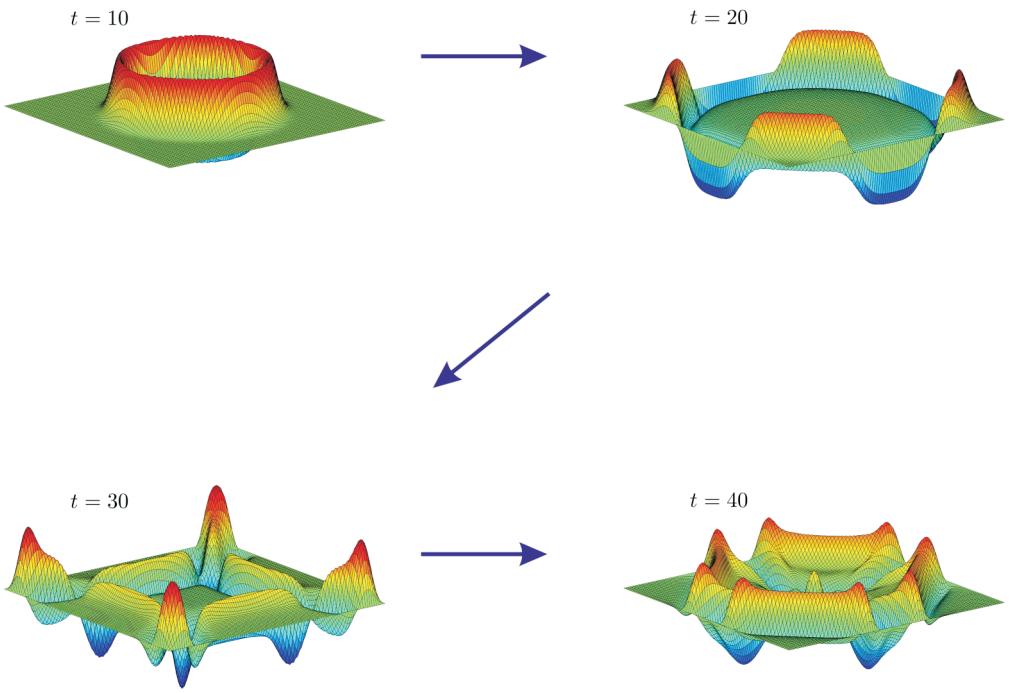
Computational Physics 430

Partial Differential Equations



Ross L. Spencer and Michael Ware

Department of Physics and Astronomy
Brigham Young University

COMPUTATIONAL PHYSICS 430

PARTIAL DIFFERENTIAL EQUATIONS

Ross L. Spencer and Michael Ware

Department of Physics and Astronomy
Brigham Young University
Last revised: January 9, 2019

© 2004–2019
Ross L. Spencer, Michael Ware, and Brigham Young University

This is a laboratory course about using computers to solve partial differential equations that occur in the study of electromagnetism, heat transfer, acoustics, and quantum mechanics. The course objectives are

- Solve physics problems involving partial differential equations numerically.
- Better be able to do general programming using loops, logic, etc.
- Have an increased conceptual understanding of the physical implications of important partial differential equations

You will need to read through each lab before class to complete the exercises during the class period. The labs are designed so that the exercises can be done in class (where you have access to someone who can help you) if you come prepared. Please work with a lab partner. It will take a lot longer to do these exercises if you are on your own. When you have completed a problem, call a TA over and explain to them what you have done.

To be successful in this class, you should already taken an introductory programming class in a standard language (e.g. C++), know how to program in Matlab, and be able to use a symbolic mathematical program such as Mathematica. We will be teaching you Python based on the assumption that you already know how to program in these other languages. We also assume that you have studied upper division mathematical physics (e.g. mathematical methods for solving partial differential equations with Fourier analysis).

Suggestions for improving this manual are welcome. Please direct them to Michael Ware (ware@byu.edu).

Contents

Review

If you are like most students, loops and logic gave you trouble in 330. We will be using these programming tools extensively this semester, so you may want to review and brush up your skills a bit. Here are some optional problems designed to help you remember your loops and logic skills. You will probably need to use online help (and you can ask a TA to explain things in class too).

(a) Write a `for` loop that counts by threes starting at 2 and ending at 101. Along the way, every time you encounter a multiple of 5 print a line that looks like this (in the printed line below it encountered the number 20.)

```
fiver: 20
```

You will need to use the commands `for`, `mod`, and `fprintf`, so first look them up in online help.

(b) Write a loop that sums the integers from 1 to N, where N is an integer value that the program receives via the `input` command. Verify by numerical experimentation that the formula

$$\sum_{n=1}^{N} n = \frac{N(N+1)}{2}$$

is correct

(c) For various values of x perform the sum

$$\sum_{n=1}^{1000} nx^n$$

with a `for` loop and verify by numerical experimentation that it only converges for $|x| < 1$ and that when it does converge, it converges to $x/(1-x)^2$.

(d) Redo (c) using a `while` loop (look it up in online help.) Make your own counter for n by using $n = 0$ outside the loop and $n = n + 1$ inside the loop. Have the loop execute until the current term in the sum, nx^n has dropped below 10^{-8}. Verify that this way of doing it agrees with what you found in (c).

(e) Verify by numerical experimentation with a `while` loop that

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

Set the `while` loop to quit when the next term added to the sum is below 10^{-6}.

(f) Verify, by numerically experimenting with a `for` loop that uses the `break` command (see online help) to jump out of the loop at the appropriate time, that the following infinite-product relation is true:

$$\prod_{n=1}^{\infty} \left(1 + \frac{1}{n^2}\right) = \frac{\sinh \pi}{\pi}$$

(g) Use a `while` loop to verify that the following three iteration processes converge. (Note that this kind of iteration is often called successive substitution.) Execute the loops until convergence at the 10^{-8} level is achieved.

$$x_{n+1} = e^{-x_n} \quad ; \quad x_{n+1} = \cos x_n \quad ; \quad x_{n+1} = \sin 2x_n$$

Note: iteration loops are easy to write. Just give x an initial value and then inside the loop replace x by the formula on the right-hand side of each of the equations above. To watch the process converge you will need to call the new value of x something like `xnew` so you can compare it to the previous x.

Finally, try iteration again on this problem:

$$x_{n+1} = \sin 3x_n$$

Convince yourself that this process isn't converging to anything. We will see in Lab 10 what makes the difference between an iteration process that converges and one that doesn't.

Lab 1

Grids and Numerical Derivatives

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 each lab, focusing on the particular ideas that you'll need to complete 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, how to declare and use entities called NumPy arrays, and also learn some basic plotting techniques.

With that Python knowledge under our belts, 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. And 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 $0 \le x \le L$. 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 *cell-edge* 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.

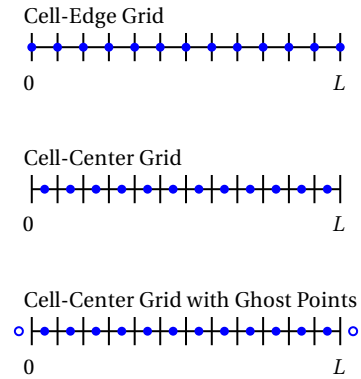


Figure 1.1 Three common spatial grids

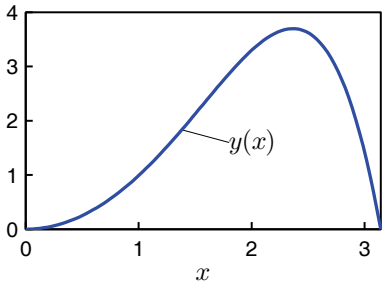


Figure 1.2 Plot from 1.2(a)

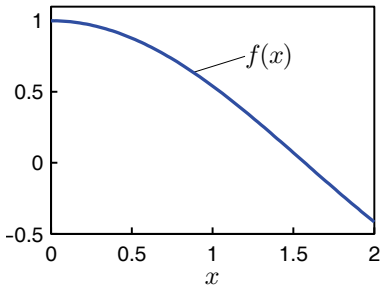


Figure 1.3 Plot from 1.2(b)

P1.2 (a) Write a Python program that creates a cell-edge spatial grid in the variable `x` as follows:

```
from numpy import linspace, pi

N=100   # the number of grid points
a=0
b=pi
x,h = 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 number of grid points 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 $0 \le x \le 2$.

Evaluate the function $f(x) = \cos x$ on this grid and plot this function. Then estimate the area under the curve by summing the products of the centered function values f_j with the widths of the cells h like this (midpoint integration rule):

```
sum(f)*h
```

Compare this result to the exact answer obtained by integration:

$$A = \int_0^2 \cos x \, dx = \sin(x)\Big|_0^2 = \sin(2)$$

(c) Build a cell-center grid with ghost points over the interval $0 \le x \le \pi/2$ with 500 cells (502 grid points), and evaluate the function $f(x) = \sin x$ 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 $\sin x$ has the property that $f(0) = 0$ and $f'(\pi/2) = 0$. 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 used in connection with interior points to specify both function-value boundary conditions and derivative-value boundary conditions.

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.

The simplest way to estimate function values is to use the fact that two points define a straight 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

$$y - y_1 = \frac{(y_2 - y_1)}{(x_2 - x_1)}(x - x_1) \tag{1.1}$$

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. (1.1) at x values between or beyond x_1 and x_2.

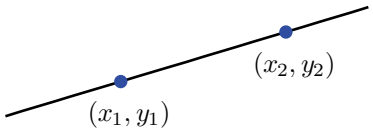


Figure 1.4 The line defined by two points can be used to interpolate between the points and extrapolate beyond the points.

P1.3 Use Eq. (1.1) 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?

(c) If the spacing between grid points is h (i.e. $x_2 - x_1 = h$), show that the linear extrapolation formula for $y(x_2 + h)$ is

$$y(x_2 + h) = 2y_2 - y_1 \tag{1.2}$$

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

$$y(x_2 + h/2) = 3y_2/2 - y_1/2. \tag{1.3}$$

We will use both of these formulas during the course.

Derivatives on grids

When solving 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

$$f'(x) \approx \frac{f(x + h) - f(x)}{h}. \tag{1.4}$$

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. (1.4) in the limit that h approaches zero. However, we can't make h arbitrarily small when we represent a function on a grid because (i) the number of cells needed to represent a region of space becomes infinite as h goes to zero; and (ii) computers represent numbers with a finite number of significant digits so the subtraction in the numerator of Eq. (1.4) 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 isn't one of them.

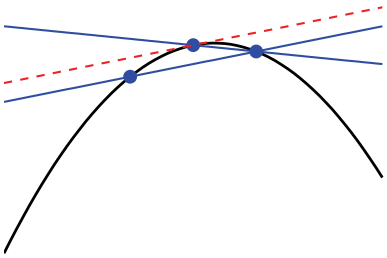


Figure 1.5 The forward and centered difference formulas both approximate the derivative as the slope of a line connecting two points. The centered difference formula gives a more accurate approximation because it uses points before and after the point where the derivative is being estimated. (The true derivative is the slope of the dotted tangent line).

The best first derivative formula that uses only two function values is usually the *centered difference* formula:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}. \tag{1.5}$$

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

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \tag{1.6}$$

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. (1.5) and (1.6) on a grid. 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 = numpy.zeros_like(f)
fp[1:N-2]=(f[2:N-1]-f[0:N-3])/(2*h)
```

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

```
fpp = numpy.zeros_like(f)
fpp[1:N-2]=(f[2:N-1]-2*f[1:N-2]+f[0:N-3])/h^2
```

The variable `h` is the spacing between grid points and `N` is the number of grid points. Study this code until you are convinced that it represents Eqs. (1.5) and (1.6) correctly.

The derivative at the first and last points on the grid can't be calculated using Eqs. (1.5) and (1.6) since there are not grid points on both sides of the endpoints. About the best we can do is to 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. (1.2):

```
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]
```

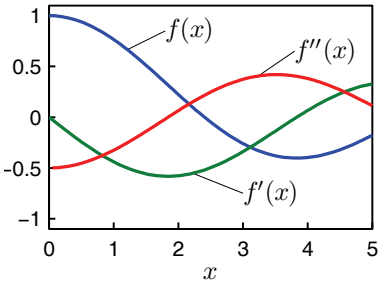


Figure 1.6 Plots from 1.4

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 $f''(x)$. Then make overlaid plots of the numerical derivatives with the exact derivatives:

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

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

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's expansion of the function f a small distance h away from the point x

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \cdots + f^{(n)}(x)\frac{h^n}{n!} + \cdots \tag{1.7}$$

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. (1.4), we get

$$\frac{f(x+h)-f(x)}{h} = \frac{\left[f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \cdots\right] - f(x)}{h} \tag{1.8}$$

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 assume to be small). If we neglect these higher order terms, we can solve Eq. (1.8) for the exact derivative $f'(x)$ to find

$$f'(x) \approx \frac{f(x+h)-f(x)}{h} - \frac{h}{2}f''(x) \tag{1.9}$$

From Eq. (1.9) 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 $f(x-h)$ in Eq. (1.5) to write

$$\begin{aligned}\frac{f(x+h)-f(x-h)}{2h} &= \frac{\left[f(x) + f'(x)h + f''(x)\frac{h^2}{2} + f'''(x)\frac{h^3}{6} + \cdots\right]}{2h} \\ &\quad - \frac{\left[f(x) - f'(x)h + f''(x)\frac{h^2}{2} - f'''(x)\frac{h^3}{6} + \cdots\right]}{2h}\end{aligned} \tag{1.10}$$

If we again neglect the higher-order terms, we can solve Eq. (1.10) for the exact derivative $f'(x)$. This time, we find that the f'' terms exactly cancel to give

$$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h} - \frac{h^2}{6}f'''(x) \tag{1.11}$$

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.

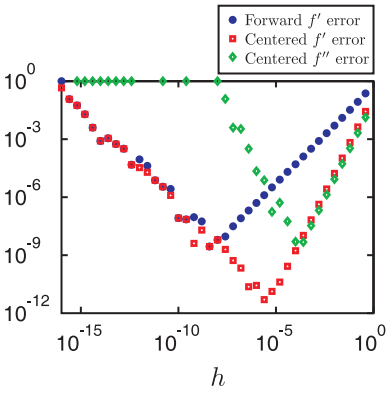


Figure 1.7 Error in the forward and centered difference approximations to the first derivative and the centered difference formula for the second derivative as a function of h. The function is e^x and the approximations are evaluated for $x = 0$.

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 $h = 0.1,\ 0.01,\ 0.001$. Also calculate the centered second derivative formula for these values of h. Verify that the error estimates in Eqs. (1.9) and (1.11) 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 should have found that $h = 0.001$ in the centered-difference formula gives a better approximation than $h = 0.01$. These errors are due to the finite grid spacing h, which 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. Finally (at about $h = 10^{-16}$, and sooner for the second derivative) the finite difference formulas have no accuracy at all—the error is the same order as the derivative.

The reason 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. This is normally plenty of precision, but look what happens in a subtraction problem where the two numbers are nearly the same:

$$\begin{array}{rr} & 7.38905699669556 \\ - & 7.38905699191745 \\ \hline & 0.00000000477811 \end{array} \tag{1.12}$$

Notice that our nice 15-digit accuracy has disappeared, leaving behind only 6 significant figures. This problem occurs in calculations with real 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. In a problem with a size of about L it doesn't do any good to use values of h any smaller than about $0.0001L$.

Lab 2

Differential Equations with Boundary Conditions

More Python

P2.1 Work through Chapter 2 of *Introduction to Python*.

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 flight 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 (like `ode45`) 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 differential equation solvers. Fortunately, there are better ways to solve these systems. In this section we develop a method for using a grid and the finite difference formulas we developed in Lab 1 to solve ordinary differential equations with linear algebra techniques.

Solving differential equations with linear algebra

Consider the differential equation

$$y''(x) + 9y(x) = \sin(x) \quad ; \quad y(0) = 0, \quad y(2) = 1 \tag{2.1}$$

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_j, with $j = 1..N$. We represent the (as yet unknown) function values $y(x_j)$ on our grid using the notation $y_j = y(x_j)$.

Now we can write the differential equation in finite difference form as it would appear on the grid. The second derivative in Eq. (2.1) is rewritten using the

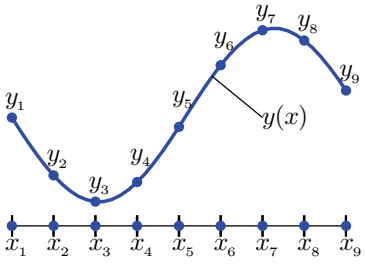


Figure 2.1 A function $y(x)$ repre-

centered difference formula (see Eq. (1.5)), so that the finite difference version of Eq. (2.1) becomes:

$$\frac{y_{j+1}-2y_j+y_{j-1}}{h^2}+9y_j=\sin(x_j) \tag{2.2}$$

Now let's think about Eq. (2.2) 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 j, except at $j=1$ and at $j=N$ where this formula reaches beyond the ends of the grid and cannot, therefore, be used. Because this equation involves y_{j-1}, y_j, and y_{j+1} for the interior grid points $j=2\ldots N-1$, Eq. (2.2) is really a system of $N-2$ coupled equations in the N unknowns $y_1\ldots y_N$. If we had just two more equations we could find the y_j's by solving a linear system of equations. But we do have two more equations; they are the boundary conditions:

$$y_1=0 \quad ; \quad y_N=1 \tag{2.3}$$

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

$$\mathbf{Ay}=\mathbf{b}, \tag{2.4}$$

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. (2.2). For the particular case of the system represented by Eqs. (2.2) and (2.3), the matrix equation is given by

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \ldots & 0 & 0 & 0 \\ \frac{1}{h^2} & -\frac{2}{h^2}+9 & \frac{1}{h^2} & 0 & \ldots & 0 & 0 & 0 \\ 0 & \frac{1}{h^2} & -\frac{2}{h^2}+9 & \frac{1}{h^2} & \ldots & 0 & 0 & 0 \\ . & . & . & . & \ldots & . & . & . \\ . & . & . & . & \ldots & . & . & . \\ . & . & . & . & \ldots & . & . & . \\ 0 & 0 & 0 & 0 & \ldots & \frac{1}{h^2} & -\frac{2}{h^2}+9 & \frac{1}{h^2} \\ 0 & 0 & 0 & 0 & \ldots & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ . \\ . \\ . \\ y_{N-1} \\ y_N \end{bmatrix} = \begin{bmatrix} 0 \\ \sin(x_2) \\ \sin(x_3) \\ . \\ . \\ . \\ \sin(x_{N-1}) \\ 1 \end{bmatrix}. \tag{2.5}$$

Convince yourself that Eq. (2.5) is equivalent to Eqs. (2.2) and (2.3) 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 unknown y-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_j. NumPy can do this solve with this command:

```
numpy.linalg.solve(a, b)
```

P2.2 (a) Set up a cell-edge grid with $N=30$ grid points, like this:

```
from numpy import linspace

N=30   # the number of grid points
```

```
a=0
b=2
x,h = linspace(a,b,N,retstep = True)
```

Look over this code and make sure you understand what it does. You may be wondering about the command `x=x'`. This turns the row vector `x` into a column vector `x`. This is not strictly necessary, but it is convenient because the `y` vector that we will get when we solve will be a column vector and Python needs the two vectors to be the same dimensions for plotting.

(b) Solve Eq. (2.1) symbolically using Mathematica's `DSolve` command. Then type the solution formula into the Python script that defines the grid above and plot the exact solution as a blue curve on a cell-edge grid with N points.

(c) Now load the matrix in Eq. (2.5) and do the linear solve to obtain y_j 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.

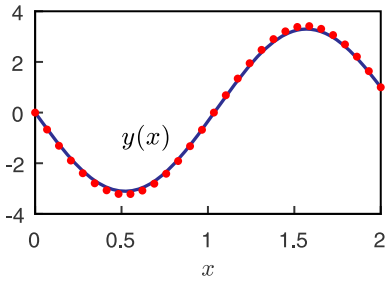


Figure 2.2 The solution to 2.2(c) with $N = 30$

Let's pause and take a minute to review how to apply the technique to solve a problem. First, write out the differential equation as a set of finite difference equations on a grid, similar to what we did in Eq. (2.2). Then translate this set of finite difference equations (plus the boundary conditions) into a matrix form analogous to Eq. (2.5). Finally, build the matrix **A** and the column vector **y** in Python and solve for the vector **y**. Our example, Eq. (2.1), had only a second derivative, but first derivatives can be handled using the centered first derivative approximation, Eq. (1.5).

Now let's practice this procedure for a couple more differential equations:

P2.3 (a) Write out the finite difference equations on paper for the differential equation

$$y'' + \frac{1}{x}y' + (1 - \frac{1}{x^2})y = x \quad ; \quad y(0) = 0, \quad y(5) = 1 \tag{2.6}$$

Then write down the matrix **A** and the vector **b** for this equation. Finally, build these matrices in a Python script 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.)

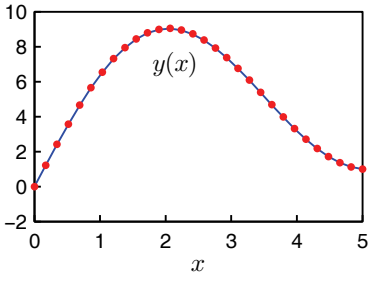


Figure 2.3 Solution to 2.3(a) with $N = 30$ (dots) compared to the exact solution (line)

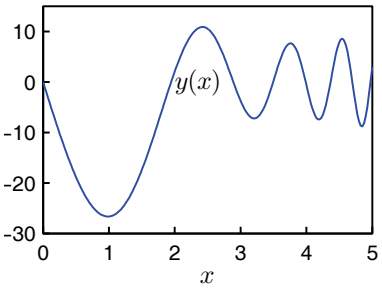


Figure 2.4 Solution to 2.3(b) with $N = 200$

(b) Solve the differential equation

$$y'' + \sin(x)\,y' + e^x y = x^2 \quad ; \quad y(0) = 0, \quad y(5) = 3 \tag{2.7}$$

in Python using the matrix method. Also solve this equation numerically using Mathematica's `NDSolve` command and plot the numeric solution. Compare the Mathematica plot with the Python plot. Do they agree? Check both solutions at $x = 4.5$; is the agreement reasonable? How many points do you have to use in your numerical method to get agreement with Mathematica to 3 decimal places?

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

$$y''(x) + 9y(x) = x \quad ; \quad y(0) = 0 \quad ; \quad y'(2) = 0 \tag{2.8}$$

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?

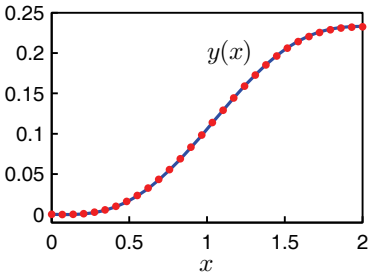


Figure 2.5 The solution to 2.4(a) with $N = 30$. The RMS difference from the exact solution is 8.8×10^{-4}

P2.4 (a) A crude way to implement the derivative boundary condition is to use a forward difference formula

$$\frac{y_N - y_{N-1}}{h} = y'\Big|_{x=2} \; . \tag{2.9}$$

In the present case, where $y'(2) = 0$, this simply means that we set $y_N = y_{N-1}$. Solve Eq. (2.8) in Python using the matrix method with this boundary condition. (Think about what the new boundary conditions will do to the final row of matrix **A** and the final element of vector **b**). Compare the resulting numerical solution to the exact solution obtained from Mathematica:

$$y(x) = \frac{x}{9} - \frac{\sin(3x)}{27\cos(6)} \tag{2.10}$$

(b) Let's improve the boundary condition formula using quadratic extrapolation. Use Mathematica to fit a parabola of the form

$$y(x) = a + bx + cx^2 \tag{2.11}$$

to the last three points on your grid. To do this, use (2.11) to write down three equations for the last three points on your grid and then solve these three equations for a, b, and c. Write the x-values in terms of the

last grid point and the grid spacing ($x_{N-2} = x_N - 2h$ and $x_{N-1} = x_N - h$) but keep separate variables for y_{N-2}, y_{N-1}, and y_N. (You can probably use your code from Problem **??** with a little modification.)

Now take the derivative of Eq. (2.11), evaluate it at $x = x_N$, and plug in your expressions for b and c. This gives you an approximation for the $y'(x)$ at the end of the grid. You should find that the new condition is

$$\frac{1}{2h}y_{N-2} - \frac{2}{h}y_{N-1} + \frac{3}{2h}y_N = y'(x_N) \tag{2.12}$$

Modify your script from part (a) to include this new condition and show that it gives a more accurate solution than the 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:

```
sqrt(mean((y-yexact).^2))
```

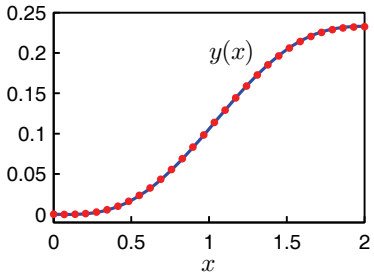


Figure 2.6 The solution to 2.4(b) with $N = 30$. The RMS difference from the exact solution is 5.4×10^{-4}

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.

P2.5 (a) Here is a simple example of a differential equation that isn't linear:

$$y''(x) + \sin\big[y(x)\big] = 1 \quad ; \quad y(0) = 0, \;\; y(3) = 0 \tag{2.13}$$

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) Find a way to use a combination of linear algebra and iteration (initial guess, refinement, etc.) to solve Eq. (2.13) in Python on a grid. Check your answer by using Mathematica's built-in solver to plot the solution.

HINT: Write the equation as

$$y''(x) = 1 - \sin\big[y(x)\big] \tag{2.14}$$

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

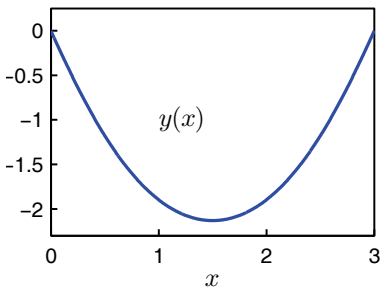


Figure 2.7 The solution to 2.5(b).

side of Eq. (2.14) 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. (2.14) 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 (2.13) to a specified criterion, *not* when the change in $y(x)$ from one iteration to the next falls below a certain level. An appropriate error vector would be `Ay-(1-sin(y)` evaluated at interior points only, `n=2:N-1`. This is necessary because the end points don't satisfy the differential equation.

Lab 3

The Wave Equation: Steady State and Resonance

To see why we did so much work in Lab 2 on ordinary differential equations when this is a course on partial differential equations, let's look at the wave equation in one dimension. 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

$$\mu \frac{\partial^2 y}{\partial t^2} = T \frac{\partial^2 y}{\partial x^2} + f(x)\cos\omega t \quad ; \quad y(0,t) = 0, \quad y(L,t) = 0 \tag{3.1}$$

where $y(x,t)$ is the (small) sideways displacement of the string as a function of position and time, assuming that $y(x,t) \ll L$. [1] This equation may look a little unfamiliar to you, so let's discuss each term. We have written it in the form of Newton's second law, $F = ma$. The "ma" part is on the left of Eq. (3.1), except that μ is not the mass, but rather the linear mass density (mass/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. (Take a minute and verify that this is true.) 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 ω is the frequency of the driving force.

Before we start 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 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. Now suppose that there was a little bit of damping in the system (not included in the equation above, but in Lab 5 we will add it). Then what would happen is that 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.)

Steady state solution

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

$$y(x,t) = g(x)\cos(\omega t) \tag{3.2}$$

[1] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 87-110.

This function has the expected form of a spatially dependent amplitude which oscillates at the frequency of the driving force. Substituting this "guess" into the wave equation to see if it works yields (after some rearrangement)

$$T g''(x) + \mu\omega^2 g(x) = -f(x) \quad ; \quad g(0) = 0, \quad g(L) = 0 \tag{3.3}$$

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.

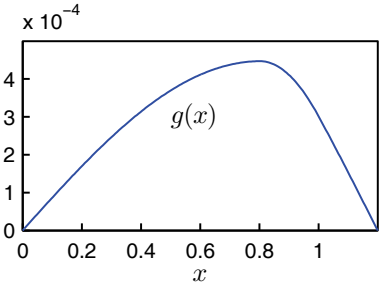


Figure 3.1 Solution to 3.1(a)

P3.1 (a) Modify one of your Python scripts from Lab 2 to solve Eq. (3.3) 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:

$$f(x) = \begin{cases} 0.73 & \text{if } 0.8 \le x \le 1 \\ \\ 0 & \text{if } x < 0.8 \text{ or } x > 1 \end{cases} \tag{3.4}$$

(b) Repeat the calculation in part (a) for 100 different frequencies between $\omega = 400$ and $\omega = 1200$ by putting a loop around your calculation in (a) that varies ω. Use this loop to load the maximum amplitude as a function of ω and plot it to see the resonance behavior of this system. Can you account qualitatively for the changes you see in $g(x)$ as ω varies? (Use a pause command after the plots of $g(x)$ and watch what happens as ω changes. Using `pause(.3)` will make an animation and using `ylim([0 0.03])` will prevent Python's autoscaling of plots from making all of the string responses from looking like they have the same amplitude.)

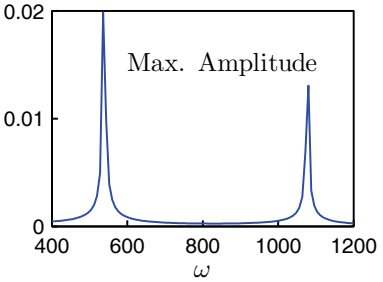


Figure 3.2 Solution to problem 3.1(b).

In problem 3.1(b) you should have noticed an apparent resonance behavior, with resonant frequencies near $\omega = 550$ and $\omega = 1100$ (see Fig. 3.2). Now we will learn how to use Python to find these resonant frequencies directly (i.e. without solving 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. (3.3) for which the driving force is zero. With $f(x) = 0$, Eq. (3.3) takes on the form

$$-\mu\omega^2 g(x) = T g''(x) \quad ; \quad g(0) = 0, \quad g(L) = 0 \tag{3.5}$$

If we rewrite this equation in the form

$$g''(x) = -\left(\frac{\mu\omega^2}{T}\right) g(x) \tag{3.6}$$

then we see that it is in the form of a classic eigenvalue problem:

$$Ag = \lambda g \tag{3.7}$$

where A is a linear operator (the second derivative on the left side of Eq. (3.6)) and λ is the eigenvalue ($-\mu\omega^2/T$ in Eq. (3.6).)

Equation (3.6) is easily solved analytically, and its solutions are just the familiar sine and cosine functions. The condition $g(0) = 0$ tells us to try a sine function form, $g(x) = g_0 \sin(kx)$. To see if this form works we substitute it into Eq. (3.6 and find that it does indeed work, provided that the constant k is $k = \omega\sqrt{\mu/T}$. We have, then,

$$g(x) = g_0 \sin\left(\omega\sqrt{\frac{\mu}{T}}x\right) \tag{3.8}$$

where g_0 is the arbitrary amplitude. But we still have one more condition to satisfy: $g(L) = 0$. This boundary condition tells us the values that resonance frequency ω can take on. When we apply the boundary condition, we find that the resonant frequencies of the string are given by

$$\omega = n\frac{\pi}{L}\sqrt{\frac{T}{\mu}} \tag{3.9}$$

where n is an integer. Each value of n gives a specific resonance frequency from Eq. (3.9) and a corresponding spatial amplitude $g(x)$ given by Eq. (3.8). Figure 3.3 shows photographs of a string vibrating for $n = 1, 2, 3$.

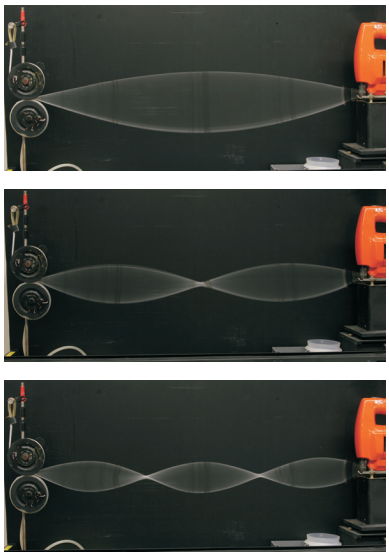


Figure 3.3 Photographs of the first three resonant modes for a string fixed at both ends.

For this simple example we were able to do the eigenvalue problem analytically without much trouble. However, when the differential equation is not so simple we will need to do the eigenvalue calculation numerically, so let's see how it works in this simple case. Rewriting Eq. (3.5) in matrix form, as we learned to do by finite differencing the second derivative, yields

$$\mathbf{Ag} = \lambda\mathbf{g} \tag{3.10}$$

which is written out as

$$\begin{bmatrix} ? & ? & ? & ? & \dots & ? & ? & ? \\ \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & \dots & 0 & 0 & 0 \\ 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & \dots & 0 & 0 & 0 \\ . & . & . & . & \dots & . & . & . \\ . & . & . & . & \dots & . & . & . \\ . & . & . & . & \dots & . & . & . \\ 0 & 0 & 0 & 0 & \dots & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} \\ ? & ? & ? & ? & \dots & ? & ? & ? \end{bmatrix} \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ . \\ . \\ . \\ g_{N-1} \\ g_N \end{bmatrix} = \lambda \begin{bmatrix} ? \\ g_2 \\ g_3 \\ . \\ . \\ . \\ g_{N-1} \\ ? \end{bmatrix} \tag{3.11}$$

where $\lambda = -\omega^2\frac{\mu}{T}$. The question marks in the first and last rows remind us that we have to invent something to put in these rows that will implement the correct

boundary conditions. Note that having question marks in the g-vector on the right is a real problem because without g_1 and g_N in the top and bottom positions, we don't have an eigenvalue problem (i.e. the vector **g** on left side of Eq. (3.11) is not the same as the vector **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. (3.7) to the slightly more complicated form of a *generalized eigenvalue problem*, like this:

$$\mathbf{Ag} = \lambda \mathbf{Bg} \tag{3.12}$$

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. (3.11) with **B** and the top and bottom rows of **A** chosen to apply the boundary conditions $g(0) = 0$ and $g(L) = 0$.

$$\begin{array}{cccccc} & & & \mathbf{A} & & \end{array} \quad \begin{array}{c} \mathbf{g} \\ \end{array} \quad = \lambda \quad \begin{array}{cccccc} & & & \mathbf{B} & & \end{array} \quad \begin{array}{c} \mathbf{g} \\ \end{array}$$

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & \dots & 0 & 0 \\ 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \dots & 0 & 0 \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ 0 & 0 & 0 & \dots & -\frac{2}{h^2} & \frac{1}{h^2} \\ 0 & 0 & 0 & \dots & 0 & 1 \end{bmatrix} \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ . \\ . \\ . \\ g_{N-1} \\ g_N \end{bmatrix} = \lambda \begin{bmatrix} 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ 0 & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 \end{bmatrix} \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ . \\ . \\ . \\ g_{N-1} \\ g_N \end{bmatrix} \tag{3.13}$$

Notice that the matrix **B** is very simple: it is just the identity matrix (made in Python with `eye(N,N)`) except that the first and last rows are completely filled with zeros. Take a minute now and do the matrix multiplications corresponding the first and last rows and verify that they correctly give $g_1 = 0$ and $g_N = 0$, no matter what the eigenvalue λ turns out to be.

To numerically solve this eigenvalue problem you simply do the following in Python:

(i) Load the matrix **A** with the matrix on the left side of Eq. (3.13) and the matrix **B** with the matrix on the right side.

(ii) Use Python's generalized eigenvalue and eigenvector command:

```
[V,D]=eig(A,B);
```

which returns the eigenvalues as the diagonal entries of the square matrix `D` and the eigenvectors as the columns of the square matrix `V` (these column arrays are the amplitude functions $g_j = g(x_j)$ associated with each eigenvalue on the grid x_j.)

(iii) Convert eigenvalues to frequencies via $\omega^2 = -\frac{T}{\mu}\lambda$, sort the squared frequencies in ascending order, and plot each eigenvector with its associated frequency displayed in the plot title.

This is such a common calculation that we will give you a section of a Matlab script below that does steps (ii) and (iii). You can get this code snippet on the Physics 430 web site so you don't have to retype it.

Listing 3.1 (eigen.m)

```
[V,D]=eig(A,B);  % find the eigenvectors and eigenvalues

w2raw=-(T/mu)*diag(D);  % convert lambda to omega^2

[w2,k]=sort(w2raw);  % sort omega^2 into ascending along with a
                     % sort key k(n) that remembers where each
                     % omega^2 came from so we can plot the proper
                     % eigenvector in V

for n=1:N  % run through the sorted list and plot each eigenvector
   % load the plot title into t
   t=sprintf(' w^2 = %g w = %g ',w2(n),sqrt(abs(w2(n))) );
   gn=V(:,k(n)); % extract the eigenvector
   plot(x,gn,'b-');  % plot the eigenvector that goes with omega^2
   title(t);xlabel('x');ylabel('g(n,x)');  % label the graph
   pause
end
```

P3.2 (a) Use Matlab to numerically find the eigenvalues and eigenvectors of Eq. (3.5) using the procedure outlined above. Use $\mu = 0.003$, $T = 127$, and $L = 1.2$. Note that there is a pause command in the code, so you'll need to hit a key to step to the next eigenvector. When you plot the eigenvectors, you will see that two infinite eigenvalues appear 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. You will also see that 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. (3.5).

(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. (3.8). Calculate the exact values for ω using Eq. (3.9) 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 two values for ω should match the resonances that you found in 3.1(b). Go back to your calculation in 3.1(b) and make two plots

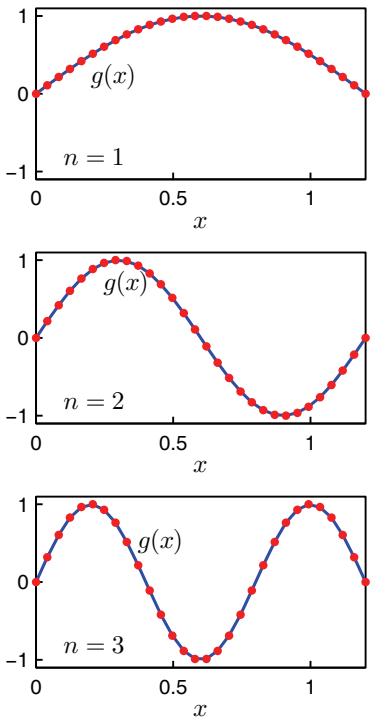


Figure 3.4 The first three eigenfunctions found in 3.2. The points are the numerical eigenfunctions and the line is the exact solution.

of the steady state amplitude for driving frequencies near these two resonant values of ω. (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.

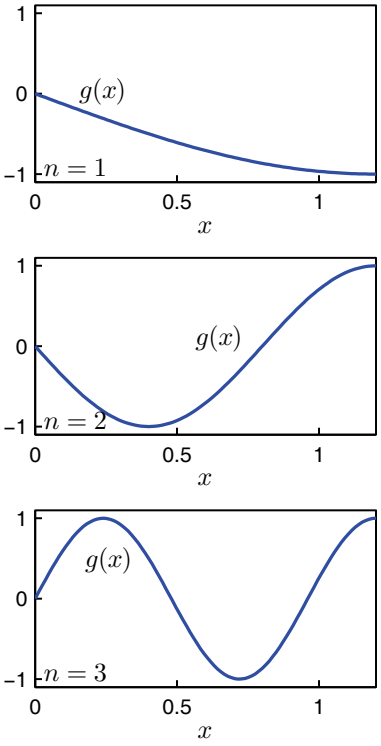


Figure 3.5 The first three eigenfunctions for 3.3(a).

P3.3 (a) Change your program from problem 3.2 to implement the boundary condition

$$g'(L) = 0$$

Use the approximation you derived in problem 2.4(b) for the derivative $g'(L)$ to implement this boundary condition, i.e.

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

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 this nice smooth eigenfunction is actually unphysical in our wave resonance problem with this boundary condition. The `sqrt(abs(w2(n)))` command in the code snippet we gave you misleads you in this case–be careful.

Lab 4

The Hanging Chain and Quantum Bound States

The resonance modes that we studied in Lab 3 were simply sine functions. We can also use these techniques to analyze more complicated systems. In this lab we first study 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 also solved in the 1700s 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. Then we will jump forward several centuries in physics history and study bound quantum states using the same techniques.

Resonance for a hanging chain

Consider the chain hanging from the ceiling in the classroom.[1] We are going to find its normal modes of vibration using the method of Problem 3.2. The wave equation for transverse waves on a chain with varying tension $T(x)$ and constant linear mass density μ[2] is given by

$$\mu\frac{\partial^2 y}{\partial t^2} - \frac{\partial}{\partial x}\left(T(x)\frac{\partial y}{\partial x}\right) = 0 \tag{4.1}$$

Let's use a coordinate system that starts at the bottom of the chain at $x = 0$ and ends on the ceiling at $x = L$.

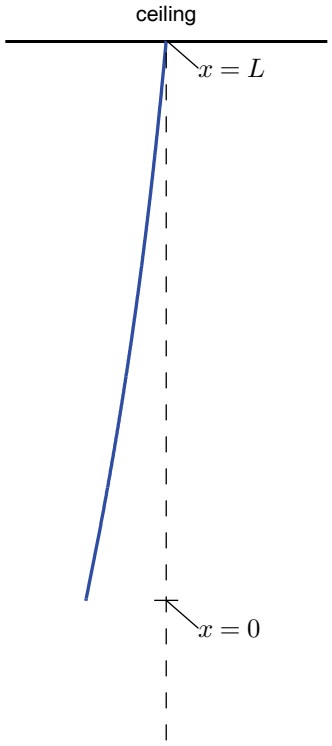


Figure 4.1 The first normal mode for a hanging chain.

P4.1 Use the fact that the stationary chain is in vertical equilibrium to help you draw a carefully chosen free-body diagram that shows that the tension in the chain as a function of x is given by

$$T(x) = \mu g x \tag{4.2}$$

where μ is the linear mass density of the chain and where $g = 9.8\ \text{m/s}^2$ is the acceleration of gravity. Then show that Eq. (4.1) reduces to

$$\frac{\partial^2 y}{\partial t^2} - g\frac{\partial}{\partial x}\left(x\frac{\partial y}{\partial x}\right) = 0 \tag{4.3}$$

for a freely hanging chain.

[1] For more analysis of the hanging chain, see N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 299-305.

[2] Equation (4.1) also works for systems with varying mass density if you replace μ with a function $\mu(x)$, but the equations derived later in the lab are more complicated with a varying $\mu(x)$.

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 (4.3) and simplify to obtain

$$x\frac{d^2f}{dx^2} + \frac{df}{dx} = -\frac{\omega^2}{g}f \tag{4.4}$$

which is in eigenvalue form with $\lambda = -\omega^2/g$. (This relationship is different than in the preceding lab; consequently you will have to change a line in the `eigen.m` code to reflect this difference.)

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. (4.4) as $x \to 0$ to find

$$f'(0) = -\frac{\omega^2}{g}f(0) = \lambda f(0) \tag{4.5}$$

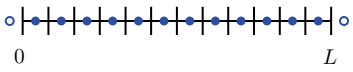


Figure 4.2 A cell-centered grid with ghost points. (The open circles are the ghost points.)

In the past couple labs we have 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 (see Problems 2.4(b) and 3.3). 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 computing 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 $x_1 = -h/2$ and the other at $x_{N+2} = L + h/2$. The ghost points are used to apply the boundary conditions. Notice that 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. You can do it any way you like, as long as your counting method works.

Notice that there isn't a grid point at either endpoint, but rather that the two grid points on each end straddle the endpoints. If the boundary condition specifies a value, like $f(L) = 0$ in the problem at hand, we use a simple average like this:

$$\frac{f_{N+2} + f_{N+1}}{2} = 0 \ , \tag{4.6}$$

and if the condition were $f'(L) = 0$ we would use

$$\frac{f_{N+2} - f_{N+1}}{h} = 0 \ . \tag{4.7}$$

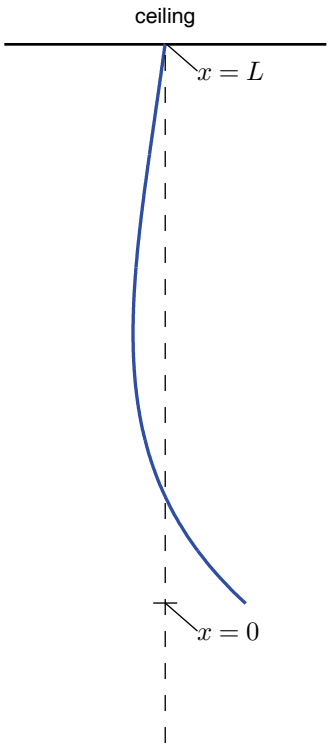


Figure 4.3 The shape of the second mode of a hanging chain

When we did boundary conditions in the eigenvalue calculation of Problem 3.2 we used a **B** matrix with zeros in the top and bottom rows and we loaded the top and bottom rows of **A** with an appropriate boundary condition operator. Because the chain is fixed at the ceiling ($x = L$) we use this technique again in the bottom rows of **A** and **B**, like this (after first loading **A** with zeros and **B** with the identity matrix):

$$A(N+2,N+1) = \frac{1}{2} \quad A(N+2,N+2) = \frac{1}{2} \quad B(N+2,N+2) = 0 \tag{4.8}$$

P4.2 (a) Verify that these choices for the bottom rows of A and B in the generalized eigenvalue problem

$$\mathbf{A}f = \lambda \mathbf{B}f \tag{4.9}$$

give the boundary condition in Eq. (4.6) at the ceiling no matter what λ turns out to be.

(b) Now let's do something similar with the derivative boundary condition at the bottom, Eq. (4.5). Since this condition is already in eigenvalue form we don't need to load the top row of **B** with zeros. Instead we load **A** with the operator on the left ($f'(0)$) and **B** with the operator on the right ($f(0)$), leaving the eigenvalue $\lambda = -\omega^2/g$ out of the operators so that we still have $\mathbf{A}f = \lambda \mathbf{B}f$. Verify that the following choices for the top rows of **A** and **B** correctly produce Eq. (4.5).

$$A(1,1) = -\frac{1}{h} \quad A(1,2) = \frac{1}{h} \quad B(1,1) = \frac{1}{2} \quad B(1,2) = \frac{1}{2} \tag{4.10}$$

(c) Write the finite difference form of Eq. (4.4) and use it to load the matrices **A** and **B** for a chain with $L = 2$ m. (Notice that for the interior points the matrix **B** is just the identity matrix with 1 on the main diagonal and zeros everywhere else.) Use Matlab to solve for the normal modes of vibration of a hanging chain. As in Lab 3, some of the eigenvectors are unphysical because they don't 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) Solve Eq. (4.4) analytically using Mathematica without any boundary conditions. You will encounter the Bessel functions J_0 and Y_0, but because Y_0 is singular at $x = 0$ this function is not allowed in the problem. Apply the condition $f(L) = 0$ to find analytically the mode frequencies ω and verify that they agree with the frequencies you found in part (c).

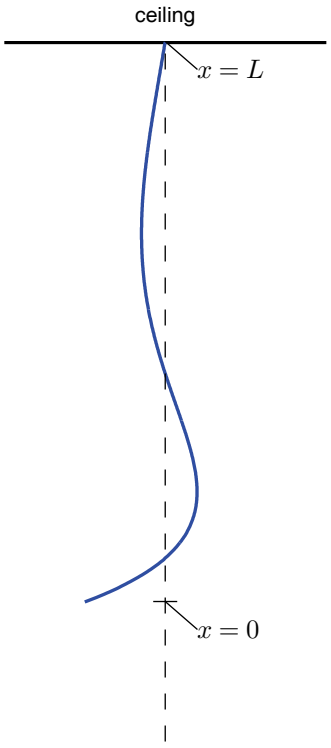


Figure 4.4 The shape of the third mode of a hanging chain

Quantum bound states

Consider the problem of a particle in a one-dimensional harmonic oscillator well in quantum mechanics.[3] Schrödinger's equation for the bound states in this well is

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + \frac{1}{2}kx^2\psi = E\psi \tag{4.11}$$

with boundary conditions $\psi = 0$ at $\pm\infty$. Note that k in this equation is not the wave number; it is the spring constant, $F = -kx$, with units of Newtons/meter.

[3] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 470-506.

The numbers that go into Schrödinger's equation are so small that it makes it difficult to tell what size of grid to use. For instance, our usual trick of using lengths like 2, 5, or 10 would be completely ridiculous for the bound states of an atom where the typical size is on the order of 10^{-10} m. We could just set $\hbar$, m, and k to unity, but then we wouldn't know what physical situation our numerical results describe. When computational physicists encounter this problem a common thing to do is to "rescale" the problem so that all of the small numbers go away. And, as an added bonus, this procedure can also allow the numerical results obtained to be used no matter what m and k our system has.

P4.3 This probably seems a little nebulous, so follow the recipe below to see how to rescale in this problem (write it out on paper).

(i) In Schrödinger's equation use the substitution $x = a\xi$, where a has units of length and ξ is dimensionless. After making this substitution put the left side of Schrödinger's equation in the form

$$C\left(-\frac{D}{2}\frac{d^2\psi}{d\xi^2}+\frac{1}{2}\xi^2\psi\right)=E\psi \tag{4.12}$$

where C and D involve the factors $\hbar$, m, k, and a.

(ii) Make the differential operator inside the parentheses (...) on the left be as simple as possible by choosing to make $D = 1$. This determines how the characteristic length a depends on $\hbar$, m, and k. Once you have determined a in this way, check to see that it has units of length. You should find

$$a=\left(\frac{\hbar^2}{km}\right)^{1/4}=\sqrt{\frac{\hbar}{m\omega}} \quad \text{where} \quad \omega=\sqrt{\frac{k}{m}} \tag{4.13}$$

(iii) Now rescale the energy by writing $E = \epsilon\bar{E}$, where $\bar{E}$ has units of energy and ϵ is dimensionless. Show that if you choose $\bar{E} = C$ in the form you found above in (i) that Schrödinger's equation for the bound states in this new dimensionless form is

$$-\frac{1}{2}\frac{d^2\psi}{d\xi^2}+\frac{1}{2}\xi^2\psi=\epsilon\psi \tag{4.14}$$

You should find that

$$\bar{E}=\hbar\sqrt{\frac{k}{m}} \tag{4.15}$$

Verify that $\bar{E}$ has units of energy.

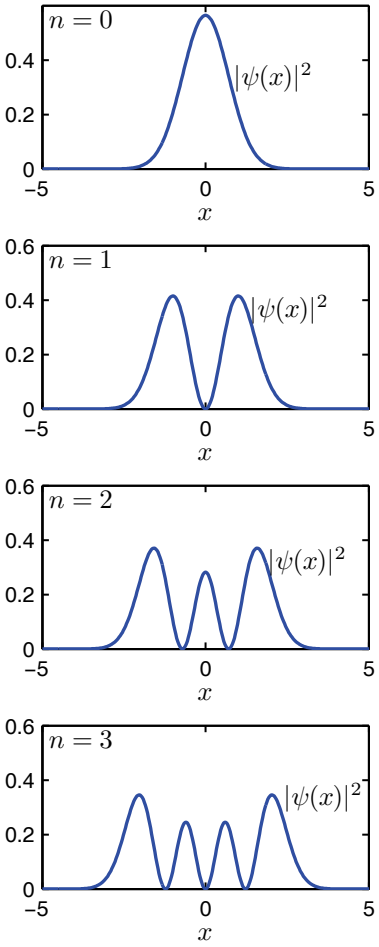


Figure 4.5 The probability distributions for the ground state and the first three excited states of the harmonic oscillator.

Now that Schrödinger's equation is in dimensionless form it makes sense to choose a grid that goes from -4 to 4, or some other similar pair of numbers. These numbers are supposed to approximate infinity in this problem, so make sure (by looking at the eigenfunctions) that they are large enough that the wave function

goes to zero with zero slope at the edges of the grid. As a guide to what you should find, Figure 4.5 displays the square of the wave function for the first few excited states. (The amplitude has been appropriately normalized so that $\int |\psi(x)|^2 = 1$

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

$$E_n = (n + \frac{1}{2})\hbar\sqrt{\frac{k}{m}} = (n + \frac{1}{2})\bar{E} \tag{4.16}$$

so that the dimensionless energy eigenvalues ϵ_n are given by

$$\epsilon_n = n + \frac{1}{2} \tag{4.17}$$

P4.4 Use Matlab's ability to do eigenvalue problems to verify that this formula for the bound state energies is correct for $n = 0, 1, 2, 3, 4$.

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

$$V(x) = \mu x^4 \tag{4.18}$$

so that we have

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + \mu x^4\psi = E\psi \tag{4.19}$$

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

$$-\frac{1}{2}\frac{d^2\psi}{d\xi^2} + \xi^4\psi = \epsilon\psi \tag{4.20}$$

with $E = \epsilon\bar{E}$. Use Mathematica and/or algebra by hand to show that

$$a = \left(\frac{\hbar^2}{m\mu}\right)^{1/6} \qquad \bar{E} = \left(\frac{\hbar^4\mu}{m^2}\right)^{1/3} \tag{4.21}$$

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

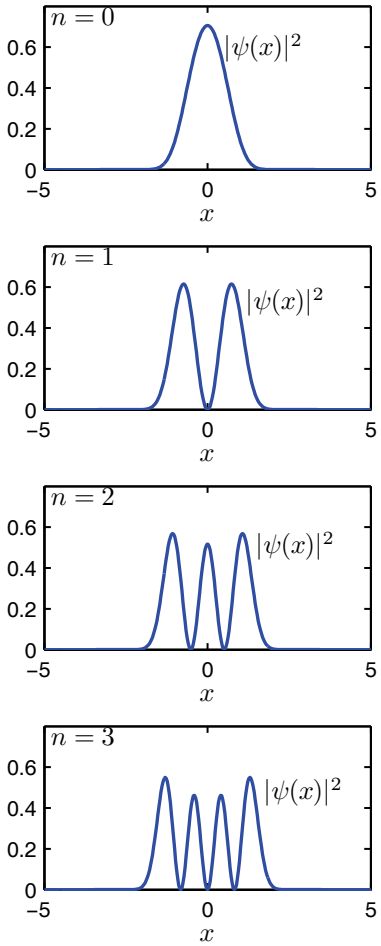


Figure 4.6 The probability distributions for the ground state and the first three excited states for the potential in Problem 4.5.

Lab 5

Animating the Wave Equation: Staggered Leapfrog

Labs 3 and 4 should have seemed pretty familiar, since they handled the wave equation by Fourier analysis, turning the partial differential equation into a set of ordinary differential equations, as you learned in Mathematical Physics.[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. And as an added attraction, this method automatically supplies a beautiful animation of the solution. We will only show you one of 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. (For instance, for waves on a string $c = \sqrt{T/\mu}$.)

$$\frac{\partial^2 y}{\partial t^2} - c^2 \frac{\partial^2 y}{\partial x^2} = 0 \tag{5.1}$$

The boundary conditions to be applied are usually either of *Dirichlet* type (values specified):

$$y(0,t) = f_{\text{left}}(t) \quad ; \quad y(L,t) = f_{\text{right}}(t) \tag{5.2}$$

or of *Neumann* type (derivatives specified):

$$\frac{\partial y}{\partial x}(0) = g_{\text{left}}(t) \quad ; \quad \frac{\partial y}{\partial x}(L) = g_{\text{right}}(t) \tag{5.3}$$

Some mixed boundary conditions specify a relation between the value and derivative (as at the bottom of the hanging chain). These conditions tell us what is happening at the ends of the string. For example, maybe the ends are pinned ($f_{\text{left}}(t) = f_{\text{right}}(t) = 0$); perhaps the ends slide up and down on frictionless rings attached to frictionless rods ($g_{\text{left}}(t) = g_{\text{right}}(t) = 0$); or perhaps the left end is fixed and someone is wiggling the right end up and down sinusoidally ($f_{\text{left}}(t) = 0$ and $f_{\text{right}}(t) = A\sin\omega t$). In any case, some set of conditions at the ends are required to be able to solve the wave equation.

[1] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 87-110.

It is also necessary to specify the initial state of the string, giving its starting position and velocity as a function of position:

$$y(x, t=0) = y_0(x) \quad ; \quad \frac{\partial y(x,t)}{\partial t}|_{t=0} = v_0(x) \tag{5.4}$$

Both of these initial conditions are necessary because the wave equation is second order in time, just like Newton's second law, so initial displacements and velocities must be specified to find a unique solution.

To numerically solve the classical wave equation via staggered leapfrog we approximate both the time and spatial derivatives with centered finite differences. In the notation below 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 steps 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.

$$\frac{\partial^2 y}{\partial t^2} \approx \frac{y_j^{n+1} - 2y_j^n + y_j^{n-1}}{\tau^2} \tag{5.5}$$

$$\frac{\partial^2 y}{\partial x^2} \approx \frac{y_{j+1}^n - 2y_j^n + y_{j-1}^n}{h^2} \tag{5.6}$$

The staggered leapfrog algorithm is simply a way of finding y_j^{n+1} (y_j one time step into the future) from the current and previous values of y_j. To derive the algorithm just put these two approximations into the classical wave equation and solve for y_j^{n+1}: [2]

$$y_j^{n+1} = 2y_j^n - y_j^{n-1} + \frac{c^2\tau^2}{h^2}\left(y_{j+1}^n - 2y_j^n + y_{j-1}^n\right) \tag{5.7}$$

P5.1 Derive Eq. (5.7) from the approximate second derivative formulas. (You can use mathematica if you like, but this is really simple to do by hand.)

Equation (5.7) 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 (Dirichlet) and derivative (Neumann) 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 (Dirichlet boundary conditions) we have

$$\frac{y_1^{n+1} + y_2^{n+1}}{2} = f_{\text{left}}(t_{n+1}) \quad \Rightarrow \quad y_1^{n+1} = -y_2^{n+1} + 2f_{\text{left}}(t_{n+1}) \tag{5.8}$$

$$\frac{y_{N+2}^{n+1} + y_{N+1}^{n+1}}{2} = f_{\text{right}}(t_{n+1}) \quad \Rightarrow \quad y_{N+2}^{n+1} = -y_{N+1}^{n+1} + 2f_{\text{right}}(t_{n+1}) \tag{5.9}$$

[2] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 421-429.

If the derivatives are specified (Neumann boundary conditions) then we have

$$\frac{y_2^{n+1} - y_1^{n+1}}{h} = g_{\text{left}}(t_{n+1}) \quad \Rightarrow \quad y_1^{n+1} = y_2^{n+1} - h g_{\text{left}}(t_{n+1}) \tag{5.10}$$

$$\frac{y_{N+2}^{n+1} - y_{N+1}^{n+1}}{h} = g_{\text{right}}(t_{n+1}) \quad \Rightarrow \quad y_{N+2}^{n+1} = y_{N+1}^{n+1} + h g_{\text{right}}(t_{n+1}) \tag{5.11}$$

To use staggered leapfrog, we first advance the solution at all interior points to the next time step using Eq. (5.7), then we apply the boundary conditions using the appropriate equation from Eqs. (5.8)-(5.11) 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. (5.7) requires not just y at the current time level y_j^n but also y at the previous time level y_j^{n-1}. 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. (5.4) into

$$\frac{y_j^1 - y_j^{-1}}{2\tau} = v_0(x_j) \tag{5.12}$$

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. (5.7) to obtain another relation between y_j^1 and y_j^{-1}. Leapfrog at the first step ($n = 0$) says that

$$y_j^1 = 2y_j^0 - y_j^{-1} + \frac{c^2\tau^2}{h^2}\left(y_{j+1}^0 - 2y_j^0 + y_{j-1}^0\right) \tag{5.13}$$

If we insert this expression for y_j^1 into Eq. (5.12), we can solve for y_j^{-1} in terms of known quantities:

$$y_j^{-1} = y_j^0 - v_0(x_j)\tau + \frac{c^2\tau^2}{2h^2}\left(y_{j+1}^0 - 2y_j^0 + y_{j-1}^0\right) \tag{5.14}$$

P5.2 Derive Eq. (5.14) from Eqs. (5.12) and (5.13).

OK; we are now ready to code. We will give you a template below with some code in it and also with some empty spaces you have to fill in using the formulas above. The dots indicate where you are supposed to write your own code.

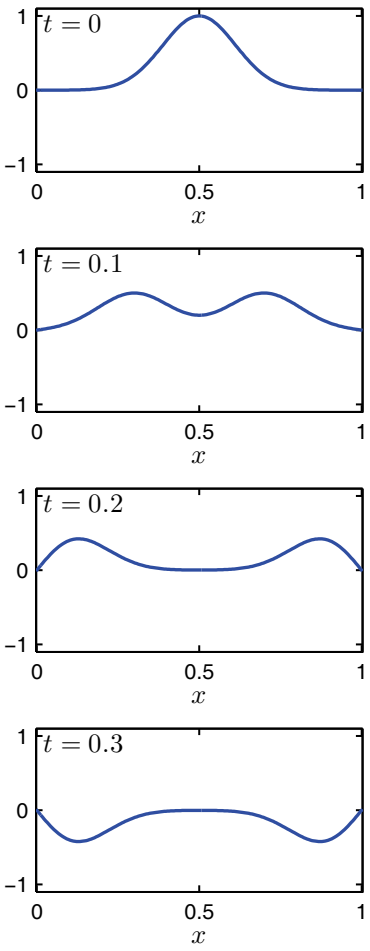


Figure 5.1 Snapshots of the evolution of a wave on a string with fixed ends and an initial displacement but no initial velocity. (See Problem 5.3(a))

Listing 5.1 (stlf.m)

```
% Staggered Leapfrog Script Template
clear;close all;

% Set the values for parameters
c=2; % wave speed

% build a cell-centered grid with N=200 cells
% on the interval x=0 to x=L, with L=1
. . .

% define the initial displacement and velocity vs. x
y = exp(-(x-L/2).^2*160/L^2)-exp(-(0-L/2).^2*160/L^2); vy =
zeros(1,length(x));

% Choose a time step (Suggest that it is no larger than taulim)
taulim=h/c;
fprintf(' Courant time step limit %g \n',taulim)
tau=input(' Enter the time step - ')

% Get the initial value of yold from the initial y and vy
. . .

% Apply the boundary conditions for yold(1) and yold(N+2)
. . .

% plot the initial conditions and pause to look at them
subplot(2,1,1) plot(x,y) xlabel('x');ylabel('y(x,0)');title('Initial
Displacement') subplot(2,1,2) plot(x,vy)
xlabel('x');ylabel('v_y(x,0)');title('Initial Velocity') pause;

% Choose how long to run and when to plot
tfinal=input(' Enter tfinal - ') skip=input(' Enter # of steps to skip
between plots (faster) - ') nsteps=tfinal/tau;

% here is the loop that steps the solution along

figure  % open a new frame for the animation
for n=1:nsteps
   time=n*tau;  % compute the time

   % Use leapfrog and the boundary conditions to load ynew with y
   % at the next time step using y and yold, i.e., ynew(2:N+1)=...
```

```
   % Be sure to use colon commands so it will run fast.
. . .

   %update yold and y
   yold=y;y=ynew;

% make plots every skip time steps
   if mod(n,skip)==0
      plot(x,y,'b-')
      xlabel('x');ylabel('y');
      title(['Staggered Leapfrog Wave: time=' num2str(time)])
      axis([min(x) max(x) -2 2]);
      pause(.1)
   end
end
```

P5.3 (a) Fill in the missing code for the `stlf.m` template. You can make your code read more cleanly by defining a variable `j` like this

```
j = 2:N+1;
```

Then you can write the array y_j^n as `y(j)`, the array y_{j-1}^n as `y(j-1)`, and the array y_{j+1}^n as `y(j+1)`. Use fixed-end boundary conditions for the guitar string:

$$y(0) = 0 \quad ; \quad y(L) = 0$$

When you are finished you should be able to run, debug, then successfully run an animation of a guitar string with no initial velocity and an initial upward displacement localized near the center of the string. (Look at the initial conditions plot to see what they look like.)

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 *Courant-Friedrichs-Lewy condition*, or sometimes the *CFL condition*, or sometimes (unfairly) just the *Courant condition*.

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.

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

(c) 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

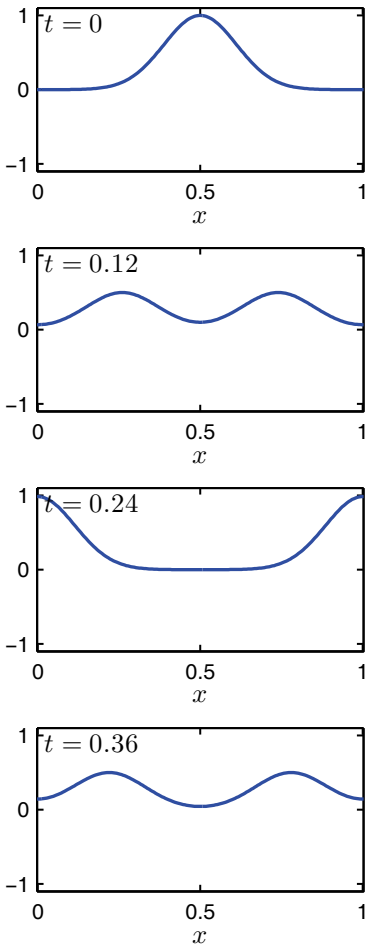
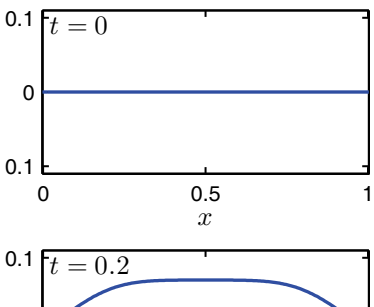


Figure 5.2 Snapshots of the evolution of a wave on a string with free ends and an initial displacement but no initial velocity. (See



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:

$$\frac{\partial^2 y}{\partial t^2} + \gamma \frac{\partial y}{\partial t} - c^2 \frac{\partial^2 y}{\partial x^2} = 0 \tag{5.15}$$

with c constant. The staggered leapfrog method can be used to solve Eq. (5.15) also. To do this, we use the approximate first derivative formula

$$\frac{\partial y}{\partial t} \approx \frac{y_j^{n+1} - y_j^{n-1}}{2\tau} \tag{5.16}$$

along with the second derivative formulas in Eqs. (5.5) and (5.6) and find an expression for the values one step in the future:

$$y_j^{n+1} = \frac{1}{2+\gamma\tau}\left(4y_j^n - 2y_j^{n-1} + \gamma\tau y_j^{n-1} + \frac{2c^2\tau^2}{h^2}\left(y_{j+1}^n - 2y_j^n + y_{j-1}^n\right)\right) \tag{5.17}$$

P5.4 (a) Derive Eq. (5.17).

(b) Find a new formula for the initial value of `yold` using Eqs. (5.12) and (5.17). 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 $\gamma = 0.2$. 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

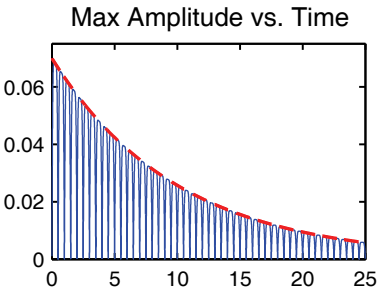


Figure 5.5 The maximum amplitude of oscillation decays exponentially for the damped wave equation. (Problem 5.4(c))

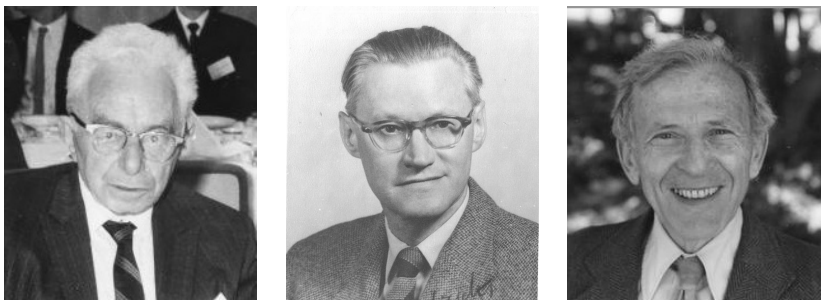


Figure 5.3 Richard Courant (left), Kurt Friedrichs (center), and Hans Lewy (right) described the CFL instability condition in 1928.

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 γ. The decay of a simple harmonic oscillator is exponential, with amplitude proportional to $e^{-\gamma t/2}$. 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.)

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

$$\frac{\partial^2 y}{\partial t^2} + \gamma \frac{\partial y}{\partial t} - c^2 \frac{\partial^2 y}{\partial x^2} = \frac{f(x)}{\mu} \cos(\omega t) \tag{5.18}$$

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 Re-derive the staggered leapfrog algorithm to include both driving and damping forces as in Eq. (5.18). 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 y_{old}. Just set $y = 0$ and $y_{\text{old}} = 0$ 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 3.1(a)

$$f(x) = \begin{cases} 0.73 & \text{if } 0.8 \le x \le 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.19}$$

and set the driving frequency at $\omega = 400$. Choose a damping constant γ 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

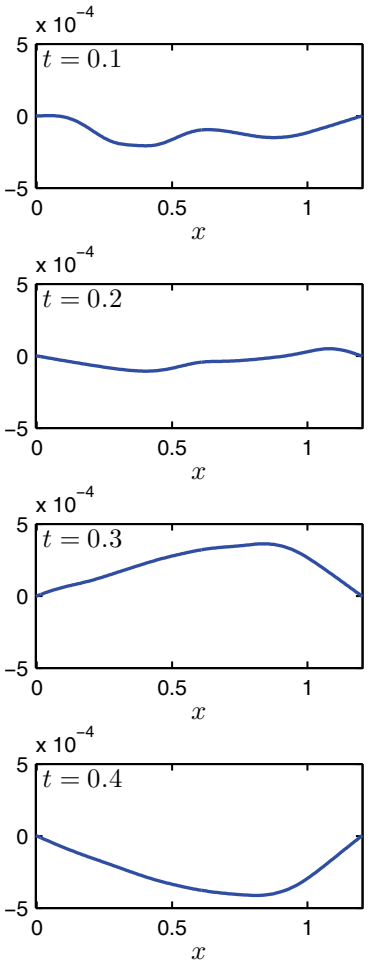


Figure 5.6 Snapshots of the evolution a driven and damped wave with $\omega = 400$. As the transient behavior dies out, the oscillation goes to the resonant mode. To make the pictures more interesting, the string was not started from rest in these plots. (In Problem 5.5 you start from rest for

that you are using and about your damping rate result from problem 5.4 to decide which value of γ 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 -1 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 3.1(a) (see Fig. 3.1).

Then run again with $\omega = 1080$, which is close to a resonance (see Fig. 3.2), and again see the system come into steady oscillation at the driving frequency.

Lab 6

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.

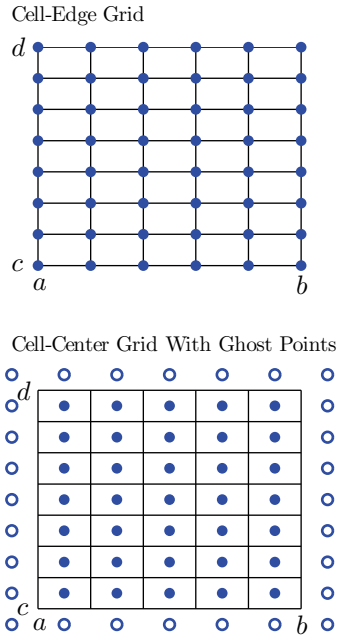


Figure 6.1 Two types of 2-D grids.

P6.1 Matlab has a nice way of representing rectangular two-dimensional grids using the `ndgrid` command. Let's take a minute to remember how to make surface plots using this command. Consider a 2-d rectangle defined by $x \in [a,b]$ and $y \in [c,d]$. To create a 30-point cell-edge grid in x and a 50-point cell-edge grid in y with $a = 0$, $b = 2$, $c = -1$, $d = 3$, we use the following code:

```
Nx=30; a=0; b=2; dx=(b-a)/(Nx-1); x=a:dx:b;

Ny=50; c=-1; d=3; dy=(d-c)/(Ny-1); y=c:dy:d;

[X,Y]=ndgrid(x,y);
```

(a) Put the code fragment above in a script and run it to create the 2-D grid. Examine the contents of `X` and `Y` thoroughly enough that you can explain what the `ndgrid` command does. See the first section of chapter 6 in *Introduction to Matlab* from physics 330 for help understanding the indexing.

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

$$f(x,y) = e^{(-(x^2+y^2))} \cos\left(5\sqrt{x^2+y^2}\right) \tag{6.1}$$

HINT: Remember to use 'dotted' operators with `X` and `Y` (not `x` and `y`) when you build $f(x,y)$.

Use Matlab's `surf` command to make a surface plot of this function. Compare these three ways of using `surf`:

```
surf(f) surf(X,Y,f) surf(x,y,f)
```

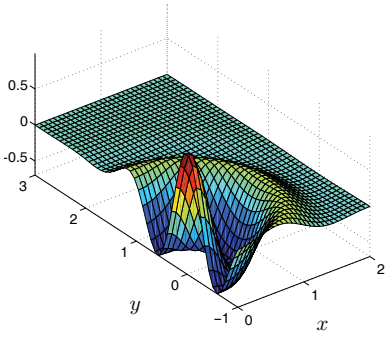


Figure 6.2 Plot from Problem 6.1

and explain what is different between the three forms. Then properly label the x and y axes with the symbols x and y, to get a plot like Fig. 6.2. As you do this lab and the ones that follow, use whichever form is convenient. `surf(f)` is nice if you want to see your data displayed in grid coordinates (good for debugging). The other two forms are pretty much equivalent.

The two-dimensional wave equation

The wave equation for transverse waves on a rubber sheet is [1]

$$\mu \frac{\partial^2 z}{\partial t^2} = \sigma \left(\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} \right) \tag{6.2}$$

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 σ 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 $z^n_{j,k} = z(x_j, y_k, t_n)$ to represent the function values. With this notation, the derivatives can be approximated as

$$\frac{\partial^2 z}{\partial t^2} \approx \frac{z^{n+1}_{j,k} - 2z^n_{j,k} + z^{n-1}_{j,k}}{\tau^2} \tag{6.3}$$

$$\frac{\partial^2 z}{\partial x^2} \approx \frac{z^n_{j+1,k} - 2z^n_{j,k} + z^n_{j-1,k}}{h_x^2} \tag{6.4}$$

$$\frac{\partial^2 z}{\partial y^2} \approx \frac{z^n_{j,k+1} - 2z^n_{j,k} + z^n_{j,k-1}}{h_y^2} \tag{6.5}$$

where h_x and h_y are the grid spacings in the x and y dimensions. We insert these three equations into Eq. (6.2) to get an expression that we can solve for z at the next time (i.e. $z^{n+1}_{j,k}$). Then we use this expression along with the discrete version of the initial velocity condition

$$v_0(x_j, y_k) \approx \frac{z^{n+1}_{j,k} - z^{n-1}_{j,k}}{2\tau} \tag{6.6}$$

[1] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 129-134.

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 Matlab script that animates the solution of the two dimensional wave equation on a square region that is $[-5,5] \times [-5,5]$ and that has fixed edges. Use a cell-edge square grid with the edge-values pinned to zero to enforce the boundary condition. Choose $\sigma = 2$ N/m and $\mu = 0.3$ kg/m^2 and use a displacement initial condition that is a Gaussian pulse with zero velocity

$$z(x, y, 0) = e^{-5(x^2+y^2)} \tag{6.7}$$

This initial condition doesn't strictly satisfy the boundary conditions, so you should pin the edges to zero.

Run the simulation long enough that you see the effect of repeated reflections from the edges. You may get annoyed that the colors on the plot keep changing. You can stop this by using the command

```
caxis([-0.25 0.25])
```

after your `surf` command to fix the range of values that the colors represent.

(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:

$$\tau < f\frac{h}{c} \tag{6.8}$$

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 in two (and higher) even 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. (6.7). 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.

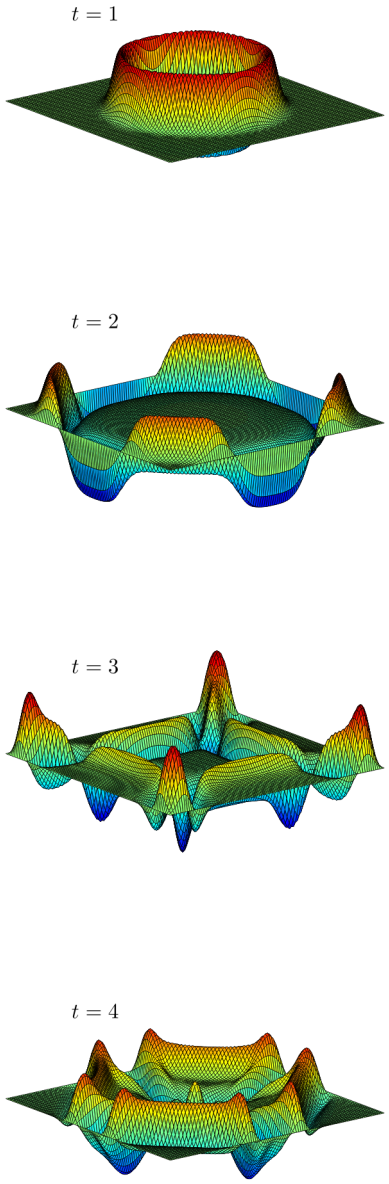


Figure 6.3 A wave on a rubber sheet with fixed edges.

Elliptic, hyperbolic, and parabolic PDEs and their boundary conditions

Now let's take a step back and look at some general concepts related to solving partial differential equations. Probably the three most famous PDEs of classical physics are

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

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = \frac{-\rho}{\epsilon_0} \quad + \text{ Boundary Conditions} \tag{6.9}$$

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

$$\frac{\partial^2 y}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 y}{\partial t^2} = 0 \quad + \text{ Boundary Conditions} \tag{6.10}$$

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

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2} \quad + \text{ Boundary Conditions} \tag{6.11}$$

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:

$$\text{ellipse}: \quad \frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \tag{6.12}$$

$$\text{hyperbola}: \quad \frac{x^2}{a^2} - \frac{y^2}{b^2} = 1 \tag{6.13}$$

$$\text{parabola}: \quad y = ax^2 \tag{6.14}$$

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. Fortunately, because you are physicists and have developed some intuition about the physics of these three partial differential equations, you can remember the proper boundary conditions by thinking about physical examples instead of memorizing theorems.

And in case you haven't developed this level of intuition, here is a brief review of the matter.

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. Since we will be talking about time-dependence in the hyperbolic and parabolic cases, 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 end conditions at the two ends of the string, but it makes no sense to specify conditions at both $t = 0$ and $t = t_{\text{final}}$ 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. (6.11), 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:

$$i\hbar\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\psi}{\partial x^2} + V\psi \tag{6.15}$$

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?

$$\frac{\partial^2 y}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 y}{\partial t^2} - \frac{1}{D}\frac{\partial y}{\partial t} = 0 \tag{6.16}$$

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.

Lab 7

The Diffusion, or Heat, Equation

Now let's attack the diffusion equation [1]

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2} \quad . \tag{7.1}$$

This equation describes how the distribution T (often associated with temperature) diffuses through a material with diffusion coefficient D. We'll study the diffusion equation analytically first and then we'll look at how to solve it numerically.

Analytic approach to the diffusion equation

The diffusion equation can be approached analytically by assuming that T is of the form $T(x,t) = g(x)f(t)$. If we plug this form into the diffusion equation, we can use separation of variables to find that $g(x)$ must satisfy

$$g''(x) + a^2 g(x) = 0 \tag{7.2}$$

where a is a separation constant. If we specify the boundary conditions that $T(x=0,t) = 0$ and $T(x=L,t) = 0$ then the solution to Eq. (7.2) is simply $g(x) = \sin(ax)$ 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.

P7.1 (a) Find how an initial temperature distribution $T_n(x,0) = T_0 \sin(n\pi x/L)$ decays with time, by substituting the form $T(x,t) = T(x,0)f(t)$ into the diffusion equation and finding $f_n(t)$ for each integer n. Do long wavelengths or short wavelengths decay more quickly?

(b) Separation of variables is not the only way to find an analytic solution to the diffusion equation. Show that an initial Gaussian temperature distribution like this

$$T(x) = T_0 e^{-(x-L/2)^2/\sigma^2} \tag{7.3}$$

decays according to the formula

$$T(x,t) = \frac{T_0}{\sqrt{1+4Dt/\sigma^2}} e^{-(x-L/2)^2/(\sigma^2+4Dt)} \tag{7.4}$$

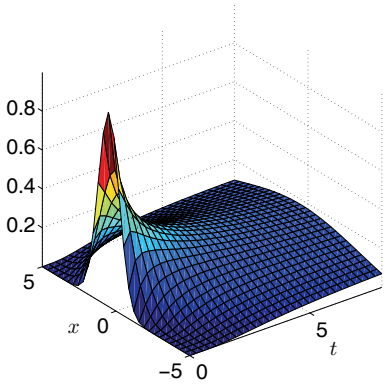


Figure 7.1 Diffusion of the Gaussian temperature distribution given in Problem 7.1(b) with $\sigma = 1$ and $D = 1$.

[1] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 110-129.

by showing that this expression satisfies the diffusion equation Eq. (7.1) and the initial condition. (It doesn't satisfy finite boundary conditions, however; it is zero at $\pm\infty$.) Use Mathematica.

(c) Show by using dimensional arguments that the approximate time it takes for the distribution in Eq. (7.4) to increase its width by a distance a must be on the order of $t = a^2/D$. Also argue that if you wait time t, then the distance the width should increase by must be about $a = \sqrt{Dt}$. (The two arguments are really identical.)

(d) Show that Eq. (7.4) agrees with your dimensional analysis by finding the time it takes for the $t = 0$ width of the Gaussian (σ) to increase to 2σ (look in the exponential in Eq. (7.4).)

Numerical approach: a first try

Now let's try to solve the diffusion equation numerically on a grid as we did with the wave equation. It is similar to the wave equation in that it requires boundary conditions at the ends of the computing interval in x. But because its time derivative is only first order we only need to know the initial distribution of T. This means that the trouble with the initial distribution of $\partial T/\partial t$ that we encountered with the wave equation is avoided. But in spite of this simplification, the diffusion equation is actually more difficult to solve numerically than the wave equation.

If we finite difference the diffusion equation using a centered time derivative and a centered second derivative in x to obtain an algorithm that is similar to leapfrog then we would have

$$\frac{T_j^{n+1} - T_j^{n-1}}{2\tau} = \frac{D}{h^2}\left(T_{j+1}^n - 2T_j^n + T_{j-1}^n\right) \tag{7.5}$$

$$T_j^{n+1} = T_j^{n-1} + \frac{2D\tau}{h^2}\left(T_{j+1}^n - 2T_j^n + T_{j-1}^n\right) \tag{7.6}$$

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 if we work around this problem this algorithm turns out to be worthless because no matter how small a time step τ we choose, we encounter the same kind of instability that plagues staggered leapfrog (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. It seems almost intuitively obvious that making an algorithm more accurate is better, but in this case the increased accuracy achieved by using a centered time derivative leads to numerical instability.

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

$$\frac{T_j^{n+1} - T_j^n}{\tau} = \frac{D}{h^2}\left(T_{j+1}^n - 2T_j^n + T_{j-1}^n\right) \tag{7.7}$$

$$T_j^{n+1} = T_j^n + \frac{D\tau}{h^2}\left(T_{j+1}^n - 2T_j^n + T_{j-1}^n\right) \tag{7.8}$$

You might expect this algorithm to have problems since the left side of Eq. (7.7) is centered at time $t_{n+\frac{1}{2}}$, but the right side is centered at time t_n. This problem makes the algorithm inaccurate, but it turns out that it is stable if τ is small enough. In the next lab we'll learn how to get a stable algorithm with both sides of the equation centered on the same time, but for now let's use this inaccurate (but stable) method.[2]

P7.2 (a) Modify one of your staggered leapfrog programs that uses a cell-center grid to implement this algorithm to solve the diffusion equation on the interval $[0, L]$ with initial distribution

$$T(x,0) = \sin(\pi x/L) \tag{7.9}$$

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. 7.2. 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

$$\tau \le C\frac{h^2}{D} \tag{7.10}$$

Determine the value of C by numerical experimentation.

Test the accuracy of your numerical solution by overlaying a graph of the exact solution found in 7.1(a). 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 error using something like

```
error = mean( abs( y - exact ) )
```

(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) Verify your answer to the question in problem 7.1(a) about the decay rate of long versus short wavelengths by trying initial distributions of $T(x,0) = \sin(2\pi x/L)$, $T(x,0) = \sin(3\pi x/L)$, $T(x,0) = \sin(4\pi x/L)$, etc. and comparing decay rates.

[2] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 412-421.

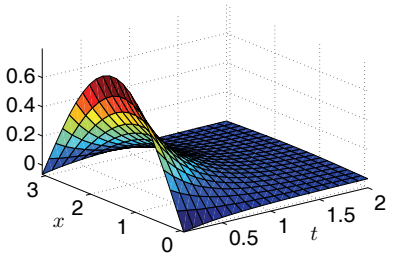


Figure 7.2 Diffusion of the $n = 1$ sine temperature distribution given in Problem 7.2(a).

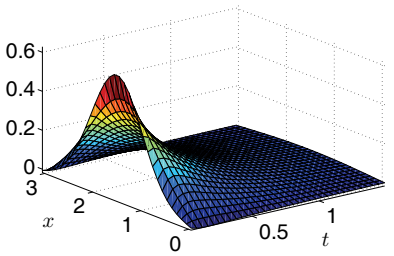


Figure 7.3 Diffusion of the Gaussian temperature distribution given in Problem 7.2(d) with fixed T boundary conditions.

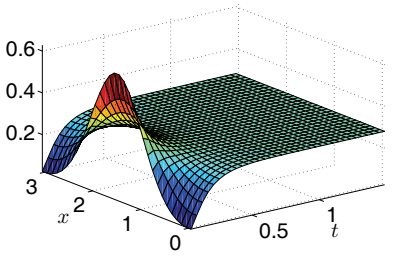


Figure 7.4 Diffusion of the Gaussian temperature distribution

(d) Now use as an initial condition a Gaussian distribution centered at $x = L/2$:

$$T(x,0) = e^{-40(x/L-1/2)^2} \tag{7.11}$$

Use two different kinds of boundary conditions:

(i) $T = 0$ at both ends and

(ii) $\partial T/\partial x = 0$ at both ends.

Explain what these 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 on the watermelon and explain what the temperature history of the watermelon has to do with your plots of $T(x)$ vs. time.

(Notice how in case (i) the distribution that started out Gaussian quickly becomes very much like the $n = 1$ sine wave. This is because all of the higher harmonics die out rapidly.)

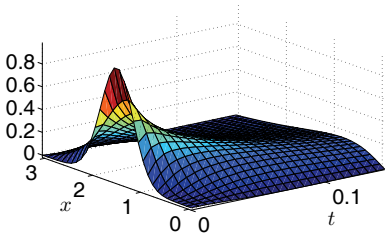


Figure 7.5 Diffusion of an initial Gaussian with D as in Problem 7.2(e).

(e) Modify your program to handle a diffusion coefficient which varies spatially like this:

$$D(x) = D_0 \frac{x^2 + L/5}{(L/5)} \tag{7.12}$$

with $D_0 = 2$. Note that in this case the diffusion equation is

$$\frac{\partial T}{\partial t} = \frac{\partial}{\partial x}\left(D(x)\frac{\partial T}{\partial x}\right) \tag{7.13}$$

Use the two different boundary conditions of part (d) and discuss why $T(x,t)$ behaves as it does in this case.

Even though this technique can give us OK results, the time step constraint for this method is pretty extreme. The constraint is of the form $\tau < Bh^2$, where B is a constant, and this limitation scales horribly with h. 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 script take forever to run, which is usually intolerable. In the next lab we'll learn a better way.

Lab 8

Implicit Methods: the Crank-Nicolson Algorithm

You may have noticed that all of 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, and as we saw in the last lab, even when they aren't they can 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.

Implicit methods

To give you a better feel for what "implicit" means, let's study the simple first-order differential equation

$$\dot{y} = -\gamma y \tag{8.1}$$

P8.1 (a) Solve this equation using Euler's method:

$$\frac{y_{n+1} - y_n}{\tau} = -\gamma y_n \,. \tag{8.2}$$

Show by writing a simple Matlab script and doing numerical experimentation that Euler's method is unstable for large τ. Show by experimenting and by looking at the algorithm that it is unstable if $\tau > 2/\gamma$. Use $y(0) = 1$ as your initial condition. This is an example of an explicit method.

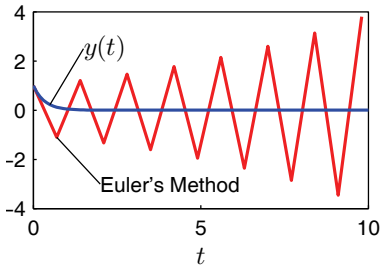


Figure 8.1 Euler's method is unstable for $\tau > 2/\gamma$. ($\tau = 2.1/\gamma$ in this case.)

(b) Notice that the left side of Eq. (8.2) is centered on time $t_{n+\frac{1}{2}}$ but the right side is centered on t_n. Let's center the the right-hand side at time $t_{n+\frac{1}{2}}$ by using an average of the advanced and current values of y,

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

Show by numerical experimentation in a modified script 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.

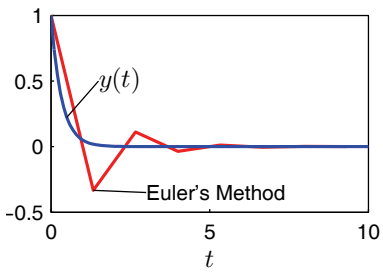


Figure 8.2 The implicit method in 8.1(b) with $\tau = 4/\gamma$.

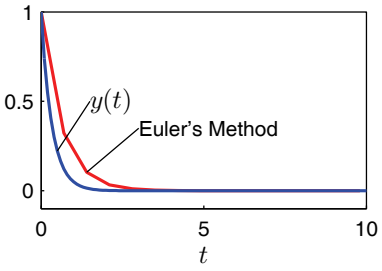


Figure 8.3 The fully implicit method in 8.1(c) with $\tau = 2.1/\gamma$.

(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. (8.2) (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 script that this fully implicit method damps even when tau is large. For instance, see what happens if you choose $\gamma = 1$ and $\tau = 5$ 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. Just to make things more interesting we'll let the diffusion coefficient be a function of x:

$$\frac{\partial T}{\partial t} = \frac{\partial}{\partial x}\left(D(x)\frac{\partial T}{\partial x}\right) \tag{8.3}$$

We begin by finite differencing the right side, taking care to handle the spatial dependence of D. Rather than expanding the nested derivatives in Eq. (8.3) let's difference it in place on the grid. In the equation below $D_{j\pm\frac{1}{2}} = D(x_j \pm h/2)$.

$$\frac{\partial T_j}{\partial t} = \frac{D_{j+\frac{1}{2}}\left(T_{j+1} - T_j\right) - D_{j-\frac{1}{2}}\left(T_j - T_{j-1}\right)}{h^2} \tag{8.4}$$

P8.2 Show that the right side of Eq. (8.4) is correct by finding a centered difference expression for $D(x)\frac{\partial T}{\partial x}$ at $x_{j-\frac{1}{2}}$ and $x_{j+\frac{1}{2}}$. Then use these two expressions to find a centered difference formula for the entire expression at x_j. Draw a picture of a grid showing x_{j-1}, $x_{j-\frac{1}{2}}$, x_j, $x_{j+\frac{1}{2}}$, and x_{j+1} and show that this form is centered properly.

Now we take care of the time derivative by doing something similar to problem 8.1(b): we take a forward time derivative on the left, putting that side of the equation at time level $n+\frac{1}{2}$. 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

$$T^{n+\frac{1}{2}} = \frac{T^{n+1} + T^n}{2} \tag{8.5}$$

like this:

$$\frac{T_j^{n+1}-T_j^n}{\tau}=\frac{D_{j+\frac{1}{2}}\left(T_{j+1}^{n+1}-T_j^{n+1}+T_{j+1}^n-T_j^n\right)-D_{j-\frac{1}{2}}\left(T_j^{n+1}-T_{j-1}^{n+1}+T_j^n-T_{j-1}^n\right)}{2h^2} \tag{8.6}$$

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 can't 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.

$$-D_{j-\frac{1}{2}}T_{j-1}^{n+1}+\left(\frac{2h^2}{\tau}+D_{j+\frac{1}{2}}+D_{j-\frac{1}{2}}\right)T_j^{n+1}-D_{j+\frac{1}{2}}T_{j+1}^{n+1}= D_{j-\frac{1}{2}}T_{j-1}^n+\left(\frac{2h^2}{\tau}-D_{j+\frac{1}{2}}-D_{j-\frac{1}{2}}\right)T_j^n+D_{j+\frac{1}{2}}T_{j+1}^n \tag{8.7}$$

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 *Crank-Nicolson algorithm*.

Phyllis Nicolson (1917–1968, English)

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. (8.7) and another matrix **B** to describe the right side, like this:

$$\mathbf{A}T^{n+1}=\mathbf{B}T^n \tag{8.8}$$

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

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

$$A_{j,j-1}=-D_{j-\frac{1}{2}} \;\;;\;\; A_{j,j}=\frac{2h^2}{\tau}+(D_{j+\frac{1}{2}}+D_{j-\frac{1}{2}}) \;\;;\;\; A_{j,j+1}=-D_{j+\frac{1}{2}} \tag{8.9}$$

and the elements of **B** are given by

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

$$B_{j,j-1}=D_{j-\frac{1}{2}} \;\;;\;\; B_{j,j}=\frac{2h^2}{\tau}-(D_{j+\frac{1}{2}}+D_{j-1/2}) \;\;;\;\; B_{j,j+1}=D_{j+\frac{1}{2}} \tag{8.10}$$

John Crank (1916–2006, English)

Once the boundary conditions are added to these matrices, Eq. (8.8) can easily be solved symbolically to find T^{n+1}

$$T^{n+1}=\mathbf{A}^{-1}\mathbf{B}T^n\,. \tag{8.11}$$

However, since inverting a matrix is computationally expensive we will use Gauss elimination instead when we actually implement this in Matlab, as we did in lab 2 (see Matlab help on the \ operator). Here is a sketch of how you would implement the Crank-Nicolson algorithm in Matlab.

(i) Load the matrices **A** and **B** as given in Eq. (8.9) and Eq. (8.10) above for all of the rows except the first and last. As usual, the first and last rows involve the boundary conditions. Usually it is a little easier to handle the boundary conditions if we plan to do the linear solve in two steps, like this:

```
% compute the right-hand side of the equation
r=B*T;

% load r(1) and r(N+2) as appropriate
% for the boundary conditions

r(1)=...;r(N+2)=...;

% load the new T directly into T itself
T=A\r;
```

Notice that 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. (An example of how this works will be given in the Crank-Nicolson script below.) Note that if the diffusion coefficient $D(x)$ doesn't change with time you can load **A** and **B** just once before the time loop starts.

(ii) Once the matrices **A** and **B** are loaded finding the new temperature inside the time loop is easy. Here is what it would look like if the boundary conditions were $T(0) = 1$ and $T(L) = 5$ using a cell-centered grid.

The top and bottom rows of **A** and **B** and the top and bottom positions of r would have been loaded like this (assuming a cell-center grid with ghost points):

$$A(1,1) = \frac{1}{2} \quad A(1,2) = \frac{1}{2} \quad B(1,1) = 0 \quad r(1) = 1 \tag{8.12}$$

$$A(N+2,N+2) = \frac{1}{2} \quad A(N+2,N+1) = \frac{1}{2} \quad B(N+2,N+2) = 0 \quad r(N+2) = 5 \tag{8.13}$$

so that the equations for the top and bottom rows are

$$\frac{T_1 + T_2}{2} = r_1 \qquad \frac{T_{N+1} + T_{N+2}}{2} = r_{N+2} \tag{8.14}$$

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

The time advance would then look like this:

```
% find the right-hand side for the solve at interior points
r=B*T;
```

```
% load T(0) and T(L)
r(1)=1;r(N+2)=5;

% find the new T and load it directly into the variable T
% so we will be ready for the next step
T=A\r;
```

P8.3 (a) Below is a Matlab program that implements the Crank-Nicolson algorithm. Download it from the class web site and study it until you know what each part does. Test `cranknicholson.m` by running it with $D(x) = 2$ and an initial temperature given by $T(x) = \sin(\pi x/L)$. As you found in Lab 7, the exact solution for this distribution is:

$$T(x,t) = \sin(\pi x/L)e^{-\pi^2 Dt/L^2} \tag{8.15}$$

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, i.e., use a skip factor of 1.

Also 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 5 seconds and find the maximum difference between the exact and numerical solutions. You should find that the time step τ hardly matters at all. The number of cells N is the main thing to worry about if you want high accuracy in diffusion problems solved with Crank-Nicolson.

(b) Modify the Crank-Nicolson script 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. Use a "microscope" on the plots early in time to see what happens in the first few grid points during the first few time steps.

(c) Repeat part (b) with $D(x)$ chosen so that $D = 1$ over the range $0 \le x < L/2$ and $D = 5$ over the range $L/2 \le x \le L$. Explain physically why your results are reasonable. In particular, explain why even though D is completely different, the final value of T is the same as in part (b).

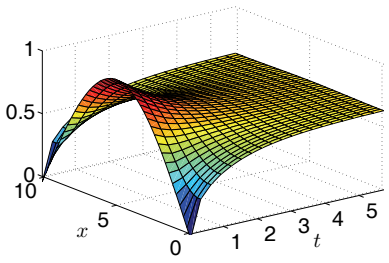


Figure 8.4 Solution to 8.3(b)

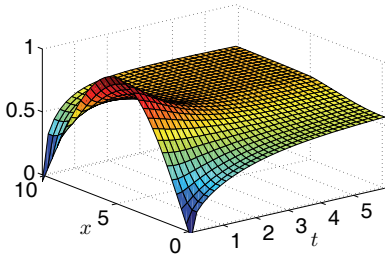


Figure 8.5 Solution to 8.3(c)

Listing 8.1 (cranknicholson.m)

```
clear;close all;

% Set the number of grid points and build a cell-center grid
N=input(' Enter N, cell number - ') L=10; h=L/N; x=-.5*h:h:L+.5*h;
x=x';  % Turn x into a column vector.

%  Load the diffusion coefficient array (make it a column vector)
```

```
D=ones(N+2,1); % (just 1 for now--we'll change it later)

% Load Dm with average values D(j-1/2) and Dp with D(j+1/2)
Dm=zeros(N+2,1);Dp=zeros(N+2,1);  % Make the column vectors
Dm(2:N+1)=.5*(D(2:N+1)+D(1:N));   % average j and j-1
Dp(2:N+1)=.5*(D(2:N+1)+D(3:N+2)); % average j and j+1

% Set the initial temperature distribution
T=sin(pi*x/L);

% Find the maximum of T for setting plot limits
Tmax=max(T);Tmin=min(T);

% Choose the time step tau.
% The max tau for explicit stability is a reasonable choice
fprintf(' Maximum explicit time step: %g \n',h^2/max(D))
tau = input(' Enter the time step - ')

% Create the matrices A and B by loading them with zeros
A=zeros(N+2); B=zeros(N+2);

% load A and B at interior points
const = 2*h^2 / tau; for j=2:N+1
   A(j,j-1)= -Dm(j);
   A(j,j)  = const + (Dm(j)+Dp(j));
   A(j,j+1)= -Dp(j);

   B(j,j-1)=  Dm(j);
   B(j,j)  = const-(Dm(j)+Dp(j));
   B(j,j+1)=  Dp(j);
end

% load the boundary conditions into A and B
A(1,1)=0.5; A(1,2)=0.5; B(1,1)=0.;            % T(0)=0
A(N+2,N+1)=0.5; A(N+2,N+2)=0.5; B(N+2,N+2)=0; % T(L)=0

% Set the number of time steps to take.
tfinal=input(' Enter the total run time - ') nsteps=tfinal/tau;

% Choose how many iterations to skip between plots
skip = input(' Number of iterations to skip between plots - ')

% This is the time advance loop.
for mtime=1:nsteps
```

```
    % define the time
    t=mtime*tau;

    % find the right-hand side for the solve at interior points
    r=B*T;

    % apply the boundary conditions
    r(1)=0;   % T(0)=0
    r(N+2)=0; % T(L)=0

    % do the linear solve to update T
    T=A\r;

    % Make a plot of T every once in a while.
    if(rem(mtime,skip) == 0)
        plot(x,T)
        axis([0 L Tmin Tmax])
        pause(.1)
    end

end
```

Lab 9

Schrödinger's Equation

Here is the time-dependent Schrödinger equation which governs the way a quantum wave function changes with time in a one-dimensional potential well $V(x)$:[1]

$$i\hbar\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\psi}{\partial x^2} + V(x)\psi \tag{9.1}$$

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.

P9.1 Derive the Crank-Nicolson algorithm for Schrödinger's equation. It will probably be helpful to use the material in Lab 8 as a guide (beginning with Eq. (8.3)). Schrödinger's equation is simpler in the sense that you don't have to worry about a spatially varying diffusion constant, but make sure the $V(x)\psi$ term enters the algorithm correctly.

Modify one of your scripts from Lab 8 to implement this algorithm. Note that if you need to turn row vectors into column vectors you should use the Matlab command `.'` (transpose without complex conjugate) instead of `'` (transpose and complex conjugate) since we are doing quantum mechanics and the imaginary parts matter now.

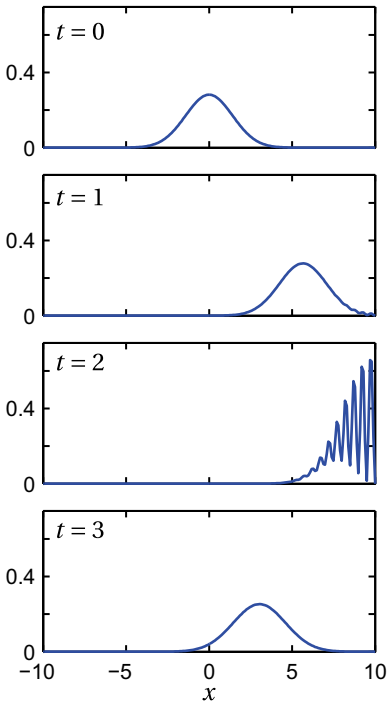


Figure 9.1 The probability density $|\psi(x)|^2$ of a particle in a box that initially moves to the right and then interferes with itself as it reflects from an infinite potential (Problem 9.2(a)).

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

$$V(x) = \begin{cases} 0 & \text{if } -L < x < L \\ +\infty & \text{otherwise} \end{cases} \tag{9.2}$$

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

$$\psi(-L) = 0 \quad ; \quad \psi(L) = 0 \tag{9.3}$$

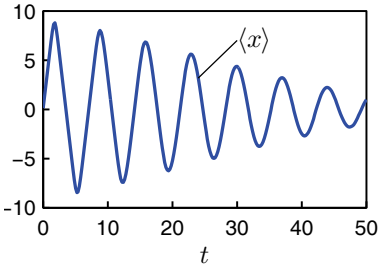


Figure 9.2 The expectation position $\langle x\rangle$ for the particle in Fig. 9.1 as time progresses and the packet spreads out (Problem 9.2(c)).

[1] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 470-506.

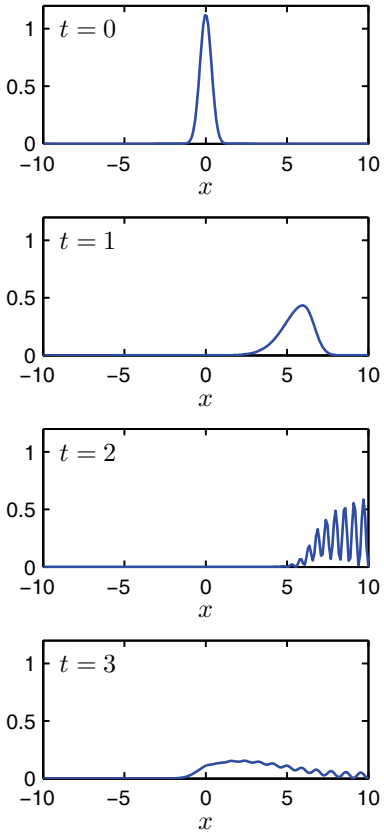


Figure 9.3 The probability density $|\psi(x)|^2$ of a particle that is initially more localized quickly spreads (Problem 9.2(d)).

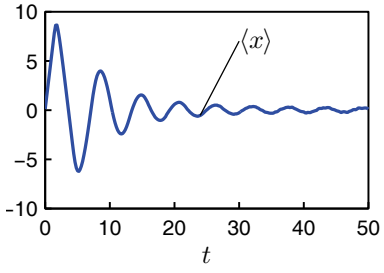


Figure 9.4 The expectation position of the particle in Fig. 9.3 as time progresses.

P9.2 (a) Write a script to solve the time-dependent Schrödinger equation using Crank-Nicolson as discussed in Lab 8. Use natural units in which $\hbar = m = 1$ and $L = 10$. We find that a cell-edge grid

```
h=2*L/(N-1); x=-L:h:L;
```

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:

$$\psi(x,0) = \frac{1}{\sqrt{\sigma\sqrt{\pi}}} e^{ipx/\hbar} e^{-x^2/(2\sigma^2)} \tag{9.4}$$

with $p = 2\pi$ and $\sigma = 2$. 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 the sizes of L and σ must compare in order 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 it stays properly normalized, even though the wave function is bouncing and spreading within the well. (If you are on a cell-edge grid you will need to do the integrals with `trapz` rather than `sum`.)

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

$$\langle x \rangle = \int_{-L}^{L} \psi^*(x,t)\, x\, \psi(x,t)\, dx \tag{9.5}$$

is correct for a bouncing particle. Plot $\langle x \rangle(t)$ 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

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

$$\langle E\rangle = \int_{-\infty}^{\infty} \psi^* H\psi\, dx \tag{9.6}$$

where ψ^* is the complex conjugate of ψ and where

$$H\psi = -\frac{\hbar^2}{2m}\frac{\partial^2\psi}{\partial x^2} + V(x)\psi(x) \tag{9.7}$$

In our case the initial wave function $\psi(x,0)$ 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.

P9.3 (a) Use Mathematica to compute $\langle E\rangle$ for your wave packet. You should find that

$$\langle E\rangle = \frac{p^2}{2m} + \frac{\hbar^2}{4m\sigma^2} \approx 20 \tag{9.8}$$

Since this is a conservative system, the energy remains constant and you can just use the initial wave function in the integral.

HINT: You will need to use the `Assuming` command to specify that certain quantities are real and/or positive to get Mathematica to do the integral.

(b) Modify your script from Problem 9.2 so that it uses a computing region that goes from $-2L$ to $3L$ and a potential

$$V(x) = \begin{cases} 0 & \text{if } -2L < x < 0.98L \\ V_0 & \text{if } 0.98L \le x \le L \\ 0 & \text{if } L < x < 3L \\ \\ +\infty & \text{otherwise} \end{cases} \tag{9.9}$$

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.

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

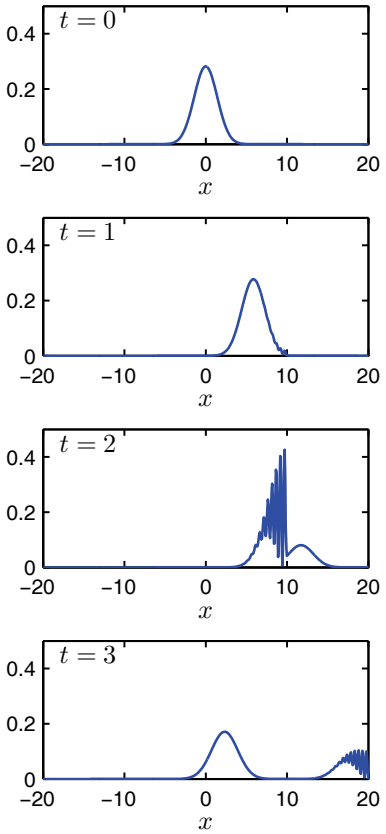


Figure 9.5 The probability distribution $|\psi(x)|^2$ for a particle incident on a narrow potential barrier located just before $x = 10$ with $V_0 > \langle E\rangle$. Part of the wave tunnels through the barrier and part interferes with itself as it is reflected.

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.

Lab 10

Poisson's Equation I

Now let's consider Poisson's equation in two dimensions:[1]

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = -\frac{\rho}{\epsilon_0} \tag{10.1}$$

Poisson's equation is used to describe the electric potential in a two-dimensional region of space with charge density described by ρ. 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 h_y. This creates a rectangular grid with $N_x \times N_y$ 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

$$\frac{V_{j+1,k} - 2V_{j,k} + V_{j-1,k}}{h_x^2} + \frac{V_{j,k+1} - 2V_{j,k} + V_{j,k-1}}{h_y^2} = -\frac{\rho_{j,k}}{\epsilon_0} \tag{10.2}$$

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 (10.2) 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 $N_x \times N_y$, which for a 100×100 grid is 10,000 unknowns. So to do the solve directly we would have to be working with a $10,000 \times 10,000$ matrix, requiring 800 megabytes of RAM just 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

[1] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 138-150.

now 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 $100 \times 100 \times 100$ grid, this would require $(10^4)^3 \times 8$, or 8 million megabytes of memory. Computers like this are not going to be available for your use any time soon. So even though gigabyte computers are common, people still use iteration methods like the ones we are about to describe.

Iteration methods

Let's look for a version of Poisson's equation that helps us develop a less memory-intensive way to solve it.

P10.1 Solve the finite-difference version of Poisson's equation (Eq. (10.2)) for $V_{j,k}$ to obtain

$$V_{j,k} = \left(\frac{V_{j+1,k} + V_{j-1,k}}{h_x^2} + \frac{V_{j,k+1} + V_{j,k-1}}{h_y^2} + \frac{\rho_{j,k}}{\epsilon_0} \right) \Bigg/ \left(\frac{2}{h_x^2} + \frac{2}{h_y^2} \right) \tag{10.3}$$

With the equation in this form we could just iterate over and over by doing the following. (1) Choose an initial guess for the interior values of $V_{j,k}$. (2) Use this initial guess to evaluate the right-hand side of Eq. (10.3) and then to 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. [2]

Carl Friedrich Gauss (1777–1855, German)

It may seem that it would take a miracle for this to work, and it really is pretty amazing that it does, but we shouldn't be too surprised because you can do something similar just by pushing buttons on a calculator. Consider solving this equation by iteration:

$$x = e^{-x} \tag{10.4}$$

If we iterate on this equation like this:

$$x_{n+1} = e^{-x_n} \tag{10.5}$$

we find that the process converges to the 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 n^{th} iteration level we are close to the solution, only missing it by the small quantity δ_n like this: $x_n = \bar{x} + \delta_n$. Let's substitute this approximate solution into Eq. (10.5) and expand using a Taylor series. Recall that the general form for a Taylor's series is

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \cdots + f^{(n)}(x)\frac{h^n}{n!} + \cdots \tag{10.6}$$

[2] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 429-441.

When we substitute our approximate solution into Eq. (10.5) and expand around the exact solution $\bar{x}$ we get

$$x_{n+1} = e^{-\bar{x}-\delta_n} \approx e^{-\bar{x}} - \delta_n e^{-\bar{x}} + \cdots \tag{10.7}$$

If we ignore the terms that are higher order in δ (represented by $\cdots$), then Eq. (10.7) 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 miracle—it is just a consequence of having this expansion technique result in an error multiplier that is less than 1 in magnitude.

P10.2 Write a short Matlab script to solve the equation $x = e^{-x}$ by iteration and verify that it converges. Then try solving this same equation the other way round: $x = -\ln x$ and show that the algorithm doesn't converge. Then use the $\bar{x} + \delta$ analysis above to show why it doesn't.

Well, what does this have to do with our problem? To see, let's notice that the iteration process indicated by Eq. (10.3) can be written in matrix form as

$$V_{n+1} = \mathbf{L}V_n + r \tag{10.8}$$

where $\mathbf{L}$ is the matrix which, when multiplied into the vector V_n, produces the $V_{j,k}$ part of the right-hand side of Eq. (10.3) and r is the part that depends on the charge density $\rho_{j,k}$. (Don't worry about what $\mathbf{L}$ 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 n^{th} iteration. The iteration process on the error is, then,

$$\delta_{n+1} = \mathbf{L}\delta_n \tag{10.9}$$

Now think about the eigenvectors and eigenvalues of the matrix $\mathbf{L}$. If the matrix is well-behaved enough that its eigenvectors span the full solution vector space of size $N_x \times N_y$, then we can represent δ_n 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 $\mathbf{L}$ to have magnitudes less than 1. So we can now restate the original miracle, "Iteration on Eq. (10.3) converges," in this way: "All of the eigenvalues of the matrix $\mathbf{L}$ on the right-hand side of Eq. (10.3) 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. (10.8) is known as *Jacobi iteration*. Unfortunately, even though all of the eigenvalues have magnitudes 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. (10.3), they used values of $V_{j,k}$ as they became available during the iteration. (This means that the right side of Eq. (10.3) contains a mixture of V-values at the n and $n+1$ iteration levels.) This change, which is called *Gauss-Seidel 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. (When you see this algorithm coded you will understand this better.)

Successive over-relaxation

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

$$x_{n+1} = e^{-x_n} \tag{10.10}$$

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

$$x_{n+1} = \omega e^{-x_n} + (1-\omega)x_n \tag{10.11}$$

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

P10.3 Verify that even though Eq. (10.11) looks quite different from Eq. (10.10), it is still solved by $x = e^{-x}$. Now insert $x_n = \bar{x} + \delta_n$ and expand as before to show that the error changes as we iterate according to the following

$$\delta_{n+1} = (-\omega e^{-\bar{x}} + 1 - \omega)\delta_n \tag{10.12}$$

Now look: 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.

P10.4 Write a Matlab script that accepts a value of ω and runs the iteration in Eq. (10.11). 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 won't give convergence in one step. See if you can figure out why not. (Think about the approximations involved in obtaining Eq. (10.12).)

As you can see from Eq. (10.12), 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:

$$V_{n+1} = \omega(\mathbf{L}V_n + r) + (1-\omega)V_n \tag{10.13}$$

then play with ω until we achieve almost instantaneous convergence.

Sadly, this doesn't quite work. The problem is that in solving for $N_x \times N_y$ unknown values $V_{j,k}$ we don't have just one multiplier; we have many thousands of them, 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 -1 and 1. (Draw a picture of an arbitrary eigenvalue range between -1 and 1 and imagine shifting the range to verify this statement.)

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

$$V_{j,k} = \omega \left(\frac{V_{j+1,k} + V_{j-1,k}}{h_x^2} + \frac{V_{j,k+1} + V_{j,k-1}}{h_y^2} + \frac{\rho_{j,k}}{\epsilon_0} \right) \Bigg/ \left(\frac{2}{h_x^2} + \frac{2}{h_y^2} \right) + (1-\omega) V_{j,k} \tag{10.14}$$

or (written in terms of Rhs, the right-hand side of Eq. (10.3)):

$$V_{j,k} = \omega Rhs + (1-\omega) V_{j,k} \tag{10.15}$$

with the values on the right updated as we go, i.e., we don't have separate arrays for the new V's and the old V's. 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 ω 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

$$\omega = \frac{2}{1 + \sqrt{1 - R^2}} \tag{10.16}$$

where

$$R = \frac{h_y^2 \cos(\pi / N_x) + h_x^2 \cos(\pi / N_y)}{h_x^2 + h_y^2}. \tag{10.17}$$

These formulas are easy to code and 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. (10.16) 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 a value of $V_{j,k}$ at some grid point and quit when its value has stabilized at some level, like this for instance: quit when $\epsilon = |V(j,k)_{n+1} - V(j,k)_n| < 10^{-6}$. 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 don'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

$$\epsilon = \left| \frac{Lhs - Rhs}{V_{\text{scale}}} \right| \tag{10.18}$$

where Lhs is the left-hand side of Eq. (10.3) 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 error criterion should we choose 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. OK, we're finally ready to code this all up.

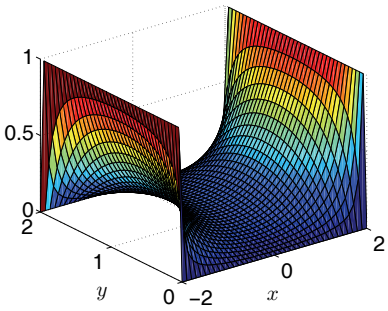


Figure 10.1 The electrostatic potential $V(x, y)$ with two sides grounded and two sides at constant potential.

P10.5 (a) Below is a piece of Matlab code `sor.m` that implements these ideas for a two-dimensional cell-edge grid $[-L_x/2, L_x/2] \times [0, L_y]$ with x corresponding to L_x and y corresponding to L_y. You just have to fill in the blank sections of code and it will be ready to go. Finish writing the script and run it repeatedly with $N_x = N_y = 30$ and different values of ω. Note that the boundary conditions on $V(x, y)$ 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. (10.16) is the best one to use.

(b) Using the optimum value of ω in each case, run `sor.m` for $N_x = N_y =$ 20, 40, 80, and 160. See if you can find a rough power law formula for how long it takes to push the error below 10^{-5}, i.e., guess that Run Time $\approx AN_x^p$, and find A and p. The `tic` and `toc` commands will help you with the timing. (Make sure not to include any plotting commands between `tic` and `toc`.) The Matlab fitting tool `cftool` can be helpful for curve fitting.

NOTE: If you've finished this much and still have time, you should continue on to lab 11 (it is a continuation of this lab and is a bit long).

Listing 10.1 (sor.m)

```
% Solve Poisson's equation by Successive-Over-relaxation
% on a rectangular Cartesian grid
clear; close all;
eps0=8.854e-12;  % set the permittivity of free space
```

```
Nx=input('Enter number of x-grid points - '); Ny=input('Enter number of
y-grid points - ');

Lx=4; % Length in x of the computation region
Ly=2; % Length in y of the computation region

% define the grids
hx=Lx/(Nx-1);          % Grid spacing in x
hy=Ly/(Ny-1);          % Grid spacing in y
x = (0:hx:Lx)-.5*Lx;  %x-grid, x=0 in the middle
y = 0:hy:Ly;           %y-grid

% estimate the best omega to use
R = (hy^2 * cos(pi/Nx)+hx^2*cos(pi/Ny))/(hx^2+hy^2); omega=2/(1+sqrt(1-R^2));
fprintf('May I suggest using omega = %g ? \n',omega);
omega=input('Enter omega for SOR - ');

% define the voltages
V0=1;        % Potential at x=-Lx/2 and x=+Lx/2
Vscale=V0;   % set Vscale to the potential in the problem

% set the error criterion
errortest=input(' Enter error criterion - say 1e-6 - ') ;

% Initial guess is zeroes
V = zeros(Nx,Ny);

% set the charge density on the grid
rho=zeros(Nx,Ny);

% set the boundary conditions
% recall that in V(j,k), j is the x-index and k is the y-index
. . .

% MAIN LOOP

Niter = Nx*Ny*Nx;  %Set a maximum iteration count

%  set  factors used repeatedly in the algorithm
fac1 = 1/(2/hx^2+2/hy^2); facx = 1/hx^2; facy = 1/hy^2;

for n=1:Niter
```

```
   err(n)=0; % initialize the error at iteration n to zero

   for j=2:(Nx-1)   % Loop over interior points only
      for k=2:(Ny-1)

        % load rhs with the right-hand side of the Vjk equation,
        % Eq. (10.3)
        rhs =  . . .

        % calculate the relative error for this point, Eq. (10.18)
        currerr = . . .

        % Add some logic to compare err(n) and currerr and store
        % the bigger one in err(n). err(n) should hold
        % the biggest error on the grid for this n step after
        % finishing the j and k loops
        err(n)= . . .

         % SOR algorithm Eq. (10.14), to
         % update V(j,k) (use rhs from above)
         V(j,k) = . . .

      end
   end

   % if err < errortest break out of the loop

   fprintf('After %g iterations, error= %g\n',n,err(n));

   if(err(n) < errortest)
      disp('Desired accuracy achieved; breaking out of loop');
      break;
   end

end

% make a contour plot
figure
cnt=[0,.1,.2,.3,.4,.5,.6,.7,.8,.9,1]; % specify contours
cs = contour(x,y,V',cnt);  % Contour plot with labels
xlabel('x'); ylabel('y'); clabel(cs,[.2,.4,.6,.8])

% make a surface plot
figure
```

```
surf(x,y,V');  % Surface plot of the potential
xlabel('x'); ylabel('y');

% make a plot of error vs. iteration number
figure semilogy(err,'b*') xlabel('Iteration'); ylabel('Relative Error')
```

Lab 11

Poisson's Equation II

In three dimensions, Poisson's equation is given by

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} + \frac{\partial^2 V}{\partial z^2} = -\frac{\rho}{\epsilon_0} \tag{11.1}$$

You can solve this equation using the SOR technique, but we won't make you do that here. Instead, we'll look at several 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 we can use the 2-D SOR code that we developed in the last lab to solve for the potential of a cross-sectional area in the x and y dimensions.

P11.1 (a) Modify `sor.m` to model a rectangular pipe with $V(-L_x/2, y) = -V_0$, $V(L_x/2, y) = V_0$, and the $y = 0$ and $y = L_y$ edges held at $V = 0$. Then make a plot of the surface charge density at the inside surface of the $y = 0$ side of the pipe. Assume that the pipe is metal on this surface.

To do this you will need to remember that the connection between surface charge density and the normal component of **E** is

$$\sigma = \epsilon_0 (\mathbf{E}_2 - \mathbf{E}_1) \cdot \hat{\mathbf{n}} \tag{11.2}$$

where $\mathbf{E}_{1,2}$ are the fields on sides 1 and 2 of the boundary and $\hat{\mathbf{n}}$ is a unit vector pointing normal to the surface from side 1 to side 2. You'll also need to recall how to compute **E** from the potential V:

$$\mathbf{E} = -\nabla V \tag{11.3}$$

HINT: Since you only care about the normal component of **E**, you only need to do the y-derivative part of the gradient. You can do an appropriate finite difference derivative to find E_y half way between the first and second points, and another difference to find E_y half way between the second and third points. Then you can use Eq. (1.3) with these two interior values of E_y to find the field at the edge of your grid. Since the boundary is metal, the field on the other side is zero.

(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. (2.12)). Both methods work fine, but note that you will need to enforce boundary conditions

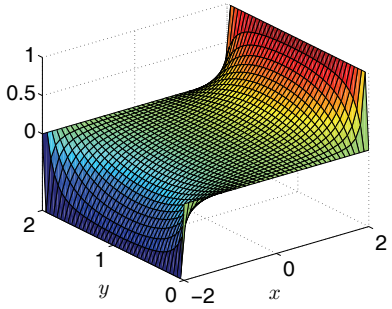


Figure 11.1 The potential $V(x, y)$ from Problem 11.1(a).

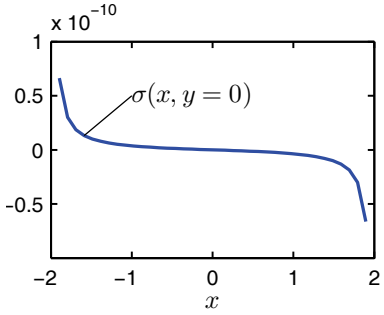


Figure 11.2 The surface charge density from Problem 11.1(a).

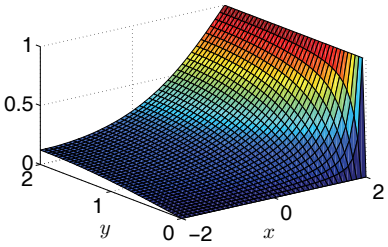


Figure 11.3 The potential $V(x, y)$ with zero-derivative boundary conditions on two sides (Problem 11.1(b).)

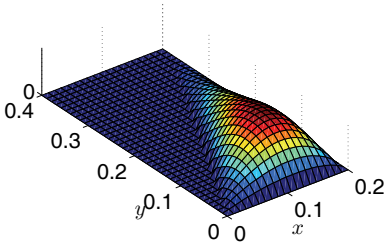


Figure 11.4 The potential $V(x, y)$ with constant charge density on a triangular region grounded at its edges (Problem 11.2(a).)

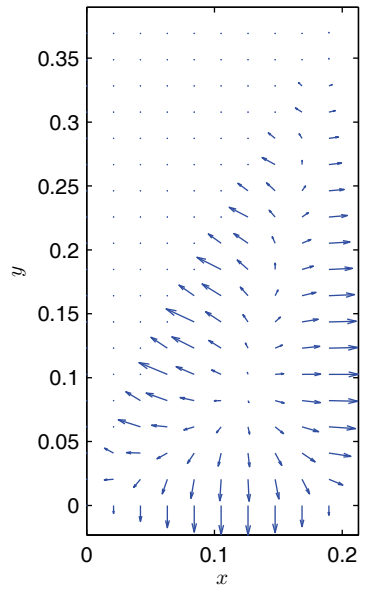


Figure 11.5 The electric field from Problem 11.2(b).)

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. (10.16) only works for cell-edge grids with fixed-value boundary conditions, so it only gives a ballpark suggestion for this problem.)

P11.2 (a) Modify `sor.m` to solve the problem of an infinitely long hollow rectangular pipe of x-width 0.2 m and y-height 0.4 m with an infinitely long thin diagonal plate from the lower left corner to the upper right corner. The edges of the pipe and the diagonal plate are all grounded. There is uniform charge density $\rho = 10^{-10}$ C/m^3 throughout the lower triangular region and no charge density in the upper region (see Fig. 11.4). Find $V(x, y)$ in both triangular regions. You will probably want to have a special relation between N_x and N_y and use a cell-edge grid in order to apply the diagonal boundary condition in a simple way.

(b) Make a `quiver` plot of the electric field at the interior points of the grid. Use online help to see how to use `quiver`, and make sure that the `quiver` command is followed by the command `axis equal` so that the x and y axes have the same scale. Matlab's `gradient` command

```
[Ex,Ey] = gradient(-V',hx,hy)
```

will let you quickly obtain the components of **E**. The transpose on `V` in this command is necessary because of the (row,column) versus (x,y) notation issue.

P11.3 Study electrostatic shielding by going back to the boundary conditions of Problem 11.1(a) with $L_x = 0.2$ and $L_y = 0.4$, 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.

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 to $V = 0$ before doing the next iteration. 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=ones(Nx,Ny);
```

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 `mask(j,k)` is one:

```
if (mask(j,k)==1)
    % SOR stuff and error calculation in here
end
```

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 calculate the potential for quite complicated shapes just by changing the mask array.

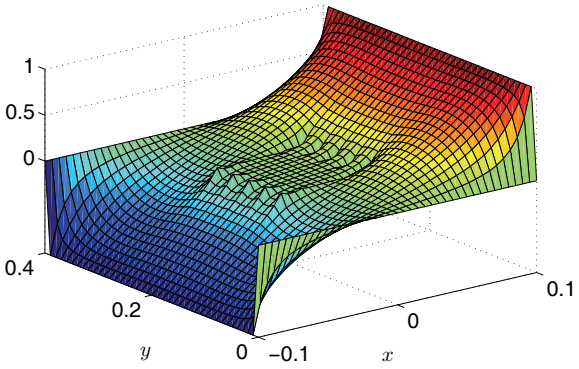


Figure 11.6 The potential $V(x, y)$ for an electrostatic "cage" formed by grounding some interior points. (Problem 11.3.)

Lab 12

Gas Dynamics I

So far we have only 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. We'll discuss each of the three equations separately, and then we'll look at how to solve the coupled system.

Conservation of mass

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

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}\left(\rho v\right) = 0 \tag{12.1}$$

This equation says that as the gas particles are moved by the flow velocity $v(x,t)$, the density is carried along with the flow, and can also be compressed or rarefied. As we will see shortly, if $\partial v/\partial x > 0$ then the gas expands, decreasing ρ; if $\partial v/\partial x < 0$ then the gas compresses, increasing ρ.

P12.1 (a) Roughly verify the statement above by expanding the spatial derivative in Eq. (12.1) to put the equation in the form

$$\frac{\partial \rho}{\partial t} + v\frac{\partial \rho}{\partial x} = -\rho\frac{\partial v}{\partial x} \tag{12.2}$$

If v = const, show that the simple moving pulse formula $\rho(x,t) = \rho_0(x - vt)$, where $\rho_0(x)$ is the initial distribution of density solves this equation. (Just substitute it in and show that it works.) This simply means that the density distribution at a later time is the initial one moved over by a distance vt: this is called *convection*.

(b) Now suppose that the initial distribution of density is $\rho(x,0) = \rho_0 =$ const but that the velocity distribution is an "ideal explosion", with $v = 0$ at $x = 0$ and velocity increasing linearly away from 0 like this: $v(x) = v_0 x/a$ (v doesn't vary with time.) Show that the solution of Eq. (12.1) is now given by $\rho(x,t) = \rho_0 e^{-\alpha t}$ and determine the value of the decay constant α. (You may be concerned that even though ρ is constant in x initially, it may not be later. But we have given you a very special $v(x)$, namely the only one that keeps an initially constant density constant forever. So just assume that ρ doesn't depend on

x, then show that your solution using this assumption satisfies the equation and the boundary conditions.)

Think carefully about this flow pattern long enough that you are convinced that the density should indeed go down as time passes.

(c) Now repeat part (b) with an implosion (the flow is inward): $v(x) = -v_0 x/a$. Does your solution make sense?

These are both very simple cases, but they illustrate the basic physical effects of a velocity flow field $v(x,t)$: the density is convected along with the flow and either increases or decreases as it flows depending on the sign of $\partial v/\partial x$.

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

$$\frac{\partial T}{\partial t} + v\frac{\partial T}{\partial x} = -(\gamma - 1)T\frac{\partial v}{\partial x} + D_T\frac{\partial^2 T}{\partial x^2} \tag{12.3}$$

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

$$D_T = \frac{(\gamma - 1)M\kappa}{k_B\rho} \tag{12.4}$$

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.

P12.2 Show that the ideal gas law $P = nk_BT$ (where n is the number of particles per unit volume) specifies the following connection between pressure P and the density ρ and temperature T:

$$P = \frac{k_B}{M}\rho T \tag{12.5}$$

Newton's second law

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

$$\frac{\partial v}{\partial t} + v\frac{\partial v}{\partial x} = -\frac{1}{\rho}\frac{\partial P}{\partial x} + \frac{4\mu}{3\rho}\frac{\partial^2 v}{\partial x^2} \tag{12.6}$$

You should quickly 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. (12.5). 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. (12.6) is acceleration. To become convinced, let's think about this situation more carefully. Notice that Newton's second law does not apply directly to a place in space where there is a moving fluid. Newton's second law is for particles that are moving, not for a piece of 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 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 acceleration in fluids than just $\partial v/\partial t = 0$.

P12.3 To see what's missing, let's force ourselves to ride along with the flow by writing $v = v(x(t), t)$, where $x(t)$ is the position of the moving piece of honey. Carefully use the rules of calculus to evaluate dv/dt and derive the acceleration formula on the left-hand side of Eq. (12.6).

Numerical approaches to the continuity equation

In the next lab we will actually tackle the hard problem of simultaneously advancing ρ, T, and v in time and space, but for now we will just practice on one of them to develop the tools we need to do the big problem. And to keep things simple, we will work with the simplest equation of the set:

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho v}{\partial x} = 0 \tag{12.7}$$

with a specified flow profile $v(x)$ which is independent of time and an initial density distribution $\rho(x, 0) = \rho_0(x)$.

The apparent simplicity of this equation is deceptive; 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.) In the following problem we will try an algorithm that is unstable, another that is stable but inaccurate, and finally one that is both stable and conservative, but only works well if the solution doesn't become too steep. (Warning: we are talking about gas dynamics here, so shock waves routinely show up as solutions. Numerical methods that properly handle shocks are much more difficult than the ones we will show you here.)

Before we continue we need to tell you about the boundary conditions on Eq. (12.7). 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 somebody might be feeding 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.)

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

P12.4 Let's start with something really simple and inaccurate just to see what can go wrong.

$$\frac{\rho_j^{n+1}-\rho_j^n}{\tau}+\frac{1}{2h}\left(\rho_{j+1}^n v_{j+1}-\rho_{j-1}^n v_{j-1}\right)=0 \tag{12.8}$$

This involves a nice centered difference in x and an inaccurate forward difference in t. 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. We suggest that you use a cell-center grid with ghost points because we will be using a grid like this in the next lab. Use about 400 grid points. Use

$$\rho(x,0)=1+e^{-200(x/L-1/2)^2} \tag{12.9}$$

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

$$v(x)=v_0 \tag{12.10}$$

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 it leave by using linear extrapolation:

$$\rho(L,t)=2\rho(L-h,t)-\rho(L-2h,t) \quad \text{or} \quad \rho_{N+2}=2\rho_{N+1}-\rho_N \tag{12.11}$$

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.

P12.5 Now 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.

$$\rho(x, t+\tau) = \rho(x,t) + \tau\frac{\partial\rho}{\partial t} + \frac{\tau^2}{2}\frac{\partial^2\rho}{\partial t^2} \tag{12.12}$$

(a) Use this Taylor expansion and Eq. (12.7) to derive the following expression (assume that v is not a function of time):

$$\rho(x, t+\tau) = \rho(x,t) - \tau\frac{\partial\rho v}{\partial x} + \frac{\tau^2}{2}\frac{\partial}{\partial x}\left(v\frac{\partial\rho v}{\partial x}\right) \ . \tag{12.13}$$

If you stare at this equation for a minute you will see that a diffusion-like term has showed up. To see this subtract $\rho(x,t)$ from both sides and divide by τ, then interpret the new left-hand side as a time derivative.

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 12.4. Notice also that the diffusion coefficient you found above 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.

(b) Now finite difference the expression in Eq. (12.13) assuming that $v(x) = v_0 = \text{const}$, as in 12.4 to find the Lax-Wendroff algorithm:

$$\rho_j^{n+1} = \rho_j^n - \frac{v_0\tau}{2h}[\rho_{j+1}^n - \rho_{j-1}^n] + \frac{v_0^2\tau^2}{2h^2}[\rho_{j+1}^n - 2\rho_j^n + \rho_{j-1}^n] \tag{12.14}$$

Change your script from 12.4 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 12.4 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 12.4, we suppose we should be grateful.

P12.6 Finally, let's try an implicit method, Crank-Nicolson in fact. Proceeding as we did with the diffusion equation and Schrödinger's equation we finite difference Eq. (12.7) like this:

$$\frac{\rho_j^{n+1}-\rho_j^n}{\tau} = -\frac{1}{2h}\left(\frac{\rho_{j+1}^{n+1}+\rho_{j+1}^n}{2}v_{j+1} - \frac{\rho_{j-1}^{n+1}+\rho_{j-1}^n}{2}v_{j-1}\right) \tag{12.15}$$

Note that v has no indicated time level because we are treating it as constant in time for this lab. (In the next lab we will let v change in time as well as space.) And now because we have ρ_{j-1}^{n+1}, ρ_j^{n+1}, and ρ_{j+1}^{n+1} involved in this equation at each grid point j we need to solve a linear system of equations to find ρ_j^{n+1}.

(a) Put the Crank-Nicolson algorithm above into matrix form like this:

$$\mathbf{A}\rho^{n+1} = \mathbf{B}\rho^n \tag{12.16}$$

by finding $A_{j,j-1}$, $A_{j,j}$, $A_{j,j+1}$, $B_{j,j-1}$, $B_{j,j}$, and $B_{j,j+1}$ (the other elements of **A** and **B** are zero.)

(b) 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 **A** and **B**. This involves multiplying $\mathbf{B}\rho^n$ to find an `r`-vector as you have done before.

(c) 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.

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

$$v(x) = 1.2 - x/L \tag{12.17}$$

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.

(e) 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

$$\rho(x,0) = \begin{cases} 1.0 & \text{if } 0 \le x \le L/2 \\ 0 & \text{otherwise} \end{cases} \tag{12.18}$$

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.

(f) For comparison, try the same step-function initial condition in your Lax-Wendroff script from Problem 12.5.

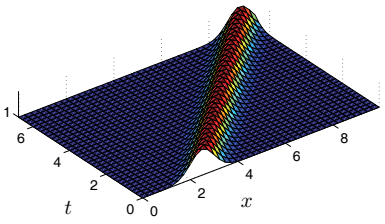


Figure 12.1 A pulse is convected across a region in which the convection velocity $v(x)$ is constant (Problem 12.6(c)).

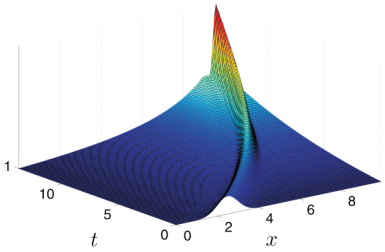


Figure 12.2 A pulse is convected across a region in which the convection velocity $v(x)$ is decreasing. Note that the pulse narrows and grows, conserving mass. (Problem 12.6(d))

Lab 13

Gas Dynamics II

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. As a reminder, the three equations we need to solve are conservation of mass

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}\left(\rho v\right) = 0 \,, \tag{13.1}$$

conservation of energy

$$\frac{\partial T}{\partial t} + v\frac{\partial T}{\partial x} + (\gamma - 1)T\frac{\partial v}{\partial x} = \frac{(\gamma-1)M\kappa}{k_B}\frac{1}{\rho}\frac{\partial^2 T}{\partial x^2} \,, \tag{13.2}$$

and Newton's second law

$$\frac{\partial v}{\partial t} + v\frac{\partial v}{\partial x} = -\frac{1}{\rho}\frac{\partial P}{\partial x} + \frac{4\mu}{3\rho}\frac{\partial^2 v}{\partial x^2} \tag{13.3}$$

The pressure P is given by the ideal gas law

$$P = \frac{k_B}{M}\rho T \tag{13.4}$$

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.

Simultaneously advancing ρ, T, and v

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. (13.1) to predict $\tilde{\rho}^{n+1}$. Then we'll use $\tilde{\rho}^{n+1}$ to help us calculate $\tilde{T}^{n+1}$ using Eq. (13.2), while

still treating v as fixed in time. Once these predicted values are obtained we can use them with Eq. (13.3) to obtain a predicted $\tilde{v}$. With all of these predictors for the 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. (13.1) → Solve for the predicted density $\tilde{\rho}$.

Step 2 Use v^n and $\tilde{\rho}$ as inputs for Eqn. (13.2) → Solve for the predicted temperature $\tilde{T}$.

Step 3 Use $\tilde{\rho}$ and $\tilde{T}$ as inputs for Eqn. (13.3) → Solve for the predicted velocity $\tilde{v}$.

Step 4 Use $\tilde{v}$ as input for Eqn. (13.1) → Solve for the new density ρ^{n+1}.

Step 5 Use $\tilde{v}$ and ρ^{n+1} as inputs for Eqn. (13.2) → Solve for the new temperature T^{n+1}.

Step 6 Use ρ^{n+1} and T^{n+1} as inputs for Eqn. (13.3) → 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.

We'll start off with the continuity equation, Eq. (13.1). 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 Crank-Nicolson. As usual for Crank-Nicolson, we forward difference in time and center difference in space to find

$$\frac{\rho_j^{n+1}-\rho_j^n}{\tau} = -\frac{v_{j+1}^n\rho_{j+1}^n - v_{j-1}^n\rho_{j-1}^n}{2h} \tag{13.5}$$

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):

$$\rho_j^{n+1}-\rho_j^n = C_1\left(\rho_{j+1}^n+\rho_{j+1}^{n+1}\right) - C_2\left(\rho_{j-1}^n+\rho_{j-1}^{n+1}\right) \tag{13.6}$$

where

$$C_1 = -\frac{\tau}{8h}\left(v_{j+1}^n+v_{j+1}^{n+1}\right) \tag{13.7}$$

$$C_2 = -\frac{\tau}{8h}\left(v_{j-1}^n+v_{j-1}^{n+1}\right) \tag{13.8}$$

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

$$C_2\rho_{j-1}^{n+1}+\rho_j^{n+1}-C_1\rho_{j+1}^{n+1} = -C_2\rho_{j-1}^n+\rho_j^n+C_1\rho_{j+1}^n \tag{13.9}$$

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

$$\mathbf{A}\rho^{n+1} = \mathbf{B}\rho^n \tag{13.10}$$

which we solve using linear algebra techniques. In order for the algorithm represented by Eq. 13.10 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.

Now let's tackle the energy equation (13.2) 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

$$\frac{T_j^{n+1} - T_j^n}{\tau} = -v_j^n \frac{T_{j+1}^n - T_{j-1}^n}{2h} - (\gamma - 1) T_j^n \frac{v_{j+1}^n - v_{j-1}^n}{2h} + F \frac{1}{\rho_j^n} \frac{T_{j+1}^n - 2T_j^n + T_{j-1}^n}{h^2} \tag{13.11}$$

where

$$F = \frac{(\gamma - 1) M \kappa}{k_B} \tag{13.12}$$

We then rearrange Eq. (13.11) into a form that makes the upcoming algebra (and coding) more readable:

$$\frac{T_j^{n+1} - T_j^n}{\tau} = T_{j-1}^n D_1 + T_j^n D_2 + T_{j+1}^n D_3 \tag{13.13}$$

where

$$D_1 = \frac{v_j^n}{2h} + \frac{F}{\rho_j^n h^2} \tag{13.14}$$

$$D_2 = -(\gamma - 1) \frac{v_{j+1}^n - v_{j-1}^n}{2h} - \frac{2F}{\rho_j^n h^2} \tag{13.15}$$

$$D_3 = -\frac{v_j^n}{2h} + \frac{F}{\rho_j^n h^2} \tag{13.16}$$

P13.1 Finish deriving the Crank-Nicolson algorithm for T^{n+1} by putting the right-hand side of Eq. (13.13) at the $n + 1/2$ time level. This means replacing T^n terms with $(T^n + T^{n+1})/2$ in Eq. (13.13) 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 13.1 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.

Finally, we get to Newton's law (Eq. (13.3) with Eq. (13.4)). Notice that Eq. (13.3) has a nonlinear term: $v(\partial v/\partial x)$. 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

$$\frac{v_j^{n+1} - v_j^n}{\tau} = -\bar{v}_j^n \left(\frac{v_{j+1}^n - v_{j-1}^n}{2h}\right) - \frac{k_B}{M\rho_j^n}\left(\frac{\rho_{j+1}^n T_{j+1}^n - \rho_{j-1}^n T_{j-1}^n}{2h}\right) + \frac{4\mu}{3\rho_j^n}\left(\frac{v_{j+1}^n - 2v_j^n + v_{j-1}^n}{h^2}\right) \tag{13.17}$$

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

$$\frac{v_j^{n+1} - v_j^n}{\tau} = E_0 + v_{j-1}^n E_1 + v_j^n E_2 + v_{j+1}^n E_3 \tag{13.18}$$

where

$$E_0 = -\frac{k_B}{M\rho_j^n}\left(\frac{\rho_{j+1}^n T_{j+1}^n - \rho_{j-1}^n T_{j-1}^n}{2h}\right) \tag{13.19}$$

$$E_1 = \frac{\bar{v}_j^n}{2h} + \frac{4\mu}{3\rho_j^n h^2} \tag{13.20}$$

$$E_2 = -\frac{8\mu}{3\rho_j^n h^2} \tag{13.21}$$

$$E_3 = -\frac{\bar{v}_j^n}{2h} + \frac{4\mu}{3\rho_j^n h^2} \tag{13.22}$$

P13.2 Finish deriving the Crank-Nicolson algorithm for v by making the replacements $v^n \Rightarrow (v^{n+1} + v^n)/2$ the right-hand side of Eq. (13.18) and $\rho^n \Rightarrow (\rho^n + \tilde{\rho}^{n+1})/2$, $T^n \Rightarrow (T^n + \tilde{T}^{n+1})/2$, and $\bar{v}^n \Rightarrow (\bar{v}^n + \bar{v}^{n+1})/2$ 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 **A** and **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.

OK, now that you have algorithms for all three equations, we can restate the predictor-corrector 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

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

Waves in a closed tube

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/m^3, 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.

P13.3 (a) As you might guess, debugging the algorithm that we just developed is painful 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 here are four m-files (one main file and three function files) that implement the algorithm. Go to the class web site now and download these m-files. Study them and make sure you understand how they work. Then call the TA over and explain how the code works.

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

$$\partial T/\partial x = 0 \tag{13.23}$$

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

$$v(0,t) = v(L,t) = 0 \tag{13.24}$$

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

$$\frac{\partial \rho}{\partial t} + \rho \frac{\partial v}{\partial x} = 0 \quad \text{at } x = 0 \text{ and } x = L \tag{13.25}$$

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.

Listing 13.1 (gas.m)

```
% Gas Dynamics in a closed tube, Physics 430
clear;close all;

global gamma kB mu M F A B h tau N;

% Physical Constants
gamma = 1.4;      % Adiabatic Exponent
kappa = 0.024;    % Thermal conductivity
kB = 1.38e-23;    % Boltzman Constant
mu = 1.82e-5;     % Coefficient of viscosity
M = 29*1.67e-27; % Mass of air molecule (Average)
F = (gamma-1)*M*kappa/kB;   % a useful constant

% System Parameters
L=10.0;           % Length of tube
T0 = 293;         % Ambient temperature
rho0 = 1.3;       % static density (sea level)

% speed of sound
c=sqrt(gamma * kB * T0 /M);
```

```
% cell-center grid with ghost points
N=100; h=L/N; x=-.5*h:h:L+.5*h;
x=x'; % turn x into a column vector

% initial distributions
rho = rho0 * ones(N+2,1); T = T0 * ones(N+2,1); v = exp(-200*(x/L-0.5).^2) *
c/100;

% taulim=...;
% fprintf(' Courant limit on time step: %g \n',taulim);
tau=1e-4;     %input(' Enter the time step - ')
tfinal = 0.1; %input(' Enter the run time - ')
nsteps = tfinal/tau;

skip = 5;  %input(' Steps to skip between plots - ')

A=zeros(N+2,N+2); B=A;

for n=1:nsteps

    time = (n-1) * tau;

    % Plot the current values before stepping
    if mod((n-1),skip)==0
        subplot(3,1,1)
        plot(x,rho);
        ylabel('\rho');
        axis([0 L 1.28 1.32])
        title(['time=' num2str(time)])
        subplot(3,1,2)
        plot(x,T);
        ylabel('T')
        axis([0 L 292 294])
        subplot(3,1,3)
        plot(x,v);
        ylabel('v');
        axis([0 L -4 4])
        pause(0.1)
    end

% 1. Predictor step for rho
    rhop = Srho(rho,v,v);

% 2. Predictor step for T
```

```
    Tp = ST(T,v,v,rho,rhop);

% 3. Predictor step for v
    vp = Sv(v,v,v,rho,rhop,T,Tp);

% 4. Corrector step for rho
    rhop = Srho(rho,v,vp);

% 5. Corrector step for T
    Tp = ST(T,v,vp,rho,rhop);

% 6. Corrector step for v
    v = Sv(v,v,vp,rho,rhop,T,Tp);

    % Now put rho and T at the same time-level as v
    rho = rhop;
    T = Tp;
end
```

Listing 13.2 (Srho.m)

```
function rho = Srho(rho,v,vp)
% Step rho forward in time by using Crank-Nicolson
% on the continuity equation

global A B h tau N;

% Clear A and B (pre-allocated in main program)
A=A*0; B=A;

% Load interior points
for j=2:N+1
    C1 = -tau * (v(j+1) + vp(j+1)) / (8*h);
    C2 = -tau * (v(j-1) + vp(j-1)) / (8*h);
    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;
end

% Apply boundary condition
. . .
```

```
% Crank Nicolson solve to step rho in time
r = B*rho; rho = A\r;

end
```

Listing 13.3 (ST.m)

```
function T = ST(T,v,vp,rho,rhop)

global gamma F A B h tau N;

% Clear A and B (pre-allocated in main program)
A=A*0; B=A;

% Load interior points
for j=2: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;
end

% Apply boundary condition
% Insulating: dt/dx = 0
. . .

% Crank Nicolson solve to step rho in time
r = B*T; T = A\r;

end
```

Listing 13.4 (Sv.m)

```
function v = Sv(v,vbar,vbarp,rho,rhop,T,Tp)

global kB mu M A B h tau N;
```

```
% Clear A and B (pre-allocated in main program)
A=A*0; B=A;

E0 = v * 0; % create the constant term the right size

% Load interior points
for j=2: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;
end

% Apply boundary condition
% Fixed: v = 0
. . .

% Crank Nicolson solve to step rho in time
r = B*v + E0; v = A\r;

end
```

P13.4 (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

$$v(x,0) = v_0 e^{-200(x/L-1/2)^2} \tag{13.26}$$

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 Crank-Nicolson. 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.

Lab 14

Solitons: Korteweg-deVries Equation

At the Lagoon amusement park in the town of Farmington, just north of Salt Lake City, 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 Korteweg-deVries equation

$$\frac{\partial y}{\partial t} + y\frac{\partial y}{\partial x} + \alpha\frac{\partial^3 y}{\partial x^3} = 0, \tag{14.1}$$

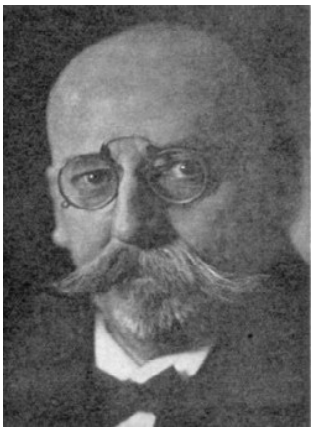
which describes surface waves in shallow water. In the first two terms of this equation you can see the convective behavior we studied in Lab 12, but the last term, with its rather odd third derivative, is something new. We will be studying this equation in this laboratory.

Figure 14.1 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.

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 ($j = (N+1)$) is just a repeat of the first point: $x_{N+1} = x_1$, because our system is periodic.

Diederik Korteweg (1848–1941, Dutch)



Gustav de Vries (1866–1934, Dutch) Diederik Korteweg was Gustav's dissertation advisor.

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. (14.1) has a third derivative, which we haven't done yet in our numerical methods. If you look back at Eq. (**??**) in Problem **??**, you'll find a convenient finite difference form for the third derivative. When we time average Eq. (**??**), we find

$$\alpha\frac{\partial^3 y}{\partial x^3} = \frac{\alpha}{2h^3}\left(y_{j+2}^{n+1} - 3y_{j+1}^{n+1} + 3y_j^{n+1} - y_{j-1}^{n+1} + y_{j+2}^n - 3y_{j+1}^n + 3y_j^n - y_{j-1}^n\right) \tag{14.2}$$

Look closely at Eq. (14.2) 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 finite differences will be cell centered.

With this wrinkle in mind, we can write the first term in Eq. (14.1) at time level $n+1/2$ and space location $j+1/2$ as

$$\frac{\partial y}{\partial t} = \frac{1}{2\tau}\left(y_j^{n+1} + y_{j+1}^{n+1} - y_j^n - y_{j+1}^n\right) \tag{14.3}$$

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

$$y\frac{\partial y}{\partial x} = \frac{\bar{y}_{j+1} + \bar{y}_j}{4h}\left(y_{j+1}^{n+1} - y_j^{n+1} + y_{j+1}^n - y_j^n\right) \tag{14.4}$$

For now, we've ignored the problem that the derivative in Eq. (14.4) 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. (14.2)–(14.4) is now substituted into Eq. (14.1), 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

$$\mathbf{A}y^{n+1} = \mathbf{B}y^n \tag{14.5}$$

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

$$\begin{aligned} A_{j,j-1} &= a_{--} & A_{j,j} &= a_- \\ A_{j,j+1} &= a_+ & A_{j,j+2} &= a_{++} \\ B_{j,j-1} &= b_{--} & B_{j,j} &= b_- \\ B_{j,j+1} &= b_+ & B_{j,j+2} &= b_{++} \end{aligned} \tag{14.6}$$

then the matrix coefficients turn out to be

$$\begin{aligned}
a_{--} &= -\frac{\alpha}{2h^3} & a_{-} &= \frac{1}{2\tau} + \frac{3\alpha}{2h^3} - \frac{(\bar{y}_- + \bar{y}_+)}{4h} \\
a_{+} &= \frac{1}{2\tau} - \frac{3\alpha}{2h^3} + \frac{(\bar{y}_- + \bar{y}_+)}{4h} & a_{++} &= \frac{\alpha}{2h^3} \\
b_{--} &= \frac{\alpha}{2h^3} & b_{-} &= \frac{1}{2\tau} - \frac{3\alpha}{2h^3} + \frac{(\bar{y}_- + \bar{y}_+)}{4h} \\
b_{+} &= \frac{1}{2\tau} + \frac{3\alpha}{2h^3} - \frac{(\bar{y}_- + \bar{y}_+)}{4h} & b_{++} &= -\frac{\alpha}{2h^3}
\end{aligned} \tag{14.7}$$

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).

P14.1 Derive the formulas in Eq. (14.7) for the a and b coefficients using the finite-difference approximations to the three terms in the Korteweg-deVries equation given in Eqs. (14.2)-(14.4).

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:

$$\mathbf{A} = \begin{bmatrix}
a_- & a_+ & a_{++} & 0 & 0 & ... & 0 & a_{--} \\
a_{--} & a_- & a_+ & a_{++} & 0 & ... & 0 & 0 \\
0 & a_{--} & a_- & a_+ & a_{++} & ... & 0 & 0 \\
. & . & . & . & ... & . & . & . \\
0 & ... & 0 & 0 & a_{--} & a_- & a_+ & a_{++} \\
a_{++} & 0 & ... & 0 & 0 & a_{--} & a_- & a_+ \\
a_+ & a_{++} & 0 & 0 & ... & 0 & a_{--} & a_-
\end{bmatrix} \tag{14.8}$$

$$\mathbf{B} = \begin{bmatrix}
b_- & b_+ & b_{++} & 0 & 0 & ... & 0 & b_{--} \\
b_{--} & b_- & b_+ & b_{++} & 0 & ... & 0 & 0 \\
0 & b_{--} & b_- & b_+ & b_{++} & ... & 0 & 0 \\
. & . & . & . & ... & . & . & . \\
0 & ... & 0 & 0 & b_{--} & b_- & b_+ & b_{++} \\
b_{++} & 0 & ... & 0 & 0 & b_{--} & b_- & b_+ \\
b_+ & b_{++} & 0 & 0 & ... & 0 & b_{--} & b_-
\end{bmatrix} \tag{14.9}$$

P14.2 Discuss these matrices with your lab partner and convince yourselves that this structure correctly models a periodic system (it may help to think about the computing grid as a circle with $x_1 = x_{N+1}$.)

An easy way to load the coefficients in this way is to invent integer arrays `jmm, jm, jp, jpp` corresponding to the subscripts $--$, $-$, $+$, and $++$ used in the coefficients above. These arrays are built by these four lines of Matlab code below. Run them and verify that they produce the correct "wrap-around" integer arrays to load **A**, as shown in the `for` loop below.

```
N=10; jm=1:N; jp=mod(jm,N)+1; jpp=mod(jm+1,N)+1; jmm=mod(jm-2,N)+1;

for j=1:N
   A(j,jmm(j))=...; % a(--)
   A(j,jm(j))=...;  % a(-)
   A(j,jp(j))=...;  % a(+)
   A(j,jpp(j))=...; % a(++)
end
```

OK, we are almost ready to go. All we need to settle now is what to do with $\bar{y}$. To properly center Crank-Nicolson in time between t_n and t_{n+1} we need $\bar{y} = y^{n+1/2}$, but this is not directly possible. But if we use a predictor-corrector technique like we did in the last lab, we can approximately achieve this goal. It goes like this.

We will apply Crank-Nicolson twice in each time step. In the first step (the predictor step) we simply replace $\bar{y}$ with y^n, the present set of values of y, and call the resulting new value (after Crank-Nicolson is used) $\tilde{y}^{n+1}$, the predicted future value. In the second step we combine this predicted value with the current value to approximately build $\bar{y}^{n+1/2}$ using $\bar{y}^{n+1/2} \approx (y^n + \tilde{y}^{n+1})/2$, then rebuild **A** and **B** and do Crank-Nicolson again.

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 Matlab script `kdv.m` 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.

Listing 14.1 (kdv.m)

```
% Korteweg-deVries equation on a periodic
% cell-centered grid using Crank-Nicolson
clear; close all;

N=500; L=10; h=L/N; x=h/2:h:L-h/2;
x=x'; % turn x into a column vector

alpha=input(' Enter alpha - ')

ymax=input(' Enter initial amplitude of y - ')

% load an initial Gaussian centered on the computing region
```

```
y=ymax*exp(-(x-.5*L).^2);

% choose a time step
tau=input(' Enter the time step - ')

% select the time to run
tfinal=input(' Enter the time to run - ') Nsteps=ceil(tfinal/tau);

iskip=input(' Enter the plot skip factor - ')

% Initialize the parts of the A and B matrices that
% do not depend on ybar and load them into At and Bt.
% Make them be sparse so the code will run fast.

At=sparse(N,N); Bt=At;

% Build integer arrays for handling the wrapped points at the ends
% (periodic system)

jm=1:N; jp=mod(jm,N)+1; jpp=mod(jm+1,N)+1; jmm=mod(jm-2,N)+1;

% load the matrices with the terms that don't depend on ybar
for j=1:N
    At(j,jmm(j))=-0.5*alpha/h^3;
    At(j,jm(j))=0.5/tau+3/2*alpha/h^3;
    At(j,jp(j))=0.5/tau-3/2*alpha/h^3;
    At(j,jpp(j))=0.5*alpha/h^3;
    Bt(j,jmm(j))=0.5*alpha/h^3;
    Bt(j,j)=0.5/tau-3/2*alpha/h^3;
    Bt(j,jp(j))=0.5/tau+3/2*alpha/h^3;
    Bt(j,jpp(j))=-0.5*alpha/h^3;
end

for n=1:Nsteps

    % do the predictor step
    A=At;B=Bt;
    % load ybar, then add its terms to A and B
    ybar=y;
    for j=1:N
        tmp=0.25*(ybar(jp(j))+ybar(jm(j)))/h;
        A(j,jm(j))=A(j,jm(j))-tmp;
        A(j,jp(j))=A(j,jp(j))+tmp;
        B(j,jm(j))=B(j,jm(j))+tmp;
```

```
        B(j,jp(j))=B(j,jp(j))-tmp;
    end

    % do the predictor solve
    r=B*y;
    yp=A\r;

    % corrector step
     A=At;B=Bt;
     % average current and predicted y's to correct ybar
     ybar=.5*(y+yp);
     for j=1:N
        tmp=0.25*(ybar(jp(j))+ybar(jm(j)))/h;
        A(j,jm(j))=A(j,jm(j))-tmp;
        A(j,jp(j))=A(j,jp(j))+tmp;
        B(j,jm(j))=B(j,jm(j))+tmp;
        B(j,jp(j))=B(j,jp(j))-tmp;
    end

    % do the final corrected solve
    r=B*y;
    y=A\r;

    if rem(n-1,iskip)==0
      plot(x,y)
      xlabel('x');ylabel('y')
      pause(.1)
    end

end
```

Solitons

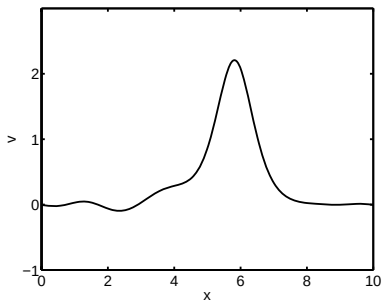


Figure 14.2 A Gaussian pulse after 1 second of propagation by the Korteweg-deVries equation (Problem 14.3)

P14.3 (a) Run `kdv.m` with $\alpha = 0.1$, $y_{\max} = 2$, $\tau = 0.5$, $t_{\text{final}} = 100$, 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 $t_{\text{final}} = 10$ for both runs and `iskip` 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 14.3 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.m` in such a way that we can see the effect of each of these terms separately.

P14.4 (a) Dispersion (wave-spreading) dominates: Run `kdv.m` with $\alpha = 0.1$, $y_{\max} = 0.001$, $\tau = 0.1$, and $t_{\text{final}} = 10$. 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 Schrödinger'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.m` with $\alpha = 0.01$, $y_{\max} = 2$, $\tau = 0.01$, and $t_{\text{final}} = 10$. (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.)

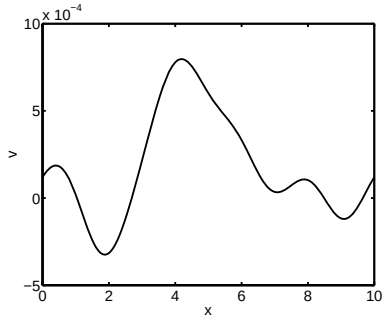


Figure 14.3 Dispersion dominates (Problem 14.4(a).): after 3 seconds of time.

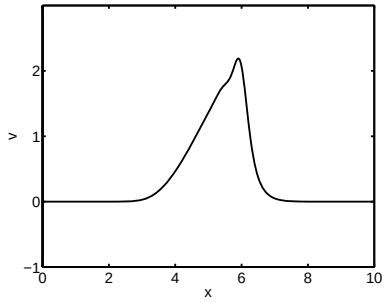


Figure 14.4 Steepening dominates (Problem 14.4(b).): after 0.5 seconds of time.

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:

$$y(x,t) = \frac{12k^2\alpha}{\cosh^2\left(k(x - x_0 - 4\alpha k^2 t)\right)} \tag{14.10}$$

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

P14.5 (a) Use Mathematica to show that this expression does indeed satisfy the Korteweg-deVries equation.

(b) Now replace the Gaussian initial condition in `kdv.m` 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 14.3. 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

$$c_{\text{soliton}} = 4\alpha k^2 \tag{14.11}$$

Verify by numerical experimentation that your soliton moves at this speed. The commands `max` and `polyfit` are useful in this part.

P14.6 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 14.5 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?

Appendix A

Fourier Methods for Solving the Wave Equation

The wave equation is often solved analytically using Fourier methods. In this method, you first consider the behavior of a plane wave solution to the wave equation in the form $y(t,x) = Ae^{i(\omega t - kx)}$. This type of plane wave satisfies linear wave equations for arbitrary frequencies ω as long as k is chosen appropriately. The relationship between k and ω is referred to as the *dispersion relation*, and encodes much of the physics describing wave propagation.

Once you have the appropriate dispersion relation for the system, you can calculate wave propagation through the system using Fourier theory. To do this, you first recall that if you add a bunch of sinusoids of the form $g(\omega)e^{i\omega t}$ with appropriately chosen amplitudes $g(\omega)$, you can get them to interfere and produce any wave form with any temporal shape you like. If you know the temporal shape of the pulse at a given spatial point, say $x = 0$, you can get the amplitudes $g(\omega)$ for your coefficients by taking a Fourier transform:

$$g(\omega, x=0) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t, x=0) e^{i\omega t} dt \tag{A.1}$$

Then, to calculate the temporal form of the pulse at points besides $x = 0$ you add up traveling waves of the form $g(\omega)e^{i(kx-\omega t)}$. The different frequency components travel at their individual phase velocities, given by $v_p = \omega/k$. If there is no dispersion, k and ω are related by the simple dispersion relation $k = \omega/c$ (where c is a constant with units of speed), so the phase velocity is the same for all frequency components: $v_p = c$. However, in many situations we have a different dispersion relation $k(\omega)$ because different frequency components move at a different speeds. As the frequency components shift phase relative to one another, the shape of the pulse evolves.

If the medium responds linearly to the waves, Fourier analysis provides an easy way to add up all of these frequency components with different phase velocities. If we freeze time, the phase change for each frequency component due to moving to a different point in space is given by $k(\omega)x$. In complex notation, this means that the spectrum at a point x is related to the spectrum at $x = 0$ through

$$g(\omega, x) = g(\omega, x=0) e^{ik(\omega)x} \tag{A.2}$$

If we take an inverse Fourier transform of this spectrum

$$f(t,x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(\omega, x) e^{-i\omega t} d\omega = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(\omega, x=0) e^{i(kx-\omega t)} d\omega \tag{A.3}$$

we can find the form of the pulse at an arbitrary x. Note that the last expression explicitly has the sum of traveling waves that we talked about conceptually.

PA.1 Let's use this technique to model the propagation of a water wave. Since Eqs. (A.1) and (A.3) are written as standard Fourier integrals, use the Matlab functions `ft.m` and `ift.m` as discussed in *Introduction to Matlab* rather than the regular `fft`. Use a time array that is 200 s wide with $N = 2^{16} = 65536$, like this

```
N = 2^16; tmax = 200; tau = tmax/(N-1); t=0:tau:(N-1)*tau; dw = 2*pi/tmax;
w = -(N/2)*dw:dw:dw*(N/2-1);
```

Notice that we've chosen a symmetric ω array because we will be using `ft.m` and `ift.m`.

Make your initial water pulse using $f(t, x = 0) = e^{-(t-20)^2/0.5^2}$. This is a bump of water, sort of like the wave that a speedboat pushes away from it as it drives. (A boat makes two waves that propagate away from each other to make the wake—we'll just look at one of the waves.) Plot $f(t, x = 0)$ to see its shape, and then, find the spectrum $g(\omega, x = 0)$ of this pulse using `ft.m`.

Now find the spectrum of the pulse at $x = 30$ by multiplying $g(\omega, x = 0)$ by $e^{ik(\omega)x}$. The dispersion relation for water waves is

$$\omega^2 = gk\tanh(kd) \tag{A.4}$$

where g is the acceleration of gravity and d is the depth of the water. Since we need $k(\omega)$, Eq. (A.4) needs to be solved numerically. You know how to solve this equation using `fzero`, but since our ω array has 65,536 elements it would take a while to just do each element directly. In the interest of time, we've given you the matlab function `waterk.m` below that calculates $k(\omega)$ for you. Look it over briefly to understand how it works, then just use it.

Finally, find $f(t, x = 30)$ by taking the inverse Fourier transform of $g(\omega, x = 30)$ (using `ift.m`). Plot $f(t, x = 0)$ and $f(t, x = 30)$ on the same axis and explain what you see. Decide whether high or low frequency water waves travel faster in this model.

You have probably seen the dispersion of water waves behavior before. When the wave that defines the edge of a wake leaves the boat, it is mostly just a single bump of water. But it takes a lot of frequencies to make that bump. As the wave propagates, each frequency component travels at its own phase velocity and the bump spreads out and develops ripples. At the shore you get a long train of ripples rather than a single bump.

Listing A.1 (waterk.m)

```
% function to calculate the water dispersion relation
function k = waterk(w,g,d)

% Make an approximate k. Error is less that 1e-15 for |kd| > 20
```

```
k = w.*abs(w) / g;

% Fix the errors for |kd| < 20 using dsolve
disp('Refining the dispersion relation.  Please be patient.'); disp('If your
w array has a lot of small values, this can take a while.')

for istep = 1:length(k)
    if (abs(k(istep)*d) < 20)
        eq = @(k) w(istep).^2-g*k*tanh(k*d);
        k(istep) = fzero(eq, k(istep));
    end
end
```

Appendix B

Implicit Methods in 2-Dimensions: Operator Splitting

Consider the diffusion equation in two dimensions, simplified so that the diffusion coefficient is a constant:

$$\frac{\partial T}{\partial t} = D\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) \tag{B.1}$$

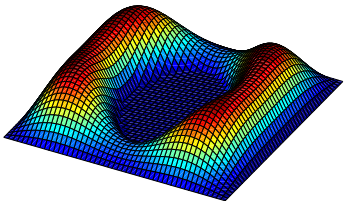


Figure B.1 Diffusion in 2-dimensions with an elliptical central region set to zero.

If we define the finite difference operators $\mathscr{L}_x$ and $\mathscr{L}_y$ as follows,

$$\mathscr{L}_x T_{j,k} = \frac{T_{j+1,k} - 2T_{i,j} + T_{j-1,k}}{h^2} \quad ; \quad \mathscr{L}_y T_{j,k} = \frac{T_{j,k+1} - 2T_{j,k} + T_{j,k-1}}{h^2} \tag{B.2}$$

then it is easy to write down the Crank-Nicolson algorithm in 2-dimensions (We have suppressed the spatial subscripts to avoid clutter):

$$\frac{T^{n+1} - T^n}{\tau} = \frac{D}{2}\left(\mathscr{L}_x T^{n+1} + \mathscr{L}_x T^n + \mathscr{L}_y T^{n+1} + \mathscr{L}_y T^n\right) \tag{B.3}$$

If we could solve simply for $T^{n+1} = T^{n+1}_{i,j}$, as we did in the Lab 8, we would be on our way. But, unfortunately, the required solution of a large system of linear equations for the unknown T^{n+1}_j's is not so simple.

To see why not, suppose we have a 100×100 grid, so that there are 10,000 grid points, and hence 10,000 unknown values of $T^{n+1}_{i,j}$ to find. And, because of the difficulty of numbering the unknowns on a 2-dimensional grid, note that the matrix problem to be solved is not tridiagonal, as it was in 1-dimension. Well, even with modern computers, solving 10,000 equations in 10,000 unknowns is a pretty tough job, so it would be better to find a clever way to do Crank-Nicolson in 2-dimensions.

One such way is called *operator splitting*, and was invented by Douglas[1] and by Peaceman and Rachford.[2] The idea is to turn each time step into two half-steps, doing a fully implicit step in x in the first half-step and another one in y in the second half-step. It looks like this:

$$\frac{T^{n+1/2} - T^n}{\tau/2} = D\left(\mathscr{L}_x T^{n+1/2} + \mathscr{L}_y T^n\right) \tag{B.4}$$

$$\frac{T^{n+1} - T^{n+1/2}}{\tau/2} = D\left(\mathscr{L}_x T^{n+1/2} + \mathscr{L}_y T^{n+1}\right) \tag{B.5}$$

[1] Douglas, J., Jr., *SIAM J.* , **9**, 42, (1955)

[2] Peaceman, D. W. and Rachford, H. H. , *J. Soc. Ind. Appl. Math.* , **3**, 28, (1955)

If you stare at this for a while you will be forced to conclude that it doesn't look like Crank-Nicolson at all.

PB.1 Use Mathematica to eliminate the intermediate variable $T^{n+1/2}$ and show that the algorithm gives

$$\frac{T^{n+1}-T^n}{\tau} = \frac{D}{2}\left(\mathscr{L}_x T^{n+1} + \mathscr{L}_x T^n + \mathscr{L}_y T^{n+1} + \mathscr{L}_y T^n\right) - \frac{D^2\tau^2}{4}\mathscr{L}_x\mathscr{L}_y\left(\frac{T^{n+1}-T^n}{\tau}\right) \tag{B.6}$$

which is just like 2-dimensional Crank-Nicolson (Eq. (B.3)) except that an extra term corresponding to

$$\frac{D^2\tau^2}{4}\frac{\partial^5 T}{\partial x^2 \partial y^2 \partial t} \tag{B.7}$$

has erroneously appeared. But if $T(x,y,t)$ has smooth derivatives in all three variables this error term is second-order small in the time step τ.

We saw in Lab 8 that the accuracy of Crank-Nicolson depends almost completely on the choice of h and that the choice of τ mattered but little. This will not be the case here because of this erroneous term, so τ must be chosen small enough that $D^2\tau^2/\ell^4 \ll 1$, where ℓ is the characteristic distance over which the temperature varies (so that we can estimate $\mathscr{L}_x\mathscr{L}_y \approx 1/\ell^4$.) Hence, we have to be a little more careful with our time step in operator splitting.

So why do we want to use it? Notice that the linear solve in each half-step only happens in one dimension. So operator splitting only requires many separate tridiagonal solves instead of one giant solve. This makes for an enormous improvement in speed and makes it possible for you to do the next problem.

PB.2 Consider the diffusion equation with $D = 2.3$ on the xy square $[-5,5] \times [-5,5]$. Let the boundary conditions be $T = 0$ all around the edge of the region and let the initial temperature distribution be

$$T(x,y,0) = \cos\left(\frac{\pi x}{10}\right)\cos\left(\frac{\pi y}{10}\right) \tag{B.8}$$

First solve this problem by hand using separation of variables (let $T(x,y,t) = f(t)T(x,y,0)$ from above, substitute into the diffusion equation and obtain a simple differential equation for $f(t)$.) Then modify your Crank-Nicolson script from Lab 7 to use the operator splitting algorithm described in Problem B.1. Show that it does the problem right by comparing to the analytic solution. Don't use ghost points; use a grid with points right on the boundary instead.

PB.3 Modify your script from Problem B.2 so that the normal derivative at each boundary vanishes. Also change the initial conditions to something more interesting than those in B.2; it's your choice.

PB.4 Modify your script from Problem B.2 so that it keeps $T = 0$ in the square region $[-L, 0] \times [-L, 0]$, so that $T(x, y)$ relaxes on the remaining L-shaped region. Note that this does not require that you change the entire algorithm; only the boundary conditions need to be adjusted.

Appendix C

Tri-Diagonal Matrices

Many of the algorithms that we used in the labs involved solving matrix equations of the form

$$\mathbf{A}x = r \tag{C.1}$$

where the matrix **A** had a tri-diagonal form (i.e. it has non-zero elements only on the diagonal and one element to either side). The full matrix **A** can get very large if you need a lot of grid points. However, it is possible to make the algorithm more efficient because of the simple form of **A**. Because only the main diagonal and its nearest diagonal neighbors are non-zero, it is possible to do Gauss elimination using just these three sets of numbers while ignoring the rest of the matrix. The Matlab script below called `tridag.m` does this and its calling syntax is `tridag(A,r)`. The input matrix `A` has N rows, where N is the number of grid points, and it has three columns, one for each of the diagonals. The vector `r` is the single-column right-hand side vector in the linear system of Eq. (C.1) The routine `tridag` first puts the tridiagonal matrix in upper triangular form. In this simple form it is easy to solve for x_N because now the bottom row of the matrix is simple, and the algorithm then just works its way back up to x_1. This algorithm was invented a long time ago and is usually called the *Thomas algorithm*.

To use it, first load the diagonal just below the main diagonal into `A(:,1)` (put a zero in `A(1,1)` because the lower diagonal doesn't have a point there); then fill `A(:,2)` with the main diagonal, and finally put the upper diagonal into `A(:,3)` (loading a zero into `A(N,3)`.) Then do the solve with the command

```
x=tridag(A,r);
```

Here is the script `tridag(A,r)`.

Listing C.1 (tridag.m)

```
function x = tridag(A,r)

% Solves A*x=r where A contains the three diagonals
% of a tridiagonal matrix.  A contains the three
% non-zero elements of the tridiagonal matrix,
% i.e., A has n rows and 3 columns.
% r is the column vector on the right-hand side

% The solution x is a column vector

% first check that A is tridiagonal and compatible
% with r
```

```
[n,m]=size(A); [j,k]=size(r);

if n ~= j
    error(' The sizes of A and r do not match')
end

if m ~= 3
    error(' A must have 3 columns')
end

if k ~= 1
    error(' r must have 1 column')
end

% load diagonals into separate vectors
a(1:n-1) = A(2:n,1); b(1:n) = A(1:n,2); c(1:n-1) = A(1:n-1,3);

% forward elimination
for i=2:n
    coeff = a(i-1)/b(i-1);
    b(i) = b(i) - coeff*c(i-1);
    r(i) = r(i) - coeff*r(i-1);
end

% back substitution
x(n) = r(n)/b(n); for i=n-1:-1:1
    x(i) = (r(i) - c(i)*x(i+1))/b(i);
end

x = x.';   % Return x as a column vector

return;
```

Appendix D

Answers to Review Problems

Problem 0.1(a)

Listing D.1 (lab0p1a.m)

```
% Lab Problem 0.1a, Physics 430

for n=2:3:101
    if mod(n,5)==0
        fprintf(' fiver: %g \n',n);
    end
end
```

Problem 0.1(b)

Listing D.2 (lab0p1b.m)

```
% Lab Problem 0.1b, Physics 430

N=input(' Enter N - ') sum=0; for n=1:N
    sum=sum+n;
end

fprintf(' Sum = %g  Formula = %g \n',sum,N*(N+1)/2);
```

Problem 0.1(c)

Listing D.3 (lab0p1c.m)

```
% Lab Problem 0.1c, Physics 430

x=input(' Enter x - ')

sum=0;

for n=1:1000
    sum=sum + n*x^n;
end
```

```
fprintf(' Sum = %g  Formula = %g \n',sum,x/(1-x)^2);
```

Problem 0.1(d)

Listing D.4 (lab0p1d.m)

```
% Lab Problem 0.1d, Physics 430

sum=0; n=0;

term=1;

while term > 1e-8
    n=n+1;
    term=n*x^n;
    sum=sum+term;
end
fprintf(' Sum = %g  Formula = %g \n',sum,x/(1-x)^2);
```

Problem 0.1(e)

Listing D.5 (lab0p1e.m)

```
% Lab Problem 0.1e, Physics 430

% initialize the counter n, the term to be added, and the sum
n=1; term=1.; sum=term;

while term>1e-6
    n=n+1;
    term=1/n^2;
    sum=sum+term ;
end

fprintf(' Sum/(pi^2/6) = %g \n',sum*6/pi^2);
```

Problem 0.1(f)

Listing D.6 (lab0p1f.m)

```
% Lab Problem 0.1f, Physics 430
```

```
prod=1;

for n=1:100000
    term=1+1/n^2;
    prod=prod*term;
    if term-1 < 1e-8
        break
    end
end

fprintf(' Product = %g  Formula = %g \n',prod,sinh(pi)/pi);
```

Problem 0.1(g)

Listing D.7 (lab0p1g.m)

```
% Lab Problem 0.1g, Physics 430

x1=1;x2=1;x3=1;

chk=1;

while chk > 1e-8

    x1new=exp(-x1);
    x2new=cos(x2);
    x3new=sin(2*x3);

    chk=max([abs(x1new-x1),abs(x2-x2new),abs(x3-x3new)]);

    x1=x1new;
    x2=x2new;
    x3=x3new;

end

fprintf(' x1 = %g x2 = %g  x3 = %g \n',x1,x2,x3);
```

Appendix E

Glossary of Terms

by Dr. Colton and students in the Winter 2010 semester

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, ..., 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, ..., 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 cell-edge grid ranging from 0 to 10 by steps of 1 would have points at 0, 1, ... , 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 a formula for calculating 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 $\mathbf{g}$ 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 $\mathbf{V}^{new} = \mathbf{L} \times \mathbf{V}^{old} + \mathbf{r}$ by iterations, by shifting the eigenvalues. This can be done via solving the equivalent problem of $\mathbf{V}^{new} = w \times$ [RHS of previous equation] + $(1 - w) \times \mathbf{V}^{old}$. 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.

Index