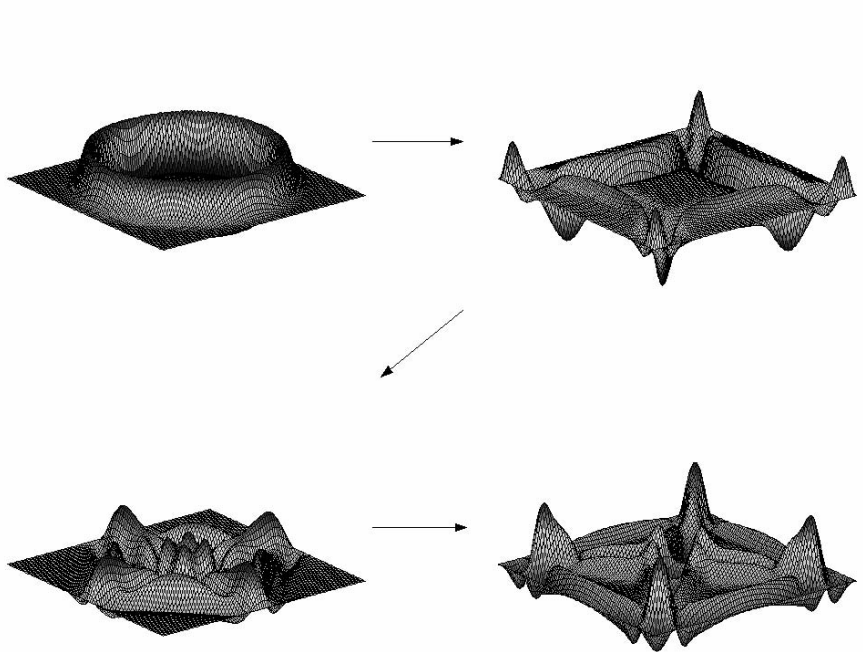
COMPUTATIONAL PHYSICS 430

PARTIAL DIFFERENTIAL EQUATIONS



Ross L. Spencer and Michael Ware

Department of Physics and Astronomy

Brigham Young University

COMPUTATIONAL PHYSICS 430

PARTIAL DIFFERENTIAL EQUATIONS

Ross L. Spencer and Michael Ware

N263 ESC 422-2186 michael_ware@byu.edu

Department of Physics and Astronomy

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 especially Matlab, which will be used extensively in this course. 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. The course objectives are to

- learn to use loops, logic commands, and other programming techniques to solve partial differential equations;
- gain a better understanding of what partial differential equations mean
- learn to apply the ideas of linear algebra to physical systems described by data on grids.

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 when things go wrong. It will take a lot longer to do these exercises if you are completely on your own.

Finally, you should consider buying the student version of Matlab while you still have a student ID and it is cheap. You will become quite skilled in its use and it would be very helpful to have it on your own computer.

Contents

Lab 1

Grids and Loops

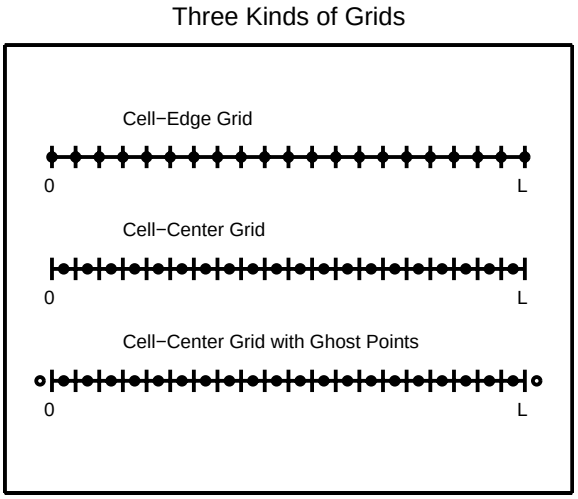


Figure 1.1 Three common computational grids

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.

1.1. (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;          % calculate the step size
x=a:h:b;            % build 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 a 50-point cell-edge grid in x and a 70-point cell-edge grid in y using $a = 0$, $b = 2$, $c = -1$, $d = 3$. Then use Matlab's `ndgrid` command to make 2-d grids X and Y from the 1-d grids x and y like this.

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

Examine the contents of X and Y thoroughly 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:

$$f(x,y) = e^{(-(x^2+y^2))} \cos\left(5\sqrt{x^2+y^2}\right) \quad ; \quad g(x,y) = J_0(x^2+3y^2) \tag{1.1}$$

It will be instructive to compare these two ways of using `surf`:

```
surf(f)
```

and

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

See if you can understand why the two plots are not the same, and convince yourself that you should always use the second form. Then properly label the x and y axes with the symbols x and y, like this:

```
surf(X,Y,f);
xlabel('x');
ylabel('y');
```

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

When you have succeeded in getting the plots to come out right, see what happens when you do the surface plot this way instead:

```
surf(f);
xlabel('x');
ylabel('y');
```

Make sure that you understand what's wrong here and remind yourself always to use the long form `surf(X,Y,f)` when using `ndgrid`.

Loops and Logic

This 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.

1.2. (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

$$\sum_{n=1}^{N} n = \frac{N(N+1)}{2} \tag{1.2}$$

is correct

(c) For various values of x perform the sum

$$\sum_{n=1}^{1000} nx^n \tag{1.3}$$

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

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6} \tag{1.4}$$

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:

$$\prod_{n=1}^{\infty} \left(1 + \frac{1}{n^2}\right) = \frac{\sinh \pi}{\pi} \tag{1.5}$$

(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.

$$x_{n+1} = e^{-x_n} \quad ; \quad x_{n+1} = \cos x_n \quad ; \quad x_{n+1} = \sin 2x_n \tag{1.6}$$

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.

Finally, try iteration again on this problem:

$$x_{n+1} = \sin 3x_n \tag{1.7}$$

Convince yourself that this process isn't converging to anything. We will see in Lab 11 what makes the difference between an iteration process that converges and one that doesn't.

Interpolation and Extrapolation

We will be using *interpolation* and *extrapolation* techniques fairly often during this course, so let's review these ideas. Even though we are using a grid, there are times when we want to find good values of a function *between* grid points (interpolation), or even beyond the grid (extrapolation).

The simplest way to estimate these values is to use the fact that two points define a straight line. For example, suppose that we have function values (x_1, y_1) and (x_2, y_2). The formula for a straight line that passes through these two points is

$$y - y_1 = \frac{(y_2 - y_1)}{(x_2 - x_1)}(x - x_1) \tag{1.8}$$

Once this line has been established it provides an approximation to the true function $y(x)$ that is pretty good in the neighborhood of these two data points. To linearly interpolate or extrapolate is simply to use this formula.

1.3. Use Eq. (1.8) to do the following special cases:

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

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

(c) If $x_2 = x_1 + h$, find an approximate formula for $y(x_2 + h)$, i.e., estimate the function value one grid step beyond the last grid point, then do it again for $y(x_2 + h/2)$. This is called linear extrapolation, and we will use both of these formulas during the course.

A fancier technique is to use a parabola instead of a line. It takes 3 data points to define a parabola, so we need to start with the function values (x_1, y_1), (x_2, y_2), and (x_3, y_3). For simplicity, let's assume that these are points on a grid so that $x_2 = x_1 + h$ and $x_3 = x_1 + 2h$.

The general formula for a parabola is

$$y = a + bx + cx^2 \tag{1.9}$$

where the coefficients a, b, c are unknown at this point and need to be chosen so that the parabola passes through our three data points. To determine these constants, set up three equations that force the parabola to match the data points, like this:

$$y_i = a + bx_i + cx_i^2 \tag{1.10}$$

with $i = 1, 2, 3$.

1.4. Ask Maple to solve the equations to obtain really messy formulas for a, b, c, then use this formula to solve the following problems:

(a) Estimate $y(x)$ half way between x_1 and x_2, and then again halfway between x_2 and x_3. (Use Maple's `simplify` command). Do you see a pattern?

(b) Estimate $y(x)$ one grid point beyond x_3, i.e., at $x_3 + h$ to obtain a formula for quadratic extrapolation. Do it again for $x_3 + h/2$.

Lab 2

Derivatives on Grids

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

$$f'(x) = \frac{f(x+h) - f(x)}{h} \tag{2.1}$$

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.

2.1. 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:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{2.2}$$

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

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \tag{2.3}$$

You will use Maple to derive both of these formulas a little later, but for now I just want you to understand how to use them.

2.2. 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.

If you do this problem correctly you will find that $h = 0.001$ in the centered-difference formula gives a better approximation than $h = 0.01$, which invites you to keep making h smaller and smaller to achieve any accuracy you want. This doesn't work, as Fig. 2.1 shows. When h becomes too small the roundoff effect discussed at the beginning of this problem starts to be important until finally (at about $h = 10^{-16}$, and sooner for the second

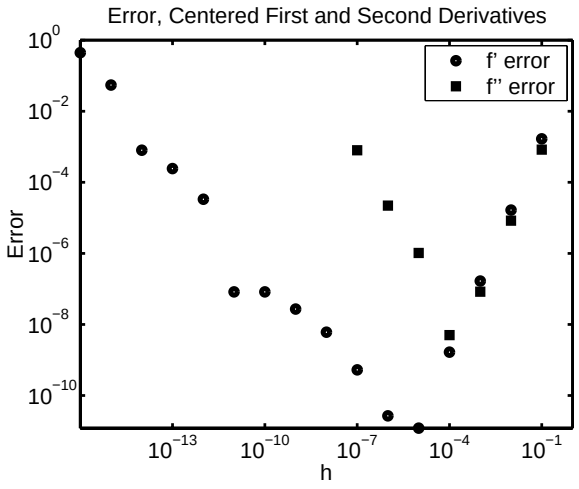


Figure 2.1 Error in the centered-difference approximation to df/dx as a function of h.

derivative) the subtraction in the numerator of the difference formula has no accuracy at all and the error is of order 1. To make this clear, look at this subtraction problem where the two numbers are nearly the same. Notice that our nice 15-digit accuracy has disappeared, leaving behind only 6 significant figures:

$$\begin{array}{rr} & 7.38905699669556 \\ - & 7.38905699191745 \\ \hline & 0.00000000477811 \end{array} \tag{2.4}$$

Also notice in Fig. 2.1 that this problem is worse for the second derivative formula than it is for the first derivative formula. The lesson here is that it is impossible to achieve high accuracy by using tiny values of h. In a problem with a size of about L it doesn't do any good to use values of h any smaller than about $0.0001L$.

Understanding Approximate Derivative Formulas

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

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \ \cdots \ + f^{(n)}(x)\frac{h^n}{n!} + \ \cdots \tag{2.5}$$

This series usually converges quite rapidly if h is small.

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

$$\begin{aligned} \frac{f(x+h)-f(x)}{h} &= \frac{\left[f(x)+f'(x)h+\frac{1}{2}f''(x)h^2+\cdots\right]-f(x)}{h} \\ &\approx f'(x)+\frac{h}{2}f''(x) \ . \end{aligned} \tag{2.6}$$

The expansion of $f(x+h)$ has been terminated at the $f''(x)$ term in the second line because higher order terms just give smaller corrections to the result (i.e. they are multiplied by higher powers of h). Solving this equation for the exact derivative $f'(x)$ (while ignoring the higher order terms) gives us

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} - \frac{h}{2} f''(x) \tag{2.7}$$

From this formula we see that the forward difference does indeed give the first derivative back, but that it doesn't quite do 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. (2.2) yields

$$\begin{aligned} \frac{f(x+h) - f(x-h)}{2h} &= \frac{\left[f(x) + f'(x)h + f''(x)\frac{h^2}{2} + f'''(x)\frac{h^3}{6} + \cdots\right]}{2h} \\ &\quad - \frac{\left[f(x) - f'(x)h + f''(x)\frac{h^2}{2} - f'''(x)\frac{h^3}{6} + \cdots\right]}{2h} \\ &\approx f'(x) + f'''(x)\frac{h^2}{6} \end{aligned} \tag{2.8}$$

Solving this equation for the exact derivative $f'(x)$ (again ignoring the higher-order terms indicated by $\cdots$) gives us

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6} f'''(x) \tag{2.9}$$

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.

2.3. Verify that the error estimates in Eqs. (2.7) and (2.9) agree with the numerical testing you did with e^x in Problem 2.2.

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.

2.4. (a) 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.

(b) 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.

(c) Examine your solution from (b) 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 find Eq. (2.3).

(d) 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 (a)-(c) 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.

Now let's see how we use Matlab to differentiate a function. You've seen this before in Physics 330, but it will probably help to review. 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. (2.2) and (2.3) 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);
```

2.5. (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. To do this, remember that the formula for a parabola is

$$f(x) = a + bx + cx^2 \quad . \tag{2.10}$$

To make a parabola that goes through the three points $(x_1, f_1), (x_2, f_2), (x_3, f_3)$ just write down the three equations that force the parabola to go through these three points (here is the first one)

$$f_1 = a + bx_1 + cx_1^2 \tag{2.11}$$

and solve them simultaneously for a, b, and c. Then make x_1, x_2, and x_3 equally spaced with step size h, and evaluate the parabola at $x_4 = x_3 + h$ and simplify to find the extrapolation formula.

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 reducing the error, 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. (If your second derivative traces don't match it may be because you used `/x` instead of `./x` which is always required when plotting arrays of data.)

Lab 3

Differential Equations on Grids

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

$$y''(x) + 9y(x) = x \quad ; \quad y(0) = 0, \quad y(2) = 1 \tag{3.1}$$

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;
x=x';
```

We will be using lots of grids this semester; stare at this piece of code and make sure you understand what it does. You may be wondering about the command `x=x'`. This turns the row vector `x` into a column vector `x`. This is not strictly necessary, but it is good programming practice because of all of the linear algebra we will be doing during the course. Many of our results will be stored as column vectors, so it is helpful for the x-array to also be a column vector.

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:

$$\frac{y_{j+1} - 2y_j + y_{j-1}}{h^2} + 9y_j = x_j \tag{3.2}$$

Now let's think about this equation for a bit. First notice that it is not *an* equation; it is many equations, because 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 $j = 2 \ldots N-1$, it is really a system of $N-2$ coupled equations in the N unknowns $y_1 \ldots y_N$. If we had just two more equations we could find the y_j's by solving a linear system of equations. But we do have two more equations; they are the boundary conditions:

$$y_1 = 0 \quad ; \quad y_N = 1 \tag{3.3}$$

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,

$$\mathbf{Ay} = \mathbf{b}$$

with **A** the matrix of coefficients, **y** the column vector of unknown y-values, and **b** the column vector of known values on the right-hand side of Eq. (3.2). Stare at the matrix equation below until you are convinced that it is equivalent to Eqs. (3.2) and (3.3). (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. (3.2) and (3.3).)

$$\begin{bmatrix} 1 & 0 & 0 & 0 & ... & 0 & 0 & 0 \\ \frac{1}{h^2} & -\frac{2}{h^2}+9 & \frac{1}{h^2} & 0 & ... & 0 & 0 & 0 \\ 0 & \frac{1}{h^2} & -\frac{2}{h^2}+9 & \frac{1}{h^2} & ... & 0 & 0 & 0 \\ . & . & . & . & ... & . & . & . \\ . & . & . & . & ... & . & . & . \\ . & . & . & . & ... & . & . & . \\ 0 & 0 & 0 & 0 & ... & \frac{1}{h^2} & -\frac{2}{h^2}+9 & \frac{1}{h^2} \\ 0 & 0 & 0 & 0 & ... & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ . \\ . \\ . \\ y_{N-1} \\ y_N \end{bmatrix} = \begin{bmatrix} 0 \\ b_2 \\ b_3 \\ . \\ . \\ . \\ b_{N-1} \\ 1 \end{bmatrix} \tag{3.4}$$

with $b_j = x_j$, the right-hand side of Eq. (3.2).

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. Matlab does this solve with this command: `y=A\b`.

3.1. (a) Use Maple's `dsolve` command to solve the differential equation Eq. (3.1) 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 as a red curve (`'r-'`).

(b) Now load the matrix above and do the linear solve to obtain y_j and plot it on top of the exact solution with blue stars (`'b*'`) 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 using the matrix method in Matlab and compare the two solutions.

$$y'' + \frac{1}{x}y' + (1 - \frac{1}{x^2})y = x \quad ; \quad y(0) = 0, \quad y(5) = 1 \tag{3.5}$$

(The solution is displayed in Fig. 3.1.)

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

$$y'' + \sin(x)y' + e^x y = x^2 \quad ; \quad y(0) = 0, \quad y(5) = 3 \tag{3.6}$$

(If you try to solve this equation with Maple it will fail, unless you use `dsolve(...type=numeric)` and increase `abserr` to about 10^{-5} and set `maxmesh` to about 2000.) Use `odeplot` to display the solution, then compare the Maple plot with the Matlab plot. Do they agree? Check both solutions at $x = 4.5$; is the agreement reasonable?

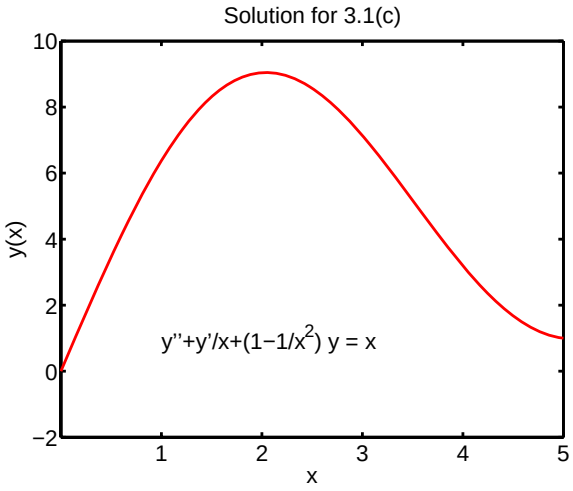


Figure 3.1 Solution to Problem 3.1(c)

(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, so they can be translated into *linear* algebra problems. Here is a simple example of one that isn't linear:

$$y''(x) + \sin[y(x)] = 1 \quad ; \quad y(0) = 0, \quad y(3) = 0 \tag{3.7}$$

Work at turning this problem into a linear algebra problem just long enough to see that it can't be done. Then solve it in Maple by using `dsolve` with the `type=numeric` option. If you assign the result of `dsolve` to a variable, then you can use `odeplot` to plot $y(x)$ like this:

```
s:=dsolve({...},y(x),type=numeric);
with(plots):
odeplot(s,[x,y(x)],0..3);
```

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. (3.7) in Matlab on a grid. Partial answer: $y(x)$ is negative with a minimum at $x = 1.5$ of value $y(1.5) = -2.13$.

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

$$y''(x) + 9y(x) = x \quad ; \quad y(0) = 0 \quad ; \quad y'(2) = 0 \tag{3.8}$$

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 crude way to implement the derivative boundary condition is to use a forward difference formula

$$\frac{y_N - y_{N-1}}{h} = y'|_{x=2} \quad . \tag{3.9}$$

In the present case, where $y'(2) = 0$, this simply means that we set $y_N = y_{N-1}$.

3.2. (a) Modify your script for Problem 3.1 to use the boundary conditions in Eq. (3.8) and compare the resulting numerical solution to the exact solution obtained from Maple:

$$y(x) = \frac{x}{9} - \frac{\sin(3x)}{27\cos(6)} \tag{3.10}$$

(b) Improve your script 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 $[x_{N-2}, x_N]$, with $y(x) = ax^2 + bx + c$. 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 find a new numerical boundary condition to apply at N. You should find that the new condition is

$$\frac{1}{2h}y_{N-2} - \frac{2}{h}y_{N-1} + \frac{3}{2h}y_N = y'(2) = 0 \tag{3.11}$$

(c) Modify your script from part (a) to include this new condition and show that it gives a more accurate solution than the crude technique of part (a).

Lab 4

The Wave Equation: Steady State and Resonance

To see why we did so much work in Lab 3 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:

$$\mu \frac{\partial^2 y}{\partial t^2} = T \frac{\partial^2 y}{\partial x^2} + f(x) \cos \omega t \quad ; \quad y(0,t) = 0, \quad y(L,t) = 0 \tag{4.1}$$

where $y(x,t)$ is the (small) sideways displacement of the string as a function of position and time, assuming that $y(x,t) \ll L$. [1] This equation may look a little unfamiliar to you, so let's discuss each term. 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 rather the linear mass density (mass/length). This means that the right side should have units of force/length, and it does because T is the tension (force) in the string and $\partial^2 y/\partial x^2$ has units of 1/length (take a minute and verify that this is true.) Finally, $f(x)$ is the 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 this steady-state solution.

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

$$y(x,t) = g(x) \cos \omega t \tag{4.2}$$

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, after some rearrangement,

$$T g''(x) + \mu \omega^2 g(x) = -f(x) \quad ; \quad g(0) = 0, \quad g(L) = 0 \tag{4.3}$$

which is just a two-point boundary value problem of the kind we studied in Lab 3.

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

$$f(x) = \begin{cases} 0.73 & \text{if} \quad 0.8 \le x \le 1 \\ \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

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

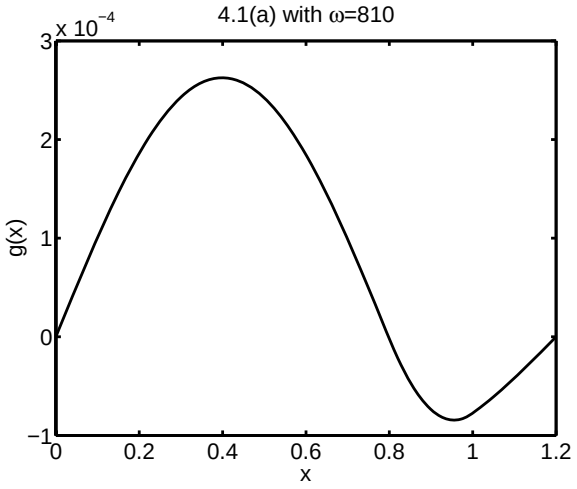


Figure 4.1 Amplitude of a driven string under tension

(All quantities are in SI units.)

Modify one of your Matlab scripts from Lab 3 to solve Eq. (4.3) 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.)

Resonance and the Eigenvalue Problem

In problem 4.1(b) you should have noticed an apparent resonance behavior, with resonant frequencies near $\omega = 550$ and $\omega = 1100$. Now we will learn how to use Matlab to find these resonant frequencies directly (i.e. without solving the differential equation over and over again). 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. (4.3) for which the driving force $f(x) = 0$, i.e., we try to solve Eq. (4.3) in the form

$$-\mu\omega^2 g(x) = T g''(x) \quad ; \quad g(0) = 0, \quad g(L) = 0 \tag{4.5}$$

If we rewrite this equation in the form

$$g''(x) = -\left(\frac{\mu\omega^2}{T}\right) g(x) \tag{4.6}$$

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

$$Ag = \lambda g \tag{4.7}$$

where A is a linear operator (the second derivative on the left side of Eq. (4.6)) and λ is the eigenvalue ($-\mu\omega^2/T$ in Eq. (4.6).)

Equation (4.6) is easily solved analytically, and its solutions are just the familiar sine and cosine functions. The condition $g(0) = 0$ tells us to try a sine function form, $g(x) = g_0 \sin kx$. To see if this form works we substitute it into Eq. (4.6 and find that it does indeed work, provided that the constant k is $k = \omega\sqrt{\mu/T}$. We have, then,

$$g(x) = g_0 \sin\left(\omega\sqrt{\frac{\mu}{T}}x\right) \tag{4.8}$$

where g_0 is the arbitrary amplitude. But we still have one more condition to satisfy: $g(L) = 0$. The only thing that remains undetermined is the frequency ω, so this final condition tells us the resonant frequencies of the string:

$$\omega = \frac{n\pi}{L}\sqrt{\frac{T}{\mu}} \tag{4.9}$$

For this simple example we were able to do the eigenvalue problem analytically without much trouble. However, when the differential equation is not so simple we will need to do the eigenvalue calculation numerically, so let's see how it works in this simple case. Rewriting Eq. (4.5) in matrix form, as we learned to do by finite differencing the second derivative, yields

$$\mathbf{Ag} = \lambda\mathbf{g} \tag{4.10}$$

which is written out as

$$\begin{bmatrix} ? & ? & ? & ? & ... & ? & ? & ? \\ \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & ... & 0 & 0 & 0 \\ 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & ... & 0 & 0 & 0 \\ . & . & . & . & ... & . & . & . \\ . & . & . & . & ... & . & . & . \\ . & . & . & . & ... & . & . & . \\ 0 & 0 & 0 & 0 & ... & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} \\ ? & ? & ? & ? & ... & ? & ? & ? \end{bmatrix} \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ . \\ . \\ . \\ g_{N-1} \\ g_N \end{bmatrix} = \lambda \begin{bmatrix} ? \\ g_2 \\ g_3 \\ . \\ . \\ . \\ g_{N-1} \\ ? \end{bmatrix} \tag{4.11}$$

where $\lambda = -\omega^2\frac{\mu}{T}$. The question marks in the first and last rows remind us that we have to invent something to put in these rows that will implement the correct boundary conditions. Note that having question marks in the g-vector on the right is a real problem because without g_1 and g_N in the top and bottom positions, we don't have an eigenvalue problem (i.e. the vector $\mathbf{g}$ on left side of Eq. (4.11) is not the same as the vector $\mathbf{g}$ on the right side).

The simplest way to deal with this question-mark problem and to also handle the boundary conditions is to change the form of Eq. (4.7) to the slightly more complicated form of a *generalized eigenvalue problem*, like this:

$$Ag = \lambda Bg \tag{4.12}$$

where B is another matrix, whose elements we will choose to make the boundary conditions come out right. To see how this is done, here is the generalized modification of Eq. (4.11)

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

$$\underset{A}{\begin{bmatrix} 1 & 0 & 0 & ... & 0 & 0 \\ \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & ... & 0 & 0 \\ 0 & \frac{1}{h^2} & -\frac{2}{h^2} & ... & 0 & 0 \\ . & . & . & ... & . & . \\ . & . & . & ... & . & . \\ . & . & . & ... & . & . \\ 0 & 0 & 0 & ... & -\frac{2}{h^2} & \frac{1}{h^2} \\ 0 & 0 & 0 & ... & 0 & 1 \end{bmatrix}} \underset{g}{\begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ . \\ . \\ . \\ g_{N-1} \\ g_N \end{bmatrix}} = \lambda \underset{B}{\begin{bmatrix} 0 & 0 & 0 & ... & 0 & 0 \\ 0 & 1 & 0 & ... & 0 & 0 \\ 0 & 0 & 1 & ... & 0 & 0 \\ . & . & . & ... & . & . \\ . & . & . & ... & . & . \\ . & . & . & ... & . & . \\ 0 & 0 & 0 & ... & 1 & 0 \\ 0 & 0 & 0 & ... & 0 & 0 \end{bmatrix}} \underset{g}{\begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ . \\ . \\ . \\ g_{N-1} \\ g_N \end{bmatrix} = \lambda \quad (4.13)$$

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

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

(i) Load the matrix A with the matrix on the left side of Eq. (4.13) and the matrix B with the matrix on the right side.

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

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

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

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

This is such a common calculation that I will give you a section of a Matlab script below that does steps (ii) and (iii).

eigen.m

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

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

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

```
                    % eigenvector in V

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

4.2. Use these ideas to numerically find the eigenvalues and eigenvectors of Eq. (4.5). In the process you will see that two infinite eigenvalues appear together with odd-looking eigenvectors that don't satisfy the boundary conditions. These two show up because of the two rows of the B matrix that are filled with zeros. They are numerical artifacts with no physical meaning, so just ignore them. You will also see that most of the eigenvectors are very jagged. These must also be ignored because they are poor approximations to the continuous differential equation in Eq. (4.5). But a few of the smooth eigenfunctions are very good approximations and you should find that their corresponding values of ω match the resonances that you found in 4.1(b). Their values are given by Eq. (4.9).

Go back to your calculation in part 4.1(b) and use these resonant values of ω as driving frequencies. You should find very large amplitudes, indicating that you are right on the resonances.

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

4.3. (a) Change your program for 4.2 to implement the boundary condition

$$\frac{dg}{dx} = 0 \quad \text{at} \quad x = L$$

Use the approximation you derived last week for the derivative $g'(L)$

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

to implement this boundary condition. Explain physically why the resonant frequencies change as they do.

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

$$\frac{dg}{dx} = 2g \quad \text{at} \quad x = L$$

Find the first few resonant frequencies and eigenfunctions for this case. Look at your eigenfunctions and verify that the boundary condition is satisfied.

Lab 5

The Hanging Chain and Quantum Bound States

Consider the chain hanging from the ceiling in the classroom. [1] We are going to find its normal modes of vibration using the method of Problem 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 $x = L$.

5.1. 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

$$T(x) = \mu g x \tag{5.1}$$

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 transverse waves on a chain with varying tension $T(x)$ and varying linear mass density $\mu(x)$ are described by the equation

$$\mu(x)\frac{\partial^2 y}{\partial t^2} - \frac{\partial}{\partial x}\left(T(x)\frac{\partial y}{\partial x}\right) = 0 \tag{5.2}$$

As in Problem 4.1, 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

$$x\frac{d^2 f}{dx^2} + \frac{df}{dx} = -\frac{\omega^2}{g} f \tag{5.3}$$

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

$$f'(0) = -\frac{\omega^2}{g} f(0) = \lambda f(0) \tag{5.4}$$

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 cell-centered grid with ghost points, as discussed in Lab 1. Recall that 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

[1] N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 299-305.

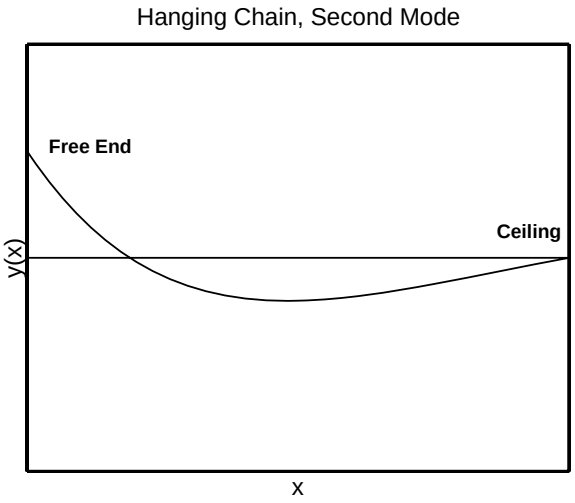


Figure 5.1 The shape of the second mode of a hanging chain

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.

Run the script 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.

chainex.m

```
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:

$$\frac{f_{N+2} + f_{N+1}}{2} = 0 \quad , \tag{5.5}$$

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

$$\frac{f_{N+2} - f_{N+1}}{h} = 0 \quad . \tag{5.6}$$

When we did boundary conditions in the eigenvalue calculation of Lab Problem 4.2 we used a B matrix with zeros in the top and bottom rows and we loaded the top and bottom rows of A with an appropriate boundary condition operator. Because the chain is fixed at the ceiling ($x = L$) we use this technique again in the bottom rows of A and B, like this (after first loading A with zeros and B with the identity matrix):

$$A(N+2, N+1) = \frac{1}{2} \quad A(N+2, N+2) = \frac{1}{2} \quad B(N+2, N+2) = 0 \tag{5.7}$$

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

$$Af = \lambda Bf \tag{5.8}$$

give the boundary condition in Eq. (5.5) at the ceiling no matter what λ turns out to be.

(b) Now let's try to do something similar with the condition at the bottom, Eq. (5.4). Since this condition is already in eigenvalue form we don't need to load the top row of B with zeros. Instead we load A with the operator on the left ($f'(0)$) and B with the operator on the right ($f(0)$), leaving the eigenvalue $\lambda - \omega^2/g$ out of the operators so that we still have $Af = \lambda Bf$. Verify that the following choices for the top rows of A and B correctly produce Eq. (5.4).

$$A(1,2) = \frac{1}{h} \quad A(1,1) = -\frac{1}{h} \quad B(1,1) = \frac{1}{2} \quad B(1,2) = \frac{1}{2} \tag{5.9}$$

Now that the top and bottom rows are built all that remains is to load the rest of the rows by finite differencing Eq. (5.3). (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.)

(c) Load the matrices A and B and use Matlab to solve for the normal modes of vibration of a hanging chain. (As in Lab 4, some of the eigenvectors are unphysical because they don't satisfy the boundary conditions; ignore them.) We will then compare your answers to measurements we will make on the chain hanging from the ceiling in the classroom.

(d) Solve Eq. (5.3) 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.

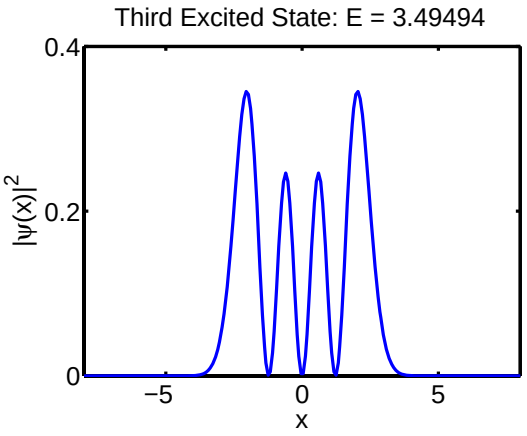


Figure 5.2 The probability distribution for the third excited state of the harmonic oscillator

Quantum Bound States

Consider the problem of a particle in a 1-dimensional harmonic oscillator well in quantum mechanics. [2] Schrödinger's equation for the bound states in this well is

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2}+\frac{1}{2}kx^2\psi=E\psi \tag{5.10}$$

with boundary conditions $\psi = 0$ at $\pm\infty$.

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

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

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

$$C\left(-\frac{D}{2}\frac{d^2\psi}{d\xi^2}+\frac{1}{2}\xi^2\psi\right)=E\psi \tag{5.11}$$

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

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

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

$$a = \left(\frac{\hbar^2}{km}\right)^{1/4} = \sqrt{\frac{\hbar}{m\omega}} \quad \text{where} \quad \omega = \sqrt{\frac{k}{m}} \tag{5.12}$$

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

$$-\frac{1}{2}\frac{d^2\psi}{d\xi^2} + \frac{1}{2}\xi^2\psi = \epsilon\psi \tag{5.13}$$

You should find that

$$\bar{E} = \hbar\sqrt{\frac{k}{m}} \tag{5.14}$$

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

Now that Schrödinger's equation is in dimensionless form it makes sense to choose a grid that goes from -4 to 4, or some other similar pair of numbers. These numbers are supposed to approximate infinity in this problem, so make sure (by looking at the eigenfunctions) that they are large enough that the wave function goes to zero with zero slope at the edges of the grid. As a guide to what you should find, Figure 5.2 displays the square of the wave function for the third excited state.

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

$$E_n = (n + \frac{1}{2})\hbar\sqrt{\frac{k}{m}} = (n + \frac{1}{2})\bar{E} \tag{5.15}$$

so that the dimensionless energy eigenvalues ϵ_n are given by

$$\epsilon_n = n + \frac{1}{2} \tag{5.16}$$

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

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

$$V(x) = \mu x^4 \tag{5.17}$$

so that we have

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + \mu x^4\psi = E\psi \tag{5.18}$$

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

$$-\frac{1}{2}\frac{d^2\psi}{d\xi^2} + \xi^4\psi = \epsilon\psi \tag{5.19}$$

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

$$a = \left(\frac{\hbar^2}{m\mu}\right)^{1/6} \qquad \bar{E} = \left(\frac{\hbar^4\mu}{m^2}\right)^{1/3} \tag{5.20}$$

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

Lab 6

Animating the Wave Equation

Note: this lab is long, so you may not finish it in one session. But Lab 7 is shorter, so you can continue into the next lab period and still get both of them finished.

Labs 4 and 5 should have seemed pretty familiar, since they handled the wave equation by Fourier analysis, turning the partial differential equation into a set of ordinary differential equations, as you learned in Mathematical Physics. [1] But separating the variables and expanding in orthogonal functions is not the only way to solve partial differential equations, and in fact in many situations this technique is awkward, ineffective, or both. In this lab we will study another way of solving partial differential equations using a spatial grid and stepping forward in time. And as an added attraction, this method automatically supplies a beautiful animation of the solution. I will only show you one of several algorithms of this type that can be used on wave equations, so this is just an introduction to a larger subject. The method I will show you here is called *staggered leapfrog*; it is the simplest good method that I know.

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

$$\frac{\partial^2 y}{\partial t^2} - c^2 \frac{\partial^2 y}{\partial x^2} = 0 \tag{6.1}$$

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

$$y(0,t) = f_{\text{left}}(t) \quad ; \quad y(L,t) = f_{\text{right}}(t) \tag{6.2}$$

or of *Neumann* type (derivatives specified):

$$\frac{\partial y}{\partial x}(0) = g_{\text{left}}(t) \quad ; \quad \frac{\partial y}{\partial x}(L) = g_{\text{right}}(t) \tag{6.3}$$

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

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

$$y(x, t=0) = y_0(x) \quad ; \quad \frac{\partial y(x,t)}{\partial t}|_{t=0} = v_0(x) \tag{6.4}$$

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

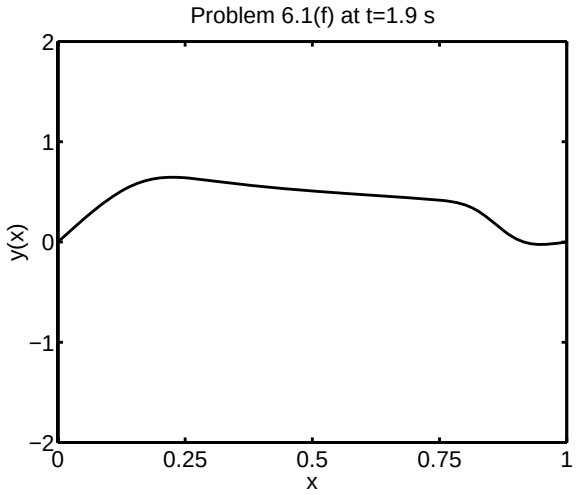


Figure 6.1 A wave on a string with variable linear mass density

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

To numerically solve the classical wave equation via staggered leapfrog we approximate both the time and spatial derivatives with centered finite differences. In the notation below spatial position is indicated by a subscript j, referring to grid points x_j, while position in time is indicated by superscripts n, referring to time steps t_n so that $y(x_j, t_n) = y_j^n$. The time steps and the grid spacings are assumed to be uniform with time step called τ and grid spacing called h.

$$\frac{\partial^2 y}{\partial t^2} \approx \frac{y_j^{n+1} - 2y_j^n + y_j^{n-1}}{\tau^2} \tag{6.5}$$

$$\frac{\partial^2 y}{\partial x^2} \approx \frac{y_{j+1}^n - 2y_j^n + y_{j-1}^n}{h^2} \tag{6.6}$$

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

$$y_j^{n+1} = 2y_j^n - y_j^{n-1} + \frac{c^2\tau^2}{h^2}\left(y_{j+1}^n - 2y_j^n + y_{j-1}^n\right) \tag{6.7}$$

6.1. Use Maple to derive this algorithm from the approximate second derivative formulas.

Hint: If you choose your Maple variable names carefully you can generate expressions that can be copied in Maple and pasted into Matlab to produce code that only requires minor editing.

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

This algorithm can only be used at interior spatial grid points, however, because the $j+1$ or $j-1$ indices reach beyond the grid at the first and last grid points. The behavior of the solution at these two end points is determined by the boundary conditions. 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 the calculation starts previous time array `yold` is not available. As you will see a little later, we will have to make creative use of the initial velocity condition to find a starting value for `yold`.

Let's worry about the boundary conditions at the ends first. 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. (6.7), then we use the boundary conditions to find the new values of y_1^{n+1} and y_{N+2}^{n+1},, like this:

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

$$\frac{y_1^{n+1}+y_2^{n+1}}{2} = f_{\text{left}}(t_{n+1}) \quad \Rightarrow \quad y_1^{n+1} = -y_2^{n+1} + 2f_{\text{left}}(t_{n+1}) \tag{6.8}$$

$$\frac{y_{N+2}^{n+1}+y_{N+1}^{n+1}}{2} = f_{\text{right}}(t_{n+1}) \quad \Rightarrow \quad y_{N+2}^{n+1} = -y_{N+1}^{n+1} + 2f_{\text{right}}(t_{n+1}) \tag{6.9}$$

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

$$\frac{y_2^{n+1}-y_1^{n+1}}{h} = g_{\text{left}}(t_{n+1}) \quad \Rightarrow \quad y_1^{n+1} = y_2^{n+1} - hg_{\text{left}}(t_{n+1}) \tag{6.10}$$

$$\frac{y_{N+2}^{n+1}-y_{N+1}^{n+1}}{h} = g_{\text{right}}(t_{n+1}) \quad \Rightarrow \quad y_{N+2}^{n+1} = y_{N+1}^{n+1} + hg_{\text{right}}(t_{n+1}) \tag{6.11}$$

To implement these conditions we just use the appropriate equation from Eqs. (6.8)-(6.11) 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. (6.7).) 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 two simultaneous equations that determine 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:

$$\frac{y_j^1 - y_j^{-1}}{2\tau} = v_0(x_j) \tag{6.12}$$

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

$$y_j^1 = 2y_j^0 - y_j^{-1} + \frac{c^2\tau^2}{h^2}\left(y_{j+1}^0 - 2y_j^0 + y_{j-1}^0\right) \tag{6.13}$$

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

$$y_j^{-1} = y_j^0 - v_0(x_j)\tau + \frac{c^2\tau^2}{2h^2}\left(y_{j+1}^0 - 2y_j^0 + y_{j-1}^0\right) \tag{6.14}$$

6.2. 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 with no initial velocity starts with an initial upward displacement localized near the center of the string. Once you have it running, do the numerical studies listed after the template.

stlf.m

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

% build a cell-centered grid with N=200 and L=1;
.
.
.
% define the initial displacement and velocity vs. x
y = exp(-(x-L/2).^2*160/L^2)-exp(-(0-L/2).^2*160/L^2);
vy = 0*x; % multiplying x by zero makes an array of zeroes
         % of the same length as x

subplot(2,1,1)
plot(x,y) % plot the initial conditions
xlabel('x');ylabel('y(x,0)');title('Initial Displacement')
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 (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
   time=n*tau;  % compute the time
   % Use leapfrog and the boundary conditions to load ynew with y at
   % the next time step using y and yold, i.e., ynew(2:N+1)=...
   % Be sure to use colon commands so it will run fast.
.
.
.
   %update yold and y
   yold=y;y=ynew;

% make plots every skip time steps
   if mod(n,skip)==0
      plot(x,y,'b-')
      xlabel('x');ylabel('y');
% Print a top label with the time included
      topline=sprintf('Staggered Leapfrog Wave: time= %g',time)
      title(topline)
      axis([min(x) max(x) -2 2]);
      pause(.1)
   end
end
```

6.3. (a) 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 $\tau > h/c$ the algorithm blows up spectacularly. This failure is called a *numerical instability* and we will be trying to avoid it all semester. This limit is called the *Courant-Friedrichs-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.

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

(c) Change the initial conditions from initial displacement with zero velocity to initial velocity with zero displacement. (Use an initial Gaussian velocity pulse just like the displacement pulse you used earlier.) 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 does the physics right.

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

$$\frac{\partial^2 y}{\partial t^2} + \gamma \frac{\partial y}{\partial t} - c^2 \frac{\partial^2 y}{\partial x^2} = 0 \tag{6.15}$$

with c constant. Use fixed ends. Use Maple to derive a damping-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 problem 6.3(c) to verify that the waves damp away. You will need to run for about 25 s and you will want to use a big skip factor so that you don't have to wait forever for the run to finish. Include some code to record the maximum value of $y(x)$ over the entire grid as a function of time and then plot it as a function of time at the end of the run so that you can see the decay caused by γ. The decay of a simple harmonic oscillator is exponential, with amplitude proportional to $e^{-\gamma t/2}$. Scale this time decay function properly and lay it over your maximum y plot to see if it fits. Can you explain why the fit is as good as it is? (Hint: think about doing this problem via separation of variables.)

6.5. Apply a driving force to the damped wave equation using the form we studied in Problem 4.1(a). Note: to do this problem you will need to abandon the simple ideas of setting all coefficients to 1 and just making interesting pictures. This involves the real physics of waves on a real guitar string, so go back to Lab 4 and look up T, μ, and L, find the wave speed, and carefully include the driving force in Eq. (4.1). Also add damping, as you did in part(f), so that the string can settle down to a steady oscillation after a while. In particular, be careful about where μ goes in the staggered leapfrog algorithm.

There is one simplification however: since the string will start from rest you don't need to worry about finding y_{old}. Just set $y = 0$ and $y_{\text{old}} = 0$ and enter the time-stepping loop.

Use the parameter values given in Problem 4.1(a) and choose a damping constant γ that is the proper size to make the system settle down after 20 or 30 bounces of the string. (You will have to think about the value of $\omega = 400$ that you will be using and about your results from problem 6.4 to decide which value of γ to use to make this happen.)

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. You may find yourself looking at a flat-line plot with no oscillation at all. If this happens look at the vertical scale of your plot and remember that we are doing real physics here. If your vertical scale goes from -1 to 1, you are expecting an amplitude of 1 meter on your string.

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.

Lab 7

Staggered Leapfrog in Two Dimensions

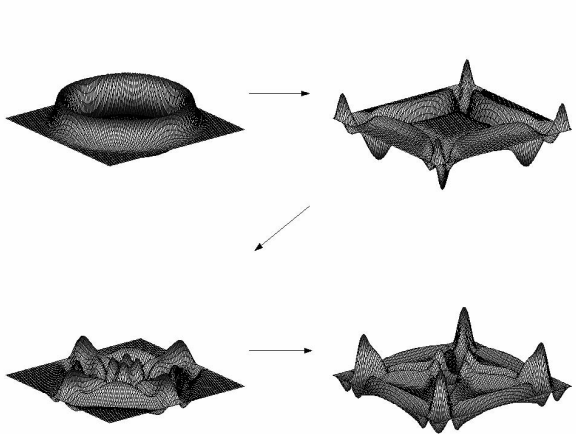


Figure 7.1 A wave on a rubber sheet

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

$$\mu \frac{\partial^2 z}{\partial t^2} = \sigma \left(\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} \right) \tag{7.1}$$

In this equation μ is the surface mass density of the sheet, with units of mass/area. The quantity σ is the surface tension, which has rather odd units. By inspecting the equation above you can find that σ has units of force/length, which doesn't seem right for a surface. But it is, in fact, correct as you can see by performing the following thought experiment. Cut a slit of length L in the rubber sheet and think about how much force you have to exert to pull the lips of this slit together. Now imagine doubling L; doesn't it seem that you should have to pull twice as hard to close the slit? Well, if it doesn't, it should; the formula for this closing force is given by σL, which defines the meaning of σ.

7.1. (a) 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 use a cell-edge square grid with the edge-values pinned to zero.) Use a displacement initial condition that is a Gaussian pulse with zero velocity:

$$z(x, y, 0) = e^{-5(x^2+y^2)} \tag{7.2}$$

and be sure to use double-colon commands so that your script will run fast, i.e.,

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

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

Run the simulation long enough that you see the effect of repeated reflections from the edges. Choose $\sigma = 2$ N/m and $\mu = 0.3$ kg/m^2.

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

$$\tau < f\frac{h}{c} \tag{7.3}$$

Determine the value of the constant f by numerical experimentation. (Try various values of τ and discover where the boundary is between numerical stability and instability.)

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

(d) Finally, change the initial conditions so that the sheet is initially flat but with the initial velocity given by the Gaussian pulse of Eq. (7.2). In one dimension when you pulse the system like this the string at the point of application of the pulse moves up and stays up until the reflection comes back from the ends of the system. (We did this experiment with the slinky in Lab 6.) Does the same thing happen in the middle of the sheet when you apply this initial velocity pulse? Answer this question by looking at your plot of $z(0, 0, t)$. You should find that the two-dimensional wave equation behaves very differently from the one-dimensional wave equation.

Lab 8

The Diffusion, or Heat, Equation

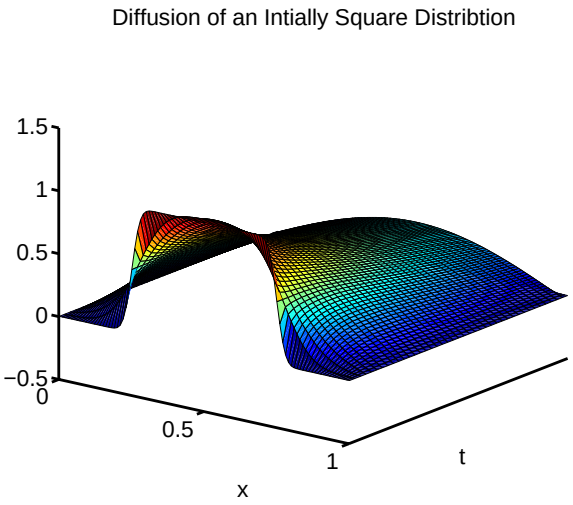


Figure 8.1 Diffusion with an initial square temperature distribution and zero temperature end boundary conditions.

Now we're going to attack the diffusion equation [1]

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2} \quad . \tag{8.1}$$

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 we only need to know the initial distribution of T. This means that the trouble with the initial distribution of $\partial T/\partial t$ that we encountered with the wave equation is avoided. But in spite of this simplification, the diffusion equation is actually more difficult to solve numerically than the wave equation.

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

$$T(x) = T_0 e^{-(x-L/2)^2/\sigma^2} \tag{8.2}$$

decays according to the formula

$$T(x,t) = \frac{T_0}{\sqrt{1+4Dt/\sigma^2}} e^{-(x-L/2)^2/(\sigma^2+4Dt)} \tag{8.3}$$

by showing that this expression satisfies the diffusion equation Eq. (8.1) and the initial condition. (It doesn't satisfy finite boundary conditions, however; it is zero at $\pm\infty$.) Use Maple.

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

(b) Use separation of variables to find how an initial temperature distribution $T_n(x,0) = T_0 \sin(n\pi x/L)$ 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 $f_n(t)$ for each integer n. Do long wavelengths or short wavelengths decay more quickly?

(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 by finding the time it takes for the $t = 0$ width of the Gaussian (σ) to increase to 2σ (look in the exponential in Eq. (8.3).)

Now let's 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 obtain an algorithm that is similar to leapfrog then we would have

$$\frac{T_j^{n+1} - T_j^{n-1}}{2\tau} = \frac{D}{h^2}\left(T_{j+1}^n - 2T_j^n + T_{j-1}^n\right) \tag{8.4}$$

$$T_j^{n+1} = T_j^{n-1} + \frac{2D\tau}{h^2}\left(T_{j+1}^n - 2T_j^n + T_{j-1}^n\right) \tag{8.5}$$

There is a problem starting this algorithm because of the need to have T one time step in the past (T_j^{n-1}), but even if we work around this problem this algorithm turns out to be worthless because no matter how small a time step τ we choose, we encounter the same kind of instability that plagues staggered leapfrog (infinite zig-zags). Such an algorithm is called *unconditionally unstable*, and is an invitation to keep looking. This must have been a nasty surprise for the pioneers of numerical analysis who first encountered it. It seems almost intuitively obvious that making an algorithm more accurate is better, but in this case the increased accuracy achieved by using a centered time derivative leads to numerical instability.

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

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

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

You might expect this algorithm to have problems since the left side of Eq. (8.6) is centered at time $t_{n+\frac{1}{2}}$, but the right side is centered at time t_n. This problem makes the algorithm inaccurate, but it turns out that it is stable if τ is small enough. In the next lab we'll learn how to get a stable algorithm with both sides of the equation centered on the same time, but for now let's use this inaccurate (but stable) method.[2]

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

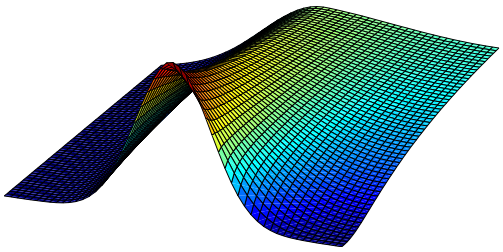


Figure 8.2 Diffusion of an initial Gaussian with D on the right much larger than D on the left. Time runs into the page and position x is across the front.

8.2. (a) Modify your staggered leapfrog program (keep the cell-center grid) to use this algorithm to solve the diffusion equation on the interval $[0, L]$ with initial distribution

$$T(x, 0) = \sin(\pi x/L) \tag{8.8}$$

and boundary conditions $T(0) = T(L) = 0$. Use $D = 2$, $L = 3$, and $N = 20$. This algorithm has a CFL condition on the time step τ of the form

$$\tau \leq C\frac{h^2}{D} \tag{8.9}$$

Determine the value of C by numerical experimentation.

To test the accuracy of your numerical solution compare to your answer to problem 8.1(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 from 20 to 40 and then to 80.

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

$$T(x, 0) = e^{-40(x/L-1/2)^2} \tag{8.10}$$

Use two different kinds of boundary conditions:

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

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

Explain what these boundary conditions mean by thinking about a watermelon that is warmer in the middle than at the edge. Tell physically how you would impose both of these boundary conditions on the watermelon and explain what the temperature history of the watermelon has to do with your plots of $T(x)$ vs. time.

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

$$D(x) = D_0\frac{x^2 + L/5}{(L/5)} \tag{8.11}$$

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

$$\frac{\partial T}{\partial t} = \frac{\partial}{\partial x}\left(D(x)\frac{\partial T}{\partial x}\right) \tag{8.12}$$

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

Lab 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 present, and perhaps past, values of $y(x,t)$ at that grid point and at neighboring grid points to find the future $y(x,t)$ at j. Methods like this, that depend in a simple way on present and past values to predict future values, are said to be *explicit* and are easy to code. They are also often numerically unstable. For example, as you saw in the previous lab, 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 script take forever to run, which is usually intolerable.

Implicit methods, by contrast, are harder to implement, but have much better stability properties. The reason they are harder is that they assume that you already know the future. To give you a better feel for what "implicit" means, let's study the simple first-order differential equation

$$\dot{y} = -\gamma y \tag{9.1}$$

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

$$\frac{y_{n+1} - y_n}{\tau} = -\gamma y_n \ . \tag{9.2}$$

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

(b) Notice that the left side of Eq. (9.2) is centered on time $t_{n+\frac{1}{2}}$ but the right side is centered on t_n. Let's center the the right-hand side at time $t_{n+\frac{1}{2}}$ by using an average of the advanced and current values of y,

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

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

(c) Now Modify Euler's method by making it *fully implicit* by using y_{n+1} in place of y_n on the right side of Eq. (9.2) (this makes both sides of the equation reach into the future). This method is no more accurate than Euler's method for small time steps, but it is much more stable. Show by numerical experimentation in a modified script that this fully implicit method damps away very quickly when τ is large. Extra damping is usually a feature of fully implicit algorithms.

Now let's look at the diffusion equation again, and see how implicit methods can help us. Just to make things more interesting we'll let the diffusion coefficient be a function of x:

$$\frac{\partial T}{\partial t} = \frac{\partial}{\partial x}\left(D(x)\frac{\partial T}{\partial x}\right) \tag{9.3}$$

We begin by finite differencing the right side, taking care to handle the spatial dependence of D. Rather than expanding the nested derivatives in Eq. (9.3) let's difference it in place on the grid. In the equation below $D_{j\pm\frac{1}{2}} = D(x_j \pm h/2)$.

$$\frac{\partial T_j}{\partial t} = \frac{1}{h^2}\left(D_{j+\frac{1}{2}}(T_{j+1} - T_j) - D_{j-\frac{1}{2}}(T_j - T_{j-1})\right) \tag{9.4}$$

9.2. Draw a picture of a grid showing x_{j-1}, $x_{j-\frac{1}{2}}$, x_j, $x_{j+\frac{1}{2}}$, and x_{j+1} and show that this form is centered properly.

Hint: Find centered difference expressions for $D(x)\frac{\partial T}{\partial x}$ at $x_{j-\frac{1}{2}}$ and $x_{j+\frac{1}{2}}$. Then use these two expressions to find a centered difference formula for the entire expression at x_j.

Now we take care of the time derivative by doing something similar to problem 9.1(b): we take a forward time derivative on the left, putting that side of the equation at time level $n+\frac{1}{2}$. To put the right side at the same time level (so that the algorithm will be second-order accurate), we replace each occurrence of T on the right-hand side by the average

$$T^{n+\frac{1}{2}} = \frac{T^{n+1} + T^n}{2} \tag{9.5}$$

like this:

$$\frac{T_j^{n+1} - T_j^n}{\tau} = \frac{1}{2h^2}\left(D_{j+\frac{1}{2}}(T_{j+1}^{n+1} - T_j^{n+1} + T_{j+1}^n - T_j^n) - D_{j-\frac{1}{2}}(T_j^{n+1} - T_{j-1}^{n+1} + T_j^n - T_{j-1}^n)\right) \tag{9.6}$$

If you look carefully at this equation you will see that there is a problem: how are we supposed to solve for T_j^{n+1}? The future values T^{n+1} are all over the place, and they involve three neighboring grid points (T_{j-1}^{n+1}, T_j^{n+1}, and T_{j+1}^{n+1}), so we can't just solve in a simple way for T_j^{n+1}. This is an example of why implicit methods are harder than explicit methods.

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

$$-D_{j-\frac{1}{2}}T_{j-1}^{n+1} + \left(\frac{2h^2}{\tau} + D_{j+\frac{1}{2}} + D_{j-\frac{1}{2}}\right)T_j^{n+1} - D_{j+\frac{1}{2}}T_{j+1}^{n+1} = D_{j-\frac{1}{2}}T_{j-1}^n + \left(\frac{2h^2}{\tau} - D_{j+\frac{1}{2}} - D_{j-\frac{1}{2}}\right)T_j^n + D_{j+\frac{1}{2}}T_{j+1}^n \tag{9.7}$$

I know this looks ugly, but it really isn't so bad. To solve for T_j^{n+1} we just need to solve a linear system, as we did in Lab 3 on two-point boundary value problems. When a system of equations must be solved to find the future values, we say that the method is *implicit*. This particular implicit method is called the *Crank-Nicholson algorithm*.

To see more clearly what we are doing, and to make the algorithm a bit more efficient, let's define a matrix **A** to describe the left side of Eq. (9.7) and another matrix **B** to describe the right side, like this:

$$\mathbf{A}T^{n+1} = \mathbf{B}T^n \tag{9.8}$$

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

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

$$A_{j,j-1} = -D_{j-\frac{1}{2}} \quad ; \quad A_{j,j} = \frac{2h^2}{\tau} + (D_{j+\frac{1}{2}} + D_{j-\frac{1}{2}}) \quad ; \quad A_{j,j+1} = -D_{j+\frac{1}{2}} \tag{9.9}$$

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

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

$$B_{j,j-1} = D_{j-\frac{1}{2}} \quad ; \quad B_{j,j} = \frac{2h^2}{\tau} - (D_{j+\frac{1}{2}} + D_{j-1/2}) \quad ; \quad B_{j,j+1} = D_{j+\frac{1}{2}} \tag{9.10}$$

Once the boundary conditions are added to these matrices, Eq. (9.8) can easily be solved symbolically to find T^{n+1}

$$T^{n+1} = \mathbf{A}^{-1}\mathbf{B}T^n \ . \tag{9.11}$$

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

(i) Load the matrices **A** and **B** as given in Eq. (9.9) and Eq. (9.10) above for all of the rows except the first and last. As usual, the first and last rows involve the boundary conditions. Usually it is a little easier to handle the boundary conditions if we plan to do the linear solve in two steps, like this:

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

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

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

Notice that we can just load the top and bottom rows of **B** with zeros, creating a right-hand-side vector r with zeros in the top and bottom positions. The top and bottom rows of **A** can then be loaded with the appropriate terms to enforce the desired boundary conditions on T^{n+1}, and the top and bottom positions of r can be loaded as required just before the linear solve, as indicated above. (An example of how this works will be given in the Crank-Nicholson script below.) Note that if the diffusion coefficient $D(x)$ doesn't change with time you can load **A** and **B** just once before the time loop starts.

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

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

$$A(1,1) = \frac{1}{2} \quad A(1,2) = \frac{1}{2} \quad B(1,1) = 0 \quad r(1) = 1 \tag{9.12}$$

$$A(N{+}2, N{+}2) = \frac{1}{2} \quad A(N{+}1, N{+}2) = \frac{1}{2} \quad B(N{+}2, N{+}2) = 0 \quad r(N{+}2) = 5 \tag{9.13}$$

so that the equations for the top and bottom rows are

$$\frac{T_1 + T_2}{2} = r_1 \qquad \frac{T_{N+1} + T_{N+2}}{2} = r_{N+2} \tag{9.14}$$

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

The time advance would then look like this:

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

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

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

Here is a Matlab program that implements the Crank-Nicholson algorithm.

cranknicholson.m

```
%begin cranknicholson.m

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;
% Turn x into a column vector.
x=x';

%  Load the diffusion coefficient array (just 1 for now--
```

```
%  later you will load it with something more interesting.)
%  Make it a column vector.

D=ones(N+2,1);

% Load Dm with average values D(j-1/2) and Dp with D(j+1/2)
% Make them column vectors.

Dm=zeros(N+2,1);Dp=zeros(N+2,1);
Dm(2:N+1)=.5*(D(2:N+1)+D(1:N)); % average j and j-1
Dp(2:N+1)=.5*(D(2:N+1)+D(3:N+2)); % average j and j+1

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

% Find 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.
const = 2*h^2 / tau;

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

% load A and B at interior points using
% colon commands

for j=2:N+1
   A(j,j-1)= -Dm(j);
   A(j,j)  = const + (Dm(j)+Dp(j));
   A(j,j+1)= -Dp(j);

   B(j,j-1)=  Dm(j);
```

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

% now load the boundary conditions for the
% case T(0)=0 and T(L)=0

A(1,1)=0.5;A(1,2)=0.5;B(1,1)=0.;
A(N+2,N+1)=0.5;A(N+2,N+2)=0.5;B(N+2,N+2)=0;


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

t=mtime*tau; % define the time

r=B*T;
r(1)=0;r(N+2)=0; % set the right side vector r for T(0)=0 and T(L)=0
T=A\r; % do the linear solve

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

end
```

9.3. (a) Test `cranknicholson.m` by running it with $D(x) = 2$ and an initial temperature given by $T(x) = \sin(\pi x/L)$. As you found in Lab 8, the exact solution for this distribution is:

$$T(x,t) = \sin(\pi x/L)\exp(-\pi^2 Dt/L^2) \tag{9.15}$$

Try various values of τ and see how it compares with the exact solution. Verify that when the time step is too large the solution is inaccurate, but still stable. To do the checks at large time step you will need to use a long run time and not skip any steps in the plotting, i.e., use a skip factor of 1.

Also study the accuracy of this algorithm by using various values of the cell number N and the time step τ. For each pair of choices run for 5 seconds and find the maximum difference between the exact and numerical solutions. You should find that the time step τ hardly matters at all. The number of cells N is the main thing to worry about if you want high accuracy in diffusion problems.

(b) Modify the Crank-Nicholson script to use boundary conditions $\partial T/\partial x = 0$ at the ends. Run with the same initial condition as in part (a) (which does not satisfy

these boundary conditions) and watch what happens. Use a "microscope" on the plots early in time to see what happens in the first few grid points during the first few time steps.

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

Lab 10

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 $V(x)$ (assuming that it is a function of a single spatial variable x): [1]

$$i\hbar\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\psi}{\partial x^2} + V(x)\psi \tag{10.1}$$

Note that except for the presence of the imaginary unit i, this is very much like the diffusion equation. In fact, a good way to solve it is with the Crank-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.

Write a script to solve the time-dependent Schrödinger equation using Crank-Nicholson as discussed in Lab 9. 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 script to do the following problems.

10.1. 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:

$$\psi(-L) = 0 \quad ; \quad \psi(L) = 0 \tag{10.2}$$

Use a cell-edge grid:

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

so we have N grid points.

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

$$\psi(x,0) = \frac{1}{\sqrt{\sigma\sqrt{\pi}}}e^{ipx/\hbar}e^{-x^2/(2\sigma^2)} \tag{10.3}$$

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 script with $N = 200$ and watch the particle (wave packet) bounce back and forth in the

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

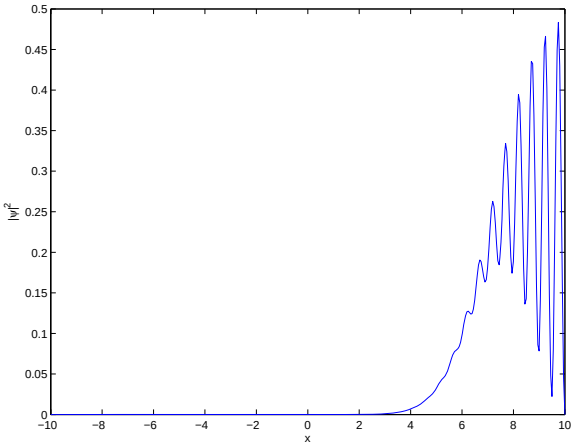


Figure 10.1 The probability density $|\psi|^2$ of a particle moving to the right interfering with itself as it reflects from a wall.

well. Plot the real part of ψ as an animation to visualize the spatial oscillation of the wave packet, then plot an animation of $\psi^*\psi$ so that you can visualize the probability distribution of the particle.

(b) Verify by doing a numerical integral that $\psi(x, 0)$ in the formula given above is properly normalized. Then run the script and check that it stays properly normalized, even though the wave function is bouncing and spreading within the well. (Since you are on a cell-edge grid you will need to do the integrals with `trapz`.)

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

$$\langle x \rangle = \int_{-L}^{L} \psi^*(x,t)\ x\ \psi(x,t) dx \tag{10.4}$$

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

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

Now we will allow the pulse to collide with the barrier and study what happens. Classically, the answer is simple: if the particle has a kinetic energy less than V_0 it will be unable to get over the barrier, but if its kinetic energy is greater than V_0 it will slow down as it passes over the barrier, then resume its normal speed in the region beyond the barrier.

To see how this picture is modified in quantum mechanics we must first compute the energy of our pulse. The quantum mechanical formula for the expectation value of the

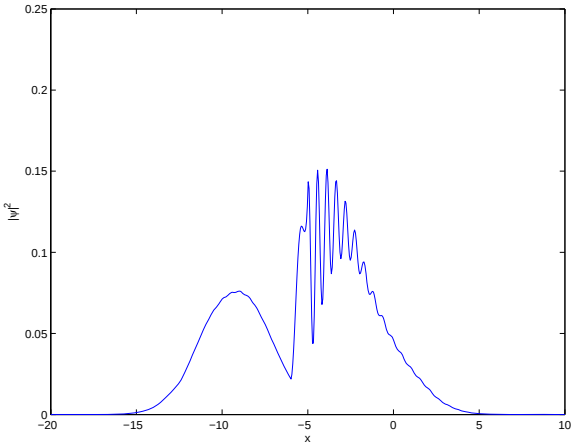


Figure 10.2 A particle moving to the left is incident on a potential barrier. Part is transmitted and part interferes with itself as it is reflected.

energy is

$$\langle E\rangle = \int_{\infty}^{\infty} \psi^* H\psi dx \tag{10.5}$$

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

$$H\psi = -\frac{\hbar^2}{2m}\frac{\partial^2\psi}{\partial x^2} + V(x)\psi(x) \tag{10.6}$$

In our case $\psi(x)$ is essentially zero at the location of the potential barrier, so we may take $V(x) = 0$ in the integral.

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

$$\langle E\rangle = \frac{p^2}{2m} + \frac{\hbar^2}{4m\sigma^2} \approx 20 \tag{10.7}$$

(b) Modify your script from Problem 10.1 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.) (Warning: don't just add V_0 to the main diagonal and call it good. Re-derive the Crank-Nicholson algorithm from scratch and see how $V(x)$ enters.)

Run your script several times, varying the height V_0 from less than your pulse energy to more than your pulse energy. Can the particle get past a barrier that is higher than its kinetic energy? The answer is yes, and this effect is called tunneling. Also vary the width of the hill to see its effect on tunneling.

Lab 11

Poisson's Equation I

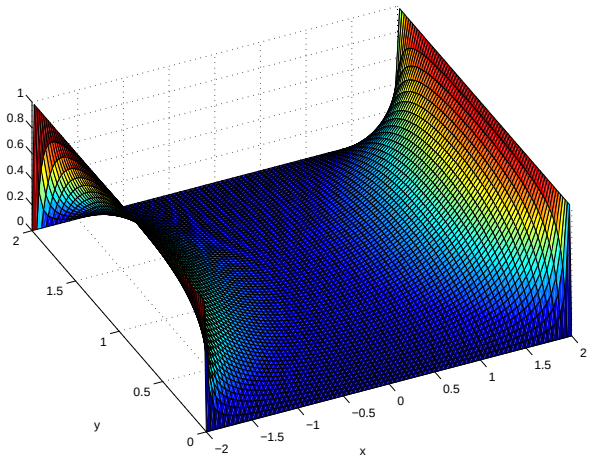


Figure 11.1 The electrostatic potential $V(x, y)$ with two sides grounded and two sides at constant potential.

Consider Poisson's equation on a two-dimensional rectangle $[0, a] \times [0, b]$ with x corresponding to a and y corresponding to b. [1]

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

(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 obtain a grid-based version of Poisson's equation. In the finite-difference version of the equation shown below I have used the notation

$$V(x, y) = V_{i,j} \quad ; \quad V(x + \Delta x, y) = V_{i+1,j} \quad ; \quad V(x - \Delta x, y) = V_{i-1,j} \tag{11.2}$$

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

$$V(x, y + \Delta y) = V_{i,j+1} \quad ; \quad V(x, y - \Delta y) = V_{i,j-1} \tag{11.3}$$

Here's the new Poisson equation:

$$\frac{V_{i+1,j} - 2V_{i,j} + V_{i-1,j}}{\Delta x^2} + \frac{V_{i,j+1} - 2V_{i,j} + V_{i,j-1}}{\Delta y^2} = -\frac{\rho}{\epsilon_0} \tag{11.4}$$

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 10,000×10,000 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 (many gigabytes) 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 obtain a version of Poisson's equation that helps us develop a less memory-intensive way to solve it, let's solve Eq. (11.4) for $V_{i,j}$, like this:

$$V_{i,j} = \left(\frac{V_{i+1,j} + V_{i-1,j}}{\Delta x^2} + \frac{V_{i,j+1} + V_{i,j-1}}{\Delta y^2} + \frac{\rho}{\epsilon_0} \right) / \left(\frac{2}{\Delta x^2} + \frac{2}{\Delta y^2} \right) \tag{11.5}$$

With the equation in this form we could just iterate over and over by doing the following. (1) Choose an initial guess for the interior values of $V_{i,j}$. (2) Use this initial guess to evaluate the right-hand side of Eq. (11.5) 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. [2]

11.1. Derive Eq. (11.5) from the Eq. (11.4) (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 (as we did in Lab 1):

$$x = e^{-x} \tag{11.6}$$

If we iterate on this equation like this:

$$x_{n+1} = e^{-x_n} \tag{11.7}$$

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

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. (11.6) and expanding to first order in δ then gives

$$\bar{x} + \delta_{n+1} = e^{-\bar{x}} - e^{-\bar{x}}\delta_n \qquad \Rightarrow \qquad \delta_{n+1} = -e^{-\bar{x}}\delta_n \tag{11.8}$$

So when we are close to the solution the error becomes smaller every iteration by the factor $-e^{-\bar{x}}$. Since $\bar{x}$ is positive, $e^{-\bar{x}}$ is less than 1, and the algorithm converges. When iteration works it is not a miracle—it is just a consequence of having this expansion technique result in an error multiplier that is less than 1 in magnitude.

11.2. Write a short Matlab script to solve the equation $x = e^{-x}$ by iteration and verify that it converges. Then try solving this same equation the other way round: $x = -\ln x$ and show that the algorithm doesn't converge. Then use the $\bar{x}$-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. (11.5) can be written in matrix form as

$$V_{n+1} = \mathbf{L}V_n + r \tag{11.9}$$

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. (11.5) 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,

$$\delta_{n+1} = \mathbf{L}\delta_n \tag{11.10}$$

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 $\mathbf{L}$ to have magnitudes less than 1. So we can now restate the original miracle, "Iteration on Eq. (11.5) converges," in this way: "All of the eigenvalues of the matrix $\mathbf{L}$ on the right-hand side of Eq. (11.5) 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. (11.9) 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. (11.5), they used values of V as they became available during the iteration. (This means that the right side of Eq. (11.5) 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

$$x_{n+1} = e^{-x_n} \tag{11.11}$$

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

$$x_{n+1} = \omega e^{-x_n} + (1-\omega)x_n \tag{11.12}$$

where ω is a number which is yet to be determined. (Take a minute and verify that even though this equation looks quite different, it is still solved by $x = e^{-x}$.)

Now linearize as before to find how the error changes as we iterate:

$$x_n = \bar{x} + \delta_n \quad \Rightarrow \quad \delta_{n+1} = (-\omega e^{-\bar{x}} + 1 - \omega)\delta_n \tag{11.13}$$

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

11.3. Write a Matlab script that accepts a value of ω and runs the iteration in Eq. (11.12). Experiment with various values of ω until you find one that does the best job of accelerating the convergence of the iteration. You should find that the best ω is near 0.64, but it won't give convergence in one step. See if you can figure out why not. (Think about the approximations involved in obtaining Eq. (11.13).)

As you can see from Eq. (11.13), 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:

$$V_{n+1} = \omega\left(\mathbf{L}V_n + r\right) + (1-\omega)V_n \tag{11.14}$$

then play with ω until we achieve almost instantaneous convergence.

Sadly, this doesn't quite work. The problem is that in solving for $(N_x - 1)(N_y - 1)$ 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:

$$V_{i,j} = \omega\left(\frac{V_{i+1,j} + V_{i-1,j}}{\Delta x^2} + \frac{V_{i,j+1} + V_{i,j-1}}{\Delta y^2} + \frac{\rho_0}{\epsilon_0}\right) \Big/ \left(\frac{2}{\Delta x^2} + \frac{2}{\Delta y^2}\right) + (1-\omega)V_{i,j} \tag{11.15}$$

or (written in terms of Rhs, the right-hand side of Eq. (11.5)):

$$V_{i,j} = \omega Rhs + (1-\omega)V_{i,j} \tag{11.16}$$

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

$$\rho = \frac{\Delta y^2 \cos(\pi/N_x) + \Delta x^2 \cos(\pi/N_y)}{\Delta x^2 + \Delta y^2} \tag{11.17}$$

$$\omega = \frac{2}{1+\sqrt{1-\rho^2}} \tag{11.18}$$

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. (11.18) 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

$$\epsilon = \left| \frac{Lhs - Rhs}{V_{scale}} \right| \tag{11.19}$$

where Lhs is the left-hand side of Eq. (11.5) and Rhs is its right-hand side. Because this equation is just an algebraic rearrangement of our finite-difference approximation to Poisson's equation, ϵ can only be small when Poisson's equation is satisfied. (Note the use of absolute value; can you explain why it is important to use it? Also note that this error is to be computed at all of the interior grid points. Be sure to find the maximum error on the grid so that you only quit when the solution is conserved throughout the grid.)

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

For more details, and for other ways of improving the algorithm, see *Numerical Recipes*, Chapter 19.

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.

sor.m

```
%begin 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,
        % Eq. (11.5)
        rhs =  . . .

         % calculate the relative error, left side - right side
         % of Eq. (11.5)
         err(n)= . . .

         % SOR algorithm [Eq. (11.16)], to
         % 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
```

11.3. (a) 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. (11.18) is the best one to use.

(b) Using the optimum value of ω in each case, run `sor.m` for $N_x = N_y = 20,\ 40,\ 80,$ and 160. See if you can find a rough power law formula for how long it takes to push the error below 10^{-5}, i.e., guess that Run Time $\approx AN_x^p$, and find A and p. The `cputime` command will help you with the timing.

Lab 12

Poisson's Equation II

12.1. 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

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = \frac{-\rho}{\epsilon_0} \quad + \text{ Boundary Conditions} \tag{12.1}$$

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

$$\frac{\partial^2 y}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 y}{\partial t^2} = 0 \quad + \text{ Boundary Conditions} \tag{12.2}$$

for the wave displacement $y(x, t)$; and the thermal diffusion equation:

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2} \quad + \text{ Boundary Conditions} \tag{12.3}$$

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:

$$\text{ellipse}: \quad \frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \tag{12.4}$$

$$\text{hyperbola}: \quad \frac{x^2}{a^2} - \frac{y^2}{b^2} = 1 \tag{12.5}$$

$$\text{parabola}: \quad y = ax^2 \tag{12.6}$$

Compare these equations with the classical PDE's above and make sure you can use their resemblances to each other to remember the following rules: Poisson's equation is elliptic, the wave equation is hyperbolic, and the diffusion equation is parabolic. These names are important because each different type of equation requires a different type of algorithm and boundary conditions. Fortunately, because you are physicists and have developed some intuition about the physics of these three partial differential equations, you can remember the proper boundary conditions by thinking about physical examples instead of memorizing theorems. And in case you haven't developed this level of intuition, here is a brief review of the matter.

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

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

Parabolic boundary conditions are similar to hyperbolic ones, but with one difference. Think about a thermally-conducting bar with its ends held at fixed temperatures. Once again, surrounding-type boundary conditions are inappropriate because we don't want to specify the future. So as in the hyperbolic case, we can specify conditions at the ends of the bar, but we also want to give initial conditions at $t = 0$. For thermal diffusion we specify the initial temperature $T(x, 0)$, but that's all we need; the "velocity" $\partial T/\partial t$ is determined by Eq. (12.3), 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 make the boundary conditions work. It's important to understand what the appropriate boundary conditions are.

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

$$i\hbar\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\psi}{\partial x^2} + V\psi \tag{12.7}$$

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,

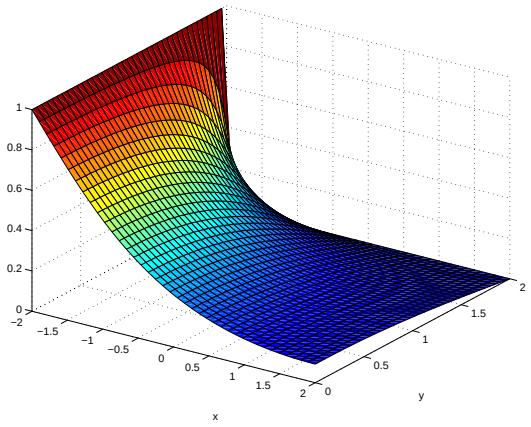


Figure 12.1 The potential $V(x, y)$ with zero-derivative boundary conditions on two sides (Problem 12.2(b).)

like the wave equation with damping?

$$\frac{\partial^2 y}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 y}{\partial t^2} - \frac{1}{D}\frac{\partial y}{\partial t} = 0 \tag{12.8}$$

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

12.2. (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 **E** and how to compute **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 script runs slower on this problem. See if you can make it run a little faster by experimenting with the value of ω that you use. (Again, changing the boundary conditions can change the eigenvalues of the operator.) 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.

12.3. (a) Modify `sor.m` to solve the problem of an infinitely long rectangular pipe of x-width 0.2 m and y-height 0.4 m 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.

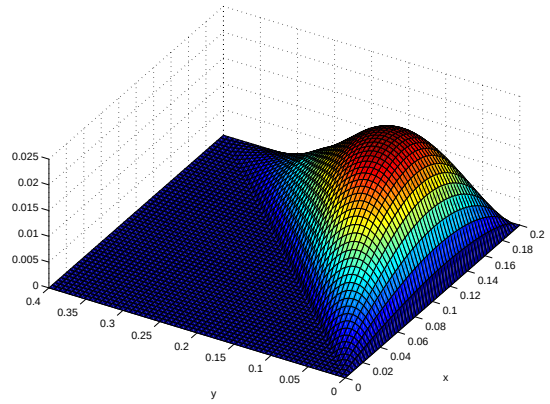


Figure 12.2 The potential $V(x, y)$ with constant charge density on a triangular region grounded at its edges (Problem 12.3.)

(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`, and make sure that the `quiver` command is followed by the command `axis equal` so that the x and y axes have the same scale.

12.4. Study electrostatic shielding by going back to the boundary conditions of Problem 12.2(a), while grounding some points in the interior of the full computation region to build an approximation to a grounded cage. Allow some holes in your cage so you can see how fields leak in. 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.

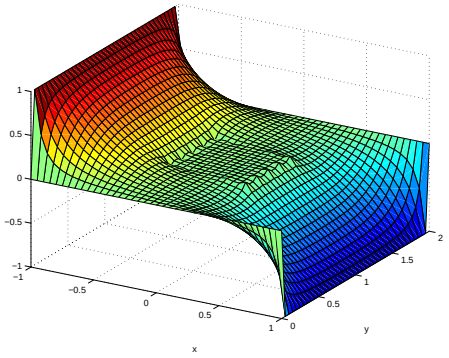


Figure 12.3 An electrostatic "cage" formed by grounding some interior points. (Problem 12.4.)

Lab 13

Gas Dynamics I

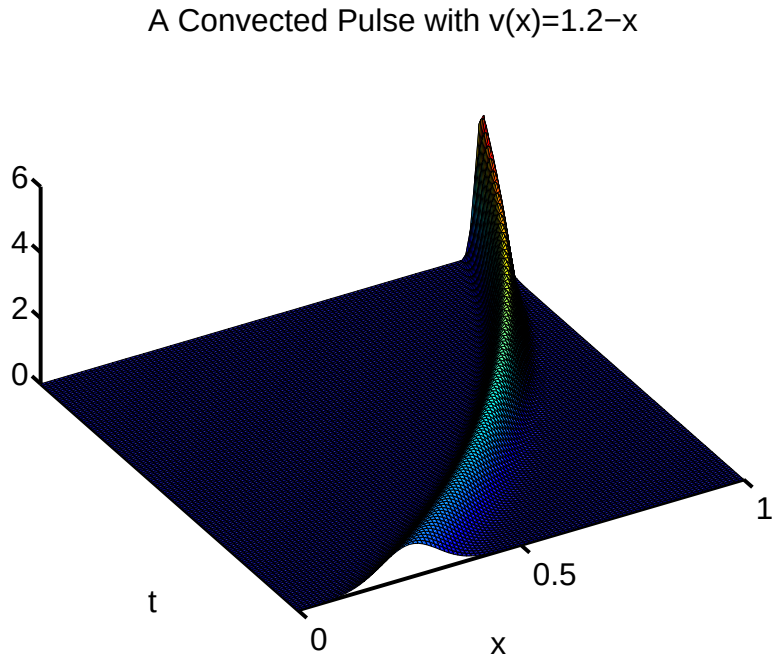


Figure 13.1 A pulse is convected across a region in which the convection velocity $v(x)$ is decreasing. Note that the pulse narrows and grows, conserving mass.

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

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

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

13.1. (a) Roughly verify the statement above by expanding the spatial derivative in Eq. (13.1) to put the equation in the form

$$\frac{\partial \rho}{\partial t} + v\frac{\partial \rho}{\partial x} = -\rho\frac{\partial v}{\partial x} \tag{13.2}$$

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

(c) Now suppose that the initial distribution of density is $\rho(x,0) = \rho_0 =$ const but that the velocity distribution is an "ideal explosion", with $v = 0$ at $x = 0$ and velocity increasing linearly away from 0 like this: $v(x) = v_0 x/a$ (v doesn't vary with time.) Show that the solution of Eq. (13.1) is now given by $\rho(x,t) = \rho_0 e^{-\gamma t}$ and determine the value of the decay constant γ. (You may be concerned that even though ρ is constant in x initially, it may not be later. But I have given you a very special $v(x)$, namely the only one that keeps an initially constant density constant forever. So just assume that ρ doesn't depend on x, then show that your solution using this assumption satisfies the equation and the boundary conditions.)

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

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

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

Conservation of energy

$$\frac{\partial T}{\partial t} + v\frac{\partial T}{\partial x} = -(\gamma - 1)T\frac{\partial v}{\partial x} + D_T\frac{\partial^2 T}{\partial x^2} \tag{13.3}$$

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

$$D_T = \frac{(\gamma - 1)M\kappa}{k_B\rho} \tag{13.4}$$

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

13.2. 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 p involving ρ and T.

Newton's second law

$$\frac{\partial v}{\partial t} + v\frac{\partial v}{\partial x} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \frac{4\mu}{3\rho}\frac{\partial^2 v}{\partial x^2} \tag{13.5}$$

with the pressure p given by the ideal gas law:

$$p = \frac{k_B}{M}\rho T \tag{13.6}$$

and with viscosity included (μ is the coefficient of viscosity.) 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, as well as the force of internal friction called viscosity (tar has high viscosity, water has medium viscosity, and air has almost none.)

The reason for the $1/\rho$ term on the right is that if we multiply by ρ to move it to the left side of the equation we obtain mass times acceleration (or at least mass density times acceleration.)

You may be unconvinced that the left side of Eq. (13.5) is acceleration. To become convinced, read the following and do the exercise:

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 $\partial v/\partial t = 0$.

13.3. To see what's missing, let's force ourselves to ride along with the flow by writing $v = v(x(t), t)$, where $x(t)$ is the position of the moving piece of honey. Carefully use the rules of calculus to evaluate dv/dt and derive the acceleration formula on the left-hand side of Eq. (13.5).

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:

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho v}{\partial x} = 0 \tag{13.7}$$

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

The apparent simplicity of this equation is deceptive; it is one of the most difficult equations to solve numerically in all of computational physics because stable methods tend to be inaccurate and accurate methods tend either to be unstable, or non-conservative (as time runs mass spontaneously disappears), or unphysical (mass density and/or pressure become negative.) In the following problem we will try an algorithm that is unstable,

another that is stable but inaccurate, and finally one that is both stable and conservative, but only works well if the solution doesn't become too steep. (Warning: we are talking about gas dynamics here, so shock waves routinely show up as solutions. Numerical methods that properly handle shocks are much more difficult than the ones I will show you here.)

Before we continue I need to tell you about the boundary conditions on Eq. (13.7). This is a convection equation, meaning that if you stand at a point in the flow, the solution at your location arrives (is convected to you) from further "upwind". This has a strong effect on the boundary conditions. Suppose, for instance, that the flow field $v(x)$ is always positive, meaning that the wind is blowing to the right. At the left-hand boundary it makes sense to specify ρ because somebody might be feeding density in at that point so that it can be convected across the grid. But at the right boundary it makes no sense at all to specify a boundary condition because when the solution arrives there we just want to let the wind blow it away. (An exception to this rule occurs if $v = 0$ at the boundary. In this case there is no wind to blow the solution from anywhere and it would be appropriate to specify a boundary condition.)

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

$$\frac{\rho_j^{m+1} - \rho_j^m}{\tau} + \frac{1}{2h}\left(\rho_{j+1}^m v_{j+1} - \rho_{j-1}^m v_{j-1}\right) = 0 \tag{13.8}$$

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

$$\rho(x, 0) = 1 + e^{-200(x/L - 1/2)^2} \tag{13.9}$$

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

$$v(x) = v_0 \tag{13.10}$$

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

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

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

$$\rho(L, t) = 2\rho(L - h, t) - \rho(L - 2h, t) \quad \text{or} \quad \rho_N = 2\rho_{N-1} - \rho_{N-2} \tag{13.11}$$

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.

13.5. Now let's try another method, known as 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 13.4 and use the extrapolated boundary condition that just lets the pulse leave.

The idea of the Lax-Wendroff algorithm is to use a Taylor series in time to obtain a second-order accurate method.

$$\rho(x,t+\tau) = \rho(x,t) + \tau\frac{\partial\rho}{\partial t} + \frac{\tau^2}{2}\frac{\partial^2\rho}{\partial t^2} \tag{13.12}$$

(a) Use this Taylor expansion and Eq. (13.7) to derive the following expression (assume that v is not a function of time):

$$\rho(x,t+\tau) = \rho(x,t) - \tau\frac{\partial\rho v}{\partial x} + \frac{\tau^2}{2}\frac{\partial}{\partial x}\left(v\frac{\partial\rho v}{\partial x}\right) \quad . \tag{13.13}$$

If you stare at this equation for a minute you will see that a diffusion-like term has showed up. (To see this subtract $\rho(x,t)$ from both sides and divide by τ, then interpret the new left-hand side as a time derivative. Make sure you do this little exercise or the second sentence below will make no sense.) 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 13.4. 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 Eq. (13.13) assuming that $v(x) = v_0 =$ const, as in 13.4 to find the Lax-Wendroff algorithm:

$$\rho_j^{m+1} = \rho_j^m - \frac{v_0\tau}{2h}[\rho_{j+1}^m - \rho_{j-1}^m] + \frac{v_0^2\tau^2}{2h^2}[\rho_{j+1}^m - 2\rho_j^m + \rho_{j-1}^m] \tag{13.14}$$

Change your script from 13.4 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 a time step just below the stability constraint.

Warning: do not run with $\tau = h/v_0$. If you do you will conclude that this algorithm is perfect, which is only true for this one choice of time step.) This problem is caused by the diffusive term in the algorithm, but since this diffusive term is the reason that this algorithm is not unstable like the one in 13.4, I suppose we should be grateful.

13.6. 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. (13.7) like this:

$$\frac{\rho_j^{m+1} - \rho_j^m}{\tau} + \frac{1}{4h}\left(\rho_{j+1}^{m+1}v_{j+1} - \rho_{j-1}^{m+1}v_{j-1} + \rho_{j+1}^m v_{j+1} - \rho_{j-1}^m v_{j-1}\right) = 0 \tag{13.15}$$

(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:

$$\mathbf{A}\rho^{m+1} = \mathbf{B}\rho^{m} \tag{13.16}$$

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

(b) Work out how to implement the boundary conditions ($\rho(0,t) = 1$ and $\rho(L,t)$ is just allowed to leave) by properly defining the top and bottom rows of the matrices $\mathbf{A}$ and $\mathbf{B}$. (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.

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

$$v(x) = 1.2 - x/L \tag{13.17}$$

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

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

$$\rho(x,0) \;= \begin{cases} 1.0 & \text{if} \quad 0 \le x \le L/2 \\ \\ 0 & \text{otherwise} \end{cases} \tag{13.18}$$

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.

(f) For comparison, try this same initial condition in your Lax-Wendroff script from Problem 13.5.

Lab 14

Solitons: Korteweg-deVries Equation

At the Lagoon amusement park in the town of Farmington, just north of Salt Lake City, Utah, there is a water ride called the Log Flume. It is a standard, old-fashioned water ride where people sit in a 6-seater boat shaped like a log which slowly travels along a fiberglass trough through some scenery, then is pulled up a ramp to an upper level. The slow ascent is followed by a rapid slide down into the trough below, which splashes the passengers a bit, after which the log slowly makes its way back to the loading area. But you can see something remarkable happen as you wait your turn to ride if you watch what happens to the water in the trough when the log splashes down. A large water wave is pushed ahead of the log, as expected, but instead of gradually dying away, as you might think a single pulse should in a dispersive system like surface waves on water, the pulse lives on and on. It rounds the corner ahead of the log that created it, enters the area where logs are waiting to be loaded, pushes each log up and down in turn, then heads out into the scenery beyond, still maintaining its shape.

This odd wave is called a "soliton", or "solitary wave", and it is an interesting feature of non-linear dynamics that has been widely studied in the last 30 years, or so. The simplest mathematical equation which produces a soliton is the Korteweg-deVries equation:

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

In the first two terms of this equation you can see the convective behavior we studied in Lab 13, but the last term, with its rather odd third derivative, is something new.

We will be studying this equation in this laboratory, and we will begin by using Crank-Nicholson to finite difference it on a grid so that we can explore its behavior by numerical experimentation. The first step is to define a grid, and since we want to be able to see the waves travel for a long time we will copy the trough at Lagoon and make our computing region be a closed loop. We can do this by choosing an interval from $x = 0$ to $x = L$, as usual, but then we will make the system be periodic by declaring that $x = 0$ and $x = L$ are actually the same point, as would be the case in a circular trough. We will subdivide this region into N subintervals and let $h = L/N$ and $x_j = (j-1)h$, so that the grid is cell-edge. Normally such a cell-edge grid would have $N+1$ points, but ours doesn't because the last point ($j = (N+1)$) is just a repeat of the first point: $x_{N+1} = x_1$, because our system is periodic.

Before we can use Crank-Nicholson we will have to decide what to do about the non-linear convection term $v\partial v/\partial x$. We will assume that the leading v is known somehow by designating it as v^* and decide later how to properly estimate its value. So this term will be taken to be of the form $v^*\partial v/\partial x$. We now do the usual Crank-Nicholson differencing, with one twist: we will evaluate each term in the equation not only at time level $n+1/2$, but also at spatial location $j+1/2$, at the center of each subinterval. This means that we will be using a cell-edge grid, but that the spatial finite differences will be cell centered. This makes the first and third spatial derivatives be a little more compact, as you will see below.

OK, here is how each term in the equation is finite-differenced in this scheme:

$$\frac{\partial v}{\partial t} = \frac{1}{2\tau}\left(v_j^{n+1} + v_{j+1}^{n+1} - v_j^n - v_{j+1}^n\right) \tag{14.2}$$

$$v\frac{\partial v}{\partial x} = \frac{v^*}{2h}\left(v_{j+1}^{n+1} - v_j^{n+1} + v_{j+1}^n - v_j^n\right) \tag{14.3}$$

$$\alpha\frac{\partial^3 v}{\partial x^3} = \frac{\alpha}{2h^3}\left(v_{j+2}^{n+1} - 3v_{j+1}^{n+1} + 3v_j^{n+1} - v_{j-1}^{n+1} + v_{j+2}^n - 3v_{j+1}^n + 3v_j^n - v_{j-1}^n\right) \tag{14.4}$$

There are now two problems to handle with respect to the quantity v^*. The first is that the derivative in Eq. (14.3) is centered in space at $j+1/2$, so we would like whatever we use for v^* to be centered there as well. The second problem is that the derivative in Eq. (14.3) is centered in time at $n+1/2$. We will solve the first problem now, because it is easy, and deal with the more difficult second problem later. To center v^* in space we simply replace the simple term v^* in Eq. (14.3) by its spatial average:

$$v^* \to \frac{v_{j+1}^* + v_j^*}{2} \tag{14.5}$$

Each of these approximations is now substituted into Eq. (14.1), the v^{n+1} terms are gathered on the left side of the equation and the v^n terms are gathered on the right, and then the coefficients of the matrices **A** and **B** are read off to put the equation in the form

$$\mathbf{A}v^{n+1} = \mathbf{B}v^n \tag{14.6}$$

in the usual Crank-Nicholson way. (There is still a problem with the way the nonlinear term involving v^* has been handled. Be patient, we will take care of it shortly.)

If we denote the four nonzero elements of **A** and **B** like this:

$$A_{j,j-1} = a_{--} \quad A_{j,j} = a_- \quad A_{j,j+1} = a_+ \quad A_{j,j+2} = a_{++} \tag{14.7}$$

$$B_{j,j-1} = b_{--} \quad B_{j,j} = b_- \quad B_{j,j+1} = b_+ \quad B_{j,j+2} = b_{++} \tag{14.8}$$

then the matrix coefficients turn out to be

$$a_{--} = -\frac{\alpha}{2h^3} \quad a_- = \frac{1}{2\tau} + \frac{3\alpha}{2h^3} - \frac{(v_-^* + v_+^*)}{4h} \quad a_+ = \frac{1}{2\tau} - \frac{3\alpha}{2h^3} + \frac{(v_-^* + v_+^*)}{4h} \quad a_{++} = \frac{\alpha}{2h^3} \tag{14.9}$$

$$b_{--} = \frac{\alpha}{2h^3} \quad b_- = \frac{1}{2\tau} - \frac{3\alpha}{2h^3} + \frac{(v_-^* + v_+^*)}{4h} \quad b_+ = \frac{1}{2\tau} + \frac{3\alpha}{2h^3} - \frac{(v_-^* + v_+^*)}{4h} \quad b_{++} = -\frac{\alpha}{2h^3} \tag{14.10}$$

where $v_- = v_j$ and where $v_+ = v_{j+1}$, the points on the left and the right of the j^{th} subinterval.

14.1. Use Maple to derive the formulas in Eqs. (14.9) and (14.10) for the a and b coefficients using the finite-difference approximations to the three terms in the Korteweg-deVries equation given in Eqs. (14.2)-(14.4).

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

$$\mathbf{A} = \begin{bmatrix} a_- & a_+ & a_{++} & 0 & 0 & ... & 0 & a_{--} \\ a_{--} & a_- & a_+ & a_{++} & 0 & ... & 0 & 0 \\ 0 & a_{--} & a_- & a_+ & a_{++} & ... & 0 & 0 \\ . & . & . & . & ... & . & . & . \\ 0 & ... & 0 & 0 & a_{--} & a_- & a_+ & a_{++} \\ a_{++} & 0 & ... & 0 & 0 & a_{--} & a_- & a_+ \\ a_+ & a_{++} & 0 & 0 & ... & 0 & a_{--} & a_- \end{bmatrix} \tag{14.11}$$

$$\mathbf{B} = \begin{bmatrix} b_- & b_+ & b_{++} & 0 & 0 & ... & 0 & b_{--} \\ b_{--} & b_- & b_+ & b_{++} & 0 & ... & 0 & 0 \\ 0 & b_{--} & b_- & b_+ & b_{++} & ... & 0 & 0 \\ . & . & . & . & ... & . & . & . \\ 0 & ... & 0 & 0 & b_{--} & b_- & b_+ & b_{++} \\ b_{++} & 0 & ... & 0 & 0 & b_{--} & b_- & b_+ \\ b_+ & b_{++} & 0 & 0 & ... & 0 & b_{--} & b_- \end{bmatrix} \tag{14.12}$$

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

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

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

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

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

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

$$v^* = v^n \tag{14.13}$$

$$v^p = \mathbf{A}(v^n)^{-1}\left[\mathbf{B}(v^n)v^n\right] \tag{14.14}$$

$$v^* = \frac{v^p + v^n}{2} \tag{14.15}$$

$$v^{n+1} = \mathbf{A}(v^*)^{-1}\left[\mathbf{B}(v^*)v^n\right] \tag{14.16}$$

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

kdv.m

```
% Korteweg-deVries equation on a periodic cell-centered grid using
% Crank-Nicholson

clear;close;

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

alpha=input(' Enter alpha - ')

vmax=input(' Enter initial amplitude of v - ')

% load an initial Gaussian centered on the computing region
v=vmax*exp(-(x-.5*L).^2);

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

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

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

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

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

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

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

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

for n=1:Nsteps

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

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

    % do the predictor solve
    r=B*v;
    vp=A\r;

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

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

    if rem(n-1,iskip)==0
      plot(x,v,'LineWidth',2.5)
      xlabel('x');ylabel('v')
      pause(.1)
    end

end
```

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

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

14.4. The standard "lore" in the field of solitons is that the moving bump you saw in 14.3 is

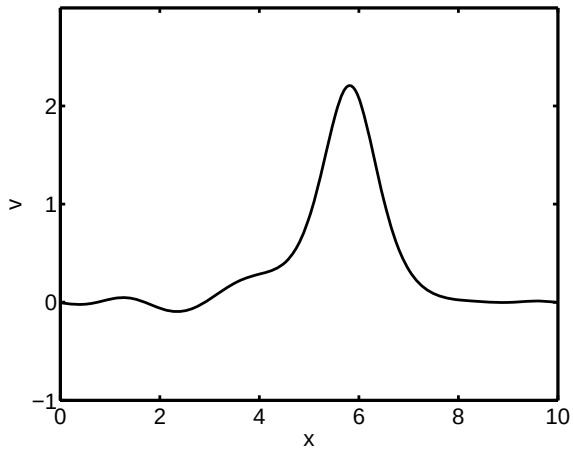


Figure 14.1 A Gaussian pulse after 1 second of propagation by the Korteweg-deVries equation (Problem 14.3).

produced by a competition between the wave spreading caused by the third derivative in the Korteweg-deVries equation and the wave steepening caused by the $v\partial v/\partial x$ term. Let's run `kdv.m` in such a way that we can see the effect of each of these terms separately.

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

(b) Non-linear wave-steepening dominates: Run `kdv.m` with $\alpha = 0.01$, $v_{\max} = 2$, $\tau = 0.01$, and $t_{\text{final}} = 3$. (When your solution develops short wavelength wiggles this is an invitation to use a smaller time step. The problem is that the predictor-correction algorithm we used on the nonlinear term is not stable enough, so we have a Courant condition in this problem.)

Now it is the dispersion term that is small and we can see the effect of the non-linear convection term. Where v is large the convection is rapid, but out in front where v is small the convection is slower. This allows the fast peak to catch up with the slow front end, causing wave steepening. (An effect just like this causes ocean waves to steepen and then break at the beach.)

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

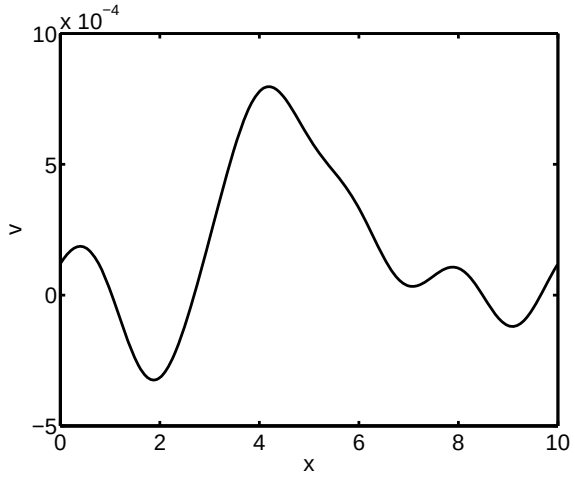


Figure 14.2 Dispersion dominates (Problem 14.4(a).): after 3 seconds of time.

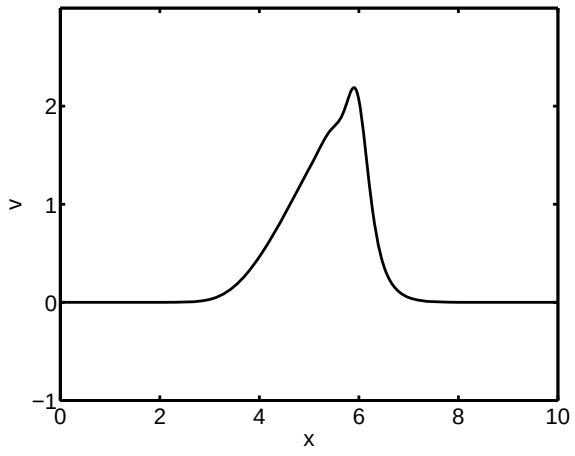


Figure 14.3 Steepening dominates (Problem 14.4(b).): after 0.5 seconds of time.

equation:

$$v(x,t) = \frac{12k^2\alpha}{\cosh^2\left(k(x - x_0 - 4\alpha k^2 t)\right)} \tag{14.17}$$

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

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

(b) Now replace the Gaussian initial condition in `kdv.m` with this pulse shape, using $k = 1.1$, $x_0 = L/2$, and adjusting α so that the height of the initial pulse is exactly equal to 2, so that it matches the Gaussian pulse you ran in 14.3. You should find that this time the pulse does not leave trailing pulses behind, but that it moves without changing shape. It is a perfect soliton.

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

$$c_{\text{soliton}} = 4\alpha k^2 \tag{14.18}$$

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

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

To see what happens keep $\alpha = 0.1$, but modify your code from 14.5 so that you have a soliton pulse with $k = 1.5$ centered at $x = 3L/4$ and another soliton pulse with $k = 2$ centered at $x = L/4$. Run for about 20 seconds and watch how they interact when the fast large amplitude pulse in the back catches up with the slower small amplitude pulse in the front. Is it correct to say that they pass through each other? If not, can you think of another qualitative way to describe their interaction?

Lab 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, in one spatial dimension x. (Note: these are the equations of one-dimensional sound waves in a long tube pointed in the x-direction, assuming that the tube is wide enough that friction with the walls doesn't matter.)

First we have conservation of mass:

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}(\rho v) = 0 \tag{15.1}$$

Here is conservation of energy:

$$\frac{\partial T}{\partial t} + v\frac{\partial T}{\partial x} + (\gamma - 1)T\frac{\partial v}{\partial x} = \frac{(\gamma - 1)M\kappa}{k_B}\frac{1}{\rho}\frac{\partial^2 T}{\partial x^2} \tag{15.2}$$

where $k_B = 1.38 \times 10^{-23}$ J/K is Boltzmann's constant in SI units and where M is the mass of the molecules of the gas ($M = 29 \times 1.67 \times 10^{-27}$ kg for air.)

And here is Newton's second law:

$$\frac{\partial v}{\partial t} + v\frac{\partial v}{\partial x} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \frac{4\mu}{3\rho}\frac{\partial^2 v}{\partial x^2} \tag{15.3}$$

with the pressure p given by the ideal gas law:

$$p = \frac{k_B}{M}\rho T \tag{15.4}$$

For disturbances in air at sea level at 20° C we have temperature $T = 293$ K, mass density $\rho = 1.3$ kg/m^3, adiabatic exponent $\gamma = 1.4$, coefficient of viscosity $\mu = 1.82 \times 10^{-5}$ kg/(m·s), and coefficient of thermal conductivity $\kappa = 0.024$ J/(m·s·K). 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.

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

$$\frac{\partial \rho}{\partial t} + \rho\frac{\partial v}{\partial x} = 0 \quad \text{at} \quad x = 0 \quad \text{and} \quad x = L \tag{15.5}$$

This condition simply says that the density at the ends goes up and down in obedience to the compression or rarefaction produced by the divergence of the velocity.

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 by first taking an approximate step in time of length τ to obtain predicted values ρ_j^*, T_j^*, and v_j^*. To make this possible using the linear techniques we have been using all semester, in this predictor step we will use v_j^m in Eqs. (15.1) and (15.2), and in the thermal diffusion term in Eq. (15.2) we will use ρ_j^m and the just-calculated density ρ_m^* to form a centered time average of the density at $t_{m+1/2}$. With these choices we can use Crank-Nicholson on Eqs. (15.1) and (15.2) to obtain estimates of the future values of density and pressure ρ^* and p^*. Once these predicted values are obtained we can use average values involving them in the velocity equation to obtain a predicted v^*.

In what follows the notation will be simplified by defining the constant transport coefficients

$$F = \frac{(\gamma-1)M\kappa}{k_B} \qquad G = \frac{4\mu}{3} \tag{15.6}$$

Here are the equations above, written so that Crank-Nicholson can be applied to each one in turn to obtain predicted values of ρ^*, T^*, and v^*:

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}\left(\rho v^m\right) = 0 \qquad \text{is solved to obtain} \qquad \rho^* \tag{15.7}$$

$$\frac{\partial T}{\partial t} + v^m\frac{\partial T}{\partial x} + (\gamma-1)T\frac{\partial v^m}{\partial x} = F\frac{2}{(\rho^m+\rho^*)}\frac{\partial^2 T}{\partial x^2} \qquad \text{is solved to obtain} \qquad T^* \tag{15.8}$$

$$\frac{\partial v}{\partial t} + v^m\frac{\partial v}{\partial x} = -\left(\frac{k_B}{M}\right)\frac{2}{(\rho^m+\rho^*)}\frac{\partial}{\partial x}\left[\left(\frac{\rho^m+\rho^*}{2}\right)\left(\frac{T^m+T^*}{2}\right)\right] + G\frac{2}{(\rho^m+\rho^*)}\frac{\partial^2 v}{\partial x^2} \tag{15.9}$$

is solved to obtain v^*.

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 either with starred quantities, or with values at t_{m+1}, as they become available, like this.

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}\left[\rho\left(\frac{v^m+v^*}{2}\right)\right] = 0 \qquad \text{is solved to obtain} \qquad \rho^{m+1} \tag{15.10}$$

$$\frac{\partial T}{\partial t} + \frac{(v^m+v^*)}{2}\frac{\partial T}{\partial x} + (\gamma-1)T\frac{\partial}{\partial x}\left(\frac{v^m+v^*}{2}\right) = F\frac{2}{(\rho^m+\rho^{m+1})}\frac{\partial^2 T}{\partial x^2} \tag{15.11}$$

is solved to obtain T^{m+1}.

$$\frac{\partial v}{\partial t} + \frac{(v^m+v^*)}{2}\frac{\partial v}{\partial x} = \tag{15.12}$$

$$-\left(\frac{k_B}{M}\right)\frac{2}{(\rho^m+\rho^{m+1})}\frac{\partial}{\partial x}\left[\left(\frac{(\rho^m+\rho^{m+1})}{2}\right)\left(\frac{T^m+T^{m+1}}{2}\right)\right] + G\frac{2}{(\rho^m+\rho^{m+1})}\frac{\partial^2 v}{\partial x^2} \tag{15.13}$$

is solved to obtain v^{m+1}.

15.1. (a) Implement this algorithm by modifying your Crank-Nicholson script from Lab 13, taking special care to implement the boundary conditions properly on a cell-centered grid.

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

$$v(x,0) = v_0 e^{-200(x/L-1/2)^2} \tag{15.14}$$

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

(c) With $F = 0$ and $G = 0$ 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 G and watch thermal conduction and viscosity widen the shock and prevent wiggles. Try artificially large values of F and G as well as their actual atmospheric values.

Lab 16

Implicit Methods in 2-Dimensions: Operator Splitting

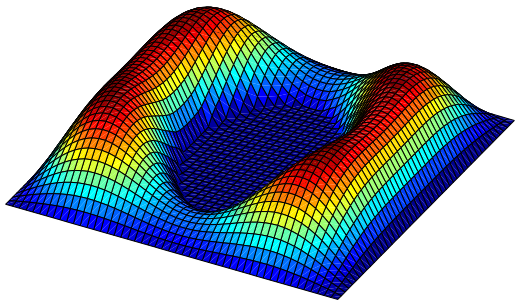


Figure 16.1 Diffusion in 2-dimensions with an elliptical central region set to zero.

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

$$\frac{\partial T}{\partial t} = D\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) \tag{16.1}$$

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

$$\mathcal{L}_x T_{i,j} = \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{h^2} \quad ; \quad \mathcal{L}_y T_{i,j} = \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{h^2} \tag{16.2}$$

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

$$\frac{T^{n+1} - T^n}{\tau} = \frac{D}{2}\left(\mathcal{L}_x T^{n+1} + \mathcal{L}_x T^n + \mathcal{L}_y T^{n+1} + \mathcal{L}_y T^n\right) \tag{16.3}$$

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.

One such way is called *operator splitting*, and was invented by Douglas[1] and by Peaceman and Rachford.[2] The idea is to turn each time step into two half-steps, doing a fully implicit step in x in the first half-step and another one in y in the second half-step. It looks like this:

$$\frac{T^{n+1/2} - T^n}{\tau/2} = D\left(\mathcal{L}_x T^{n+1/2} + \mathcal{L}_y T^n\right) \tag{16.4}$$

$$\frac{T^{n+1} - T^{n+1/2}}{\tau/2} = D\left(\mathcal{L}_x T^{n+1/2} + \mathcal{L}_y T^{n+1}\right) \tag{16.5}$$

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

16.1. Use Maple to eliminate the intermediate variable $T^{n+1/2}$ and show that the algorithm gives

$$\frac{T^{n+1} - T^n}{\tau} = \frac{D}{2}\left(\mathcal{L}_x T^{n+1} + \mathcal{L}_x T^n + \mathcal{L}_y T^{n+1} + \mathcal{L}_y T^n\right) - \frac{D^2\tau^2}{4}\mathcal{L}_x\mathcal{L}_y\left(\frac{T^{n+1} - T^n}{\tau}\right) \tag{16.6}$$

which is just like 2-dimensional Crank-Nicholson (Eq. (16.3)) except that an extra term corresponding to

$$\frac{D^2\tau^2}{4}\frac{\partial^5 T}{\partial x^2 \partial y^2 \partial t} \tag{16.7}$$

has erroneously appeared. But if $T(x, y, t)$ has smooth derivatives in all three variables this error term is second-order small in the time step τ.

We saw in Lab 8 that the accuracy of Crank-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 $D^2\tau^2/\ell^4 \ll 1$, 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.

16.2. 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

$$T(x, y, 0) = \cos\left(\frac{\pi x}{10}\right)\cos\left(\frac{\pi y}{10}\right) \tag{16.8}$$

First solve this problem by hand using separation of variables (let $T(x, y, t) = f(t)T(x, y, 0)$ from above, substitute into the diffusion equation and obtain a simple differential

[1] Douglas, J., Jr., *SIAM J.* , **9**, 42, (1955)

[2] Peaceman, D. W. and Rachford, H. H. , *J. Soc. Ind. Appl. Math.* , **3**, 28, (1955)

equation for $f(t)$.) Then modify your Crank-Nicholson script from Lab 8 to use the operator splitting algorithm described in 16.1. 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.

16.3. Modify your script from Problem 16.2 so that the normal derivative at each boundary vanishes. Also change the initial conditions to something more interesting than those in 16.2; it's your choice.

16.4. Modify your script from Problem 16.2 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.

Index