flux
% alpha = 1 — central flux

alpha=0;
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 = u(vmapO); uout = u(vmapI);
% du (mapI) = (u(vmapI)— uin ).*(a*nx(mapI)—(1—alpha)*abs(a*nx(mapI)))/2;
% du (mapO) = (u(vmapO)— uout).*(a*nx(mapO)—(1—alpha)*abs(a*nx(mapO)))/2;

uin = g(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)) — h(x,time);

return
```

```
function [vmapM, vmapP, vmapB, mapB] = BuildMaps1D

% function [vmapM, vmapP, vmapB, mapB] = BuildMaps1D
% Purpose: Connectivity and boundary tables for nodes given in the K # of elements,
%          each with N+1 degrees of freedom.

Globals1D;

% number volume nodes consecutively
nodeids = reshape(1:K*Np, Np, K);
vmapM   = zeros(Nfp, Nfaces, K);
vmapP   = zeros(Nfp, Nfaces, K);

for k1=1:K
  for f1=1:Nfaces
    % find index of face nodes with respect to volume node ordering
    vmapM(:,f1,k1) = nodeids(Fmask(:,f1), k1);
  end
end

for k1=1:K
  for f1=1:Nfaces
    % find neighbor
    k2 = EToE(k1,f1); f2 = EToF(k1,f1);

    % find volume node numbers of left and right nodes
    vidM = vmapM(:,f1,k1); vidP = vmapM(:,f2,k2);

    x1  = x(vidM); x2  = x(vidP);

    % Compute distance matrix
    D = (x1 -x2 ).^2;
    if (D<NODETOL) vmapP(:,f1,k1) = vidP; end;
  end
end

vmapP = vmapP(:); vmapM = vmapM(:);

% Create list of boundary nodes
mapB = find(vmapP==vmapM); vmapB = vmapM(mapB);

% Create specific left (inflow) and right (outflow) maps
mapI = 1; mapO = K*Nfaces; vmapI = 1; vmapO = K*Np;
return
```

```
function [EToE, EToF] = Connect1D(EToV)

% function [EToE, EToF] = Connect1D(EToV)
% Purpose  : Build global connectivity arrays for 1D grid based on standard
%            EToV input array from grid generator

Nfaces = 2;
% Find number of elements and vertices
K = size(EToV,1); TotalFaces = Nfaces*K; Nv = K+1;

% List of local face to local vertex connections
vn = [1,2];

% Build global face to node sparse array
SpFToV = spalloc(TotalFaces, Nv, 2*TotalFaces);
sk = 1;
for k=1:K
  for face=1:Nfaces
     SpFToV( sk, EToV(k, vn(face))) = 1;
     sk = sk+1;
  end
end

% Build global face to global face sparse array
SpFToF = SpFToV*SpFToV' - speye(TotalFaces);

% Find complete face to face connections
[faces1, faces2] = find(SpFToF==1);

% Convert face global number to element and face numbers
element1 = floor( (faces1-1)/Nfaces )  + 1;
face1    =   mod( (faces1-1), Nfaces ) + 1;
element2 = floor( (faces2-1)/Nfaces )  + 1;
face2    =   mod( (faces2-1), Nfaces ) + 1;

% Rearrange into Nelements x Nfaces sized arrays
ind = sub2ind([K, Nfaces], element1, face1);
EToE      = (1:K)'*ones(1,Nfaces);
EToF      = ones(K,1)*(1:Nfaces);
EToE(ind) = element2; EToF(ind) = face2;
return
```

```
function [Dr] = Dmatrix1D(N,r,V)

% function [Dr] = Dmatrix1D(N,r,V)
% Purpose : Initialize the (r) differentiation matrices on the interval,
%           evaluated at (r) at order N

Vr = GradVandermonde1D(N, r);
Dr = Vr/V;
return
```

```
function [rx,J] = GeometricFactors1D(x,Dr)

% function [rx,J] = GeometricFactors1D(x,Dr)
% Purpose  : Compute the metric elements for the local mappings of the 1D elements

xr  = Dr*x; J = xr; rx = 1./J;
return
```

```
% Purpose: declare global variables

global N Nfp Np K
global r x  VX
global Dr LIFT
global nx Fx Fscale
global vmapM vmapP vmapB mapB Fmask
global vmapI vmapO mapI mapO
global rx J
global rk4a rk4b rk4c
global Nfaces EToE EToF
global V invV
global NODETOL

% Low storage Runge—Kutta coefficients
rk4a = [            0.0 ...
        -567301805773.0/1357537059087.0 ...
        -2404267990393.0/2016746695238.0 ...
        -3550918686646.0/2091501179385.0  ...
        -1275806237668.0/842570457699.0];
rk4b = [ 1432997174477.0/9575080441755.0 ...
         5161836677717.0/13612068292357.0 ...
         1720146321549.0/2090206949498.0  ...
         3134564353537.0/4481467310338.0  ...
         2277821191437.0/14882151754819.0];
rk4c = [             0.0  ...
         1432997174477.0/9575080441755.0 ...
         2526269341429.0/6820363962896.0 ...
         2006345519317.0/3224310063776.0 ...
         2802321613138.0/2924317926251.0];
```

```
function [dP] = GradJacobiP(r, alpha, beta, N);

% function [dP] = GradJacobiP(r, alpha, beta, N);
% Purpose: Evaluate the derivative of the Jacobi polynomial of type (alpha,beta)>—1,
%          at points r for order N and returns dP[1:length(r))]

dP = zeros(length(r), 1);
if(N == 0)
  dP(:,:) = 0.0;
else
  dP = sqrt(N*(N+alpha+beta+1))*JacobiP(r(:),alpha+1,beta+1, N—1);
end;
return
```

```
function [DVr] = GradVandermonde1D(N,r)

% function [DVr] = GradVandermonde1D(N,r)
% Purpose : Initialize the gradient of the modal basis (i) at (r) at order N

DVr = zeros(length(r),(N+1));

% Initialize matrix
for i=0:N
   [DVr(:,i+1)] = GradJacobiP(r(:),0,0,i);
end
return
```

```
function [x] = JacobiGL(alpha,beta,N);

% function [x] = JacobiGL(alpha,beta,N)
% Purpose: Compute the N'th order Gauss Lobatto quadrature
%          points, x, associated with the Jacobi polynomial,
%          of type (alpha,beta) > -1 ( <> -0.5).

x = zeros(N+1,1);
if (N==1) x(1)=-1.0; x(2)=1.0; return; end;

[xint,w] = JacobiGQ(alpha+1,beta+1,N-2);
x = [-1, xint', 1]';
return;
```

```
function [x,w] = JacobiGQ(alpha,beta,N);

% function [x,w] = JacobiGQ(alpha,beta,N)
% Purpose: Compute the N'th order Gauss quadrature points, x,
%          and weights, w, associated with the Jacobi
%          polynomial, of type (alpha,beta) > -1 ( <> -0.5).

if (N==0) x(1)=(alpha-beta)/(alpha+beta+2); w(1) = 2; return; end

% Form symmetric matrix from recurrence.
J = zeros(N+1);
h1 = 2*(0:N)+alpha+beta;
J = diag(-1/2*(alpha^2-beta^2)./(h1+2)./h1) + ...
    diag(2./(h1(1:N)+2).*sqrt((1:N).*((1:N)+alpha+beta).*...
    ((1:N)+alpha).*((1:N)+beta)./(h1(1:N)+1)./(h1(1:N)+3)),1);
if (alpha+beta<10*eps) J(1,1)=0.0;end;
J = J + J';

%Compute quadrature by eigenvalue solve
[V,D] = eig(J); x = diag(D);
w = (V(1,:)').^2*2^(alpha+beta+1)/(alpha+beta+1)*gamma(alpha+1)*...
    gamma(beta+1)/gamma(alpha+beta+1);
return;
```

```
function [P] = JacobiP(x,alpha,beta,N);

% function [P] = JacobiP(x,alpha,beta,N)
% Purpose: Evaluate Jacobi Polynomial of type (alpha,beta) > -1
%          (alpha+beta <> -1) at points x for order N and returns P[1:length(xp))]
% Note   : They are normalized to be orthonormal.

% Turn points into row if needed.
xp = x; dims = size(xp);
if (dims(2)==1) xp = xp'; end;

PL = zeros(N+1,length(xp));

% Initial values P_0(x) and P_1(x)
gamma0 = 2^(alpha+beta+1)/(alpha+beta+1)*gamma(alpha+1)*...
    gamma(beta+1)/gamma(alpha+beta+1);
PL(1,:) = 1.0/sqrt(gamma0);
if (N==0) P=PL'; return; end;
gamma1 = (alpha+1)*(beta+1)/(alpha+beta+3)*gamma0;
PL(2,:) = ((alpha+beta+2)*xp/2 + (alpha-beta)/2)/sqrt(gamma1);
if (N==1) P=PL(N+1,:)'; return; end;

% Repeat value in recurrence.
aold = 2/(2+alpha+beta)*sqrt((alpha+1)*(beta+1)/(alpha+beta+3));

% Forward recurrence using the symmetry of the recurrence.
for i=1:N-1
  h1 = 2*i+alpha+beta;
  anew = 2/(h1+2)*sqrt( (i+1)*(i+1+alpha+beta)*(i+1+alpha)*...
      (i+1+beta)/(h1+1)/(h1+3));
  bnew = - (alpha^2-beta^2)/h1/(h1+2);
  PL(i+2,:) = 1/anew*( -aold*PL(i,:) + (xp-bnew).*PL(i+1,:));
  aold =anew;
end;
P = PL(N+1,:)';
return
```

```
function [LIFT] = Lift1D

% function [LIFT] = Lift1D
% Purpose  : Compute surface integral term in DG formulation

Globals1D;
Emat = zeros(Np,Nfaces*Nfp);

% Define Emat
Emat(1,1) = 1.0; Emat(Np,2) = 1.0;

% inv(mass matrix)*\s_n (L_i,L_j)_{edge_n}
LIFT = V*(V'*Emat);
return
```

```
function [Nv, VX, K, EToV] = MeshGen1D(xmin,xmax,K)

% function [Nv, VX, K, EToV] = MeshGen1D(xmin,xmax,K)
% Purpose  : Generate simple equidistant grid with K elements

Nv = K+1;

% Generate node coordinates
VX = (1:Nv);
for i = 1:Nv
  VX(i) = (xmax-xmin)*(i-1)/(Nv-1) + xmin;
end

% read element to node connectivity
EToV = zeros(K, 2);
for k = 1:K
  EToV(k,1) = k; EToV(k,2) = k+1;
end
return
```

```
function [nx] = Normals1D

% function [nx] = Normals1D
% Purpose : Compute outward pointing normals at elements faces

Globals1D;
nx = zeros(Nfp*Nfaces, K);

% Define outward normals
nx(1, :) = -1.0; nx(2, :) = 1.0;
return
```

```
% Purpose : Setup script, building operators, grid, metric and connectivity for 1D solver.

% Definition of constants

Globals1D; NODETOL = 1e-10;
Np = N+1; Nfp = 1; Nfaces=2;

% Compute basic Legendre Gauss Lobatto grid
r = JacobiGL(0,0,N);

% Build reference element matrices
V  = Vandermonde1D(N, r); invV = inv(V);
Dr = Dmatrix1D(N, r, V);

% Create surface integral terms
```

```
LIFT = Lift1D();

% build coordinates of all the nodes
va = EToV(:,1)'; vb = EToV(:,2)';
x = ones(N+1,1)*VX(va) + 0.5*(r+1)*(VX(vb)-VX(va));

% calculate geometric factors
[rx,J] = GeometricFactors1D(x,Dr);

% Compute masks for edge nodes
fmask1 = find( abs(r+1) < NODETOL)';
fmask2 = find( abs(r-1) < NODETOL)';
Fmask  = [fmask1;fmask2]';
Fx = x(Fmask(:), :);

% Build surface normals and inverse metric at surface
[nx] = Normals1D();
Fscale = 1./(J(Fmask,:));

% Build connectivity matrix
[EToE, EToF] = Connect1D(EToV);

% Build connectivity maps
[vmapM, vmapP, vmapB, mapB] = BuildMaps1D;
```

```
function [V1D] = Vandermonde1D(N,r)

% function [V1D] = Vandermonde1D(N,r)
% Purpose : Initialize the 1D Vandermonde Matrix, V_{ij} = phi_j(r_i);

V1D = zeros(length(r),N+1);
for j=1:N+1
    V1D(:,j) = JacobiP(r(:), 0, 0, j-1);
end;

return
```