COMPUTATIONAL PHYSICS 430

PARTIAL DIFFERENTIAL EQUATIONS

Ross L. Spencer

N245 ESC 378-2341 ross_spencer@byu.edu

Department of Physics and Astronomy

Brigham Young University

© 2001 Ross L. Spencer 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. I assume that you already know how to program in Maple and Matlab, which you should have learned in Physics 230 and 330. These are serious prerequisites; you won't survive this course unless you can program in both. I also assume that you have studied mathematical physics at the level of Physics 318.

This material is quite a bit more demanding than the topics you studied in Physics 230 and 330, so you will need to come to class better prepared. I expect you to have read through each lab before class and will pass out a short reading quiz at the beginning of each period to encourage you to meet this requirement. Your performance on these quizzes will affect your grade. If you just show up and try to learn as you go you will most likely get behind.

Please work with a lab partner as you do the assigned exercises in each laboratory. When you have completed a problem, call a TA over and use the computer screen to explain to them what you have done. We will not be printing. The course has been designed so that almost all of the work can be done during class time, so that you have access to someone who can help you with difficulties that you may encounter. It will take a lot longer to do these exercises if you are completely on your own.

There will be three short oral exams during the semester which will be held in my office. During these exams I will give you some problems to do to see how you approach them, and I will also have you program a little so that I can assess your level of mastery of the course material. There will also be a final exam during our regularly scheduled exam time. It will not be a programming exam. Instead, it will consist of short essay questions and problems for which I will ask you questions about possible approaches to the problem and about what difficulties might be encountered.

Your grade will be determined by how well you keep up with the assignments, by your performance on the reading quizzes, by how well you do on the oral exams, and by how well you do on the final. Students who keep up and do OK on the final will receive a grade of A-. An A requires evidence of superior skill and mastery of the material.

Contents

1 Grids and Loops

1.1 When we solved differential equations in Physics 330 we were usually moving something forward in time, so you may think that differential equations always “flow”. This is not true. If we solve a spatial differential equation, for instance, 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 we seek to find the correct shape by somehow finding the height of the chain at each link.

In this course we will be solving partial differential equations, which usually means that the desired solution is a function of both x, which just sits, and time t, which flows. And when we solve problems like this in this course we will be using *spatial grids*, to represent the x-part that doesn't flow, so let's get comfortable with them before we proceed with the job of using them to solve partial differential equations.

(a) You have already used grids in Matlab to do simple jobs like plotting functions and doing integrals numerically. For instance, this kind of code should be familiar to you: (type each line and watch it execute in Matlab.)

```
N=100;  % define the number of steps in x
a=0;b=pi;  % define the left and right endpoints
h=(b-a)/N; % get the step size
x=a:h:b;   % get the grid (array of points between a and b)
f=sin(x).*sinh(x);  % define the function
plot(x,f)  ;  % plot the function
```

This kind of grid is called a *cell-edge* grid because the interval between x_1 and x_2 is divided into N cells with the grid points $x_1 = a$, $x_2 = a + h$, ... $x_{N+1} = b$ at the edges of the cells. Verify by using Matlab's `whos` command that the number of points in this x-grid between a and b is $N + 1$. Then draw a rough sketch showing the grid points for $N = 3$ and see if you can tell why there aren't N grid points, as you might have thought.

Note: if you want the number of grid points to be N then you have to define the step size h this way:

```
h=(b-a)/(N-1);
```

(b) Another commonly used grid is the *cell-center* grid. This grid divides the interval from a to b into N cells just like the cell-edge grid in part (a), but the grid points sit at the centers of each cell: $x_1 = a + h/2$, ... $x_N = b - h/2$. Write a script like the one in part (a) that uses Matlab's colon command to build a 5000-point cell-center grid between $x = 0$ and $x = 2$ and fill `f` with the function $f(x) = \cos x$. 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;
```

Verify that this result is quite close to the exact answer obtained by integration: $A = \int_0^2 \cos x dx$.

(c) Another grid that we will often use this semester is the cell-center grid with *ghost points*. This is almost the same as the cell-center grid you used in part (b), but this one has an extra cell to the left of a and another one to the right of b, like this:

```
x=a-h/2:h:b+h/2;  % cell-center grid with ghost points
```

If N is the number of cells between a and b, how many grid points are there in this grid?

Build this grid with $a = 0$ and $b = \pi/2$, then define the function $f(x) = \sin x$ on this grid. Now look carefully at the function values at the first two grid points (f_1, f_2) and at the last two grid points (f_{N+1}, f_{N+2}). The function $\sin x$ has the property that $f(0) = 0$ and $f'(\pi/2) = 0$. Since the cell-center grid doesn't have points at the ends of the interval, these boundary conditions on the function can't be represented by values at the ends (and it takes more than one point to do a derivative, anyway.) Explain how the function values at the ghost points help to get these boundary conditions right.

(d) We will also do problems this semester in two spatial dimensions, x and y. Matlab has a very nice way of representing grids in two-dimensional spaces, as long as the two-dimensional region is rectangular in shape. Consider a 2-d rectangle defined by $x \in [a, b]$ and $y \in [c, d]$. Make 50-point cell-edge grids in both x and y using $a = 0$, $b = 2$, $c = -1$, $d = 3$. Then use Matlab's `meshgrid` command to make 2-d grids X and Y from the 1-d grids x and y like this.

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

Examine the contents of X and Y long enough that you can explain what this command does. Then use these 2-d grids and Matlab's `surf` command to make surface plots of the following two functions of x and y:

$$f(x,y) = e^{(-(x^2+y^2))} \cos\left(5\sqrt{x^2+y^2}\right) \quad ; \quad g(x,y) = J_0(x^2+3y^2) \tag{1}$$

Properly label the x and y axes. Use online help and remember to use 'dotted' operators when you build these functions using X and Y in place of x and y.

1.2 This next section is in this lab because if you are like most students, loops and logic give you lots of trouble. We will be using these programming tools extensively this semester, so let's do some practice here. You will probably need to use online help quite a bit and also call the TA over to explain things.

(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 an integer value that the program receives via the `input` command. Verify by numerical experimentation that the formula

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

is correct

(c) For various values of x perform the sum

$$\sum_{n=1}^{1000} nx^n \tag{3}$$

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

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6} \tag{4}$$

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:

$$\prod_{n=1}^{\infty} \left(1 + \frac{1}{n^2}\right) = \frac{\sinh \pi}{\pi} \tag{5}$$

(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.

$$x_{n+1} = e^{-x_n} \quad ; \quad x_{n+1} = \cos x_n \quad ; \quad x_{n+1} = \sin 2x_n \tag{6}$$

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.

(h) If you have arrived here and there is lots of time left in the lab period, go on to Lab 2.

2 Derivatives on Grids

2.1 In calculus books the derivative is defined by the *forward difference* formula

$$f'(x) = \frac{f(x+h) - f(x)}{h} \tag{7}$$

as h goes to zero. The word "forward" refers to the way this formula reaches forward from x to $x+h$ to calculate the slope. On a numerical computer there is a limit to how small h can be and still have $x+h$ be different from x, so derivatives on such systems can only be approximate. (Take a minute and experiment with this Matlab command:

```
h=1e-17;(1+h);ans-1
```

Try various small values of h and explain why $(1+h)-1$ doesn't always give you h back. This problem which occurs in calculations with real numbers on all digital computers is called *roundoff*.)

But given this limitation we want to be as accurate as possible, so we prefer to use the best derivative formulas available. The forward difference formula isn't one of them

The best first derivative formula using two function values is the *centered difference* formula:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{8}$$

which you may remember from Physics 330. It is called "centered" because the point x at which we want the slope is centered between the places where the function is evaluated. The corresponding centered second derivative formula is

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

You will use Maple to derive both of these formulas in part (b) of this problem, but in this part I just want you to understand how to use them.

Using the function $f(x) = e^x$ and $h = 0.1,\ 0.01,\ 0.001$, check to see how well the forward and centered difference formulas, and the second derivative formula, do at $x = 0$ (use Matlab.) Note that at $x = 0$ the exact values of both f' and f'' are equal to 1.

2.2 Now let's use Maple to see where these approximate derivative formulas come from. The starting point is Taylor's expansion of the function f about the point x

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \ ... \ + f^{(n)}(x)\frac{h^n}{n!} + \ ... \tag{10}$$

This series usually converges quite rapidly if h is small.

(a) Let's start our study of the origin of the difference formulas in Problem 2.1. with the forward difference approximation to $f'(x)$, Eq. (**??**), by using the Taylor expansion in place of $f(x+h)$:

$$f'(x) \approx \frac{[f(x) + f'(x)h + f''(x)h^2/2] - f(x)}{h} = f'(x) + f''(x)h/2 \tag{11}$$

(note that the expansion of $f(x+h)$ has been terminated at the $f''(x)$ term because higher order terms just give smaller corrections to the result.) From this formula we see that the forward difference does indeed give the first derivative back, but that it doesn't quite get it right because of the extra term which is proportional to h. But, of course, 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 $f'(x)$.

Now let's perform the same analysis on the centered difference formula to see why it is better. Using the Taylor expansion in Eq. (**??**) yields

$$f'(x) \approx$$
$$\left([f(x)+f'(x)h+f''(x)h^2/2+f'''(x)h^3/6]-[f(x)-f'(x)h+f''(x)h^2/2-f'''(x)h^3/6]\right)/(2h)$$
$$= f'(x)+f'''(x)h^2/6 \tag{12}$$

Notice that for this approximate formula the error term is much smaller, only of order h^2. To get a feel why this is so much better, imagine decreasing 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 100. Throughout this course we will try to use centered formulas whenever possible.

Take a minute now and verify that these error estimates agree with the numerical testing you did with e^x in Problem 2.1.

(b) To study the second derivative formula let's try something a little more general. Let's use Maple and the Taylor expansion to find approximate formulas for $f'(x)$ and $f''(x)$ from the three data points $[x, f(x)]$, $[x+p, f(x+p)]$, and $[x-m, f(x-m)]$, where p and m are small positive numbers.

(i) First use Maple's `taylor` command to generate the expansions of $f(x+p)$ and $f(x-m)$ through the $f''''(x)$ term, then change the names of the derivatives to variables with which Maple can do algebra, like this:

```
eqplus:=fplus=f+fp*p+fpp*p^2/2 + fppp*p^3/6+fpppp*p^4/24
```

where `fp` stands for f', `fpp` stands for f'', etc.. Make a similar equation called `eqminus` for $f(x-m)$ that also contains the derivative variables `fp`, `fpp`, etc..

(ii) Now use `solve` to solve these two equations for the first derivative `fp` and the second derivative `fpp`. The terms involving `fplus`, `f`, and `fminus` give the approximate derivative formula to use, while the terms involving the higher derivatives `fppp` and `fpppp` tell you what the order of the error is.

(iii) Examine your solution from (ii) and write down the approximate formulas for the first and second derivatives. Then examine them further to show that the error in the first derivative formula using these three points is of second order in the step sizes p and m (note that pm is also second order) but that the second derivative formula only has a second order error if we set $p = m$, in which case we get Eq. (**??**).

(iv) Suppose you have function values $f(x-3h/2)$, $f(x-h/2)$, $f(x+h/2)$, and $f(x+3h/2)$. Use Maple and the procedure in (i)-(iii) to find a formula for the third derivative and the order of its error ($O(h)$, $O(h^2)$, etc.. You will need to use a Taylor expansion that goes clear out to the fifth derivative.

(If you look at the centered formulas for f', f'', and now f''', do you see a pattern?)

2.3 Using the ideas from 2.1 and 2.2 we will now see how Matlab can differentiate a function. Since Matlab is purely numerical, the only way it can represent a function is through arrays of numbers. For instance the cosine function on the interval [0,5] could be represented this way:

```
xmin=0;xmax=5;N=1001;h=(xmax-xmin)/(N-1); % set up the grid in x
x=xmin:h:xmax;  % build the x-array
f=cos(x); % build the function
plot(x,f);  % plot it
```

This works great for looking at the function, but what if we wanted to take its first and second derivatives? Using Matlab's colon command we can do it easily. The idea is to use a centered derivative formula at each point x in the array, except at the beginning and the end where these formulas reach outside the data set. (We will see shortly how to handle the end points.) Here is Matlab code to efficiently use the centered first and second derivative formulas at each interior point on the grid:

```
fp(2:N-1)=(f(3:N)-f(1:N-2))/(2*h);        % first derivative
fpp(2:N-1)=(f(3:N)-2*f(2:N-1)+f(1:N-2))/h^2;  % second derivative
```

Note how Matlab's colon operator allows us to do these calculations very compactly. Stare at these two lines of code and discuss them with your partner until you see that they represent Eqs. (**??**) and (**??**) correctly.

But as you can see on the left hand sides of these two lines of code, points 1 and N are not included in this calculation. Since the centered difference formulas don't work at the end points, about the best we can do is to *extrapolate* the interior values of the two derivatives to the end points.

If we extrapolate using a straight line (linear extrapolation) then we just need two nearby points and the formulas for the derivatives at the end points are:

```
fp(1)=2*fp(2)-fp(3);
fp(N)=2*fp(N-1)-fp(N-2);
fpp(1)=2*fpp(2)-fpp(3);
fpp(N)=2*fpp(N-1)-fpp(N-2);
```

If we want to extrapolate by using parabolas (quadratic extrapolation) we need to use three nearby points, like this:

```
fp(1)=3*fp(2)-3*fp(3)+fp(4);
fp(N)=3*fp(N-1)-3*fp(N-2)+fp(N-3);
fpp(1)=3*fpp(2)-3*fpp(3)+fpp(4);
fpp(N)=3*fpp(N-1)-3*fpp(N-2)+fpp(N-3);
```

(a) Use Maple to show that these are the correct parabolic extrapolation formulas to use at the endpoints by doing the parabolic fit symbolically and evaluating it at x_1 and at x_N.

Then make overlaid Matlab plots to see that `fp` and `fpp` on $x = [x_{min}, x_{max}]$ are indeed good approximations to the first and second derivatives of the cosine function.

(b) Now let's take a small detour and learn some wisdom about using these formulas on experimental data. Suppose you had acquired some data that you needed to numerically differentiate. Since it's real data there are random errors in the numbers, which we can model by using Matlab's random number function `rand` like this:

```
f=cos(x)+.001*rand(1,length(x));
```

So now f contains the cosine function, plus experimental error at the 0.1% level.

Repeat the two derivative calculations and the comparison plots on this "data" to verify that the following statement is true: "Differentiating your data is a bad idea, and differentiating it twice is even worse." If you can't avoid differentiating experimental data, you had better work pretty hard at getting the error down, or perhaps make a least-squares fit of your data to a smooth function, then differentiate the function.

(c) Load $f(x)$ with the Bessel function $J_0(x)$ and numerically differentiate it to obtain $f'(x)$ and $f''(x)$. Then compare the numerical derivatives with the exact derivatives obtained from Maple by making overlaid plots in Matlab.

3 Differential Equations on Grids

3.1 We are now ready to use these grid ideas to solve a problem very close to the main subject matter of this course: we are going to combine finite-difference approximations with linear algebra to solve differential equations. Consider the differential equation

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

Notice that instead of having initial conditions at $x = 0$, this differential equation has *boundary conditions*, meaning that the conditions are specified at both ends of the interval. This seemingly simple change in the boundary conditions means that Matlab's differential equation solvers (like `ode45`) are hard to use for problems like this. But if we use a grid and the ideas in 2.1-2.3 we can easily and naturally solve this differential equation numerically.

We begin by setting up a grid:

```
xmin=0;xmax=2;N=21;h=(xmax-xmin)/(N-1);
x=xmin:h:xmax;
```

We will be using lots of grids this semester; stare at this piece of code and make sure you understand what it does.

We now rewrite the differential equation as it would appear on the grid, with $y(x)$ replaced by $y_j = y(x_j)$ and with the second derivative rewritten using the centered finite difference equation:

$$\frac{y_{j+1} - 2y_j + y_{j-1}}{h^2} + 9y_j = x_j \tag{14}$$

Now let's think about this equation for a bit. First notice that it is not *an* equation; it is many equations, for 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. And because this equation involves y_{j-1}, y_j, and y_{j+1} for the interior grid points $j = 2..N-1$, it is really a system of $N-2$ coupled equations in the N unknowns $y_1...y_N$. 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:

$$y_1 = 0 \quad ; \quad y_N = 1 \tag{15}$$

which completes our system of N equations in N unknowns.

Before Matlab can solve this system we have to put it in matrix form, so here is the translation of the equations above into matrix form. Stare at the matrix equation below until you are convinced that it is equivalent to Eqs. (**??**) and (**??**). (To make your staring productive, mentally do each row of the matrix multiply below 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. Verify that you recover Eqs. (**??**) and (**??**).)

$$\begin{bmatrix} 1 & 0 & 0 & 0 & ... & 0 & 0 & 0 \\ \frac{1}{h^2} & -\frac{2}{h^2}+9 & \frac{1}{h^2} & 0 & ... & 0 & 0 & 0 \\ 0 & \frac{1}{h^2} & -\frac{2}{h^2}+9 & \frac{1}{h^2} & ... & 0 & 0 & 0 \\ . & . & . & . & ... & . & . & . \\ . & . & . & . & ... & . & . & . \\ . & . & . & . & ... & . & . & . \\ 0 & 0 & 0 & 0 & ... & \frac{1}{h^2} & -\frac{2}{h^2}+9 & \frac{1}{h^2} \\ 0 & 0 & 0 & 0 & ... & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ . \\ . \\ . \\ y_{N-1} \\ y_N \end{bmatrix} = \begin{bmatrix} 0 \\ h \\ 2h \\ . \\ . \\ . \\ (N-1)h \\ 0 \end{bmatrix} \tag{16}$$

Once we have the finite-difference approximation to the differential equation in this matrix form ($\mathbf{Ay} = \mathbf{b}$), a simple linear solve is all that is required to find the solution array y_j: `y=A\b`.

(a) Use Maple's `dsolve` command to solve the differential equation Eq. (**??**) at the beginning of this problem, then import the solution formula into a Matlab script that defines the grid above and plot the exact solution on the grid.

(b) Now load the matrix above and do the linear solve to obtain y_j and plot it on top of the exact solution 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. I think you'll be impressed at how well the numerical method works, if you use enough grid points.

(c) Solve the differential equation below both with Maple and by using the matrix method in Matlab and compare the two solutions.

$$y'' + \frac{1}{x}y' + (1 - \frac{1}{x^2})y = x \quad ; \quad y(0) = 0, \quad y(5) = 1 \tag{17}$$

(d) By making small changes in your script for (c), solve this differential equation with Matlab:

$$y'' + \sin(x)y' + e^x y = 1 \quad ; \quad y(0) = 0, \quad y(5) = 1 \tag{18}$$

(If you try to solve this equation with Maple it will fail, unless you use `dsolve(...type=numeric)` and increase `abserr` to about 10^{-4}.) How will you know that you have done the problem right in Matlab?

(e) Finally, I must confess that I 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 I have given you so far are easy because they are *linear* differential equations, which is why they can be translated into linear algebra problems. Here is a simple example of one that isn't linear:

$$y''(x) + \sin[y(x)] = 1 \quad ; \quad y(0) = 0, \quad y(3) = 0 \tag{19}$$

Work at turning this problem into a linear algebra problem just long enough to see that it can't be done. Then solve it by the method of shooting in the Maple textbook for Physics

230 as described in Problem 7.10 in that text. Set $y(0) = 0$ and $dy/dx = \lambda$, where λ is the parameter that is to be varied until the other boundary condition $y(3) = 0$ is satisfied. (Note: Maple can solve some nonlinear problems like this by using `dsolve(...type=numeric)`.)

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 weeks to come.

(f) Extra credit: Find a way to use a combination of linear algebra and iteration (initial guess, refinement, etc.) to solve Eq. (**??**) in Matlab on a grid.

3.2 Now let's see how to modify the ideas in Problem 3.1 to handle boundary conditions where derivatives are specified instead of values. Consider the differential equation

$$y''(x) + 9y(x) = x \quad ; \quad y(0) = 0 \quad ; \quad y'(2) = 0$$

If we use the grid we used in Problem 3.1, $x = xmin : h : xmax$, then we would again have at the first point $y_1 = 0$ to satisfy $y(0) = 0$. But what do we do at the last point, y_N?

(a) A crude way to implement the zero-derivative boundary condition is to try to make the function flat at $x = 2$ by setting $y_N = y_{N-1}$. Modify your code for Problem 3.1 to include this boundary condition and compare the resulting numerical solution to the exact solution obtained from Maple:

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

(b) Improve your code in part (a) by doing the following bit of fancy extrapolation. Use Maple to find an approximate formula for the function $y(x)$ over the range $[x_{N-2}, x_N]$ where Determine a, b, c by assuming that the three function values y_{N-2}, y_{N-1}, y_N are known. Take the derivative of this approximate form, then evaluate it at $x = x_N$ and set it to zero to get a new numerical boundary condition to apply at N. Modify your code from part (a) to include this new condition and show that this gives a more accurate solution than the crude technique of part (a).

4 The Wave Equation: Steady State and Resonance

4.1 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 for a string of length L fixed at both ends with a force applied to it that varies sinusoidally in time:

$$\mu\frac{\partial^2 y}{\partial t^2} = T\frac{\partial^2 y}{\partial x^2} + f(x)\cos\omega t \quad ; y(0,t) = 0, \quad y(L,t) = 0 \tag{20}$$

where $y(x,t)$ is the (small) sideways displacement of the string as a function of position and time assuming that $y(x,t) \ll L$. This equation may look a little unfamiliar to you, so let's discuss each term. I have written it in the form of Newton's second law to make it easier to understand, so you should recognize ma on the left, except that μ is not the mass, but 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. Finally, $f(x)$ is the force/length applied to the string as a function of position.

Before we start calculating, let's use our intuition to guess at how the solutions of this equation behave. If we suddenly started to push and pull on a string under tension 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. But suppose that there was a little bit of damping in the system (not included in the equation above, but a little later we will include 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 is the wave equation analog of damped transients and the steady final state of a driven harmonic oscillator.) Let's find the steady-state solution of the driven wave equation above.

We will find it by looking for a solution of the form

$$y(x,t) = g(x)\cos\omega t \tag{21}$$

because 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

$$-\mu\omega^2 g(x) = Tg''(x) + f(x) \quad ; \quad g(0) = 0, \quad g(L) = 0 \tag{22}$$

which is just a two-point boundary value problem of the kind we studied in the first lab.

(a) Let $\mu = 0.003$, $T = 127$, $L = 1.2$, $\omega = 400$, and

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

(All quantities are in SI units.)

Modify one of your Matlab scripts from Lab 2 to solve Eq. (**??**) to find the steady-state amplitude associated with this driving force density.

(b) Repeat the calculation in part (a) for 100 different frequencies between $\omega = 400$ and $\omega = 1200$ by putting a loop that varies ω around your calculation in (a). 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.)

4.2 In part 4.1(b) you should have noticed an apparent resonance behavior, with resonant frequencies near $\omega = 550$ and $\omega = 1100$. In this part we will see how to use Matlab to find these resonant frequencies. 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. (**??**) for which the driving force $f(x) = 0$:

$$\frac{T}{\mu} g''(x) = -\omega^2 g(x) \quad ; \quad g(0) = 0, \quad g(L) = 0 \tag{23}$$

This is a classic eigenvalue problem which is of the form

$$\mathcal{L} g = \lambda g \tag{24}$$

where $\mathcal{L}$ is a linear operator (the second derivative on the left side of Eq. (**??**)) and where λ is the eigenvalue ($-\omega^2$ in Eq. (**??**).) This equation is easily solved analytically because its solutions are just the familiar sine and cosine functions. The condition $y(0) = 0$ tells us to use the sine function, and the differential equation then tells us that the solution is

$$g(x) = A \sin\left(\omega \sqrt{\frac{\mu}{T}} x\right) \tag{25}$$

The resonant frequencies are then determined by the final condition $g(L) = 0$, which yields

$$\omega = \frac{n\pi}{L} \sqrt{\frac{T}{\mu}} \tag{26}$$

But if the differential equation is not so simple we need to do this eigenvalue calculation numerically, so let's see how it works in this simple case. Rewriting Eq. (**??**) in matrix form, as we learned to do by finite differencing the second derivative, yields

$$\begin{bmatrix} 1 & 0 & 0 & 0 & ... & 0 & 0 & 0 \\ \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & ... & 0 & 0 & 0 \\ 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & ... & 0 & 0 & 0 \\ . & . & . & . & ... & . & . & . \\ . & . & . & . & ... & . & . & . \\ . & . & . & . & ... & . & . & . \\ 0 & 0 & 0 & 0 & ... & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} \\ 0 & 0 & 0 & 0 & ... & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ . \\ . \\ . \\ g_{N-1} \\ g_N \end{bmatrix} = \lambda \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ . \\ . \\ . \\ g_{N-1} \\ g_N \end{bmatrix} \tag{27}$$

where $\lambda = -\omega^2 \frac{\mu}{T}$.

If you are paying close attention as you examine this matrix equation you will notice that something has gone wrong with the boundary conditions. The correct equation in the top row of the matrix should be

$$g_1 = 0 \tag{28}$$

but instead I have written

$$g_1 = \lambda g_1 \tag{29}$$

These look quite different, but they are in fact almost exactly the same, because if $\lambda \neq 1$, the only finite solution of $g_1 = \lambda g_1$ is $g_1 = 0$. Of course, if one of the eigenvalues has the value 1, then this method will fail, but in the present case where the eigenvalues are all negative numbers, it will work fine provided that you remember to discard as unphysical any numerical eigenvalues with $\lambda = 1$. (You will see later that Matlab's linear algebra will give you two $\lambda = 1$ solutions to throw away.)

To numerically solve this eigenvalue problem you simply do the following in Matlab.

(i) Load the matrix A with the matrix on the left side of Eq. (**??**).

(ii) Use Matlab's eigenvalue and eigenvector command:

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

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 $y(x)$ on the grid associated with each eigenvalue.)

(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 I will give you a section of a Matlab script below that does steps (ii) and (iii).

```
[V,D]=eig(A);  % get 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))) );
   plot(x,V(:,k(n)),'b-');  % plot the eigenvector that goes with omega^2
   title(t);xlabel('x');ylabel('y(x)');  % label the graph
   pause
end
```

Use these ideas to numerically find the eigenvalues and eigenvectors of Eq. (**??**). In the process you will see that two eigenvalues with $\lambda = 1$ and with associated eigenvectors that do not satisfy the boundary conditions will appear. These show up because of the slightly flaky way we handled the boundary conditions (remember that our method only works if $\lambda \neq 1$.) Just ignore these. You will also see that most of the eigenvectors are very jagged. These must also be ignored because they are poor approximations to the continuous differential equation in Eq. (**??**). But a few of the smooth eigenfunctions are very good approximations and you should find that their corresponding values of ω correspond to the resonances that you found in part (b) and whose values are given by Eq. (**??**).

Go back to your calculation in part (b) and use these resonant values of ω. You should get very large amplitudes, indicating that you are right on the resonances.

5 The Hanging Chain

5.1 If you are behind in your work on the first three labs, get caught up.

5.2 Consider the chain hanging from the ceiling in the classroom. We are going to find its normal modes of vibration using the method of Problem 4.2. Let's use a coordinate system that starts at the bottom of the chain at $x = 0$ and ends on the ceiling at $x = L$.

(a) By using the fact that the stationary chain is in vertical equilibrium, show that the tension in the chain as a function of x is given by

$$T(x) = \mu g x \tag{30}$$

where μ is the linear mass density of the chain and where $g = 9.8$ m/s^2 is the acceleration of gravity. If you were paying really close attention a couple of years ago, you may remember from your sophomore waves class that the wave equation for transverse waves on a chain with varying tension is

$$\mu(x)\frac{\partial^2 y}{\partial t^2} - \frac{\partial}{\partial x}\left(T(x)\frac{\partial y}{\partial x}\right) = 0 \tag{31}$$

As in Problem 2.2, 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 the wave equation and simplify to obtain

$$x\frac{d^2 f}{dx^2} + \frac{df}{dx} = -\frac{\omega^2}{g}f \tag{32}$$

which is in eigenvalue form with $\lambda = -\omega^2/g$.

The boundary condition at the ceiling is $f(L) = 0$ while the boundary condition at the bottom is obtained by taking the limit of Eq. (**??**) as $x \to 0$ to find

$$f'(0) = -\frac{\omega^2}{g}f(0) = \lambda f(0) \tag{33}$$

This condition introduces a new wrinkle that we have to handle before we can proceed to the matrix and find the eigenvalues and eigenvectors: the boundary condition has a derivative in it.

Such problems will be handled this semester by using a special kind of grid called a *cell-centered* grid. The idea is to divide the computing region from 0 to L into N subintervals, and then to put the grid points in the center of each subinterval, or "cell." We then add two more grid points outside of $[0, L]$, one at $x_1 = -h/2$ and the other at $x_{N+2} = L + h/2$. These two points are called *ghost points* and they are used to apply the boundary conditions. Notice that this means that there isn't a grid point at either endpoint, but rather that the two grid points on each end straddle the endpoints. Take a minute now and run the code below to draw a picture of this situation so that you have clearly in mind how this cell-centered grid is set up. The red stars are the grid points and the blue lines show the cells that contain the grid points.

```
clear;close;
N=20;h=1/N;
```

```
xc=-h/2:h:1+h/2;
xe=-h:h:1+h;
plot(xc,0*xc,'r*',xe,0*xe,'b+',xe,0*xe,'b-')
hold on
plot([0,0],[-.06,.06],'k-',[1,1],[-.06,.06],'k-')
axis([-2*h 1+2*h -.1 1])
```

(Notice that by doing it this way there are $N+2$ grid points, which may seem weird to you. I prefer it, however, because it reminds me that I am using a cell-centered grid with N physical grid points and 2 ghost points. You can do it any way you like, as long as your counting method works.)

If the boundary condition specifies a value, like $f(L) = 0$ in the problem at hand, we use a simple average like this:

$$\frac{f_{N+2} + f_{N+1}}{2} = 0 \tag{34}$$

or, in an eigenvalue problem

$$\frac{f_{N+2} + f_{N+1}}{2} = \lambda \frac{f_{N+2} + f_{N+1}}{2} \tag{35}$$

so that only when $\lambda = 1$ does this equation allow anything but zero for the average of the two values. As before, we watch for these special 1-eigenvalues and throw them out as unphysical.

Because the right side of this condition is not simply λf_{N+2} we will have to handle an eigenvalue problem of slightly more general form than just $Af = \lambda f$. The new type is called a *generalized eigenvalue problem* and takes the form

$$Af = \lambda B f \tag{36}$$

where B is a square matrix. Matlab knows how to handle problems like this through the command

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

In our case, since the equation to be satisfied at $N+2$ is Eq. (**??**), the bottom row of A would be

$$A(N+2,:) = [0\ 0\ ...1/2\ 1/2] \tag{37}$$

and the bottom row of B would be identical:

$$B(N+2,:) = [0\ 0\ ...1/2\ 1/2] \tag{38}$$

(Make sure you understand how this produces Eq. (**??**) before moving on.)

Now let's do the same thing to the condition at the bottom, Eq. (**??**). The derivative in this condition can be approximated with an accurate centered difference because points 1 and 2 straddle $x = 0$ (this is the real reason for using the cell-centered grid.) So Eq. (**??**) translated onto the grid becomes

$$\frac{f_2 - f_1}{h} = -\omega^2 \frac{f_2 + f_1}{2g} = \lambda \frac{f_2 + f_1}{2} \tag{39}$$

Translating this equation into linear algebra makes the top rows of the matrices A and B look like this:

$$A(1,:) = [-\frac{1}{h}, \frac{1}{h} ...0\ 0] \tag{40}$$

$$B(1,:) = [\frac{1}{2}, \frac{1}{2} ...0\ 0] \tag{41}$$

Now that the top and bottom rows are built all that remains is to load the rest of the rows by finite differencing Eq. (**??**). (Notice that for the interior points the matrix B is just the identity matrix with 1's on the main diagonal and zeros everywhere else.)

(b) Load the matrices A and B and use Matlab to solve for the normal modes of vibration of a hanging chain. We will then compare your answers to measurements we will make on the chain hanging from the ceiling in the classroom.

(c) Solve Eq. (**??**) analytically using Maple. 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 (a).

Note: this problem of standing waves on a hanging chain was solved in the 1700's by Johann Bernoulli and is the first time that the function that later became known as the J_0 Bessel function showed up in physics.

5.3 (a) Consider the problem of a particle in a 1-dimensional harmonic oscillator well in quantum mechanics. Schrödinger's equation for the bound states in this well is

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + \frac{1}{2}kx^2\psi = E\psi \tag{42}$$

with boundary conditions $\psi = 0$ at $\pm\infty$. I will help you out by telling you that the characteristic length in this problem is

$$a = \left(\frac{km}{\hbar^2}\right)^{1/4} \tag{43}$$

and that the bound state energies are given by

$$E_n = (n + \frac{1}{2})\hbar\sqrt{\frac{k}{m}} \tag{44}$$

Use Matlab's ability to do eigenvalue problems to verify that this formula for the bound state energies is correct for $n = 0, 1, 2, 3, 4$. (You can simplify matters by choosing $\hbar = k = m = 1$ if you like.)

(b) Now redo (a) but with the harmonic oscillator potential replaced by

$$V(x) = \mu x^4 \tag{45}$$

Let all of the constants be unity again and find the first 5 bound state energies.

6 Animating the Wave Equation

6.1 Lab 3 should have seemed pretty familiar, since it handled the wave equation by Fourier analysis to turn the partial differential equation into a set of ordinary differential equations, as you learned in Mathematical Physics. But separating the variables and expanding in orthogonal functions is not the only way to solve partial differential equations, and in fact in many situations this technique is awkward, ineffective, or both. In this lab we will study another way of solving partial differential equations using a spatial grid and stepping forward in time, as we did in Physics 330. And as an added attraction, this method automatically supplies a beautiful animation of the solution. I will only show you one of the many algorithms of this type that can be used on wave equations, so this is just an introduction to a very big subject. The method I will show you here is called *staggered leapfrog*, and although there are better methods, they are more complicated. This is the simplest good one that I know.

So, consider again the classical wave equation with wave speed c. (For instance, for waves on a string $c = \sqrt{T/\mu}$.)

$$\frac{\partial^2 y}{\partial t^2} - c^2 \frac{\partial^2 y}{\partial x^2} = 0 \tag{46}$$

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

$$y(0,t) = f_{left}(t) \quad ; \quad y(L,t) = f_{right}(t) \tag{47}$$

or of *Neumann* type (derivatives specified):

$$\frac{\partial y}{\partial x}(0) = g_{left}(t) \quad ; \quad \frac{\partial y}{\partial x}(L) = g_{right}(t) \tag{48}$$

or, perhaps, a mix of value and derivative boundary conditions (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 ($f_{left}(t) = f_{right}(t) = 0$); perhaps the ends slide up and down on frictionless rings attached to frictionless rods ($g_{left}(t) = g_{right}(t) = 0$); or perhaps the left end is fixed and someone is wiggling the right end up and down sinusoidally ($f_{left}(t) = 0$ and $f_{right}(t) = A \sin \omega t$). These conditions at the ends are required to be able to solve the wave equation.

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

$$y(x,0) = y_0(x) \quad ; \quad \frac{\partial y}{\partial t}(0) = v_0(x) \tag{49}$$

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 get 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 values τ and h.

$$\frac{\partial^2 y}{\partial t^2} \approx \frac{y_j^{n+1} - 2y_j^n + y_j^{n-1}}{\tau^2} \tag{50}$$

$$\frac{\partial^2 y}{\partial x^2} \approx \frac{y_{j+1}^n - 2y_j^n + y_{j-1}^n}{h^2} \tag{51}$$

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 get the algorithm just put these two approximations into the classical wave equation and solve for y_j^{n+1}:

$$y_j^{n+1} = 2y_j^n - y_j^{n-1} + \frac{c^2\tau^2}{h^2}\left(y_{j+1}^n - 2y_j^n + y_{j-1}^n\right) \tag{52}$$

(a) Use Maple to derive this algorithm from the approximate second derivative formulas.

This algorithm can only be used at interior spatial grid points, however, because the $j+1$ and $j-1$ indices would 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. In addition, the algorithm requires not just y at the current time level n but also y at the previous time level $n-1$. This means that we have to have an array `y` for the current values and another array `yold` for the previous values. But at time $t = 0$ when we have to start the calculation the previous time array `yold` is not available. To get its starting values we will have to make use of the initial condition on the velocity.

So before we can actually use this algorithm we have to handle the boundary conditions and the initial conditions. But you will have to be patient; handling these special points is a bit more involved than the simple algorithm for the interior points.

Let's worry about the boundary conditions at the ends first because they are relatively easy to implement. And since we will want to use both fixed value (Dirichlet) and derivative (Neumann) boundary conditions, let's use a cell-centered grid so we can easily handle both types without changing our grid. Whichever type is specified, we first advance the solution at all interior points using Eq. (**??**), then we use the boundary conditions to get the new values of y_1^{n+1} and y_{N+2}^{n+1}.

If the values at the ends are specified (Dirichlet boundary conditions) we would translate the boundary conditions like this:

$$\frac{y_1^{n+1} + y_2^{n+1}}{2} = f_{left}(t_{n+1}) \quad \Rightarrow \quad y_1^{n+1} = -y_2^{n+1} + 2f_{left}(t_{n+1}) \tag{53}$$

$$\frac{y_{N+2}^{n+1} + y_{N+1}^{n+1}}{2} = f_{right}(t_{n+1}) \quad \Rightarrow \quad y_{N+2}^{n+1} = -y_{N+1}^{n+1} + 2f_{right}(t_{n+1}) \tag{54}$$

If the derivatives are specified (Neumann boundary conditions) then we would do this:

$$\frac{y_2^{n+1} - y_1^{n+1}}{h} = g_{left}(t_{n+1}) \quad \Rightarrow \quad y_1^{n+1} = y_2^{n+1} - hg_{left}(t_{n+1}) \tag{55}$$

$$\frac{y_{N+2}^{n+1} - y_{N+1}^{n+1}}{h} = g_{right}(t_{n+1}) \quad \Rightarrow \quad y_{N+2}^{n+1} = y_{N+1}^{n+1} + hg_{right}(t_{n+1}) \tag{56}$$

Just use the appropriate equation from Eqs. (??)-(??) after staggered leapfrog has been used to advance the interior points, and we are ready to take the next time step.

Finally, let's see how to use the initial conditions to find the starting value of the old solution (y_j^{n-1} in Eq. (??).) To make things clear, we will 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}. Our strategy will be to use both the initial velocity condition and the staggered leapfrog algorithm to find the two sets of unknowns y_j^1 and y_j^{-1}. A centered time derivative at $t = 0$ turns the initial velocity condition into one relation between these quantities:

$$\frac{y_j^1 - y_j^{-1}}{2\tau} = v_0(x_j) \tag{57}$$

Then we use staggered leapfrog to obtain another relation between y_j^1 and y_j^{-1}. Leapfrog at the first step says that

$$y_j^1 = 2y_j^0 - y_j^{-1} + \frac{c^2\tau^2}{h^2}\left(y_{j+1}^0 - 2y_j^0 + y_{j-1}^0\right) \tag{58}$$

These two equations can be solved simultaneously at each j to get both y_j^{-1} and y_j^1, but we only really care about y_j^{-1} since the regular Leapfrog algorithm will get us to y_j^1 once we have y_j^{-1} and y_j^0. Doing the solve yields

$$y_j^{-1} = y_j^0 - v_0(x_j)\tau + \frac{c^2\tau^2}{2h^2}\left(y_{j+1}^0 - 2y_j^0 + y_{j-1}^0\right) \tag{59}$$

(b) Use Maple to derive this formula.

OK; we are now ready to code. I 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 vertical column of dots indicates where you are supposed to write your own code. Use fixed-end boundary conditions.

When you are finished you should be able to run, debug, then successfully run an animation of what happens when a guitar string initially at rest starts with an initial upward displacement localized near the center of the string. Once you get it running do the numerical studies listed after the template.

```
% Staggered Leapfrog Script Template

clear;close;

% build a cell-centered grid with N=200 and L=1;
.
.
.

% Define inline functions that specify the initial displacement
% and the initial velocity
yinit=inline('exp(-(x-L/2).^2*160/L^2)-exp(-(0-L/2).^2*160/L^2)','x','L');
```

```
vinit=inline('0*x','x','L') % zero velocity initial condition
% load y(x,0)
y=yinit(x,L);
% load vy(x,0)
vy=vinit(x,L);

subplot(2,1,1)
plot(x,y) % plot the initial conditions
xlabel('x');ylabel('y(x,0)');title('Initial Displacment')
subplot(2,1,2)
plot(x,vy) % plot the initial conditions
xlabel('x');ylabel('v_y(x,0)');title('Initial Velocity')

pause;

% Set the wave speed;
c=2; % wave speed

% Suggest to the user that a time step no larger than taulim be used
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
.
.
.


% load the ghost point boundary conditions in yold(1) and yold(N+2)
.
.
.

tfinal=input(' Enter tfinal - ')
skip=input(' Enter # of steps to skip between plots (code runs 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
   % 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 Equation');
      axis([min(x) max(x) -2 2]);
      pause(.1)
   end
end
```

(c) Run with the initial conditions in the script above and with fixed end points and experiment with various time steps τ. Show by numerical experimentation that if $\tau > c/h$ 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-Levy 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.

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

(e) Change the initial conditions from initial displacement with zero velocity to initial velocity with zero displacement. Watch how the wave motion develops in this case. Use fixed end boundary conditions. Then find a slinky, stretch it out, and whack it in the middle to verify that the math gets the physics right.

(f) Modify your leapfrog code to use $T = 1$ and $\mu = 0.1 + x/L$, so that the linear mass density increases toward the right. Use fixed ends. (This makes the wave speed be a function of position, $c = c(x)$.) Watch how the pulses propagate and explain qualitatively why they behave as they do.

(g) Modify the leapfrog algorithm to include damping of the waves using a linear damping term, like this.

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

with c constant. Use fixed ends. Use Maple to derive modified staggered leapfrog algorithm for this new wave equation, including a new formula for the initial value of `yold`. Then let $\gamma = .2$ and run one of your animations in part (e) and verify that the waves damp away. You will need to run for about 25 s and you will want to use a big skip factor so that you don't have to wait forever for the run to finish. Include some code to record the maximum value of $y(x)$ as a function of time and then plot it as a function of time after the time loop 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?

(h) Apply a driving force to the damped wave equation using the form we studied in Problem 4.1(a). (You will need to figure out how to include a driving force in the staggered leapfrog algorithm.) Use the parameter values given in that problem and choose a damping constant γ that is the proper size to make the system settle down after a 20 or 30 bounces of the string. Start the string at rest and use fixed ends. Run long enough that you can see the transients die away and the string settle into the steady oscillation at the driving frequency. Then run again with $\omega = 1080$, which is close to a resonance, and again see the system come into steady oscillation at the driving frequency.

Challenge Problem. Use the staggered leapfrog algorithm to study how pulses propagate along a hanging chain. You will have to work at getting the boundary condition at the bottom of the chain to work right. Derive this boundary condition from the wave equation on the chain by looking at the wave equation at $x = 0$.

7 Staggered Leapfrog in Two Dimensions

7.1 The wave equation for transverse waves on a rubber sheet is

$$\mu \frac{\partial^2 z}{\partial t^2} = \sigma \left(\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} \right) \tag{61}$$

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? If it doesn't, it should, and the necessary force is given by σL.

Write a Matlab script that animates the solution of this equation on a square region that is [-5,5]×[-5,5] and that has fixed edges. (Don't use ghost points; just set the edge values to zero and leave them that way.) Use a displacement initial condition that is an initial Gaussian pulse with zero velocity:

$$z(x, y, 0) = e^{-5(x^2+y^2)} \tag{62}$$

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

Also watch what happens at the center of the sheet by making a plot of $z(t)$ 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 $z(t)$ before the first reflection comes back.

Be sure to use double-colon commands so that your code will run fast, i.e.,

```
z(2:N+1,2:N+1)= ...  z(1:N,2:N+1)+z(3:N+2,2:N+1) ...
```

8 The Diffusion Equation

Now we're going to attack the diffusion equation

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2} \quad . \tag{63}$$

This equation 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 its initial conditions are quite different. We only need to know the initial distribution of T, so the complication with $\partial T/\partial t$ in the wave equation is avoided. This is a simplification but, as we shall see, the diffusion equation is actually more difficult to solve numerically than the wave equation.

8.1 (a) As a warm up, show that an initial Gaussian temperature distribution like this

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

decays according to the formula

$$T(x,t) = \frac{T_0}{\sqrt{1+4Dt/\sigma^2}} e^{-(x-L/2)^2/(\sigma^2+4Dt)} \tag{64}$$

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

(b) Use separation of variables to find how an initial temperature distribution $T(x,0) = T_0 \sin(n\pi x/L)$ decays with time, n an integer. Do this by substituting the form $T(x,t) = T(x,0)f(t)$ into the diffusion equation and finding $f(t)$ for each integer n.

(c) Show by using dimensional arguments that the approximate time it takes for a distribution 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.) Then show that the solution in 8.1(a) agrees with your analysis and find the numerical factor in the equation for the time that it takes for the width of the temperature distribution to double from σ to 2σ.

(d) Let's now try to solve the diffusion equation numerically on a grid as we did with the wave equation. If we finite difference the diffusion equation using a centered time derivative and a centered second derivative in x to get an algorithm that is similar to leapfrog then we have

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

$$T_j^{n+1} = T_j^{n-1} + \frac{2D\tau}{h^2}\left(T_{j+1}^n - 2T_j^n + T_{j-1}^n\right) \tag{65}$$

There is a problem getting this algorithm started 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. In other words, the CFL condition for the nice centered algorithm above is $\tau \le 0$, which makes it difficult to step forward boldly into the future.

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 of the algorithm leads to numerical instability (infinite zig-zags). We will see a little later how to get both accuracy and stability for the diffusion equation, but for now let's sacrifice a little bit of accuracy to get a stable algorithm. If we use an inaccurate forward difference for the time derivative we find

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

$$T_j^{n+1} = T_j^n + \frac{D\tau}{h^2}\left(T_{j+1}^n - 2T_j^n + T_{j-1}^n\right) \tag{66}$$

Modify your staggered leapfrog program to use this algorithm to solve the diffusion equation on the interval $[0, L]$ with initial distribution

$$T(0, x) = \sin\left(\pi x/L\right) \tag{67}$$

and boundary conditions $T(0) = T(L) = 0$. Use $D = 2$, $L = 3$, and $N = 20,\ 40,\ 80$.

This algorithm has a CFL condition on the time step τ of the form

$$\tau \le C\frac{h^2}{D} \tag{68}$$

Determine the value of C by numerical experimentation.

To test the accuracy of your numerical solution compare to your answer to part (b). Show by using overlaid graphs that your grid solution matches the exact solution with increasing accuracy as the number of grid points N is increased.

(e) Do as in (d), but use as an initial condition a Gaussian distribution centered at $x = L/2$:

$$T(x, 0) = e^{-40(x/L-1/2)^2} \tag{69}$$

Use two different kinds of boundary conditions: (1) $T = 0$ at both ends and (2) $\partial T/\partial x = 0$ 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.

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

$$D(x) = D_0\frac{x^2 + L/5}{(L/5)} \tag{70}$$

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

$$\frac{\partial T}{\partial t} = \frac{\partial}{\partial x}\left(D(x)\frac{\partial T}{\partial x}\right) \tag{71}$$

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

9 Implicit Methods: the Crank-Nicholson Algorithm

You may have noticed that all of the algorithms we have discussed so far are of the same type: at each spatial grid point j you use 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, are said to be *explicit* and are easy to code. They are also often numerically unstable. *Implicit* methods, by contrast, are harder to implement, but have much better stability properties. And the reason they are harder is easy to understand: they assume that you already know the future. And as a wise person once said, "Prediction is difficult, especially of the future." It will become clear in a minute what I mean by knowing the future, but let's first discuss what's wrong with applying the explicit methods we have been using so far to the diffusion equation.

As you have learned from the assigned problems, the time step constraint for explicit methods applied to the diffusion equation are of the form $\tau < Bh^2$, where B is a constant. 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 code take forever to run, which is usually intolerable.

To see how to fix this problem, let's look at the diffusion equation again, and just to make things more interesting we'll let the diffusion coefficient be a function of x:

$$\frac{\partial T}{\partial t} = \frac{\partial}{\partial x}\left(D(x)\frac{\partial T}{\partial x}\right) \tag{72}$$

We begin by finite differencing the right side, taking care to handle the spatial dependence of D. In the equation below $D_{j\pm 1/2} = D(x_j \pm h/2)$.

$$\frac{\partial T_j}{\partial t} = \frac{1}{h^2}\Big(D_{j+1/2}(T_{j+1} - T_j) - D_{j-1/2}(T_j - T_{j-1})\Big) \tag{73}$$

And now we take care of the time derivative by doing something that looks a bit odd: we will take a forward time derivative on the left, putting this side of the equation at time level $n + 1/2$. And then to get the right side at the same time level so that the algorithm will be second-order accurate, we will replace each occurrence of T on the right-hand side by the average

$$T^{n+1/2} = \frac{T^{n+1} + T^n}{2} \tag{74}$$

like this:

$$\frac{T_j^{n+1} - T_j^n}{\tau} = \frac{1}{2h^2}\Big(D_{j+1/2}(T_{j+1}^{n+1} - T_j^{n+1} + T_{j+1}^n - T_j^n) - D_{j-1/2}(T_j^{n+1} - T_{j-1}^{n+1} + T_j^n - T_{j-1}^n)\Big) \tag{75}$$

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 of T_j^{n+1} are all over the place, and these future values 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}. Well, this is what I meant when I said that these methods are harder because you have to know the future.

In the hope that something useful will show 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.

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

$$\frac{D_{j-1/2}\tau}{2h^2}T_{j-1}^{n}+\left(1-\frac{(D_{j+1/2}+D_{j-1/2})\tau}{2h^2}\right)T_j^{n}+\frac{D_{j+1/2}\tau}{2h^2}T_{j+1}^{n} \tag{76}$$

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

To see more clearly what we are doing, and to make the algorithm a bit more efficient, let's define the tridiagonal matrix **Q** like this:

$$Q_{jk}=0 \quad \text{except for :}$$

$$Q_{j,j-1}=\frac{D_{j-1/2}\tau}{2h^2} \quad ; \quad Q_{j,j}=-\frac{(D_{j+1/2}+D_{j-1/2})\tau}{2h^2} \quad ; \quad Q_{j,j+1}=\frac{D_{j+1/2}\tau}{2h^2} \tag{77}$$

Then Eq. (**??**) can be written in operator form (T is now a column vector and **I** is the identity matrix) like this:

$$(\mathbf{I}-\mathbf{Q})T^{n+1}=(\mathbf{I}+\mathbf{Q})T^n \tag{78}$$

so we may write

$$T^{n+1}=(\mathbf{I}-\mathbf{Q})^{-1}(\mathbf{I}+\mathbf{Q})T^n \tag{79}$$

Of course, we have no intention of inverting a matrix because that's expensive; we will use Gauss elimination instead.

And to make the algorithm even more efficient, we will also take advantage of the simple tridiagonal form of **Q**. The Matlab script below called `tridag.m` does Gauss elimination on a tridiagonal matrix using only the three diagonals. The input matrix `A` has three columns and as many rows as there are grid points. The matrix `r` is the single-columns right-hand side vector in the linear system

$$Ax=r \tag{80}$$

The routine first puts the tridiagonal matrix in upper triangular form, then solves for x_N because now the bottom row of the matrix is simple and works its way up to x_1. 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,N)`.) Then do the solve with the command

```
    x=tridag(A,r);
```

Here is the script:

```
%beginfunction 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;
```

Solving for T^{n+1} using this script would proceed as follows: (1) build the right-hand side $(\mathbf{I}+\mathbf{Q})T^n$ by running over the grid points and loading a column vector r with the right-hand side of Eq. (**??**).

$$r = (\mathbf{I} + \mathbf{Q})T^n \tag{81}$$

(2) Then pass the three non-zero diagonals of the matrix $\mathbf{Q}$ and the right-hand side column vector r into `tridag` to find T^{n+1}.

$$T^{n+1} = \texttt{tridag}(\mathbf{I} - \mathbf{Q}, r) \tag{82}$$

We can eliminate the somewhat expensive step of evaluating the right-hand side $(\mathbf{I}+\mathbf{Q})T^n$ as follows. In Eq. (**??**) replace $\mathbf{I}+\mathbf{Q}$ in the second term on the right by $(-\mathbf{I}+\mathbf{Q})+2\mathbf{I}$, then perform one of the matrix multiplications to get

$$T^{n+1} = -T^n + 2(\mathbf{I} - \mathbf{Q})^{-1}T^n \tag{83}$$

which would be coded this way:

```
r=T;
Tnew=-T+2*tridag(ImQ,r);
```

where `ImQ` is the $N \times 3$ $\mathbf{I}-\mathbf{Q}$ matrix in banded form. This is better because the right-hand side vector r in the tridiagonal solve is just T_j^n, which we already have, so we save some computational work.

So far we have left out an important part of this problem, which is the boundary conditions. To see how to implement them we need to undo the inverse-multiply in Eq. (**??**) like this:

$$(\mathbf{I} - \mathbf{Q})T^{n+1} = -(\mathbf{I} - \mathbf{Q})T^n + 2T^n \tag{84}$$

Suppose we are using ghost points and we want $T = 0$ at the left edge, which requires

$$\frac{T_1 + T_2}{2} = 0 \tag{85}$$

To impose this condition we would load the $j = 1$ row of $\mathbf{I}-\mathbf{Q}$ with [1/2,1/2,0,0,...,0], and hope that there would be a way to have the right-hand side of Eq. (**??**) be equal to zero. But this is actually not too difficult because if T^n satisfies the boundary conditions then the first term on the right-hand side of Eq. (**??**) vanishes, and we just need to set `r(1)=0` after the line `r=T` in the section of code above. We would do something similar at the right edge, and the modification of this procedure for derivative boundary conditions should be familiar by now.

Here is a Matlab program that implements the Crank-Nicholson algorithm. Its use requires that you have the tridiagonal solver `tridag.m` in the same directory.

```
%beginfunction cranknicholson.m

% this code solves the diffusion equation using tridag.m
% to implement Crank-Nicholson

clear;close;

% Set the number of grid points and build the cell-center
% grid.

N=input(' Enter N, cell number - ')
L=10;
h=L/N;

x=-.5*h:h:L+.5*h;

%  load the diffusion coefficient array

D=ones(1,N+2);

% initialize the temperature with a sine function
T=sin(pi*x/L);

% turn T into a column vector.  Note that we use .' (transpose
% without complex conjugate) instead of ' so that if we
% are doing quantum mechanics we don't reverse the direction
% a wave packet moves.

T=T.';

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

% Suggest a timestep by giving the time step for
% explicit stability, then ask for the time step

fprintf(' Maximum explicit time step: %g \n',h^2/max(D))
tau = input(' Enter the time step - ')

tfinal=input(' Enter the total run time - ')

% set the number of time steps to take
nsteps=tfinal/tau;

% define a useful constant
coeff=tau/h^2 ;

% load the tridiagonal matrix with the Crank-Nicholson
% operator (I-Q): imq
```

```
imq=zeros(N+2,3);

% set T=0 at both the left and right edges
imq(1,2)=1/2;imq(1,3)=1/2;imq(N+2,2)=1/2;imq(N+2,1)=1/2;

for j=2:N+1
   Dm=.5*(D(j)+D(j-1)); % D(j-1/2)
   Dp=.5*(D(j)+D(j+1)); % D(j+1/2)
   imq(j,1) = -Dm*.5*coeff;
   imq(j,2) = (1+(Dm+Dp)*.5*coeff);
   imq(j,3) = -Dp*.5*coeff;
end

% this is the time advance loop

for m=1:nsteps

% load the right-hand side for the tridiagonal solve; current T
% plus boundary condition modifications at j=1 and j=N+2
   r=T;
   r(1)=0;r(N+2)=0;

% Crank-Nicholson advance
   s=tridag(imq,r);
   T=-T+2*s;

% set the boundary conditions by hand to make sure they work right
   T(1)=-T(2);T(N+2)=-T(N+1);

   t=m*tau; % set the time

% make a plot of the radial T(r) profile every once in a while
   if(rem(m,5) == 0)
   plot(x,T)
   axis([0 L Tmin Tmax])
   pause(.1)
   end

end

%end cranknicholson.m
```

9.1 To give you a better feel for what Crank-Nicholson is doing, and what "implicit" means, let's study the simple first-order differential equation

$$\dot{y} = -\gamma y \qquad (86)$$

(a) Euler's method looks like this:

$$\frac{y_{n+1} - y_n}{\tau} = -\gamma y_n \qquad \Rightarrow \qquad y_{n+1} = y_n(1 - \gamma\tau) \tag{87}$$

Show by numerical experimentation that this algorithm is unstable (meaning that y blows up, instead of decaying in time as it should) for $\tau > 2/\gamma$. This is an explicit method.

(b) Modify Euler's method by replacing the right-hand side $-\gamma y_n$ by an average of the advanced and current values of y, as in Crank-Nicholson, and show by numerical experimentation that when τ gets large this method doesn't blow up. It isn't correct because y_n bounces between positive and negative values, but at least it doesn't blow up. The presence of τ in the denominator is the tip-off that this is an implicit method, and the improved stability is the point of using something implicit.

(c) Now Modify Euler's method by making it *fully implicit* by using y_{n+1} in place of y_n on the right. This method is no more accurate than Euler's method for small time steps, but it is much more stable. Show by numerical experimentation that this fully implicit method damps away very quickly when τ is large. Extra damping is usually a feature of fully implicit algorithms.

9.2 (a) Test the code above by running it for various values of τ and seeing how it compares with the exact solution from Lab 8. Verify that when the time step is too large the solution is inaccurate, but stable. Use a constant $D = 2$.

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 time units and find the maximum difference between the exact and numerical solutions. You should find that time step τ hardly matters at all. The number of cells N is the main thing to worry about if you want high accuracy.

(b) Modify the Crank-Nicholson code 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 why your results are physically reasonable. In particular, explain why even though D is completely different, the final value of T is the same as in part (b).

10 Implicit Methods in 2-Dimensions: Operator Splitting

10.1 Consider the diffusion equation in two dimensions, simplified so that the diffusion coefficient is a constant:

$$\frac{\partial T}{\partial t} = D\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) \tag{88}$$

If we define the finite difference operators $\mathcal{L}_x$ and $\mathcal{L}_y$ as follows,

$$\mathcal{L}_x T_{i,j} = \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{h^2} \quad ; \quad \mathcal{L}_y T_{i,j} = \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{h^2} \tag{89}$$

then it is easy to write down the Crank-Nicholson algorithm in 2-dimensions (I have suppressed the spatial subscripts to avoid clutter):

$$\frac{T^{n+1} - T^n}{\tau} = \frac{D}{2}\left(\mathcal{L}_x T^{n+1} + \mathcal{L}_x T^n + \mathcal{L}_y T^{n+1} + \mathcal{L}_y T^n\right) \tag{90}$$

If we could solve simply for $T^{n+1} = T^{n+1}_{i,j}$, as we did in the previous lab, 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-Nicholson in 2-dimensions.

This clever way is called *operator splitting*, and was invented by Douglas [Douglas, J. Jr. , *SIAM J.* , **9**, 42, (1955)] and by Peaceman and Rachford [Peaceman, D. W. and Rachford, H. H. , *J. Soc. Ind. Appl. Math.* , **3**, 28, (1955)]. 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:

$$\frac{T^{n+1/2} - T^n}{\tau/2} = D\left(\mathcal{L}_x T^{n+1/2} + \mathcal{L}_y T^n\right)$$

$$\frac{T^{n+1} - T^{n+1/2}}{\tau/2} = D\left(\mathcal{L}_x T^{n+1/2} + \mathcal{L}_y T^{n+1}\right) \tag{91}$$

If you stare at this for a while you will be forced to conclude that it doesn't look like Crank-Nicholson at all.

So, here's Problem 10.1: use Maple to eliminate the intermediate variable $T^{n+1/2}$ and show that the algorithm gives

$$\frac{T^{n+1} - T^n}{\tau} = \frac{D}{2}\left(\mathcal{L}_x T^{n+1} + \mathcal{L}_x T^n + \mathcal{L}_y T^{n+1} + \mathcal{L}_y T^n\right) - \frac{D^2\tau^2}{4}\mathcal{L}_x\mathcal{L}_y\left(\frac{T^{n+1} - T^n}{\tau}\right) \tag{92}$$

which is just like 2-dimensional Crank-Nicholson (Eq. (**??**)) except that an extra term corresponding to

$$\frac{D^2\tau^2}{4}\frac{\partial^5 T}{\partial x^2 \partial y^2 \partial t} \tag{93}$$

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-Nicholson 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 $D^2\tau^2/\ell^4 << 1$, where ℓ is the characteristic distance over which the temperature varies (so that we can estimate $\mathcal{L}_x\mathcal{L}_y \approx 1/\ell^4$.) 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.

10.2 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

$$T(x,y,0) = \cos\left(\frac{\pi x}{10}\right)\cos\left(\frac{\pi y}{10}\right) \tag{94}$$

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 get a simple differential equation for $f(t)$.) Then modify your Crank-Nicholson code from Lab 8 to use the operator splitting algorithm described in 10.1. 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.

10.3 Modify your code from Problem 10.2 so that the normal derivative at each boundary vanishes. Also change the initial conditions to something more interesting than those in 10.2; it's your choice.

10.4 Modify your code from Problem 10.2 so that it keeps $T = 0$ in the 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.

11 Schrödinger's Equation

Here is the time-dependent Schrödinger equation which governs the way a quantum wave function changes with time in a potential well (assuming that it is a function of a single spatial variable x):

$$i\hbar\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\psi}{\partial x^2} + V(x)\psi \tag{95}$$

Note that except for the presence of the imaginary unit i, this is very much like the diffusion equation. And, in fact, a good way to solve it is with the Crank-Nicholson 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 after a while you have 3/4 of a particle, then 1/2, etc.. This is clearly unacceptable.

11.1 Write a code to solve the time-dependent Schrödinger equation using Crank-Nicholson as discussed in Lab 8. Use natural units in which $\hbar = m = 1$. I suggest that you rewrite the Schrödinger equation in the form of a diffusion equation with an imaginary diffusion coefficient so that you don't have to modify `cranknicholson` too much. (Be sure to add the effect of $V(x)$ correctly.) Then use your code to do the following problems.

(a) Study the evolution of a particle in a box with $V(x) = 0$ from $x = -L$ to $x = L$ with $L = 10$. The infinite potential at the box edges is imposed with boundary conditions:

$$\psi(-L) = 0 \quad ; \quad \psi(L) = 0$$

Use a cell-edge grid:

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

so we have N grid points.

Start with a localized wave packet of width σ and momentum p:

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

with $p = 6.28319$ and $\sigma = 2$. (To make sure that the boundary conditions are satisfied, subtract $\psi(L,0)$ from $\psi(x,0)$ after you load it.) Run the code with $N = 200$ and watch the particle (wave packet) bounce back and forth in the well. Plot the real part of ψ to visualize the wave packet.

(b) Verify by doing a numerical integral that $\psi(x,0)$ in the formula given above is properly normalized. Then run the code and check that it stays properly normalized, even though the wave function is bouncing and spreading within the well. (Since you are on a cell-centered grid it is natural to use the midpoint method to do the integrals. Be careful to only include the interior points in your integration.)

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

$$\langle x \rangle = \int_{-L}^{L} \psi^*(x,t)\ x\ \psi(x,t) dx \tag{96}$$

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 get 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 code and explain what you see.

(e) You probably noticed that the expectation value of x damps away to zero, suggesting that the particle is sitting still and no longer has any energy. This is false. Calculate the expectation value of the energy

$$\langle E \rangle = \int_{-L}^{L} \psi^*(x,t)\ [-\frac{\hbar^2}{2m}\frac{\partial^2 \psi}{\partial x^2} + V(x)\psi(x,t)] dx \tag{97}$$

and plot it vs. time to verify that energy is conserved. This calculation will work better if you do an integration by parts to get rid of the second derivative in the integral.

11.2 Modify your code from Problem 11.1 so that the computing region goes from $-2L$ to L with a square potential hill $V(x) = V_0$ between $x = -.4L$ and $x = -.5L$ ($V = 0$ everywhere else.) Run your code again

12 Poisson's Equation I

12.1 Consider Poisson's equation on a two-dimensional rectangle $[0,a] \times [0,b]$ with x corresponding to a and y corresponding to b.

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = -\frac{\rho}{\epsilon_0} \tag{98}$$

(Note that by studying this equation we are also studying Laplace's equation, which is just Poisson's equation with $\rho = 0$.) The first step is to define a spatial grid that subdivides the x interval into N_x subintervals and the y interval into N_y subintervals, like this:

```
dx=a/Nx;dy=b/Ny;
x=0:dx:a;
y=0:dy:b;
```

The second step is to write down the finite-difference approximation to the second derivatives in Poisson's equation to get a grid-based version of Poisson's equation. In the new version of the equation shown below I have used the notation

$$V(x,y) = V_{i,j} \quad ; \quad V(x+\Delta x, y) = V_{i+1,j} \quad ; \quad V(x-\Delta x, y) = V_{i-1,j}$$
$$V(x, y+\Delta y) = V_{i,j+1} \quad ; \quad V(x, y-\Delta y) = V_{i,j-1} \tag{99}$$

OK, here's the new Poisson equation:

$$\frac{V_{i+1,j} - 2V_{i,j} + V_{i-1,j}}{\Delta x^2} + \frac{V_{i,j+1} - 2V_{i,j} + V_{i,j-1}}{\Delta y^2} = -\frac{\rho}{\epsilon_0} \tag{100}$$

Notice two things about this equation. (1) It doesn't work for points on the boundary of the region because on the edges it reaches beyond the rectangle; this equation can only be used at interior grid points. This is OK, however, because the boundary conditions tell us what V is on the edges of the region. (2) This is a set of linear equations for the unknown V_{ij}'s, so we could imagine just doing a big linear solve. 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 unknown V's is $(N_x - 1)(N_y - 1)$, which for a typical 100×100 grid is about 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. The day is not far off when this section of the course will explain that doing this big solve is the right way to do 2-dimensional problems like this because computers with much more memory than this will be common. But by then the numerical state of the art will be 3-dimensional solves, which 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 when gigabyte computers are common, people will still be using iteration methods like the ones I am about to describe.

To get a version of Poisson's equation that helps us develop a less memory-intensive way to solve it, we solve Eq. (**??**) for $V_{i,j}$, like this:

$$V_{i,j} = \left(\frac{V_{i+1,j} + V_{i-1,j}}{\Delta x^2} + \frac{V_{i,j+1} + V_{i,j-1}}{\Delta y^2} + \frac{\rho}{\epsilon_0} \right) / \left(\frac{2}{\Delta x^2} + \frac{2}{\Delta y^2} \right) \tag{101}$$

We will now simply iterate on this equation by doing the following. (1) Choose an initial guess for the interior values of $V_{i,j}$. (2) Use this initial guess to evaluate the right-hand side of Eq. (**??**) and then to replace $V_{i,j}$ by this right-hand side. If all goes well, then after many iterations the left and right sides of this equation will agree and we will have a solution.

(a) Derive Eq. (**??**) from the the previous equation (the finite-difference version of Poisson's equation.)

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:

$$x = e^{-x} \tag{102}$$

If we iterate on this equation like this:

$$x_{n+1} = e^{-x_n} \tag{103}$$

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: $x_n = \bar{x} + \delta_n$. Substituting this equation into Eq. (**??**) and expanding to first order in δ then gives

$$\bar{x} + \delta_{n+1} = e^{-\bar{x}} - e^{-\bar{x}}\delta_n \qquad \Rightarrow \qquad \delta_{n+1} = -e^{-\bar{x}}\delta_n \tag{104}$$

So when we are close to the solution the error gets smaller every iteration by the factor $-e^{-\bar{x}}$. Since $\bar{x}$ is positive, $e^{-\bar{x}}$ is less than 1, and we get convergence. 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.

(b) 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 $\bar{x}$-δ 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. (**??**) can be written in matrix form as

$$V_{n+1} = \mathbf{L}V_n + r \tag{105}$$

where $\mathbf{L}$ is the matrix which, when multiplied into the vector V_n, produces the $V_{i,j}$ part of the right-hand side of Eq. (**??**) and r is the part that depends on the charge density ρ. (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,

$$\delta_{n+1} = \mathbf{L}\delta_n \tag{106}$$

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 $(N_x-1)(N_y-1)$, 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 **L** to have magnitudes less than 1. So we can now restate the original miracle, "Iteration on Eq. (**??**) converges," in this way: "All of the eigenvalues of the matrix **L** on the right-hand side of Eq. (**??**) are less than 1 in magnitude." This statement is a theorem which can be proved if you are really good at linear algebra, and the entire iteration procedure described by Eq. (**??**) 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 on the right-hand side of Eq. (**??**), they used values of V as they became available during the iteration. (This means that the right side of Eq. (**??**) 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 $V_{i,j}$ and you use new values as they become available. (When you see this algorithm coded you will understand this better.)

However, even this change is not the best we can do. To understand the next improvement let's go back to the exponential example

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

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

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

where ω is a number which is yet to be determined. Now linearize as before to find how the error changes as we iterate:

$$x_n = \bar{x} + \delta_n \quad \Rightarrow \quad \delta_{n+1} = (-\omega e^{-\bar{x}} + 1 - \omega)\delta_n \tag{107}$$

Now look: what would happen if we chose ω so that the factor in parentheses were zero? The equation says that we would get 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.

(c) Write a Matlab script that accepts a value of ω and runs the iteration in Eq. (**??**). Experiment with various values of ω until you find one that does the best job of accelerating the convergence of the iteration. You should find that the best ω is near 0.64, but it won't give convergence in one step. See if you can figure out why not. (Think about the approximations involved in getting Eq. (**??**).)

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

$$V_{n+1} = \omega\left(\mathbf{L}V_n + r\right) + (1-\omega)V_n \tag{108}$$

then play with ω until we get almost instantaneous convergence.

Sadly, this doesn't quite work. The problem is that in solving for $(N_x - 1)(N_y - 1)$ unknown values of $V_{i,j}$ 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:

$$V_{i,j} = \omega \left(\frac{V_{i+1,j} + V_{i-1,j}}{\Delta x^2} + \frac{V_{i,j+1} + V_{i,j-1}}{\Delta y^2} + \frac{\rho_0}{\epsilon_0} \right) / \left(\frac{2}{\Delta x^2} + \frac{2}{\Delta y^2} \right) + (1-\omega)V_{i,j} \tag{109}$$

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 is given by the formulas

$$\rho = \frac{\Delta y^2 \cos(\pi/N_x) + \Delta x^2 \cos(\pi/N_y)}{\Delta x^2 + \Delta y^2}$$

$$\omega = \frac{2}{1 + \sqrt{1-\rho^2}} \tag{110}$$

whenever the computing region is rectangular and the boundary values of V are fixed (Dirichlet boundary conditions).

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 grid with the potential specified at the edges. If you use a cell-centered grid with ghost points, and especially if you change to normal-derivative boundary conditions, this value of ω won't be quite right. But there is still a best value of ω somewhere near the value given in Eq. (**??**) and you can find it by numerical experimentation.

Finally, we come to the question of when to quit. It is tempting just to watch a value of $V_{i,j}$ at some grid point and quit when its value has stabilized at some level, like this for instance: quit when $\epsilon = |V(i,j)_{n+1} - V(i,j)_n| < 10^{-6}$. You will see this error criterion sometimes used in books, but *do not use it.* I 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

$$\epsilon = \frac{Lhs - Rhs}{V_{scale}} \tag{111}$$

where Lhs is the left-hand side of Eq. (**??**) and Rhs is its right-hand side. Because this equation is just an algebraic rearrangement of our finite-difference approximation to Poisson's equation, ϵ can only be small when Poisson's equation is satisfied. And what error criterion should we choose? 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.

And almost last of all, here is a piece of Matlab code that implements these ideas. You just have to fill in the blank sections of code and it will be ready to go.

```
%beginfunction sor.m

% Solve Poisson's equation by Successive-Over-relaxation
% on a rectangular Cartesian grid
clear; clear memory
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
dx=Lx/(Nx-1); % Grid spacing in x
dy=Ly/(Ny-1); % Grid spacing in y
x = (0:dx:Lx)-.5*Lx;  %x-grid, x=0 in the middle
y = 0:dy:Ly;  %y-grid

% estimate the best omega to use

r = (dy^2*cos(pi/Nx)+dx^2*cos(pi/Ny))/(dx^2+dy^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=0 and x=Lx
Vscale=V0; % set Vscale to the size of the potential in the problem
fprintf('Potential at ends equals %g \n',V0);
fprintf('Potential is zero on all other boundaries\n');

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

%%%%%% Set initial conditions and boundary conditions %%%%%

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

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

% in V(i,j), i is the x-index and j is the y-index

% set the boundary conditions here

% left and right edges are at V0
% (put boundary condition code here:)
.
.
.

% top and bottom are grounded
% (put boundary condition code here:)
.
.
.

%%%%%%% MAIN LOOP %%%%%%%%

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

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

for n=1:Niter

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

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

        % load rhs with the right-hand side of the Vij equation,
        % bottom of page 33
        rhs =  . . .

         % get the relative error, left side - right side
         err(n)= . . .

         % SOR algorithm, update V(i,j) (use rhs from above)
         V(i,j) = . . .

      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
   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])
   pause;

% make a surface plot
   surf(x,y,V');  % Surface plot of the potential
   xlabel('x'); ylabel('y');
   pause

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

%end sor.m
```

(d) Finish writing the script `sor.m` above 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 $V(-a, y) = V(a, y) = 1$ and $V(x, 0) = V(x, b) = 0$ where $a = L_x/2$ and where $b = L_y$ (L_x and L_y are defined in `sor.m`). Set the error criterion to 10^{-4}. Verify that the optimum value of ω given by Eq. (**??**) is the best one to use.

(e) Using the optimum value of ω in each case, run `sor.m` for $N_x = N_y = 10,\ 20,\ 40,$ and 80. See if you can find a rough power law formula for how long it takes to get the error below 10^{-5}, i.e., guess that Run Time $\approx AN_x^p$, and find A and p. The `cputime` command will help you with the timing.

13 Poisson's Equation II

13.1 Read and be prepared to take a short quiz on the following material

Elliptic, Hyperbolic, and Parabolic PDEs and Their Boundary Conditions

The three most famous partial differential equations of classical physics are:
Poisson's equation:

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = \frac{-\rho}{\epsilon_0} \qquad + \text{ Boundary Conditions} \tag{112}$$

for the electrostatic potential $V(x, y)$ given the charge density $\rho(x, y)$;

the wave equation:

$$\frac{\partial^2 y}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 y}{\partial t^2} = 0 \qquad + \text{ Boundary Conditions} \tag{113}$$

for the wave displacement $y(x, t)$;

and the thermal diffusion equation:

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2} \qquad + \text{ Boundary Conditions} \tag{114}$$

for the temperature $T(x, t)$.

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:

$$\text{ellipse}: \quad \frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \tag{115}$$

$$\text{hyperbola}: \quad \frac{x^2}{a^2} - \frac{y^2}{b^2} = 1 \tag{116}$$

$$\text{parabola}: \quad y = ax^2 \tag{117}$$

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 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 $t = t_{final}$ because we don't know the displacement in the future. This means that you can't pretend that (x, t) are like (x, y) in Poisson's equation and use "surrounding"-type boundary conditions. But we can see the right thing to do by thinking about what a guitar string does. With the end positions specified, the motion of the string is determined by giving it an initial displacement $y(x, 0)$ and an initial velocity $\partial y(x, t)/\partial t|_{t=0}$, and then letting the motion run until we reach the final time. So for hyperbolic equations the proper boundary conditions are to specify end conditions on y as a function of time and to specify the initial conditions $y(x, 0)$ and $\partial y(x, t)/\partial t|_{t=0}$.

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

If this seems like an arcane side trip into theory, I'm 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 get the boundary conditions to 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 equations to use when you encounter a new one. Here, for instance, is Schrödinger's equation:

$$i\hbar\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\psi}{\partial x^2} + V\psi \tag{118}$$

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?

$$\frac{\partial^2 y}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 y}{\partial t^2} - \frac{1}{D}\frac{\partial y}{\partial t} = 0 \tag{119}$$

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

13.2 (a) Modify `sor.m` so that the potential on the right side of the rectangular pipe is $-V_0$ instead of V_0. Then make a plot of the surface charge density at the bottom inside surface of the pipe. (To do this you will need to remember the connection between surface charge density and the normal component of $\mathbf{E}$ and how to compute $\mathbf{E}$ from V.)

(b) Modify `sor.m` so that the boundary condition on the right side of the computation region is $\partial V/\partial x = 0$ and the boundary condition on the bottom is $\partial V/\partial y = 0$ You will discover that the code 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.) You can do this problem either by changing your grid and using ghost points or by using a quadratic extrapolation technique. Both methods work fine.

13.3 (a) Modify `sor.m` to solve the problem of a an infinitely long rectangular pipe of x-width 20 cm and y-height 40 cm with the bottom, right side, and 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. Find $V(x,y)$ in both triangular regions. You will probably want to have a special relation between N_x and N_y 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. Matlab's `gradient` command `[Ex,Ey]=gradient(V,x,y)` will let you quickly obtain $\mathbf{E}$ from $\mathbf{E} = -\nabla V$. Use online help to see how to use `gradient` and `quiver`.

13.4 Study electrostatic shielding by going back to the boundary conditions of Problem 13.2, while grounding some points in the interior 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.

14 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. The equations are as follows.

Conservation of mass:

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}(\rho v) = 0 \tag{120}$$

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 also compressed or rarefied, depending on the sign of $\partial v/\partial x$.

14.1 Roughly verify the statement above by expanding the spatial derivative in Eq. (**??**) to put the equation in the form

$$\frac{\partial \rho}{\partial t} + v\frac{\partial \rho}{\partial x} = -\rho\frac{\partial v}{\partial x} \tag{121}$$

(a) If $v =$ const, show that the solution of this equation is $\rho(x,t) = \rho_0(x - vt)$, where $\rho_0(x)$ is the initial distribution of density. 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 $n(x,0) = n_0 =$ 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. (**??**) is now given by $n(x,t) = n_0 e^{-\gamma t}$ and determine the value of the decay constant γ. 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 the solution you get 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:

$$\frac{\partial p}{\partial t} + \frac{\partial}{\partial x}(pv) = -(\gamma - 1)p\frac{\partial v}{\partial x} + F\frac{\partial^2}{\partial x^2}\left(\frac{p}{\rho}\right) \tag{122}$$

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 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 quantity

$$F = \frac{\kappa M}{k_B} \tag{123}$$

where κ is the thermal conductivity, M is the mass of a molecule of the gas, and where k_B is Boltzmann's constant. And the reason that the ratio p/ρ appears in this term is that the ideal gas law tells us that $T \propto p/\rho$.

14.2 Use the ideal gas law in the form $p = nk_BT$, where n is the number of particles per unit volume, to find a formula for T involving the ratio p/ρ.

Newton's second law:

$$\frac{\partial v}{\partial t} + v\frac{\partial v}{\partial x} = -\frac{1}{\rho}\frac{\partial p}{\partial x} \tag{124}$$

On the left side of this equation you should recognize the acceleration dv/dt, and on the right is the pressure force that pushes fluid from high pressure toward low pressure. And the reason for the $1/\rho$ term on the right is that if you multiply by ρ to move it to the left side of the equation we get mass times acceleration (or at least mass density times acceleration.)

You are probably unconvinced that the left side of Eq. (**??**) is acceleration. To become convinced, read and do the following.

14.3 (a) (Just read.) 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 $\partial v/\partial t = 0$.

(b) 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. (**??**).

In the next lab we will actually tackle the hard problem of simultaneously advancing ρ, p, and v in time and space, but in this one 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:

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho v}{\partial x} = 0 \tag{125}$$

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.) In the following problem we will try an algorithm that is unstable, one that is stable but inaccurate, and finally one that is both stable and conservative, but only works well if the solution doesn't get 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 I will show you here.)

Before we continue I need to tell you about the boundary conditions on Eq. (**??**). 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.)

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

$$\frac{\rho_j^{m+1} - \rho_j^m}{\tau} + \frac{1}{2h}\left(\rho_{j+1}^m v_{j+1} - \rho_{j-1}^m v_{j-1}\right) = 0 \qquad (126)$$

This involves a nice centered difference in x and an inaccurate forward difference in t. Solve this equation for ρ_j^{m+1} and use it in a time-advancing script like the one you built to do the wave equation in Lab 6. I 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

$$\rho(x,0) = 1 + e^{-200(x/L-1/2)^2} \qquad (127)$$

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

$$v(x) = v_0 \qquad (128)$$

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:

$$\rho(L,t) = 2\rho(L-h,t) - \rho(L-2h,t) \qquad (129)$$

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

14.5 Now let's try the Lax-Wendroff method. Again, use a cell-center grid with ghost points and about 400 grid points. Also use the same initial condition as in 14.4 and use the extrapolated boundary condition that just lets the pulse leave.

The idea here is to use a Taylor series in time to get a second-order accurate method.

$$\rho(x,t+\tau) = \rho(x,t) + \tau\frac{\partial\rho}{\partial t} + \frac{\tau^2}{2}\frac{\partial^2\rho}{\partial t^2} \tag{130}$$

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

$$\rho(x,t+\tau) = \rho(x,t) - \tau\frac{\partial\rho v}{\partial x} + \frac{\tau^2}{2}\frac{\partial}{\partial x}\left(v\frac{\partial\rho v}{\partial x}\right) \quad . \tag{131}$$

If you stare at this equation for a minute you will see that a diffusion-like term has showed up. 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 14.4. Notice also that the diffusion coefficient is proportional to τ, so if small time steps are being used (stare at it until you can see that this is true) diffusion won't hurt us too much.

(b) Now finite difference the expression in (a) assuming that $v(x) = v_0 =$ const, as in 14.4 to find the Lax-Wendroff algorithm:

$$\rho_j^{m+1} = \rho_j^m - \frac{v0\tau}{2h}[\rho_{j+1}^m - \rho_{j-1}^m] + \frac{v_0^2\tau^2}{2h^2}[\rho_{j+1}^m - 2a_j^m + \rho_{j-1}^m] \tag{132}$$

Change your code from 14.4 to use the Lax-Wendroff algorithm and show that it 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 the largest time step possible.) 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 14.4, I suppose we should be grateful.

14.6 Now let's try an implicit method, Crank-Nicholson in fact. Proceeding as we did with the diffusion equation and Schrödinger's equation we finite difference Eq. (**??**) like this:

$$\frac{\rho_j^{m+1} - \rho_j^m}{\tau} + \frac{1}{4h}\left(\rho_{j+1}^{m+1}v_{j+1} - \rho_{j-1}^{m+1}v_{j-1} + \rho_{j+1}^m v_{j+1} - \rho_{j-1}^m v_{j-1}\right) = 0 \tag{133}$$

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

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

$$(\mathbf{I} - \mathbf{Q})\rho^{m+1} = (\mathbf{I} + \mathbf{Q})\rho^m \tag{134}$$

by finding $Q_{j,j-1}$, $Q_{j,j}$, and $Q_{j,j+1}$ (the other elements of $\mathbf{Q}$ 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 matrix $\mathbf{Q}$ and the right-hand side of Eq. (**??**). (This is tricky, so be careful.)

(c) Implement this algorithm with a constant convection velocity and show that it 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.

(You won't be able to use `tridag` because the extrapolation boundary condition at the right edge uses too many grid points. So define $\mathbf{Q}$ to be a square matrix and use \ to do the linear solve.)

(d) Explore the way this algorithm behaves when we have a shock wave (discontinuous density) by using as the initial condition

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

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

(e) For comparison, try this same initial condition in your Lax-Wendroff script from Problem 14.5.

15 Gas Dynamics II

Now we are going to use the implicit algorithm of the previous lab as a tool to solve the three nonlinear coupled partial differential equations of one-dimensional gas dynamics. Here they are.

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}(\rho v) = 0 \tag{135}$$

(Note: these are the equations of one-dimensional sound waves in a long tube which is wide enough that friction with the walls doesn't matter.)

$$\frac{\partial p}{\partial t} + \frac{\partial}{\partial x}(pv) = -(\gamma - 1)p\frac{\partial v}{\partial x} + F\frac{\partial^2}{\partial x^2}\left(\frac{p}{\rho}\right) \tag{136}$$

$$\frac{\partial v}{\partial t} + v\frac{\partial v}{\partial x} = -\frac{1}{\rho}\frac{\partial p}{\partial x} \tag{137}$$

For disturbances in air at sea level we have $p = 1 \times 10^5$ Pa, $\rho = 1.3$ kg/m^3, $\gamma = 1.4$, and $F = 4.2 \times 10^{-5}$ kg/m·sec. We will solve these equations in a tube of length $L = 10$ m with closed ends through which there is no flow of heat so that $\partial T/\partial x = 0$ at the ends.

15.1 Because the wall ends are fixed and the gas can't pass through these walls it should be clear that the boundary conditions on the velocity are $v(0,t) = v(L,t) = 0$. Use this fact, the condition $\partial T/\partial x = 0$ and the equation of state $p = \rho k_B T/M$, and the three gas dynamic equations at the beginning of this lab to derive the following boundary conditions on ρ and p.

$$\frac{\partial \rho}{\partial x} = 0 \quad \text{and} \quad \frac{\partial p}{\partial x} = 0 \quad \text{at } x = 0 \quad \text{and} \quad x = L$$

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, p, 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. I'm going to show you one way to do it, but the computational physics literature is full of other ways, including methods that handle shock waves. This is still a very active and evolving area of research, especially for problems in 2 and 3 dimensions.

Let's try a predictor-corrector method similar to second-order Runge-Kutta. For simplicity, in the predictor step we will use v_j^m in Eqs. (**??**) and (**??**), and in the thermal diffusion term in Eq. (**??**) we will use ρ_j^m as well. With these choices we can use Crank-Nicholson on Eqs. (**??**) and (**??**) to obtain estimates of the future values of density and pressure ρ^* and p^*. Here are the equations above written so that Crank-Nicholson can be applied to each one in turn to get predicted values of ρ, p, and v:

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}(\rho v^m) = 0 \tag{138}$$

$$\frac{\partial p}{\partial t} + \frac{\partial}{\partial x}(p v^m) = -(\gamma - 1)p\frac{\partial v^m}{\partial x} + F\frac{\partial^2}{\partial x^2}\left(\frac{p}{\rho^m}\right) \tag{139}$$

$$\frac{\partial v}{\partial t} + v^m \frac{\partial v}{\partial x} = -\frac{1}{\rho^m}\frac{\partial p^m}{\partial x} \tag{140}$$

We can obtain v^* similarly by using ρ_j^m and p_j^m on the right-hand side of Eq. (**??**) and also by using v_j^m in the multiplier in front of the term containing $\partial v/\partial x$.

The corrector step also uses Crank-Nicholson on the same equations, but with the terms that we left back at time level m in the predictor step advanced approximately to time level $m + 1/2$ by averaging with starred quantities like this.

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}\left(\rho[v^m + v^*]/2\right) = 0 \tag{141}$$

$$\frac{\partial p}{\partial t} + \frac{\partial}{\partial x}\left(p[v^m + v^*]/2\right) = -(\gamma - 1)p\frac{\partial [v^m + v^*]/2}{\partial x} + F\frac{\partial^2}{\partial x^2}\left(\frac{p}{[\rho^m + \rho^*]/2}\right) \tag{142}$$

$$\frac{\partial v}{\partial t} + [v^m + v^*]/2\frac{\partial v}{\partial x} = -\frac{1}{[\rho^m + \rho^*]/2}\frac{\partial [p^m + p^*]/2}{\partial x} \tag{143}$$

15.2 (a) Implement this algorithm by modifying your Crank-Nicholson code from Lab 14.

(b) Test the code by making sure that small disturbances travel at the sound speed $c = \sqrt{\gamma p/\rho}$. To do this set p and ρ to their atmospheric values and set the velocity to

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

with $v_0 = c/100$. If you look carefully at the deviation of the pressure from atmospheric you should see two oppositely propagating signals traveling at the sound speed.

(c) With $F = 0$ 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-Nicholson's failure to properly deal with shock waves.

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