CONTENTS

GRIDS AND NUMERICAL DERIVA-TIVES

INTRODUCTION TO PYTHON

In this course we will use Python to study numerical techniques for solving some partial differential equations that arise in Physics. Don't be scared of this new language. Most of the ideas, and some of the syntax, that you learned for Matlab will transfer directly to Python. We'll work through some brief tutorials about Python at the beginning of eachlab, focusing on the particular ideas that you'll need tocomplete that lab. Pretty soon you will be Python wizards.

P1.1 Work through Chapter 1 of Introduction to Python. There you will learn the basics of how to write a Python program, howtodeclare anduseentities called NumPyarrays, and also learn some basic plotting techniques. With that Python knowledge underourbelts, let's move on to begin our study of partial differential equations.

SPATIAL GRIDS

When we solved ordinary differential equations in Physics 330 we were usually moving something forward in time, so you may have the impression that differential equations always 'flow'. This is not true. If we solve a spatial differential equation, like the one that gives the shape of a chain draped between two posts, the solution just sits in space; nothing flows. Instead, we choose a small spatial step size (think of each individual link in the chain) and seek to find the correct shape by somehow finding the height of the chain at each link.

In this course we will solve partial differential equations, which usually means that the desired solution is a function of both space x, which just sits, and time t, which flows. When we solve problems like this we will be using spatial grids, to represent the x-part that doesn't flow. The NumPy arrays that you just learned about above are perfect for representing these kinds of spatial grids.

We'll encounter three basic types of spatial grids in this class. Figure 1.1 shows a graphical representation of these three types of spatial grids for the region $0 \leqslant x \leqslant l$. We divide this region into spatial cells (the spaces between vertical lines) and functions are evaluated at N discrete grid points (the dots). In a celledge grid, the grid points are located at the edge of the cell. In a cell-center grid, the points are located in the middle of the cell. Another useful grid is a cell-center grid with ghost points. The ghost points (unfilled dots) are extra grid points on either side of the interval of interest and are useful when we need to consider the derivatives at the edge of a grid.

Cell-Edge Grid

0   L

Cell-Center Grid

0   L

Cell-Center Grid with Ghost Points

0   L

Figure 1: Three common-spatial grids

P1.2

(a) Write a Python program that creates a cell-edge spatial grid in the variable x as follows:

```
import numpy as np

N=100
a=0
b=np.pi
x,h = np.linspace(a,b,N,retstep = True)
```

Plot the function y(x) = sin(x)sinh(x) on this grid. Explain the relationship between the number of cells and the numberofgridpoints in a cell-edge grid.

(b) Explain the relationship between the number of cells and the number of grid points in a cell-center grid. Then write some code using NumPy's arange function to create a cell-centered grid that has exactly 100 cells over the interval $0 \geq x \geq 2$.

Evaluate the function f (x)=cosx on this grid and plot this function. Thenestimate the area under the curve by summingtheproductsof the centered function values fj with the widths of the cells h like this (midpoint integration rule):
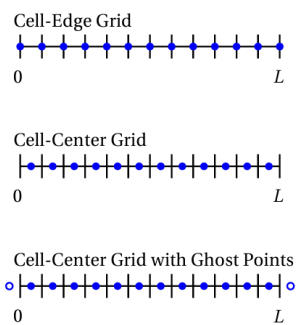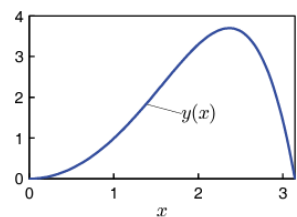
```
np.sum(f)*h
```


Figure 2: Plot from 1.2(a)

Comparethis result to the exact answer obtained by integration:
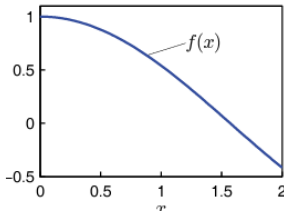
$$A = \int_0^2 cosx\,dx = sin(x)|_0^2 = sin(2)$$

(c) Build a cell-center grid with ghost points over the interval $0 \geq x \geq \pi/2$ with 500 cells (502 grid points), and evaluate the function $f(x) = sinx$ on this grid. Now look carefully at the function values at the first two grid points and at the last two grid points. The function sinx has the property that $f(0) = 0$ and $f\prime(\pi/2) = 0$. The cell-center grid doesn't have points at the ends of the interval, so these boundary conditions on the function need to be enforced using more than one point. Explain how the ghost points can be usedinconnectionwith interior points to specify both function-value boundary conditions andderivative-value boundary conditions.

Figure 3: Plot from 1.2(b)

INTERPOLATION AND EXTRAPOLATION

Grids only represent functions at discrete points, and there will be times when we want to find good values of a function between grid points (interpolation) or beyond the last grid point (extrapolation). We will use interpolation and extrapolation techniques fairly often during this course, so let's review these ideas. Thesimplest way to estimate function values is to use the fact that two points defineastraight line. For example, suppose that we have function values (x_1, y_1) and (x_2, y_2). The formula for a straight line that passes through these two points is

$$y - y_1 = \frac{(y_2 - y_1)}{(x_2 - x_1)}(x - x_1) \tag{1}$$

Once this line has been established it provides an approximation to the true function $y(x)$ that is pretty good in the neighborhood of the two data points. To linearly interpolate or extrapolate we simply evaluate Eq. (1) at x values between or beyond x_1 and x_2.

P1.3 UseEq. (1) to do the following special cases:

(a) Find an approximate value for $y(x)$ halfway between the two points x_1 and x_2. Does your answer make sense?

(b) Find an approximate value for $y(x)$ 3/4 of the way from x_1 to x_2. Do you see a pattern?

Figure 4: The line defined bytwo points can be usedtointerpolate betweenthepoints andextrapolate beyond the points.

(c) If the spacing between grid points is h (i.e. $x_2x_1 = h$), show that the linear extrapolation formula for $y(x2+h)$ is

$$y(x_2 + h) = 2y_2 - y_1 \tag{2}$$

This provides a convenient way to estimate the function value one grid step beyond the last grid point. Also show that

$$y(x_2 + h/2) = 3y_2/2 - y_1/2 \tag{3}$$

We will use both of these formulas during the course.

DERIVATIVES ON GRIDS

Whensolving partial differential equations, we will frequently need to calculate derivatives on our grids. In your introductory calculus book, the derivative was probably introduced using the forward difference formula

$$f/prime(x) \approx \frac{f(x+h) - f(x)}{h} \tag{4}$$

The word $''$ forward $''$ refers to the way this formula reaches forward from x to $x+h$ to calculate the slope. The exact derivative represented by Eq.(4) in the limit that h approaches zero. However, wecan't make h arbitrarily small when we represent a function on agrid because (i) the number of cells needed to represent a region of space becomes infinite ash goes to zero; and (ii) computers represent numbers with a finite number of significant
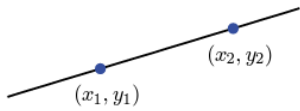
digits so the subtraction in the numerator of Eq.(4) loses accuracy when the two function values are very close. But given these limitation we want to be as accurate as possible, so we want to use the best derivative formulas available. The forward difference formula isnâĂŹt one of them.The best first derivative formula that uses only two function values is usually the centered difference formula:

$$f\prime(x) \approx \frac{f(x+h) - f(x-h)}{2h} \qquad (5)$$

It is called "centered" because the point x at which we want the slope is centered between the places where the function is evaluated. Take a minute to study Fig. 1.5 to understand visually why the centered difference formula is so much better than the forward difference formula. The corresponding centered second derivative formula is

$$f\prime\prime(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \qquad (6)$$

We will derive both of these formulas later, but for now we just want you to understand how to use them. The colon operator provides a compact way to evaluate Eqs. (5) and (6) on a grid. Unfortunately for those of you familiar with Matlab, Python′ s colon operator acts a little differently from Matlab′ s in that the last index is not included in the range, as we noted in the tutorial. If the function we want to take the derivative of is stored in an array f, we can calculate the centered first derivative like this (remember that Python array indexes are zero-based):

```
fp = np.zeros_like(f)
fp[1:N-1]=(f[2:N]-f[0:N-2])/(2*h)
```

and the centered second derivative at each interior grid point like this:

```
fpp = np.zeros_like(f)
fpp[1:N-1]=(f[2:N]-2*f[1:N-1]+f[0:N-2])/h**2
```

The variable h is the spacing between grid points and N is the number of grid points. Study this code (focus on the indexing) until you are convinced that it represents Eqs. (5) and (6) correctly.

The derivative at the first and last points on the grid can't be calculated with Eqs. (5) and (6) since there are not grid points on both sides of the endpoints. Instead, we extrapolate the interior values of the two derivatives to the end points. If we use linear extrapolation then we just need two nearby points, and the formulas for the derivatives at the end points are found using Eq. (2)

```
fp[0]=2*fp[1]-fp[2]
fp[N-1]=2*fp[N-2]-fp[N-3]
fpp[0]=2*fpp[1]-fpp[2]
fpp[N-1]=2*fpp[N-2]-fpp[N-3]
```

P1.4 Create a cell-edge grid with N = 20 on the interval $0 \le x \le 5$. Load $f(x)$ with the Bessel function $J_0(x)$ and numerically differentiate it to obtain $f'(x)$ and $f'\prime(x)$. Then make overlaid plots of the numerical derivatives with the exact derivatives:

$$f'(x) = -J_1(x)$$

$$f'' = \frac{1}{2}(-J_0(x) + J_2(x))$$

ERRORS IN THE APPROXIMATE DERIVATIVE FORMULAS

We'll conclude this lab with a look at where the approximate derivative formulas come from and at the types of the errors that pop up when using them. The starting point is Taylor'expansion of the function f a small distance h away from the point x

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + ... + f^{(n)}(x)\frac{h^n}{n!} + ... \qquad (7)$$

Let's use this series to understand the forward difference approximation to $f'(x)$. If we apply the Taylor expansion to the $f(x+h)$ term in Eq. (4) , we get

$$\frac{f(x+h) - f(x)}{h} = \frac{[f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + ...] - f(x)}{h} \qquad (8)$$
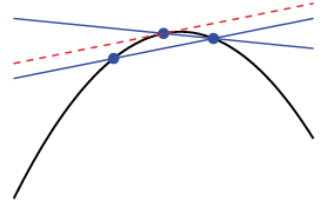

Figure 5: The forward and-centered difference formulas both approximate the derivative as the slope of a line connecting two points. The centered difference formula gives a moreaccurate approximation because it uses points before and after the point where the derivative is being estimated. (The true derivative is the slope of the dotted tangent line).
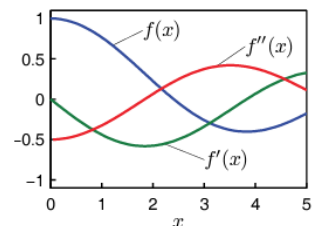

Figure 6: Plots from 1.4

The higher order terms in the expansion (represented by the dots) are smaller than the f'' term because they are all multiplied by higher powers of h (which we assumetobesmall). If we neglect these higher order terms, we can solve Eq. (8) for the exact derivative $f'(x)$ to find

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} - \frac{h}{2} f''(x) \tag{9}$$

FromEq. (9) we see that the forward difference does indeed give the first derivative back, but it carries an error term which is proportional to h. But if h is small enough then the contribution from the term containing $f''(x)$ will be too small to matter and we will have a good approximation to $f'(x)$. For the centered difference formula, we use Taylor expansions for both $f(x+h)$ and $f(xh)$in Eq. (5) to write

$$\frac{f(x+h) - f(x-h)}{2h} = \frac{[f(x) + f'(x)h + f''(x)\frac{h^2}{2} + f'''(x)\frac{h3}{6} + ...]}{2h} - \frac{[f(x) - f'(x)h + f''(x)\frac{h^2}{2} - f'''(x)\frac{h3}{6} +}{2h} \tag{10}$$

If we again neglect the higher-order terms, we can solve Eq. (10) for the exact derivative $f'(x)$. This time, we find that the f'' terms exactly cancel to give

$$f' \approx \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6} f'''(x) \tag{11}$$

Notice that for the centered formula the error term is much smaller, only of order h^2. So if we decrease h in both the forward and centered difference formulas by a factor of 10, the forward difference error will decrease by a factor of 10, but the centered difference error will decrease by a factor of 100. This is the reason we try to use centered formulas whenever possible in this course.

P1.5 Write a Python program to compute the forward and centered difference formulas for the first derivative of the function $f(x) = e^x$ at $x = 0$ with $h = 0.1, 0.01, 0.001$. Verify that the error estimates in Eqs. (9) and (11) agree with the numerical testing. Note that at $x = 0$ the exact values of both f' and f'' are equal to $e^0 = 1$,so just subtract 1 from your numerical result to find the error.

In problem 1.5, you found that $h = 0.001$ in the centered-difference formula gives a better approximation than $h = 0.01$. This trend might entice you to try to keep making h smaller and smaller to achieve any accuracy you want. This doesn't work. Figure 1.7 shows a plot of the error you calculated in problem 1.5 as h continues to decrease (note the log scales). For the larger values of h, the errors track well with the predictions made by the Taylor's series analysis. However, when h becomes too small, the error starts to increase. By about $h = 1016$, and sooner for the second derivative, the error is the same order as the derivative. Thereason for this behavior is that numbers in computers are represented with a finite number of significant digits. Most computational languages (including Python) use double precision variables, which have 15-digit accuracy.[1] This is normally plenty of precision, but look what happens in a subtraction problem where the two numbers are nearly the same:

$$\begin{array}{r} 7.3\ 8\ 9\ 0\ 5\ 6\ 9\ 9\ 6\ 6\ 9\ 5\ 5\ 6 \\ -\ 7.3\ 8\ 9\ 0\ 5\ 6\ 9\ 9\ 1\ 9\ 1\ 7\ 4\ 5 \\ \hline 0.0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 4\ 7\ 7\ 8\ 1\ 1 \end{array} \tag{12}$$

Notice that our nice 15-digit accuracy has disappeared, leaving behind only 6 significant figures. This problem occurs in calculations with floating-point numbers on all digital computers, and is called roundoff. You can see this effect by experimenting with the Python console:

```
h=1e-17
g=1+h
print(g-1)
```

for different values of h and noting that you don't always get h back. Also notice in Fig. 1.7 that this problem is worse for the second derivative formula than it is for the first derivative formula. The lesson here is that it is impossible to achieve arbitrarily high accuracy by using arbitrarily tiny values of h.

Figure 7: Error in the forward and centered difference approximations to the first derivative and the centered difference formula for the second derivative as a function of h. Thefunction is e^x and the approximations are evaluated for $x = 0$.

---

[1]A computer uses 64 bits to represent a double precision number. 53 bits are used to represent the significant digits. Thus the significant digit part can be any integer between 0 and 253 = 9007199254740992 (almost 16 digits). The exponent is represented by 11 bits. After you add the possibility of NaN and Inf, the exponent can be âĹŠ308 to +308. This leaves 1 bit for the overall sign of the number. Extended precision uses more bits (memory) and computation time, so double precision is mostly the standard for computational physics.
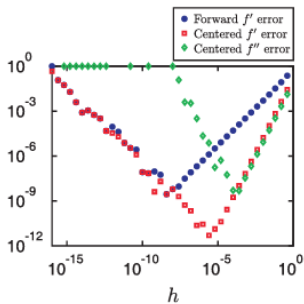
DIFFERENTIAL EQUATIONS WITH BOUNDARY CONDITIONS

MORE PYTHON

P2.1 Work through Chapter 2 of Introduction to Python, where you will learn about lists, loops, and logic.

INITIAL CONDITIONS VS. BOUNDARY CONDITIONS

In Physics 330, we studied the behavior of systems where the initial conditions were specified and we calculated how the system evolved forward in time (e.g. the f light of a baseball given its initial position and velocity). In these cases we were able to use Matlab's convenient built-in differential equation solvers to model the system. The situation becomes somewhat more complicated if instead of having initial conditions, a differential equation has boundary conditions specified at both ends of the interval (rather than just at the beginning). This seemingly simple change in the boundary conditions makes it hard to use canned ODE solvers like Matlab's ode45. Fortunately, there are better ways to solve these systems.

SOLVING DIFFERENTIAL EQUATIONS WITH LINEAR ALGEBRA

Consider the differential equation

$$y''(x) + 9y(x) = sin(x); y(0) = 0, y(2) = 1 \tag{13}$$

Notice that this differential equation has boundary conditions at both ends of the interval instead of having initial conditions at $x = 0$. If we represent this equation on a grid, we can turn this differential equation into a set of algebraic equations that we can solve using linear algebra techniques. Before we see how this works, let's first specify the notation that we'll use. We assume that we have set up a cell-edge spatial grid with N grid points, and we refer to the x values at the grid points using the notation x_n, with $n = 0..N1$. We represent the(as yet unknown) function values $y(xn)$ on our grid using the notation $yn = y(xn)$. Now we can write the differential equation in finite difference form as it would appear on the grid. The second derivative in Eq. (13) is rewritten using the centered difference formula (see Eq. (5)), so that the finite difference version of Eq. (13) becomes:

$$\frac{y_{n+1} - 2y_n + y_{n-1}}{h^2} + 9y_n = sin(x_n) \tag{14}$$

Now let's think about Eq.(14) for a bit.First notice that it is not an equation, but a system of many equations.We have one of these equations at every grid point n, except the end points j=0 and at j=NâĹŠ1 where this formula reaches beyond the ends of the grid and cannot,therefore,be used.Because this equation involves y_{n1}, y_n,and y_{n+1} for the interior grid points n=1...NâĹŠ2, Eq.(2.2) is really a system of NâĹŠ2 coupled equations in the N unknowns y_0 ...y_{N1}.If we had just two more equations we could find the y_n's by solving a linear system of equations. But we do have two more equations; they are the boundary conditions:

$$y_0 = 0; y_{N-1} = 1 \tag{15}$$

which completes our system of N equations in N unknowns.
Before Python can solve this system we have to put it in a matrix equation of the form

$$A_y = b, \tag{16}$$

where A is a matrix of coefficients, y the column vector of unknown y-values, and b the column vector of known values on the right-hand side of Eq.(14).For the particular case of the system represented by Eqs.(14) and (15) ,the matrix equation is given by
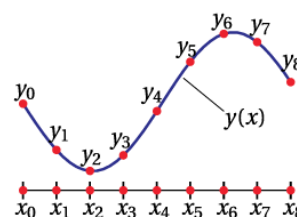

Figure 8: A function $y(x)$ represented onacell-edge x-grid with $N = 9$.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ \frac{1}{h^2} & -\frac{2}{h^2}+9 & \frac{1}{h^2} & 0 & \dots & 0 & 0 & 0 \\ 0 & \frac{1}{h^2} & -\frac{2}{h^2}+9 & \frac{1}{h^2} & \dots & 0 & 0 & 0 \\ . & . & . & . & \dots & . & . & . \\ . & . & . & . & \dots & . & . & . \\ . & . & . & . & \dots & . & . & . \\ 0 & 0 & 0 & 0 & \dots & \frac{1}{h^2} & -\frac{2}{h^2}+9 & \frac{1}{h^2} \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ . \\ . \\ . \\ y_{N-2} \\ y_{N-1} \end{bmatrix} = \begin{bmatrix} 0 \\ sin(x_1) \\ sin(x_2) \\ . \\ . \\ . \\ sin(x_{N-2}) \\ 1 \end{bmatrix} \quad (17)$$

Convince yourself that Eq.(17) is equivalent to Eqs. (14) and (15) by mentally doing each row of the matrix multiply by tipping one row of the matrix up on end,dotting it into the column of unknowny-values, and setting it equal to the corresponding element in the column vector on the right. Once we have the finite-difference approximation to the differential equation in this matrix form $(Ay = b)$, a simple linear solve is all that is required to find the solution array y_n. NumPy can do this solve with the command linalg.solve(a,b).

P2.2

(a) Setup a cell-edge grid with $N = 30$ grid points, like this:

```
import numpy as np
N=30        #thenumberofgridpoints
a=0
b=2
x,h=np.linspace(a,b,N,retstep=True)
```

Look over this code and make sure you understand what it does before just using it.

(b) If you solve Eq. (13) analytically, you find

$$y(x) = \frac{16sin(3x)+cos(6-x)-cos(6+x)+cos(2+3x)-cos(2-3x)}{16sin(6)} \quad (18)$$

Type this solution formula into the Python program that defines the grid above andplot the exact solution as a blue curve on a cell-edge grid with N points.
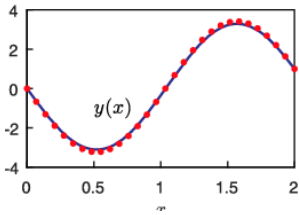
Figure 9: The solution to 2.2(c) with N = 30

(c) Now create a matrix A filled with zeros using np.zeros, and write a for loop to load A like the matrix in Eq. (17) and do the linear solve to obtain y_n and plot it on top of the exact solution with red dots ('r.') to see how closely the two agree. Experiment with larger values of N and plot the difference between the exact and approximate solutions to see how the error changes with N. We think you'll be impressed at how well the numerical method works, if you use enough grid points.

Let's pause a moment to review how to apply this technique to solve a problem. First, write out the differential equation as a set of finite difference equations on a grid, as we did in Eq. (14). Then translate this set of finite difference equations (plus the boundary conditions) into a matrix form analogous to Eq. (17). Finally, build the matrix A and the column vector y in Python and solve for the vectory. Our example, Eq. (13), had only a second derivative, but first derivatives can be handled using the centered first derivative approximation, Eq. (5). Let's practice this procedure for a couple more differential equations:

P2.3

Figure 10: Solution to 2.3(a) with N =30 (dots) compared to the exact solution (line)

(a) Write out the finite difference equations on paper for the differential equation

$$y''+\frac{1}{x}y'+(1-\frac{1}{x^2})y=x; y(0)=0, y(5)=1 \quad (19)$$

Then write down the matrix A and the vector b for this equation. Finally, build these matrices in a Python program and solve the equation using the matrix method. Compare the solution found using the matrix method with the exact solution

$$y(x) = \frac{-4}{J_1(5)}J_1(x)+x$$

$J_1(x)$ is the first order Bessel function.

HINT: When creating your b vector, you'll be tempted to write some code like this: $b = x$. Don't. This code just assigns b to be a reference to the matrix x. What you want is to create a copy of the matrix like this:

```
b = np.copy(x)
```

(b) Solve the differential equation

$$y'' + sin(x)y' + e^x y = x^2; y(0) = 0, y(5) = 3 \qquad (20)$$

in Python using the matrix method. Mathematica's NDSolve command claims that, to 15-digit accuracy

$$y(4.5) = 8.720623277763513$$

Figure 11: Solution to 2.3(b) with N = 200

You'd run out of memory with the matrix method long before achieving this level of accuracy, but how many points do you have to use in your numerical method to get agreement with Mathematica to 3 digits of accuracy (i.e. $y(4.5) = 8.72$)?

Hint: If your step size variable is h, the index for the for element where $x_j \approx 4.5$ can be found this way:

```
j = int(4.5/h)
```

The int command rounds the result and casts it as an integer so you can use it to access a specific element.

DERIVATIVE BOUNDARY CONDITIONS

Now let's see how to modify the linear algebra approach to differential equations so that we can handle boundary conditions where derivatives are specified instead of values. Consider the differential equation

$$y''(x) + 9y(x) = x; y(0) = 0, Y'(2) = 0 \qquad (21)$$

We can satisfy the boundary condition $y(0) = 0$ as before (just use $y_1 = 0$), but what do we do with the derivative condition at the other boundary?

P2.4

(a) A crude way to implement the derivative boundary condition is to use a forward difference formula for the last two points:

$$\frac{y_{N-1} - y_{N-2}}{h} = y'|_{x=2}$$

In the present case, where $y'(2) = 0$, this simply means that we need to write the last row of our matrix to enforce

$$y_{N-1} - y_{N-2} = 0$$

Figure 12: The solution to 2.4 (a) with $N = 30$. The RMS difference from the exact solution is 8.8×10^{-4}

Think about what the new boundary conditions will do to the final row of matrix A and the final element of vector b, and then solve Eq. (21) using the matrix method with this boundary condition. Compare the resulting numerical solution to the analytic solution:

$$y(x) = \frac{x}{9} - \frac{sin(3x)}{27 cos(6)}$$

(b) You can improve the boundary condition formula using quadratic extrapolation. In this method, you fit a parabola of the form

$$y(x) = a + bx + cx^2 \qquad (22)$$

to the last three points on your grid to find a, b, and c in terms of your data points. Then you take the derivative of Eq. (21) and evaluate it at the edge $x = x_{N1}$. Normally we'd make you go through the math to derive this, but since time is short,

we'll just tell you that this process gives you the following finite difference approximation for the $y'(x)$ at the end of the grid:
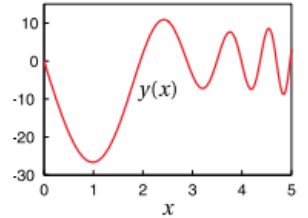
$$\frac{1}{2h}y_{N-3} - \frac{2}{h}y_{N-2} + \frac{3}{2h}y_{N-1} = y'(X_{N-1}) \tag{23}$$

Modify your program from part (a) to include this new condition and show that it gives a more accurate solution than the crude technique of part (a). When you check the accuracy, don't just look at the end of the interval. All of the points are coupled by the matrix A, so you should use a full-interval accuracy check like the RMS (root-mean-square) error:

```
np.sqrt(np.mean((y-yexact)**2))
```

NONLINEAR DIFFERENTIAL EQUATIONS

Finally, we must confess that we have been giving you easy problems to solve, which probably leaves the impression that you can use this linear algebra trick to solve all second-order differential equations with boundary conditions at the ends. The problems we have given you so far are easy because they are linear differential equations, so they can be translated into linear algebra problems. Linear problems are not the whole story in physics, of course, but most of the problems we will do in this course are linear, so these finite-difference and matrix methods will serve us well in the labs to come.

Figure 14: The solution to 2.5(b).

P2.5

(a) Here is a simple example of a differential equation that isn't linear:

$$y''(x) + \sin[y(x)] = 1; y(0) = 0, y(3) = 0 \tag{24}$$

Work at turning this problem into a linear algebra problem to see why it can't be done, and explain the reasons to the TA.

(b) Let's find a way to use a combination of linear algebra and iteration (initial guess, refinement, etc.) to solve Eq. (24) in Python on a grid. First, write the equation as

$$y''(x) = 1 - \sin[y(x)] \tag{25}$$

Make a guess for $y(x)$. It doesn't have to be a very good guess. In this case, the guess $y(x) = 0$ works just fine. Then treat the whole right side of Eq.(25) as known so it goes in the b vector. Then you can solve the equation to find an improved guess for $y(x)$. Use this better guess to rebuild b (again treating the right side of Eq. (25) as known), and then re-solve to get and even better guess. Keep iterating until your $y(x)$ converges to the desired level of accuracy. This happens when your $y(x)$ satisfies (25) to a specified criterion, not when the change in $y(x)$ from one iteration to the next falls below a certain level. Iterate until your RMS error is less than 10^{-5}, as described in the hint below. HINT: An appropriate error vector would be

```
err = A@y-(1-np.sin(y))
```

(remember that @ is the matrix multiplication operator). Compare this to Eq. (25) and convince yourself that the entire vector err will be zero when the equation is exactly solved. We'll compute the RMS error at interior points only, like this

```
rmserr = np.sqrt(np.mean(err[1:-2]**2))
```

because the end points don't satisfy the differential equation. Use rmserr as your check condition.

THE WAVE EQUATION: STEADY STATE AND RESONANCE

PYTHON MECHANICS

P3.1 Work through Chapter 3 of Introduction to Python, where you will learn about user-defined functions and good commenting practices. For the remainder of the course, you will need to demonstrate good commenting practices in your code before your exercises will be passed off.

WAVE EQUATION

We are now ready to tackle our first partial differential equation: the wave equation. For a string of length L fixed at both ends with a force applied to it that varies sinusoidally in time, the wave equation can be written as

$$\mu \frac{\partial^2 y}{\partial t^2} = T \frac{\partial^2 y}{\partial x^2} + f(x) \cos \omega t; y(0,t) = 0, y(L,t) = 0 \qquad (26)$$

where $y(x,t)$ is the lateral displacement of the string as a function of position and time, assuming that $y(x,t)$⊥L. 1 We have written Eq. (**??**) in the form of Newton's second law, $F = ma$. The âĂIJmaâĂİ part is on the left of the equation, except that μ is not the mass, but rather the linear mass density (mass per length). This means that the right side should have units of force/length, and it does because T is the tension (force) in the string and $\partial^2 y / \partial x^2$ has units of 1/length. Finally, $f(x)$ is the amplitude of the driving force (in units of force/length) applied to the string as a function of position (so we are not necessarily just wiggling the end of the string) and ÏĽ is the frequency of the driving force.

Before calculating, let's train our intuition to guess how the solutions of this equation behave. If we suddenly started to push and pull on a string under tension with force density $f(x) \cos(\omega t)$, we would launch traveling waves, which would reflect back and forth on the string as the driving force continued to launch more waves. The string motion would rapidly become very messy. But suppose that there is a bit of damping in the system (not included in the equation above, but in Lab 5 we will add it). With damping, all of the transient waves due to the initial launch and subsequent reflections would die away and we would be left with a steady-state oscillation of the string at the driving frequency ω. This behavior is the wave equation analog of damped transients and the steady final state of a driven harmonic oscillator which you studied in Physics 330.

STEADY STATE SOLUTION BY SEPARATION OF VARIABLES

Let's look for this steady-state solution by guessing that the solution has the form

$$y(x,t) = g(x) \cos(\omega t) \qquad (27)$$

This function has the form of a spatially dependent amplitude $g(x)$ which oscillates in time at the frequency of the driving force. Substituting this "guess "into the wave equation and carrying out the derivatives yields (after some rearrangement)

$$Tg''(x) + \mu \omega^2 g(x) = -f(x); g(0) = 0, g(L) = 0 \qquad (28)$$

This is just a two-point boundary value problem of the kind we studied in Lab 2, so we can solve it using our matrix technique.

P3.2

(a) Write a program to solve Eq. (**??**) for $N = 100$ points with $\mu = 0.003, T = 127, L = 1.2$, and $\omega = 400$. All quantities are in SI units. Find the steady-state amplitude associated with the driving force density:

$$f(x) = \begin{cases} 0.73 \ if \ 0.8 \ \geq x \geq 1 \\ 0 \ otherwise \end{cases} \qquad (29)$$
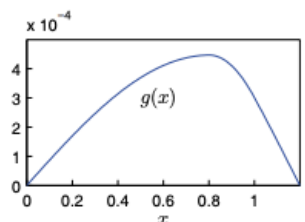

Figure 15: Solution to P3.2(a)
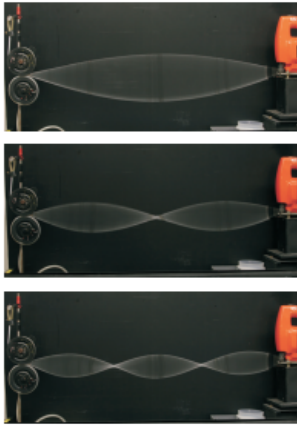
Figure 16: Photographs of the first three resonant modes for a string fixed at both ends.
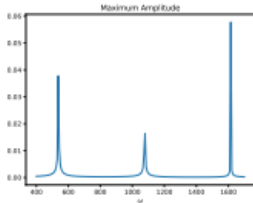
Figure 17: Solution to problem 3.2(c).

which is something like grabbing the string toward one end and wiggling. Plot $g(x)$, and properly label the axes of your plot.

(b) Take your code from (a) and turn it into a library with two functions:

    (a) A function force that accepts the grid x as an argument and returns the column vector representing $f(x)$

    (b) A function *steadySol* that accepts the arguments $f(x)$ (returned from force), h, ω, T, and μ. This function should construct the matrix, solve the matrix equation, and return $g(x)$.

Save this library as âĂIJLab3Funcs.pyâĂİ and then write another program that imports this library. In this program, write a *for* loop that sweeps the value of ω from $\omega = 400$to $\omega = 1700$ in 200 steps keeping the values of the other parameters constant. Plot $g(x)$ at each frequency so that you make an animation of what happens when you sweep the driving frequency. Here is some example code to help you make the animation, assuming you've stored your ÏĹ values in wplot:

```
1      plt.figure(1)
2  for n in range(len(wplot)):
3  w = wplot[n]
4  g = l3.steadySol(f,h,w,T,u)
5  plt.clf() # Clear the previous plot
6  plt.plot(x,g)
7  plt.title('$\omega={:1.2e}$'.format(w))
8  plt.xlabel('x')
9  plt.ylim([-0.05, 0.05]) # prevent auto-scaling
10 plt.draw() # Request to draw the plot
11 plt.pause(0.1) # Allow time to draw it
12
```

At certain frequencies, you should see distinct resonance modes appear as in Fig. 3.2.

(c) Modify your code from part (b) so that you find and record the maximum amplitude of $g(x)$ at each ω. Then plot the maximum amplitude as a function of ÏĹ to see the resonance behavior of this system. Describe what the peaks in this plot mean.

The height of the peaks in this Fig. 3.3 isn't meaningfulâĂŤthe height of the actual peak is essentially infinite (since we donâĂŹt have any damping yet), while the height displayed on the plot is just a reflection of how closely our chosen frequencies happened to line up with the exact resonance frequency. Now we will learn how to find these resonant frequencies directly without needing to solve the differential equation over and over again.

RESONANCE AND THE EIGENVALUE PROBLEM

The essence of resonance is that at certain frequencies a large steady-state amplitude is obtained with a very small driving force. To find these resonant frequencies we seek solutions of Eq. (28) for which the driving force is zero. With $f(x) = 0$, Eq. (28) takes on the form

$$Tg''(x) + \mu \omega^2 g(x) = 0 \; ; \; g(0) = 0, g(L) = 0 \tag{30}$$

If we rewrite this equation in the form

$$g''(x) = -(\frac{\mu \omega^2}{T} g(x) \tag{31}$$

$$Ag = \lambda g \tag{32}$$

where A is a linear operator (the second derivative on the left side of Eq. (**??**)) and λ is the eigenvalue ($\mu \omega^2 / T$). Equation (**??**) is easily solved analytically, and its solutions are just the familiar sine and cosine functions. The boundary condition $g(0) = 0$ tells us to try a sine function form, $g(x) = g_0 \sin(kx)$. If we substitute this form into Eq. (**??**) , we find that it works, provided that k satisfies k = $\omega \sqrt{\frac{\mu}{T}}$. We the have

$$g(x) = g_0 \sin(\omega \sqrt{\frac{\mu}{T}} x) \tag{33}$$

where g_0 is the arbitrary amplitude. When we apply the boundary condition $g(L) = 0$, we find that the resonant frequencies take on discrete values given by

$$\omega = n\frac{\pi}{L}\sqrt{\frac{T}{\mu}} \tag{34}$$

where n is an integer. Each value of n gives a specific resonance frequency from Eq. (**??**) and a corresponding spatial mode $g(x)$ given by Eq. (**??**) For this simple example we were able to do the eigenvalue problem analytically, but when the differential equation is not so simple we will need to do the eigenvalue calculation numerically. Let's see practice the numerical calculations in this case where we know the answer. Rewriting Eq. (**??**) in matrix form using finite differences, as we learned to do in the last lab, yields

$$Ag = \lambda g \tag{35}$$

where $\lambda = -\mu\omega^2/T$. With finite differencing, this becomes the matrix equation

$$
\begin{bmatrix}
? & ? & ? & ? & \cdots & ? & ? & ? \\
\frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & \cdots & 0 & 0 & 0 \\
0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & \cdots & 0 & 0 & 0 \\
\cdot & \cdot & \cdot & \cdot & \cdots & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdots & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdots & \cdot & \cdot & \vdots \\
0 & 0 & 0 & 0 & \cdots & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} \\
? & ? & ? & ? & \cdots & ? & ? & ?
\end{bmatrix}
\begin{bmatrix}
g_0 \\ g_1 \\ g_2 \\ \cdot \\ \cdot \\ \cdot \\ g_{N-2} \\ g_{N-1}
\end{bmatrix}
= \lambda
\begin{bmatrix}
? \\ g_1 \\ g_2 \\ \cdot \\ \cdot \\ \cdot \\ g_{N-2} \\ ?
\end{bmatrix}
\tag{36}
$$

The question marks in the first and last rows remind us that we have to invent something to put in these rows to implement the boundary conditions. The answer we are seeking is the function g, and it appears on both sides of the equation. Thus, the question marks in the g -vector on the right are a real problem because without g_0 and g_{N-1}, we don't have an eigenvalue problem (i.e. g on left side of Eq. (**??**) is not the same as g on the right side).

The simplest way to deal with this question-mark problem and to also handle the boundary conditions is to change the form of Eq. (**??**) to the slightly more complicated form of a generalized eigenvalue problem, like this:

$$Ag = \lambda Bg \tag{37}$$

where B is another matrix, whose elements we will choose to make the boundary conditions come out right. To see how this is done, here is the generalized modification of Eq. (**??**) with B and the top and bottom rows of A chosen to apply the boundary conditions $g(0) = 0$ and $g(L) = 0$:

$$
\begin{bmatrix}
1 & 0 & 0 & \cdots & 0 & 0 \\
\frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & \cdots & 0 & 0 \\
0 & \frac{1}{h^2} & -\frac{2}{h^2} & \cdots & 0 & 0 \\
\cdot & \cdot & \cdot & \cdots & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot & \vdots \\
0 & 0 & 0 & \cdots & -\frac{2}{h^2} & \frac{1}{h^2} \\
0 & 0 & 0 & \cdots & 0 & 1
\end{bmatrix}
\begin{bmatrix}
g_0 \\ g_1 \\ g_2 \\ \cdot \\ \cdot \\ \cdot \\ g_{N-2} \\ g_{N-1}
\end{bmatrix}
= \lambda
\begin{bmatrix}
0 & 0 & 0 & \cdots & 0 & 0 \\
0 & 1 & 0 & \cdots & 0 & 0 \\
0 & 0 & 1 & \cdots & 0 & 0 \\
\cdot & \cdot & \cdot & \cdots & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot & \cdot \\
0 & 0 & 0 & \cdots & 1 & 0 \\
0 & 0 & 0 & \cdots & 0 & 0
\end{bmatrix}
\begin{bmatrix}
g_0 \\ g_1 \\ g_2 \\ \cdot \\ \cdot \\ \cdot \\ g_{N-2} \\ g_{N-1}
\end{bmatrix}
\tag{38}
$$

Note that B is just the identity matrix (made with NumPy's $np.identity(N)$) except that the first and last rows are completely filled with zeros. Take a minute to do the matrix multiplications corresponding the first and last rows and verify that they correctly give $g_0 = 0$ and $g_{N1} = 0$ no matter what the eigenvalue λ turns out to be

To numerically solve the generalized eigenvalue problem you do the following:

1. Load a NumPy array A with the matrix on the left side of Eq. (**??**) and an array B with the matrix on the right side.

2. Use SciPy's generalized eigenvalue and eigenvector command:

```
import scipy.linalg as la
vals,vecs = la.eig(A,B)
```
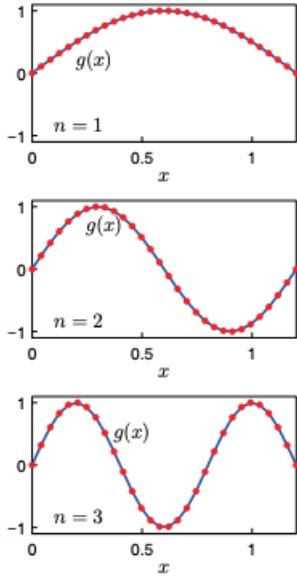
Figure 18: The first three
eigenfunctions found in 3.3.
The points are the numerical
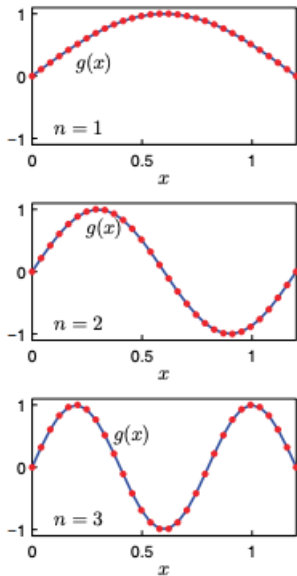eigenfunctions and the line
is the exact solution.

Figure 19: The first three
eigenfunctions for 3.4 (a).

which returns the eigenvalues λ as the entries of the matrix vals and the eigenvectors as the columns of the square matrix vecs. These column in this array are the amplitude functions $g_n = g(x_n)$ associated with each eigenvalue on the grid x_n.

3. Convert eigenvalues λ to frequencies via $\omega^2 = T\lambda/\mu$, sort the squared frequencies in ascending order, like this.

```
import numpy as np
# Compute the eigen-frequencies
w = np.sqrt(-T*np.real(vals)/u)
# Sort the eigenvalues and eigenvectors
ind = np.argsort(w)
w=w[ind]
vecs = vecs[:,ind]
```

The eigenvalues come back as complex numbers, even though the imaginary parts are all zero, so we use the *np.real(vals)* function to switch them back to usual floats. The algorithm np.argsort returns an array of indices showing how the array w should be rearranged to be in ascending order. The next two lines of code rearrange the *w* array and the associated columns in the *vecs* array so that they are sorted in ascending order.

P3.3

(a) Write a program to numerically find the eigenvalues and eigenvectors of Eq. (**??**) using the procedure outlined above. Use $N = 30, \mu = 0.003, T = 127$, and $L = 1.2$. Write a loop that plots each of the eigenvectors. You will find two infinite eigenvalues together with odd-looking eigenvectors that don't satisfy the boundary conditions. These two show up because of the two rows of the B matrix that are filled with zeros. They are numerical artifacts with no physical meaning, so just ignore them. The eigenvectors of the higher modes start looking jagged. These must also be ignored because they are poor approximations to the continuous differential equation in Eq. (**??**).

(b) A few of the smooth eigenfunctions are very good approximations. Plot the eigenfunctions corresponding to $n = 1, 2, 3$ and compare them with the exact solutions in Eq. (**??**). Since the modes include an arbitrary scaling g_0, you won't find the amplitudes match unless you choose an appropriate value for g_0. Calculate the exact values for λ using Eq. (**??**) and compare them with the numerical eigenvalues. Now compare your numerical eigenvalues for the $n = 20$ mode with the exact solution. What is the trend in the accuracy of the eigenvalue method?

(c) The first few values for λ should match the resonances that you found in 3.2(b). Go back to your calculation in 3.2(b) and make two plots of the steady state amplitude for driving frequencies near these two resonant values of λ. For each plot, choose a small range of frequencies that brackets the resonance frequency above and below. You should find very large amplitudes, indicating that you are right on the resonances.

Finally let's explore what happens to the eigenmode shapes when we change the boundary conditions.

P3.4

(a) Change your program from problem 3.3 to implement the boundary condition

$$g'(L) = 0$$

This boundary condition describes what happens if one end of the string is free to slide up and down rather than attached to a fixed position. Use the approximation you derived in problem 2.4(b) for the derivative $g'(L)$ to implement this boundary condition, i.e

$$g'(L) \approx \frac{1}{2h}g_{N-3} - \frac{2}{h}g_{N-2} + \frac{3}{2h}g_{N-1}$$

Explain physically why the resonant frequencies change as they do.

(b) In some problems mixed boundary conditions are encountered, for example

$$g'(L) = 2g(L)$$

Find the first few resonant frequencies and eigenfunctions for this case. Look at your eigenfunctions and verify that the boundary condition is satisfied. Also notice that one of your eigenvalues corresponds to ω^2 being negative. This means that the nice smooth eigenfunction associated with this eigenvalue is not physical in our wave resonance problem with this boundary condition. The code snippet we gave you above will have trouble with the np.sqrt. You can just look at the values of ω^2 instead

THE HANGING CHAIN AND QUANTUM BOUND STATES

RESONANCE FOR A HANGING CHAIN

In the last lab, we studied waves on a string with constant tension and observed sinusoidal normal modes of vibration. We'll start off this lab by studying the problem of standing waves on a hanging chain. It was the famous Swiss mathematician Johann Bernoulli who discovered in the 1700s that a draped hanging chain has the shape of a "catenary", or the hyperbolic cosine function. The problem of the normal mode frequencies of a vertical hanging chain was solved by Johann's son, Daniel Bernoulli, and is the first time that the function that later became known as the J_0 Bessel function showed up in physics. For a hanging chain, the tension varies with positionâĂŤthe tension at the top is large (since it supports the weight of the whole chain) and the tension at the bottom is essentially zero.1 We are going to find its normal modes of vibration of a hanging chain using the method of Problem 3.3. The wave equation for transverse waves on a chain with varying tension $T(x)$ and constant linear mass density μ is given by[2]

$$\mu \frac{\partial^2 y}{\partial t^2} - \frac{\partial}{\partial x}\left(T(x)\frac{\partial y}{\partial x}\right) = 0 \tag{39}$$

We'll use a coordinate system that defines $x = 0$ as the bottom of the chain and $x = L$ as the ceiling.

P4.1 Use the fact that a stationary hanging chain is in equilibrium to draw a free-body diagram for a link at an arbitrary x. Use this diagram to show that the tension in the chain as a function of x is given by

$$T(x) = \mu g x \tag{40}$$

where μ is the linear mass density of the chain and $g = 9.8m/s^2$ is the acceleration of gravity. Then show that Eq. (**??**) reduces to

$$\frac{\partial^2 y}{\partial t^2} - g\frac{\partial}{\partial x}\left(x\frac{\partial y}{\partial x}\right) = 0 \tag{41}$$

for a freely hanging chain. As in Lab 3, we now look for normal modes by separating the variables: $y(x,t) = f(x)\cos(\omega t)$. We then substitute this form for $y(x,t)$ into (**??**) and simplify to obtain

$$x\frac{d^2 f}{dx^2} + \frac{df}{dx} = -\frac{\omega^2}{g}f \tag{42}$$

which is in eigenvalue form with $\lambda = \omega^2/g$. The boundary condition at the ceiling is $f(L) = 0$ while the boundary condition at the bottom is obtained by taking the limit of Eq. (**??**) as $x \to 0$ to find

$$f'(x) = -\frac{\omega^2}{g}f(0) = \lambda f(0) \tag{43}$$

In the past couple labs we dealt with derivative boundary conditions by fitting a parabola to the last three points on the grid and then taking the derivative of the parabola (e.g. Problems 2.4(b) and 3.4). This time we'll handle the derivative boundary condition by using a cell-centered grid with ghost points, as discussed in Lab 1. Recall that a cell-center grid divides the spatial region from 0 to L into N cells with a grid point at the center of each cell. We then add two more grid points outside of $[0,L]$, one at $x_0 = h/2$ and the other at $x_{N+1} = L + h/2$. The ghost points are used to apply the boundary conditions. By defining N as the number of interior grid points (or cells), we have $N+2$ total grid points, which may seem weird to you. We prefer it, however, because it reminds us that we are using a cell-centered grid with N physical grid points and two ghost points.

With this grid there isn't a grid point at each endpoint, but rather we have two grid points that straddle each endpoint. If the boundary condition specifies a value, like $f(L) = 0$ in the problem at hand, we use a simple average like this:
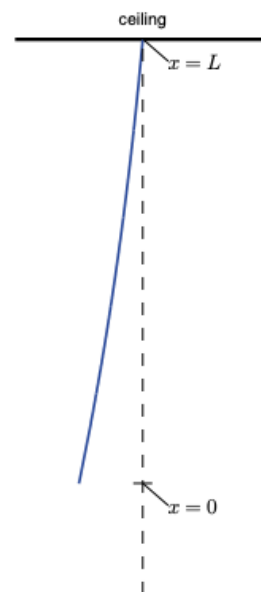
$$\frac{f_{N+1} + f_N}{2} = 0 \tag{44}$$
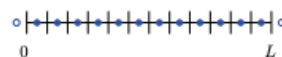
Figure 20: The first normal mode for a hanging chain.

Figure 21: A cell-centered grid with ghost points. (The open circles are the ghost points.)
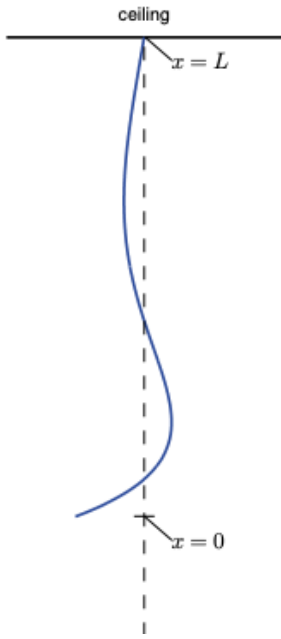
ceiling

$x = L$

$x = 0$

Figure 23: The shape of the third mode of a hanging chain

For $f'(L) = 0$, a centered derivative around $x = L$ yields

$$\frac{f_{N+1} + f_N}{h} = 0 \tag{45}$$

P4.2

(a) On paper, write out the discretized version of Eq. (**??**) and put it in the form of a generalized eigenvalue problem

$$Af = \lambda Bf \tag{46}$$

Remember that for the interior points, the matrix B is just the identity matrix with 1 on the main diagonal and zeros everywhere else. Decide on the values needed in the bottom rows of A and B to give the boundary condition in Eq. (**??**) at $x = L$ (the ceiling) no matter what λ turns out to be.

(b) Now let's decide how to handle the derivative boundary condition at $x = 0$ (the bottom of the chain), given by Eq. (**??**). Since this condition is already in eigenvalue form we don't load the top row of B with zeros. Instead we load A with the left operator $f'(0)$ according to Eq. (**??**) and B with the right operator $f(0)$ according to Eq. (**??**). Leave the eigenvalue $\lambda = \omega^2/g$ out of the top row of B since the matrix equation $Af = \lambda Bf$ already has the $\hat{I}\dot{z}$ multiplying B. Write down the values needed in the top rows of A and B, perform the matrix multiplication for this row, and verify that your choices correctly produce Eq. (**??**).

(c) Write a program to load the matrices A and B with $L = 2m$ (or the measured length if different). Then solve for the normal modes of vibration of a hanging chain. As in Lab 3, some of the eigenvectors are not physical because they donâĂŹt satisfy the boundary conditions; ignore them. Compare your numerical resonance frequencies to measurements made on the chain hanging from the ceiling in the classroom

(d) The analytic solution to Eq. (**??**) without any boundary conditions is

$$f(x) = c_1 J_0(2\omega\sqrt{x/g}) + c_2 Y_0(2\omega\sqrt{2\omega\sqrt{x/g}})$$

where J_0 and Y_0 are the Bessel functions of the first and second kind, respectively. The boundary condition at $x = 0$ rules out Y_0, since it is singular at $x = 0$. Apply the condition $f(L) = 0$ to find analytically the mode frequencies ω_i in terms of the values xi that satisfy $J_0(xi) = 0$. Verify that these values agree with the ÏL̈ values from part (c). HINT: The *scipy.special* library has a function $jn_zeros(n, i)$ that will return the first i values of x for which $J_n(x)$ is zero.

QUANTUM BOUND STATES

Now let's jump forward several centuries in physics history and study bound quantum states using the same techniques we used to study the modes of a hanging chain. SchrÃűdinger's equation for a potential well $V(x)$ is

$$i\bar{h}\frac{\partial\Psi}{\partial t} = -\frac{\bar{h}^2}{2m}\frac{\partial^2\Psi}{\partial x^2} + V(x)\Psi \tag{47}$$

If we assume a separable solution of the form $\Psi(x,t) = \Psi(x)e^{iEt/}$ and plug this into Eq. (**??**), we find the time-independent SchrÃűdinger equation

$$-\frac{\bar{h}^2}{2m}\frac{d^2\psi}{dx^2} + V(x)\psi = E\psi \tag{48}$$

For a particle in a one-dimensional harmonic oscillator, with $V(x) = kx^2/2$, this becomes

$$-\frac{\bar{h}^2}{2m}\frac{d^2\psi}{dx^2} + \frac{1}{2}kx^2\psi = E\psi \tag{49}$$

with boundary conditions $\psi = 0$ at $\hat{A}\acute{s}\hat{a}\acute{L}đ$. Note that k is not the wave number; it is the spring constant, $F = kx$, with units of Newtons/meter. The numbers that go into SchrÃűdingerâĂŹs equation are so small that it makes it difficult to tell what size of grid to use. For instance, using lengths like 2, 5, or 10 meters would be completely ridiculous

for the bound states of an atom where the typical size is on the order of 10^{10} m. Some physicists just set ,m, and k to unity, but this is bad practice. If you do this, you won't know what physical parameters your numerical results describe. The correct procedure is to "rescale "the problem.

The goal of rescaling is to replace physical variables like x with unitless variables like $\xi = x/a$, where a is a "characteristic "length written in terms of the other variables in the problem. Since a is a length, ξ is unitless, and since a is scaled to the problem parameters, ξ will typically have magnitudes around the size of 1. Let's practice this procedure for the problem at hand.

P4.3 Substitute $x = \alpha\xi$ into Eq. (**??**), and then do some algebra to put the equation in the form

$$-\frac{C}{2}\frac{d^2\psi}{d\xi^2} + \frac{1}{2}\xi^2\psi = \frac{E}{\bar{E}}\psi \tag{50}$$

where the constants C and $\bar{E}$ involve factors like ,m,k, and a. Now make the differential operator on the left be as simple as possible by choosing to make $C = 1$. This determines how the characteristic length a depends on ,m, and k. Once you have determined a in this way, check to see that it has units of length. You should find

$$a = \left(\frac{\bar{h}^2}{km}\right)^{1/4} = \sqrt{\frac{\bar{h}}{m\omega}}, \quad \text{where} \quad \omega = \sqrt{\frac{k}{m}} \tag{51}$$

Finally, rescale the energy by writing introducing a new variable $\varepsilon = E/\bar{E}$. Show that $\bar{E}$ has units of energy, so that İţ is unitless. You should find that

$$\bar{E} = \bar{h}\sqrt{\frac{k}{m}} \tag{52}$$

The final scaled version of SchrÃűdinger's equation then becomes

$$-\frac{1}{2}\frac{d^2\psi}{d\xi^2} + \frac{1}{2}\xi^2\psi = \varepsilon\psi \tag{53}$$

When you solve this equation and find results in terms of ε and ξ, you can use the equations above to translate them into real-world values of E and x.

Now that SchrÃűdinger's equation is in dimensionless form, we are ready to solve it with out standard technique.

P4.4 Discretize Eq. (53) on paper. Then write a program to do this eigenvalue problem, similar to the other programs we've written recently. The boundary conditions are that both the value and derivative of ψ should go to zero at $\xi = \infty+$. Since you've scaled the problem, it makes sense to choose a cell-edge grid that goes from $\xi = -5$ to $\xi = 5$, or some other similar pair of numbers. These numbers are supposed to approximate infinity in this problem. Just set the value of ψ to zero at the edge-points and make sure (by looking at the eigenfunction solutions) that your grid is large enough that the wave function has zero slope at the edges of the grid.

(a) Plot the first several bound states. As a guide, Figure 4.5 displays the square of the wave function for the first few excited states. (The amplitude has been appropriately normalized so that $\int |(x)|^2 = 1$

(b) If you look in a quantum mechanics textbook you will find that the bound state energies for the simple harmonic oscillator are given by the formula

$$E_n = (n+\frac{1}{2})\bar{h}\sqrt{\frac{k}{m}} = (b+\frac{1}{2})\bar{E} \tag{54}$$

so that the dimensionless energy eigenvalues ε_n are given by

$$\varepsilon_n = n + \frac{1}{2} \tag{55}$$

Verify that this formula for the bound state energies is correct for $n = 0, 1, 2, 3, 4$
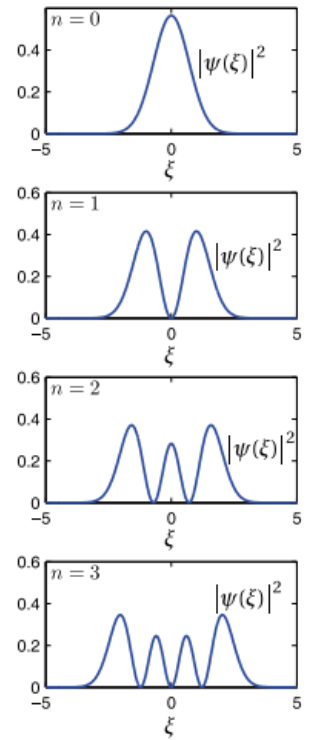
Figure 24: The probability distributions for the ground state and the first three excited states of the harmonic oscillator.
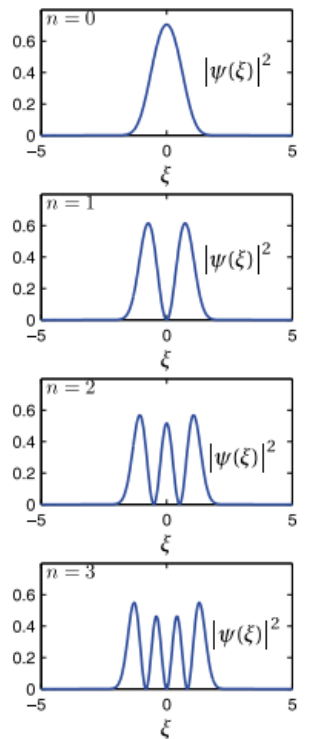
Figure 25: The probability distributions for the ground state and the first three excited states for the potential in Problem 4.5

P4.5 Now redo this entire problem, but with the harmonic oscillator potential replaced by

$$V(x) = \mu x^4 \tag{56}$$

so that we have

$$-\frac{\bar{h}^2}{2m}\frac{d^2\psi}{dx^2} + \mu x^4 \psi = E\psi \tag{57}$$

With this new potential you will need to find new formulas for the characteristic length and energy so that you can use dimensionless scaled variables as you did with the harmonic oscillator. Choose a so that your scaled equation is

$$-\frac{1}{2}\frac{d^2\psi}{d\xi^2} + \xi^4\psi = \varepsilon\psi \tag{58}$$

with $E = \varepsilon\bar{3}$. Use algebra by hand to show that

$$a = \left(\frac{\bar{h}^2}{m\mu}\right)^{1/6} \qquad \bar{E} = \left(\frac{\bar{h}^4\mu}{m^2}\right)^{1/3} \tag{59}$$

Find the first 5 bound state energies by finding the first 5 values of ε_n in the formula $E_n = \varepsilon_n\bar{E}$.

ANIMATING THE WAVE EQUATION: STAGGERED LEAPFROG

In the last couple of labs we handled the wave equation by Fourier analysis, turning the partial differential equation into a set of ordinary differential equations using separation of variables.1 But separating the variables and expanding in orthogonal functions is not the only way to solve partial differential equations, and in fact in many situations this technique is awkward, ineffective, or both. In this lab we will study another way of solving partial differential equations using a spatial grid and stepping forward in time. As an added attraction, this method automatically supplies a beautiful animation of the solution. There are several algorithms of this type that can be used on wave equations, so this is just an introduction to a larger subject. The method we will show you here is called staggered leapfrog; it is the simplest good method that we know

THE WAVE EQUATION WITH STAGGERED LEAPFROG

Consider again the classical wave equation with wave speed c.

$$\frac{\partial^2 y}{\partial t^2} - c^2 \frac{\partial^2 y}{\partial x^2} = 0 \tag{60}$$

For a string, the wave speed is related to tension and density via $c = \sqrt{T/\mu}$ The boundary conditions are usually either of Dirichlet type (values specified):

$$y(0,t) = f_{left}(t); y(L,t) = f_{right}(t) \tag{61}$$

or of Neumann type (derivatives specified):

$$\frac{\partial y}{\partial x}(0) = g_{left}(t); \frac{\partial y}{\partial x}(L) = g_{right}(t) \tag{62}$$

Sometimes mixed boundary conditions specify a relation between the value and derivative, as at the bottom of the hanging chain. In any case, some set of conditions at the endpoints are required to solve the wave equation. It is also necessary to specify the initial state of the string, giving its starting position and velocity as a function of position:

$$y(x, t = 0) = y_0(x) \quad ; \quad \frac{\partial y(x,t)}{\partial t}\Big|_{t=0} = v_0(x) \tag{63}$$

Both of these initial conditions are necessary because the wave equation is second order in time, just like Newton's second law, so initial displacement and velocity must be specified at each point to find a unique solution. To numerically solve the classical wave equation via staggered leapfrog we approximate both the temporal and spatial derivatives in Eq.(**??**) with centered finite differences, like this:

$$\begin{aligned}\frac{\partial^2 y}{\partial t^2} &\approx \frac{y_j^{n+1} - 2y_j^n + y_j^{n-1}}{\tau^2} \\ \frac{\partial^2 y}{\partial x^2} &\approx \frac{y_{j+1}^n - 2y_j^n + y_{j-1}^n}{h^2}\end{aligned} \tag{64}$$

In this notation, spatial position is indicated by a subscript j, referring to grid points x_j, while position in time is indicated by superscripts n, referring to time points t_n so that $y(x_j, t_n) = y_j^n$. The time steps and the grid spacings are assumed to be uniform with time step called τ and grid spacing called h. The staggered leapfrog algorithm aims to find y_j one time step into the future, denoted by y_j^{n+1}, from the current and previous values of y_j. To derive the algorithm put Eqs.(**??**) into Eq. (**??**) and solve for y_j^{n+1} to find

$$y_j^{n+1} = 2y_j^n - y_j^{n-1} + \frac{c^2 \tau^2}{h^2}(y_{j+1}^n - 2y_j^n + y_{j-1}^n) \tag{65}$$

P5.1 Derive Eq. (**??**) using the approximate second derivative formulas. This is really simple, so just do it on paper. Equation (**??**) can only be used at interior spatial grid points because the $j+1 or j-1$ indices reach beyond the grid at the first and last grid points. The

behavior of the solution at these two end points is determined by the boundary conditions. Since we will want to use both fixed value and derivative boundary conditions, let's use a cell-centered grid with ghost points (with N cells and $N+2$ grid points) so we can easily handle both types without changing our grid. If the values at the ends are specified we have

$$\frac{y_0^{n+1} + y_1^{n+1}}{2} = f_{\text{left}}(t_{n+1}) \Rightarrow y_0^{n+1} = -y_1^{n+1} + 2f_{\text{left}}(t_{n+1})$$

$$\frac{y_{N+1}^{n+1} + y_N^{n+1}}{2} = f_{\text{right}}(t_{n+1}) \Rightarrow y_{N+1}^{n+1} = -y_N^{n+1} + 2f_{\text{right}}(t_{n+1}) \tag{66}$$

If the derivatives are specified then we have

$$\frac{y_1^{n+1} - y_0^{n+1}}{h} = g_{\text{left}}(t_{n+1}) \Rightarrow y_0^{n+1} = y_1^{n+1} - hg_{\text{left}}(t_{n+1})$$

$$\frac{y_{N+1}^{n+1} - y_N^{n+1}}{h} = g_{\text{right}}(t_{n+1}) \Rightarrow y_{N+1}^{n+1} = y_N^{n+1} + hg_{\text{right}}(t_{n+1}) \tag{67}$$

To use staggered leapfrog, we first advance the solution at all interior points to the next time step using Eq. (**??**), then we apply the boundary conditions using the appropriate equation from Eqs. (**??**)-(**??**) to find the values of y at the end points, and then we are ready to take another time step. The staggered leapfrog algorithm in Eq. (**??**) requires not just y at the current time level y_j^n but also y at the previous time level y_j^{n1}. This means that we'll need to keep track of three arrays: an array y for the current values y_j^n, an array yold for the values at the previous time step y_j^{n-1}, and an array ynew for the values at the next time step y_j^{n+1}. At time $t = 0$ when the calculation starts, the initial position condition gives us the current values y_j^n, but we'll have to make creative use of the initial velocity condition to create an appropriate yold to get started. To see how this works, let's denote the initial values of y on the grid by y_j^0, the values after the first time step by y_j^1, and the unknown previous values (yold) by y_j^{-1}. A centered time derivative at $t = 0$ turns the initial velocity condition from Eq. (**??**) into

$$\frac{y_j^1 - y_j^{-1}}{2\tau} = v_0(x_j) \tag{68}$$

This gives us an equation for the previous values y_j^{-1}, but it is in terms of the still unknown future values y_j^1. However, we can use Eq. (5.6) to obtain another relation between y_j^1 and y_j^{-1}. Leapfrog at the first step ($n = 0$) says that

$$y_j^1 = 2Y_j^0 - y_j^{-1} + \frac{c^2\tau^2}{h^2}(y_{j+1}^0 - 2y_j^0 + y_{j-1}^0) \tag{69}$$

If we insert this expression for y_j^1 into Eq. (**??**), we can solve for y_j^{-1} in terms of known quantities:

$$y_j^{-1} = Y_j^0 - v_0(x_j)\tau + \frac{c^2\tau^2}{2h^2}(y_{j+1}^0 - 2y_j^0 + y_{j-1}^0) \tag{70}$$

P5.2   Derive Eq. (70) from Eqs. (68) and (69). Again, just use paper and pencil. OK; we are now ready to code the staggered leapfrog algorithm.

P5.3

(a) Start by making a cell-centered x grid with ghost points over the region $0 \le x \le L$, with $L = 1$ m. Use $N = 200$ cells, so you have 202 grid points. Define the initial displacement y as Gaussian bump with 1 cm amplitude in the middle of the string, like this

```
y = 0.01 * np.exp(-(x-L/2)**2 / 0.02)
```

and the initial velocity vy to be zero everywhere along the string. Used fixed-value boundary conditions, with $y(0) = 0$ and $y(L) = 0$. Plot the initial position just to make sure your grid is right and that your initial position array looks reasonable

(b) Assume that the wave speed on this string is $c = 2$ m/s, and pick the time step as $tau = 0.2 * h/c$. (We'll explore this choice more later.) Then create a variable yold using Eq. (**??**), and enforce the boundary conditions on yold. As you write this code, don't write a bunch of for loops to iterate through all the points. Instead, assign all interior points at once using the NumPy array colon indexing method (e.g. $y[1 : -1]$ accesses all the interior points of y) and then set the boundary conditions explicitly

(c) Now it is time to code the main staggered leapfrog algorithm and make an animation of the solution. Since it has been a couple of labs since we made an animation, here is a framework for the code to remind you of the basic animation commands:

```
ynew = np.zeros_like(y)
j = 0
t = 0
tmax = 2
plt.figure(1) # Open the figure window
# the loop that steps the solution along
while t < tmax:
    j = j+1
    t = t + tau
    # Use leapfrog and the boundary conditions
    # to load ynew with y at the next time
    # step using y and yold
    # update yold and y for next timestep
    # remember to use np.copy
    # make plots every 50 time steps
    if j % 50 == 0:
        plt.clf() # clear the figure window
        plt.plot(x,y,'b-')
        plt.xlabel('x')
        plt.ylabel('y')
        plt.title('time={:1.3f}'.format(t))
        plt.ylim([-0.03,0.03])
        plt.xlim([0,1])
        plt.draw() # Draw the plot
        plt.pause(0.1) # Give the computer time to draw
```

The actual staggered leapfrog code is missing above. YouâĂŹll need to write that. Run the animations long enough that you can see the reflection from the ends and the way the two pulses add together and pass right through each other.

(d) Once you have it running, experiment with various time steps τ. Show by numerical experimentation that if $\tau > h/c$ the algorithm blows up spectacularly. This failure is called a numerical instability and we will be trying to avoid it all semester. This limit is called the CourantFriedrichs-Lewy condition, or sometimes the CFL condition, or

Figure 27: Richard Courant (left), Kurt Friedrichs (center), and Hans Lewy (right) described the CFL instability condition in 1928 .
sometimes (unfairly) just the Courant condition

(e) Change the boundary conditions so that $\frac{\partial y}{\partial x} = 0$ at each end and watch how the reflection occurs in this case.

(f) Change the initial conditions from initial displacement with zero velocity to initial velocity with zero displacement. Use an initial Gaussian velocity pulse just like the displacement pulse you used earlier and use fixed-end boundary conditions. Watch how the wave motion develops in this case. (You will need to change the y-limits in the axis command to see the vibrations with these parameters.) Then find a slinky, stretch it out, and whack it in the middle to verify that the math does the physics right.

THE DAMPED WAVE EQUATION

We can modify the wave equation to include damping of the waves using a linear damping term, like this:

$$\frac{\partial^2 y}{\partial t^2} + \Upsilon \frac{\partial y}{\partial t} - c^2 \frac{\partial^2 y}{\partial x^2} = 0 \tag{71}$$

with c constant. The staggered leapfrog method can be used to solve Eq. (**??**) also. To do this, we use the approximate first derivative formula

$$\frac{\partial y}{\partial t} \approx = \frac{y_j^{n+1} - y_j^{n-1}}{2\tau} \tag{72}$$

along with the second derivative formulas in Eqs. (**??**) and find an expression for the values one step in the future:

$$y_j^{n+1} = \frac{1}{2+\Upsilon\tau}\left(4y_j^n - 2y_j^{n-1} + \Upsilon\tau y_j^{n-1} + \frac{2c^2\tau^2}{h^2}(y_{j+1}^n - 2y_j^n + y_{j-1}^n)\right) \tag{73}$$
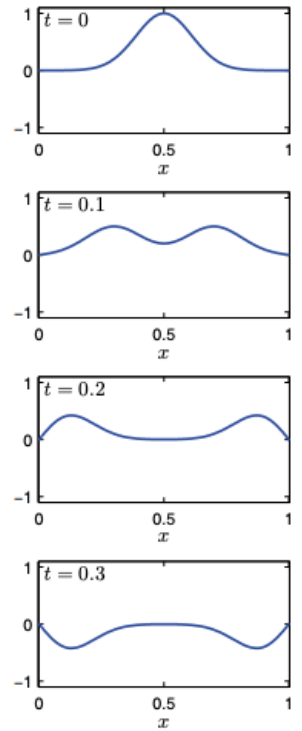

Figure 26: Snapshots of the evolution of a wave on a string with fixed ends and an initial displacement but no initial velocity. (See Problem 5.3(b))
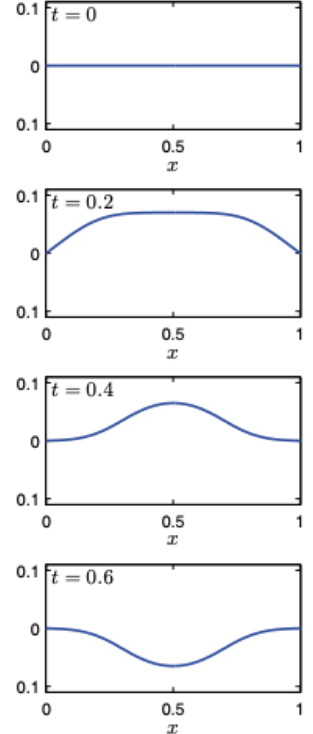

Figure 28: Snapshots of the evolution of a wave on a string with fixed ends and no initial displacement but with an initial velocity. (See Problem 5.3(f))

P5.4

(a) Derive Eq. (**??**).

(b) Find a new formula for the initial value of yold using Eqs.(**??**) and (**??**). When you get the answer, ask your TA or instructor to check to see if you got it right.

(c) Modify your staggered leapfrog code to include damping with $\Upsilon = 0.2$. Then run your animation with the initial conditions in Problem 5.3(c) and verify that the waves damp away. You will need to run for about 25 s and you will want to use a big skip factor so that you don't have to wait forever for the run to finish. Include some code to record the maximum value of $y(x)$ over the entire grid as a function of time and then plot it as a function of time at the end of the run so that you can see the decay caused by Υ. The decay of a simple harmonic oscillator is exponential, with amplitude proportional to $e^{\Upsilon t/2}$. Scale this time decay function properly and lay it over your maximum y plot to see if it fits. Can you explain why the fit is as good as it is? (Hint: think about doing this problem via separation of variables.)
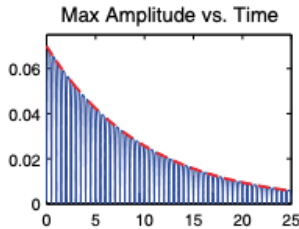
Figure 29: The maximum amplitude of oscillation decays exponentially for the damped wave equation. (Problem 5.4(c))

THE DAMPED AND DRIVEN WAVE EQUATION

Finally, let's look at what happens when we add an oscillating driving force to our string, so that the wave equation becomes

$$\frac{\partial^2 y}{\partial t^2} + \Upsilon \frac{\partial y}{\partial t} - c^2 \frac{\partial^2 y}{\partial x^2} = \frac{f(x)}{\mu} \cos(\omega t) \qquad (74)$$

At the beginning of Lab 3 we discussed the qualitative behavior of this system. Recall that if we have a string initially at rest and then we start to push and pull on a string with an oscillating force/length of $f(x)$, we launch waves down the string. These waves reflect back and forth on the string as the driving force continues to launch more waves. The string motion is messy at first, but the damping in the system causes the the transient waves from the initial launch and subsequent reflections to eventually die away. In the end, we are left with a steady-state oscillation of the string at the driving frequency ω. Now that we have the computational tools to model the time evolution of the system, let's watch this behavior.

P5.5

(a) Re-derive the staggered leapfrog algorithm to include both driving and damping forces as in Eq. (**??**).

(b) Modify your code from Problem 5.4 to use this new algorithm. We'll have the string start from rest, so you don't need to worry about finding yold. Just set $y = 0$ and $yold = 0$ and enter the time-stepping loop. This problem involves the physics of waves on a real guitar string, so we'll need to use realistic values for our parameters. Use $T = 127$, $\mu = 0.003$, and $L = 1.2$ (in SI units) and remember that $c = \sqrt{T/\mu}$. Use the same driving force as in Problem 3.2(a)

$$f(x) = \begin{cases} 0.73 & \text{if } 0.8 \le x \le 1 \\ 0 & \text{otherwise} \end{cases} \qquad (75)$$

and set the driving frequency at $\omega = 400$. Choose a damping constant Υ that is the proper size to make the system settle down to steady state after 20 or 30 bounces of the string. (You will have to think about the value of ω that you are using and about your damping rate result from problem 5.4 to decide which value of Υ to use to make this happen.) Run the model long enough that you can see the transients die away and the string settle into the steady oscillation at the driving frequency. You may find yourself looking at a flat-line plot with no oscillation at all. If this happens look at the vertical scale of your plot and remember that we are doing real physics here. If your vertical scale goes from âĽŠ1 to 1, you are expecting an oscillation amplitude of 1 meter on your guitar string. Compare the steady state mode to the shape found in Problem 3.2(a) (see Fig. 3.1). Then run again with $\omega = 1080$, which is close to a resonance, and again see the system come into steady oscillation at the driving frequency.

Figure 30: Snapshots of the evolution a driven and

THE 2-D WAVE EQUATION WITH STAGGERED LEAPFROG

TWO DIMENSIONAL GRIDS

In this lab we will do problems in two spatial dimensions, x and y, so we need to spend a little time thinking about how to represent 2-D grids. For a simple rectangular grid where all of the cells are the same size, 2-D grids are pretty straightforward. We just divide the x-dimension into equally sized regions and the y-dimension into equally sized regions, and the two one dimensional grids intersect to create rectangular cells. Then we put grid points either at the corners of the cells (cell-edge) or at the centers of the cells (cell-centered). On a cell-center grid we'll usually want ghost point outside the region of interest so we can get the boundary conditions right.

NumPy has a nice way of creating rectangular two-dimensional grids using the meshgrid command. You can create 2-d rectangle defined by $x \in [a,b]$ and $y \in [c,d]$ and then plot a function on that grid this way:

Cell-Edge Grid

Cell-Center Grid with Ghost Points

Figure 31: Two types of 2-D grids.

```python
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np
# Make 1D x and y arrays
Nx=20
a=-1.5
b=1.5
x,hx = np.linspace(a,b,Nx,retstep = True)
Ny=10
c=-1.2
d=1.2
y,hy = np.linspace(c,d,Ny,retstep = True)
# Make the 2D grid and evaluate a function
X, Y = np.meshgrid(x,y,indexing='ij')
Z = X**2 + Y**2
# Plot the function as a surface.
fig = plt.figure(1)
ax = fig.gca(projection='3d')
surf = ax.plot_surface(X, Y, Z, cmap=cm.viridis)
plt.xlabel('x')
plt.ylabel('y')
fig.colorbar(surf)
```

The argument $indexing =' ij'$ in the meshgrid function switches the ordering of the elements in the resulting matrices from the matrix indexing convention $Z[row, col]$ to the function indexing convention $Z[x_i, y_y]$ for representing $Z(x_i, y_i)$.

P6.1

(a) Use the code fragment above in a program to create a 30-point celledge grid in x and a 50-point cell-edge grid in y with $a = 0, b = 2, c = 1, d = 3$. Switch back and forth between $indexing =' ij'$ and $indexing =' xy'$, and look at the different matrices that result. Convince the TA that you know what the difference is. (HINT: When we write matrices, rows vary in the y dimension and columns vary in the x direction, whereas with $Z(x_i, y_i)$ we have a different convention. NumPy's naming convention seems backward to us, but we didn't come up with it. Sorry.) We recommend that you use the $indexing =' ij'$ convention, as it tends to more readable code for representing $Z(x_i, y_i)$.

(b) Using this 2 - D grid, evaluate the following function of x and y:

$$f(x,y) = e^{-(x^2+y^2)} \cos(5\sqrt{x^2 + y^2}) \tag{76}$$

Use the $plot_surface$ command to make a surface plot of this function. Properly label the x and y axes with the symbols x and y, to get a plot like Fig. 6.2.

There are a lot more options for plotting surfaces in Python, but we'll let you explore those on your own. For now, let's do some physics on a two-dimensional grid.
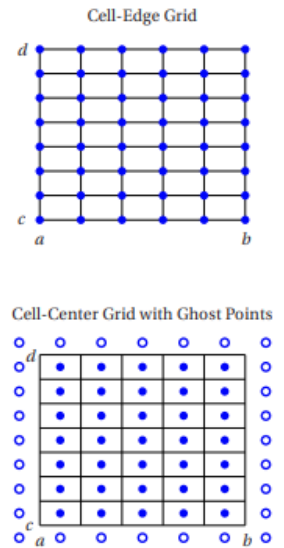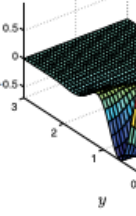
THE TWO-DIMENSIONAL WAVE EQUATION

The wave equation for transverse waves on a rubber sheet is

$$\mu \frac{\partial^2 z}{\partial t^2} = \sigma \left(\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} \right) \tag{77}$$

In this equation μ is the surface mass density of the sheet, with units of mass/area. The quantity σ is the surface tension, which has rather odd units. By inspecting the equation above you can find that ĬČ has units of force/length, which doesn't seem right for a surface. But it is, in fact, correct as you can see by performing the following thought experiment. Cut a slit of length L in the rubber sheet and think about how much force you have to exert to pull the lips of this slit together. Now imagine doubling L; doesn't it seem that you should have to pull twice as hard to close the slit? Well, if it doesn't, it should; the formula for this closing force is given by σL, which defines the meaning of σ. We can solve the two-dimensional wave equation using the same staggered leapfrog techniques that we used for the one-dimensional case, except now we need to use a two dimensional grid to represent $z(x,y,t)$. We'll use the notation $z_{j,k}^n = z(x_j, y_k, t_n)$ to represent the function values. With this notation, the derivatives can be approximated as

$$\frac{\partial^2 z}{\partial t^2} \approx \frac{z_{j,k}^{n+1} - 2z_{j,k}^n + z_{j,k}^{n-1}}{\tau^2}) \tag{78}$$

$$\frac{\partial^2 z}{\partial x^2} \approx \frac{z_{j+1,k}^n - 2z_{j,k}^n + z_{j-1,k}^n}{h_x^2}) \tag{79}$$

$$\frac{\partial^2 z}{\partial y^2} \approx \frac{z_{j,k+1}^n - 2z_{j,k}^n + z_{j,k-1}^n}{h_y^2}) \tag{80}$$

where h_x and h_y are the grid spacings in the x and y dimensions. We insert these three equations into Eq. (??) to get an expression that we can solve for z at the next time (i.e. $z_{j,k}^{n+1}$). Then we use this expression along with the discrete version of the initial velocity condition

$$v_0(x_j, y_k) \approx \frac{z_{j,k}^{n+1} - z_{j,k}^{n-1}}{2\tau} \tag{81}$$

to find an expression for the initial value of $z_{j,k}^{n-1}$ (i.e. *zold*) so we can get things started.

P6.2

(a) Derive the staggered leapfrog algorithm for the case of square cells with $h_x = h_y = h$. Write a program that animates the solution of the two dimensional wave equation on a square region that is $[5,5][5,5]$ and that has fixed edges. Use a cell-edge square grid with the edge-values pinned to zero to enforce the boundary condition. Choose $\sigma = 2N/m$ and $\mu = 0.3kg/m^2$ and use a displacement initial condition that is a Gaussian pulse with zero velocity

$$z(x,y,0) = e^{-5(x^2+y^2)} \tag{82}$$

This initial condition doesn't strictly satisfy the boundary conditions, so you should pin the edges to zero. Getting the animation to work can be tricky, so here is a framework of code for the animation loop:

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
# your code to initialize things
tfinal=10
t=np.arange(0,tfinal,tau)
skip=10
fig = plt.figure(1)
# here is the loop that steps the solution along
for m in range(len(t)):
# Your code to step the solution
# make plots every skip time steps
if m % skip == 0:
plt.clf()
ax = fig.gca(projection='3d')
surf = ax.plot_surface(X,Y,z)
```

Figure 32: F
lem 6.1. T
this figure w
Matlab. Pyt
engine is so
in compariso
so you won't
quite so nice
getting the s
painful in Py

```
17 ax.set_zlim(-0.5, 0.5)
18 plt.xlabel('x')
19 plt.ylabel('y')
20 plt.draw()
21 plt.pause(0.1)
```

Run the simulation long enough that you see the effect of repeated reflections from the edges.

(b) You will find that this two-dimensional problem has a Courant condition similar to the one-dimensional case, but with a factor out front:

$$\tau < fh\sqrt{\frac{\mu}{\sigma}} \tag{83}$$

where $\sqrt{\frac{\mu}{\sigma}}$ is the wave speed and f is an arbitrary constant. Determine the value of the constant f by numerical experimentation. (Try various values of τ and discover where the boundary is between numerical stability and instability.)

(c) Also watch what happens at the center of the sheet by making a plot of $z(0,0,t)$ there. In one dimension the pulse propagates away from its initial position making that point quickly come to rest with $z = 0$. This also happens for the three-dimensional wave equation. But something completely different happens for an even number of dimensions; you should be able to see it in your plot by looking at the behavior of $z(0,0,t)$ before the first reflection comes back.

(d) Finally, change the initial conditions so that the sheet is initially flat but with the initial velocity given by the Gaussian pulse of Eq. (**??**). In one dimension when you pulse the system like this the string at the point of application of the pulse moves up and stays up until the reflection comes back from the ends of the system. (We did this experiment with the slinky in Lab 5.) Does the same thing happen in the middle of the sheet when you apply this initial velocity pulse? Answer this question by looking at your plot of $z(0,0,t)$. You should find that the two-dimensional wave equation behaves very differently from the one-dimensional wave equation.
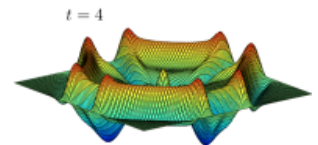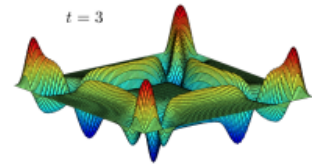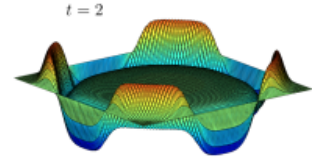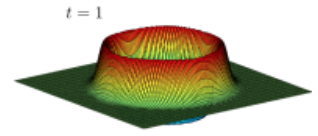
Figure 33: A wave on a rubber sheet with fixed edges.

ELLIPTIC, HYPERBOLIC, AND PARABOLIC PDES

Let's step back and look at some general concepts related to solving partial differential equations. Three of the most famous PDEs of classical physics are

(i) Poisson's equation for the electrostatic potential $V(x,y)$ given the charge density (x,y)

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = \frac{-\rho}{\varepsilon_0} + \text{Boundary Conditions} \tag{84}$$

(ii) The wave equation for the wave displacement $y(x,t)$

$$\frac{\partial^2 y}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 y}{\partial t^2} = 0 + \text{Boundary Conditions} \tag{85}$$

(iii) The thermal diffusion equation for the temperature distribution $T(x,t)$ in a medium with diffusion coefficient D

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2} + \text{Boundary Conditions} \tag{86}$$

To this point in the course, we've focused mostly on the wave equation, but over the next several labs we'll start to tackle some of the other PDEs.

Mathematicians have special names for these three types of partial differential equations, and people who study numerical methods often use these names, so let's discuss them a bit. The three names are elliptic, hyperbolic, and parabolic. You can remember which name goes with which of the equations above by remembering the classical formulas for these conic sections:

$$\text{ellipse:} \frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \tag{87}$$

$$\text{hyperbola:} \frac{x^2}{a^2} - \frac{y^2}{b^2} = 1 \tag{88}$$

$$\text{parabola:} y = ax^2 \tag{89}$$

Compare these equations with the classical PDE's above and make sure you can use their resemblances to each other to remember the following rules: Poisson's equation is elliptic, the wave equation is hyperbolic, and the diffusion equation is parabolic. These names are important because each different type of equation requires a different type of algorithm and boundary conditions.

Elliptic equations require the same kind of boundary conditions as Poisson's equation: $V(x,y)$ specified on all of the surfaces surrounding the region of interest. Notice that there is no time delay in electrostatics. When all of the bounding voltages are specified, Poisson's equation says that $V(x,y)$ is determined instantly throughout the region surrounded by these bounding surfaces. Because of the finite speed of light this is incorrect, but Poisson's equation is a good approximation to use in problems where things happen slowly compared to the time it takes light to cross the computing region.

To understand hyperbolic boundary conditions, think about a guitar string described by the transverse displacement function $y(x,t)$. It makes sense to give spatial boundary conditions at the two ends of the string, but it makes no sense to specify conditions at both $t = 0$ and $t = t_{final}$ because we don't know the displacement in the future. This means that you can't pretend that (x,t) are like (x,y) in Poisson's equation and use "surrounding"-type boundary conditions. But we can see the right thing to do by thinking about what a guitar string does. With the end positions specified, the motion of the string is determined by giving it an initial displacement $y(x,0)$ and an initial velocity $\partial y(x,t)/\partial t|_{t=0}$, and then letting the motion run until we reach the final time. So for hyperbolic equations the proper boundary conditions are to specify end conditions on y as a function of time and to specify the initial conditions $y(x,0)$ and $\partial y(x,t)/\partial t|_{t=0}$.

Parabolic boundary conditions are similar to hyperbolic ones, but with one difference. Think about a thermally-conducting bar with its ends held at fixed temperatures. Once again, surrounding-type boundary conditions are inappropriate because we don't want to specify the future. So as in the hyperbolic case, we can specify conditions at the ends of the bar, but we also want to give initial conditions at $t = 0$. For thermal diffusion we specify the initial temperature $T(x,0)$, but that's all we need; the "velocity" $\partial T/\partial t$ is determined by Eq. (**??**), so it makes no sense to give it as a separate boundary condition. Summarizing: for parabolic equations we specify end conditions and a single initial condition $T(x,0)$ rather than the two required by hyperbolic equations.

If this seems like an arcane side trip into theory, we're sorry, but it's important. When you numerically solve partial differential equations you will spend 10 % of your time coding the equation itself and 90 % of your time trying to make the boundary conditions work. It's important to understand what the appropriate boundary conditions are.

Finally, there are many more partial differential equations in physics than just these three. Nevertheless, if you clearly understand these basic cases you can usually tell what boundary conditions to use when you encounter a new one. Here, for instance, is Schrödinger's equation:

$$i\bar{h}\frac{\partial \psi}{\partial t} = -\frac{\bar{h}^2}{2m}\frac{\partial^2 \psi}{\partial x^2} + V\psi \tag{90}$$

which is the basic equation of quantum (or "wave") mechanics. The wavy nature of the physics described by this equation might lead you to think that the proper boundary conditions on $\psi(x,t)$ would be hyperbolic: end conditions on ψ and initial conditions on ψ and $\partial \psi/\partial t$. But if you look at the form of the equation, it looks like thermal diffusion. Looks are not misleading here; to solve this equation you only need to specify ψ at the ends in x and the initial distribution $\psi(x,0)$, but not its time derivative. And what are you supposed to do when your system is both hyperbolic and parabolic, like the wave equation with damping?

$$\frac{\partial^2 y}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 y}{\partial t^2} - \frac{1}{D}\frac{\partial y}{\partial t} = 0 \tag{91}$$

The rule is that the highest-order time derivative wins, so this equation needs hyperbolic boundary conditions

P6.3 Make sure you understand this material well enough that you are comfortable answering basic questions about PDE types and what types of boundary conditions go with them on a quiz and/or an exam. Then explain it to the TA to pass this problem off.

THE DIFFUSION EQUATION AND IMPLICIT METHODS

ANALYTIC APPROACH TO THE DIFFUSION EQUATION

Now let's attack the diffusion equation

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2} \tag{92}$$

This equation describes how the distribution T (often temperature) diffuses through a material with a constant diffusion coefficient D. The diffusion equation can be approached analytically via separation of variables by assuming that T is of the form $T(x,t) = g(x)f(t)$. Plugging this form into the diffusion equation, we find

$$\frac{1}{D}\frac{\dot{f}(t)}{f(t)} = \frac{g''(x)}{g(x)} \tag{93}$$

The left-hand side depends only on time, while the right-hand side depends only on space, so both sides must equal a constant, say $-a^2$. Thus, $f(t)$ must satisfy

$$\dot{f}(t) = -\gamma f(t) \tag{94}$$

where $\gamma = a^2 D$ so that $f(t)$ is

$$f(t) = e^{-\gamma t} \tag{95}$$

Meanwhile $g(x)$ must satisfy

$$g''(x) + a^2 g(x) = 0 \tag{96}$$

If we specify edge-value boundary conditions so that $T(x = 0, t) = 0$ and $T(x = L, t) = 0$ then the solution to Eq. (**??**) is simply

$$g(x) = \sin(ax) \tag{97}$$

and the separation constant can take on the values $a = n\pi/L$, where n is an integer. Any initial distribution $T(x, t = 0)$ that satisfies these boundary conditions can be composed by summing these sine functions with different weights using Fourier series techniques. Notice that higher spatial frequencies (i.e. large n) damp faster, according to Eq. (**??**). We already studied how to use separation of variables computationally in the first several labs of this manual, so let's move directly to time-stepping methods for solving the diffusion equation.

NUMERICAL APPROACH: A FIRST TRY

Let's try to solve the diffusion equation on a grid as we did with the wave equation. If we finite difference the diffusion equation using a centered time derivative and a centered second derivative in x we get

$$\frac{T_j^{n+1} - T_j^{n-1}}{2\tau} = \frac{D}{h^2}\left(T_{j+1}^n - 2T_j^n + T_{j-1}^n\right) \tag{98}$$

Solving for T_j^{n+1} to obtain an algorithm similar to leapfrog then gives

$$T_j^{n+1} = T_j^{n-1} + \frac{2D\tau}{h^2}\left(T_{j+1}^n - 2T_j^n + T_{j-1}^n\right) \tag{99}$$

There is a problem starting this algorithm because of the need to have T one time step in the past (T_j^{n+1}), but even after we work around this problem this algorithm turns out to be worthless. We won't make you code it up, but if you did, you'd find that no matter how small a time step τ you choose, you encounter the same kind of instability that plagues staggered leapfrog when the step size got too big (infinite zig-zags). Such an algorithm is called unconditionally unstable, and is an invitation to keep looking. This must have

been a nasty surprise for the pioneers of numerical analysis who first encountered it.

For now, let's sacrifice second-order accuracy to obtain a stable algorithm. If we don't center the time derivative, but use instead a forward difference we find

$$\frac{T_j^{n+1} - T_j^n}{\tau} = \frac{D}{h^2}\left(T_{j+1}^n - 2T_j^n + T_{j-1}^n\right) \tag{100}$$

This algorithm has problems since the left side of Eq. (7.9) is centered at time $t_{n+1/frac12}$, while the right side is centered at time t_n. This makes the algorithm inaccurate, but it turns out that it is stable if τ is small enough. Solving for T_j^{n+1} yields

$$T_j^{n+1} = T_j^n + \frac{D\tau}{h^2}\left(T_{j+1}^n - 2T_j^n + T_{j-1}^n\right) \tag{101}$$

Figure 34: Diffusion of the $n = 1$ sine temperature distribution given in Problem 7.1(a).

P7.1

(a) Modify one of your staggered leapfrog programs that uses a cell-center grid to implement Eq. (**??**) to solve the diffusion equation on the interval $[0, L]$ with initial distribution

$$T(x, 0) = \sin(\pi x / L) \tag{102}$$

and boundary conditions $T(0) = T(L) = 0$. Use $D = 2$, $L = 3$, and $N = 20$. You don't need to make a space-time surface plot like Fig. 7.1. Just make a line plot that updates each time step as we've done previously. This algorithm has a CFL condition on the time step τ of the form

$$\tau \leq C\frac{h^2}{D} \tag{103}$$

Determine the value of C by numerical experimentation. Test the accuracy of your numerical solution by overlaying a graph of the analytic solution. Plot the numerical solution as points and the exact solution as a line so you can tell the difference. Show that your grid solution matches the exact solution with increasing accuracy as the number of grid points N is increased from 20 to 40 and then to 80. You can calculate the RMS error using something like

```
error = np.sqrt( np.mean( (T - exact)**2 ))
```

(b) Get a feel for what the diffusion coefficient does by trying several different values for D in your code. Give a physical description of this parameter to the TA.

(c) Now switch your boundary conditions to be insulating, with $\partial T / \partial x = 0$ at both ends. Explain what these two types of boundary conditions mean by thinking about a watermelon that is warmer in the middle than at the edge. Tell physically how you would impose both of these boundary conditions (specifying the value and specifying the derivative) on the watermelon and explain what the temperature history of the watermelon has to do with your plots of $T(x)$ vs. time.

Figure 35: Diffusion of the Gaussian temperature distribution given in Problem 7.1 (c) with insulating boundary conditions.

Even though this technique can give us OK results, the time step constraint for this method is onerous. The constraint is of the form $\tau < Ch^2$, where C is a constant. Suppose, for instance, that to resolve some spatial feature you need to decrease h by a factor of 5; then you will have to decrease τ by a factor of 25. This will make your code take forever to run, which motivates us to find a better way

IMPLICIT METHODS: THE CRANK-NICOLSON ALGORITHM

The time-stepping algorithms we have discussed so far are of the same type: at each spatial grid point j you use present, and perhaps past, values of $y(x, t)$ at that grid point and at neighboring grid points to find the future $y(x, t)$ at j. Methods like this, that depend in a simple way on present and past values to predict future values, are said to be *explicit* and are easy to code. They are also often numerically unstable, or have severe constraints on the size of the time step.

Implicit methods are generally harder to implement than explicit methods, but they have much better stability properties. The reason they are harder is that they assume that you already know the future. To give you a better feel for what "implicit" means, let's study the simple first-order differential equation
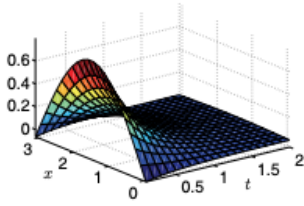
$$\frac{dy}{dt} = -y \tag{104}$$

P7.2

(a) Write a program to solve this equation using Euler's method:

$$\frac{y_{n+1} - y_n}{\tau} = -y_n \tag{105}$$

The program to solve for *y* using Euler's method is only a few lines of code (like less than 10 lines, including the plot command). Here are the first few lines:

```
tau = 0.5
tmax = 20.
t = np.arange(0,tmax,tau)
y = np.zeros_like(t)
```

Show by numerical experimentation that Euler's method is unstable for large τ. You should find that the algorithm that is unstable if $\tau > 2$. Use $y(0) = 1$ as your initial condition. This is an example of an explicit

(b) Notice that the left side of Eq. (**??**) is centered on time $t_{n+\frac{1}{2}}$ but the right side is centered on t_n. Fix this by centering the right-hand side at time $t_{n+\frac{1}{2}}$ by using an average of the advanced and current values of *y*, method.

$$y_n \Rightarrow \frac{y_n + y_{n+1}}{2}$$

Modify your program to implement this fix, then show by numerical experimentation that when τ becomes large this method doesn't blow up. It isn't correct because y_n bounces between positive and negative values, but at least it doesn't blow up. The presence of τ in the denominator is the tip-off that this is an implicit method, and the improved stability is the point of using something implicit.

(c) Now Modify Euler's method by making it fully implicit by using y_{n+1} in place of y_n on the right side of Eq. (**??**) (this makes both sides of the equation reach into the future). This method is no more accurate than Euler's method for small time steps, but it is much more stable and it doesn't bounce between positive and negative values. Show by numerical experimentation in a modified program that this fully implicit method damps even when τ is large. For instance, see what happens if you choose $\tau = 5$ with a final time of 20 seconds. The time-centered method of part (b) would bounce and damp, but you should see that the fully implicit method just damps. It's terribly inaccurate, and actually doesn't even damp as fast as the exact solution, but at least it doesn't bounce like part (b) or go to infinity like part (a). Methods like this are said to be "absolutely stable". Of course, it makes no sense to choose really large time steps, like $\tau = 100$ when you only want to run the solution out to 10 seconds.

THE DIFFUSION EQUATION WITH CRANK-NICOLSON

Now let's look at the diffusion equation again and see how implicit methods can help us find a stable numerical algorithm. Here's the equation again

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2} \tag{106}$$

We begin by finite differencing the right side as usual:

$$\frac{\partial T_j}{\partial t} = D\frac{T_{j+1} - 2T_j + T_{j-1}}{h^2} \tag{107}$$

Now we discretize the time derivative by taking a forward time derivative on the left:

$$\frac{T_j^{n+1} - T_j^n}{\tau} = D\frac{T_{j+1}^n - 2T_j^n + T_{j-1}^n}{h^2} \tag{108}$$

This puts the left side of the equation at time level $t_{n+\frac{1}{2}}$, while the right side is at tn. To put the right side at the same time level (so that the algorithm will be second-order accurate), we replace each occurrence of *T* on the right-hand side by the average
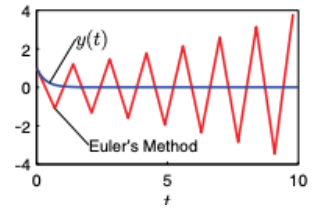
$$T^{n+\frac{1}{2}} = \frac{T^{n+1} + T^n}{2} \tag{109}$$

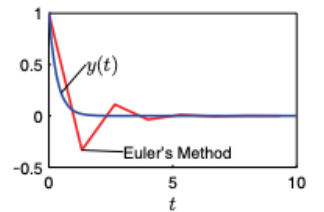Figure 36: Euler's method is unstable for $\tau > 2$. ($\tau = 2.1$ in this case.)

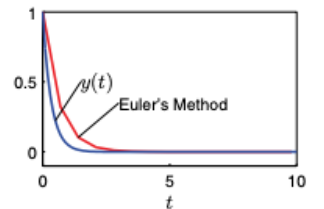Figure 37: Euler's method is unstable for $\tau > 2$. ($\tau = 2.1$ in this case.)

Figure 38: The fully implicit method in 7.2(c) with $\tau = 2.1$.

Figure 39: Phyllis Nicolson (1917-1968, English)

like this:

$$\frac{T_j^{n+1} - T_j^n}{\tau} = D\frac{T_{j+1}^{n+1} + T_{j+1}^n - 2T_j^{n+1} - 2T_j^n + T_{j-1}^{n+1} + T_{j-1}^n}{2h^2} \tag{110}$$

If you look carefully at this equation you will see that there is a problem: how are we supposed to solve for T_j^{n+1}? The future values T^{n+1} are all over the place, and they involve three neighboring grid points (T_{j-1}^{n+1}, T_j^{n+1}, and T_{j+1}^{n+1}) so we canâĂŹt just solve in a simple way for T_j^{n+1}. This is an example of why implicit methods are harder than explicit methods.

In the hope that something useful will turn up, let's put all of the variables at time level $n+1$ on the left, and all of the ones at level n on the right.

$$-T_{j-1}^{n+1} + \left(\frac{2h^2}{\tau D} + 2\right)T_j^{n+1} - T_{j+1}^{n+1} = T_{j-1}^n + \left(\frac{2h^2}{\tau D} - 2\right)T_j^n + T_{j+1}^n \tag{111}$$

We know this looks ugly, but it really isn't so bad. To solve for T_j^{n+1} we just need to solve a linear system, as we did in Lab 2 on two-point boundary value problems. When a system of equations must be solved to find the future values, we say that the method is implicit. This particular implicit method is called the CrankNicolson algorithm.

To see more clearly what we are doing, and to make the algorithm a bit more efficient, let's define a matrix A to describe the left side of Eq. (**??**) and another matrix B to describe the right side, like this:

$$\mathbf{A}T^{n+1} = \mathbf{B}T^n \tag{112}$$

T is now a column vector. The elements of A are given by

$$A_{j,k} = 0 \, except \, for:$$

$$A_{j,j-1} = -1; \quad A_{j,j} = \frac{2h^2}{\tau D} + 2; A_{j,j+1} = -1 \tag{113}$$

and the elements of B are given by

$$B_{j,k} = 0 \, except \, for:$$

$$B_{j,j-1} = -1; \quad B_{j,j} = \frac{2h^2}{\tau D} - 2; B_{j,j+1} = 1 \tag{114}$$

Once the boundary conditions are added to these matrices, Eq. (**??**) could be solved symbolically to find T^{n+1}

$$T^{n+1} = \mathbf{A}^{-1}\mathbf{B}T^n \tag{115}$$

However, since inverting a matrix is computationally expensive we will use Gauss elimination instead as we did in lab 2 (with SciPy's linalg.solve function). Here is a sketch of how you would implement the Crank-Nicolson algorithm in Python.

- Load the matrices A and B as given in Eq. (**??**) and Eq. (**??**) for all of the rows except the first and last. Since the diffusion coefficient D doesn't change with time you can load A and B just once before the time loop starts.

- The first and last rows involve the boundary conditions. Usually it is easier to handle the boundary conditions if we plan to do the linear solve of our matrix equation $\mathbf{A}T^{n+1} = \mathbf{B}T^n$ in two steps, like this:

```
import scipy.linalg as la
# matrix multiply to get the right-hand side
r = B@T
# set r as appropriate for the boundary conditions
r[0] = ...
r[-1] = ...
# Solve AT = r. The T we get is for the next time
    step.
# We don't need to keep track of previous T values
    , so just
# load the new T directly into T itself
T = la.solve(A,r)
```

  With this code we can just load the top and bottom rows of B with zeros, creating a right-hand-side vector *r* with zeros in the top and bottom positions. The top and bottom rows of A can then be loaded with the appropriate terms to enforce the desired boundary conditions on T^{n+1}, and the top and bottom positions of r can be

loaded as required just before the linear solve, as indicated above. For example, if the boundary conditions were $T(0) = 1$ and $T(L) = 5$, the top and bottom rows of A and the top and bottom positions of r would have been loaded like this (assuming a cell-center grid with ghost points):

```
1 # Set the A portion up where you define the matrix
2 A[0,0]   = 0.5
3 A[0,1]   = 0.5
4 A[-1,-1] = 0.5
5 A[-1,-2] = 0.5
6 ... # skipped code
7 # Set the r portion of the boundary condition
8 # down in the time loop for each iteration
9 r[0]  = 1
10 r[-1] = 5
```

so that the equations for the top and bottom rows are

$$\frac{T_0 + T_1}{2} = r_0 \quad \frac{T_N + T_{N+1}}{2} = r_{N+1} \tag{116}$$

The matrix B just stays out of the way (is zero) in the top and bottom rows.

- Once the matrices A and B are loaded, finding the new temperature inside the time loop is accomplished in the time loop by solving the matrix equation using the code fragment listed above.

P7.3

(a) Write program that implements the Crank-Nicolson algorithm with fixed-edge boundary conditions, $T(0) = 0$ and $T(L) = 0$. Test your program by running it with $D = 2$ and an initial temperature given by $T(x) = sin(\pi x/L)$. Try various values of τ and see how it compares with the exact solution. Verify that when the time step is too large the solution is inaccurate, but still stable. To do the checks at large time step you will need to use a long run time and not skip any steps in the plotting.

(b) Now study the accuracy of this algorithm by using various values of the cell number N and the time step τ. For each pair of choices run for $t = 5$ s and find the maximum difference between the exact and numerical solutions. You should find that the time step τ matters less than N. The number of cells N is the more important parameter for high accuracy in diffusion problems solved with Crank-Nicolson.

(c) Modify the Crank-Nicolson program to use boundary conditions $\partial T/\partial x = 0$ at the ends. Run with the same initial condition as in part (a) (which does not satisfy these boundary conditions) and watch what happens. Zoom in on the plots early in time to see what happens in the first few grid points during the first few time steps.
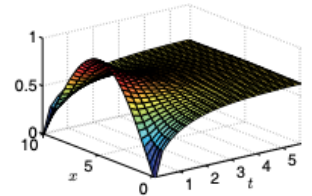

Figure 41: Solution to 7.3(c)

SCHRÖDINGER'S EQUATION

DERIVATIONS

Here is the time-dependent Schrödinger equation which governs the way a quantum wave function changes with time in a one-dimensional potential well $V(x)$:

$$i\bar{h}\frac{\partial \psi}{\partial t} = -\frac{\bar{h}^2}{2m}\frac{\partial^2 \psi}{\partial x^2} + V(x)\psi \tag{117}$$

Note that except for the presence of the imaginary unit i, this is very much like the diffusion equation. In fact, a good way to solve it is with the Crank-Nicolson algorithm. Not only is this algorithm stable for Schrödinger's equation, but it has another important property: it conserves probability. This is very important. If the algorithm you use does not have this property, then as ψ for a single particle is advanced in time you have (after a while) 3/4 of a particle, then 1/2, etc.

The Schrödinger equation usually describes the behavior of very tiny things on very short timescales, so SI quantities like kilograms, meters, and Joules are essentially infinite in quantum mechanics. Instead, we'll use a system of units called atomic units. In this system, lengths are measured in units of a_0 (the Bohr radius, $a_0 = 5.2910^{11}$), masses are measured in units of the electron mass me , energies are measured in units of E_0 ($E_0 = \alpha^2 mc^2 \approx 27eV$), and time is measured in units of t_0($t_0 = 24.2 attoseconds$). These base units are chosen so that the constants $\bar{h}$ and m both have the numerical value of 1 (e.g. $m = 1m_e$ for an electron).

P8.1 Use paper and pencil to derive a Crank-Nicolson algorithm to solve the Schrödinger equation. It will probably be helpful to use the material in Lab 7 as a guide (beginning with Eq. (**??**)). Make sure the $V(x)\psi$ term enters the algorithm correctly

PARTICLE IN A BOX

Let's use this algorithm for solving Schrödinger's equation to study the evolution of a particle in a box with

$$V(x) = \begin{cases} 0 & \text{if } -L < x < L \\ +\infty & \text{otherwise} \end{cases} \tag{118}$$

The infinite potential at the box edges is imposed by forcing the wave function to be zero at these points:

$$\psi(-L) = 0 \quad ; \quad \psi(L) = 0 \tag{119}$$

P8.2 Modify one of your programs from Lab 7 to implement the Crank-Nicolson algorithm derived above for the case of a particle in a box with $L = 10$. Note we are doing quantum mechanics and the imaginary parts matter now. When assigning complex variables in NumPy, you use the engineering complex number j, like this:

```
a = 1.0 + 0.5j
```

When you allocate your arrays, you'll need to specify up-front that they will hold complex values, like this:

```
A = np.zeros((N,N),dtype=np.complex_)
B = np.zeros_like(A,dtype=np.complex_)
```

(a) Write a script to solve the time-dependent Schrödinger equation using Crank-Nicolson. We find that a cell-edge grid is easiest, but you can also do cell-center with ghost points if you like. Start with a localized wave packet of width σ and momentum p:

$$\psi(x,0) = \frac{1}{\sqrt{\sigma\sqrt{\pi}}}e^{ipx/\bar{h}}e^{-x^2/(2\sigma^2)} \tag{120}$$

with $p = 2\pi$ and $\sigma = 2$. (Remember that $\bar{h}$ in (**??**) has the numerical value of 1 in our units.) This initial condition does not exactly satisfy the boundary conditions, but it is very close. Check to see how far off it is at the boundary, and decide how
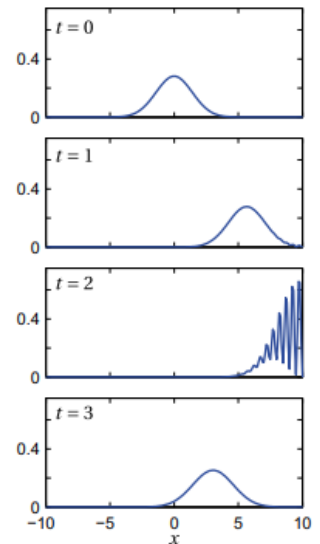
Figure 42: The probability density $|\psi(x)|^2$ of a particle in a box that initially moves to the right and then interferes with itself as it reflects from an infinite potential (Problem 8.2(a)).

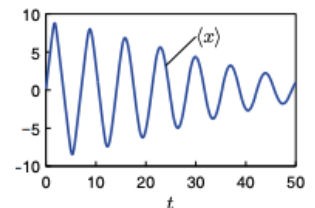Figure 43: The expectation position $\langle x \rangle$ for the particle in Fig. 8.1 as time progresses and the packet spreads out (Problem 8.2(c)).
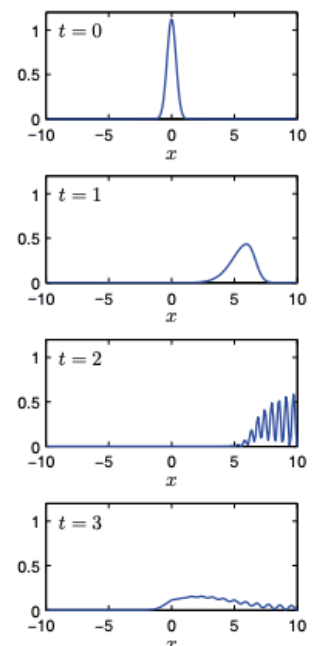
Figure 44: The probability density $|\psi(x)|^2$ of a particle that is initially more localized quickly spreads (Prob-

the sizes of L and ÏČ must compare in to use this initial condition. Run the script with $N = 200$ and watch the particle (wave packet) bounce back and forth in the well. Plot the real part of ψ as an animation to visualize the spatial oscillation of the wave packet, then plot an animation of $\psi * \psi$ so that you can visualize the probability distribution of the particle. Try switching the sign of p and see what happens.

(b) Verify by doing a numerical integral that $\psi(x,0)$ in the formula given above is properly normalized. Then run the script and check that the wave packet stays properly normalized, even though the wave function is bouncing and spreading within the well. If you are on a cell-edge grid you should do the integrals with NumPy's trapz rather than sum.

(c) Run the script and verify by numerical integration that the expectation value of the particle position

$$\langle x \rangle = \int_{-L}^{L} \psi^*(x,t)x\psi(x,t)dx \tag{121}$$

is correct for a bouncing particle. Plot $< x > (t)$ to see the bouncing behavior. Run long enough that the wave packet spreading modifies the bouncing to something more like a harmonic oscillator. (Note: you will only see bouncing-particle behavior until the wave packet spreads enough to start filling the entire well.)

(d) You may be annoyed that the particle spreads out so much in time. Try to fix this problem by narrowing the wave packet (decrease the value of σ) so the particle is more localized. Run the script and explain what you see in terms of quantum mechanics.

TUNNELING

Figure 46: he probability distribution $|\psi(x)|^2$ for a particle incident on a narrow potential barrier located just before $x = 10$ with $V_0 \rangle \langle E \rangle$. Part of the wave tunnels through the barrier and part interferes with itself as it is reflected.

Now we will allow the pulse to collide with a with a non-infinite potential barrier of height V_0 and width $\delta x = 0.02L$, and study what happens. Classically, the answer is simple: if the particle has a kinetic energy less than V_0 it will be unable to get over the barrier, but if its kinetic energy is greater than V_0 it will slow down as it passes over the barrier, then resume its normal speed in the region beyond the barrier. (Think of a ball rolling over a bump on a frictionless track.) Can the particle get past a barrier that is higher than its kinetic energy in quantum mechanics? The answer is yes, and this effect is called tunneling.

To see how the classical picture is modified in quantum mechanics we must first compute the energy of our pulse so we can compare it to the height of the barrier. The quantum mechanical formula for the expectation value of the energy is

$$\langle E \rangle = \int_{-\infty}^{\infty} \psi^* H \psi dx \tag{122}$$

where ψ is the complex conjugate of ψ and where

$$H\psi = -\frac{\bar{h}^2}{2m}\frac{\partial^2 \psi}{\partial x^2} + V(x)\psi(x) \tag{123}$$

In our case the initial wave function $\psi(x,0)$ given by Eq. (??) is essentially zero at the location of the potential barrier, so we may take $V(x) = 0$ in the integral when we compute the initial energy. Performing the integral, we find

$$\langle E \rangle = \frac{p^2}{2m} + \frac{\bar{h}^2}{4m\sigma^2} \tag{124}$$

Since this is a conservative system, the energy should remain constant throughout the propagation.

P8.3

(a) Modify your script from Problem 8.2 so that it uses a computing region hat goes from $-2L$ to $3L$ and a potential

$$V(x) = \begin{cases} 0 & \text{if } -2L < x < 0.98L \\ V_0 & \text{if } 0.98L \leq x \leq L \\ 0 & \text{if } L < x < 3L \\ +\infty & \text{otherwise} \end{cases} \tag{125}$$

so that we have a square potential hill $V(x) = V_0$ between $x = 0.98L$ and $x = L$ and $V = 0$ everywhere else in the well. Note: Since $V(x)$ was just zero in the last problem, this is the first time to check if your $V(x)$ terms in Crank-Nicolson are right. If you see strange behavior, you might want to look at these terms in your code.

(b) Run your script several times, varying the height V_0 from less than your pulse energy to more than your pulse energy. Overlay a plot of $V(x)/V_0$ on your plot of $|\psi|^2$ and look at the behavior of ψ in the barrier region. You should do several experiments with your code. (1) Try making the barrier height both higher than your initial energy and lower than your initial energy. Explain to your TA how the quantum behavior differs from classical behavior. You should find that even when the barrier is low enough that a classical particle could get over it, some particles still come back. (2) Experiment with the width of your barrier and see what its effect is on how many particles tunnel through. As part of this experiment, figure out a way to calculate what fraction of the particles make it through the barrier.

POISSON'S EQUATION: ITERATION METHODS

In three dimensions, Poisson's equation is given by

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} + \frac{\partial^2 V}{\partial z^2} = -\frac{\rho}{\varepsilon_0} \tag{126}$$

Poisson's equation is used to describe the electric potential in a region of space with charge density described by :[1] You can solve the full 3D equation using the technique we teach you in this lab, but we wonâĂŹt make you do that here. Instead, weâĂŹll focus on geometries that are infinitely long in the z-dimension with a constant cross-section in the $x - y$ plane. In these cases the z derivative goes to zero, and Poisson's equation reduces to

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = -\frac{\rho}{\varepsilon_0} \tag{127}$$

Note that by studying this equation we are also studying Laplace's equation (Poisson's equation with $\rho = 0$) and the steady state solutions to the diffusion equation in two dimensions ($\partial T / \partial t = 0$ in steady state).

FINITE DIFFERENCE FORM

The first step in numerically solving Poisson's equation is to define a 2-D spatial grid. For simplicity, we'll use a rectangular grid where the x coordinate is represented by N_x values x_j equally spaced with step size h_x , and the y coordinate is represented by N_y values y_k equally spaced with step size hy . This creates a rectangular grid with $N_x N_y$ grid points, just as we used in Lab 6 for the 2-d wave equation. We'll denote the potential on this grid using the notation $V(x_j, y_k) = V_{j,k}$

The second step is to write down the finite-difference approximation to the second derivatives in Poisson's equation to obtain a grid-based version of Poisson's equation. In our notation, Poisson's equation is the represented by

$$\frac{V_{j+1,k} - 2V_{j,k} + V_{j-1,k}}{h_x^2} + \frac{V_{j,k+1} - 2V_{j,k} + V_{j,k-1}}{h_y^2} = -\frac{\rho_{j,k}}{\varepsilon_0} \tag{128}$$

This set of equations can only be used at interior grid points because on the edges it reaches beyond the grid, but this is OK because the boundary conditions tell us what V is on the edges of the region.

Equation (**??**) plus the boundary conditions represent a set of linear equations for the unknowns $V_{j,k}$, so we could imagine just doing a big linear solve to find V all at once. Because this sounds so simple, let's explore it a little to see why we are not going to pursue this idea. The number of unknowns $V_{j,k}$ is $N_x N_y$, which for a 100 100 grid is 10,000 unknowns. So to do the solve directly we would have to be working with a 10,000 10,000 matrix, requiring 800 megabytes of RAM to store the matrix. Doing this big solve is possible for 2-dimensional problems like this because computers with much more memory than this are common. However, for larger grids the matrices can quickly get out of hand. Furthermore, if you wanted to do a 3-dimensional solve for a 100 100 100 grid, this would require $(10^4)^3 8 = 8 10^{12}$, or about 8 terabytes of memory. Computers like this are becoming possible, but this is still a tiny computational grid. So even though computers with large amounts of RAM are becoming common, people still use iteration methods like the ones we are about to describe.

ITERATION METHOD ON A SIMPLE EXAMPLE

Consider solving this equation:

$$x = e^{-x} \tag{129}$$

One method to solve this equation is to make a guess for the solution, call it x_0, and then iterate on the equation like this:

$$x_{n+1} = e^{-x_n} \tag{130}$$

For large values of n, we find that the process converges to the exact solution $\bar{x} = 0.567$. Let's do a little analysis to see why it works. Let $\bar{x}$ be the exact solution of this equation and suppose that at the nth iteration level we are close to the solution, only missing it by the small quantity δn like this: $x_n = \bar{x} + \delta n$. Let's substitute this approximate solution into Eq. (**??**) and expand using a Taylor series. Recall that the general form for a Taylor's series is

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \cdots + f^{(n)}(x)\frac{h^n}{n!} + \cdots \qquad (131)$$

When we substitute our approximate solution into Eq. (130) and expand around the exact solution $\bar{x}$ we get

$$f(x_{n+1} = e^{-\bar{x}-\delta_n} \approx e^{-\bar{x}} - \delta_n e^{-\bar{x}} + \cdots \qquad (132)$$

If we ignore the terms that are higher order in δ (represented by ÂůÂůÂů), then Eq. (**??**) shows that the error at the next iteration step is $\delta_{n+1} = e^{\bar{x}}\delta_n$. When we are close to the solution the error becomes smaller every iteration by the factor $e^{-\bar{x}}$. Since $\bar{x}$ is positive, $e^{-\bar{x}}$ is less than 1, and the algorithm converges. When iteration works it is not a miracleâĂŤit is just a consequence of having this expansion technique result in an error multiplier that is less than 1 in magnitude.

P9.1   Write a program to solve the equation $x = e^x$ by iteration and verify that it converges. Then try solving this same equation the other way round: $x = lnx$ and show that the algorithm doesn't converge. If you were to use the $\bar{x} + \delta$ analysis above for the logarithm equation, you'd find that the multiplier is bigger than one. Our time is short today, so we won't make you do this analysis now, but it is a good exercise if you want to look at it on your own time.

ITERATION METHOD ON POISSON'S EQUATION

Well, what does this have to do with Poisson's equation? If we solve the finitedifference version of Poisson's equation (Eq. (**??**)) for $V_{j,k}$, we find

$$f(x_{n+1} = e^{-\bar{x}-\delta_n} \approx e^{-\bar{x}} - \delta_n e^{-\bar{x}} + \cdots \qquad (133)$$

With the equation in this form we could just iterate over and over by doing the following.

- Choose an initial guess for the interior values of $V_{j,k}$.

- Use this initial guess to evaluate the right-hand side of Eq. (**??**)

- Replace our initial guess for $V_{j,k}$ by this right-hand side, and then repeat.

If all goes well, then after many iterations the left and right sides of this equation will agree and we will have a solution.
To see, let's notice that the iteration process indicated by Eq. (**??**) can be written in matrix form as

$$V_{n+1} = \mathbf{L}V_n + r \qquad (134)$$

where L is the matrix which, when multiplied into the vector V_n, produces the $V_{j,k}$ part of the right-hand side of Eq. (**??**) and r is the part that depends on the charge density $\rho_{j,k}$. (Don't worry about what L actually looks like; we are just going to apply general matrix theory ideas to it.) As in the exponential-equation example given above, let $\bar{V}$ be the exact solution vector and let δ_n be the error vector at the nth iteration. The iteration process on the error is, then,

$$\delta_{n+1} = \mathbf{L}\delta_n \qquad (135)$$

Now think about the eigenvectors and eigenvalues of the matrix L. If the matrix is well-behaved enough that its eigenvectors span the full solution vector space of size N_xN_y , then we can represent δn as a linear combination of these eigenvectors. This then invites us to think about what iteration does to each eigenvector. The answer, of course, is that it just multiplies each eigenvector by its eigenvalue. Hence, for iteration to work we need all of the eigenvalues of the matrix L to have magnitudes less than 1.
So we can now restate the conditions for this approach to work: Iteration on Eq. (**??**) converges if all of the eigenvalues of the matrix L on the right-hand side of Eq. (**??**) are less than 1 in magnitude. This statement is a theorem which can be proved if you are really good at linear algebra, and the entire iteration procedure described by Eq. (9.9) is known as Jacobi iteration. Unfortunately, even though all of the eigenvalues have magnitudes

Friedrich   Gauss
(1777−1855, German)

less than 1 there are lots of them that have magnitudes very close to 1, so the iteration takes forever to converge (the error only goes down by a tiny amount each iteration).

But Gauss and Seidel discovered that the process can be accelerated by making a very simple change in the process. Instead of only using old values of $V_{j,k}$ on the right-hand side of Eq. (**??**), they used values of $V_{j,k}$ as they became available during the iteration. (This means that the right side of (**??**) contains a mixture of V-values at the n and $n+1$ iteration levels.) This change, which is called GaussSeidel iteration is really simple to code; you just have a single array in which to store $V_{j,k}$ and you use new values as they become available. Here as a coded example of Guass-Seidel iteration for a rectangular region grounded on two sides, with the two other sides held at a potential:

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

# Make the grid
xmin = 0
xmax = 2
Nx = 80
x,hx = np.linspace(xmin,xmax,Nx,retstep = True)
hx2 = hx**2

ymin = 0
ymax = 2
Ny = 40
y,hy = np.linspace(ymin,ymax,Ny,retstep = True)
hy2 = hy**2
X,Y = np.meshgrid(x,y,indexing='ij')

# Initialize potential
V = 0.5*np.ones_like(X)

# Enforce boundary conditions
V[:,0] = 0

V[:,-1] = 0
V[0,:] = 1
V[-1,:] = 1

# Allow possibility of charge distribution
rho = np.zeros_like(X)

# Iterate
denom = 2/hx2 + 2/hy2
fig = plt.figure(1)
for n in range(200):
  # make plots every few steps
  if n % 10 == 0:
    plt.clf()
    ax = fig.gca(projection='3d')
    surf = ax.plot_surface(X,Y,V)
    ax.set_zlim(-0.1, 2)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.draw()
    plt.pause(0.1)

  # Iterate the solution
  for j in range(1,Nx-1):
    for k in range(1,Ny-1):
      V[j,k] = ( (V[j+1,k] + V[j-1,k])/hx2
        +(V[j,k+1] + V[j,k-1])/hy2
        +rho[j,k]) / denom
```

P9.2 Paste the code above into a program, run it, and watch the solution iterate. Study the code, especially the part in the loop. When you understand the code, call a TA over and explain it to them to pass off this part.

SUCCESSIVE OVER-RELAXATION

Gauss-Seidel iteration is not the best we can do, however. To understand the next improvement let's go back to the exponential example

$$x_{n+1} = e^{-x_n} \tag{136}$$

and change the iteration procedure in the following non-intuitive way:

$$x_{n+1} = \omega e^{-x_n} + (1-\omega)x_n \tag{137}$$

where ω is a number which is yet to be determined.

P9.3 Verify quickly on paper that even though Eq. (**??**) looks quite different from Eq. (**??**), it is still solved by $x = e^{-x}$. If we insert xn = xÂŕ + Ît'n into this new equation and expand as before, the error changes as we iterate according to the following

$$x_{n+1} = \omega e^{-x_n} + (1-\omega)x_n \tag{138}$$

Notice what would happen if we chose ω so that the factor in parentheses were zero: The equation says that we would find the correct answer in just one step! Of course, to choose ω this way we would have to know $\bar{x}$, but it is enough to know that this possibility exists at all. All we have to do then is numerically experiment with the value of ω and see if we can improve the convergence.

P9.4 Write a program that accepts a value of ω and runs the iteration in Eq. (**??**). Experiment with various values of ω until you find one that does the best job of accelerating the convergence of the iteration. You should find that the best ω is near 0.64, but it wonâĂŹt give convergence in one step. See if you can figure out why not.

Hint: Think about the approximations involved in obtaining Eq. (**??**). Specifically go back to the previous derivation and look for terms represented by dots.

As you can see from Eq. (**??**), this modified iteration procedure shifts the error multiplier to a value that converges better. So now we can see how to improve Gauss-Seidel: we just use an ω multiplier like this:

$$V_{n+1} = \omega(\mathbf{L}V_n + r) + (1-\omega)V_n \tag{139}$$

then play with ω until we achieve almost instantaneous convergence.

Sadly, this doesn't quite work. The problem is that in solving for $N_x N_y$ unknown values $V_{j,k}$ we don't have just one multiplier; we have one for each eigenvalue of the matrix. So if we shift one of the eigenvalues to zero, we might shift another one to a value with magnitude larger than 1 and the iteration will not converge at all. The best we can do is choose a value of ω that centers the entire range of eigenvalues symmetrically between âĹŠ1 and 1.

Using an ω multiplier to shift the eigenvalues is called Successive OverRelaxation, or SOR for short. Here it is written out so you can code it:

$$V_{j,k} = \omega \left(\frac{V_{j+1,k} + V_{j-1,k}}{h_x^2} + \frac{V_{j,k+1} + V_{j,k-1}}{h_y^2} + \frac{\rho_{j,k}}{\varepsilon_0} \right) / \left(\frac{2}{h_x^2} + \frac{2}{h_y^2} \right) + (1-\omega)V_{j,k} \tag{140}$$

And what value should we use for ω? The answer is that it depends on the values of N_x and N_y. In all cases ÏĹ should be between 1 and 2, with $\omega = 1.7$ being a typical value. Some wizards of linear algebra have shown that the best value of ω when the computing region is rectangular and the boundary values of V are fixed (Dirichlet boundary conditions) is given by

$$\omega = \frac{2}{1 + \sqrt{1-R^2}} \tag{141}$$

where

$$R = \frac{h_y^2 \cos(\pi/N_x) + h_x^2 \cos(\pi/N_y)}{h_x^2 + h_y^2} \tag{142}$$

These formulas usually give a reasonable estimate of the best ω to use. Note, however, that this value of ω was found for the case of a cell-edge grid with the potential specified at the edges. If you use a cell-centered grid with ghost points, and especially if you change to derivative boundary conditions, this value of ω won't be quite right. But there is still a best value of ω somewhere near the value given in Eq. (**??**) and you can find it by numerical experimentation.

Finally, we come to the question of when to stop iterating. It is tempting just to watch the values of $V_{j,k}$ and quit when the values stabilize at some level, like this for instance: quit when $\varepsilon = |V(j,k)_{n+1} - V(j,k)_n| < 10^{-6}$. You will see this error criterion sometimes used in books, but do not use it. We know of one person who published an incorrect result in a journal because this error criterion lied. We done't want to quit when the algorithm has quit changing V; we want to quit when Poisson's equation is satisfied. (Most of the time these are the same, but only looking at how V changes is a dangerous habit to acquire.) In addition, we want to use a relative (%) error criterion. This is easily done by setting a scale voltage V_{scale} which is on the order of the biggest voltage in the problem and then using for the error criterion

$$\varepsilon = \left| \frac{\text{Lhs} - \text{Rhs}}{V_{\text{scale}}} \right| \tag{143}$$

where Lhs is the left-hand side of Eq. (**??**) and Rhs is its right-hand side. Because this equation is just an algebraic rearrangement of our finite-difference approximation to Poisson's equation, ε can only be small when Poisson's equation is satisfied. (Note the use of absolute value; can you explain why it is important to use it? Also note that this error is to be computed at all of the interior grid points. Be sure to find the maximum error on the grid so that you only quit when the solution has converged throughout the grid.)

And what value should we choose for the error criterion so that we know when to quit? Well, our finite-difference approximation to the derivatives in Poisson's equation is already in error by a relative amount of about $1/(12N^2)$, where N is the smaller of N_x and N_y. There is no point in driving ε below this estimate. For more details, and for other ways of improving the algorithm, see Numerical Recipes, Chapter 19.

P9.5 Starting with the Gauss-Seidel example above, implement all of the improvements described above to write a full successive over-relaxation routine. Note that you will need to replace the *for* loop on the variable n into a while loop that is based on the error criterion. Also move the plot command out of the loop so it only executes once, after the solution has converged. (So that you don't have to wait for a bunch of plots to be drawn.) Run your code for a spatial region of size $L_x = 4$ in the x-dimension and $L_y = 2$ the y-dimension arranged as shown in Fig. 9.1. Use Nx = Ny = 30 and several different values of ω. Note that the boundary conditions on the potential $V(x,y)$ are $V(-L_x/2, y) = V(L_x/2, y) = 1$ and $V(x,0) = V(x, L_y) = 0$. Set the error criterion to 10^{-4}. Verify that the optimum value of ω given by Eq. (**??**) is the best one to use.

SOME DIFFERENT GEOMETRIES

P9.6

(a) Modify your code to model a rectangular pipe with $V(x = -2, y) = -1$, $V(x = 2, y) = 1$, and the $y = 0$ and $y = L_y$ edges held at $V = 0$.

(b) Modify your code from (a) so that the boundary condition at the $x = -L_x/2$ edge of the computation region is $\partial V / \partial x = 0$ and the boundary condition on the $y = L_y$ edge is $\partial V / \partial y = 0$. You can do this problem either by changing your grid and using ghost points or by using a quadratic extrapolation technique (see Eq. (**??**)). Both methods work fine, but note that you will need to enforce boundary conditions inside the main SOR loop now instead of just setting the values at the edges and then leaving them alone. You may discover that the script runs slower on this problem. See if you can make it run a little faster by experimenting with the value of ω that you use. Again, changing the boundary conditions can change the eigenvalues of the operator. (Remember that Eq. (9.16) only works for cell-edge grids with fixed-value boundary conditions, so it only gives a ballpark suggestion for this problem.)

(c) Study electrostatic shielding by going back to the boundary conditions of Problem 9.6(a), while grounding some points in the interior of the full computation region to build an approximation to a grounded cage. Allow some holes in your cage so you can see how fields leak in. First make a rectangular cage similar to Fig. 9.4, but then you can try other geometries. You will need to be creative about how you build your cage and about how you make SOR leave your cage points grounded as it iterates. One thing that won't work is to let SOR change all the potentials, then set the cage points back to $V = 0$ before doing the next iteration. This takes forever to converge. It is much better to set them to zero and force SOR to never change them. An easy way to do this is to use a cell-edge grid with a mask. A mask is an array that you build that is the same size as V, initially defined to be full of ones like this

```
mask = np.ones_like(V)
```

Then you go through and set the elements of mask that you don't want SOR to change to have a value of zero. (We'll let you figure out the logic to do this for the cage.) Once you have your mask built, you add an if statement to our code so that the SOR stuff inside the j and k for loops only changes a given point and updates the error if $\text{mask}(j,k)$ is one. This logic assumes you have to set the values of V for these points before the for loop execute, just like the boundary conditions. Using this technique you can calcu-
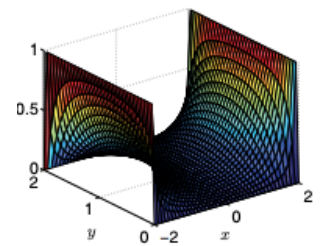
Figure 47: The electrostatic potential $V(x,y)$ with two sides grounded and two sides at constant potential.
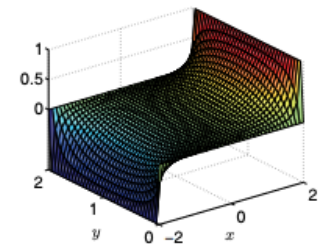
Figure 48: The potential $V(x,y)$ rom Problem 9.6(a).
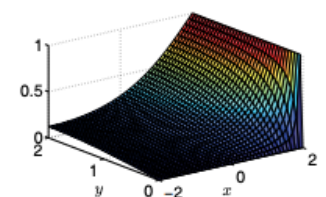
Figure 49: The potential $V(x,y)$ with zero-derivative boundary conditions on two sides (Problem 9.6(b).)

late the potential for quite complicated shapes just by changing the mask array.
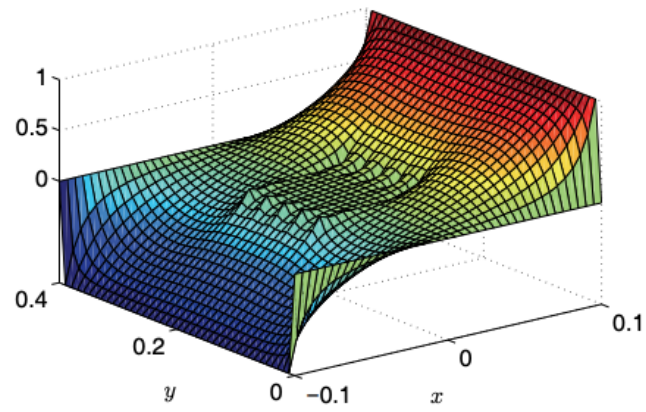
Figure 50: The potential $V(x,y)$ for an electrostatic "cage" formed by grounding some interior points. (Problem 9.6(c).)

THE CONTINUITY EQUATION

So far we have studied the numerical solution of partial differential equations one at a time, but in many interesting situations the problem to be solved involves coupled systems of differential equations. A "simple" example of such a system are the three coupled one-dimensional equations of gas dynamics. These are the equations of acoustics in a long tube with mass density $\rho(x,t)$, pressure $p(x,t)$, and gas velocity $v(x,t)$ as the dynamic variables. In the next lab we will tackle the hard problem of simultaneously advancing ρ, T, and v in time and space, which will require three equations. But for now we will just practice on the continuity equation to develop the tools we need to do the full problem.

THE CONTINUITY EQUATION

The equation that enforces conservation of mass in acoustics is called the continuity equation:

$$\varepsilon = \left| \frac{\text{Lhs} - \text{Rhs}}{V_{\text{scale}}} \right| \tag{144}$$

This equation says that as the gas particles are moved by the flow velocity $v(x,t)$, the density $\rho(x,t)$ is carried along with the flow and can be compressed or rarefied, but mass is not created or destroyed in this process.

Boundary conditions for the continuity equation are a bit different than we've encountered to this point. This is a convection equation, meaning that if you stand at a point in the flow, the solution at your location arrives (is convected to you) from further "upwind."This has a strong effect on the boundary conditions. Suppose, for instance, that the flow field $v(x)$ is always positive, meaning that the wind is blowing to the right. At the left-hand boundary it makes sense to specify ρ because somehow you might arrange to feed density in at that point so that it can be convected across the grid. But at the right boundary it makes no sense at all to specify a boundary condition because when the solution arrives there we just want to let the wind blow it away. (An exception to this rule occurs if $v = 0$ at the boundary. In this case there is no wind to blow the solution from anywhere and it would be appropriate to specify a boundary condition.) We'll try several approaches to represent these boundary conditions.

P10.1 Let's start with something really simple and inaccurate just to see what can go wrong. If we use a nicely centered difference in x and an inaccurate forward difference in t, we find

$$\frac{\rho_j^{n+1} - \rho_j^n}{\tau} + \frac{1}{2h} \left(\rho_{j+1}^n v_{j+1} - \rho_{j-1}^n v_{j-1} \right) = 0 \tag{145}$$

Solve this equation for ρ_j^{n+1} and use it in a time-advancing script like the one you built to do the wave equation in Lab 5. Use a cell-center grid with ghost points (use about 400 grid points). For initial conditions use

$$\rho(x,0) = 1 + e^{-200(x/L - 1/2)^2} \tag{146}$$

with $x \in [0,L], L = 10$, and

$$v(x) = v_0 \tag{147}$$

with $v_0 = 1$. At the left end use $\rho(0,t) = 1$ and at the right end try the following two things:

(i) Set a boundary condition: $\rho(L,t) = 1$.

(ii) Just let mass leave by using linear extrapolation:

$$\rho(L,t) = 2\rho(L - h,t) - \rho(L - 2h,t) \quad \text{or} \quad \rho_{N+1} = 2\rho_N - \rho_{N-1} \tag{148}$$

Run this algorithm with these two boundary conditions enough times, and with small enough time steps, that you become convinced that $\rho(L,t) = 1$ is wrong and that the entire algorithm is worthless because it is unstable.

As you might guess from the previous problem, the diffusion equation's simple appearance can be deceiving; it is one of the most difficult equations to solve numerically in

all of computational physics because stable methods tend to be inaccurate and accurate methods tend either to be unstable, or non-conservative (as time runs mass spontaneously disappears), or unphysical (mass density and/or pressure become negative.)

Let's try another method, known as the Lax-Wendroff method. The idea of the Lax-Wendroff algorithm is to use a Taylor series in time to obtain a second-order accurate method. Taylor expanding the density function in time gives us

$$\rho(x, t + \tau) = \rho(x, t) + \tau \frac{\partial \rho}{\partial t} + \frac{\tau^2}{2} \frac{\partial^2 \rho}{\partial t^2} \tag{149}$$

We can write the continuity equation as

$$\frac{\partial \rho}{\partial t} = -\frac{\partial}{\partial x}(\rho v) \tag{150}$$

Substituting into our Taylor expansion then gives

$$\rho(x, t + \tau) = \rho(x, t) - \tau \frac{\partial}{\partial x}(\rho v) + \frac{\tau^2}{2} \frac{\partial}{\partial t} \left(-\frac{\partial}{\partial x}(\rho v) \right) \tag{151}$$

If we reverse the order of the derivatives in the last term and assume that v is not a function of time, we can use Eq. (**??**) again to obtain.

$$\rho(x, t + \tau) = \rho(x, t) - \tau \frac{\partial \rho v}{\partial x} + \frac{\tau^2}{2} \frac{\partial}{\partial x} \left(v \frac{\partial \rho v}{\partial x} \right) \tag{152}$$

Finally, we subtract (x, t) from both sides and divide by τ to obtain

$$\frac{\rho(x, t + \tau) - \rho(x, t)}{\tau} = -\frac{\partial \rho v}{\partial x} + \frac{\tau}{2} \frac{\partial}{\partial x} \left(v \frac{\partial \rho v}{\partial x} \right) \tag{153}$$

If we interpret the left-hand side as a time derivative, the second term on the right looks essentially like the diffusion equation. Since the equation we are solving is pure convection, the appearance of diffusion is not good news, but at least this algorithm is better than the horrible one in 10.1. Notice also that the diffusion coefficient in Eq. (**??**) is proportional to τ (stare at it until you can see that this is true), so if small time steps are being used diffusion won't hurt us too much.

P10.2 Finite difference the expression in Eq. (153) assuming that $v(x) = v0 = const$, to find the Lax-Wendroff algorithm:

$$\rho_j^{n+1} = \rho_j^n - \frac{v_0 \tau}{2h} \left(\rho_{j+1}^n - \rho_{j-1}^n \right) + \frac{v_0^2 \tau^2}{2h^2} \left(\rho_{j+1}^n - 2\rho_j^n + \rho_{j-1}^n \right) \tag{154}$$

Change your script from 10.1 to use the Lax-Wendroff algorithm. Again, use a cell-center grid with ghost points and about 400 grid points. Also use the same initial condition as in Problem 10.1 and use the extrapolated boundary condition that just lets the pulse leave.

Show that Lax-Wendroff works pretty well unless the time step exceeds a Courant condition. Also show that it has the problem that the peak density slowly decreases as the density bump moves across the grid. (To see this use a relatively coarse grid and a time step just below the stability constraint.

Warning: do not run with $\tau = h/v_0$. If you do you will conclude that this algorithm is perfect, which is only true for this one choice of time step.) This problem is caused by the diffusive term in the algorithm, but since this diffusive term is the reason that this algorithm is not unstable like the one in 10.1, we suppose we should be grateful.

CRANK-NICOLSON AGAIN

Finally, let's try an implicit method, Crank-Nicolson in fact. As a reminder, the continuity equation is

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}(\rho v) = 0 \tag{155}$$

We can't solve this equation directly because it has two unknowns (ρ and v). But if we assume that v is known, then it is possible to solve the equation using CrankNicolson. As usual for Crank-Nicolson, we forward difference in time and center difference in space to find

$$\frac{\rho_j^{n+1} - \rho_j^n}{\tau} = -\frac{v_{j+1}^n \rho_{j+1}^n - v_{j-1}^n \rho_{j-1}^n}{2h} \tag{156}$$

Peter Lax (b. 1926, American) Lax was the PhD advisor for Burton Wendroff.

Then we use time averaging to put the right side of the equation at the same time level as the left (i.e. at the $n + 1/2$ time level):

$$\rho_j^{n+1} - \rho_j^n = C_1 \left(\rho_{j+1}^n + \rho_{j+1}^{n+1} \right) - C_2 \left(\rho_{j-1}^n + \rho_{j-1}^{n+1} \right) \tag{157}$$

where

$$C_1 = -\frac{\tau}{8h} \left(v_{j+1}^n + v_{j+1}^{n+1} \right) \tag{158}$$

$$C_2 = -\frac{\tau}{8h} \left(v_{j-1}^n + v_{j-1}^{n+1} \right) \tag{159}$$

Then we put the ρ^{n+1} terms on the left and the ρ^n terms on the right:

$$C_2 \rho_{j-1}^{n+1} + \rho_j^{n+1} - C_1 \rho_{j+1}^{n+1} = -C_2 \rho_{j-1}^n + \rho_j^n + C_1 \rho_{j+1}^n \tag{160}$$

Then we write these equations along with the boundary conditions in matrix form

$$\mathbf{A}\rho^{n+1} = \mathbf{B}\rho^n \tag{161}$$

which we solve using linear algebra techniques. For this algorithm to calculate ρ^{n+1}, we need to feed it values for ρ^n, v^n, and v^{n+1}. For now, let's side-step this issue by assuming that $v(x,t)$ is known, and a constant in time. In the next lab we'll worry about advancing the velocity solution in parallel with the density.

P10.3

(a) Write a program that implements this algorithm, perhaps starting from one of your programs from the Schrödinger equation lab. Work out how to implement the boundary conditions, $\rho(0,t) = 1$ and $\rho(L,t)$ is just allowed to leave, by properly defining the top and bottom rows of the matrices $\mathbf{A}$ and $\mathbf{B}$. This involves multiplying $\mathbf{B}\rho^n$ to find an r-vector as you have done before.

(b) Implement this algorithm with a constant convection velocity $v = v_0$ and show that the algorithm conserves amplitude to very high precision and does not widen due to diffusion. These two properties make this algorithm a good one as long as shock waves don't develop.

(c) Now use a convection velocity that varies with x :

$$v(x) = 1.2 - x/L \tag{162}$$

This velocity slows down as the flow moves to the right, which means that the gas in the back is moving faster than the gas in the front, causing compression and an increase in density. You should see the slowing down of the pulse and the increase in density in your numerical solution.

(d) Go back to a constant convection velocity $v = v_0$ and explore the way this algorithm behaves when we have a shock wave (discontinuous density) by using as the initial condition

$$\rho(x,0) = \begin{cases} 1.0 & \text{if } 0 \le x \le L/2 \\ 0 & \text{otherwise} \end{cases} \tag{163}$$

The true solution of this problem just convects the step to the right; you will find that Crank-Nicolson fails at this seemingly simple task.

(e) For comparison, try the same step-function initial condition in your Lax-Wendroff script from Problem 10.2.

Figure 51: A pulse is convected across a region in which the convection velocity $v(x)$ is constant (Problem 10.3(b)).
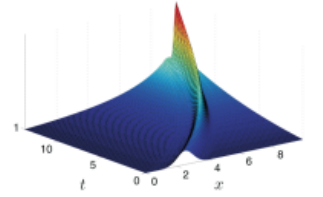
Figure 52: A pulse is convected across a region in which the convection velocity $v(x)$ is decreasing. Note that the pulse narrows and grows, conserving mass. (Problem 10.3(c))

Our Crank-Nicolson algorithm is both stable and conservative, but it only works well if the solution doesn't become too steep. This is a severe limitation, since we are talking about gas dynamics here and shock waves routinely show up as solutions in gas dynamics. Numerical methods that properly handle shocks are much more involved than the ones we will show you here.
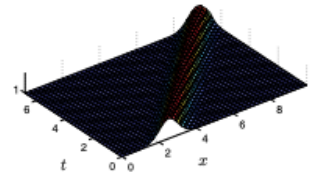
ONE DIMENSIONAL GAS DYNAMICS

SIMULTANEOUSLY ADVANCING ρ, T, AND v

Now we are going to use the implicit algorithm that we developed in the previous lab as a tool to solve the three nonlinear coupled partial differential equations of one-dimensional gas dynamics. These are the equations of one-dimensional sound waves in a long tube pointed in the x-direction, assuming that the tube is wide enough that friction with the walls doesn't matter.

The three equations we need to advance are the continuity equation

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}(\rho v) = 0 \tag{164}$$

the conservation of energy

$$\frac{\partial T}{\partial t} + v\frac{\partial T}{\partial x} + (\gamma - 1)T\frac{\partial v}{\partial x} = \frac{(\gamma - 1)M\kappa}{k_B}\frac{1}{\rho}\frac{\partial^2 T}{\partial x^2} \tag{165}$$

and Newton's second law

$$\frac{\partial v}{\partial t} + v\frac{\partial v}{\partial x} = -\frac{1}{\rho}\frac{\partial P}{\partial x} + \frac{4\mu}{3\rho}\frac{\partial^2 v}{\partial x^2} \tag{166}$$

Here $\rho(x,t)$ is the density of the gas, $v(x,t)$ is the velocity of the gas, and $T(x,t)$ is the temperature of the gas. The pressure P is given by the ideal gas law

$$P = \frac{k_B}{M}\rho T \tag{167}$$

Because of the nonlinearity of these equations and the fact that they are coupled we are not going to be able to write down a simple algorithm that will advance ρ, T, and v in time. But if we are creative we can combine simple methods that work for each equation separately into a stable and accurate algorithm for the entire set. We are going to show you one way to do it, but the computational physics literature is full of other ways, including methods that handle shock waves. This is still a very active and evolving area of research, especially for problems in 2 and 3 dimensions. Let's try a predictor-corrector approach similar to second-order Runge-Kutta (which you learned about back in 330) by first taking an approximate step in time of length τ to obtain predicted values for our variables one time step in the future. We'll refer to these first-order predictions for the future values as $\tilde{\rho}^{n+1}$, $\tilde{T}^{n+1}$, and $\tilde{v}^{n+1}$. In the predictor step we will treat v as constant in time in Eq. (164) to predict $\tilde{\rho}^{n+1}$. Then we'll use $\tilde{\rho}^{n+1}$ to help us calculate $\tilde{T}^{n+1}$ using Eq. (165), while still treating v as fixed in time. Once these predicted values are obtained we can use them with Eq. (166) to obtain a predicted $\tilde{v}$. With all of these predicted future values of our variables in hand, we do another round of Crank-Nicolson on all three equations to step the solution forward in time. We can represent this procedure schematically as follows:

Step 1 Use the old velocity v^n as input for Eqn. (168) $\rightarrow$ Solve for the predicted density $\tilde{\rho}$.

Step 2 Use v^n and $\tilde{\rho}$ as inputs for Eqn. (165) $\rightarrow$ Solve for the predicted temperature $\tilde{T}$.

Step 3 Use $\tilde{\rho}$ and $\tilde{T}$ as inputs for Eqn. (166) $\rightarrow$ Solve for the predicted velocity $\tilde{v}$.

Step 4 Use $\tilde{v}$ as input for Eqn. (168) $\rightarrow$ Solve for the new density ρ^{n+1}.

Step 5 Use $\tilde{v}$ and ρ^{n+1} as inputs for Eqn. (165) $\rightarrow$ Solve for the new temperature T^{n+1}.

Step 6 Use ρ^{n+1} and T^{n+1} as inputs for Eqn. (166) $\rightarrow$ Solve for the new velocity v^{n+1}.

This procedure probably seems a bit nebulous at this point, so let's go through it in more detail. First we'll derive the Crank-Nicolson algorithms for our three equations, then we'll show how to use these algorithms to solve the system using the predictor-corrector method.

CONTINUITY EQUATION

Conservation of mass is governed by the continuity equation

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}(\rho v) = 0 \tag{168}$$

We can't solve this equation directly because it has two unknowns (ρ and v). But if we assume that v is known as we did in the last lab, then it is possible to solve the equation using Crank-Nicolson, as we did in the last lab. As usual for Crank-Nicolson, we forward difference in time and center difference in space to find

$$\frac{\rho_j^{n+1} - \rho_j^n}{\tau} = -\frac{v_{j+1}^n \rho_{j+1}^n - v_{j-1}^n \rho_{j-1}^n}{2h} \tag{169}$$

Then we use time averaging to put the right side of the equation at the same time level as the left (i.e. at the $n + 1/2$ time level):

$$\rho_j^{n+1} - \rho_j^n = C_1 \left(\rho_{j+1}^n + \rho_{j+1}^{n+1} \right) - C_2 \left(\rho_{j-1}^n + \rho_{j-1}^{n+1} \right) \tag{170}$$

where

$$C_1 = -\frac{\tau}{8h} \left(v_{j+1}^n + v_{j+1}^{n+1} \right) \tag{171}$$

$$C_2 = -\frac{\tau}{8h} \left(v_{j-1}^n + v_{j-1}^{n+1} \right) \tag{172}$$

Then we put the ρ^{n+1} terms on the left and the ρ^n terms on the right:

$$C_2 \rho_{j-1}^{n+1} + \rho_j^{n+1} - C_1 \rho_{j+1}^{n+1} = -C_2 \rho_{j-1}^n + \rho_j^n + C_1 \rho_{j+1}^n \tag{173}$$

Then we write these equations along with the boundary conditions in matrix form

$$\mathbf{A}\rho^{n+1} = \mathbf{B}\rho^n \tag{174}$$

which we solve using linear algebra techniques. For the algorithm represented by Eq. (**??**) to calculate ρ^{n+1}, we need to feed it values for ρ^n, v^n, and v^{n+1}. Since the inputs for these variables will be different in the predictor and the corrector steps, we need to invent some notation. We'll refer to this Crank-Nicolson algorithm for stepping forward to find ρ^{n+1} using the notation $S_\rho \left(\rho^n, v^n, v^{n+1} \right)$ so the variables the algorithm needs as inputs are explicitly shown.

CONSERVATION OF ENERGY

The temperature of a gas is a macroscopic manifestation of the energy of the thermal motions of the gas molecules. The equation that enforces conservation of energy for our system is

$$\frac{\partial T}{\partial t} + v\frac{\partial T}{\partial x} = -(\gamma - 1)T\frac{\partial v}{\partial x} + D_T \frac{\partial^2 T}{\partial x^2} \tag{175}$$

where γ is the ratio of specific heats in the gas: $\gamma = C_p/C_v$. This equation says that as the gas is moved along with the flow and squeezed or stretched, the energy is convected along with the flow and the pressure goes up and down adiabatically (that's why γ is in there). It also says that thermal energy diffuses due to thermal conduction. Thermal diffusion is governed by the diffusion-like term containing the thermal diffusion coefficient D_T given in a gas by

$$D_T = \frac{(\gamma - 1)M\kappa}{k_B \rho} \tag{176}$$

where κ is the thermal conductivity, M is the mass of a molecule of the gas, and where k_B is Boltzmann's constant.

It is probably easier to conceptualize pressure waves rather than temperature waves. The ideal gas law gives us a way to calculate the pressure, given a density ρ and a temperature T, so we'll use the ideal gas law $P = nk_B T$ (where n is the number of particles per unit volume) to calculate pressure P once the density ρ and temperature T are known via

$$P = \frac{k_B}{M}\rho T \tag{177}$$

To find a predicted value for T one step in the future, we forward difference the time derivative and center difference the space derivatives to find

$$\frac{T_j^{n+1} - T_j^n}{\tau} = -v_j^n \frac{T_{j+1}^n - T_{j-1}^n}{2h} - (\gamma - 1)T_j^n \frac{v_{j+1}^n - v_{j-1}^n}{2h} + F \frac{1}{\rho_j^n} \frac{T_{j+1}^n - 2T_j^n + T_{j-1}^n}{h^2} \quad (178)$$

where

$$F = \frac{(\gamma - 1)M\kappa}{k_B} \quad (179)$$

We then rearrange Eq. (**??**) into a form that makes the upcoming algebra (and coding) more readable:

$$\frac{T_j^{n+1} - T_j^n}{\tau} = T_{j-1}^n D_1 + T_j^n D_2 + T_{j+1}^n D_3 \quad (180)$$

where

$$D_1 = \frac{v_j^n}{2h} + \frac{F}{\rho_j^n h^2} \quad (181)$$

$$D_2 = -(\gamma - 1)\frac{v_{j+1}^n - v_{j-1}^n}{2h} - \frac{2F}{\rho_j^n h^2} \quad (182)$$

$$D_3 = -\frac{v_j^n}{2h} + \frac{F}{\rho_j^n h^2} \quad (183)$$

P11.1 Finish deriving the Crank-Nicolson algorithm for T^{n+1} by putting the right-hand side of Eq. (**??**) at the $n + 1/2$ time level. This means replacing T^n terms with $(T^n + T^{n+1})/2$ in Eq. (**??**) and making the replacements $\rho^n \Rightarrow (\rho^n + \rho^{n+1})/2$ and $v^n \Rightarrow (v^n + v^{n+1})/2$ in D_1, D_2, and D_3. Then put your system of equations in the form

$$\mathbf{A}T^{n+1} = \mathbf{B}T^n$$

and write out the coefficients in the **A** and **B** matrices so we can code them later. When you are finished with Problem 11.1 you will have an algorithm for stepping T forward in time. We'll refer to this algorithm as $S_T\left(T^n, v^n, v^{n+1}, \rho^n, \rho^{n+1}\right)$, so we explicitly see the variables that are required as inputs.

NEWTON'S SECOND LAW

Finally, let's consider Newton's second law for this system in a form analogous to $a = F/m$:

$$\frac{\partial v}{\partial t} + v\frac{\partial v}{\partial x} = -\frac{1}{\rho}\frac{\partial P}{\partial x} + \frac{4\mu}{3\rho}\frac{\partial^2 v}{\partial x^2} \quad (184)$$

You should recognize the first term dv/dt as acceleration, and we'll discuss the origin of the other acceleration term in a minute. The first term on the right is the pressure force that pushes fluid from high pressure toward low pressure, with the pressure P given by the ideal gas law in Eq. (177). The second term on the right represents the force of internal friction called viscosity, and the parameter μ is referred to as the coefficient of viscosity. (Tar has high viscosity, water has medium viscosity, and air has almost none.) The $1/\rho$ factor in the force terms represents the mass m in $a = F/m$ (but of course we are working with mass density ρ here). You may be unconvinced that the left side of Eq. (184) is acceleration. To become more convinced, let's think about this situation more carefully. Newton's second law does not apply directly to a location in space where there is a moving fluid. Newton's second law is for particles that are moving through space, not for a location in space that is sitting still with fluid moving through it. This distinction is subtle, but important. Think, for instance, about a steady stream of honey falling out of a honey bear held over a warm piece of toast. If you followed a piece of honey along its journey from the spout down to the bread you would experience acceleration. But if you watched a piece of the stream at a specific location (say, 10 cm above the bread) you would see that the velocity of this part of the stream is constant in time: $\partial v/\partial t = 0$. This is a strong hint that there is more to describing the acceleration of fluids through a region of space than just the local $\partial v/\partial t$ for a given location. You also need to compare the local velocity at your chosen point to the velocities at nearby points in space, which is what the $v\partial v/\partial x$ term does in the left side of Eq. (184). Notice that Eq. (**??**) has a nonlinear term

on the left: $v(\partial v/\partial x)$. There is no way to directly represent this nonlinear term using a linear matrix form like $\mathbf{A}v^{n+1} = \mathbf{B}v^n$, so we'll have to make an approximation. We'll assume that the leading v in the nonlinear term is somehow known and designate it as $\bar{v}$. (We'll deal with finding something to use for $\bar{v}$ later.) With a forward time derivative and a centered space derivative, we have

$$\frac{v_j^{n+1} - v_j^n}{\tau} = -\bar{v}_j^n \left(\frac{v_{j+1}^n - v_{j-1}^n}{2h} \right) - \frac{k_B}{M\rho_j^n} \left(\frac{\rho_{j+1}^n T_{j+1}^n - \rho_{j-1}^n T_{j-1}^n}{2h} \right) + \frac{4\mu}{3\rho_j^n} \left(\frac{v_{j+1}^n - 2v_j^n + v_{j-1}^n}{h^2} \right) \tag{185}$$

Again, we'll rewrite the equations with named groups of expressions that don't depend on v so that our algebra is manageable:

$$\frac{v_j^{n+1} - v_j^n}{\tau} = E_0 + v_{j-1}^n E_1 + v_j^n E_2 + v_{j+1}^n E_3 \tag{186}$$

where

$$E_0 = -\frac{k_B}{M\rho_j^n} \left(\frac{\rho_{j+1}^n T_{j+1}^n - \rho_{j-1}^n T_{j-1}^n}{2h} \right) \tag{187}$$

$$E_1 = \frac{\bar{v}_j^n}{2h} + \frac{4\mu}{3\rho_j^n h^2} \tag{188}$$

$$E_2 = -\frac{8\mu}{3\rho_j^n h^2} \tag{189}$$

$$E_3 = -\frac{\bar{v}_j^n}{2h} + \frac{4\mu}{3\rho_j^n h^2} \tag{190}$$

P11.2   Finish deriving the Crank-Nicolson algorithm for v by making the replacements $v^n \Rightarrow \left(v^{n+1} + v^n\right)/2$ the right-hand side of Eq. (**??**) and $\rho^n \Rightarrow \left(\rho^n + \tilde{\rho}^{n+1}\right)/2, T^n \Rightarrow \left(T^n + \tilde{T}^{n+1}\right)/2$, and $\bar{v}^n \Rightarrow \left(\bar{v}^n + v^{n+1}\right)/2$ in E_0, E_1, E_2, and E_3. Show that your system of equations needs to be in the form

$$\mathbf{A}v^{n+1} = \mathbf{B}v^n + E_0$$

where E_0 is a column vector. Write out the coefficients in the $\mathbf{A}$ and $\mathbf{B}$ matrices so you can code them later.

We'll refer to this v-stepping algorithm as $S_v\left(v^n, \bar{v}^n, \bar{v}^{n+1}, \rho^n, \rho^{n+1}, T^n, T^{n+1}\right)$, where, as usual, we explicitly show the variables that are required as inputs.

WAVES IN A CLOSED TUBE

Now that you have algorithms for all three equations, we can restate the predictorcorrector algorithm using our newly-developed notation. Predictor Step: First we predict ρ^{n+1} while treating v as a constant:

$$\tilde{\rho}^{n+1} = S_\rho\left(\rho^n, v^n, v^{n+1} = v^n\right)$$

Then we predict T^{n+1} using $\tilde{\rho}^{n+1}$, still treating v as a constant

$$\tilde{T}^{n+1} = S_T\left(T^n, v^n, v^{n+1} = v^n, \rho^n, \rho^{n+1} = \tilde{\rho}^{n+1}\right)$$

Then we predict v^{n+1} using $\tilde{\rho}^{n+1}$ and $\tilde{T}^{n+1}$, while treating $\bar{v}$ from the nonlinear term as a constant equal to the current v

$$\tilde{v}^{n+1} = S_v\left(v^n, \bar{v}^n = v^n, \bar{v}^{n+1} = v^n, \rho^n, \rho^{n+1} = \tilde{\rho}^{n+1}, T^n, \tilde{T}^{n+1}\right)$$

Corrector Step: Now that we have predicted values for each variable, we step ρ forward using

$$\rho^{n+1} = S_\rho\left(\rho^n, v^n, v^{n+1} = \tilde{v}^n\right)$$

Then we step T using

$$T^{n+1} = S_T\left(T^n, v^n, v^{n+1} = \tilde{v}^n, \rho^n, \rho^{n+1}\right)$$

And finally, we step v forward using

$$v^{n+1} = S_v\left(v^n, \tilde{v}^n = v^n, \tilde{v}^{n+1} = \tilde{v}^n, \rho^n, \rho^{n+1}, T^n, T^{n+1}\right)$$

Now let's put this algorithm into a script and use it to model waves in a tube of length $L = 10$ m with closed ends through which there is no flow of heat. For disturbances in air at sea level at 20°C we have temperature $T = 293$ K, mass density $\rho = 1.3$ kg/m³, adiabatic exponent $\gamma = 1.4$, coefficient of viscosity $\mu = 1.82 \times 10^{-5}$ kg/(m · s), and coefficient of thermal conductivity $\kappa = 0.024$ J/(m · s · K). Boltzmann's constant is $k_B = 1.38 \times 10^{-23}$ J/K and the mass of the molecules of the gas is $M = 29 \times 1.67 \times 10^{-27}$ kg for air.

P11.3

(a) As you might guess, debugging the algorithm that we just developed takes a while to debug because there are so many steps and so many terms to get typed accurately. (When we wrote the solution code, it took over an hour to track down two minus sign errors and a factor of two error.) We'd rather have you use the algorithm than beat your head on the wall debugging it, so below is the code that implements the algorithm. Go to the class web site now and download these files. Study them and make sure you understand how they work.

(b) The one thing we haven't included in the code is the boundary conditions. The ends are insulating, so we have

$$\partial T / \partial x = 0 \tag{191}$$

at both ends. Because the wall ends are fixed and the gas can't pass through these walls, the boundary conditions on the velocity are

$$v(0, t) = v(L, t) = 0 \tag{192}$$

Use this fact to obtain the following differential boundary condition on the density at the ends of the tube:

$$\frac{\partial \rho}{\partial t} + \rho \frac{\partial v}{\partial x} = 0 \quad \text{at } x = 0 \text{ and } x = L \tag{193}$$

This condition simply says that the density at the ends goes up and down in obedience to the compression or rarefaction produced by the divergence of the velocity. Write down the finite difference form for all three of these boundary conditions. Make sure they are properly centered in time and space for a cell-center grid with ghost points. Then code these boundary conditions in the proper spots in the code.

P11.4

(a) Test the script by making sure that small disturbances travel at the sound speed $c = \sqrt{\frac{\gamma k_B T}{M}}$. To do this set T and ρ to their atmospheric values and set the velocity to

$$\frac{\partial \rho}{\partial t} + \rho \frac{\partial v}{\partial x} = 0 \quad \text{at } x = 0 \text{ and } x = L \tag{194}$$

with $v_0 = c/100$. If you look carefully at the deviation of the density from its atmospheric value you should see two oppositely propagating signals. Verify that they travel at the speed of sound.

(b) The predictor-corrector algorithm is not as stable as plain CrankNicolson. Vary the time step and find where it begins to be unstable. Then change N and see how the stability threshold changes. Come up with an equation that estimates a limit on the step size in terms of h.

(c) Remove the effects of viscosity and thermal conduction by setting $\mu = 0$ and $\kappa = 0$. Increase the value of v_0 to $c/10$ and beyond and watch how the pulses develop. You should see the wave pulses develop steep leading edges and longer trailing edges; you are watching a shock wave develop. But if you wait long enough you will see your shock wave develop ugly wiggles; these are caused by Crank-Nicolson's failure to properly deal with shock waves.

(d) Repeat part (c) with non-zero μ and κ and watch thermal conduction and viscosity widen the shock and prevent wiggles. Try artificially large values of μ and κ as well as their actual atmospheric values.

```
1  import Lab11Funcs as S
2  import matplotlib.pyplot as plt
3  import numpy as np
4
5  # System Parameters
6  L = 10.0 # Length of tube
7  T0 = 293. # Ambient temperature
8  rho0 = 1.3 # static density (sea level)
9
10 # speed of sound
11 c = np.sqrt(S.gamma * S.kB * T0 / S.M)
12
13 # cell-center grid with ghost points
14 N = 100
15 h = L/N
16 x = np.linspace(-.5*h,L+.5*h,N+2)
17
18 # initial distributions
19 rho = rho0 * np.ones_like(x)
20 T = T0 * np.ones_like(x)
21 v = np.exp(-200*(x/L-0.5)**2) * c/100
22
23 tau = 1e-4
24 tfinal = 0.1
25 t = np.arange(0,tfinal,tau)
26
27 skip = 5 #input(' Steps to skip between plots - ')
28
29 for n in range(len(t)):
30
31    # Plot the current values before stepping
32    if n % skip == 0:
33      plt.clf()
34      plt.subplot(3,1,1)
35      plt.plot(x,rho)
36      plt.ylabel('rho')
37      plt.ylim(1.28, 1.32)
38      plt.title('time={:1.3f}'.format(t[n]))
39      plt.subplot(3,1,2)
40      plt.plot(x,T)
41      plt.ylabel('T')
42      plt.ylim(292,294)
43      plt.subplot(3,1,3)
44      plt.plot(x,v)
45      plt.ylabel('v')
46      plt.ylim(-4,4)
47      plt.xlabel('x')
48      plt.pause(0.1)
49
50 # 1. Predictor step for rho
51 rhop = S.Srho(rho,v,v,tau,h)
52
53 # 2. Predictor step for T
54 Tp = S.ST(T,v,v,rho,rhop,tau,h)
55
56 # 3. Predictor step for v
57 vp = S.Sv(v,v,v,rho,rhop,T,Tp,tau,h)
58
59 # 4. Corrector step for rho
60 rhop = S.Srho(rho,v,vp,tau,h)
61
62 # 5. Corrector step for T
63 Tp = S.ST(T,v,vp,rho,rhop,tau,h)
64
65 # 6. Corrector step for v
66 v = S.Sv(v,v,vp,rho,rhop,T,Tp,tau,h)
67
68 # Now put rho and T at the same time-level as v
69 rho = rhop
70 T = Tp
```

```
1  import numpy as np
2  import scipy.linalg as la
3
4  # Physical Constants
5  gamma = 1.4 # Adiabatic Exponent
6  kappa = 0.024 # Thermal conductivity
7  kB = 1.38e-23 # Boltzman Constant
8  M = 29*1.67e-27 # Mass of air molecule (Average)
9  mu = 1.82e-5 # Coefficient of viscosity
10 F = (gamma-1)*M*kappa/kB # a useful constant
11
12 def Srho(rho,v,vp,tau,h):
13    # Step rho forward in time by using Crank-Nicolson
14    # on the continuity equation
15
16    N = len(rho)
17    A = np.zeros((N,N))
```

```python
    B = np.zeros_like(A)

    # Load interior points
    const = -tau/8/h
    for j in range(1,N-1):
        C1 = const * (v[j+1] + vp[j+1])
        C2 = const * (v[j-1] + vp[j-1])
        A[j,j-1] = C2
        A[j,j]   = 1
        A[j,j+1] = -C1
        B[j,j-1] = -C2
        B[j,j]   = 1
        B[j,j+1] = C1
    # Apply boundary condition
    # Write your code here

    # Crank Nicolson solve to step rho in time
    r = B@rho

    return la.solve(A,r)

def ST(T,v,vp,rho,rhop,tau,h):

    N = len(T)
    A = np.zeros((N,N))
    B = np.zeros_like(A)

    # Load interior points
    for j in range(1,N-1):
        D1 = (v[j] + vp[j])/(4*h) + 2*F/(rho[j]+rhop[j])/
        h**2
        D2 = (-(gamma-1) * (v[j+1] + vp[j+1] - v[j-1] -
        vp[j-1] )/(4*h)
              - 4*F/(rho[j] + rhop[j])/h**2 )
        D3 = -(v[j] + vp[j])/(4*h) + 2*F/(rho[j]+rhop[j])
        /h**2

        A[j,j-1] = -0.5*D1
        A[j,j]   = 1/tau - 0.5*D2
        A[j,j+1] = -0.5*D3
        B[j,j-1] = 0.5*D1
        B[j,j]   = 1/tau + 0.5*D2
        B[j,j+1] = 0.5*D3
    # Apply boundary condition
    # Insulating: dt/dx = 0

    # Write your code here

    # Crank Nicolson solve to step rho in time

    r = B@T
    return la.solve(A,r)

def Sv(v,vbar,vbarp,rho,rhop,T,Tp,tau,h):
    N = len(rho)
    A = np.zeros((N,N))
    B = np.zeros_like(A)
    E0 = np.zeros_like(v)

    # Load interior points
    for j in range(1,N-1):
        E0[j] = (-kB/4/h/M/(rho[j]+rhop[j]) *
            ( (rho[j+1] + rhop[j+1]) * (T[j+1] + Tp[j+1])
            - (rho[j-1] + rhop[j-1]) * (T[j-1] + Tp[j-1]))
        )
        E1 = (vbar[j] + vbarp[j])/(4*h)+8*mu/3/h**2/(rho[
        j]+rhop[j])
        E2 =-16*mu/3/h**2/(rho[j]+rhop[j])
        E3 =-(vbar[j] + vbarp[j])/(4*h) +8*mu/3/h**2/(rho
        [j]+rhop[j])

        A[j,j-1] = -0.5*E1
        A[j,j]   = 1/tau - 0.5*E2
        A[j,j+1] = -0.5*E3
        B[j,j-1] = 0.5*E1
        B[j,j]   = 1/tau + 0.5*E2
        B[j,j+1] = 0.5*E3

    # Apply boundary condition
    # Fixed: v = 0

    # Write your code here

    # Crank Nicolson solve to step rho in time
    r = B@v + E0
    return la.solve(A,r)
```

SOLITONS: KORTEWEG-DEVRIES EQUATION

At the Lagoon amusement park in Utah, there is a water ride called the Log Flume. It is a standard, old-fashioned water ride where people sit in a 6-seater boat shaped like a log which slowly travels along a fiberglass trough through some scenery, then is pulled up a ramp to an upper level. The slow ascent is followed by a rapid slide down into the trough below, which splashes the passengers a bit, after which the log slowly makes its way back to the loading area. But you can see something remarkable happen as you wait your turn to ride if you watch what happens to the water in the trough when the log splashes down. A large water wave is pushed ahead of the log, as you might expect. But instead of gradually dying away, as you might think a single pulse should in a dispersive system like surface waves on water, the pulse lives on and on. It rounds the corner ahead of the log that created it, enters the area where logs are waiting to be loaded, pushes each log up and down in turn, then heads out into the scenery beyond, still maintaining its shape. This odd wave is called a "soliton", or ""solitary wave", and it is an interesting feature of non-linear dynamics that has been widely studied in the last 30 years, or so. The simplest mathematical equation which produces a soliton is the KortewegdeVries equation

$$\frac{\partial y}{\partial t} + y\frac{\partial y}{\partial x} + \alpha\frac{\partial^3 y}{\partial x^3} = 0 \tag{195}$$

which describes surface waves in shallow water. In the first two terms of this equation you can see the convective behavior we studied in Lab 10, but the last term, with its rather odd third derivative, is something new. We will be studying this equation in this laboratory.

NUMERICAL SOLUTION FOR THE KORTEWEG-DEVRIES EQUATION

We will begin our study of the Korteweg-deVries equation by using Crank-Nicolson to finite difference it on a grid so that we can explore its behavior by numerical experimentation. The first step is to define a grid, and since we want to be able to see the waves travel for a long time we will copy the trough at Lagoon and make our computing region be a closed loop. We can do this by choosing an interval from $x = 0$ to $x = L$, as usual, but then we will make the system be periodic by declaring that $x = 0$ and $x = L$ are actually the same point, as would be the case in a circular trough. We will subdivide this region into N subintervals and let $h = L/N$ and $x_j = (j-1)h$, so that the grid is cell-edge. Normally such a cell-edge grid would have $N+1$ points, but ours doesn't because the last point $(j = N)$ is just a repeat of the first point: $x_N = x_0$, because our system is periodic.

We now do the usual Crank-Nicolson differencing, where we evaluate each term in the equation at time level $n + 1/2$. The last term in Eq. (??) has a third derivative, so we'll need a reasonable approximation for the third derivative. Suppose you have function values $f(x - 3h/2), f(x - h/2), f(x + h/2)$, and $f(x + 3h/2)$. A good way to do this is to write down four Taylor's series up to the fifth derivative for the function at these four points and then solve this system of four equations to find expressions for $f(x), f'(x), f''(x)$, and $f'''(x)$. This procedure yields the approximate formula

$$f'''(x) \approx \frac{f(x+3h/2) - 3f(x+h/2) + 3f(x-h/2) - f(x-3h/2)}{h^3} \tag{196}$$

along with an error term on the order of h^2. When we time average Eq. (??), we find

$$\alpha\frac{\partial^3 y}{\partial x^3} = \frac{\alpha}{2h^3}\left(y_{j+2}^{n+1} - 3y_{j+1}^{n+1} + 3y_j^{n+1} - y_{j-1}^{n+1} + y_{j+2}^n - 3y_{j+1}^n + 3y_j^n - y_{j-1}^n\right) \tag{197}$$

Look closely at Eq. (??) and also at Eq. (??) to convince yourself that they are not centered on a grid point, but at spatial location $j + 1/2$. The use of this third derivative formula thus adds a new twist to the usual Crank-Nicolson differencing: we will evaluate each term in the equation not only at time level $n + 1/2$, but also at spatial location $j + 1/2$ (at the center of each subinterval) so that the first and third derivative terms are both properly centered. This means that we will be using a cell-edge grid, but that the spatial

Figure 53: The log flume ride at Lagoon produces a solitary wave (marked by arrows in the frames above). The leading edge of the soliton is where the water begins to spill over the side of the trough.

Diederik Korteweg (1848-1941, Dutch)
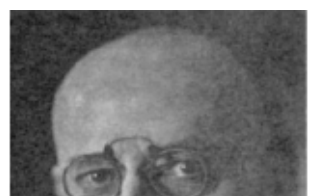
finite differences will be cell centered. With this wrinkle in mind, we can write the first term in Eq. (12.1) at time level $n+1/2$ and space location $j+1/2$ as

$$\frac{\partial y}{\partial t} = \frac{1}{2\tau} \left(y_j^{n+1} + y_{j+1}^{n+1} - y_j^n - y_{j+1}^n \right) \tag{198}$$

Now we have have to decide what to do about the nonlinear convection term $y\partial y/\partial x$. We will assume that the leading y is known somehow by designating it as $\bar{y}$ and decide later how to properly estimate its value. Recalling again that we need to evaluate at time level $n+1/2$ and space location $j+1/2$, the non-linear term becomes

$$y\frac{\partial y}{\partial x} = \frac{\bar{y}_{j+1} + \bar{y}_j}{4h} \left(y_{j+1}^{n+1} - y_j^{n+1} + y_{j+1}^n - y_j^n \right) \tag{199}$$

For now, we've ignored the problem that the derivative in Eq. (??) is centered in time at $n+1/2$ while the $\bar{y}$ term isn't. We'll have to deal with this issue later. Each of these approximations in Eqs. (??)-(??) is now substituted into Eq. (??), the y^{n+1} terms are gathered on the left side of the equation and the y^n terms are gathered on the right, and then the coefficients of the matrices **A** and **B** are read off to put the equation in the form

$$\mathbf{A}y^{n+1} = \mathbf{B}y^n \tag{200}$$

in the usual Crank-Nicolson way. If we denote the four nonzero elements of A and B like this:

$$\begin{array}{ll} A_{j,j-1} = a_{--} & A_{j,j} = a_- \\ A_{j,j+1} = a_+ & A_{j,j+2} = a_{++} \\ B_{j,j-1} = b_{--} & B_{j,j} = b_- \\ B_{j,j+1} = b_+ & B_{j,j+2} = b_{++} \end{array} \tag{201}$$

then the matrix coefficients turn out to be

$$\begin{array}{ll} a_{--} = -\frac{\alpha}{2h^3} & a_- = \frac{1}{2\tau} + \frac{3\alpha}{2h^3} - \frac{(\bar{y}_- + \bar{y}_+)}{4h} \\ a_+ = \frac{1}{2\tau} - \frac{3\alpha}{2h^3} + \frac{(\bar{y}_- + \bar{y}_+)}{4h} & a_{++} = \frac{\alpha}{2h^3} \\ b_{--} = \frac{\alpha}{2h^3} & b_- = \frac{1}{2\tau} - \frac{3\alpha}{2h^3} + \frac{(\bar{y}_- + \bar{y}_+)}{4h} \\ b_+ = \frac{1}{2\tau} + \frac{3\alpha}{2h^3} - \frac{(\bar{y}_- + \bar{y}_+)}{4h} & b_{++} = -\frac{\alpha}{2h^3} \end{array} \tag{202}$$

where $y_- = y_j$ and where $y_+ = y_{j+1}$, the grid points on the left and the right of the $j+1/2$ spatial location (where we are centering).

P12.1 Derive the formulas in Eq. (??) for the a and b coefficients using the finite-difference approximations to the three terms in the Korteweg-deVries equation given in Eqs. (??)-(??). Now that the coefficients of **A** and **B** are determined we need to worry about how to load them so that the system will be periodic. For instance, in the first row of A the entry $A_{1,1}$ is a_-, but a_{--} should be loaded to the left of this entry, which might seem to be outside of the matrix. But it really isn't, because the system is periodic, so the point to the left of $j = 1$ (which is also the point $j = (N+1)$) is the point $j-1 = N$. The same thing happens in the last two rows of the matrices as well, where the subscripts $+$ and $++$ try to reach outside the matrix on the right. So correcting for these periodic effects makes the matrices **A** and **B** look like this:

$$\mathbf{A} = \begin{bmatrix} a_- & a_+ & a_{++} & 0 & 0 & \cdots & 0 & a_{--} \\ a_{--} & a_- & a_+ & a_{++} & 0 & \cdots & 0 & 0 \\ 0 & a_{--} & a_- & a_+ & a_{++} & \cdots & 0 & 0 \\ . & . & . & . & \cdots & . & . & . \\ 0 & \cdots & 0 & 0 & a_{--} & a_- & a_+ & a_{++} \\ a_{++} & 0 & \cdots & 0 & 0 & a_{--} & a_- & a_+ \\ a_+ & a_{++} & 0 & 0 & \cdots & 0 & a_{--} & a_- \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} b_- & b_+ & b_{++} & 0 & 0 & \cdots & 0 & b_{--} \\ b_{--} & b_- & b_+ & b_{++} & 0 & \cdots & 0 & 0 \\ 0 & b_{--} & b_- & b_+ & b_{++} & \cdots & 0 & 0 \\ . & . & . & . & \cdots & . & . & . \\ 0 & \cdots & 0 & 0 & b_{--} & b_- & b_+ & b_{++} \\ b_{++} & 0 & \cdots & 0 & 0 & b_{--} & b_- & b_+ \\ b_+ & b_{++} & 0 & 0 & \cdots & 0 & b_{--} & b_- \end{bmatrix} \tag{203}$$

P12.2 Discuss these matrices with a TA and convince the TA that this structure correctly models a periodic system (it may help to think about the computing grid as a circle with $x_0 = x_N$.)

$$\tilde{v} = v^n \tag{204}$$

$$v^p = \mathbf{A}(v^n)^{-1} \left[\mathbf{B}(v^n) v^n \right] \tag{205}$$

$$v^* = \frac{v^p + v^n}{2} \tag{206}$$

$$v^{n+1} = \mathbf{A}(v^*)^{-1} \left[\mathbf{B}(v^*) v^n \right] \tag{207}$$

All right, that's it. You may have the feeling by now that this will all be a little tricky to code, and it is. We would rather have you spend the rest of the time in this lab doing physics instead of coding, so below (and on the course web site) you will find a copy of a Python program kdv. py that implements this algorithm. You and your lab partner should carefully study the script to see how each step of the algorithm described above is implemented, then work through the problems listed below by running the script and making appropriate changes to it.

SOLITONS

P12.3

(a) Run kdv . py with $\alpha = 0.1, y_{max} = 2, \tau = 0.5, t_{final} = 100$, and iskip $= 1$. After a while you should see garbage on the screen. This is to convince you that you shouldn't choose the time step to be too large.

(b) Now run (a) again, but with $\tau = 0.1$, then yet again with $\tau = 0.02$. Use $t_{final} = 10$ for both runs and skip big enough that you can see the pulse moving on the screen. You should see the initial pulse taking off to the right, but leaving some bumpy stuff behind it as it goes. The trailing bumps don't move as fast as the big main pulse, so it laps them and runs over them as it comes in again from the left side, but it still mostly maintains its shape. This pulse is a soliton. You should find that there is no point in choosing a very small time step; $\tau = 0.1$ does pretty well.

The standard "lore "in the field of solitons is that the moving bump you saw in problem 12.3 is produced by a competition between the wave spreading caused by the third derivative in the Korteweg-deVries equation and the wave steepening caused by the $y\partial y/\partial x$ term. Let's run kdv. py in such a way that we can see the effect of each of these terms separately.

P12.4

(a) Dispersion (wave-spreading) dominates: Run kdv. py with $\alpha = 0.1, y_{max} = 0.001, \tau = 0.1$, and $t_{final} = 10$. The small amplitude makes the nonlinear convection term $y\partial y/\partial x$ be so small that it doesn't matter; only the third derivative term matters. You should see the pulse fall apart into random pulses. This spreading is similar to what you saw when you solved SchrÃűdinger's equation. Different wavelengths have different phase velocities, so the different parts of the spatial Fourier spectrum of the initial pulse get out of phase with each other as time progresses.

(b) Non-linear wave-steepening dominates: Run kdv. py with $\alpha = 0.01, y_{max} = 2, \tau = 0.01$, and $t_{final} = 10$. (If your solution develops short wavelength wiggles this is an invitation to use a smaller time step. The problem is that the predictor-correction algorithm we used on the nonlinear term is not stable enough, so we have a Courant condition in this problem.) Now it is the dispersion term that is small and we can see the effect of the non-linear convection term. Where y is large the convection is rapid, but out in front where y is small the convection is slower. This allows the fast peak to catch up with the slow front end, causing wave steepening. (An effect just like this causes ocean waves to steepen and then break at the beach.)

The large pulse that is born out of our initial Gaussian makes it seem like there ought to be a single pulse that the system wants to find. This is, in fact the case. It was discovered that the following pulse shape is an exact solution of the Korteweg-deVries equation:
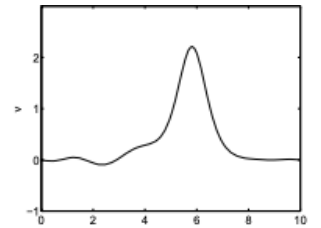

Figure 54: A Gaussian pulse after 1 second of propagation by the Korteweg-deVries equation (Problem 12.3)
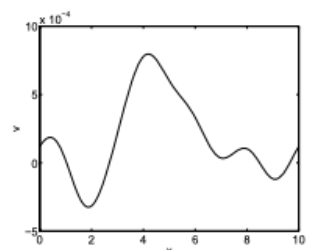

Figure 55: Dispersion dominates (Problem 12.4(a)):

$$y(x,t) = \frac{12k^2\alpha}{\cosh^2\left(k\left(x - x_0 - 4\alpha k^2 t\right)\right)} \tag{208}$$

where x_0 is the center of the pulse at time $t = 0$.

P12.5

(a) Use Mathematica to show that this expression does indeed satisfy the Korteweg-deVries equation

(b) Now replace the Gaussian initial condition in kdv. py with this pulse shape, using $k = 1.1, x_0 = L/2$, and adjusting α so that the height of the initial pulse is exactly equal to 2, so that it matches the Gaussian pulse you ran in 12.3. You should find that this time the pulse does not leave trailing pulses behind, but that it moves without changing shape. It is a perfect soliton.

(c) The formula at the beginning of this problem predicts that the speed of the pulse should be

$$c_{\text{soliton}} = 4\alpha k^2 \tag{209}$$

Verify by numerical experimentation that your soliton moves at this speed. To do this, you can just track the position of the maximum value (the NumPy function ar gmax can help you do this).

12.6   One of the most interesting things about solitons is how two of them interact with each other. When we did the wave equation earlier you saw that left and right moving pulses passed right through each other. This happens because the wave equation is linear, so that the sum of two solutions is also a solution. The Korteweg-deVries equation is nonlinear, so simple superposition can't happen. Nevertheless, two soliton pulses do interact with each other in a surprisingly simple way.

To see what happens keep $\alpha = 0.1$, but modify your code from 12.5 so that you have a soliton pulse with $k = 1.5$ centered at $x = 3L/4$ and another soliton pulse with $k = 2$ centered at $x = L/4$. Run for about 20 seconds and watch how they interact when the fast large amplitude pulse in the back catches up with the slower small amplitude pulse in the front. Is it correct to say that they pass through each other? If not, can you think of another qualitative way to describe their interaction?

```
 1 import matplotlib.pyplot as plt
 2 import numpy as np
 3 import scipy.linalg as la
 4
 5 # Physical constants
 6 alpha = 0.1
 7
 8 # Make the grid
 9 N = 500
10 L = 10
11 h = L/N
12 x = np.linspace(h/2,L-h/2,N)
13
14 # Initial Gaussian centered on the computing region
15 ymax = 2
16 y = ymax * np.exp(-(x-.5*L)**2)
17
18 # Time range
19 tau = 0.1
20 tfinal = 100
21 t = np.arange(0,tfinal,tau)
22
23 # Initialize the parts of the A and B matrices that
24 # do not depend on ybar and load them into At and Bt.
25 # Make them be sparse so the code will run fast.
26 At = np.zeros((N,N))
27 Bt = np.zeros((N,N))
28
29 # Function to wrap the column index
30 def jwrap(j):
31   if (j < 0):
32     return j + N
33   if (j >= N):
34     return j - N
35   return j
36 # load the matrices with the terms that don't depend
     on ybar
37 h3 = h**3
38 for j in range(N):
39   At[j,jwrap(j-1)] =-0.5*alpha/h3
```

```
40|    At[j,j] = 0.5/tau + 1.5*alpha/h3
41|    At[j,jwrap(j+1)] = 0.5/tau - 1.5*alpha/h3
42|    At[j,jwrap(j+2)] = 0.5*alpha/h3
43|
44|    Bt[j,jwrap(j-1)] = 0.5*alpha/h3
45|    Bt[j,j] = 0.5/tau - 1.5*alpha/h3
46|    Bt[j,jwrap(j+1)] = 0.5/tau + 1.5*alpha/h3
47|    Bt[j,jwrap(j+2)] =-0.5*alpha/h3
48|
49|
50|plt.figure(1)
51|skip = 10
52|for n in range(len(t)):
53|    # Predictor step
54|    A = np.copy(At)
55|    B = np.copy(Bt)
56|
57|    # load ybar, then add its terms to A and B
58|    ybar = np.copy(y)
59|    for j in range(N):
60|        tmp = 0.25*(ybar[jwrap(j+1)] + ybar[j])/h
61|        A[j,j] = A[j,j] - tmp
62|        A[j,jwrap(j+1)] = A[j,jwrap(j+1)] + tmp
63|        B[j,j] = B[j,j] + tmp
64|        B[j,jwrap(j+1)] = B[j,jwrap(j+1)] - tmp
65|
66|    # do the predictor solve
67|    r = B@y
68|    yp = la.solve(A,r)
69|
70|    # corrector step
71|    A = np.copy(At)
72|    B = np.copy(Bt)
73|
74|    # average current and predicted y values to
  |    correct ybar
75|    ybar=.5*(y+yp)
76|    for j in range(N):
77|        tmp = 0.25*(ybar[jwrap(j+1)] + ybar[j])/h
78|        A[j,j] = A[j,j] - tmp
79|        A[j,jwrap(j+1)] = A[j,jwrap(j+1)] + tmp
80|        B[j,j] = B[j,j] + tmp
81|        B[j,jwrap(j+1)] = B[j,jwrap(j+1)] - tmp
82|
83|    # do the final corrected solve
84|    r = B@y
85|    y = la.solve(A,r)
86|
87|    if (n % skip == 0):
88|        plt.clf()
89|        plt.plot(x,y)
90|        plt.xlabel('x')
91|        plt.ylabel('y')
92|        plt.title('time={:1.3f}'.format(t[n]))
93|        plt.ylim(0,3)
94|        plt.pause(.1)
```

MACHINE LEARNING WITH NEURAL NETWORKS

MACHINE LEARNING TERMINOLOGY

In this lab, we'll dip our toes in the field of machine learning. In very loose terms, machine learning boils down to the process of using general algorithms to model (sometimes complicated) relationships between system parameters. In a typical scenario we'll have an observable quantity which is thought to relate to a set of input data. For example, suppose you have a bunch of pictures of pets that you want to classify by the type of animal in the picture (cat, dog, parrot, etc.). The observable quantity is the type of animal, and the data consists of the pixel locations and values in the picture.

To get started on this picture classification project, you could manually classify a bunch of pictures by the type of animal in them and then provide that training data to the machine learning algorithm. The algorithm uses the training data to come up with a model that takes a picture as input, and predicts the type of animal in the pictures. You can then use the output model to automatically classify the rest of your pet pictures. This process is called supervised learning.

Supervised Learning: The researcher provide a machine learning algorithm with a "training"data set where the outcomes for each of the given data inputs are known, and the algorithm comes up with a function relating the inputs and outputs that can then be used on new data

Model: The output of a machine learning algorithm that represents a relationship between inputs and outputs.

The picture-sorting procedure outlined above is an example of a classification problem. Classification algorithms are designed to fit sets of data into discrete bins (cat, dog, etc.). In contrast, regression algorithms aim to predict a value based on the input data. For example, you might try to use machine learning to predict the temperature tomorrow. In this case, the measurable quantity is a number representing the temperature, and the input data would be things like the historic temperature record, the date, the location, recent weather, etc.

Classification: The measurable quantity is restricted to a discrete set of values. Regression: The measurable quantity is allowed to take on a continuous range of values.

Sometimes, you don't have a measurable quantity you want to predict from your data set, but rather want to explore the structure of the data itself. For example, say Facebook wants to divide their users into groups of people with similar demographics so they can target advertising to these groups. They want to be able to say something like "this ad is likely to be interesting to demographic group α but uninteresting to group β." But they don't know the best way to define the groups α and β. This is a job for a clustering algorithm, which is a type of unsupervised learning. Facebook could feed a bunch of information about a set of people into a clustering algorithm, and the algorithm would return suggestions for how to assign those people to groups "similar "demographics (i.e. tell you how to define groups α and β above). Once they've classified the data into groups, they could then use a supervised machine learning algorithm to predict which groups might be likely to click on which advertisements.

Unsupervised Learning: An algorithm that looks for patterns or groupings in a set of data without respect to any presuppositions about the data.

Clustering: A type of unsupervised learning algorithm that separates similar data into groups or categories.

P13.1 Explain the machine learning concepts introduced above to a TA. There are many, many more ideas in machine learning. You can take many full courses on the subject. But just to get the flavor of how this works, let's tackle some specific problems using a common machine learning technique referred to as a neural network.

LINEAR REGRESSION

A computational neural network can be thought of as an extension of linear regression. Linear regression finds a linear function which maps an input, x, onto an output, y. Circles

Study Time (h)	Score (%)
1.4	20
1.8	30
2.1	37
2.2	45
2.3	26
2.9	86
3.5	67
3.6	100
4.2	72
4.6	82
4.7	99

Table 1: Table of exam scores vs. study time.

are often used to represent the nodes (values) of a neural network. Using that notation, a linear regression mapping would be represented this way:

Questions that linear regression can answer are along the lines of this: given a set of known inputs with corresponding known outputs, can one predict what the output will be for a new input that wasn't part of the original set? As an example, perhaps you are trying to correlate exam scores with the amount of time students studied for the exam as shown in Table 13.1. You could make a plot of the data and fit a line to it as shown in Fig. 13.1. Then you could use your fit to make a prediction for a student not in the given data set, such as "Given a time $x = 2.7$ hours, the expected exam score would be $y = 53$ percent."The linear fit is typically done by choosing the slope and intercept parameters to minimize some cost or loss function, for example the squared error.
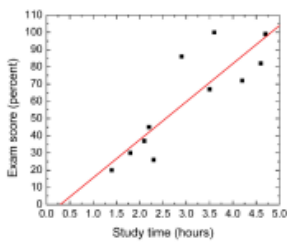
Simple linear prediction is often insufficient. For example, this linear fit will predict a negative score for particularly low values of study time, which is nonsensical. So it would make sense to apply a nonlinear function after the linear fit, something like $y = \max(0, z)$ where we are now using the symbol z to represent the value that is given from the linear fit, $z = wx + b$, so we can continue to use the symbol y for the final output. (We are using the letter w to represent the slope of the line for reasons made clear below.) Applying that particular nonlinear function is equivalent to saying, "Set y to 0 if z is negative, otherwise use $y = z$". In a neural network, a function applied after the linear fit is called an activation function, and that particular activation function is probably the most commonly used one. It is called the relu function, which stands for rectified linear unit. Other activation functions are also frequently used. For example, another common situation is when the output value must be between 0 and 1 ; in that case a nonlinear function called a sigmoid function, is used to map all the z values (potentially ranging from $-\infty$ to $+\infty$) to real values of y between 0 and 1 .

Figure 57: Simple linear fit of the test data shown in Table 13.1.

MULTIPLE INPUT LINEAR REGRESSION

Linear regression can also be done with multiple inputs. Continuing the exam score example, additional inputs could represent other known quantities such as how much sleep the student got the night before the exam, the student's class attendance, the student's grades on homework and past exams, etc. The situation might now look like this; we are using z, y notation at the final node to indicate that the output node has both pre- and post-activation values. The linear function which would now be fit is $z = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + b$, and then an activation function could be applied to the value of z to obtain y. The slopes of the linear regression are now called weights, hence the symbol w. Each w is represented in the figure by an arrow. The optimal values of the parameters w_1, w_2, w_3, w_4, and b will be calculated together using known data to
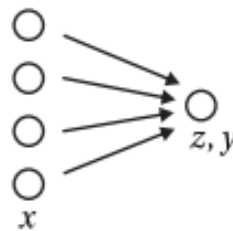
minimize a cost function.

NEURAL NETWORKS

A neural network is this same idea as multiple regression, just expanded to multiple layers. The layers between input and output are called hidden layers. Each hidden layer could have its own activation function. Post-activation values of the nodes of the hidden layers are called the activations of the nodes and are represented by the letter a. Here's an example of a three layer neural network; by convention, the input layer of x 's is not
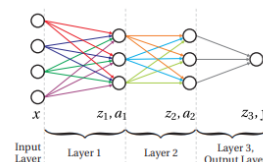
counted in the number of layers of the network.

Figure 58: Example of a fully connected three-layer network.

Note that in this example, layer 1 is a collection of 15 parameters (4 w 's and 1 b for each of the three nodes in layer 1), an activation function, and 3 nodes. This nomenclature is not necessarily uniform; some people may just refer to the nodes themselves when discussing a layer. Also, sometimes the a and z values are counted as separate layers, in which case this example would be called a six-layer neural network. The structure of the network, meaning the number of layers, and the number of nodes in each layer, and their activation functions, is chosen with the goal of obtaining the best possible predictions. It is fairly common, however, to reduce the number of nodes in each layer as one moves from input to output as in this example here.

MORE COMPLICATED LAYERS

The layers shown in Fig. 13.2 are termed fully connected layers, meaning each node of the previous layer is connected via an arrow (i.e. a w) to each node of the next layer. Incidentally, this type of network is called a multilayer perceptron. More complicated layers can also be used. One type of layer which is frequently used in image analysis is called a convolutional layer. A color image input is represented numerically by a 3 -dimensional array: 2 dimensions for horizontal and vertical values, plus a third dimension specifying whether the given numbers refer to the R, G, or B color channel. A convolutional layer takes a 3-dimensional array as an input, does an operation on it, and then passes a new 3 -d array to the next layer. One can have multiple convolutional layers in a network, but to convert from a convolutional layer to a fully connected layer at some point, the data must be "flattened" from a $3-d$ array to a $1-d$ set of values.

GRADIENT DESCENT

In the specific neural net shown in Fig. 13.2, layer 1 contained 15 fitting parameters. Similarly, layer 2 contains 12 parameters and layer 3 has 4 parameters. (Take a moment to look at the figure and make sure these numbers make sense to you.) Therefore 31 total parameters need to be optimized using the input data. For more complicated neural nets, it is not uncommon for there to be hundreds of thousands of fitting parameters! How are these to be optimized? The process is typically done using the gradient descent method (or a close relative of it). Gradient descent is an iterative process whereby the location of the minimum of a function (e.g. the cost function) is found by taking steps along the "downhill" direction, i.e. the direction of the negative gradient. This is most easily visualized with a function of two parameters, say w_1 and w_2, as illustrated in Fig 13.3. At each step in the iteration, the cost function is computed based on the current values of w_1 and w_2, the gradients of the cost function with respect to w_1 and w_2 are computed at the present location of w_1 and w_2, and then the values of w_1 and w_2 are updated according to:

$$w_{1,\text{ next}} = w_1 - \alpha \frac{d(\cos t)}{dw_1}$$

$$w_{2,\text{ next}} = w_2 - \alpha \frac{d(\cos t)}{dw_2}$$

For known activation functions and a known cost function, the gradients can be automatically computed by the machine learning software in an interesting backpropagation process that minimizes calculation time, but is beyond the scope of this lab.

HYPERPARAMETER TUNING AND TRAINING/VALIDATION SETS

In those equations, the symbol α is called the learning rate and must be carefully chosen. It describes how large a step is taken during each iteration. Too small of a value and the process will take many iterations to converge. Too large a value and the steps may overshoot the minimum that is being sought. The input α is an example of a hyperparameter, meaning a parameter that affects the machine learning but is not automatically optimized by the algorithm itself (unlike the w's and b 's). Other examples of hyperparameters include the number of layers, number of nodes in each layer, the activation functions, and even the cost function itself. Sometimes these hyperparameters can be tuned in a systematic way, but often selecting the best hyperparameters involves a degree of trial and error. How can one decide which hyperparameters are best? It would be tempting to just
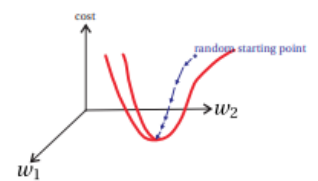

Figure 59: Taking steps by adjusting the w_1 and w_2 parameters along the negative gradient direction to find the minimum of the cost function.

look at the cost function and pick the hyperparameters which produce the lowest possible value. However, with potentially hundreds of thousands of fitting parameters, the danger of overfitting the data is very real, meaning you could end up with a set of parameters that do a fantastic job on fitting the given data, but which make horrible predictions on any new data. To guard against that, the known data is typically separated into a training set and a validation set. The validation set is also known as the development set or "dev set"; sometimes it is also called the "test set," although at times "test set" refers to a third set used for additional testing after the model is complete so we will avoid that usage. The training set is used to optimize the network parameters to minimize the cost function, i.e., to "train the model", and then the trained neural network is checked against the validation set to make sure the network can be relied on to give predictions which make sense.

THE KERAS LIBRARY

Fortunately, the many steps involved with training and testing neural networks are greatly simplified using machine learning libraries, of which one of the most used for Python is called keras. The examples below all use keras. Use this pip install command:

```
pip install keras
```

Use these additional commands if needed:

```
pip install tensorflow
pip install -U numpy
```

If keras still does not work after those commands, you may have a version conflict. In that case, see Appendix A for some basics about environments in Anaconda and download an environment file from our website, which should help. Install and activate it, and then the specified libraries should work together.

EXAMPLE 1: PREDICTING EXAM SCORES

Here is what the code would look like to use a neural net framework to analyze the exam score vs. study time data set given above, and then to make a prediction for $x = 2.7$ hours. As a quick note about notation, while the letters x and y are commonly used to refer to a single input data point and its corresponding output, the collections of all inputs and outputs are usually referred to as capital X and Y. This code sets up a 1 layer neural network with 1 node in the layer, which receives a single input from an input layer. The model=Sequent ial () command creates a sequential (normal) neural net with name model. The layer is created via the model.add (Dense $(1, activation =' linear', input_dim = 1)$) command, where the first 1 refers to the number of nodes in the layer and the second 1 designates the size of the input layer (a single input node). We use sgd as the optimizing routine, which stands for stochastic gradient descent, and a mean squared error cost (or loss) function. For this simple example we are not bothering with a validation set and we have used a linear activation function (meaning the identity, $y = z$) instead of a relu one. The predicted y for $x = 2.7$ should be about $y \approx 54.4$.

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from keras import optimizers

data = np.array([[1.4, 20.], [1.8, 30.], [2.1, 37.],
    [2.2, 45.],
        [2.3,26.], [2.9, 86.], [3.5, 67.], [3.6,
    100.],
           [4.2, 72.], [4.6, 82.], [4.7, 99.]])

[X_train,Y_train] = data.transpose()

model = Sequential()
model.add(Dense(1, activation='linear', input_dim=1))
sgd = optimizers.sgd(lr=0.03) #lr is the learning
    rate
model.compile(loss='mean_squared_error',optimizer=sgd
    )

#this command does the minimization
model.fit(X_train, Y_train, epochs=100, verbose=1)

#displays some info, note there are only 2 fitting
    parameters
model.summary()
```

```
22|
23| # the predict function on the next line assumes an
24| # array of inputs so we need to put 2.7 inside an
   |    array
25| x = np.array([2.7])
26|
27| #... and predicted_y is likewise an array of
   |    predicted outputs
28| predicted_y = model.predict(x)
29|
30| print('For x={:f} the predicted y is: {:f}'.format(
   |    float(x),float(predicted_y)))
```

P13.2   Execute the code above, and examine the output. Explain to the TA what the code does.

EXAMPLE 2: PREDICTING MEDIAN HOME PRICES

A commonly used data set for testing machine learning algorithms, and one built into keras, is that of Boston home prices from the 1970 s. It involves 13 metrics taken for 506 different communities in the Boston area, along with the median home price for the community. Reading it into your code from the keras . dat asets as is done below will automatically sort the data into 404 training data points and 102 validation data points with the 13 metrics as the *X* vector and the median home price as *Y*. Technically *X* is a 2-d array, because each *x* is a collection of 13 pieces of information, and *X* is the collection of all *x*'s for all 404 training data points.

This code sets up a five-layer neural network with $30, 15, 8, 5,$ and 1 nodes per layer respectively. There are 1,064 total fittable parameters. Notice how each layer is just added via a model. add command, and only the first input layer needs to have input_dim specified (which is the number of nodes in the input layer). In this example a dam is used as the optimizing routine; it is a slightly more sophisticated version of gradient descent and is a popular choice for many applications.

```
1| import numpy as np
2| from keras.models import Sequential
3| from keras.layers import Dense
4| from keras import optimizers
5| from keras.datasets import boston_housing
6|
7| (X_train, Y_train), (X_validate, Y_validate) =
 |    boston_housing.load_data()
8|
9| #description here: https://github.com/eric-bunch/
 |    boston_housing
10|
11| print(X_train.shape)
12| print(Y_train.shape)
13| print(X_validate.shape)
14| print(Y_validate.shape)
15|
16| model = Sequential()
17| model.add(Dense(30, activation='relu',input_dim=13))
18| model.add(Dense(15, activation='relu'))
19| model.add(Dense(8, activation='relu'))
20| model.add(Dense(5, activation='relu'))
21| model.add(Dense(1, activation='linear'))
22| optimizer_choice = optimizers.adam(lr=0.05)
23| model.compile(loss='mean_squared_logarithmic_error',
24| optimizer=optimizer_choice, metrics=['mse'])
25| model.fit(X_train, Y_train, batch_size=32, epochs=50,
 |    verbose=1)
26| model.summary()
27| score = model.evaluate(X_validate, Y_validate,
 |    verbose=0)
28|
29| print('The loss value and accuracy of the test set is
 |    : ' + str(score))
30|
31| #prediction for a specific input
32| test_num = 27 #randomly chosen number
33| x = np.array([X_validate[test_num]])
34| predicted_y = model.predict(x)
35| print('For input parameters x = ')
36| print(X_validate[test_num])
37| print('the predicted y (median value of home, in
 |    $1000s) is:
38| {:f} '.format(float(predicted_y)))
39| print('and the actual y value was: {:f}'
40| .format(float(Y_validate[test_num])))
```

P13.3   Execute the code above, and examine the output. Explain to the TA what the code does.

EXAMPLE 3: PREDICTING HANDWRITTEN DIGITS

Now we will do a more complicated example using image classification with the MNIST data set which contains images of handwritten digits, each labeled with the answer for the correct digit. Our neural net will use the data set to learn how to recognize the digits. Each data point is a 28ÃŬ28 pixel monochrome image. Reading the data set into your code from the keras.datasets as is done below will automatically sort the data into 60,000 training data points and 10,000 validation data points with the with the pixel values as X and the correct digit as Y.

This code sets up a 4 layer neural network where the first two layers are of the convolutional type mentioned above (convolutional hyperparameters: 32 convolutional "filters"and a 3 ÃŬ 3 convolutional grid for each layer), followed by two regular (dense) layers of sizes 128 and 10, respectively. The output layer has size 10 because for categorizing data like this, the output of the neural net needs to be 10 different outputs, each one corresponding to the likelihood of the image being a particular digit. So instead of 3, 5, or 6, our outputs need to be things like [0001000000], [0000010000], and [0000001000]. The input shape of the X values must be specified for the first layer; after that, keras is smart enough to determine the needed array sizes and shapes. The output of the second layer must be flattened prior to going into the third layer. Some of the layers employ dropout which helps prevent overfitting. Pooling is a technique to reduce the size of the convolutional arrays. The output layer, as mentioned above, needs to have 10 separate outputs. The softmax activation choice for the final layer makes it so the 10 output values sum to 1 and can be interpreted as probabilities. The cost function is categorical_crossentropy which is the typical choice for sorting data into categories. Because this is a more complicated example, some additional comments are given in the code.

```python
import numpy as np
from matplotlib import pyplot as plt
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.utils import np_utils
from keras.datasets import mnist

(X_train, Y_train), (X_validate, Y_validate) = mnist.
    load_data()
#X_train has dimensions (60000, 28, 28): 60,000
    images which are 28x28

#number of data points in training set traditionally
    called m
m = X_train.shape[0]

#view two of the images, images 10 and 11 just chosen
    randomly
plt.figure(1)
plt.imshow(X_train[10])
plt.figure(2)
plt.imshow(X_train[11])

#view the image labels corresponding to the two
    images
print('image 10 label:' + str(Y_train[10]))
print('image 11 label:' + str(Y_train[11]))

#There are three problems with the supplied data.

# First, the convolutional layers will expect the
    image data
# to be in a "channel", even if it's just a
    monochrome channel.
# Let's fix that.
X_train = X_train.reshape(m, 28, 28, 1)

#Now the dims are (60000, 28, 28, 1)

# Side note: this would need to be (60000, 28, 28, 3)
# if we had RGB image data

# Secondly, neural nets tend to work best for inputs
# between 0 and 1, so let's normalize the data
X_train = X_train/255
```

```
42 # Thirdly, the y-values are just digits rather than
   |    the
43 # needed arrays of size 10 (corresponding to
   |    probabilities
44 # of digits 0-9, respectively). There's a built-in
   |    function
45 # called "to_categorical" for that.
46
47 Y_train = np_utils.to_categorical(Y_train, 10) #now
   |    has dims (60000, 10)
48
49 #view the new image labels corresponding to the two
   |    images
50 print('new image 10 label:' + str(Y_train[10]))
51 print('new image 11 label:' + str(Y_train[11]))
52
53 #Now to define the neural net structure and
   |    hyperparameters.
54 model = Sequential()
55
56 #layer 1:
57 model.add(Conv2D(32, (3, 3), activation='relu',
   |    input_shape=(28,28,1)))
58
59 #layer 2:
60 model.add(Conv2D(32, (3, 3), activation='relu'))
61 model.add(MaxPooling2D(pool_size=(2,2)))
62 model.add(Dropout(0.25))
63
64 #layer 3:
65 model.add(Flatten())
66 model.add(Dense(128, activation='relu'))
67 model.add(Dropout(0.5))
68
69 #layer 4 (output):
70 model.add(Dense(10, activation='softmax'))
71 model.compile(loss='categorical_crossentropy',
   |    optimizer='adam',
72 metrics=['accuracy'])
73
74 # And now we train the model. This takes a long time,
   |    perhaps
75 # 10 mins depending on your computer, since there are
   |    over
76 # 600,000 adjustable parameters in the 4-layer model
   |    we have
77 # just defined! The bulk of them come from the 4608
   |    outputs
78 # from layer 2 connecting densely to 128 nodes in
   |    layer 3.
79 # The "batch_size" setting in the command indicates
   |    how many
80 # training inputs it should consider at once as it is
   |    adjusting
81 # the parameters, and the "epochs" number indicates
   |    how many
82 # complete iterations through the entire training set
   |    it should do.
83
84 model.fit(X_train, Y_train, batch_size=32, epochs=6,
   |    verbose=1)
85
86 model.summary()
87
88 # And now it's done! We can make predictions. To get
   |    a general sense as to
89 # the quality of the NN predictions, we can use the
   |    validation set... but we
90 #must first address the same three issues we did with
   |    the training set.
91
92 m_test = X_validate.shape[0]
93
94 print('Test set has {:d} entries.'.format(m_test))
95 X_validate = X_validate.reshape(m_test, 28, 28, 1)
96 X_validate = X_validate/255 # to normalize X values
   |    between 0 and 1
97 Y_validate = np_utils.to_categorical(Y_validate, 10)
98
99 #the output of this next command will tell you how
   |    good the NN did on the test set
100 score = model.evaluate(X_validate, Y_validate,
   |    verbose=0)
101 print('The loss value and accuracy of the test set is
   |    : ' + str(score))
102
103 #It's also fun to look at predictions for single
   |    examples. "140" here was just
104 #a random selection. You can copy and paste these
```

```
          next few lines into the
105| #console with different test numbers to see an image
          along with its predicted
106| #output value.
107|
108| testimagenumber = 140
109| singletest=X_validate[testimagenumber]
110| plt.figure(3)
111|
112| #must convert back to format imshow can use
113| plt.imshow(np.round(singletest*255).squeeze())
114|
115| #model.predict expects an array, so give it an array
          of length 1
116| singleprediction = model.predict(np.array([singletest
          ]))[0]
117|
118| #argmax function converts from the length 10 output
          array back to a single digisingleprediction = np.
          argmax(singleprediction)
119| print('The prediction for image {:d} is: {:d} '
120|     .format(testimagenumber, singleprediction))
```

P13.4 Execute the code above, and examine the output. Explain to the TA what the code does.
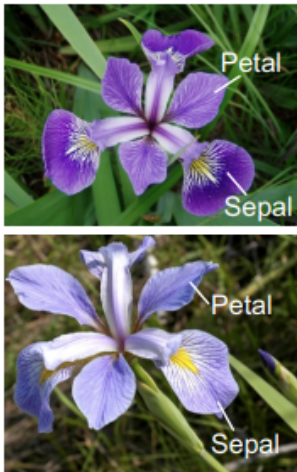
OTHER RESOURCES

We have just scratched the surface of machine learning and neural networks. Here are a few more resources which you may find helpful if you want to do additional exploring.

- How to save and load trained models: keras.io/getting-started/faq/

- List of keras built-in activation functions: keras.io/activations/

- List of keras built-in cost functions: keras.io/losses/

- List of keras built-in optimizers: keras.io/optimizers/

- Some good advice on how to choose a cost function: machinelearningmastery.com/how-to-choose-loss-functions-when-trainingdeep-learning-neural-networks/

PETALS AND SEPALS

Figure 60: Petals and a sepals for Iris versicolor (top) and Iris virginica (bottom). (Photos from Wikipedia)

Another classic data set used for testing/demonstrating machine learning algorithms is the Iris flower data set. This is a classification problem where four flower characteristics (petal length, petal width, sepal length, and sepal width) are used to determine whether an iris is of the Iris setosa, Iris virginica or Iris versicolor varieties. Below is some code which loads the dataset and randomly splits it into the training and validation sets. Note the two sklearn commands; sklearn is the implementation of scikit-learn, another free software machine learning library for Python.

```
1| from sklearn.datasets import load_iris
2| from sklearn.model_selection import train_test_split
3|
4| iris = load_iris()
5| X = iris['data'] #150 entries, 4 parameters each
6| Y = iris['target'] #150 entries, values of 0, 1, or 2
7| names = iris['target_names']
8| feature_names = iris['feature_names']
9| X_train, X_validate, Y_train, Y_validate \
10|     = train_test_split(X, Y, test_size=0.333)
```

You may want to print out any or all of the defined variables to see what you are working with.

P13.5 Develop a neural net which will learn to separate the irises based on the four characteristics, with training and validation accuracies both over 95 %. Hint: This problem is much like Example 3 in that it's a classification problem. Here are some specific things you should do which are quite similar to that example:

- You'll need to import the appropriate keras libraries.

- Use the np_utils.to_categorical function to turn the outputs which are 0, 1, or 2 into three separate outputs which are each 0 or 1. You will need to do this for Y_validate as well as for Y_train.

- Use a softmax activation function for the final layer, but with 3 nodes as opposed to 10 nodes

- Use the categorical_crossentropy cost function

- Use the adam optimizer

- Use metrics=['accuracy'] in your compile command

- Evaluate your model with X_validate and Y_validate

Unlike Example 3, you should not use any convolution layers; instead, do the problem with a fully connected network like Example 2 . You can choose the number of layers (at least 2, though) and the number of nodes in each layer. Also like Example 2, be sure to specify the size of the input layer in your first model.add command. If your network seems to be overfitting the data, as seen by a very high accuracy with the training data but a low accuracy with the validation data, try simplifying your network. A note on the model compile command: You may have noticed that the mo del. compil e command was used in two slightly different ways for Examples 2 and 3. In Example 2 it was done like this:
optimizer_choice = optimizers.adam(lr=0.05)
model.compile(loss='mean_squared_logarithmic_error',
optimizer=optimizer_choice, metrics=['mse'])

Whereas in Example 3 it was like this:
model.compile(loss='categorical_crossentropy',
optimizer='adam',metrics=['accuracy'])

The difference between the two is that if you want to specify any hyperparameters for the optimizer (e.g. adam), such as the learning rate, then you must precede the compile command with an optimizers command as in Example 2 . And then inside the mo del. compil e command you set the optimizer equal to the name of the optimizer variable you have just created, without quotation marks. If, however, you are content to use the default hyperparameters for the optimizer, then inside the model . compile command, you can set the optimizer equal to a string indicating your choice, in quotes.

APPENDIX A

ANACONDA ENVIRONMENTS

With the large number of packages and libraries available for use with Python, it is not uncommon for them to conflict with each other. (Libraries are collections of packages.) Additionally, sometimes a package may work with one version of Python but not with another. To address these types of issues, Anaconda has implemented the ability to create and load "environments", and in fact that is one of the compelling reasons to use the Anaconda version of Python in research settings where multiple people may be sharing code with each other.

An environment file is basically a list of all of the package versions that are desired for a given configuration, including the version of Python itself. By loading an environment file in Anaconda prior to running Python, the user tells Anaconda to force Python to use those specified versions. That way, for example, someone working on code can save his or her environment, send the environment file to a collaborator along with the code, and be sure that the collaborator will be able to run the code in the exact same way, with no version conflict errors or other unexpected behavior.

Environments are created and loaded from the Anaconda Prompt, an app which was installed when you installed Anaconda. The Anaconda Prompt is a command line interface which lets you run commands related to Anaconda itself (as opposed to Python commands).

Some of the details below may differ slightly based on your operating system. For simplicity we will assume a Windows machine. If you open the Anaconda Prompt from the Windows search menu, you should see the following text:

```
(base) C:\Users \yourname>
```

where yourname is your Windows username. The (base) prompt means that it is using the default environment as opposed to a specially loaded one, and the given directory (i.e. C: \ Us ers \ yourname >) is the default working directory from which Anaconda will save and load files. After you load an environment, (base) will be replaced by the name of the environment.
Environment files are often titled "environment.yml". To load an environment, first copy the environment.yml file into the working directory and type this into the command prompt:

```
conda env create -f environment. yml
```

The -f option means the next word will be the filename. This command will cause Anaconda to download all of the packages which are specified inside the environment.yml file, with specific versions. It may take a while but is a one-time thing.After Anaconda finishes with that command you have an environment you can use! The name of the environment is specified inside the environment.yml, and Anaconda will let you know its name when it is finished, by generating a response like this [where "yourenvironment" is whatever name was specified inside the file]:

```
# To activate this environment, use
#
# $ conda activate yourenvironment
#
# To deactivate an active environment, use
#
# $ conda deactivate
```

At this point you will be able to see that a new folder with the yourenvironment name has been created in your C : \ Users \ yourname \ Anaconda 3\ envs directory, filled with lots of stuff you don't need to worry about. If you have multiple environments installed, you can check that directory to see them all.
As the response indicates, to use the environment you should type this into the command prompt:

```
conda activate yourenvironment
```

That tells Anaconda to start using the specified environment and the command prompt will now look like this:

```
(yourenvironment) C: \Users $\backslash$ yourname>
```

For future usages, you can skip the conda env create command and just use the conda act ivat e command. As also indicated by the response, to go back to the default environment without exiting and restarting Anaconda, you would type in:

```
conda deactivate
```

After activating an environment, type spyder into the Anaconda Prompt command line to start up Spyder using that environment. Note that starting Spyder via the Windows search menu at this point will still run Spyder with the default packages and default Python version, not with the new environment, so you do need to start Spyder via the Anaconda Prompt. Also, even though you may activate an environment which includes various packages, when running Python code you still need to use import packagename-type commands in your Python file. The difference is that when an environment is active, Python will import the specified version of the package as opposed to just the default version.

If you would like to create your own environment, you can do so via the conda creat e command. As an example, to create an environment named "newenvironment" to run Python version 3.7.6 with the numpy, scipy, keras, tensorflow, and matplotlib packages, you would type the following into the command prompt:

```
conda create -n newenvironment python=3.7.6 numpy
    scipy keras tensorflow matplotlib
```

The -n option indicates that the next word will be the name of the environment. The above command will create a new folder in the

```
C:\Users\yourname\Anaconda3\envs
```

directory by the name of "newenvironment", and then that environment will thereafter be available for use via conda act ivate commands in the future. You might want to create an environment like this for yourself if, for example, you periodically want to use a package that requires a particular version of Python to run well. If you additionally need to specify a particular version of a package, you can do so by adding equals signs and numbers to the command, such as s cipy = 0.15.0 instead of just s c ipy. If you don't explicitly specify the version of a package, Anaconda will just use the latest version.

To additionally create an environment.yml file you can share with others, make sure the desired environment is active, then type this command:

```
conda env export > environment. yml
```

The environment.yml will be created in the working directory (overwriting any other environment.yml file which may be present, so be careful). Your colleagues will then be able to use that file to reproduce your environment on their own computers by running conda env create and conda activate commands as explained above.

Incidentally, we have created an environment specifically to assist with version conflicts in Lab 13 using a conda create command similar to the example above, and the environment.yml file is available for download on our website. It creates an environment called pythonML which should allow you to run all of the machine learning examples from Lab 12.

For more information on environments, see the documentation provided here.

APPENDIX B

GLOSSARY OF TERMS

Algorithm a set of steps used to solve a problem; frequently these are expressed in terms of a programming language.

Analytical approach finding an exact, algebraic solution to a differential equation.

Cell-center grid a grid with a point in the center of each cell. For instance, a cell-center grid ranging from 0 to 10 by steps of 1 would have points at $0.5, 1.5, \ldots, 8.5, 9.5$. Note that in this case, there are exactly as many points as cells (10 in this case).

Cell-center grid with ghost points the same as a regular cell-center grid, but with one additional point at each end of the grid (with the same spacing as the rest). A cell-center grid with ghost points ranging from 0 to 10 by steps of 1 would have points at $-0.5, 0.5, \ldots, 9.5, 10.5$. Note that in this case, there are two more points than cells. This arrangement is useful when setting conditions on the derivative at a boundary.

Cell-edge grid a grid with a point at the edge of each cell. For instance, a celledge grid ranging from 0 to 10 by steps of 1 would have points at $0, 1, \ldots, 9, 10$. Note that in this case, there are actually $N - 1$ cells (where N is the number of points: 11 in this case). The discrepancy between the number of cell and number of points is commonly called the "fence post problem" because in a straight-line fence there is one more post than spaces between posts.

Centered difference formula calculates derivatives on grids whereby the slope is calculated at a point centered between the two points where the function is evaluated. This is typically the best formula for the first derivative that uses only two values.

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Courant condition also called the Courant-Friedrichs-Lewy condition or the CFL condition, this condition gives the maximum time step for which an explicit algorithm remains stable. In the case of the Staggered-Leap Frog algorithm, the condition is $\tau > h/c$.

Crank Nicolson method an implicit algorithm for solving differential equations, e.g. the diffusion equation, developed by Phyllis Nicolson and John Crank. Dirichlet boundary conditions boundary conditions where a value is specified at each boundary, e.g. if $0 < x < 10$ and the function value at $x = 10$ is forced to be 40. Dispersion the spreading-out of waves Eigenvalue problem the linear algebra problem of finding a vector $\mathbf{g}$ that obeys $\mathbf{Ag} = \lambda \mathbf{g}$.

Explicit algorithm an algorithm that explicitly use past and present values to calculate future ones (often in an iterative process, where the future values become present ones and the process is repeated). Explicit algorithms are typically easier to implement, but more unstable than implicit algorithms. Extrapolation approximating the value of a function past the known range of values, using at least two nearby points.

Finite-difference method method that approximates the solution to a differential equation by replacing derivatives with equivalent difference equations. Forward difference formula a formula for calculating derivatives on grids whereby the slope is calculated at one of the points where the function is evaluated. This is typically less exact than the centered difference formula, although the two both become exact as h goes to zero.

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

Gauss-Seidel method a Successive Over-Relaxation technique with $w = 1$. Generalized eigenvalue problem the problem of finding a vector g that obeys $\mathbf{Ag} = \lambda \mathbf{Bg}$. Ghost points see "Cell-center grid with ghost points." Grid A division of either a spatial or temporal range (or both), used to numerically solve differential equations.

Implicit algorithm an algorithm that use present and future values to calculate future ones (often in a matrix equation, whereby all function points at a given time level are calculated at once). Implicit algorithms are typically more difficult to implement, but more stable than explicit algorithms. Interpolation approximating the value of a function somewhere between two known points, possibly using additional surrounding points.

Iteration repeating a calculation a number of times, usually until the error is smaller than a desired amount.

Korteweg-deVries Equation a differential equation that describes the motion of a soliton.

Lax-Wendroff algorithm an algorithm that uses a Taylor series in time to obtain a second-order accurate method

Neumann boundary conditions boundary conditions where the derivative of the function is specified at each boundary, e.g. if $0 < x < 10$ and at $x = 10$ the derivative dx/dt is forced to be 13 .

Resonance the point at which a system has a large steady-state amplitude with very little driving force.

Roundoff an error in accuracy that occurs when dealing with fixed-point arithmetic on computers. This can introduce very large errors when subtracting two numbers that are nearly equal.

Second derivative formula this is a centered formula for finding the second derivative on a grid:

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

Staggered-Leap Frog an algorithm for solving differential equations (e.g. the wave equation) whereby the value of a function one time step into the future is found current and past values of the function. It suffers from the difficulty at the start of needing to know the function value prior to the initial conditions.

Steady State solutions to differential equations in which the system reaches an equilibrium solution that continues on forever in the same fashion.

Successive Over-Relaxation A way to shift the eigenvalues. More specifically, it is using a multiplier w to shift the eigenvalues., with $w > 1$.

Successive Over-Relaxation (SOR) An algorithm for solving a linear system such as $\mathbf{V}^{\text{new}} = \mathbf{L} \times \mathbf{V}^{\text{old}} + \mathbf{r}$ by iterations, by shifting the eigenvalues. This can be done via solving the equivalent problem of

$$\mathbf{V}^{\text{new}} = w \times [\text{ RHS of previous equation }] + (1-w) \times \mathbf{V}^{\text{old}} .$$

For good choices of w, this can lead to much quicker convergence. Transients solutions to differential equations that are initially present but which quickly die out (due to damping), leaving only the steady-state behavior.