Homework #5

Sam Fleischer

March 24, 2017

Problem 1

In one spatial dimension the linearized equations of acoustics (sound waves) are

$$p_t + K u_x = 0$$
$$\rho u_t + p_x = 0,$$

where u is the velocity and p is the pressure, ρ is the density, and K is the bulk modulus.

(a) Show that this system is hyperbolic and find the wave speeds.

(b) Write a program to solve this system using Lax-Wendroff in original variables on $(0,1)$ using a cell centered grid $x_j = \left(j - 1/2\right)\Delta x$ for $j = 1,\ldots,N$. Write the code to use ghost cells, so that different boundary conditions can be changed by simply changing the values in the ghost cells. Ghost cells outside the domain whose values can be set at the beginning of a time step so that code for updating cells adjacent to the boundary is identical to the code for interior cells.

Set the ghost cells at the left by

$$p_0^n = p_1^n$$
$$u_0^n = -u_1^n,$$

and set the ghost cells on the right by

$$p_{N+1}^n = \frac{1}{2}\left(p_N^n + u_N^n\sqrt{K\rho}\right)$$
$$u_{N+1}^n = \frac{1}{2}\left(\frac{p_N^n}{\sqrt{K\rho}} + u_N^n\right).$$

Run simulations with different initial conditions. Explain what happens at the left and right boundaries.

(c) Give a physical interpretation and a mathematical explanation of these boundary conditions.

(a) We can write the system

$$p_t + K u_x = 0$$
$$\rho u_t + p_x = 0$$

in the following form

$$\begin{bmatrix} 1 & 0 \\ 0 & \rho \end{bmatrix}\begin{bmatrix} p \\ u \end{bmatrix}_t + \begin{bmatrix} 0 & K \\ 1 & 0 \end{bmatrix}\begin{bmatrix} p \\ u \end{bmatrix}_x = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Then, defining $B := \begin{bmatrix} 1 & 0 \\ 0 & \rho \end{bmatrix}$, $C := \begin{bmatrix} 0 & K \\ 1 & 0 \end{bmatrix}$, and $\vec{u} := \begin{bmatrix} p \\ u \end{bmatrix}$, we can write

$$B\vec{u}_t + C\vec{u}_x = \vec{0}$$

Since B is a diagonal matrix with no 0s on the diagonal, B^{-1} exists and we can write

$$\vec{u}_t + A\vec{u}_x = \vec{0}$$

where $A := B^{-1}C = \begin{bmatrix} 0 & K \\ \rho^{-1} & 0 \end{bmatrix}$.

$$\lambda_{1,2} = \pm\sqrt{K\rho^{-1}}.$$

Since ρ and K are positive, $\lambda_{1,2} \in \mathbb{R}$. These eigenvalues are the wavespeeds in the direction of their corresponding eigenvectors. The eigenvector corresponding to $\lambda_1 = \sqrt{K\rho^{-1}}$ is $v_1 = \begin{bmatrix} \sqrt{K\rho} \\ 1 \end{bmatrix}$ and the eigenvector corresponding to $\lambda_2 = -\sqrt{K\rho^{-1}}$ is $v_2 = \begin{bmatrix} -\sqrt{K\rho} \\ 1 \end{bmatrix}$.

(b) We can write the recursion

$$\vec{u}_j^{n+1} = \vec{u}_j^n - \frac{\Delta t}{2\Delta x} A\left(\vec{u}_{j+1}^n - \vec{u}_{j-1}^n\right) + \frac{\Delta t^2}{2\Delta x^2} A^2\left(\vec{u}_{j+1}^n - 2\vec{u}_j^n + \vec{u}_{j-1}^n\right),$$

as

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

For $j = 1$ and $j = N$, we have

$$p_1^{n+1} = p_1^n - \frac{\Delta t}{2\Delta x} K\left(u_2^n + u_1^n\right) + \frac{\Delta t^2}{2\Delta x^2} \frac{K}{\rho}\left(p_2^n - 2p_1^n + p_1^n\right)$$
$$p_N^{j+1} = p_N^n - \frac{\Delta t}{2\Delta x} K\left(\frac{1}{2}\left(\frac{p_N^n}{\sqrt{K\rho}} + u_N^n\right) - u_{N-1}^n\right) + \frac{\Delta t^2}{2\Delta x^2} \frac{K}{\rho}\left(\frac{1}{2}\left(p_N^n + u_N^n\sqrt{K\rho}\right) - 2p_N^n + p_{N-1}^n\right)$$

and

$$u_1^{n+1} = u_1^n - \frac{\Delta t}{2\Delta x} \frac{1}{\rho}\left(p_2^n - p_1^n\right) + \frac{\Delta t^2}{2\Delta x^2} \frac{K}{\rho}\left(u_2^n - 2u_1^n - u_1^n\right)$$
$$u_N^{j+1} = u_N^n - \frac{\Delta t}{2\Delta x} \frac{1}{\rho}\left(\frac{1}{2}\left(p_N^n + u_N^n\sqrt{K\rho}\right) - p_{N-1}^n\right) + \frac{\Delta t^2}{2\Delta x^2} \frac{K}{\rho}\left(\frac{1}{2}\left(\frac{p_N^n}{\sqrt{K\rho}} + u_N^n\right) - 2u_N^n + u_{N-1}^n\right)$$

This means we can write the recursion as

$$p^{n+1} = A_1 p^n + A_2 u^n$$
$$u^{n+1} = B_1 p^n + B_2 u^n$$

where

$$A_1 = \begin{bmatrix} 1 - \frac{\Delta t^2}{2\Delta x^2}\frac{K}{\rho} & \frac{\Delta t^2}{2\Delta x^2}\frac{K}{\rho} & & & & & \\ \frac{\Delta t^2}{2\Delta x^2}\frac{K}{\rho} & 1 - \frac{\Delta t^2}{\Delta x^2}\frac{K}{\rho} & \frac{\Delta t^2}{2\Delta x^2}\frac{K}{\rho} & & & & \\ & \frac{\Delta t^2}{2\Delta x^2}\frac{K}{\rho} & 1 - \frac{\Delta t^2}{\Delta x^2}\frac{K}{\rho} & \ddots & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & 1 - \frac{\Delta t^2}{\Delta x^2}\frac{K}{\rho} & \frac{\Delta t^2}{2\Delta x^2}\frac{K}{\rho} & \\ & & & & \frac{\Delta t^2}{2\Delta x^2}\frac{K}{\rho} & 1 - \frac{3\Delta t^2}{4\Delta x^2}\frac{K}{\rho} - \frac{\Delta t}{4\Delta x}\frac{\sqrt{K}}{\sqrt{\rho}} \end{bmatrix},$$

$$A_2 = \begin{bmatrix} -\frac{\Delta t}{2\Delta x}K & -\frac{\Delta t}{2\Delta x}K & & & & & \\ \frac{\Delta t}{2\Delta x}K & 0 & -\frac{\Delta t}{2\Delta x}K & & & & \\ & \frac{\Delta t}{2\Delta x}K & 0 & \ddots & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & 0 & -\frac{\Delta t}{2\Delta x}K & \\ & & & & \frac{\Delta t}{2\Delta x}K & -\frac{\Delta t}{4\Delta x}K + \frac{\Delta t^2}{4\Delta x^2}\frac{K\sqrt{K}}{\sqrt{\rho}} \end{bmatrix},$$

$$B_1 = \begin{bmatrix} \frac{\Delta t}{2\Delta x}\frac{1}{\rho} & -\frac{\Delta t}{2\Delta x}\frac{1}{\rho} & & & & & \\ \frac{\Delta t}{2\Delta x}\frac{1}{\rho} & 0 & -\frac{\Delta t}{2\Delta x}\frac{1}{\rho} & & & & \\ & \frac{\Delta t}{2\Delta x}\frac{1}{\rho} & 0 & \ddots & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & 0 & -\frac{\Delta t}{2\Delta x}\frac{1}{\rho} & \\ & & & & \frac{\Delta t}{2\Delta x}\frac{1}{\rho} & -\frac{\Delta t}{4\Delta x}\frac{1}{\rho} + \frac{\Delta t^2}{4\Delta x^2}\frac{\sqrt{K}}{\rho\sqrt{\rho}} \end{bmatrix}, \quad \text{and}$$

$$B_2 = \begin{bmatrix} 1 - \frac{3\Delta t^2}{2\Delta x^2}\frac{K}{\rho} & \frac{\Delta t^2}{2\Delta x^2}\frac{K}{\rho} & & & & & \\ \frac{\Delta t^2}{2\Delta x^2}\frac{K}{\rho} & 1 - \frac{\Delta t^2}{\Delta x^2}\frac{K}{\rho} & \frac{\Delta t^2}{2\Delta x^2}\frac{K}{\rho} & & & & \\ & \frac{\Delta t^2}{2\Delta x^2}\frac{K}{\rho} & 1 - \frac{\Delta t^2}{\Delta x^2}\frac{K}{\rho} & \ddots & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & 1 - \frac{\Delta t^2}{\Delta x^2}\frac{K}{\rho} & \frac{\Delta t^2}{2\Delta x^2}\frac{K}{\rho} & \\ & & & & \frac{\Delta t^2}{2\Delta x^2}\frac{K}{\rho} & 1 - \frac{\Delta t}{4\Delta x}\frac{\sqrt{K}}{\sqrt{\rho}} - \frac{3\Delta t^2}{4\Delta x^2}\frac{K}{\rho} \end{bmatrix}.$$

So, I wrote functions to create these matrices:

```
def get_A1_mat(N,K,rho,dx,dt):
    diag = (1 - ((dt**2)*K)/((dx**2)*rho))*np.ones(N)
    diag[0] = 1 - ((dt**2)*K)/(2*(dx**2)*rho)
    diag[-1] = 1 - ((3*(dt**2)*K)/(4*(dx**2)*rho)) \
        - ((dt*np.sqrt(K))/(4*dx*np.sqrt(rho)))
    subdiag = ((dt**2)*K)/(2*(dx**2)*rho)*np.ones(N)
    superdiag = ((dt**2)*K)/(2*(dx**2)*rho)*np.ones(N)
    A1 = np.vstack((superdiag,diag,subdiag))
    A1_mat = sp.dia_matrix((A1,[1,0,-1]),shape=(N,N))
    return A1_mat

def get_A2_mat(N,K,rho,dx,dt):
    diag = (0)*np.ones(N)
    diag[0] = (-dt*K)/(2*dx)
    diag[-1] = (((dt**2)*K*np.sqrt(K))/(4*(dx**2)*np.sqrt(rho))) \
        - ((dt*K)/(4*dx))
    subdiag = (dt*K)/(2*dx)*np.ones(N)
    superdiag = (-dt*K)/(2*dx)*np.ones(N)
    A2 = np.vstack((superdiag,diag,subdiag))
    A2_mat = sp.dia_matrix((A2,[1,0,-1]),shape=(N,N))
```

```
    return A2_mat

def get_B1_mat(N,K,rho,dx,dt):
    diag = (0)*np.ones(N)
    diag[0] = (dt)/(2*dx*rho)
    diag[-1] = (((dt**2)*np.sqrt(K))/(4*(dx**2)*rho*np.sqrt(rho))) \
        - ((dt)/(4*dx*rho))
    subdiag = (dt)/(2*dx*rho)*np.ones(N)
    superdiag = (-dt)/(2*dx*rho)*np.ones(N)
    B1 = np.vstack((superdiag,diag,subdiag))
    B1_mat = sp.dia_matrix((B1,[1,0,-1]),shape=(N,N))
    return B1_mat

def get_B2_mat(N,K,rho,dx,dt):
    diag = (1 - ((dt**2)*K)/((dx**2)*rho))*np.ones(N)
    diag[0] = 1 - (3*(dt**2)*K)/(2*(dx**2)*rho)
    diag[-1] = 1 - ((3*(dt**2)*K)/(4*(dx**2)*rho)) \
        - ((dt*np.sqrt(K))/(4*dx*np.sqrt(rho)))
    subdiag = ((dt**2)*K)/(2*(dx**2)*rho)*np.ones(N)
    superdiag = ((dt**2)*K)/(2*(dx**2)*rho)*np.ones(N)
    B2 = np.vstack((superdiag,diag,subdiag))
    B2_mat = sp.dia_matrix((B2,[1,0,-1]),shape=(N,N))
    return B2_mat
```

followed by a function which generates the recursion step given $\vec{u}_j$:

```
def make_LW_method(N,K,rho,dx,dt):
    # Get A_1, A_2, B_1, and B_2
    A1 = get_A1_mat(N,K,rho,dx,dt)
    A2 = get_A2_mat(N,K,rho,dx,dt)
    B1 = get_B1_mat(N,K,rho,dx,dt)
    B2 = get_B2_mat(N,K,rho,dx,dt)

    # Make the Lax-Wendroff recursion step
    def LW_step(p,u):
        p_next = A1.dot(p) + A2.dot(u)
        u_next = B1.dot(p) + B2.dot(u)
        return (p_next,u_next)

    # Return the function
    return LW_step
```

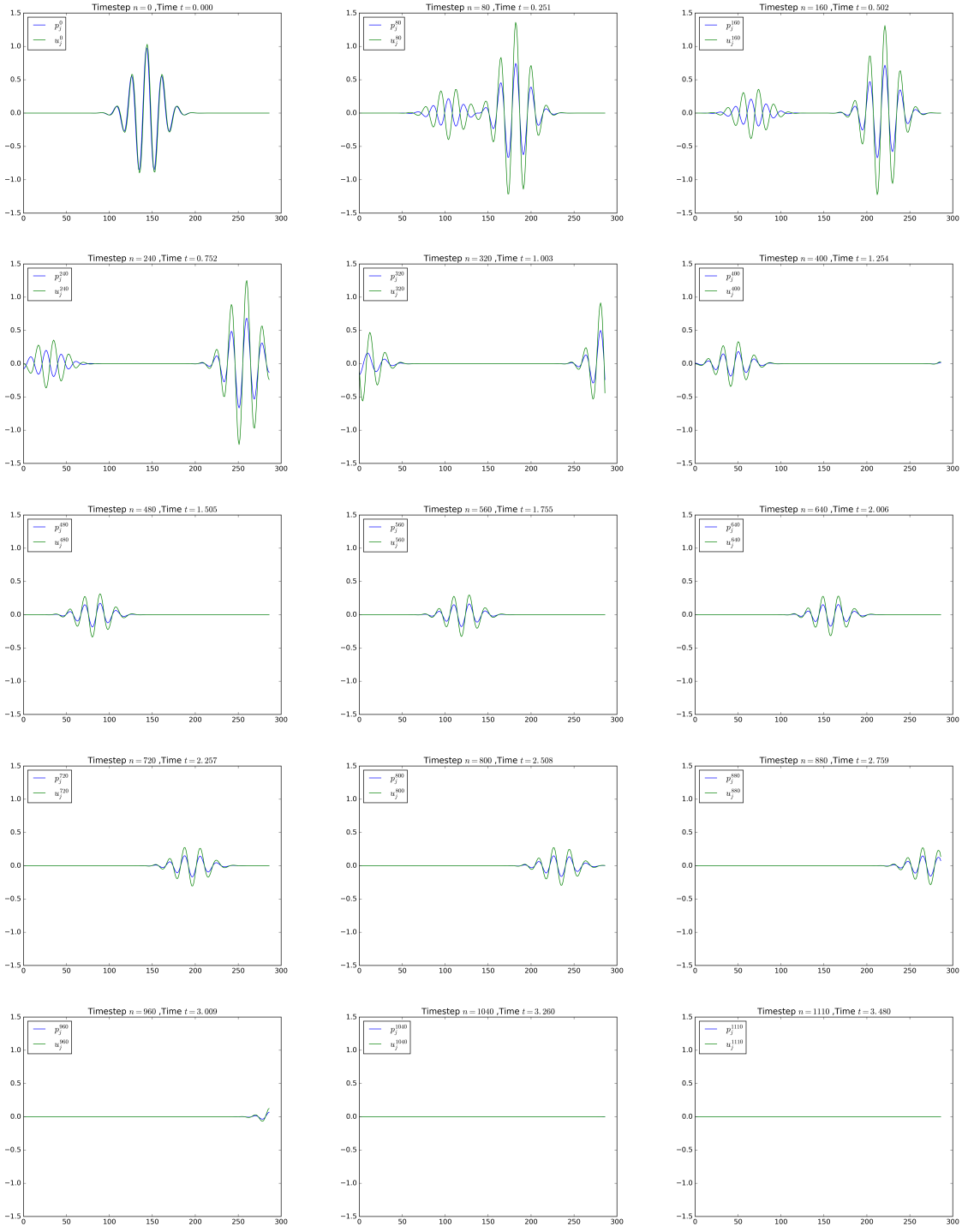
I recursively called `LW_step` for different initial conditions (provided below in Problem 2):

(a) Wave packet: $u(x,0) = \cos(32\pi x)\exp\left(-150(x-0.5)^2\right)$

(b) Smooth, low frequency: $u(x,0) = \sin(2\pi x)\sin(4\pi x)$

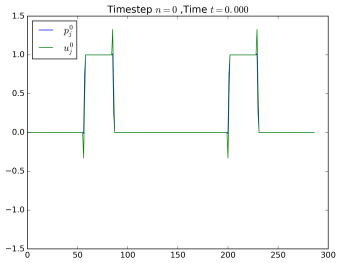
(c) Step function: $u(x,0) = u(x,0) = \begin{cases} 1 & \text{if } |x-1/4| < \frac{1}{20} \text{ or } |x-3/4| < \frac{1}{20} \\ 0 & \text{otherwise} \end{cases}$

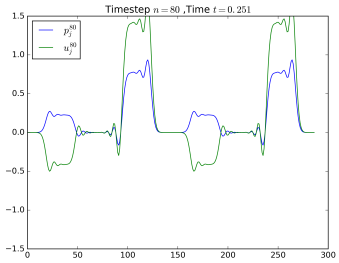
The following figures are from simulations with the wave packet as the initial condition for both u and p.

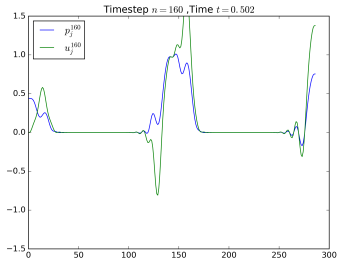


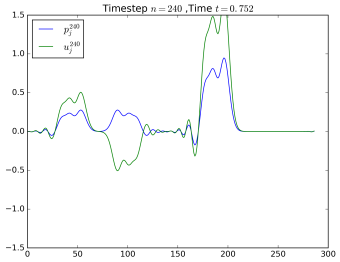
Clearly we see the waves move to both the left and right, but the wave to the left is significantly smaller. When the large wave hits the right border, it is dissipated in to a very small wave moving left, but most of the "stuff" leaves the system. When the smaller wave hits the left border, it bounces back and moves right. Notice the u wave is flipped when it hits the left side since we are assuming the boundary condition $u_0^n = -u_1^n$. Finally, the small wave endures the same fate as the initial larger wave: it leaves the system through the left border.

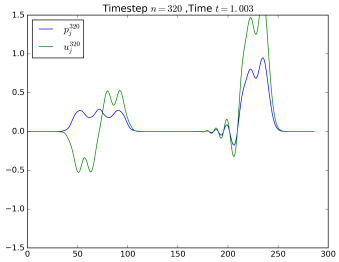
The following figures show the movement of a step function. The same general behavior is seen in this simulation. Note that the step function shape is not well-preserved because Lax-Wendroff is not a good method for discontinuous data (see Problem 2).

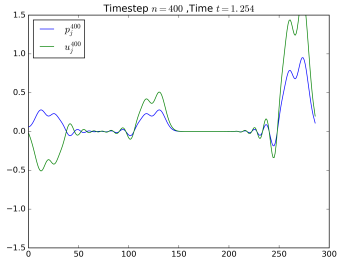


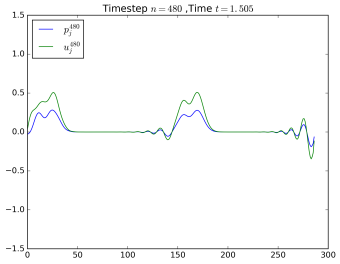


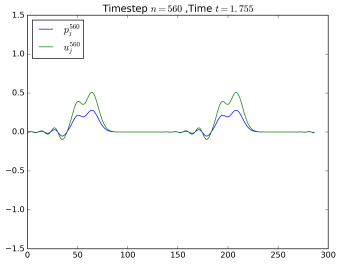


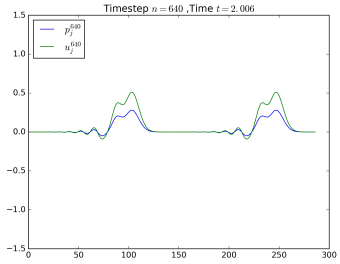


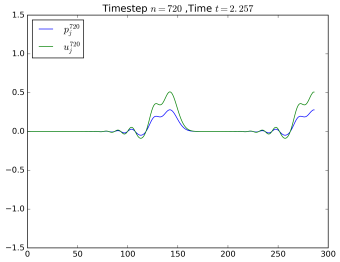


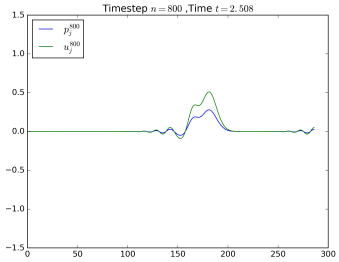


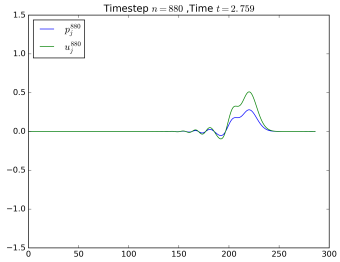


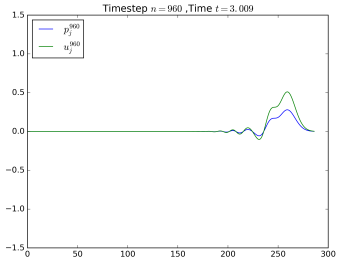


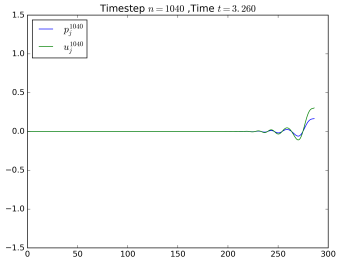


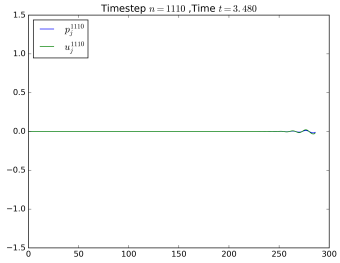




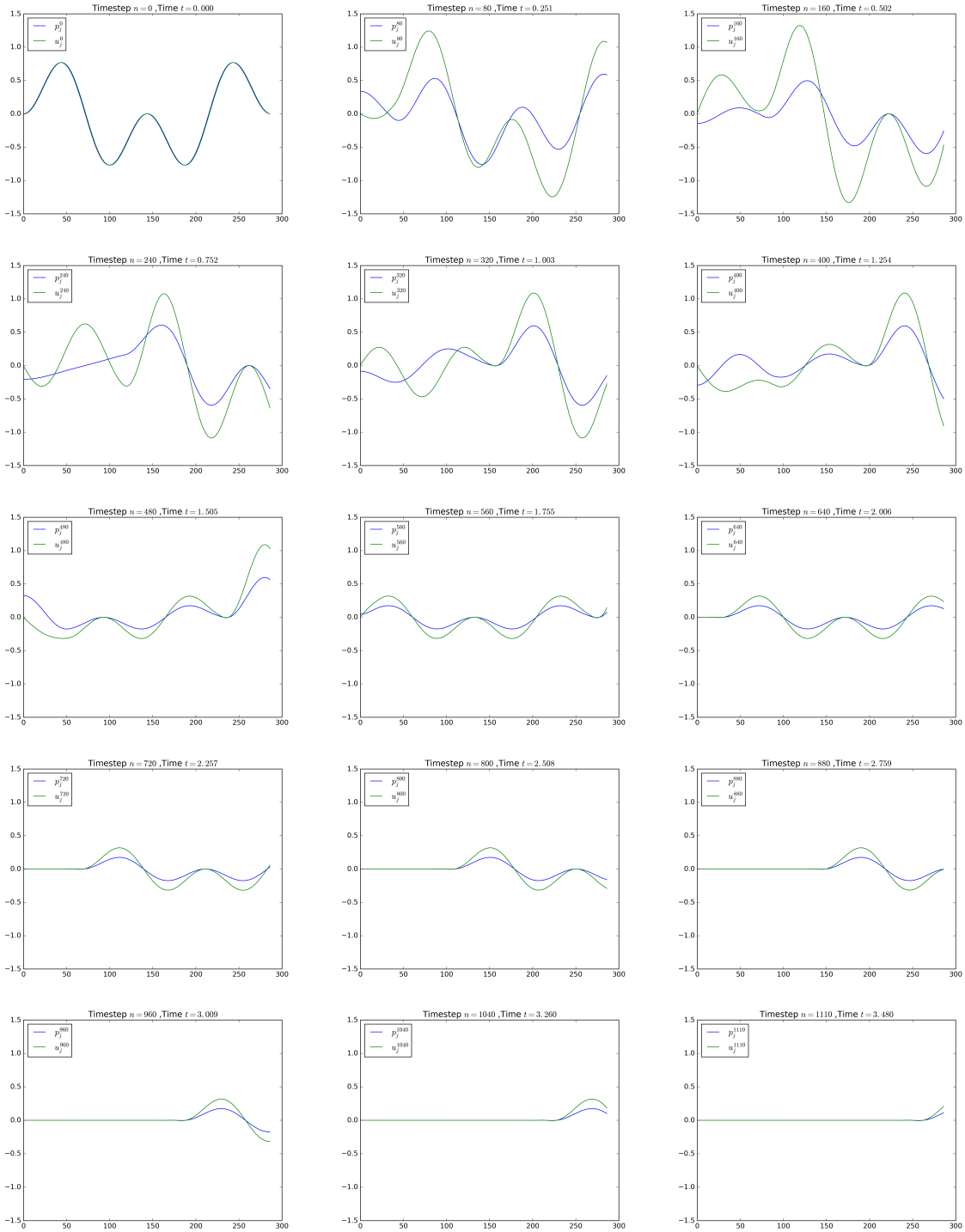








The initial behavior of this system is much more difficult to see for low frequency initial data which spans the entire region. However we do see a diffused initial condition (with u turned upside-down) move to the right, just like the above simulations, once the large wave exits the right side and the small wave is finished bouncing off the left side. Here are the results:



(c) Since the matrix A in the system $\vec{u}_t + A\vec{u}_x = \vec{0}$ is diagonalizable, we can write

$$\vec{v}_t + \Lambda \vec{v}_x = \vec{0}$$

where

$$\Lambda = \begin{bmatrix} \sqrt{K\rho^{-1}} & 0 \\ 0 & -\sqrt{K\rho^{-1}} \end{bmatrix}, \qquad \vec{v} = V^{-1}\vec{u}, \qquad \text{and} \quad V = \begin{bmatrix} \sqrt{K\rho} & -\sqrt{K\rho} \\ 1 & 1 \end{bmatrix}.$$

We can write the ghost cells on the right as

$$\vec{u}^n_{N+1} = G_R \vec{u}^n_N$$

where

$$G_R = \frac{1}{2}\begin{bmatrix} 1 & \sqrt{K\rho} \\ \frac{1}{\sqrt{K\rho}} & 1 \end{bmatrix}$$

Using $\vec{u} = V\vec{v}$, we have $V\vec{v}^n_{N+1} = G_R V \vec{v}^n_N$, or

$$\vec{v}^n_{N+1} = V^{-1} G_R V \vec{v}^n_N$$

However, $V^{-1}G_R V = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$, so the right side ghost cells in eigenspace are set by

$$(v_1)^n_{N+1} = (v_1)^n_N$$
$$(v_2)^n_{N+1} = 0$$

Since v_2 is the eigenvector with a negative wavespeed, we do not see anything bouncing off the right hand side like we do on the left. In other words, there is no inward flux of leftward-moving material on the right side. On the left side, we have

$$\vec{u}^n_0 = G_L \vec{u}^n_1$$

where

$$G_L = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

Using the same transformation, we get

$$\vec{v}^n_0 = V^{-1} G_L V \vec{v}^n_1$$

However, $V^{-1}G_L V = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$, so the left side ghost cells in eigenspace are set by

$$(v_1)^n_0 = -(v_2)^n_1$$
$$(v_2)^n_0 = -(v_1)^n_1$$

Since the material "bounces" off the left boundary, this description is intuitive since all left-moving material becomes right-moving material, and all right-moving material becomes left-moving material.

Problem 2

Write a program to solve the linear advection equation,

$$u_t + a u_x = 0,$$

on the unit interval using a finite volume method of the form

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

Use the numerical flux function

$$F_{j-1/2} = F^{\text{up}}_{j-1/2} + \frac{|a|}{2}\left(1 - \left|\frac{a\Delta t}{\Delta x}\right|\right)\delta_{j-1/2},$$

where $F^{\text{up}}_{j-1/2}$ is the upwinding flux,

$$F^{\text{up}}_{j-1/2} = \begin{cases} a u_{j-1} & \text{if } a > 0 \\ a u_j & \text{if } a < 0, \end{cases}$$

and $\delta_{j-1/2}$ is the limited difference. Let $\Delta u_{j-1/2} = u_j - u_{j-1}$ denote the jump in u across the edge at $x_{j-1/2}$. The limited difference is

$$\delta_{j-1/2} = \phi(\theta_{j-1/2})\Delta u_{j-1/2},$$

where

$$\theta_{j-1/2} = \frac{\Delta u_{J_{\text{up}}-1/2}}{\Delta u_{j-1/2}},$$

and

$$J_{\text{up}} = \begin{cases} j-1 & \text{if } a > 0 \\ j+1 & \text{if } a < 0 \end{cases}.$$

Note that you will need two ghost cells on each end of the domain. Write your program so that you may choose from the different limiter functions listed below.

(1) Upwinding $\phi(\theta) = 0$
(2) Lax-Wendroff $\phi(\theta) = 1$
(3) Beam-Warming $\phi(\theta) = \theta$
(4) minmod $\phi(\theta) = \text{minmod}(1, \theta)$
(5) superbee $\phi(\theta) = \max(0, \min(1, 2\theta), \min(2, \theta))$
(6) MC $\phi(\theta) = \max(0, \min(\frac{1+\theta}{2}, 2, 2\theta))$
(7) van Leer $\phi(\theta) = \dfrac{\theta + |\theta|}{1 + |\theta|}$

The first three are linear methods that we have already studied, and the last four are high-resolution methods. Solve the advection equation with $a = 1$ with periodic boundary conditions for the different initial conditions listed below until time $t = 5$ at Courant number 0.9.

(a) Wave packet: $u(x,0) = \cos(16\pi x)\exp\left(-50(x-0.5)^2\right)$
(b) Smooth, low frequency: $u(x,0) = \sin(2\pi x)\sin(4\pi x)$
(c) Step function: $u(x,0) = u(x,0) = \begin{cases} 1 & \text{if } |x - 1/2| < \frac{1}{4} \\ 0 & \text{otherwise} \end{cases}$

Compare the results with the exact solution, and comment on the solutions generated by the different methods. How do the different high-resolution methods perform in the different tests? What high-resolution method would you chose to use in practice?

First I defined the limiter function using a user-defined choice of ϕ.

```
1 def get_limiter_function(phi_choice):
2     # Define the flux limiter function based on user choice.
```

```
    if phi_choice == 1: # Upwinding
        def phi(theta):
            return 0
    elif phi_choice == 2: # Lax-Wendroff
        def phi(theta):
            return 1
    elif phi_choice == 3: # Beam-Warming
        def phi(theta):
            return theta
    elif phi_choice == 4: # minmod
        def phi(theta):
            return max(0,min(1,theta))
    elif phi_choice == 5: # superbee
        def phi(theta):
            return max(0,min(1,2*theta),min(2,theta))
    elif phi_choice == 6: # MC
        def phi(theta):
            return max(0,min((1+theta)/2,2,2*theta))
    elif phi_choice == 7:# Van Leer
        def phi(theta):
            return (theta + abs(theta))/(1 + abs(theta))
    # return the appropriate function
    return phi
```

Then I defined the index J_{up} based on the index j and flow direction a

```
def get_Jup(j,a):
    # Return the appropriate upwinding index for the given flow direction.
    if a >= 0:
        return j-1
    elif a < 0:
        return j+1
```

I defined a Δ function, given u and j:

```
def Delta(u, j):
    # This is just to match with notation in the homework
    return u[j] - u[j-1]
```

and used that to define $\theta_{j-1/2}$

```
def theta_j_m_half(u, j, a):
    # This returns the ratio of jumps across edges

    # First get the upwinding direction J_up
    Jup = get_Jup(j,a)
    # Then return the ratio of jumps
    return Delta(u, Jup)/Delta(u, j)
```

Next I defined the limited differences $\delta_{j-1/2}$

```
def get_delta(phi,u,j,a):
    # This function returns the limited difference delta at point j.
    D = Delta(u,j)
    # If the function is basically flat, |u_j - u_{j-1}|<1e-10, return 0
    if abs(D) < 1e-10:
        return 0
    # Otherwise, get the ratio of jumps across edges, and calculate the
```

```
    #   limited flux
    else:
        theta = theta_j_m_half(u, j, a)
        return phi(theta)*D
```

Then I defined the upwinding flux $F^{\text{up}}_{j-1/2}$

```
def get_F_up(u, j, a):
    # The upwinding flux depends on the direction of flow
    if a >= 0:
        return a*u[j-1]
    elif a < 0:
        return a*u[j]
```

and finally the numerical flux function $F_{j-1/2}$

```
def get_F_j_m_half(phi,u,j,a,dt,dx):
    # This function gets F_{j-1/2} for the given j

    # Get the limited difference delta:
    delta = get_delta(phi,u,j,a)
    # Get the upwinding flux
    F_up = get_F_up(u,j,a)
    # Return the numerical flux
    return F_up + (abs(a)/2)*(1 - abs((a*dt)/(dx)))*delta
```

The numerical flux function is used to define the recursion. I made a function `make_FV_method` to construct a recursive step.

```
def make_FV_method(phi_choice,a,dt,dx):
    # This function returns a simple iteration technique for
    #   a finite-volume method

    # First, get the flux limiter function based on the user
    #   choice of phi
    phi = get_limiter_function(phi_choice)

    def FV_step(u):
        # Initialize u_next
        u_next = np.zeros(u.shape)

        # Pad the input vector with period data from the other side
        u_padded = pad(u, (2,2), 'wrap')
        # Loop through the points
        for j in xrange(len(u_next)):
            # Get F_{j-1/2} and F_{j+1/2} (numerical flux).
            #   Note that I use j+2 and j+3 because of the padding
            F_j_m_half = get_F_j_m_half(phi,u_padded,j+2,a,dt,dx)
            F_j_p_half = get_F_j_m_half(phi,u_padded,j+3,a,dt,dx)

            # Get u^{n+1} = u^n - (dt/dx)(F_{j+1/2} - F_{j-1/2})
            u_next[j] = u_padded[j+2] - (dt/dx)*(F_j_p_half - F_j_m_half)
        return u_next

    # Return the function
    return FV_step
```

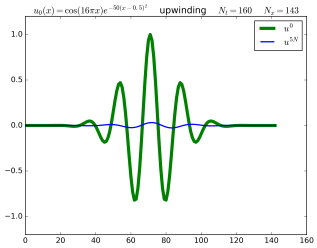
To actually use the code, I defined an initial condition function

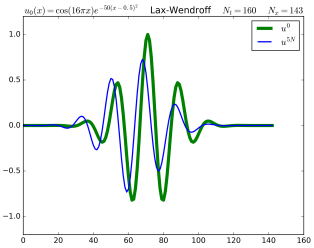
```
def IC(x,IC_choice):
    # Get the initial condition, given the choice
    if IC_choice == 1:
        # cos(16 pi x)exp(-50(x-.5)^2)
        IC = np.cos(16*np.pi*x)*np.exp(-50*(x-0.5)**2)
    elif IC_choice == 2:
        # sin(2 pi x)sin(4 pi x)
        IC = np.sin(2*np.pi*x)*np.sin(4*np.pi*x)
    elif IC_choice == 3:
        # step function, up at 1/4, down at 3/2
        IC = np.piecewise(x,[abs(x-0.5)<0.25,abs(x-0.5)>=0.25],[1,0])
    return IC
```

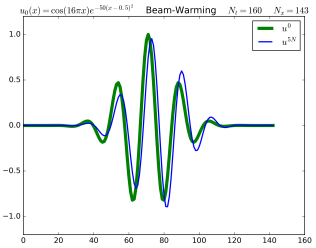
then rolled it all together with the following code:

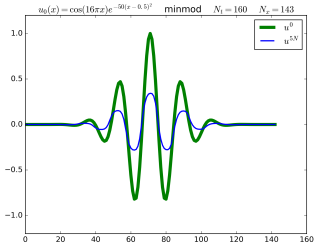
```
# Set N_t = 10*(2^4)
power = 4
Nt = 10*2**power
a = 1
# Since a = 1, set N_x = .9*(N_t - 1)
#   It is (N_t - 1) since we are using finite volume methods.
Nx = int(0.9*(Nt-1))
# Get dt and dx from N_t and N_x
dt = 1/(Nt-1)
dx = 1/Nx
# Set the final time
final_time = 5

# Define IC choice
IC_choice = 1
# Define phi choice
phi_choice = 2

# Define the recursion
FV_step = make_FV_method(phi_choice,a,dt,dx)

# Get initial condition
xaxis = np.linspace(dx/2,1-dx/2,Nx)
u_initial = IC(xaxis,IC_choice)
u_0 = u_initial + 0

# Do recursion
for t in xrange(int((Nt-1)*final_time)):
    u_1 = FV_step(u_0)
    u_0 = u_1 + 0
```

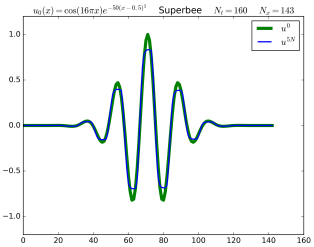
(a) Here are the results for all seven methods for $u_0(x) = \cos(16\pi x)\exp\left(-50(x-0.5)^2\right)$.

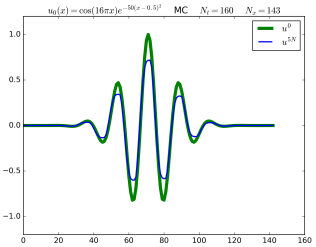


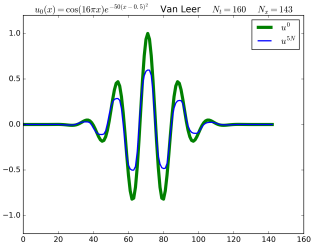


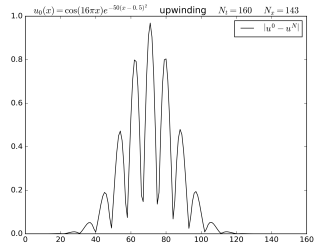


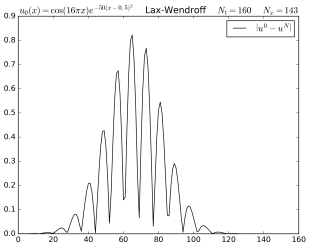


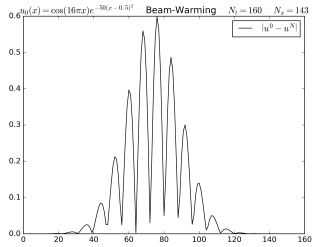


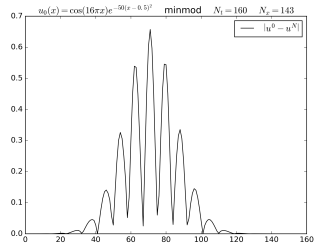


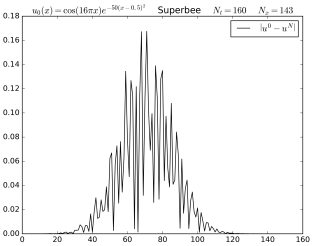


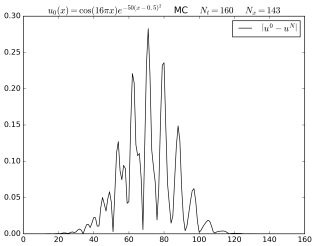


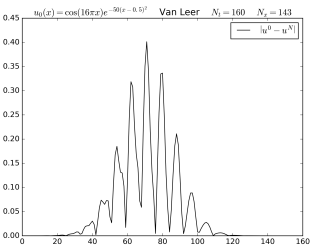




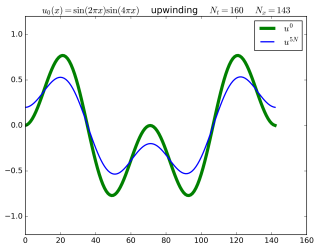


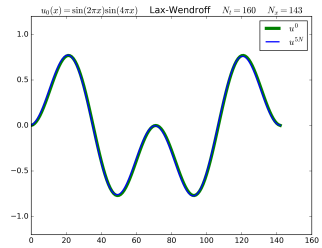


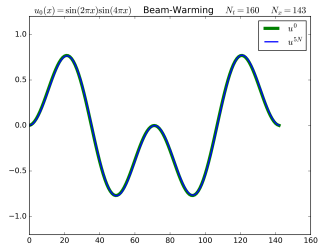


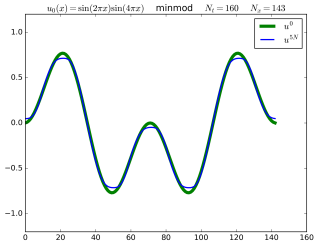


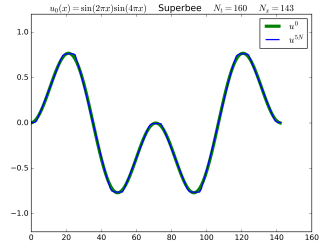
(b) Here are the results for all seven methods for $u_0(x) = \sin(2\pi x)\sin(4\pi x)$.

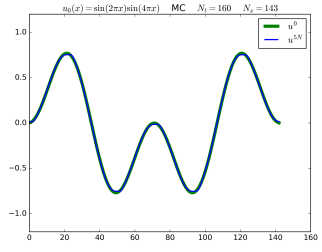


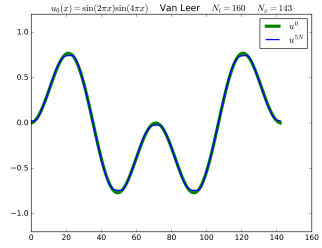


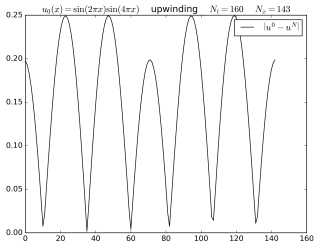


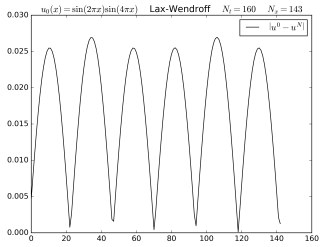


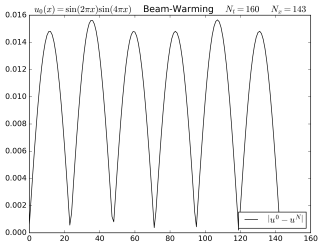


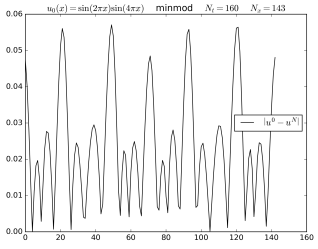


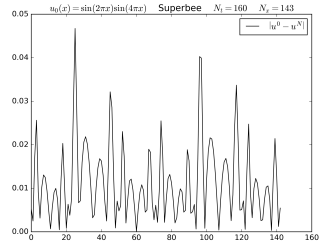


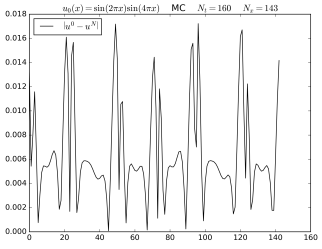


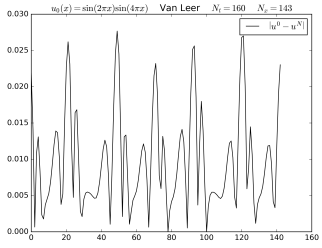




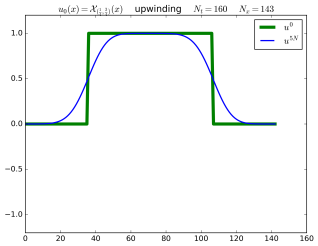


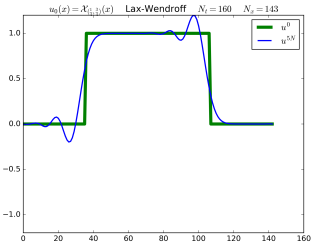


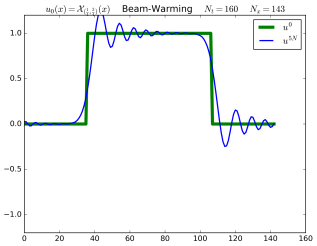


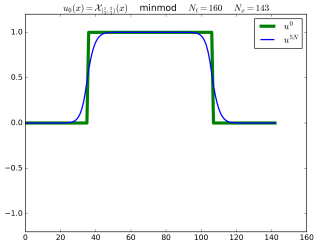


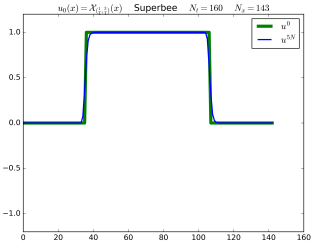
(c) Here are the results for all seven methods for $u_0(x) = \mathcal{X}_{(\frac{1}{4},\frac{3}{4})}(x)$.

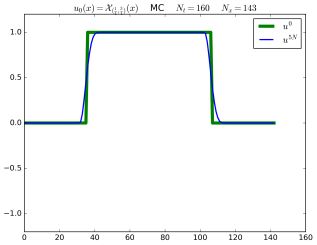


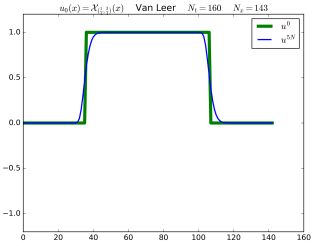


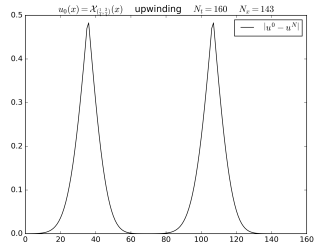


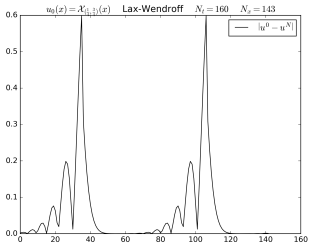


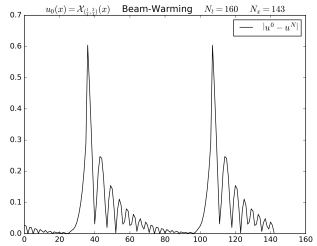


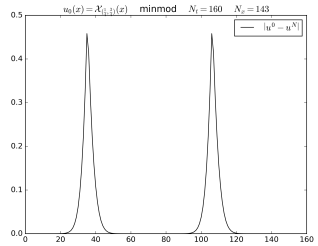


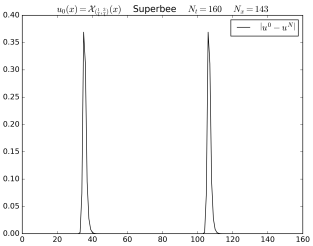


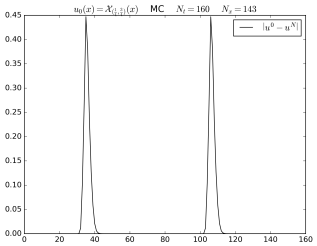


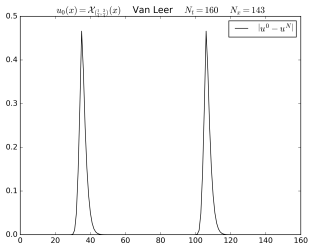












For the wave packet, we see upwinding and minmod are highly diffusive. Lax-Wendroff produces a phase lag along with diffusion, and Beam-Warming pushes the oscillations forward faster that it should. We see significant diffusion in Van Leer and MC, and a plateauing of data with Superbee. For this type of initial data, I would use Superbee since it has the smallest error out of all 7 methods for this initial data.

For the smooth, low-frequency data, Beam-Warming and MC have the lowest error. Van Leer and Superbee produce flat maxima and minima when they shouldn't (since the data is smooth). We still see significant diffusion with upwinding and minmod, and a slight phase error with Lax-Wendroff. For this type of initial data, I would use MC since the sharpening at the extrema is subdued.

For the step function, Superbee performs the best. Superbee does the best at making and/or maintaining sharp corners. Beam warming and Lax-Wendroff create wiggles to the right and left (respectively) of the discontinuities. Upwinding and minmod still produce significant diffusion.

In general, I would use Superbee if I knew my initial data was sharp or discontinuous, but I would use MC if I knew the data was smooth and low frequency.