Chapter 22

Partial Differential Equations II

22.1 First order time derivatives

The wave equation (21.5) for a vibrating string contains second-order time derivatives. It can be rewritten as a system of equations with first-order time derivatives by introducing a new variable $v = \dot{y}$, which is the transverse velocity of the string. The resulting system of PDEs

$$\dot{y} = v \ , \tag{22.1a}$$

$$\dot{v} = c^2 y'' \tag{22.1b}$$

is equivalent to the single equation $\ddot{y} = c^2 y''$.

Using the two–point forward difference formula (16.4), the first time derivative of y is approximated as

$$\dot{y}(t^n, x_j) \approx \frac{y_j^{n+1} - y_j^n}{\Delta t} \ . \tag{22.2}$$

An analogous approximation is made for $\dot{v}$. With these results and the three–point centered difference approximation for y'', we have

$$y_j^{n+1} = y_j^n + \Delta t \, v_j^n \ , \tag{22.3a}$$

$$v_j^{n+1} = v_j^n + \Delta t \, c^2 \left(\frac{y_{j+1}^n - 2y_j^n + y_{j-1}^n}{\Delta x^2} \right) \ . \tag{22.3b}$$

This is called the *forward–time, centered–space* (FTCS) discretization of the system (22.1).

The codes for the FTCS and CTCS algorithms are very similar. Here is an outline for an FTCS code:

- Import NumPy (as `np`) and other libraries.
- Choose values for the parameters.
- Set up the numerical grid as in the CTCS code.

- Define arrays:

```
y = np.zeros(J+1)      # y at time t^n
v = np.zeros(J+1)      # v at time t^n
yp = np.zeros(J+1)     # yp = "y plus" = y at time t^(n+1)
vp = np.zeros(J+1)     # vp = "v plus" = v at time t^(n+1)
```

- Choose initial data:

```
y = some function of x
v = some function of x
```

- Evolve the system forward in time using the FTCS scheme (22.3):

```
for n in range(1,N+1):
   for j in range(1,J):
      yp[j] = y[j] + dt*v[j]
      vp[j] = v[j] + (dt*c**2/dx**2)\
         *(y[j+1] - 2.0*y[j] + y[j-1])
   yp[0] = y[0]
   yp[J] = y[J]
   y = np.copy(yp)
   v = np.copy(vp)
```

  The endpoint values `v[0]` and `v[J]` are never used, so they do not need to be updated.
- Add code as necessary to plot graphs and print (or save) results.

Unfortunately, the FTCS algorithm is unconditionally unstable.

Exercise 22.1

Write a code for the wave equation using the FTCS scheme (22.3). As in the previous chapter, use mass per unit length $\mu = 0.01$ and tension $T = 25.0$ (in SI units), so the wave speed is $c = 50.0$. Set the boundaries at $x_a = -5.0$ and $x_b = 5.0$. For initial conditions, use the Gaussian pulse at rest. That is, y_j^0 is given by Eq. (21.12) and $v_j^0 = 0$. Experiment with different timestep values. Have your code display a graph of $y(x)$ showing the unphysical results.

22.2 Lax-Friedrich method

The FTCS algorithm can be modified to make it stable. Replace y_j^n on the right–hand side of Eq. (22.3a) with the average $(y_{j+1}^n + y_{j-1}^n)/2$. The resulting method,

$$y_j^{n+1} = \frac{1}{2}(y_{j+1}^n + y_{j-1}^n) + \Delta t\, v_j^n \ , \tag{22.4a}$$

$$v_j^{n+1} = v_j^n + \Delta t\, c^2 \left(\frac{y_{j+1}^n - 2y_j^n + y_{j-1}^n}{\Delta x^2} \right) \ , \tag{22.4b}$$

is stable, provided the timestep satisfies a Courant condition. We will call this the Lax–Friedrichs (LF) method.[1]

Exercise 22.2a

Solve the wave equation based on the LF algorithm. Use the same initial data as in the previous exercise. Choose a relatively small timestep so that the simulation is stable. Plot y versus x at various times.

Exercise 22.2b

Experiment with your LF code to obtain an estimate of the Courant condition $\Delta t \leq \eta_{max} \Delta x / c$. (Hint: The answer is $\eta_{max} = 1/\sqrt{2}$.)

22.3 Energy of a vibrating string

In the absence of dissipating effects like air resistance, the energy of a vibrating string is

$$E = \frac{1}{2} \int_{x_a}^{x_b} dx \left[\mu\, v^2 + T(y')^2 \right] \ , \tag{22.5}$$

where $v = \dot{y}$. Recall that μ is the mass per unit length and T is the tension. The two terms in E are the kinetic and elastic potential energies of the string. This might not be entirely obvious, but it is easy to verify that E is conserved. Start by differentiating E with respect to time, and bring the time derivative under the integral over x:

$$\begin{aligned} \frac{dE}{dt} &= \frac{1}{2} \int_{x_a}^{x_b} dx \frac{\partial}{\partial t} \left[\mu\, v^2 + T(y')^2 \right] \\ &= \int_{x_a}^{x_b} dx \left[\mu v \dot{v} + T y' \dot{y}' \right] \ . \end{aligned} \tag{22.6}$$

[1]The term “Lax–Friedrichs method” usually refers to a modification of the FTCS algorithm for the advection equation $\dot{y} = cy'$. Our modification for the wave equation is similar.

Since $\dot{v} = c^2 y''$, this simplifies to

$$\frac{dE}{dt} = \int_{x_a}^{x_b} dx \left[\mu c^2 v y'' + T y' v' \right] \ . \tag{22.7}$$

where $v' = \dot{y}'$. Now, because $c^2 = T/\mu$, we can remove a common factor of T from the integral and write

$$\frac{dE}{dt} = T \int_{x_a}^{x_b} dx \frac{\partial}{\partial x} \left[v y' \right] \ . \tag{22.8}$$

This reduces to a boundary term,

$$\frac{dE}{dt} = T \left[v y' \right] \Big|_{x_a}^{x_b} \ . \tag{22.9}$$

Since y is fixed at the boundaries, the time derivative $v = \dot{y}$ must vanish at x_a and x_b. It follows that $dE/dt = 0$; the energy is constant in time.

Although E is a constant, the energy calculated from a numerical code will not remain constant due to truncation errors. Thus, we can use the energy to help monitor the accuracy of our code. To compute E, we first need to discretize the expression (22.5). The integral over x can be replaced by a simple midpoint rule approximation. The string velocity at the midpoint of the jth subinterval is (to second order accuracy) just the average of the velocities at the adjacent nodes: $(v_j + v_{j-1})/2$. The derivative y' at the midpoint, half–way between nodes $j-1$ and j, is given to second order accuracy by the three–point centered difference formula $(y_j - y_{j-1})/\Delta x$. This yields the discrete expression

$$E = \frac{1}{2} \sum_{j=1}^{J} \Delta x \left[\mu \left(\frac{v_j + v_{j-1}}{2} \right)^2 + T \left(\frac{y_j - y_{j-1}}{\Delta x} \right)^2 \right] \ , \tag{22.10}$$

for the energy of the string.

Exercise 22.3

Compute the energy at the end of each timestep in your LF code. Plot the energy as a function of time.

22.4 Method of lines

The LF method uses the two–point forward difference formula to approximate time derivatives. This difference formula is first order accurate, so it generally limits the LF method to first order accuracy. Assuming the timestep obeys the Courant condition, the errors for the LF method are proportional to the first power of Δt. Since Δt is proportional to Δx, which in turn is proportional to $1/J$, we see that the errors are proportional to $1/J$. We can verify this using a three–point convergence test as discussed in Secs. 17.5 and 19.3.

Let $E_{(J)}$ denote the energy of the string computed numerically at some fixed final time with resolution J. Since the errors are first order,

$$E_{(J)} = E_{(exact)} + K/J \tag{22.11}$$

for some constant K. Here, $E_{(exact)}$ is the (unknown) exact answer. This relation holds for any value of J, which implies

$$\frac{E_{(J)} - E_{(2J)}}{E_{(2J)} - E_{(4J)}} = 2 \ . \tag{22.12}$$

The result is approximate because Eq. (22.11) only holds in the limit of high resolution. Thus, if we run the code at three successive resolutions and compute the left–hand side of Eq. (22.12), the result should approach 2 as J is increased.

Exercise 22.4a

Carry out the three–point convergence test with $x_a = -5$, $x_b = 5$, and $c = 50$. Set the timestep to $\Delta t = 0.25\Delta x/c$ and use the time interval $0 \le t \le 0.25$. For initial conditions, use the Gaussian pulse at rest. Determine $E_{(J)}$ for resolution $J = 100$, 200, 400, etc. Does the left–hand side of Eq. (22.12) approach 2 in the limit of high resolution?

The LF method is not very accurate unless the resolution is very high. There are better methods for solving PDEs. The challenge is to find a higher–order algorithm that is numerically stable.

One general approach is based on the *method of lines.* Here's the idea. Start by discretizing the continuum differential equations (22.1) in space. Define $y_j(t) = y(t, x_j)$ and $v_j(t) = v(t, x_j)$, where, as usual, x_j are the spatial nodes (with $j = 0, \ldots, J$). Using the second order finite difference approximation for y'', the PDEs become

$$\dot{y}_j = v_j \ , \tag{22.13a}$$

$$\dot{v}_j = c^2(y_{j+1} - 2y_j + y_{j-1})/\Delta x^2 \ . \tag{22.13b}$$

This is a set of coupled *ordinary* differential equations (ODEs), with independent variable t and dependent variables $y_0, y_1, \ldots, y_J$ and $v_0, v_1, \ldots v_J$. We can solve them using the ODE methods from earlier chapters.

The method of lines approach does not guarantee that the resulting algorithm is stable. If we use Euler's method to solve Eqs. (22.13), the result is equivalent to the FTSC method. As we saw in Sec. 22.1, this is unconditionally unstable. We could apply second–order Runge–Kutta to solve Eqs. (22.13), but that turns out to be unconditionally unstable as well.

Fourth–order Runge Kutta is stable, provided the Courant condition $\Delta t \leq \sqrt{2}\Delta x/c$ is satisfied. We can apply the RK4 algorithm (18.12) to the system (22.13) by modifying an FTCS code. Simply replace the section of code inside the loop over n with the following. First, compute the variables denoted u_a and u_b in Eqs. (18.12):

```
for j in range(1,J):
   ya[j] = y[j] + 0.5*dt*v[j]
   va[j] = v[j] + 0.5*dt*(c**2/dx**2)*(y[j+1]-2.0*y[j]+y[j-1])
ya[0] = y[0]
ya[J] = y[J]
for j in range(1,J):
   yb[j] = y[j] + 0.5*dt*va[j]
   vb[j] = v[j] + 0.5*dt*(c**2/dx**2)*(ya[j+1]-2.0*ya[j]+ya[j-1])
yb[0] = y[0]
yb[J] = y[J]
```

Continue with the analogous calculations for u_c and u_d, then compute the string amplitude and velocity at time t^{n+1}:

```
for j in range(0,J+1):
   yp[j] = (1/3)*(ya[j] + 2*yb[j] + yc[j] + yd[j]/2) - y[j]/2
   vp[j] = (1/3)*(va[j] + 2*vb[j] + vc[j] + vd[j]/2) - v[j]/2
```

Finally, copy the arrays `yp` and `vp` into `y` and `v` in preparation for the next timestep:

```
y = np.copy(yp)
v = np.copy(vp)
```

Exercise 22.4b

Solve the wave equation using the method of lines with fourth–order Runge–Kutta. Plot the energy as a function of time. How do these results compare to the results you obtained with the LF method?

As a time–evolution scheme, RK4 is fourth–order accurate. But the discretization in space using the three–point finite difference stencil is only second–order accurate. Also keep in mind that the discretization (22.10) of the energy has second order errors.

Exercise 22.4c

Use Simpson's rule to compute the energy integral (22.5) when the initial data consist of a Gaussian pulse at rest. Use a sufficiently large number of subintervals to obtain the answer accurate to 8 or more significant figures. Treat this result as the "exact" answer and compute the error in the energy for the RK4 algorithm at various resolutions. Estimate the time average of the error at each resolution. What is the order of convergence?

22.5 Other initial conditions

The initial conditions can be chosen arbitrarily as long as the boundary conditions are met. Since the string is fixed at the endpoints, the initial conditions should obey $v(0, x_a) = 0$ and $v(0, x_b) = 0$.

Exercise 22.5a

Use the method of lines with RK4 to solve the wave equation with initial conditions

$$y(0, x) = 2\sin(2\pi(x - x_a)/(x_b - x_a)) + \sin(5\pi(x - x_a)/(x_b - x_a)) \ ,$$
$$v(0, x) = 20\sin(3\pi(x - x_a)/(x_b - x_a)) \ .$$

Experiment with other initial conditions.

The general solution to the wave equation is a sum of left–moving and right–moving waves, as shown in Eq. (21.6). Consider a right–moving Gaussian pulse

$$y(t, x) = Ae^{-(x-ct)^2/\sigma^2} \ . \tag{22.14}$$

The initial data for this solution is

$$y(0, x) = Ae^{-x^2/\sigma^2} \ , \tag{22.15a}$$
$$v(0, x) = (2Acx/\sigma^2)e^{-x^2/\sigma^2} \ , \tag{22.15b}$$

where $v = \dot{y}$. These conditions do not satisfy $v(0, x_a) = 0 = v(0, x_b)$. However, if σ is not too small, the string velocity will be very close to 0 at the endpoints.

Exercise 22.5b

Solve the wave equation with the initial data (22.15) using the method of lines with RK4. Plot y versus x at various times and describe the results. What happens if you make a mistake in the formula for $v(0, x)$? For example, you omit the factor of 2? Or replace σ^2 with 3σ in the exponent?

22.6 Other boundary conditions

Up to this point we have considered the string amplitude to be fixed at the endpoints x_a and x_b. Conditions of this type, with the dependent variable fixed at the boundaries, are called *Dirichlet boundary conditions*.[2]

An alternative is *Neumann boundary conditions* in which the spatial derivative of y is fixed at the boundaries. We can implement Neumann boundary conditions by tying each end of the string to a massless ring, and letting each ring slide (without friction) on a post. Consider the ring at the right end, x_b, shown in Fig. 22.1. Because the ring is massless, the net force on the ring must vanish. Otherwise the ring would have an infinite acceleration. Now, the post cannot exert a vertical force on the ring because the ring slides without friction. The only other force on the ring is from tension in the string, so this force must be horizontal. In other words, the slope of the string at x_b must vanish.

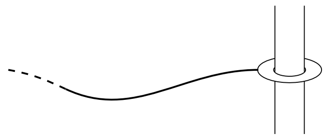

Fig. 22.1: Neumann boundary conditions. The string is attached to a massless ring that slides without friction on a vertical post. The slope of the string vanishes at the point of attachement.

Of course the same reasoning holds at the left end. The slope of the string must vanish at both ends, so that $y'(t, x_a) = y'(t, x_b) = 0$. These are Neumann boundary conditions.

[2]In general, with Dirichlet boundary conditions, the endpoint values $y(t, x_a)$ and $y(t, x_b)$ may be time dependent.

You can switch from Dirichlet to Neumann boundary conditions in your RK4 code by replacing `ya[0] = y[0]` with `ya[0] = ya[1]` and replacing `ya[J] = y[J]` with `ya[J] = ya[J-1]` at the end of the first sub–step. Make analogous changes at the end of the second, third and fourth sub–steps.

Exercise 22.6

Create a version of your RK4 code with Neumann boundary conditions. Experiment with different initial conditions. What happens when a pulse hits the boundary? How does this differ from Dirichlet boundary conditions?

22.7 Other PDEs

We can easily modify these techniques to solve other systems of PDEs. Unfortunately, there is no guarantee that the algorithm will be stable.

Consider the damped wave equation $\ddot{y} + k\dot{y} = c^2 y''$ with damping constant k. This PDE can be written as the first–order system

$$\dot{y} = v \ , \tag{22.16a}$$

$$\dot{v} = c^2 y'' - kv \ . \tag{22.16b}$$

Exercise 22.7a

Numerically solve Eqs. (22.16) using the method of lines with RK4. Use Dirichlet or Neumann boundary conditions and any initial conditions. Plot a graph of y versus x at various times using $c = 50$ and $k = 20$ (SI units).

The *heat equation* describes the temperature T in a body as a function of time and spatial location. Consider a cyindrical rod that extends along the x–axis from x_a to x_b. The sides of the rod are insulated. We will assume that the temperature $T(t, x)$ depends only on t and x—that is, the temperature is constant along each cross section of the rod. The heat equation for the rod is

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2} \ , \tag{22.17}$$

where α is the *thermal diffusivity* of the material.

We can impose various boundary conditions at the ends of the rod. For example, we can attach a large body at temperature T_a to the left end. This imposes the Dirichlet boundary condition $T(t, x_a) = T_a$. We can insulate the right end. This imposes the Neumann boundary condition $\partial T(t, x_b)/\partial x = 0$.

The heat equation can be solved numerically using FTCS (forward differencing in time, centered differencing in space). For stabillity, the Courant condition $\Delta t \le \Delta x^2/(2\alpha)$ must be satisfied.

Exercise 22.7b

Use the FTCS method to solve the heat equation for an aluminum rod of length $x_b - x_a = 1.0\,\mathrm{m}$ and thermal diffusivity $\alpha = 9.7 \times 10^{-5}\mathrm{m}^2/\mathrm{s}$. Impose the Dirichlet boundary condition at the left end with $T_a = 350\,\mathrm{K}$, and impose the Neumann boundary condition at the right end. For initial conditions, choose

$$T(0, x) = 270 + 80e^{-100(x-x_a)^2} .$$

Plot $T(t, x)$ versus x for various times t. How long does it take for the temperature at the right end of the rod to reach 280 K? 300 K? 320 K?

22.8 Animation

Animation of a time-series simulation is fun, and can lead to helpful insights. To animate your code:

- Remove old plot commands.
- Import libraries:

```
# %matplotlib ipympl      # uncomment if needed
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as anim
```

- Define parameters, constants, arrays, and initial data as before. Then add:

```
plt.close()
fig, ax = plt.subplots()
curve = ax.plot(x,y)[0]
```

In general, the function `ax.plot()` returns a list of curves to plot. In this case the list has just one element. The command `ax.plot(x,y)[0]` extracts this element (with index 0) from the list.

- At the core of your previous codes you have a loop that steps through time. For example,

```
for n in range(1,N+1):
    compute y and v at time $t^{n+1}$ from y and v at time $t^n$
```

  Replace this loop with the following function definition:

```
def update(num):
    global y
    global v
    compute y and v at time $t^{n+1}$ from y and v at time $t^n$
    curve.set_data(x,y)
    return
```

- Add this to the end of your code:

```
myanim = animation.FuncAnimation(fig, update, N, interval=50,
      repeat=False)
# plt.show()      # uncomment if needed
```

  The arguments of `FuncAnimation()` are:

  - `fig` (the name of the figure)
  - `update` (the name of the function that updates the figure)
  - `N` (the number of timesteps)
  - `interval=50` (sets the time interval between each frame to 50 ms)
  - `repeat=False` (if set to `True`, the animation will repeat)

- To save the animation as a gif, add the statement

```
myanim.save('my_movie.gif')
```

  to the end of your code.

At the time of writing, this prescription works well in most integrated development environments. (You might need to uncomment the `plt.show()` statement.) Animation is not quite as robust in Jupyter notebooks, due to the extra layer of interaction between the Python interpreter and the web browser. If animation is not working in a Jupyter notebook, try uncommenting the line `%matplotlib ipympl` at the beginning of the code. This command tells Python to use `ipympl` to translate the Matplotlib graphics commands into javascript commands that can be rendered in a web browser. If your code complains that `ipympl` is not installed, you will need to install it.

Exercise 22.8a

Create a version of your wave equation code with animation. Use the initial conditions from Exercise 22.5a with Dirichlet boundary conditions.

Exercise 22.8b

Animate the waves on a string with initial conditions

$$y(0,x) = \cos(5\pi(x-x_a)/(x_b-x_a)) + \cos(3\pi(x-x_a)/(x_b-x_a)) \ ,$$
$$v(0,x) = 25\cos(2\pi(x-x_a)/(x_b-x_a)) \ ,$$

and Neumann boundary conditions.

22.9 First order form of the wave equation

The wave equation can be reduced to a system of PDEs with first order derivatives in both time and space. Before describing the reduction, let's review what we know about the wave equation in its original form, $\ddot{y} = c^2 y''$, with second order time and space derivatives. To be definite we will consider time–independent Dirichlet boundary conditions. Thus, the amplitude of the string at each endpoint is a prescribed constant:

$$y(t,x_a) = \text{const} \ , \tag{22.18a}$$
$$y(t,x_b) = \text{const} \ . \tag{22.18b}$$

The initial data consist of the initial amplitude and velocity:

$$y(0,x) = \text{function of } x \ , \tag{22.19a}$$
$$\dot{y}(0,x) = \text{function of } x \ . \tag{22.19b}$$

These functions can be chosen freely as long as they satisfy the boundary conditions. That is, the initial data function $y(0,x)$ evaluated at x_a must agree with the constant (22.18a). Likewise $y(0,x)$ evaluated at x_b must agree with the constant (22.18b). Also observe that the time derivatives of Eqs. (22.18) imply $\dot{y}(t,x_a) = 0$ and $\dot{y}(t,x_b) = 0$. The initial data function $\dot{y}(0,x)$ must satisfy these relations as well.

The reduction of the wave equation to first order time and space derivatives begins with the definitions

$$v = \dot{y} \ , \tag{22.20a}$$
$$w = cy' \ . \tag{22.20b}$$

In terms of these new variables, the wave equation becomes

$$\dot{v} = cw' \ . \tag{22.21}$$

Together, these three equations are equivalent to the original wave equation.

Our next task is to express the boundary conditions and initial conditions in terms of the new variables y, v and w. Of course the initial conditions (22.19a) are unchanged, but $\dot{y}$ now has the new name v. Thus, the initial conditions become

$$y(0, x) = \text{function of } x \ , \tag{22.22a}$$

$$v(0, x) = \text{function of } x \ . \tag{22.22b}$$

Since $y(0, x)$ is a chosen function of x, we can compute its spatial derivative $y'(0, x)$. Therefore, by the definition (22.20b), w must satisfy

$$w(0, x) = cy'(0, x) \ . \tag{22.23}$$

Thus, the initial value for w is also determined by the initial data.

The boundary conditions tell us that the endpoints of the string are fixed; that is, Eqs. (22.18) hold. Since the endpoint values are constants in time, we also find

$$v(t, x_a) = 0 \ , \tag{22.24a}$$

$$v(t, x_b) = 0 \ . \tag{22.24b}$$

In turn, the values of v at the endpoints are constants in time (equal to zero), so the wave equation (22.21) implies

$$w'(t, x_a) = 0 \ , \tag{22.25a}$$

$$w'(t, x_b) = 0 \ . \tag{22.25b}$$

These are Neumann boundary conditions for the variable w.

Observe that Eq. (22.20b) doesn't contain any time derivatives. How do we handle this in a numerical code? Since this equation must hold for all time, we can differentiate with respect to t to obtain $\dot{w} = c\dot{y}'$. Ordinarily, we would not be allowed to replace $w = cy'$ with $\dot{w} = c\dot{y}'$, because these equations are not equivalent. In fact, $\dot{w} = c\dot{y}'$ implies $w = cy' + f(x)$, where $f(x)$ is some function of x but is constant in t. However, the initial condition (22.23) tells us that $f(x)$ vanishes. We can replace $w = cy'$ with $\dot{w} = c\dot{y}'$, provided we also impose the initial condition (22.23). Finally note that, since $\dot{y} = v$, we can write this new equation as $\dot{w} = cv'$.

To summarize, the wave equation can be written as a system of PDE's with first order time and space derivatives as

$$\dot{y} = v \ , \tag{22.26a}$$

$$\dot{v} = cw' \ , \tag{22.26b}$$

$$\dot{w} = cv' \ . \tag{22.26c}$$

The initial conditions are listed in Eqs. (22.22) and (22.23). The boundary conditions are listed in Eqs. (22.18), (22.24) and (22.25).

Exercise 22.9a

Solve the first order system (22.26) using the FTCS method:

$$\begin{aligned} y_j^{n+1} &= y_j^n + \Delta t v_j^n \ , \\ v_j^{n+1} &= v_j^n + c\Delta t (w_{j+1}^n - w_{j-1}^n)/(2\Delta x) \ , \\ w_j^{n+1} &= w_j^n + c\Delta t (v_{j+1}^n - v_{j-1}^n)/(2\Delta x) \ . \end{aligned}$$

This method is unconditionally unstable, but you should get reasonable results for short run times.

Exercise 22.9b

Apply the Lax–Friedrich modification to your previous code: Replace v_j^n and w_j^n with $(v_{j+1}^n + v_{j-1}^n)/2$ and $(w_{j+1}^n + w_{j-1}^n)/2$ in the second and third equations. (You do not need to modify the first equation. Can you guess why?) This method is stable as long as the Courant condition $\Delta t \leq \Delta x/c$ is satisfied. Graph y versus x at various times t, and compare results to your RK4 code.