Numerical Methods for 1D Computational Fluid Dynamics

Developer: Hasan Eruslu (University of Delaware)

Last update: December 5, 2018

Acknowledgement. This library is built during the course *Numerical Methods in CFD* given by Dr. Jingmei Qiu in the spring of 2018 at the Department of Mathematical Sciences at University of Delaware. Most of the schemes presented here are based on the notes and assignments from that class.

Contents

1 Finite difference for waves

1.1 Example on 1D linear advection equation

The linear advection equation in 1D is

$$\frac{d}{dt}u(x,t) + c\frac{d}{dx}(u(x,t)) = 0 \qquad \text{on } [a,b]\cup[0,T]$$
$$u(x,0) = u_0(x) \qquad \text{on } [a,b]$$

Here we will be interested in constant c and periodic boundary conditions i.e. $u(a) = u(b)$. We know the exact solution is

$$u(x,t) = u_0(x-ct).$$

Therefore there is no need for a solver. Here we will implement two numerical scheme as a demonstration of basic finite difference methods

1. Downwind scheme. Stable when $c > 0$.

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} - c\frac{u_j^n - u_{j-1}^n}{\Delta x} = 0$$

2. Upwind scheme. Stable when $c < 0$.

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} - c\frac{u_{j+1}^n - u_j^n}{\Delta x} = 0$$

Note. These methods are $\mathcal{O}(\Delta t + \Delta x)$ when they are stable. However they introduce some numerical dissipation for large time T. For better accuracy refine the numerical space and choose the CFL constant $\frac{\Delta t}{\Delta x} = 1$.

```
function u = LinearAdvectionFDSolver1D(u0,a,dx,dt,intervalx,finalT,...
    scheme_option)
%
% u = LinearAdvectionFDSolver1D(u0,a,dx,dt,intervalx,finalT,scheme_option)
%
% Solves u_t + c u_x = 0 with periodic boundary conditions
%
% Input:
%    u0             :   Vectrorized functional handle for initial data
%                       u(x,0)
%    a              :   Scalar
%    dx             :   The distance between adjacent grid points in space
%    dt             :   The distance between adjacent grid points in time
%    intervalx      :   2 x 1 vector for space domain of u(x,t)
%    finalT         :   A positive number for the final time
%    scheme_option  :   (1) u^(n+1)_(j) = u^_j − c*lambda*(u^_j − u^_(j−1))
%                       (2) u^(n+1)_(j) = u^_j − c*lambda*(u^_(j+1) − u^_j)
% Output:
%    u              :   Nt x Nx matrix of numerical approximation of the
%                       solution of Burgers' Equation in 1D. Here Nt and Nx
%                       are the dimensions of discretization space of
%                       intervalx and [0, finalT]
%
% Last update: March 13, 2018

intervalt = [0 finalT];
% setting up the discrete problem
xx = intervalx(1) : dx : intervalx(2);
tt = intervalt(1)+dt : dt : intervalt(2)−dt;
% looping over x&t to find calculate solution u(x,t)
Nx = length(xx);
Nt = length(tt);
u = zeros(Nt,Nx);
un = u0(xx);
c = dt/dx;
for nt = 1 : Nt
    switch scheme_option
        case 1 % (1) u^(n+1)_(j) = u^n_j − c*lambda*(u^n_j − u^n_(j−1))
            un(2:end) = un(2:end) − a*c*(un(2:end) − un(1:end−1)); % 1 x Nx vector
            un(1) = un(end);
        case 2 % (2) u^(n+1)_(j) = u^n_j − c*lambda*sw(u^n_(j+1) − u^n_j)
            un(1:end−1) = un(1:end−1) − a*c*(un(2:end) − un(1:end−1));
            un(end) = un(1);
    end
    u(nt,:) = un;
end
```

2 Finite volume for waves

For the problem

$$u_t + (f(u))_x = 0$$

we implement the conservative finite volume scheme with upwind flux

$$\frac{d}{dt}\overline{u}_j(t) + \frac{1}{\Delta x}(\hat{f}_{j+1/2} - \hat{f}_{j-1/2}) = 0.$$

2.1 Example on 1D linear advection equation

In particular when $f(u) = u$, and upwind flux is set to $\hat{f}_{j+1/2} = u^-_{j+1/2}$. We implement a third order reconstruction for $u^-_{j+1/2}$ in the following way

$$u^-_{j+1/2} = -\frac{1}{6}\overline{u}_{j-1} + \frac{5}{6}\overline{u}_j + \frac{1}{3}\overline{u}_{j+1}.$$

The function **`LinearAdvectionFVSolver.m`** uses the finite volume scheme given above with a third order Runge Kutta temporal solver (see Section 4.2).

```
function u = LinearAdvectionFVSolver1D(U0,a,dx,dt,intervalx,...
    finalT,scheme_option)
%
% u = LinearAdvectionFVSolver1D(U0,a,dx,dt,intervalx,finalT,scheme_option)
%
% Solves u_t + a u_x = 0 with periodic boundary conditions
%
% Input:
%    U0              :   Vectrorized functional handle, antiderivative of
%                        initial data u(x,0)
%    a               :   Scalar
%    dx              :   The distance between adjacent grid points in space
%    dt              :   The distance between adjacent grid points in time
%    intervalx       :   2 x 1 vector for space domain of u(x,t)
%    finalT          :   A positive number for the final time
%    scheme_option   :   (3) Third order Finite Volume and Runge Kutta
%                            d/dt \bar{u}_j
%                             + 1/Deltax(^f_{j+1/2} - ^f_{j-1/2}) = 0
%                            ^f_{j+1/2} = u^-_{j+1/2} upwind flux
%
% Output:
%    u               :   Nt x Nx matrix of numerical approximation of the
%                        solution of Burgers' Equation in 1D. Here Nt and Nx
%                        are the dimensions of discretization space of
%                        intervalx and [0, finalT]
%
% Last update: April 24, 2018

% setting up the discrete problem
xx = intervalx(1) : dx : intervalx(2)-dx;
tt = dt : dt : finalT;
% finite volume scheme
flux1 = @(u) (-1/6)*[u(end) u(1:end-1)] + (5/6)*u + (1/3)*[u(2:end) u(1)];
flux2 = @(u) flux1([u(end) u(1:end-1)]);
f = @(u) (-a/dx)*(flux1(u) - flux2(u)); %RHS of Runge Kutta
Nx = length(xx);
Nt = length(tt);
u = zeros(Nt,Nx);
% initial condition
un = (U0([xx(2:end) xx(1)]) - U0(xx))/dx; % 1 x (Nx - 1)
for nt = 1 : Nt
    switch scheme_option
        case 3 % Runge Kutta 3rd order
            un = RungeKuttaSolver(un,f,nt,dt,3);
    end
    u(nt,:) = un;
end
```

2.2 General solver with Lax-Friedrich's flux

The function `LaxFriedrichsSolver1D.m` implements a third order space-time solver using the Lax-Friedrichs flux

$$\hat{f}_{j+1/2} = \hat{f}(u^-_{j+1/2}, u^+_{j+1/2}) = \frac{1}{2}\left(f(u^-_{j+1/2}) + f(u^+_{j+1/2})\right) + \frac{\alpha}{2}\left(u^-_{j+1/2} - u^+_{j+1/2}\right),$$

with $\alpha = \max_u |f'(u)|$ and the following third order reconstructions

$$u^-_{j+1/2} = -\frac{1}{6}\overline{u}_{j-1} + \frac{5}{6}\overline{u}_j + \frac{1}{3}\overline{u}_{j+1}, \qquad u^+_{j+1/2} = \frac{1}{3}\overline{u}_j + \frac{5}{6}\overline{u}_{j+1} - \frac{1}{6}\overline{u}_{j+2}.$$

Minmod correction. The function `LaxFriedrichsSolver1D.m` also has the option to use minmod correction, a tool helpful for wave simulations in case of shock formation. When minmod correction is preferred, we switch to the following total variation diminishing scheme with

$$u^-_{j+1/2} = \overline{u}_j + \text{minmod}(u^-_{j+1/2} - \overline{u}_j, \overline{u}_{j+1} - \overline{u}_j, \overline{u}_j - \overline{u}_{j-1}),$$

where

$$\text{minmod}(a,b,c) = \begin{cases} \text{sign}(a)\min\{|a|,|b|,|c|\}, & \text{if } \text{sign}(a) = \text{sign}(b) = \text{sign}(c), \\ 0, & \text{otherwise.} \end{cases}$$

Similarly

$$u^+_{j+1/2} = \overline{u}_{j+1} + \text{minmod}(\overline{u}_{j+1} - u^+_{j+1/2}, \overline{u}_{j+1} - \overline{u}_j, \overline{u}_{j+2} - \overline{u}_{j+1}).$$

```
function uh = LaxFriedrichsSolver1D(U0,f,fp,dx,dt,intervalx,T,...
    scheme_option)
%
% uh = LaxFriedrichsSolver1D(U0,f,fp,dx,dt,intervalx,T,scheme_op)
%
% Solves u_t + f(u)_x = 0    for x in intervalx, t in [0,T]
% with periodic boundary conditions
%
% Input:
%    U0            :   Function handle, antiderivative of initial data u0
%    f             :   Function handle
%    fp            :   Function handle, derivative of f
%    dx            :   The distance between adjacent grid points in space
%    dt            :   The distance between adjacent grid points in time
%    intervalx     :   2 x 1 vector for space domain of u(x,t)
%    T             :   Final time for time domain of u(x,t)
%    scheme_option :   A number from the set {1,3,4} where
%                      (1) 1st space and time
%                      (3) 3rd space and time scheme
%                      (4) 3rd space and time scheme with minmod correction
%
% Output:
%    uh            :   Nt x Nx matrix of numerical approximation of the
%                      solution of the PDE in 1D. Here Nt and Nx are the
%                      dimensions of discretization spaces for t and x
%
% Last update: December 4, 2018

% setting up the discrete problem
xx = intervalx(1) : dx : intervalx(2)-dx;
tt = dt : dt : T;
% initial condition
```

```
un = (U0([xx(2:end) xx(1)]) − U0(xx))/dx; % 1 x (Nx − 1)
% initializing the solution
Nx = length(xx); Nt = length(tt);
uh = zeros(Nt,Nx);
alpha = max(abs(fp(un)));
uminus = @(u) (−1/6)*[u(end) u(1:end−1)] + (5/6)*u + (1/3)*[u(2:end) u(1)];
uplus = @(u) (1/3)*u + (5/6)*[u(2:end) u(1)] − (1/6)*[u(3:end) u(1:2)];
switch scheme_option
    case 1 % 1st order space and time
        flux = @(u)0.5*(f(u)+f([u(2:end) u(1)])+alpha*(u−[u(2:end) u(1)]));
    case 3 % 3rd space and time
        flux = @(u) 0.5*(f(uminus(u))+f(uplus(u))+...
            alpha*(uminus(u)−uplus(u)));
    case 4 % 3rd order space and time with minmod correction
        signer = @(a,b,c) 1−0.5*(abs(sign(a)−sign(b)) + ...
            abs(sign(b)−sign(c)) + abs(sign(c)−sign(a)));
        minmod = @(a,b,c) sign(a).*signer(a,b,c)...
            .*min([abs(a);abs(b);abs(c)],[],1);
        uminusUp = @(u) u + minmod(uminus(u)−u,[u(2:end) u(1)]−u,...
            u−[u(end) u(1:end−1)]);
        uplusUp = @(u) [u(2:end) u(1)] − minmod([u(2:end) u(1)] − ...
            uplus(u),[u(2:end) u(1)]−u,[u(3:end) u(1:2)]−[u(2:end) u(1)]);
        flux = @(u) 0.5*(f(uminusUp(u))+f(uplusUp(u))+...
            alpha*(uminusUp(u)−uplusUp(u)));
end

spatial_disc = @(yn) (−1/dx)*(flux(yn) − flux([yn(end) yn(1:end−1)]));
linear_scheme=(scheme_option==1);
for nt = 1 : Nt
    if ¬linear_scheme
        un = RungeKuttaSolver(un,spatial_disc,dt,3);
    else
        un = un + dt*spatial_disc(un);
    end
    uh(nt,:) = un;
end
```

3 Exact solution for some nonlinear wave equations

3.1 Burgers' equation

Burgers' Equation in 1D is

$$\begin{aligned} \frac{d}{dt}u(x,t) - \frac{d}{dx}(u(x,t)^2/2) = 0 \qquad & \text{on } [a,b] \cup [0,T] \\ u(x,0) = u_0(x) \qquad & \text{on } [a,b] \end{aligned}$$

where $a < b$ are real numbers. Since solution of this equation is constant on characteristics, for given pair of (x,t) we know that

$$u(x,t) = u_0(x_\star),$$

where

$$\frac{x - x_\star}{t - 0} = u_0(x_\star).$$

We find this $x_\star$ numerically using a Newton's iteration. To do this `BurgersSolver1D` does the following

1. Discretize the spatial and temporal domain as $\{x_n\}_{n=1}^{N}$ and $\{t_m\}_{m=1}^{M}$
2. Loops over n, m to compute $x_\star^{m,n}$

Note. The computation at each step is independent from each other. Therefore this loop can be parallelized. This is done via the function `BurgersSolver1DPar.m`.

```
function u = BurgersSolver1D(u0,u0p,dx,dt,intervalx,intervalt,MAX_ITER,EPS)
%
% u = BurgersSolver1D(u0,u0p,dx,dt,intervalx,intervalt,MAX_ITER,EPS)
%
% Input:
%    u0        :   Functional handle for initial data: u(x,0)
%    u0p       :   Function handle, derivative of u0
%    dx        :   The distance between adjacent grid points in space
%    dt        :   The distance between adjacent grid points in time
%    intervalx :   2 x 1 vector for space domain of u(x,t)
%    intervaly :   2 x 1 vector for time domain of u(x,t)
%    MAX_ITER  :   An integer. Maximum number of iterations that is allowed
%                  for the Newton's iteration (Suggested 20)
%    EPS       :   Accuracy of Newton's method. (Suggested 1e-05)
%
% Output:
%    u         :   Nt x Nx matrix of numerical approximation of the
%                  solution of Burgers' Equation in 1D. Here Nt and Nx are
%                  the dimensions of discretization space of inervalx and
%                  intervaly
%
% Last update: February 25, 2018

% setting up the discrete problem
xx = intervalx(1) : dx : intervalx(2);
tt = intervalt(1)+dt : dt : intervalt(2)-dt;
% looping over x&t to find calculate solution u(x,t)
Nx = length(xx);
Nt = length(tt);
u = zeros(Nt,Nx);
for nt = 1 : Nt
    for nx = 1 : Nx
        f = @(x) tt(nt) * u0 (x) + x - xx(nx);
        fp = @(x) tt(nt) * u0p (x) + 1;
        if nx == 1
            x0 = xx(nx);
        else
            x0 = xstar;
        end
        % using Newton's method to find the solution to
        % (x - xstar)/(t - 0) = u0(xstar)
        xstar = NewtonSolution1D(f,fp,x0,MAX_ITER,EPS);
        u(nt,nx) = u0(xstar);
    end
end
```

```
function u = BurgersSolver1DPar(u0,u0p,dx,dt,intervalx,intervalt,MAX_ITER,EPS)
%
% u = BurgersSolver1DPar(u0,u0p,dx,dt,intervalx,intervalt,MAX_ITER,EPS)
%
% Input:
%    u0        :   Functional handle for initial data: u(x,0)
%    u0p       :   Function handle, derivative of u0
%    dx        :   The distance between adjacent grid points in space
%    dt        :   The distance between adjacent grid points in time
%    intervalx :   2 x 1 vector for space domain of u(x,t)
%    intervalt :   2 x 1 vector for time domain of u(x,t)
%    MAX_ITER   :   An integer. Maximum number of iterations that is allowed
%                   for the Newton's iteration (Suggested 20)
%    EPS        :   Accuracy of Newton's method. (Suggested 1e-05)
%
% Output:
%    u          :   Nt x Nx matrix of numerical approximation of the
%                   solution of Burgers' Equation in 1D. Here Nt and Nx are
%                   the dimensions of discretization space of inervalx and
%                   intervaly
%
% Last update: March 13, 2018

% setting up the discrete problem
xx = intervalx(1) : dx : intervalx(2);
tt = intervalt(1)+dt : dt : intervalt(2);
% looping over x&t to find calculate solution u(x,t)
Nx = length(xx);
Nt = length(tt);
u = zeros(Nt,Nx);
xx1=xx(1);
parfor nt = 1 : Nt
    v = zeros(1,Nx);
    x0 = xx1;
    f = @(x) tt(nt) * u0 (x) + x - xx1;
    fp = @(x) tt(nt) * u0p (x) + 1;
    % using Newton's method to find the solution to
    % (x - xstar)/(t - 0) = u0(xstar)
    xstar = NewtonSolution1D(f,fp,x0,MAX_ITER,EPS);
    v(1) = u0(xstar);
    for nx = 2 : Nx
        f = @(x) tt(nt) * u0 (x) + x - xx(nx);
        fp = @(x) tt(nt) * u0p (x) + 1;
        x0 = xstar;
        % using Newton's method to find the solution to
        % (x - xstar)/(t - 0) = u0(xstar)
        xstar = NewtonSolution1D(f,fp,x0,MAX_ITER,EPS);
        v(nx) = u0(xstar);
    end
    u(nt,:) = v;
end
```

4 Tools for wave problems

4.1 Newton's iteration

To find the solution of $f(x) = 0, \quad x \in \mathbb{R}$:

- Start from a logical point x_0
- At step $n \geq 1$ compute the search direction

$$d = -f(x_n)/f'(x_n)$$

- To avoid overshooting the search direction, we do a line search. Define

$$h(\lambda) = \frac{d}{d\lambda}(f(x_n + \lambda d))^2$$

  - Start with $\lambda_1 = 0$ and $\lambda_2 = \lambda_{\text{MAX}}$
  - At each step check $h((\lambda_1 + \lambda_2)/2)$

  - If positive set $\lambda_2 \leftarrow (\lambda_1 + \lambda_2)/2$
  - If negative set $\lambda_1 \leftarrow (\lambda_1 + \lambda_2)/2$

  This line search minimizes $|f(x_n + \lambda d)|$.

This can be achieved by `NewtonSolution1D.m`. The iteration stops

- If it reaches maximum number of iterations given by the user
- If it achieves $x_\star$ such that $|f(x_\star) < \epsilon|$ where ϵ is given by the user

```
function xstar = NewtonSolution1D(f,fp,x0,MAX_ITER,EPS)
%
% xstar = NewtonSolution1D(f,fp,x0,MAX_ITER,EPS)
%
% Input:
%    f          :   Function handle of a single variable
%    f          :   Function handle, derivative of f
%    MAX_ITER   :   Maximum number of iteration
%    EPS        :   Accuracy. (See the description of output)
%    x0         :   Initial guess for the zero of the function f.
%
% Output:
%    xtars      :   Numerical approximation of a zero of f such that
%                   either |f(xstar)| < EPS or the approximation at the
%                   iteration number max_iteration
%
% Last update: February 25, 2018
%
MAX_LUP=1;
xn=x0;
for n = 1:MAX_ITER
    d = -f(xn)/fp(xn); % direction of the solution
    % line search
    Lup = MAX_LUP; % max search distance
    Llo = 0; % minimum search distance
    h = @(lambda) 2*d*f(xn + lambda*d)*fp(xn+lambda*d);
    % derivative of the minimizer: f^2(xn + lambda*d)
    % looping to minimize h(L)
    for i = 1:MAX_ITER
        L = (Lup + Llo)/2;
        if h(L) > 0
            Lup = L;
        elseif h(L) < 0
            Llo = L;
        end

        if abs(h(L)) < EPS
            break
        end
    end
    xn = xn + d*L; % approximation after line search
    if abs(f(xn))< EPS
        break
    end
end
xstar = xn;
return
```

4.2 Runge-Kutta method

The goal of `RungeKuttaSolver.m` is solving an ODE in the form of

$$\frac{d}{dt}y = f(y,t).$$

Currently we only implement the third order Runge-Kutta which reads as

$$\begin{aligned} y^{(1)} &= y^n + \Delta t f(y^n) \\ y^{(2)} &= \frac{3}{4}y^n + \frac{1}{4}y^{(1)} + \frac{\Delta t}{4}f(y^{(1)}) \\ y^{n+1} &= \frac{1}{3}y^n + \frac{2}{3}y^{(2)} + \frac{2\Delta t}{3}f(y^{(2)}) \end{aligned}$$

```
function ynp1 = RungeKuttaSolver(yn,f,dt,order)
%
% ynp1 = RungeKuttaSolver(yn,f,dt,order)
%
% Implements one step of Runge Kutta method for the ODE
% d/dt y = f(y,t)
%
%
% Input:
%    yn     :   N x 1 vector, current time step approximation
%    f      :   Vectorized function handle
%    dt     :   Real number, step length
%    order  :   Integer
%
% Output:
%    ynp1   :   N x 1 vector, next time step approximation
%
% Last update: April 23, 2018

switch order
    case 3
        y1 = yn + dt*f(yn);
        y2 = (3/4)*yn + (1/4)*y1 + (dt/4)*f(y1);
        ynp1 = (1/3)*yn + (2/3)*y2 + (2*dt/3)*f(y2);
end
```

5 Simulations

5.1 Exact solution of Burger's equation

By setting the following parameters

- Initial condition and its derivative: `u0, u0p`
- Space interval : `intervalx`
- Final time : `Tstar`
- Space and time step sizes : `dt, dx`

`Script_BurgersSolver1D.m` plots the animated solution of the Burger's Equation described in Section 3.1.

```
% Script to test Burgers Solver
% Last update: March 14, 2018
close all; clear; clc; tic;
% initial condition
u0 = @(x) sin(x);
u0p = @(x) cos(x);
figure_option = 2;
% 1 for plot of solutions at some given time steps
% 2 for plotting an animated solution
% discretization parameteres
intervalx = [-7 7];
dx=0.1; % space discretization
xx = intervalx(1) : dx : intervalx(2);
Tstar = 1; % the time where shock developes
intervalt = [0 Tstar];
dt=0.01; % time discretization
MAX_ITER = 20; % Max number of iterarions for Newton's method which occurs
EPS = 1e-14; % Accuracy of Newton's method
% in the solver
u = BurgersSolver1DPar (u0,u0p,dx,dt,intervalx,intervalt,MAX_ITER,EPS);
switch figure_option
    case 1
        time_vec = [Tstar/3, Tstar/2, 2*Tstar/3, 4*Tstar/5];
        time_ind = floor(time_vec/dt);
        plot(xx, u(time_ind(1),:),xx, u(time_ind(2),:),xx, ...
            u(time_ind(3),:),xx, u(time_ind(4),:));
        title('Burgers Solution','interpreter','latex')
        str1 = '$u(x,1/3)$'; str2 = '$u(x,1/2)$'; str3 = '$u(x,2/3)$';
        str4 = '$u(x,4/5)$';
        h = legend(str1,str2,str3,str4,'Location','northeast');
        set(h,'interpreter','latex')
        set(gca,'FontSize',16);
        xlim(intervalx);
    case 2
        tt = dt : dt : Tstar;
        Nt = size(tt,2);
        figure
        for nt = 1 : Nt
            plot(xx, u(nt,:));
            title(['time = ' num2str(tt(nt))]);
            pause(0.01);
        end
end
toc;
```

5.2 Other non-linear waves

Numerical approximation of the waves described in Section 2 can be simulated via the script `Script_LaxFriedrichsSolver1D.m`. The script depends on the following parameters

- Nonlinear term and its derivative : `f, fp`
- Initial condition, its integral and derivative : `u0, U0, u0p`
- Refinements : `refienement_vector`
- Final time : `finalT`
- CFL condition ratio : `CFL`

- Display option : `display_option` helps to either compute the order of convergence, or plot the animated numerical solution vs. exact solution

```
% Script to test LaxFriedrichsSolver3rOrder1D
% Last update: April 24, 2018
% Solves Solves u_t + f(u)_x = 0    for x in intervalx, t in [0,T]
% with periodic boundary conditions
% Using third order space and time approximations

clear;close all;clc;tic;
% PARAMETERS
f = @(u) u.^2/2;
fp = @(u) u; % derivative of f
intervalx = [-pi pi]; % interval of x
% initial condition
initial_option = 1; % (1) sin x
scheme_option = 4;  % (3) 3rd order
                    % (4) 3rd order with minmod correction
display_option = 2;% (1) error and order of convergence after refinements
                   % (2) plotting an animated solution vs exact solution
                   % (3) plotting refined solutions at the final time step
switch initial_option
    case 1
        u0 = @(x) sin(x);
        U0 = @(x) -cos(x);
        u0p = @(x) cos(x);
end
% discretization parameteres
refinement_size = [40,80,160,320,640];
dx_vector=(intervalx(2)-intervalx(1))./refinement_size;
finalT = 0.5; % the final time
CFL=0.5;
dt_vector=CFL*dx_vector; % time discretization
MAX_ITER = 40; % Max number of iterarions for Newton's method
EPS = 1e-15; % Accuracy of Newton's method

% SOLVER
err_vector=[];
for ind=1:length(dx_vector)
    dx=dx_vector(ind);
    xx = intervalx(1)+dx/2 : dx : intervalx(2)-dx/2;
    dt=dt_vector(ind);
    numFinalT=(floor(finalT/dt))*dt;
    if display_option≠2
        startT=numFinalT-dt;
    else
        startT=0;
    end
    if initial_option == 1
        intervalxP = [intervalx(1)+dx/2 intervalx(2)-dx/2];
        uexacNhalf = BurgersSolver1DPar (u0,u0p,dx,dt,intervalxP,...
            [startT,numFinalT],MAX_ITER,EPS);
        uexacN = BurgersSolver1DPar (u0,u0p,dx,dt,intervalx,...
            [startT,numFinalT],MAX_ITER,EPS);
        % third order numerical integration to approximate cell average
        uexac = (1/6)*(uexacN(:,1:end-1) + uexacN(:,2:end) + 4*uexacNhalf);
    end
    uh = LaxFriedrichsSolver1D(U0,f,fp,dx,dt,intervalx,finalT,scheme_option);
    error=dx*sum(abs(uh(end,:)-uexac(end,:)));
    err_vector=[err_vector;error];%#ok
    if display_option==3
```

```
        uh_plot{ind}=uh(end,:);%#ok
    end
end

% DISPLAY
disp('Error and the order of convergence:')
%order = log(err_vector(1:end-1)./err_vector(2:end))/log(2);
order = 0.5*err_vector(1:end-1)./err_vector(2:end);
order = [0;order];
disp([err_vector order]);

switch display_option
    case 2
        disp('Solution for the refined mesh')
        tt = dt : dt : finalT;
        figure
        for nt = 1 : size(uh,1)
            if initial_option == 1
                plot(xx, uh(nt,:),xx,uexac(nt,:));
            end
            title(['time = ' num2str(tt(nt))]);
            pause(0.1);
        end
        legend('Numerical','Exact');
    case 3
        figure;
        for ind=1:length(dx_vector)
            dx=dx_vector(ind);
            xx = intervalx(1)+dx/2 : dx : intervalx(2)-dx/2;
            plot(xx, uh_plot{ind});
            hold on;
            strN = num2str(floor((intervalx(2)-intervalx(1))/dx_vector(ind)));
            legends{ind}=['N = ' strN];%#ok
        end
        title(['Numerical solution using minmod at time T=' num2str(numFinalT)]);
        legend(legends,'Location','southeast');
        xlabel('$x$','interpreter','latex');
        ylabel('Cell averages')
        set(gca,'FontSize',16)
end
toc;
```