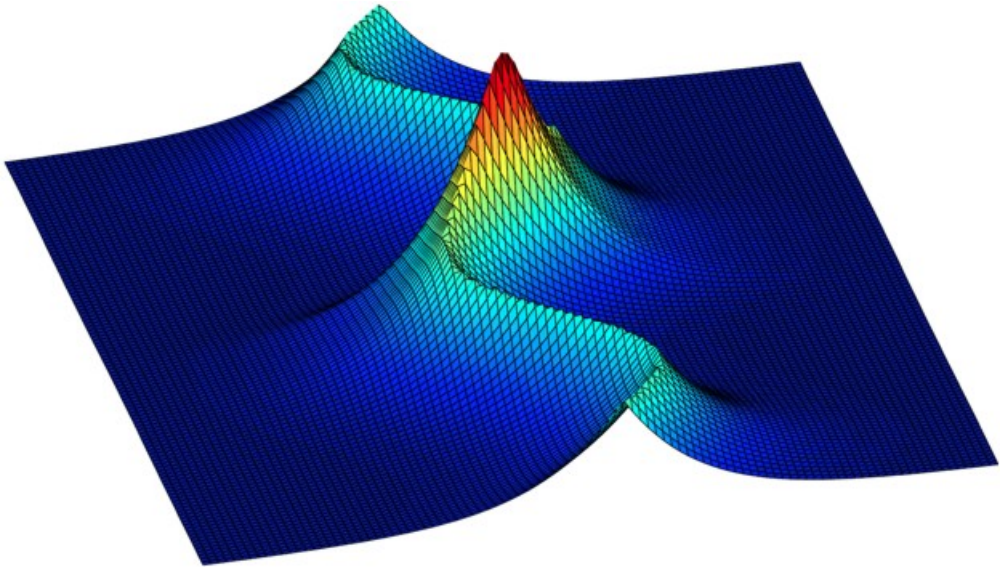
INTRODUCTION TO MATLAB



Ross L. Spencer and Michael Ware

Department of Physics and Astronomy

INTRODUCTION TO MATLAB

Ross L. Spencer and Michael Ware

Department of Physics and Astronomy

Brigham Young University

Last Revised: January 25, 2010

Preface

This is a tutorial to help you get started in Matlab. To find more details see the very helpful book *Mastering MATLAB 7* by Duane Hanselman and Bruce Littlefield. Examples of Matlab code in this pamphlet are in typewriter font `like this`. As you read through the chapters below type and execute in Matlab all of the examples, either at the ≫ command line prompt or in a test program you make called `test.m`. Longer sections of code have boxes around the code. This code can be found in files of the form `ch5ex1a.m` (for Example 5.1a) which you can find on the Physics 330 web page at www.physics.byu.edu.

This booklet can also be used as a reference manual because it is short, it has lots of examples, and it has a table of contents and an index. It is almost true that the basics of Matlab are in chapters 1-9 while physics applications are in chapters 9-17. Please tell us about mistakes and make suggestions to improve it (michael_ware@byu.edu).

Contents

Chapter 1

Running Matlab

1.1 Starting

Get on a department PC or buy Student Matlab[1] for your own machine and start the program.

1.2 It's a Calculator

You can use Matlab as a calculator by typing commands at the $\gg$ prompt, like these. Try them out.

```
1+1
2*3
5/6
exp(-3)
atan2(-1,2)
```

And just like many hand calculators, `ans` in Matlab means the last result calculated.

```
sin(5)
ans
```

Note that Matlab's trig functions are permanently set to radians mode. Note also that the way to enter numbers like 1.23×10^{15} is

```
1.23e15
```

And here's another useful thing. The up-arrow key $\uparrow$ will display previous commands. And when you back up to a previous command that you like, hit Enter and it will execute. Or you can edit it when you get to it (use $\leftarrow$, $\rightarrow$, and Del), then execute it. Try doing this now to re-execute the commands you have already typed.

[1] The student version of Matlab is cheap, powerful, and even has part of Maple in it. You should buy it while you still have a student ID because after you graduate it's very expensive.

1.3 Making Script Files

Most of the work you will do in Matlab will be stored in files called *scripts* containing Matlab commands to be executed over and over again. To make a script, first browse or type in the current directory window on the tool bar to put yourself in the directory where you want to store your Matlab scripts. Then open a new text file in the usual way by clicking on the empty document on the tool bar, or open an old one to be edited. A script can be filled with a sequence of Matlab commands that will be executed from top to bottom just as if you had typed them on the command screen. These files have .m extensions (automatically added in Windows) and can be executed by typing their names (without the .m extension) in the command window. For example, if you create a script called `test.m` you can execute it in the command window by typing `test`. *Do not choose file names that start with numbers*, like `430lab1a.m`. When Matlab receives the start of a number on the command line it thinks a calculation is coming, and since `430lab1a` is not a valid calculation Matlab will give you an error. Also, *do not use a space or a period in the file name.* If you want to separate words in your file name, you can use the underscore character (e.g. my_script.m).

During a session keep this file open so you can modify and debug it. And remember to save the changes you make (Ctrl-s is a quick way) or Matlab in the command window won't know that you have made changes to the script.

Document your code by including lines in it that begin with %, like this. (Note: don't try to execute these lines of code; they just illustrates how to put comments in.)

```
% This is a comment line
```

Or you can put comments at the end of a line of code like this:

```
f=1-exp(-g*t)   % compute the decay fraction
```

You may need to type lines into your script that are too long to see well. To make the code look better you can continue program lines onto successive lines by using the ... syntax. (Note: don't try to execute these lines of code; they just illustrate how to continue long lines.)

```
a=sin(x)*exp(-y)*...
  log(z)+sqrt(b);
```

Finally, nearly always begin your scripts with the `clear` command. This makes sure that you don't have leftover junk active in Matlab that will interfere with your code.

1.4 Running Script Files

Before you can execute a script that you have written and saved, you need to point Matlab's current directory to the place where your script is saved. Matlab displays the current directory in an editable box on the toolbar. You can change the directory by typing in the box or clicking the button next to the current directory box (i.e. the button with the three dots in it).

Once your directory is set to the right place, you can go to the window with the Matlab command prompt ≫ and type the name of your file without the `.m` extension, like this:

```
test
```

and your script will then run.

A convenient alternative for running a script is to use the "Save and Run" shortcut key, `F5`, while in the m-file editor window. This shortcut will save your script file, ask you if you want to switch the current directory to where your script file is located (if it isn't already pointed there), and then run the script for you.

1.5 Pause command

A `pause` command in a script causes execution to stop temporarily. To continue just hit Enter. You can also give it a time argument like this

```
pause(.2)
```

which will cause the script to pause for 0.2 seconds, then continue. And, of course, you can ask for a pause of any number or fractions of seconds. Note, however, that if you choose a really short pause, like 0.001 seconds, the pause will not be so quick. Its length will be determined instead by how fast the computer can run your script.

1.6 Online Help

If you need to find out about something in Matlab you can use `help` or `lookfor` at the $\gg$ prompt. There is also a wealth of information under `Help Desk` in the `Help` menu of Matlab's command window. For example maybe you are wondering about the `atan2` function mentioned in Sec. 1.2. Type

```
help atan2
```

at the $\gg$ prompt to find information about how this form of the inverse tangent function works. Also type

```
help bessel
```

to find out what Matlab's Bessel function routines are called. `Help` will only work if you know exactly how Matlab spells the topic you are looking for.

`Lookfor` is more general. Suppose you wanted to know how Matlab handles elliptic integrals. `help elliptic` is no help, but typing

```
lookfor elliptic
```

will tell you that you should use

```
help ellipke
```

to find what you want.

1.7 Making Matlab Be Quiet

Any line in a script that ends with a semicolon will execute without printing to the screen. Try, for example, these two lines of code in a script or at the ≫ command prompt.

```
a=sin(5);
b=cos(5)
```

Even though the variable `a` didn't print, it is loaded with $\sin(5)$, as you can see by typing this:

```
a
```

1.8 Debugging

When your script fails you will need to look at the data it is working with to see what went wrong. In Matlab this is easy because after you run a script all of the data in that script is available at the Matlab ≫ prompt. So if you need to know the value of `a` in your script just type

```
a
```

and its value will appear on the screen. You can also make plots of your data in the command window. For example, if you have arrays x and y in your script and you want to see what $y(x)$ looks like, just type `plot(x,y)` at the command prompt.

The following Matlab commands are also useful for debugging:

```
who       lists active variables
whos      lists active variables and their sizes
what      lists .m files available in the current directory
```

This information is also available in the workspace window on your screen and described in the next section.

1.9 Arranging the Desktop

A former student, Lance Locey, went directly from this Introduction to Matlab to doing research with it and has found the following way of arranging a few of Matlab's windows on the desktop to be very helpful. (A visual representation of this layout appears in Fig. 1.1.)

1. Make the command window wide and not very tall, stretching across the bottom of the desktop.
2. Open a script editing window (click on the open-a-file icon on the tool bar) and place on the left side, just above the command window.
3. Click on View on the tool bar, then on Workspace to open the workspace window, and place it at the upper right.

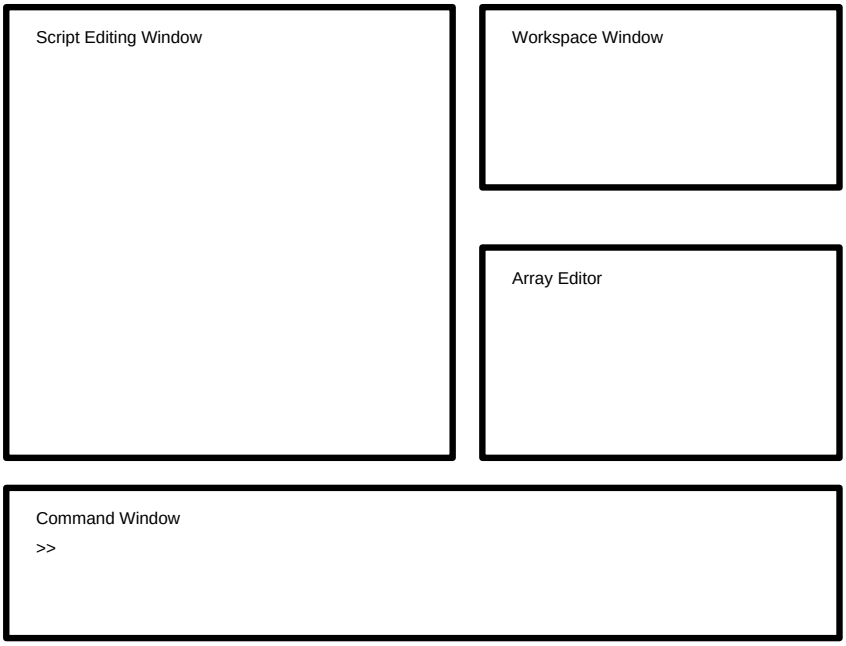


Figure 1.1 A convenient arrangement of the desktop

4. In the command window type `a=1` so that you have a variable in the workspace window, then double click on the yellow name icon for a in the workspace window to make the Array Editor appear. Then place this window just below the workspace window and above the command window, at the left side of the desktop (see the next page.)

With these windows in place you can simultaneously edit your script, watch it run in the command window, and view the values of variables in the array editor. You will find that this is very helpful later when our scripts become more complicated, because the key to debugging is to be able to see the sizes of the arrays that your script is using and the numerical values that they contain. In the next section you will work through a simple example of this procedure.

1.10 Sample Script

To help you see what a script looks like and how to run and debug it, here is a simple one that asks for a small step-size h along the x-axis, then plots the function $f(x) = e^{-x/6}\cos x$ from $x = 0$ to $x = 20$. The script then prints the step-size h and tells you that it is finished. The syntax and the commands used in this script are unfamiliar to you now, but you will learn all about them shortly. Type this script into an M-file called `test.m` using Matlab's editor with the desktop arranged as shown on the previous page. Save it, then run it by typing

```
test
```

in the command window. Or, alternatively, press F5 while the editing window is active and the script will be saved, then executed. Run the sample script below three times using these values of h: 1, 0.1, 0.01. As you run it look at the values of the variables h, x, and f in the workspace window at the upper right of the desktop, and also examine a few values in the array editor just below it so that you understand what the array editor does.

Sample Script

```
clear; % clear all variables from memory
close all; % close all figure windows

h=input(' Enter the step-size h - ') ;
x=0:h:20;  % build an array of points [0,h,2h,...,20]
f=exp(-x/6).*cos(x);  % build the array [f(0),f(h),...f(20)]

plot(x,f)

fprintf(' Plot completed, h = %g \n',h)
```

1.11 Breakpoints and Stepping

When a script doesn't work properly, you need to find out why and then fix it. It is very helpful in this debugging process to watch what the script does as it runs, and to help you do this Matlab comes with two important features: breakpoints and stepping.

To see what a breakpoint does, put the cursor on the `x=0:h:20` line in the sample script above and either click on Breakpoints on the tool bar and select Set/Clear, or press F12. Now press F12 repeatedly and note that the little red dot at the beginning of the line toggles on and off, meaning that F12 is just an on-off switch set set a breakpoint. When the red dot is there it means that a breakpoint has been set, which means that when the script runs it will execute the instructions in the script until it reaches the breakpoint, and then it will stop. Make this happen by pressing F5 and watching the green arrow appear on the line with the breakpoint. Look at the workspace window and note that h has been given a value, but that x has not. This is because the breakpoint stops execution just before the line on which it is set.

Now you can click on the Debug icon on the tool bar to see what to do next, but the most common things to do are to either step through the code executing each line in turn (F10) while watching what happens to your variables in the workspace and array editor windows, or to just continue after the breakpoint to the end (F5.) Take a minute now and use F10 to step through the script while watching what happens in the other windows.

When you write a script to solve some new problem, you should always step through it this way so that you are sure that it is doing what you designed it to do. You will have lots of chances to practice debugging this way as you work through the examples in this book.

Chapter 2

Variables

Symbolic packages like Mathematica and Maple have over 100 different data types; Matlab has just three: the matrix, the string, and the cell array. In both languages variables are not declared, but are defined on the fly as it executes. *Note that variable names in Matlab are case sensitive, so watch your capitalization.* Also: please don't follow the ridiculous trend of making your code more readable by using long variable names with mixed lower- and upper-case letters and underscores sprinkled throughout. Newton's law of gravitation written in this style would be coded this way:

```
Force_of_1_on_2 = G*Mass_of_1*Mass_of_2/Distance_between_1_and_2^2
```

You are asking for an early end to your programming career via repetitive-stress injury if you code like this. Do it this way:

```
F=G*m1*m2/r12^2
```

2.1 Numerical Accuracy

All numbers in Matlab have 15 digits of accuracy. When you display numbers to the screen, like this

```
355/113
```

you may think Matlab only works to 5 significant figures. This is not true; it's just displaying five. If you want to see them all type

```
format long e
```

The four most useful formats to set are

```
format short   (the default)
format long
format long e
format short e
```

Note: `e` stands for exponential notation.

2.2 π

Matlab knows the number π.

```
pi
```

Try displaying π under the control of each of the three formats given in the previous section. Note: you can set `pi` to anything you want, like this, `pi=2`; but please don't.

2.3 Assigning Values to Variables

The assignment command in Matlab is simply the equal sign. For instance,

```
a=20
```

assigns 20 to the variable `a`.

2.4 Matrices

Matlab thinks the number 2 is a 1x1 matrix:

```
N=2
size(N)
```

The array

```
a=[1,2,3,4]
size(a)
```

is a 1x4 matrix (row vectors are built with commas); the array

```
b=[1;2;3;4] size(b)
```

is a 4x1 matrix (column vectors are built with semicolons, or by putting rows on separate lines-see below.)

The matrix

```
c=[1,2,3;4,5,6;7,8,9]
size(c)
```

is a 3x3 matrix (row entries separated by commas, different rows separated by semicolons.) It should come as no surprise that the Mat in Matlab stands for matrix.

When matrices become large the , and ; way of entering them is awkward. A more visual way to type large matrices in a script is to use spaces in place of the commas and to press the Enter key at the end of each row in place of the semicolons, like this:

```
A = [ 1  2  3  4
      5  6  7  8
      9 10 11 12
     13 14 15 16]
```

This makes the matrix look so nice in a script that you probably ought to use it exclusively.

When you want to access the elements of a matrix you use the syntax `A(row,column)`. For example, to get the element of `A` in the 3rd row, 5th column you would use `A(3,5)`. And if you have a matrix or an array and you want to access the last element in a row or column, you can use Matlab's `end` command, like this:

```
c(end)
A(3,end)
```

2.5 Strings

A string is a set of characters, like this

```
s='This is a string'
```

And if you need an apostrophe in your string, repeat a single quote, like this:

```
t='Don''t worry'
```

And if you just want to access part of a string, like the first 7 characters of s (defined above) use

```
s(1:7)
```

Some Matlab commands require options to be selected or set by using strings. Make sure you enclose them in single quotes, as shown above. If you want to know more about how to handle strings type `help strings`.

Chapter 3

Input, Calculating, and Output

3.1 Input

To have a script request and assign a value to the variable N from the keyboard use

```
N=input(' Enter a value for N - ')
```

If you enter a single number, like 2.7, then N will be a scalar variable. If you enter an array, like this: [1,2,3,4,5], then N will be an array. If you enter a matrix, like this: [1,2,3;4,5,6;7,8,9], then N will be a 3x3 matrix. And if you don't want the variable you have entered to echo on the screen, end the input command line with a semicolon.

You can also enter data from a file filled with rows and columns of numbers. Matlab reads the file as if it were a matrix with the first line of the file going into the first row of the matrix. If the file were called `data.fil` and looked like this

```
1 2 3
4 5 6
7 8 9
```

then the Matlab command

```
load data.fil
```

would produce a matrix called `data` filled with the contents of the file.

3.2 Calculating

Matlab only crunches numbers. It doesn't do any symbolic algebra, so it is much less capable than Mathematica or Maple. But because it doesn't have to do hard symbolic stuff, it can handle numbers much faster than Mathematica or Maple can. (Testimonial: "I, Scott Bergeson, do hereby certify that I wrote a data analysis code in Maple that took 25 minutes to run. When I converted the code to Matlab it took 15 seconds.") Here's a brief summary of what it can do with numbers, arrays, and matrices.

3.3 Add and Subtract

Matlab knows how to add and subtract numbers, arrays, and matrices. As long as A and B are two variables of the same size (e.g., both 2×3 matrices), then $A + B$ and $A - B$ will add and subtract them as matrices:

```
A=[1,2,3;4,5,6;7,8,9]
B=[3,2,1;6,4,5;8,7,9]
A+B
A-B
```

3.4 Multiplication

The usual multiplication sign * has special meaning in Matlab. Because everything in Matlab is a matrix, * means matrix multiply. So if A is a 3×3 matrix and B is another 3×3 matrix, then $A * B$ will be their 3×3 product. Similarly, if A is a 3×3 matrix and C is a 3×1 matrix (column vector) then $A * C$ will be a new 3×1 column vector. And if you want to raise A to a power by multiplying it by itself n times, you just use

```
A^n
```

For a language that thinks everything in the world is a matrix, this is perfectly natural. Try

```
A*B
A*[1;2;3]
A^3
```

But there are lots of times when we don't want to do matrix multiplication. Sometimes we want to take two big arrays of numbers and multiply their corresponding elements together, producing another big array of numbers. Because we do this so often (you will see many examples later on) Matlab has a special symbol for this kind of multiplication:

```
.*
```

For instance, the dot multiplication between the arrays `[a,b,c]` and `[d,e,f]` would be the array `[a*d,b*e,c*f]`. And since we might also want to divide two big arrays this way, Matlab also allows the operation

```
./
```

This "dot" form of the division operator divides each element of an array by the corresponding element in another (equally sized) array. If we want to raise each element of an array to a power, we use

```
.^
```

For example, try

```
[1,2,3].*[3,2,1]
[1,2,3]./[3,2,1]
[1,2,3].^2
```

These "dot" operators are very useful in plotting functions and other kinds of signal processing. (You are probably confused about this dot business right now. Be patient. When we start plotting and doing real calculations this will all become clear.)

3.5 Arithmetic with Array Elements

If you just want to do some arithmetic with specific values stored in your arrays, you can just access the individual elements. For instance, if you want to divide the third element of `A` by the second element of `B`, you would just use

```
A(3)/B(2)
```

Note that in this case the things we are dividing are scalars (or 1×1 arrays in Matlab's mind), so Matlab will just treat this like the normal division of two numbers (i.e. we don't have to use the `./` command, although it wouldn't hurt if we did).

3.6 Complex Arithmetic

Matlab works as easily with complex numbers as with real ones. The variable i is the usual imaginary number $i = \sqrt{-1}$, unless you are so foolish as to assign it some other value, like this:

```
i=3
```

If you do this you no longer have access to imaginary numbers, so don't ever do it. If you accidentally do it the command `clear i` *will restore it to its imaginary luster.* By using `i` you can do complex arithmetic, like this

```
z1=1+2i
% or you can multiply by i, like this
z1=1+2*i

z2=2-3i
% add and subtract
z1+z2
z1-z2
% multiply and divide
z1*z2
z1/z2
```

And like everything else in Matlab, complex numbers work as elements of arrays and matrices as well.

When working with complex numbers we quite often want to pick off the real part or the imaginary part of the number, find its complex conjugate, or find its magnitude. Or

perhaps we need to know the angle between the real axis and the complex number in the complex plane. Matlab knows how do all of these

```
z=3+4i
real(z)
imag(z)
conj(z)
abs(z)
angle(z)
```

Matlab also knows how to evaluate many of the functions discussed in the next section with a complex argument. Perhaps you recall Euler's famous formula $e^{ix} = \cos x + i \sin x$? Matlab knows it too.

```
exp(i*pi/4)
```

Matlab knows how to handle complex arguments for all of the trig, exponential, and hyperbolic functions. It can do Bessel functions of complex arguments too.

3.7 Mathematical Functions

Matlab knows all of the standard functions found on scientific calculators and even many of the special functions like Bessel functions. We will give you a list below of a bunch of them, but first you need to know an important feature that they all share. They work just like the "dot" operators discussed in the previous section. This means, for example, that it makes sense to take the sine of an array: the answer is just an array of sine values, e.g.,

```
sin([pi/4,pi/2,pi])= [0.7071 1.0000 0.0000]
```

OK, here's the list of function names that Matlab knows about. You can use online help to find details about how to use them. Notice that the natural log function $\ln x$ is the Matlab function `log(x)`.

cos(x)	sin(x)	tan(x)	sec(x)	csc(x)	cot(x)
acos(x)	asin(x)	atan(x)	atan2(y,x)		
exp(x)	log(x)	[log(x) is ln(x)]	log10(x)	log2(x)	sqrt(x)
cosh(x)	sinh(x)	tanh(x)	sech(x)	csch(x)	coth(x)
acosh(x)	asinh(x)	atanh(x)			
sign(x)	airy(n,x)	besselh(n,x)	besseli(n,x)	besselj(n,x)	besselk(n,x)
bessely(n,x)	beta(x,y)	betainc(x,y,z)	betaln(x,y)	ellipj(x,m)	ellipke(x)
erf(x)	erfc(x)	erfcx(x)	erfinv(x)	gamma(x)	gammainc(x,a)
gammaln (x)	expint(x)	legendre(n,x)	factorial(x)		

3.8 Housekeeping Functions

Here are some functions that don't really do math but are useful in programming.

abs(x)	the absolute value of a number (real or complex)
clc	clears the command window; useful for beautifying printed output
ceil(x)	the nearest integer greater than x
clear	clears all assigned variables
close all	closes all figure windows
close 3	closes figure window 3
fix(x)	the nearest integer to x looking toward zero
fliplr(A)	flip a matrix A, left for right
flipud(A)	flip a matrix A, up for down
floor(x)	the nearest integer less than x
length(a)	the number of elements in a vector
mod(x,y)	the integer remainder of x/y; see online help if x or y are negative
rem(x,y)	the integer remainder of x/y; see online help if x or y are negative
rot90(A)	rotate a matrix A by 90°
round(x)	the nearest integer to x
sign(x)	the sign of x and returns 0 if x=0
size(c)	the dimensions of a matrix

Try `floor([1.5,2.7,-1.5])` to see that these functions operate on arrays and not just on single numbers.

3.9 Output

Now hold on for a second; so far you may have executed most of the example Matlab commands in the command window. From now on it will prepare you better for the more difficult material coming up if you have both a command window and an M-file window open. Put the examples to be run in the M-file (call it `junk.m`), then execute the examples from the command window by typing

```
junk
```

OK, let's learn about output. To display printed results you can use the `fprintf` command. For full information type

```
help fprintf
```

but to get you started, here are some examples. Try them so you know what each one produces. (Here's a hint: the stuff inside the single quotes is a string which will print on the screen; % is where the number you are printing goes; and the stuff after % is a format code. A g means use "whatever" format; if the number is really big or really small, use scientific notation, otherwise just throw 6 significant figures on the screen in a format that looks good. The format 6.2f means use 2 decimal places and fixed-point display with 6 spaces for the number. An e means scientific notation, with the number of decimal places controlled like this: 1.10e.)

```
fprintf('  N =%g \n',500)
```

```
fprintf('  x =%1.12g \n',pi)

fprintf('  x =%1.10e \n',pi)

fprintf('  x =%6.2f \n',pi)

fprintf(' x =%12.8f  y =%12.8f \n',5,exp(5))
```

Note: \n is the command for a new line. If you want all the details on this stuff, look in a C-manual or Chapter 10 of *Mastering Matlab 6*.

This command will also write output to a file. Here is an example from online help that writes a file filled with a table of values of x and $\exp(x)$ from 0 to 1. Note that when using Windows that a different line-feed character must be used with `\r\n` replacing `\n` (see the `fprintf` below.)

Note: the example in the box below is available on the Physics 330 course website, as are all of the examples labeled in this way in this booklet.

Example 3.9a (ch3ex9a.m)

```
% Example 3.8a (Physics 330)
%
%*********************************************************
% build an array of x-values from 0 to 1 in steps of 0.1
% (using the colon command discussed in the next section
% Its syntax is x=xstart:dx:xend.)
%*********************************************************

x=0:.1:1;

% check the length of the x array

N=length(x)

% build a 2xN matrix with a row of x and a row of exp(x)

y=[x;exp(x)];

%*********************************************************
% Open a file in which to write the matrix - fid is a unit
% number to be used later when we write the file and close it.
% There is nothing special about the name fid - fxd works too,
% and, in fact, any variable is OK.
%*********************************************************

fid=fopen('file1.txt','w')

%*********************************************************
% write the matrix to the file - note that it will be in the
% current Matlab directory.  Type cd to see where you are.
% Matlab writes the file as two columns instead of two rows.
```

```
%*********************************************************

fprintf(fid,'%6.2f  %12.8f \r\n',y)

% close the file

st=fclose(fid);
```

After you try this, open `file1.txt`, look at the way the numbers appear in it, and compare them to the format commands `%6.2f` and `%12.8f` to learn what they mean. Write the file again using the `%g` format for both columns and look in the file again.

Chapter 4

Arrays and x-y Plotting

4.1 Colon (:) Command

Simple plots of y vs. x are done with Matlab's plot command and arrays. These arrays are easily built with the colon (:) command. To build an array x of x-values starting at $x = 0$, ending at $x = 10$, and having a step size of $h = .01$ type this:

```
clear;close all;  % close the figure windows
x=0:.01:10;
```

Note that the semicolon at the end of this line is crucial, unless you want to see 1001 numbers scroll down your screen. If you do make this mistake and the screen print is going to take forever, ctrl-c will rescue you.

An array like this that starts at the beginning of an interval and finishes at the end of it is called a cell-edge grid. A cell-center grid is one that has N subintervals, but the data points are at the centers of the intervals, like this

```
dx=.01;
x=.5*dx:dx:10-0.5*dx;
```

Both kinds of grids are used in computational physics. (Note: Matlab's `linspace` command also makes cell-edge grids. Check it out with `help linspace`.)

And if you leave the middle number out of this colon construction, like this

```
t=0:20;
```

then Matlab assumes a step size of 1. You should use the colon command whenever possible because it is a pre-compiled Matlab command. Tests show that using `:` is about 20 times faster than using a loop that you write yourself (discussed in Chapter 9). To make a corresponding array of y values according to the function $y(x) = \sin(5x)$ simply type this

```
y=sin(5*x);
```

Both of these arrays are the same length, as you can check with the `length` command (Note that commands separated with commas just execute one after the other, like this:)

```
length(x),length(y)
```

4.2 xy Plots, Labels, and Titles

To plot y vs. x, just type this

```
close all; % (don't clear--you will lose the data you want to plot)
plot(x,y,'r-');
```

The `'r-'` option string tells the plot command to plot the curve in red connecting the dots with a continuous line. Other colors are also possible, and instead of connecting the dots you can plot symbols at the points with various line styles between the points. To see what the possibilities are type `help plot`.

And what if you want to plot either the first or second half of the x and y arrays? The `colon` and `end` commands can help:

```
nhalf=ceil(length(x)/2);
plot(x(1:nhalf),y(1:nhalf),'b-')
plot(x(nhalf:end),y(nhalf:end),'b-')
```

To label the x and y axes, do this after the `plot` command:

```
xlabel('\theta')
ylabel('F(\theta)')
```

(Note that Greek letters and other symbols are available through LaTex format–see *Greek Letters, Subscripts, and Superscripts* in Section 4.8.) And to put a title on you can do this:

```
title('F(\theta)=sin(5 \theta)')
```

You can even build labels and titles that contain numbers you have generated; use Matlab's `sprintf` command, which works just like `fprintf` except that it writes into a string variable instead of to the screen. You can then use this string variable as the argument of the commands `xlabel`, `ylabel`, and `title`, like this:

```
s=sprintf('F(\\theta)=sin(%i \\theta)',5)
title(s)
```

Note that to force LaTex symbols to come through correctly when using `sprintf` you have to use two backslashes instead of one.

4.3 Generating Multiple Plots

You may want to put one graph in figure window 1, a second plot in figure window 2, etc. To do so, put the Matlab command `figure` before each plot command, like this

```
x=0:.01:20;
f1=sin(x);
f2=cos(x)./(1+x.^2);
```

```
figure
plot(x,f1)
figure
plot(x,f2)
```

And once you have generated multiple plots, you can bring each to the foreground on your screen either by clicking on them and moving them around, or by using the command `figure(1)` to pull up figure 1, `figure(2)` to pull up figure 2, etc. This might be a useful thing to use in a script. See online help for more details.

4.4 Overlaying Plots

Often you will want to overlay two plots on the same set of axes. There are two ways you can do this.

Example 4.4a (ch4ex4a.m)

```
% Example 4.4a (Physics 330)

clear; close all;

%*********************************************************
% Here's the first way -- just ask for multiple plots on the
% same plot line
%*********************************************************
x=0:.01:20;
y=sin(x);   % load y with sin(x)
y2=cos(x);  % load y2 with cos(x), the second function

% plot both

plot(x,y,'r-',x,y2,'b-')
title('First way')

%*********************************************************
% Here's the second way -- after the first plot tell Matlab
% to hold the plot so you can put a second one with it
%*********************************************************

figure
plot(x,y,'r-')
hold on
plot(x,y2,'b-')
title('Second way')
hold off
```

You can now call as many plots as you want and they will all go on the same figure. To release it use the command

```
hold off
```

as shown in the example above.

4.5 xyz Plots: Curves in 3-D Space

Matlab will draw three-dimensional curves in space with the `plot3` command. Here is how you would do a spiral on the surface of a sphere using spherical coordinates.

Example 4.5a (ch4ex5a.m)

```
% Example 4.5a (Physics 330)

clear; close all;

dphi=pi/100; % set the spacing in azimuthal angle

N=30; % set the number of azimuthal trips
phi=0:dphi:N*2*pi;

theta=phi/N/2; % go from north to south once

r=1;  % sphere of radius 1

% convert spherical to Cartesian
x=r*sin(theta).*cos(phi);
y=r*sin(theta).*sin(phi);
z=r*cos(theta);

% plot the spiral
plot3(x,y,z,'b-')
axis equal
```

4.6 Logarithmic Plots

To make log and semi-log plots use the commands `semilogx`, `semilogy`, and `loglog`. They work like this.

Example 4.6a (ch4ex6a.m)

```
% Example 4.6a (Physics 330)

clear; close all;

x=0:.1:8;
y=exp(x);

semilogx(x,y);
title('semilogx')

figure
```

```
semilogy(x,y);
title('semilogy')

figure
loglog(x,y);
title('loglog')
```

4.7 Controlling the Axes

You have probably noticed that Matlab chooses the axes to fit the functions that you are plotting. You can override this choice by specifying your own axes, like this.

```
close all;
x=.01:.01:20;
y=cos(x)./x;
plot(x,y)
axis([0 25 -5 5])
```

Or, if you want to specify just the x-range or the y-range, you can use `xlim`:

```
plot(x,y)
xlim([ 0 25])
```

or `ylim`:

```
plot(x,y)
ylim([-5 5])
```

And if you want equally scaled axes, so that plots of circles are perfectly round instead of elliptical, use

```
axis equal
```

4.8 Greek Letters, Subscripts, and Superscripts

When you put labels and titles on your plots you can print Greek letters, subscripts, and superscripts by using the LaTex syntax. (See a book on LaTex for details.) To print Greek letters just type their names preceded by a backslash, like this:

\alpha	\beta	\gamma	\delta	\epsilon	\phi
\theta	\kappa	\lambda	\mu	\nu	\pi
\rho	\sigma	\tau	\xi	\zeta	

You can also print capital Greek letters, like this \Gamma, i.e., you just capitalize the first letter.

To put a subscript on a character use the underscore character on the keyboard: θ_1 is coded by typing `\theta_1`. And if the subscript is more than one character long do this: `\theta_{12}` (makes θ_{12}). Superscripts work the same way only using the $\wedge$ character: use `\theta^{10}` to print θ^{10}.

To write on your plot, you can use Matlab's `text` command in the format:

```
text(10,.5,'Hi');
```

which will place the text "Hi" at position $x = 10$ and $y = 0.5$ on your plot.

You can use LaTex Greek in labels and titles too. If you want Matlab to layout an equation like LaTex would (rather than just getting the Greek letters in your labels), you use the following syntax:

```
title('Plot of $\frac{\sin(x)}{x}$','Interpreter','Latex')
```

With this interpreter, you type text and equations in regular LaTex syntax and Matlab will use LaTex to typeset the text. (Read a LaTex tutorial for further information on this format.)

4.9 Changing Line Widths, Fonts, Etc.

You can also use a number of graphical tools to change line widths, put text and lines on the plot, etc. To do this, click the "Show Plot Tools" button on the toolbar. If you want to change the look of anything on your plot, like the font style or size of text, the width of the lines, the font style and size of the axis labels, etc., just left click on the thing you want to change until it is highlighted, then right click on it and select Properties.

This will take care of simple plots, but if you want to make publication quality figures you will have to work harder. See Chapter 17 at the end of this booklet titled Plots for Publication for more information.

Chapter 5

Surface, Contour, and Vector Field Plots

5.1 Meshgrid and Ndgrid

Matlab will also display functions of the type $F(x, y)$, either by making a contour plot (like a topographic map) or by displaying the function as height above the xy plane like a perspective drawing. Start by defining arrays x and y that span the region that you want to plot, then create the function $F(x, y)$ over the plane, and finally either use `contour`, `surf`, or `mesh`.

In this subsection we will try to understand how Matlab goes from one-dimensional arrays x and y to two-dimensional matrices X and Y using the commands `meshgrid` and `ndgrid`. Let's begin by executing the following example.

Example 5.1a (ch5ex1a.m)

```
% Example 5.1a (Physics 330)

%*********************************************************
% Define the arrays x and y
% Warning: don't make the step size too small or you will
% kill the system
%*********************************************************

clear;close all;
x=-1:.1:1;y=0:.1:1.5;

%*********************************************************
% Use meshgrid to convert these 1-d arrays into 2-d matrices of
% x and y values over the plane
%*********************************************************

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

%*********************************************************
% Get F(x,y) by using F(X,Y).  Note the use of .* with X and Y
```

```
% rather than with x and y
%*********************************************************

F=(2-cos(pi*X)).*exp(Y);

%*********************************************************
% Note that this function is uphill in y between x=-1 and x=1
% and has a valley at x=0
%*********************************************************

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

Well, the picture should convince you that Matlab did indeed make things two-dimensional, and that this way of plotting could be very useful. But exactly how Matlab did it is tricky, so pay close attention.

To understand how `meshgrid` turns one-dimensional arrays x and y into two-dimensional matrices X and Y, consider the following simple example. Suppose that the arrays x and y are given by

$$x = [1, 2, 3] \qquad y = [4, 5].$$

The command `[X,Y]=meshgrid(x,y)` produces the following results:

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \qquad Y = \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \end{bmatrix}.$$

As you can see, the first index of both X and Y is a y-index (since there are only 2 entries, as in y). $X(1,1)$ and $X(2,1)$ are both 1 because as the y-index changes, x stays the same. Similarly, $Y(1,1) = 4$ and $Y(2,1) = 5$ because as the y-index changes, y does change. This means that if we think of x having an array index i so that $x = x(i)$ and think of y having index j so that $y = y(j)$, then the two-dimensional matrices have indices

$$X(j,i) \quad Y(j,i).$$

To see that this is how the example above worked for you, make sure your Workspace window is open, then click on `X` and `Y` to view them with the Array Editor. Look at them until you are convinced that `meshgrid` has turned your one-dimensional arrays `x(i)` and `y(j)` into their two-dimensional versions `X(j,i)` and `Y(j,i)`.

But in physics we usually think of two-dimensional functions of x and y in the form $F(x, y)$, i.e., the first position is for x and the second one is for y. Because we think this way it would be nice if the matrix indices worked this way too, meaning that if $x = x(i)$ and $y = y(j)$, then the two-dimensional matrices would be indexed as

$$X(i,j) \quad Y(i,j)$$

instead of in the backwards order made by `meshgrid`.

Fortunately, Matlab has another command called `ndgrid` which is similar to `meshgrid` but does the conversion to two dimensions the other way round. For instance, with the example arrays for x and y used above `[X,Y]=ndgrid(x,y)` would produce the results

$$X = \begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix} \qquad Y = \begin{bmatrix} 4 & 5 \\ 4 & 5 \\ 4 & 5 \end{bmatrix}$$

which is in the $X(i,j)$, $Y(i,j)$ order. (Stare at the matrices above and discuss them with your lab partner until you are convinced that this is true.)

No matter which command you use, plots made either with `surf(X,Y,F)` or `contour(X,Y,F)` (discussed below) will look the same, but the $F(i,j)$ order is a more natural fit to the way we usually to two-dimensional physics problems, so we suggest you use `ndgrid` most of the time.

5.2 Contour Plots and Surface Plots

Now that we understand `meshgrid` and `ndgrid`, let's make some two-dimensional plots. Download, or type, the following example, read through it and watch what it does.

Example 5.2a (ch5ex2a.m)

```
% Example 5.2a (Physics 330)

clear; close all;

%*********************************************************
% make a contour plot by asking Matlab to evenly space N contours
% between the minimum of F(x,y) and the maximum F(x,y) (the default)
%*********************************************************

x=-1:.1:1;y=0:.1:1.5;
[X,Y]=ndgrid(x,y);
F=(2-cos(pi*X)).*exp(Y);

N=40;
contour(X,Y,F,N);
title('Simple Contour Plot')
xlabel('x')
ylabel('y')

%*********************************************************
% You can also tell Matlab which contour levels you want to plot.
% To do so load an array (V in this case) with the values of the
% levels you want to see.  In this example they will start at
% the minimum of F, end at the maximum of F, and there will
% be 21 contours.
%
% You can even print contour labels on the plot, which is
% a big help, by assigning the plot to a variable name
```

```
% and using the clabel command, as shown below.  Only
% every other contour is labeled in this example
%*********************************************************

top=max(max(F));  % find the max and min of F
bottom=min(min(F));
dv=(top-bottom)/20; % interval for 21 equally spaced contours
V=bottom:dv:top;
figure
cs=contour(X,Y,F,V);
clabel(cs,V(1:2:21)) % give clabel the name of the plot and
                     % an array of the contours to label
title('Fancy Contour Plot')
xlabel('x')
ylabel('y')


%*********************************************************
% Now make a surface plot of the function with the viewing
% point rotated by AZ degrees from the x-axis and
% elevated by EL degrees above the xy plane
%*********************************************************

figure
surf(X,Y,F); % or you can use mesh(X,Y,F) to make a wire plot
AZ=30;EL=45;
view(AZ,EL);
title('Surface Plot')
xlabel('x')
ylabel('y')
```

If you want to manually change the viewing angle of a surface plot, click on the circular arrow icon on the figure window, then click and move the pointer on the graph. Try it until you get the hang of it.

Here's a piece of code that lets you fly around the surface plot by continually changing the viewing angles and using the pause command; we think you'll be impressed

Example 5.2b (ch5ex2b.m)

```
% Example 5.2b (Physics 330)

clear; close all;

x=-1:.1:1;
y=0:.1:1.5;
[X,Y]=ndgrid(x,y);
F=(2-cos(pi*X)).*exp(Y);

surf(X,Y,F);
title('Surface Plot')
```

```
xlabel('x')
ylabel('y')
EL=45;
for m=1:100
   AZ=30+m/100*360;
   view(AZ,EL);
   pause(.1);  % pause units are in seconds
end
```

This same trick will let you make animations of both xy and surface plots. To make this surface oscillate up and down like a manta ray you could do this.

Example 5.2c (ch5ex2c.m)

```
% Example 5.2c (Physics 330)

clear; close all;

x=-1:.1:1;
y=0:.1:1.5;
[X,Y]=ndgrid(x,y);
F=(2-cos(pi*X)).*exp(Y);

dt=.1;

for m=1:100
   t=m*dt;
   g=F*cos(t);

   surf(X,Y,g);
   AZ=30;EL=45;
   view(AZ,EL);
   title('Surface Plot')
   xlabel('x')
   ylabel('y')
   axis([-1 1 -1 1 min(min(F)) max(max(F))])
   pause(.1)

end
```

Note that you can find all kinds of cool stuff about surface and contour plotting by typing

```
help graph3d
```

and then checking out these commands by using help on each one. Another good source is *Mastering Matlab 6*, Chapters 26-32.

5.3 Evaluating Fourier Series

Matlab will make graphical displays of infinite series of the kind discussed in Physics 318 and Physics 441 quickly and easily. Consider this solution for the electrostatic potential in a long tube of rectangular cross-section bounded by long metal strips held at different potentials. The tube has width $2b$ in x and width a in y. The two strips at $x = -b$ and $x = +b$ are held at potential V_0 while the strips at $y = 0$ and $y = a$ are grounded. (See *Introduction to Electrodynamics, Third Edition* by David Griffiths, pages 132-134.) The electrostatic potential is given by

$$V(x,y) = \frac{4V_0}{\pi} \sum_{n=0}^{\infty} \frac{1}{(2n+1)} \frac{\cosh\left[(2n+1)\pi x/a\right]}{\cosh\left[(2n+1)\pi b/a\right]} \sin\left[(2n+1)\pi y/a\right]. \tag{5.1}$$

Here is a piece of Matlab code that evaluates this function on an xy grid and displays it as a surface plot.

Example 5.3a (ch5ex3a.m)

```
% Example 5.3a (Physics 330)

clear;close all;

% set some constants

a=2;b=1;Vo=1;

% build the x and y grids

Nx=80;Ny=40;
dx=2*b/Nx;dy=a/Ny;
x=-b:dx:b;y=0:dy:a;

% build the 2-d grids for plotting

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

% set the number of terms to keep
% and do the sum

Nterms=20;

% zero V out so we can add into it

V=zeros(Ny+1,Nx+1);

% add the terms of the sum into V

for m=0:Nterms
   V=V+cosh((2*m+1)*pi*X/a)/cosh((2*m+1)*pi*b/a).*sin((2*m+1)*pi*Y/a)/(2*m+1);
end
```

```
% put on the multiplicative constant

V=4*Vo/pi*V;

% surface plot the result

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

5.4 Vector Field Plots

Matlab will plot vector fields for you with arrows. This is a good way to visualize flows, electric fields, magnetic fields, etc. The command that makes these plots is `quiver` and the code below illustrates its use in displaying the electric field of a line charge and the magnetic field of a long wire. Note that the vector field components must be computed in Cartesian geometry.

Example 5.4a (ch5ex4a.m)

```
% Example 5.4a (Physics 330)

clear;close

x=-5.25:.5:5.25;y=x; % define the x and y grids (avoid (0,0))
[X,Y]=meshgrid(x,y);

% Electric field of a long charged wire

Ex=X./(X.^2+Y.^2);
Ey=Y./(X.^2+Y.^2);

% make the field arrow plot

quiver(X,Y,Ex,Ey)
title('E of a long charged wire')

axis equal  % make the x and y axes be equally scaled

% Magnetic field of a long current-carrying wire

Bx=-Y./(  X.^2+Y.^2);
By=X./(X.^2+Y.^2);

% make the field arrow plot

figure
```

```
quiver(X,Y,Bx,By)
axis equal
title('B of a long current-carrying wire')

%*********************************************************
% The big magnitude difference across the region makes most arrows too small
% to see.  This can be fixed by plotting unit vectors instead (losing all
% magnitude information
%*********************************************************

B=sqrt(Bx.^2+By.^2);
Ux=Bx./B;
Uy=By./B;

figure
quiver(X,Y,Ux,Uy);
axis equal
title('B(wire): unit vectors')

%*********************************************************
% Or, you can still see qualitative size information
% without such a big variation in arrow size by
% having the arrow length be logarithmic.  If s is
% the desired ratio between the longest arrow and
% the shortest one, this code will make the  appropriate
% field plot.
%*********************************************************

Bmin=min(min(B));
Bmax=max(max(B));
s=2;  % choose an arrow length ratio
k=(Bmax/Bmin)^(1/(s-1));

logsize=log(k*B/Bmin);
Lx=Ux.*logsize;
Ly=Uy.*logsize;

figure
quiver(X,Y,Lx,Ly);
axis equal
title('B(wire): logarithmic arrows')
```

There may be too much detail to really see what's going on in some field plots. You can work around this problem by clicking on the zoom icon on the tool bar and then using the mouse to define the region you want to look at. Clicking on the zoom-out icon, then clicking on the figure will take you back where you came from. Or double-click on the figure will also take you back.

Chapter 6

Vector Products, Dot and Cross

Matlab will do dot and cross products for you with the commands `dot` and `cross`, like this:

```
a=[1,2,3];
b=[3,2,1];
dot(a,b)
cross(a,b)
```

(Cross products only work for three-dimensional vectors but dot products can be used with vectors of any length.)

Chapter 7

Linear Algebra

Almost anything you learned about in your linear algebra class Matlab has a command to do. Here is a brief summary of the most useful ones for physics.

7.1 Solve a Linear System

Matlab will solve the matrix equation $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a square matrix, where $\mathbf{b}$ is a known column vector, and where $\mathbf{x}$ is an unknown column vector. For instance, the system of equations

$$\begin{aligned} x + z &= 4 \\ -x + y + z &= 4 \\ x - y + z &= 2\ , \end{aligned} \tag{7.1}$$

which is solved by $(x, y, z) = (1, 2, 3)$, is handled in Matlab by defining a matrix A corresponding to the coefficients on the left side of this equation and a column vector b corresponding to the coefficients on the right (see the example below.) By use of the simple symbol \ (sort of "backwards divide") Matlab will use Gaussian elimination to solve this system of equations, like this:

```
% here are A and b corresponding to the equations above

A=[ 1, 0,1
   -1, 1,1
    1,-1,1 ];
b=[4
   4
   2];
% now solve for x and see if we obtain [1;2;3], like we should

x=A\b
```

7.2 Max and Min

The commands `max` and `min` return the maximum and minimum values of an array. And with a slight change of syntax they will also return the indices in the array at which the maximum and minimum occur.

Example 7.2a (ch7ex2a.m)

```
% Example 7.2a (Physics 330)

clear; close all;

x=0:.01:5;
y=x.*exp(-x.^2);

% take a look at the function so we know what it looks like

plot(x,y)

% find the max and min

ymin=min(y)
ymax=max(y)

% find the max and min along with the array indices imax and imin
% where they occur

[ymin,imin]=min(y)
[ymax,imax]=max(y)
```

7.3 Matrix Inverse

The `inv` command will compute the inverse of a square matrix. For instance, using the matrix

```
A=[1,0,-1;-1,1,1;1,-1,1]
```

we find

```
% load C with the inverse of A

C=inv(A)

% verify by matrix multiplication that A*C is the identity matrix

A*C
```

7.4 Transpose and Hermitian Conjugate

To find the transpose of the matrix A just use a single quote with a period, like this

```
A.'
```

To find the Hermitian conjugate of the matrix A (transpose of A with all elements replaced with their complex conjugates) type

```
A'
```

(notice that there isn't a period). If your matrices are real, then there is no difference between these two commands and you might as well just use `A'`. Notice that if `a` is a row vector then `a'` is a column vector. You will use the transpose operator to switch between row and column vectors a lot in Matlab, like this

```
[1,2,3]
[1,2,3]'
[4;5;6]
[4;5;6]'
```

7.5 Special Matrices

Matlab will let you load several special matrices. The most useful of these are given here.

```
% eye:
% load I with the 4x4 identity matrix (the programmer who invented this
% syntax must have been drunk)

I=eye(4,4)

% zeros:
% load Z with a 5x5 matrix full of zeros

Z=zeros(5,5)

% ones:
% load X with a 3x3 matrix full of ones

X=ones(3,3)

% rand:
% load Y with a 4x6 matrix full of random numbers between 0 and 1
% The random numbers are uniformly distributed on [0,1]

Y=rand(4,6)
```

```
% And to load a single random number just use

r=rand

% randn:
% load Y with a 4x6 matrix full of random numbers with a Gaussian
% distribution with zero mean and a variance of 1

Y=randn(4,6)
```

7.6 Determinant

Find the determinant of a square matrix this way

```
det(A)
```

7.7 Norm of Vector (Magnitude)

Matlab will compute the magnitude of a vector `a` (the square root of the sum of the squares of its components) with the `norm` command

```
a=[1,2,3]
norm(a)
```

7.8 Sum the Elements

For arrays the command `sum` adds up the elements of the array. For instance, the following commands calculate the sum of the squares of the reciprocals of the integers from 1 to 10,000.

```
n=1:10000;
sum(1./n.^2)
```

You can compare this answer with the sum to infinity, which is $\pi^2/6$, by typing

```
ans-pi^2/6
```

For matrices the `sum` command produces a row vector which is made up of the sum of the columns of the matrix.

```
A=[1,2,3;4,5,6;7,8,9]
sum(A)
```

7.9 Selecting Rows and Columns

Sometimes you will want to select a row or a column of a matrix and load it into an array. This is done with Matlab's all-purpose colon (:) command.

To load a column vector b with the contents of the third column of the matrix A use:

```
b=A(:,3)
```

Recall that the first index of a matrix is the row index, so this command tells Matlab to select all of the rows of A in column 3.

To load a row vector c with the contents of the second row of the matrix A use:

```
c=A(2,:)
```

You can also select just part of row or column like this:

```
c=A(2,1:2)
```

which takes only the first two elements of the second row.

7.10 Eigenvalues and Eigenvectors

To build a column vector containing the eigenvalues of the matrix A in the previous section use

```
E=eig(A)
```

To build a matrix V whose columns are the eigenvectors of the matrix A and another matrix D whose diagonal elements are the eigenvalues corresponding to the eigenvectors in V use

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

To select the 3rd eigenvector and load it into a column vector use

```
v3=V(:,3)  % i.e., select all of the rows (:) in column 3
```

7.11 Fancy Stuff

Matlab also knows how to do singular value decomposition, QR factorization, LU factorization, and conversion to reduced row-echelon form. And the commands `rcond` and `cond` will give you the condition number of a matrix. To learn about these ideas, consult a textbook on linear algebra. To learn how they are used in Matlab use the commands;

```
help svd
help QR
help LU
help rref
help rcond
help cond
```

Chapter 8

Polynomials

Polynomials are used so commonly in computation that Matlab has special commands to deal with them. The polynomial $x^4 + 2x^3 - 13x^2 - 14x + 24$ is represented in Matlab by the array `[1,2,-13,-14,24]`, i.e., by the coefficients of the polynomial starting with the highest power and ending with the constant term. If any power is missing from the polynomial its coefficient must appear in the array as a zero. Here are some of the things Matlab can do with polynomials. Try each piece of code in Matlab and see what it does.

8.1 Roots of a Polynomial

The following command will find the roots of a polynomial:

```
p=[1,2,-13,-14,24];
r=roots(p)
```

8.2 Find the polynomial from the roots

If you know that the roots of a polynomial are 1, 2, and 3, then you can find the polynomial in Matlab's array form this way

```
r=[1,2,3];
p=poly(r)
```

8.3 Multiply Polynomials

The command `conv` returns the coefficient array of the product of two polynomials.

```
a=[1,0,1];
b=[1,0,-1];
c=conv(a,b)
```

Stare at this result and make sure that it is correct.

8.4 Divide Polynomials

Remember synthetic division? Matlab can do it with the command `deconv`, giving you the quotient and the remainder.

```
a=[1,1,1]; % a=x^2+x+1
b=[1,1];   % b=x+1

% now divide b into a finding the quotient and remainder

[q,r]=deconv(a,b)
```

After you do this Matlab will give you `q=[1,0]` and `r=[0,0,1]`. This means that $q = x + 0 = x$ and $r = 0x^2 + 0x + 1 = 1$, so

$$\frac{x^2 + x + 1}{x + 1} = x + \frac{1}{x + 1}. \tag{8.1}$$

8.5 First Derivative

Matlab can take a polynomial array and return the polynomial array of its derivative:

```
a=[1,1,1,1]
ap=polyder(a)
```

8.6 Evaluate a Polynomial

If you have an array of x-values and you want to evaluate a polynomial at each one, do this:

```
% define the polynomial

a=[1,2,-13,-14,24];

% load the x-values

x=-5:.01:5;

% evaluate the polynomial

y=polyval(a,x);

% plot it

plot(x,y)
```

8.7 Fitting Data to a Polynomial

If you have some data in the form of arrays (x,y), Matlab will do a least-squares fit of a polynomial of any order you choose to this data. In this example we will let the data be the sine function between 0 and π and we will fit a polynomial of order 4 to it. Then we will plot the two functions on the same frame to see if the fit is any good. Before going on to the next section, try fitting a polynomial of order 60 to the data to see why you need to be careful when you do fits like this.

Example 8.7a (ch8ex7a.m)

```
% Example 8.7a (Physics 330)

clear; close all;

x=linspace(0,pi,50);

% make a sine function with 1% random error on it

f=sin(x)+.01*rand(1,length(x));

% fit to the data

p=polyfit(x,f,4);

% evaluate the fit

g=polyval(p,x);

% plot fit and data together

plot(x,f,'r*',x,g,'b-')
```

Chapter 9

Loops and Logic

To use Matlab to solve many physics problems you have to know how to write loops and how to use logic.

9.1 Loops

A loop is a way of repeatedly executing a section of code. It is so important to know how to write them that several common examples of how they are used will be given here. The two kinds of loops we will use are the `for` loop and the `while` loop. We will look at `for` loops first, then study `while loops` a bit later in the logic section.

The `for` loop looks like this:

```
for n=1:N . . . end
```

which tells Matlab to start n at 1, then increment it by 1 over and over until it counts up to N, executing the code between `for` and `end` for each new value of n. Here are a few examples of how the `for` loop can be used.

Summing a series with a `for` loop

Let's do the sum

$$\sum_{n=1}^{N} \frac{1}{n^2} \tag{9.1}$$

with N chosen to be a large number

Example 9.1a (ch9ex1a.m)

```
% Example 9.1a (Physics 330)

s=0; % set a variable to zero so that 1/n^2 can be repeatedly added to it
N=10000; % set the upper limit of the sum
for n=1:N % start of the loop

% add 1/n^2 to s each time, then put the answer back into s
.
```

```
.
.

end % end of the loop
fprintf(' Sum = %g \n',s) % print the answer
```

You may notice that summing with a loop takes a lot longer than the matrix operator way of doing it:

```
% calculate the sum of the squares of the reciprocals of the
% integers from 1 to 10,000

n=1:10000;
sum(1./n.^2)
```

Try both the loop way in Example 9.1a and this `sum` command way and see which is faster. To slow things down enough that you can see the difference change 10,000 to 100,000. (When we tested this, the `:` way was 21 times faster than the loop, so use array operators whenever you can.) To do timing checks use the `tic` and `toc` commands. Look them up in online help.

To get some more practice with loops, let's do the running sum

$$S_m = \sum_{n=1}^{m} a_n \qquad \text{where} \qquad a_n = \frac{1}{n^2} \tag{9.2}$$

for values of m from 1 to a large number N.

Example 9.1b (ch9ex1b.m)

```
% Example 9.1b (Physics 330)
clear;
close all;

N=100;

a = zeros(1,N);
% Fill the a array
for n=1:N
    a(n) = 1 / n^2;
end

S = zeros(1,N);
% Do the running sum
for m=1:N
    S(m) = sum( a(1:m) );
end

% Now let's plot S vs. m
m=1:N
plot(m,S)
```

Notice that in this example we pre-allocate the arrays `a` and `S` with the `zeros` command before each loop. If you don't do this, Matlab has to go grab an extra little chunk of memory to expand the arrays each time the loop iterates and this makes the loops run very slowly as N gets big.

We also could have done this cumulative sum using colon operators and the `cumsum` command, like this:

```
n=1:100;
S=cumsum(1./n.^2);
```

(but we are practicing loops here).

Products with a `for` loop

Let's calculate $N! = 1 \cdot 2 \cdot 3 \cdot ...(N-1) \cdot N$ using a for loop that starts at $n = 1$ and ends at $n = N$, doing the proper multiply at each step of the way.

Example 9.1c (ch9ex1c.m)

```
% Example 9.1c (Physics 330)

P=1;  % set the first term in the product
N=20;  % set the upper limit of the product

for n=2:N % start the loop at n=2 because we already loaded n=1
  P=P*n; % multiply by n each time and put the answer back into P
end % end of the loop

fprintf(' N!  = %g \n',P)  % print the answer
```

Now use Matlab's `factorial` command to check that you found the right answer:

```
factorial(20)
```

You should be aware that the `factorial` command is a bit limited in that it won't act on an array of numbers in the way that cos, sin, exp etc. do. A better factorial command to use is the gamma function $\Gamma(x)$ which extends the factorial function to all complex values. It is related to the factorial function by $\Gamma(N+1) = N!$, and is called in Matlab using the command `gamma(x)`, so you could also check the answer to your factorial loop this way:

```
gamma(21)
```

Recursion relations with `for` loops

Suppose that we were solving a differential equation by substituting into it a power series of the form

$$f(x) = \sum_{n=1}^{\infty} a_n x^n \tag{9.3}$$

and that we had discovered that the coefficients a_n satisfied the recursion relation

$$a_1 = 1 \quad ; \quad a_{n+1} = \frac{2n-1}{2n+1} a_n. \tag{9.4}$$

To use these coefficients we need to load them into an array a so that $a(1) = a_1, a(2) = a_2, \cdots$. Here's how we could do this using a `for` loop to load $a(1)...a(20)$:

Example 9.1d (ch9ex1d.m)

```
% Example 9.1d (Physics 330)

a(1)=1;  % put the first element into the array
N=19;  % the first one is loaded, so let's load 19 more

for n=1:N %  start the loop
   a(n+1)=(2*n-1)/(2*n+1)*a(n); % the recursion relation
end

disp(a) %  display the resulting array of values
```

Note that the recursion relation was translated into Matlab code just as it appeared in the formula: `a(n+1)=(2*n-1)/(2*n+1)*a(n)`. The counting in the loop was then adjusted to fit by starting at $n = 1$, which loaded $a(1+1) = a(2)$, then $a(3)$, etc., then ended at $n = 19$, which loads $a(19 + 1) = a(20)$. Always make the code you write fit the mathematics as closely as possible, then adjust the other coding to fit. This will make your code easier to read and you will make fewer mistakes.

9.2 Logic

Often we only want to do something when some condition is satisfied, so we need logic commands. The simplest logic command is the `if` command, which works like this. (Several examples are given; try them all.)

Example 9.2a (ch9ex2a.m)

```
% Example 9.2a (Physics 330)

clear;
a=1;b=3;

% If the number a is positive set c to 1; if a is 0 or negative,
% set c to zero

if a>0
   c=1
else
   c=0
end
```

```
% if either a or b is non-negative, add them to obtain c;
% otherwise multiply a and b to obtain c

if a>=0 | b>=0  % either non-negative
   c=a+b
else
   c=a*b  % otherwise multiply them to obtain c
end
```

You can build any logical condition you want if you just know the basic logic elements. Here they are

Equal	`==`
Less than	`<`
Greater than	`>`
Less than or equal	`<=`
Greater than or equal	`>=`
Not equal	$\sim=$
And	`&`
Or	`|`
Not	$\sim$

There is also a useful logic command that controls loops: `while`. Suppose you don't know how many times you are going to have to loop to get a job done, but instead want to quit looping when some condition is met. For instance, suppose you want to add the reciprocals of squared integers until the term you just added is less than 1e-10. Then you would change the loop in the $\sum 1/n^2$ example to look like this

Example 9.2b (ch9ex2b.m)

```
% Example 9.2b (Physics 330)

clear
term=1  % load the first term in the sum, 1/1^2=1
s=term;  % load s with this first term

% start of the loop - set a counter n to one

n=1;

while term > 1e-10  % loop until term drops below 1e-10
   n=n+1; % add 1 to n so that it will count: 2,3,4,5,...
   term=1/n^2; % calculate the next term to add
   s=s+term;  % add 1/n^2 to s until the condition is met
end  % end of the loop

fprintf(' Sum = %g \n',s)
```

This loop will continue to execute until `term`<1e-10. Note that unlike the `for` loop, here you have to do your own counting, being careful about what value n starts at and when it is incremented ($n = n + 1$). It is also important to make sure that the variable you are testing (`term` in this case) is loaded before the loop starts with a value that allows the test to take place and for the loop to run (`term` must pass the `while` test.)

Sometimes `while` is awkward to use because you would rather just loop lots of times checking some condition and then break out of the loop when it is satisfied. The `break` command is designed to do this. When `break` is executed in a loop the script jumps to just after the `end` at the bottom of the loop. Here is our sum loop rewritten with `break`

Example 9.2c (ch9ex2c.m)

```
% Example 9.2c (Physics 330)

clear
s=0; % initialize the sum variable

% start of the loop

for n=1:1000000
   term=1/n^2;

% add 1/n^2 to s until the condition is met
   s=s+term;
   if term < 1e-10
      break
   end
% end of the loop
end

fprintf(' Sum = %g \n',s)
```

9.3 Secant Method

Here is a real example that you will sometimes use to solve difficult equations of the form $f(x) = 0$ in Matlab. (Matlab's `fzero` command, which you will learn to use shortly, does this for you automatically; here you will see how they do it.)

The idea is to find two guesses x_1 and x_2 that are near a solution of this equation. (You find about where these two guesses ought to be by plotting the function and seeing about where the solution is. It's OK to choose them close to each other, like $x_1 = .99$ and $x_2 = .98$.) Once you have these two guesses find the function values that go with them: $f_1 = f(x_1)$ and $f_2 = f(x_2)$ and compute the slope $m = (f_2 - f_1)/(x_2 - x_1)$ of the line connecting the points. (You can follow what is happening here by looking at Fig. 9.1.) Then fit a straight line through these two points and solve the resulting straight line equation $y - f_2 = m(x - x_2)$ for the value of x that makes $y = 0$, i.e., solve the line equation to find

$$x_3 = x_2 - \frac{f_2}{m} = x_2 - \frac{f_2(x_2 - x_1)}{f_2 - f_1} \tag{9.5}$$

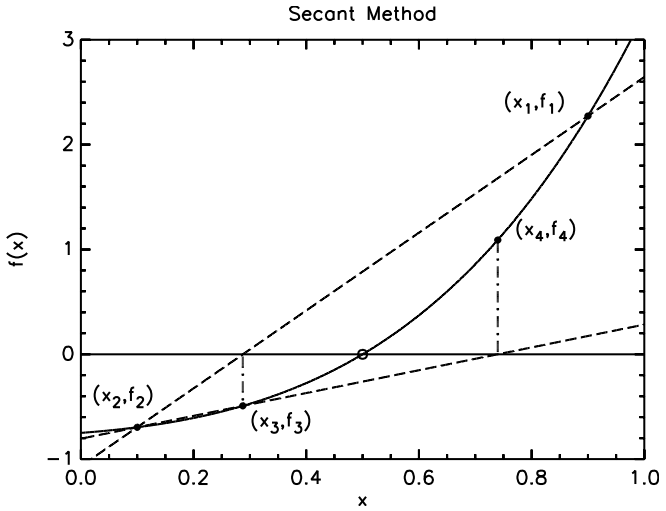


Figure 9.1 The sequence of approximate points in the secant method.

as shown in Fig. 9.1. This will be a better approximation to the solution than either of your two initial guesses, but it still won't be perfect, so you have to do it again using x_2 and the new value of x_3 as the two new points. This will give you x_4 in the figure. You can draw your own line and see that the value of x_5 obtained from the line between (x_3, f_3) and (x_4, f_4) is going to be pretty good. And then you do it again, and again, and again, until your approximate solution is good enough.

Here's what the code looks like that solves the equation $\exp(-x) - x = 0$

Example 9.3a (ch9ex3a.m)

```
% Exampmle 9.3a (Physics 330)

clear;close all;

%************************************************************
% Define the function as an in line function (See Chapter 12
% for more details.)
%************************************************************

func=inline('exp(-x)-x','x');

% First plot the function

x=0:.01:2;
f=func(x);
plot(x,f,'r-',x,0*x,'b-')

%************************************************************
% (Note that the second plot is just a blue x-axis (y=0)
% 0*x is just a quick way to load an array of zeros the
% same size as x)
```

```
% From the plot it looks like the solution is near x=.6

% Secant method to solve the equation exp(-x)-x = 0

% Use an initial guess of x1=0.6
%*********************************************************

x1=0.6;

% find f(x1)

f1=func(x1);

% find a nearby second guess

x2=0.99*x1;

% set chk, the error, to 1 so it won't trigger the while
% before the loop gets started

chk=1;

% start the loop

while chk>1e-8

% find f(x2)

   f2=func(x2);

% find the new x from the straight line approximation and print it

   xnew = x2 - f2*(x2-x1)/(f2-f1)

% find chk the error by seeing how closely f(x)=0 is approximated

   chk=abs(f2);

% load the old x2 and f2 into x1 and f1; then put the new x into x2

   x1=x2;f1=f2;x2=xnew;

% end of loop
end
```

(Note: this is similar to Newton's method, also called Newton-Raphson, but when a finite-difference approximation to the derivative is used it is usually called the secant method.)

9.4 Using Matlab's Fzero

Matlab has its own zero-finder which probably is similar to the secant method described above. To use it you must make a special M-file called a *function*, which we will discuss in more detail in chapter 12. Here we will just give you a sample file so that you can see how `fzero` works. This function file (called `fz.m` here) evaluates the function $f(x)$. You just need to build it, tell Matlab what its name is using the @-sign syntax illustrated below, and also give Matlab an initial guess for the value of x that satisfies $f(x) = 0$.

In the section of code below you will find the Matlab function `fz(x)` and the line of code invoking `fzero` that finds the root. The example illustrated here is $f(x) = \exp(-x) - x = 0$.

Here is the function M-file `fz.m` used by `fzero` in this example: (Note: both of these files must be stored in the same directory.)

Example 9.4a (fz.m)

```
% Example 9.4a (Physics 330)

function f=fz(x)

% evaluate the function fz(x) whose
% roots are being sought

f=exp(-x)-x;
```

Here is the Matlab code that uses `fzero` and `fz` to do the solve:

Example 9.4b (ch9ex4b.m)

```
% Example 9.4b (Physics 330)

%*********************************************************
%  Here is the matlab code that uses fz.m to find
%  a zero of f(x)=0 near the guess x=.7
%  Note that the @ sign is used to tell Matlab that
%  the name of an M-file is being passed into fzero
%*********************************************************

x=fzero(@fz,.7)
```

Chapter 10

Derivatives and Integrals

Matlab won't give you formulas for the derivatives and integrals of functions like a symbolic math program. But if you have closely spaced arrays filled with values of x and $f(x)$ Matlab will quickly give you numerical approximations to the derivative $f'(x)$ and the integral $\int_a^b f(x)dx$.

10.1 Derivatives

In your first calculus class the following formula for the derivative was given:

$$\frac{df}{dx} = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}. \tag{10.1}$$

To do a derivative numerically we use the following slightly different, but numerically more accurate, form:

$$\frac{df}{dx} = \lim_{h\to 0} \frac{f(x+h) - f(x-h)}{2h}. \tag{10.2}$$

It's more accurate because it's centered about the value of x where we want the derivative to be evaluated.

To see the importance of centering, consider Fig. 10.1. In this figure we are trying to find the slope of the tangent line at $x = 0.4$. The usual calculus-book formula uses the data points at $x = 0.4$ and $x = 0.5$, giving tangent line a. It should be obvious that using the "centered" pair of points $x = 0.3$ and $x = 0.5$ to obtain tangent line b is a much better approximation.

As an example of what a good job centering does, try differentiating $\sin x$ this way:

```
dfdx=(sin(1+1e-5)-sin(1-1e-5))/2e-5

% take the ratio between the numerical derivative and the
% exact answer cos(1) to see how well this does

format long e
dfdx/cos(1)
```

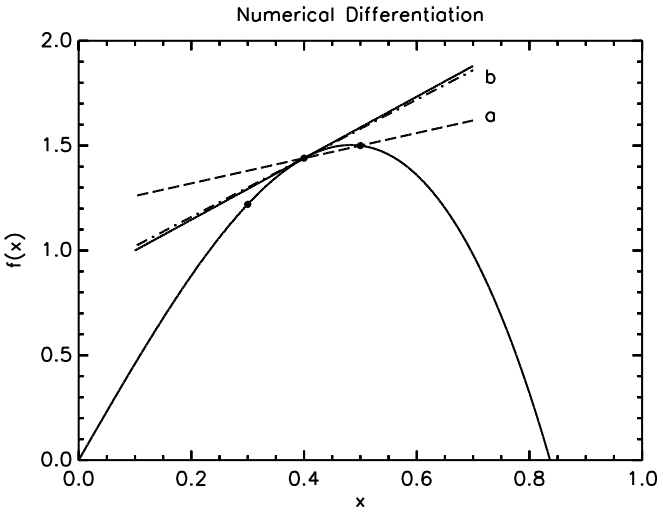


Figure 10.1 The centered derivative approximation works best.

You can also take the second derivative numerically using the formula

$$\frac{d^2f}{dx^2} = \lim_{h\to 0} \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}. \tag{10.3}$$

For example,

```
d2fdx2=(sin(1+1e-4)-2*sin(1)+sin(1-1e-4))/1e-8

% take the ratio between the numerical derivative and the
% exact answer -sin(1) to see how well this does

format long e
d2fdx2/(-sin(1))
```

You may be wondering how to choose the step size h. This is a little complicated; take a course on numerical analysis and you can see how it's done. But until you do, here's a rough rule of thumb. If $f(x)$ changes significantly over an interval in x of about L, approximate the first derivative of $f(x)$ using $h = 10^{-5}L$; to approximate the second derivative use $h = 10^{-4}L$.

If you want to differentiate a function defined by arrays x and f, then the step size is already determined; you just have to live with the accuracy obtained by using $h = \Delta x$, where Δx is the spacing between points in the x array. *Notice that the data must be evenly spaced for the example we are going to give you to work.*

The idea is to approximate the derivative at $x = x_j$ in the array by using the function values f_{j+1} and f_{j-1} like this

$$f'(x_j) \approx \frac{f_{j+1} - f_{j-1}}{2h}. \tag{10.4}$$

This works fine for an N element array at all points from x_2 to x_{N-1}, but it doesn't work at the endpoints because you can't reach beyond the ends of the array to find the needed

values of f. So we use this formula for x_2 through x_{N-1}, then use linear extrapolation to find the derivatives at the endpoints, like this

Example 10.1a (ch10ex1a.m)

```
% Example 10.1a (Physics 330)

clear; close all;

dx=1/1000;
x=0:dx:4;
N=length(x);
f=sin(x);

% Do the derivative at the interior points all at once using
% the colon command

dfdx(2:N-1)=(f(3:N)-f(1:N-2))/(2*dx);

% linearly extrapolate to the end points (see the next section)

dfdx(1)=2*dfdx(2)-dfdx(3);
dfdx(N)=2*dfdx(N-1)-dfdx(N-2);

% now plot both the approximate derivative and the exact
% derivative cos(x) to see how well we did

plot(x,dfdx,'r-',x,cos(x),'b-')

% also plot the difference between the approximate and exact

figure
plot(x,dfdx-cos(x),'b-')
title('Difference between approximate and exact derivatives')
```

The second derivative would be done the same way. Matlab has its own routines for doing derivatives; look in online help for `diff` and `gradient`.

10.2 Definite Integrals

There are many ways to do definite integrals numerically, and the more accurate these methods are the more complicated they become. But for everyday use the midpoint method usually works just fine, and it's very easy to code. The idea of the midpoint method is to approximate the integral $\int_a^b f(x)dx$ by subdividing the interval $[a, b]$ into N subintervals of width $h = (b-a)/N$ and then evaluating $f(x)$ at the center of each subinterval. We replace $f(x)dx$ by $f(x_j)h$ and sum over all the subintervals to obtain an approximate integral. This

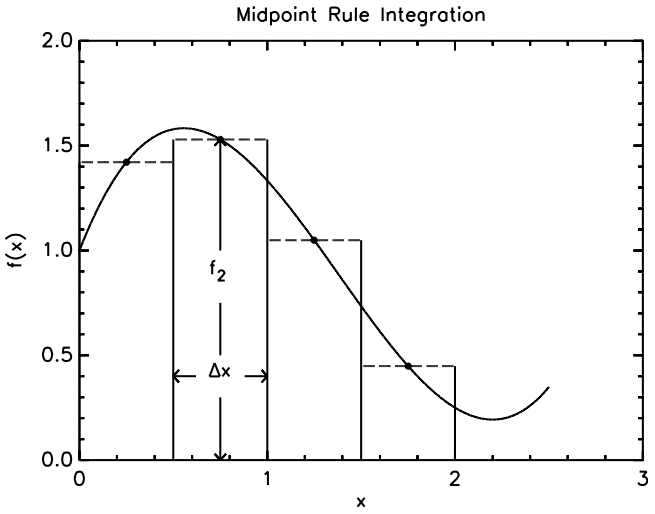


Figure 10.2 The midpoint rule works OK if the function is nearly straight across each interval.

method is shown in Fig. 10.2. Notice that this method should work pretty well over subintervals like [1.0,1.5] where $f(x)$ is nearly straight, but probably is lousy over subintervals like [0.5,1.0] where the function curves.

Example 10.2a (ch10ex2a.m)

```
% Example 10.2a (Physics 330)

close all;
N=1000;
a=0;
b=5;
dx=(b-a)/N;
x=.5*dx:dx:b-.5*dx;  % build an x array of centered values
f=cos(x);  % load the function

% do the approximate integral

s=sum(f)*dx

% compare with the exact answer, which is sin(5)

err=s-sin(5)
```

If you need to do a definite integral in Matlab, this is an easy way to do it. And to see how accurate your answer is, do it with 1000 points, then 2000, then 4000, etc., and watch which decimal points are changing as you go to higher accuracy. (A function that does this for you is given in Chapter 12.)

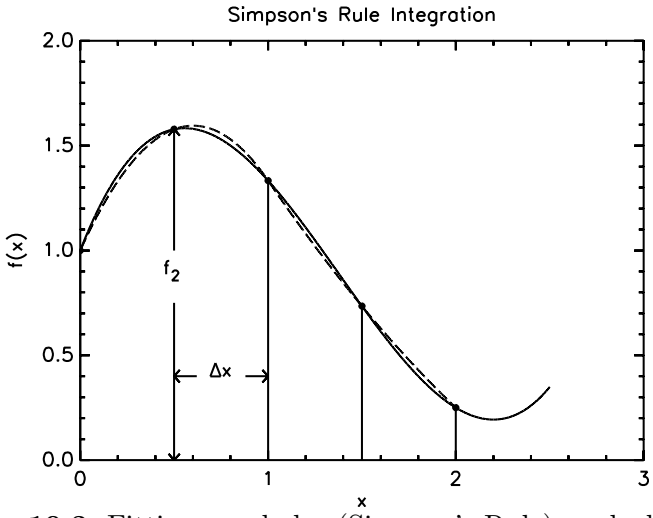


Figure 10.3 Fitting parabolas (Simpson's Rule) works better.

And if you need to find the indefinite integral, in Chapter 12 there is a piece of code that will take arrays of $(x, f(x))$ and calculate the indefinite integral function $\int_a^x f(x')dx'$.

10.3 Matlab Integrators

Matlab also has its own routines for doing definite and indefinite integrals using both data arrays and M-files; look in online help for the commands `trapz` (works on arrays, not functions), `cumtrapz`, `quad`, `quadl`, and `dblquad`, or in *Mastering Matlab 6*, Chapter 23. As an example, the Matlab routine `quad` approximates the function with parabolas (as shown in Fig. 10.3) instead of the rectangles of the midpoint method. This parabolic method is called Simpson's Rule. As you can see from Fig. 10.3, parabolas do a much better job, making `quad` a standard Matlab choice for integration. As an example of what these routines can do, here is the way you would use `quad` to integrate $\cos xe^{-x}$ from 0 to 2:

Note that these integration routines need function M-files, as `fzero` did. These will be discussed more fully in Chapter 12, so for now just use `fint.m`, given below, as a template for doing integrals with `quad`, `quadl`, etc.

Example 10.3a (fint.m)

```
% Example 10.3a (Physics 330)

% define the function to be integrated in fint.m

function f=fint(x)

%*********************************************************
% Warning: Matlab will do this integral with arrays of x,
% so use .*, ./, .^, etc. in this function.  If you forget
% to use the .-form you will encounter the error:
```

```
%
%       Inner matrix dimensions must agree.
%
%*********************************************************

f=cos(x).*exp(-x);
```

Example 10.3b (ch10ex3b.m)

```
% Example 10.3a (Physics 330)

% once fint.m is stored in your current directory
% you can use the following commands to integrate.

% simple integral, medium accuracy
quad(@fint,0,2)

% integrate with specified relative accuracy
quad(@fint,0,2,1e-8)

% integrate with specified relative accuracy
% using quadl (notice that quadl is faster--
% always try it first)

quadl(@fint,0,2,1e-8)
```

Or you can use an `inline function` like this to avoid making another file:

Example 10.3c (ch10ex3c.m)

```
% Example 10.3b (Physics 330)

f=inline('exp(-x).*cos(x)','x')

quadl(f,0,2,1e-8)
```

And if parabolas are good, why not use cubics, quartics, etc. to do an even better job? Well, you can, and Matlab does. The `quadl` integration command used in the example above uses higher order polynomials to do the integration and is the best Matlab integrator to use.

Matlab also has a command `dblquad` that does double integrals. Here's how to use it.

Example 10.3d (f2int.m)

```
% Example 10.3d (Physics 330)

% First define the integrand as a function of x and y
```

```
function f=f2int(x,y)

   f=cos(x*y);
```

Example 10.3e (ch10ex3e.m)

```
% Example 10.3e (Physics 330)

%*********************************************************
% This is how to obtain the double integral over
% the xy rectangle (0,2)X(0,2).  It runs lots
% faster if you use the 'quadl' option, as shown below
%*********************************************************

dblquad(@f2int,0,2,0,2,1e-10,'quadl')
```

Chapter 11

Interpolation and Extrapolation

Since Matlab only represents functions as arrays of values a common problem that comes up is finding function values at points not in the arrays. Finding function values between data points in the array is called *interpolation*; finding function values beyond the endpoints of the array is called *extrapolation*. A common way to do both is to use nearby function values to define a polynomial approximation to the function that is pretty good over a small region. Both linear and quadratic function approximations will be discussed here.

11.1 Linear Interpolation and Extrapolation

A linear approximation can be obtained with just two data points, say (x_1, y_1) and (x_2, y_2). You learned a long time ago that two points define a line and that the two-point formula for a line is

$$y = y_1 + \frac{(y_2 - y_1)}{(x_2 - x_1)}(x - x_1) \tag{11.1}$$

This formula can be used between any two data points to linearly interpolate. For example, if x in this formula is half-way between x_1 and x_2 at $x = (x_1 + x_2)/2$ then it is easy to show that linear interpolation gives the obvious result $y = (y_1 + y_2)/2$.

But you must be careful when using this method that your points are close enough together to give good values. In Fig. 11.1, for instance, the linear approximation to the curved function represented by the dashed line "a" is pretty poor because the points $x = 0$ and $x = 1$ on which this line is based are just too far apart. Adding a point in between at $x = 0.5$ gets us the two-segment approximation "c" which is quite a bit better. Notice also that line "b" is a pretty good approximation because the function doesn't curve much.

This linear formula can also be used to extrapolate. A common way extrapolation is often used is to find just one more function value beyond the end of a set of function pairs equally spaced in x. If the last two function values in the array are f_{N-1} and f_N, it is easy to show that the formula above gives the simple rule

$$f_{N+1} = 2f_N - f_{N-1} \tag{11.2}$$

which was used in the Matlab code in Sec. 10.1

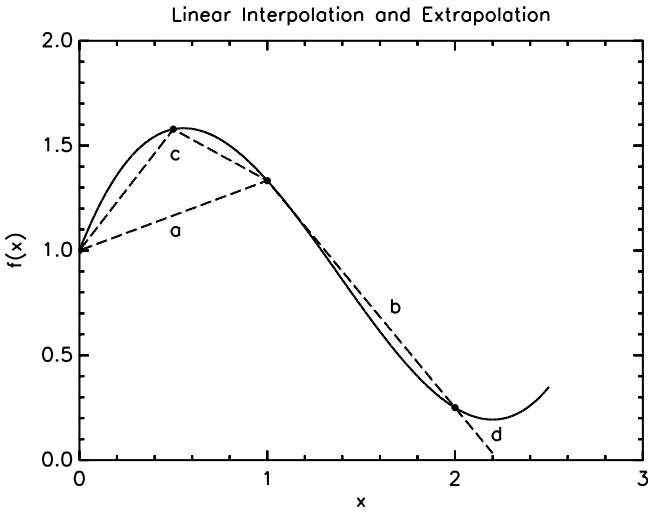


Figure 11.1 Linear interpolation only works well over intervals where the function is straight.

You must be careful here as well: segment "d" in Fig. 11.1 is the linear extrapolation of segment "b", but because the function starts to curve again "d" is a lousy approximation unless x is quite close to $x = 2$.

11.2 Quadratic Interpolation and Extrapolation

Quadratic interpolation and extrapolation are more accurate than linear because the quadratic polynomial $ax^2 + bx + c$ can more easily fit curved functions than the linear polynomial $ax + b$. Consider Fig. 11.2. It shows two quadratic fits to the curved function. The one marked "a" just uses the points $x = 0, 1, 2$ and is not very accurate because these points are too far apart. But the approximation using $x = 0, 0.5, 1$, marked "b", is really quite good, much better than a two-segment linear fit using the same three points would be.

Unfortunately, the formulas for quadratic fitting are more difficult to derive. (The Lagrange interpolation formulas, which you can find in most elementary numerical analysis books, give these formulas.) But for equally spaced data in x, Taylor's theorem, coupled with the approximations to the first and second derivatives discussed in the section on numerical derivatives, make it easy to derive and use quadratic interpolation and extrapolation. We want to do it this way because it uses the approximate derivative formulas we used in Sec.$\tilde{1}$0.1 and illustrates a technique which is widely used in numerical analysis.

You may recall Taylor's theorem that an approximation to the function $f(x)$ near the point $x = a$ is given by

$$f(x) \approx f(a) + f'(a)(x - a) + \frac{1}{2}f''(a)(x - a)^2 + \cdots \tag{11.3}$$

Let's use this approximation (ignoring all terms beyond the quadratic term in $(x - a)$) near a point (x_n, f_n) in an array of equally spaced x values. The grid spacing in x is h.

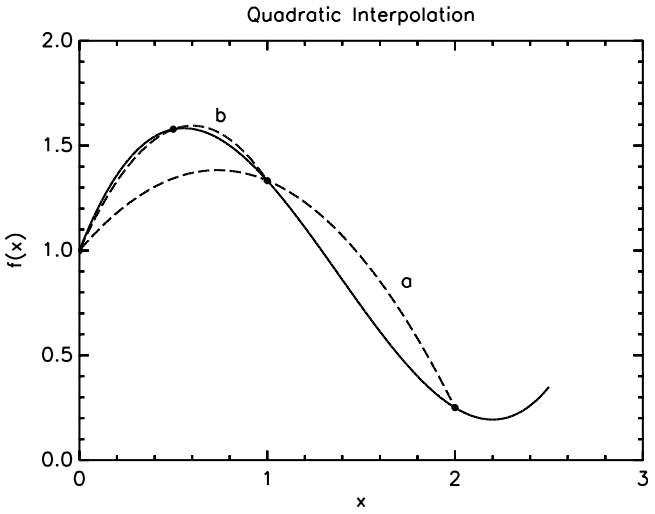


Figure 11.2 Quadratic interpolation follows the curves better if the curvature doesn't change sign.

An approximation to Taylor's theorem that uses numerical derivatives in this array is then given by

$$f(x) \approx f_n + \frac{f_{n+1} - f_{n-1}}{2h}(x - x_n) + \frac{f_{n-1} - 2f_n + f_{n+1}}{2h^2}(x - x_n)^2 \ . \tag{11.4}$$

This formula is very useful for getting function values that aren't in the array. For instance, it is easy to use this formula to obtain the interpolation approximation to $f(x_n + h/2)$

$$f_{n+1/2} = -\frac{1}{8}f_{n-1} + \frac{3}{4}f_n + \frac{3}{8}f_{n+1} \tag{11.5}$$

and also to find the quadratic extrapolation rule

$$f_{N+1} = 3f_N - 3f_{N-1} + f_{N-2} \ . \tag{11.6}$$

11.3 Interpolating With `polyfit` and `polyval`

You can also use Matlab's polynomial commands (which you learned about in Chapter 8) to build an interpolating polynomial. Here is an example of how to use them to find a 5th-order polynomial fit to a crude representation of the sine function.

Example 11.3a (ch11ex3a.m)

```
% Example 11.3a (Physics 330)

clear; close all;
% make the crude data set with dx too big for
% good accuracy

dx=pi/5;
```

```
x=0:dx:2*pi;

y=sin(x);

% make a 5th order polynomial fit to this data

p=polyfit(x,y,5);

% make a fine x-grid

xi=0:dx/20:2*pi;

% evaluate the fitting polynomial on the fine grid

yi=polyval(p,xi);

% and display the fit, the data, and the exact sine function

plot(x,y,'b*',xi,yi,'r-',xi,sin(xi),'c-')
legend('Data','Fit','Exact sine function')

% display the difference between the polynomial fit and
% the exact sine function
figure
plot(xi,yi-sin(xi),'b-')
title('Error in fit')
```

11.4 Matlab Commands Interp1 and Interp2

Matlab has its own interpolation routine `interp1` which does the things discussed in this section automatically. Suppose you have a set of data points $\{x, y\}$ and you have a different set of x-values $\{x_i\}$ for which you want to find the corresponding $\{y_i\}$ values by interpolating in the $\{x, y\}$ data set. You simply use any one of these three forms of the `interp1` command:

```
yi=interp1(x,y,xi,'linear')
yi=interp1(x,y,xi,'cubic')
yi=interp1(x,y,xi,'spline')
```

Here is an example of how each of these three types of interpolation works on a crude data set representing the sine function.

Example 11.4a (ch11ex4a.m)

```
% Example 11.4a (Physics 330)

clear; close all;
```

```
% make the crude data set with dx too big for
% good accuracy

dx=pi/5; x=0:dx:2*pi; y=sin(x);

% make a fine x-grid

xi=0:dx/20:2*pi;

% interpolate on the coarse grid to
% obtain the fine yi values

% linear interpolation

yi=interp1(x,y,xi,'linear');

% plot the data and the interpolation

plot(x,y,'b*',xi,yi,'r-')
title('Linear Interpolation')

% cubic interpolation

yi=interp1(x,y,xi,'cubic');

% plot the data and the interpolation
figure
plot(x,y,'b*',xi,yi,'r-')
title('Cubic Interpolation')

% spline interpolation

yi=interp1(x,y,xi,'spline');

% plot the data and the interpolation
figure
plot(x,y,'b*',xi,yi,'r-')
title('Spline Interpolation')
```

Matlab also knows how to do 2-dimensional interpolation on a data set of $\{x, y, z\}$ to find approximate values of $z(x, y)$ at points $\{x_i, y_i\}$ which don't lie on the data points $\{x, y\}$. You could use any of the following

```
zi = interp2(x,y,z,xi,yi,'linear')
zi = interp2(x,y,z,xi,yi,'cubic')
zi = interp2(x,y,z,xi,yi,'spline')
```

This will work fine for 1-dimensional data pairs $\{x_i, y_i\}$, but you might want to do this interpolation for a whole bunch of points over a 2-d plane, then make a surface plot of

the interpolated function $z(x, y)$. Here's some code to do this and compare these three interpolation methods (linear, cubic, and spline).

Example 11.4b (ch11ex4b.m)

```
% Example 11.4b (Physics 330)

clear; close all;

x=-3:.4:3; y=x;

% build the full 2-d grid for the crude x and y data
% and make a surface plot

[X,Y]=meshgrid(x,y);
Z=cos((X.^2+Y.^2)/2);
surf(X,Y,Z);
title('Crude Data')

%*********************************************************
% now make a finer 2-d grid, interpolate linearly to
% build a finer z(x,y) and surface plot it.

% Note that because the interpolation is linear the mesh is finer
% but the crude corners are still there
%*********************************************************

xf=-3:.1:3;
yf=xf;
[XF,YF]=meshgrid(xf,yf);
ZF=interp2(X,Y,Z,XF,YF,'linear');
figure
surf(XF,YF,ZF);
title('Linear Interpolation')

%*********************************************************
% Now use cubic interpolation to round the corners.  Note that there is
% still trouble near the edge because these points only have data on one
% side to work with, so interpolation doesn't work very well
%*********************************************************

ZF=interp2(X,Y,Z,XF,YF,'cubic');
figure
surf(XF,YF,ZF);
title('Cubic Interpolation')

%*********************************************************
% Now use spline interpolation to also round the corners and see how
% it is different from cubic.  You should notice that it looks better,
% especially near the edges.  Spline interpolation is often the
% best.
%*********************************************************
```

```
ZF=interp2(X,Y,Z,XF,YF,'spline');
figure
surf(XF,YF,ZF);
title('Spline Interpolation')
```

For more detail see *Mastering Matlab 6*, Chapter 19.

Chapter 12

Make Your Own Functions: Inline and M-files

As you use Matlab to solve problems you will probably want to build your own Matlab functions. You can do this either by putting simple expressions into your code by using Matlab's `inline` command, or by defining function files with .m extensions called M-files.

12.1 Inline Functions

Matlab will let you define expressions inside a script for use as functions *within that script only*. For instance, if you wanted to use repeated references to the function

$$f(x,y) = \frac{\sin(xy)}{x^2+y^2} \tag{12.1}$$

you would use the following syntax (to make both a line plot in x with $y = 2$ and to make a surface plot):

Example 12.1a (ch12ex1a.m)

```
% Example 12.1a

clear;close all;
f=inline('sin(x.*y)./(x.^2+y.^2)','x','y');

x=-8:.1:8;y=x;

plot(x,f(x,2))

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

The expression that defines the function is in the first string argument to `inline` and the other string entries tell `inline` which argument goes in which input slot when you use the function.

12.2 M-file Functions

M-file functions are subprograms stored in text files with .m extensions. A function is different than a script in that the input parameters it needs are passed to it with argument lists like Matlab commands (think about `sin(x)` or `plot(x,y,'r-')`). *Note, however, that the variables inside Matlab functions are invisible in the command window.* So to debug a function you need to use print and plot commands in the function file. Or you can make it a stand-alone script by commenting out the function line so that its variables are available at the Matlab command level.

You can also pass information in and out of functions by using Matlab's `global` command to declare certain variables to be visible in all Matlab routines in which the global command appears. For instance, if you put the command

```
global a b c;
```

both in a script that calls `derivs.m` (see below) and in `derivs.m` itself, then if you give a, b, and c values in the main script, they will also have these values inside `derivs.m`. This construction will be especially useful when we use Matlab's differential equation solving routines in Chapter 16.

Rather than give you a syntax lecture we will just give you three useful functions as examples, with comments about how they work.

12.3 Derivative Function `derivs.m`

The first is a function called `derivs.m` which takes as input a function array y representing the function $y(x)$, and dx the x-spacing between function points in the array. It returns `yp` and `ypp`, numerical approximations to the first and second derivatives, as discussed in the section on numerical differentiation. First we will give you the script, then we will show you how to use it.

Example 12.3a (derivs.m)

```
% Example 12.3a

function [yp,ypp]=derivs(y,dx)

%*********************************************************
% This function numerically differentiates the array y which
% represents the function y(x) for x-data points equally spaced
% dx apart.  The first and second derivatives are returned as
% the arrays yp and ypp which are the same length as the input
% array y.  Either linear or quadratic extrapolation is used
% to load the derivatives at the endpoints.  The user decides
% which to use by commenting out the undesired formula below.
%*********************************************************

% load the first and second derivative arrays at the interior points

N=length(y);
```

```
yp(2:N-1)=(y(3:N)-y(1:N-2))/(2*dx);
ypp(2:N-1)=(y(3:N)-2*y(2:N-1)+y(1:N-2))/(dx^2);

% now use either linear or quadratic extrapolation to load the
% derivatives at the endpoints

% linear
%yp(1)=2*yp(2)-yp(3);yp(N)=2*yp(N-1)-yp(N-2);
%ypp(1)=2*ypp(2)-ypp(3);ypp(N)=2*ypp(N-1)-ypp(N-2);

% quadratic
yp(1)=3*yp(2)-3*yp(3)+yp(4);yp(N)=3*yp(N-1)-3*yp(N-2)+yp(N-3);
ypp(1)=3*ypp(2)-3*ypp(3)+ypp(4);ypp(N)=3*ypp(N-1)-3*ypp(N-2)+ypp(N-3);
```

To use this function you can use the following script

Example 12.3b (ch12ex3b.m)

```
% Example 12.3b (Physics 330)

% First build an array of function values

x=0:.01:10; y=cos(x);

% Then, since the function returns two arrays in the form
% [yp,ypp], you would use it this way:

[fp,fpp]=derivs(y,.01);

% look at the approximate derivatives

plot(x,fp,'r-',x,fpp,'b-')
title('Approximate first and second derivatives')
```

Note that we didn't have to call them `[yp,ypp]` when we used them outside the function in the main script. This is because all variables inside functions are local to these programs and Matlab doesn't even know about them in the command window.

Sorry, we lied–we need to bore you with some syntax because everybody gets confused about the first line in a function program. The syntax of the first line is this

```
function output=name(input)
```

The word `function` is required. `output` is the thing the function passes back to whomever called it and its name is local to the function. If it is a single variable name then the code in the function needs to assign that variable name a value, an array of values, or a matrix of values. If the function returns more than one such result then the names of these results are put in square brackets, as in `derivs.m`. The function returns these results to the assigned variable(s), as in the derivs example:

```
[fp,fpp]=derivs(y,dx);
```

The keyword `name` in the function command line above should be the name of the `.m` file that contains the function code. (You can use other names, but you might drive yourself nuts if you do.) `input` is the argument list of the function. When the function program is called the arguments passed in and the arguments used in the function must match in number and type.

12.4 Definite Integral Function `defint.m`

This function uses the midpoint rule to integrate a function over a chosen interval using a chosen number of integration points. The function to be integrated must be coded in the sub function contained at the end of the function file `defint.m` (Note: Matlab's `quad` and especially `quadl` do the same thing, only better. This is just a simple example to show you how to program.)

Example 12.4a (defint.m)

```
% Example 12.4a (Physics 330)

function s=defint(a,b,N)

%*********************************************************
% this function uses the midpoint rule on N subintervals
% to calculate the integral from a to b of the function
% defined in the sub function at the bottom of this
% function

% load dx and build the midpoint rule x-values
%*********************************************************

dx=(b-a)/N;
x=a+.5*dx:dx:b-.5*dx;

%*********************************************************
% use the function f(x) defined in the sub function below
% to obtain the midpoint approximation to the integral and assign
% the result to s
%*********************************************************

s=sum(f(x))*dx;

% here's the sub function

function y=f(x)

% define the function f(x) and assign it to y

y=cos(x);
```

```
%end defint.m
```

To use it, first edit the file `defint.m` so that the sub function at the bottom of the file contains the function you want to integrate. Then give `defint.m` the integration limits (say $a = 0$ and $b = 1$) and the number of points $N = 1000$ to use in the midpoint rule like this

```
defint(0,1,1000)
```

or

```
s=defint(0,1,1000);
```

In the first case the approximate integral prints on the screen; in the second it doesn't print but is assigned to `s`.

12.5 Indefinite Integral Function `indefint.m`

This function takes an array of function values in `y` and an x-spacing `dx` and returns an approximate indefinite integral function $g(x) = \int_a^x y(x')dx'$. The function values must start at $x = a$ and be defined at the edges of the subintervals of size h rather than at the centers as in the midpoint method. Because of this we have to use the trapezoid rule instead of the midpoint rule. The trapezoid rule says to use as the height of the rectangle on the interval of width h the average of the function values on the edges of the interval:

$$\int_x^{x+h} y(x')dx' \approx \frac{y(x) + y(x+h)}{2} h \tag{12.2}$$

Note that this function does exactly the same thing as Matlab's function `cumtrapz`.

Example 12.5a (indefint.m)

```
% Example 12.5a (Physics 330)

function g=indefint(y,dx)

%*********************************************************
% returns the indefinite integral of the function
% represented by the array y.  y(1) is assumed to
% be y(a), the function value at the lower limit of the
% integration.  The function values are assumed to be
% values at the edges of the subintervals rather than
% the midpoint values.  Hence, we have to use the
% trapezoid rule instead of the midpoint rule:
%
% integral(y(x)) from x to x+dx is (y(x)+y(x+dx))/2*dx

% The answer is returned as an array of values defined
% at the same points as y
```

```
%*********************************************************

% the first value of g(x) is zero because at this first value
% x is at the lower limit so the integral is zero

g(1)=0;

N=length(y);

% step across each subinterval and use the trapezoid area
% rule to find each successive addition to the indefinite
% integral

for n=2:N

   % Trapezoid rule
   g(n)=g(n-1)+(y(n-1)+y(n))*.5*dx;

end
```

Take a minute now and use `derivs.m` and `indefint.m` to make overlaid plots of the function $f(x) = \cos x e^{-x}$, its first and second derivatives, and its indefinite integral $F(x) = \int_0^x f(x')dx'$ on the interval $x = [0, 3]$.

Chapter 13

Fast Fourier Transform (FFT)

13.1 Fourier Analysis

Suppose that you went to a Junior High band concert with a digital recorder and made a recording of Mary Had a Little Lamb. Your ear told you that were a whole lot of different frequencies all piled on top of each other, but perhaps you would like to know exactly what they were. You could display the signal on an oscilloscope, but all you would see is a bunch of wiggles. What you really want is the spectrum: a plot of sound amplitude vs. frequency. If this is what you want, Matlab can give it to you with the `fft` command.

13.2 Matlab's FFT

Your recorder stores the data in digital form, with numerical signal values at equally spaced time intervals. Matlab will read such data files in several formats, but the simplest to understand would just be a text file with two columns, one for time t and one for the signal $f(t)$. Or perhaps it could just be the signal in one column and you know what the time step τ is because you know the sampling rate.

If the 2-column file were called `signal.txt` and if it had two columns Matlab would read it this way:

```
load signal.txt;
% the data is now stored in the variable signal as an Nx2 matrix
% unpack it into t and f(t) arrays

t=signal(:,1);
tau=t(2)-t(1);
f=signal(:,2);
```

Or, if the file only contains one column of data and you know the interval between the data points, then Matlab would read it this way

```
load signal.txt;
f=signal;
N=length(f);
```

```
tau=.001;  % tau is the time interval between the points in f
t=0:tau:(N-1)*tau;
```

Now you have a series of data points (t_j, f_j) equally spaced in time by time interval τ and there are N of them covering a total time from $t = 0$ to $t = t_{\text{final}} = (N-1)\tau$. The Matlab function `fft` will convert the array $f(t)$ into an array $g(\omega)$ of amplitude vs. frequency, like this

```
g=fft(f);
```

If you look at the array g you will find that it is full of complex numbers. It is complex because it stores the phase relationship between frequency components as well as amplitude information, just like the Fourier transform of mathematical analysis. If you wanted to reconstruct the original time series by adding the frequency components back together, you can use the Matlab function `ifft` (inverse `fft`)

```
f=ifft(g);
```

similar to the inverse Fourier transform of mathematical analysis. When we don't care about the phase information contained in the spectrum $g(\omega)$, we can work instead with the so-called *power spectrum* $P(\omega)$, obtained this way:

```
P=abs(g).^2
```

To plot `P` or `g` vs. frequency, we need to associate a frequency with each element of the array (just like we had to associate a time with each element of `f`). Before doing so, however, it would probably be good to remind ourselves of the two types of frequencies: "regular" frequency (measured in Hz, or cycles/second) which we'll denote by ν, and angular frequency $\omega = 2\pi\nu$ (measured in radians/second). Both flavors of frequency are commonly used, and you should make sure that you clearly understand which one you are using in a given problem.

We can find the frequencies associated the elements of the array `g` by analysis of the formulas Matlab uses to compute the `fft` and the `ifft`:

$$\texttt{g = fft(f)} \quad : \qquad g(\omega_{k+1}) = \sum_{j=0}^{N-1} f(t_{j+1}) e^{-i2\pi jk/N}, \quad (k = 0, 1, 2, ..N-1)$$

$$\texttt{f = ifft(g)} \quad : \qquad f(t_{k+1}) = \frac{1}{N} \sum_{j=0}^{N-1} g(\omega_{j+1}) e^{i2\pi jk/N}, \quad (k = 0, 1, 2, ..N-1) \tag{13.1}$$

where t_j gives the time for the j^{th} element of `f` and ω_j give the frequency for the j^{th} element of `g`. Using $j = t_j/\tau$ in the `fft` equation, the exponent becomes $-i(2\pi k/N\tau)t_j$. This form allows us to identify the components in the frequency array as $\nu_k = k/N\tau$ or $\omega_k = 2\pi k/N\tau$. The frequency interval from one point in `g` to the next is then given by

$$\Delta\nu = 1/(N\tau) \qquad \text{or} \qquad \Delta\omega = 2\pi/(N\tau) \tag{13.2}$$

The lowest frequency is 0 and the highest frequency in the array is

$$\nu_{\max} = (N-1)/(N\tau) \qquad \text{or} \qquad \omega_{\max} = 2\pi(N-1)/(N\tau). \tag{13.3}$$

The frequency array ν (in cycles per second) and the ω array (in radians per second) would be built this way:

```
N=length(f);

% build v (regular frequency, cycles/sec)
dv=1/(N*tau);
v=0:dv:1/tau-dv;

% build w (angular frequency, radians/sec)
dw=2*pi/(N*tau);
w=0:dw:2*pi/tau-dw;
```

An important thing to notice from the definition (13.2) of the frequency step size $\Delta\omega$ is that if you want to distinguish between two frequencies ω_1 and ω_2 in a spectrum, then you must have $\Delta\omega \ll |\omega_2 - \omega_1|$. You can refine this resolution in frequency by choosing a large value for the length of the time series, $t_{\text{final}} = N\tau$. In addition, from Eq. (13.3) we can see that the maximum frequency you can detect is[1] $\omega_{\max} \approx 2\pi/\tau$, so if you want to see high frequencies you need a small time step τ. Since the time step τ often needs to be tiny (so that $\omega_{\max}$ is big enough), and data the taking time t_{final} often needs to be long (so that $\Delta\omega$ is small enough) you usually just need lots and lots of points, maybe even more than your computer memory will hold. So you need to design your data-taking carefully to capture the features you want to see without breaking your computer.

You will find that all data sets are not created equal when `fft` is applied to them. Sometimes `fft` will give you the answer really fast and sometimes it will be slow. You can make it run at optimum speed if you always give it data sets with 64, 128, 1024, 16384, etc. (powers of 2) data points in them. (Use `help fft` to see how to give `fft` a second argument so powers of 2 are always used.)

Here is an example to show how this process works.

Example 13.2a (ch13ex2a.m)

```
% Example 13.2a (Physics 330)

% Build a time series made up of 5 different frequencies
% then use fft to display the spectrum

clear; close all;

N=2^14;
tau=6000/N;
t=0:tau:(N-1)*tau;

% Make a signal consisting of angular frequencies
% w=1, 3, 3.5, 4, and 6
f=cos(t)+.5*cos(3*t)+.4*cos(3.5*t)+.7*cos(4*t)+.2*cos(6*t);
```

[1] The maximimum frequency you can reliably detect is actually only half of this value due to aliasing.

```
% the time plot is very busy and not very helpful
plot(t,f)
title('Time Series')

% now take the fft and display the power spectrum
g=fft(f);
P=abs(g).^2;
dw=2*pi/(N*tau);
w=0:dw:2*pi/tau-dw;

figure
plot(w,P)
xlabel('\omega')
ylabel('P(\omega)')
title('Power Spectrum, including aliased points')

%*********************************************************
% Notice that the right side of this plot is a mirror
% image of the left side.  This is called aliasing,
% and you can often ignore it by using the axis command
% to only look at half of the spectrum as follows:
%*********************************************************

figure
plot(w,P)
xlabel('\omega')
ylabel('P(\omega)')
title('Power Spectrum, without aliased points')
axis([0 max(w)/2,0 max(P)]);
```

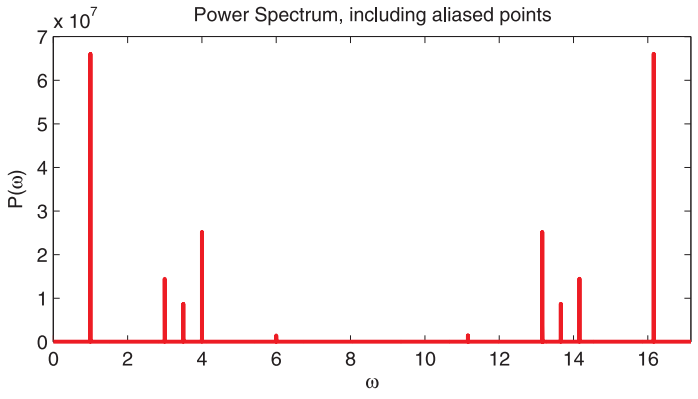


Figure 13.1 Plot of the power spectrum from Example 13.2a

The power spectrum in this example (Fig. 13.1) has peaks at the ω's where they should be, [1,3,3.5,4,6], but there are some extra peaks on the right side. These extra peaks are due to a phenomenon called aliasing, and it comes up all the time in FFTs so we'll take a minute to discuss it in more detail.

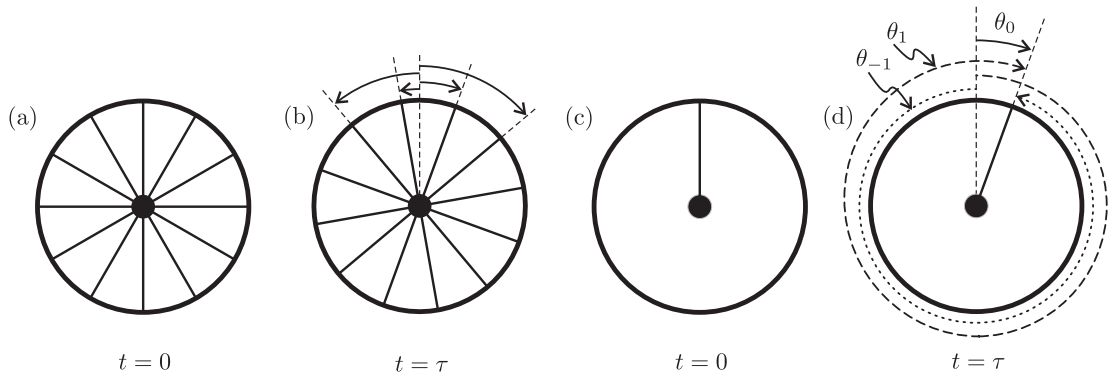


Figure 13.2 Aliasing results from sampling a signal at discrete time intervals. The ambiguity of the wheel rotation between (a) and (b) is due in part the the many indistinguishable spokes. In (c) and (d) we can see that ambiguity still remains with one spoke.

13.3 Aliasing

If you have watched an old western movie, you may have noticed that stagecoach wheels sometimes appear to be turning backwards. This phenomenon is due to the fact that movies are made by taking a bunch of pictures separated by a defined time interval. We know the wheel rotates between pictures because the spokes are oriented at different angles. But since all the spokes look the same, there are many possible rotations that are consistent with the new spoke angles, as depicted in Figs. 13.2(a) and (b). By looking at the figure, you should be able to convince yourself that there are an infinite number of rotations (both positive and negative) that are consistent with the spoke orientation in the two pictures. This ambiguity of frequency is referred to as *aliasing*, and it comes up whenever you study the frequency content of a signal that has been sampled at discrete times. Our brains generally resolve the ambiguity by picking the rotation with the smallest magnitude (whether the sign is positive or negative), which accounts for the stagecoach effect.

To get rid of complications due to the number of spokes in a wheel, imagine that we painted one of the spokes bright red, and then sampled its position at $t = 0$ and $t = \tau$ as shown in Figs. 13.2(c) and (d). The figure shows three of the possible angles that the wheel may have rotated through during this time interval: θ_0, $\theta_1 = \theta_0 + 2\pi$, and $\theta_{-1} = \theta_0 - 2\pi$. These angles of rotation correspond to three possible frequencies:

$$\begin{aligned} \omega_0 &= \theta_0/\tau \\ \omega_1 &= \theta_1/\tau = \omega_0 + 2\pi/\tau, \\ \omega_{-1} &= \theta_{-1}/\tau = \omega_0 - 2\pi/\tau. \end{aligned} \tag{13.4}$$

With some careful study, we can write down a general expression for all frequencies that are consistent with the wheel orientation at the two times:

$$\omega_n = \omega_0 + n2\pi/\tau. \qquad (\text{integer } n) \tag{13.5}$$

Here n is an integer, with $n \geq 0$ corresponding to positive frequencies and $n < 0$ corresponding to negative frequencies. Notice that every frequency can be "aliased" by neigh-

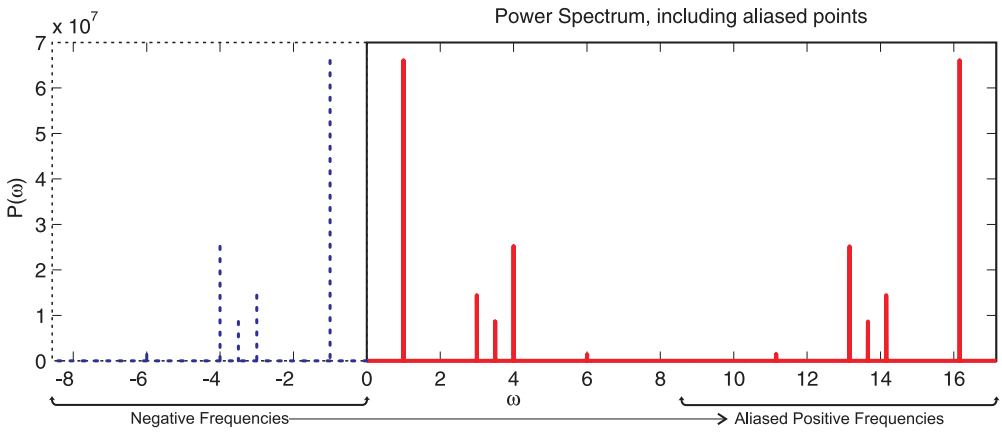


Figure 13.3 Plot of the power spectrum from Example 13.2a showing the negative frequencies that are aliased as positive frequencies.

bor frequencies located $2\pi/\tau$ above or below the frequency of interest. If we recall from Eq. (13.3) that $2\pi/\tau$ is the width of the `fft` frequency window, we realize that any aliasing we observe in the `fft` window comes from frequencies outside the frequency window defined by Eq. (13.3).

We can understand the extra peaks in Fig. 13.1 by noting that the Fourier transform of any real signal has an equal amount of positive and negative frequency content, i.e. $P(\omega) = P(-\omega)$.[2] Since our original frequency window included only positive frequencies, we didn't see the negative half of the spectrum. In Fig. 13.3 we've shown the negative frequency content of the signal from Example 13.2a in dashed lines, and the `fft` power spectrum in solid lines. Notice that each of the "extra" peaks on the right side of the spectrum are a result of an aliased negative-frequency peak located $2\pi/\tau$ below the peak. In the time domain, the negative frequency and its positive alias produce signals that have the same value at each sampling time, as shown in Fig. 13.4.

Since the aliased negative frequencies will always be present in the `fft` window (for real signals), we can only reliably detect frequency components with an absolute value less than

$$\nu_c = \frac{1}{2\tau} \qquad \text{or} \qquad \omega_c = \frac{\pi}{\tau} \tag{13.6}$$

This important limiting frequency is called the critical frequency or the Nyquist frequency. If the signal contains frequency components outside this range, the aliased frequencies will spill over into the positive frequencies and we will be unable to distinguish between actual and aliased frequency peaks. Thus, the highest frequency that you can see without aliasing trouble is ν_c (or ω_c).

[2]Some students are bothered by the idea of negative frequency components. A negative ω component behaves just like a positive ω for time-symmetric signals (i.e. $\cos(-\omega t) = \cos(\omega t)$), but has a π phase shift for antisymmetric signals (i.e. $\sin(-\omega t) = -\sin(\omega t)$).

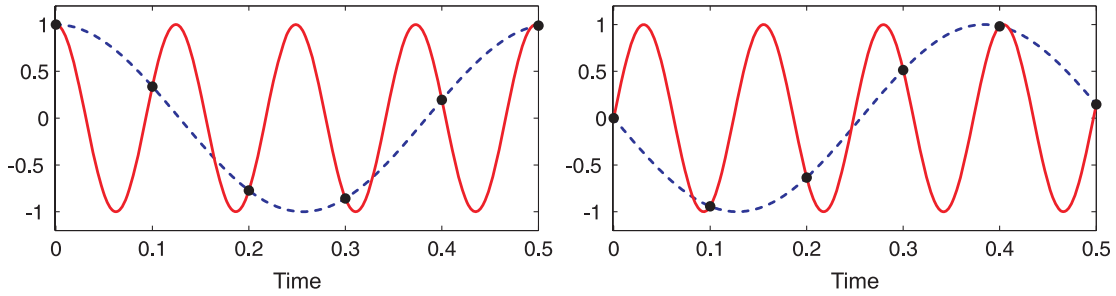


Figure 13.4 The cosine (left) and sine (right) of a negative frequency (dashed) and its aliased positive frequency (solid). In our wagon wheel example, these would be ω_{-1} and ω_0. The dots indicate times at which the signal is sampled. There are infinitely many other frequencies that also cross these sampled points, as specified by Eq. (13.5).

13.4 Using the FFT to Compute Fourier Transforms

If you just want to know where the peaks in a spectrum occur, you can take the `fft`, lop off the right half of the spectrum (after you have checked to make sure the aliased frequencies aren't spilling over), and plot the remainder as was done at the end of Example 13.2a. There are many other uses for the `fft`, however. For instance, it is often useful to numerically calculate the Fourier transform of a signal, work on the spectrum in the frequency domain, and then transform back into the time domain. To make this easier, we need to relate the `fft` and `ifft` to the Fourier transform and its inverse:

$$\begin{aligned} \text{Fourier transform} \quad &: \quad \mathcal{F}[f(t)] = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{i\omega t} dt \\ \text{Inverse Fourier transform} \quad &: \quad \mathcal{F}^{-1}[g(\omega)] = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(\omega) e^{-i\omega t} d\omega, \end{aligned} \tag{13.7}$$

We should warn you up front that these forms are not universally used. For instance, you can put a factor of $1/2\pi$ on one transform rather than a factor of $1/\sqrt{2\pi}$ on both. Physicists frequently use the form in Eq. (13.7) because it makes an energy conservation theorem known as Parseval's theorem more transparent. Also, it is arbitrary which equation is called the transform and which is the inverse transform (i.e., you can switch the minus signs in the exponents). We prefer the convention shown, because the inverse transform can then be used to cleanly represent a sum of traveling waves of the form $e^{i(kx-\omega t)}$. The other sign convention is also mathematically permissible, and often used. You should make sure you clearly understand the conventions you are using, so you don't have factors of 2π floating around or time running backward in your models!

The `fft` and `ifft` functions in Eq. (13.1) are related to the the forms in Eq. (13.7), but there are several differences that need to be addressed: (1) The `fft` is a sum with no normalization, so the height of your peaks scales with N (the number of points you sample). (2) The `fft` has a negative exponent and (in our convention) the Fourier transform has a positive exponent. (3) The `fft` aliases negative frequency components to positive frequencies as discussed in the previous section. To address the first issue, we just need some judiciously multiplication by factors of N and 2π. The second issue can be addressed

by using the `ifft` to calculate the Fourier transform and the `fft` to calculate the inverse. Finally, to address the aliasing issue, we take the aliased positive frequencies, and put them back where they belong as negative frequencies (Matlab uses the function `fftshift` to do this). When you want to take the inverse Fourier transform, you have to put the negative frequencies back where the Matlab functions expect them to be. When you put all of this together, it looks like this:

Example 13.4a (ft.m)

```
% function to calculate the Fourier Transform of the time series St
function fourtran = ft(St,dt)

  fourtran = length(St)*fftshift(ifft(ifftshift(St)))*dt/sqrt(2*pi);

return
```

Example 13.4b (ift.m)

```
% function to calculate the inverse Fourier Transform of the frequency series Sw
function invfourtran = ift(Sw,dw)

  invfourtran = fftshift(fft(ifftshift(Sw)))*dw/sqrt(2*pi);

return
```

The nesting of functions in these these scripts is not particularly intuitive (so don't spend a lot of time working out the details of the shifting functions), but they get the job done: `ft.m` inputs a time series and a dt (i.e. τ) and spits out a properly normalized (according to our convention) Fourier transform with the negative frequencies at the beginning of the array; `ift.m` inputs a $d\omega$ and a properly normalized frequency spectrum with the negative frequencies at the beginning of the array, and spits out its inverse Fourier transform.

Since we put the negative frequencies at the beginning of the series, the frequency series that goes with the spectrum also needs to be fixed:

```
% Use this if N is even (usually the case since powers of 2 are even)
w = -(N/2)*dw:dw:dw*(N/2-1)

% Use this if N is odd (puts w=0 in the right place)
w = -((N-1)/2)*dw:dw:dw*((N-1)/2);
```

To illustrate how to use these functions, lets use them to analyze the same signal as in Example 13.2a

Example 13.4c (ch13ex4c.m)

```
% Example 13.4c (Physics 330)

clear; close all;
```

```
%*********************************************************
% build a time series made up of 5 different frequencies
% then use ft.m to display the spectrum
%*********************************************************

N=2^14;
tau=6000/N;
t=0:tau:(N-1)*tau;

% Notice that the w array is different than before
dw=2*pi/(N*tau);
w = -(N/2)*dw:dw:dw*(N/2-1);

% Make a signal consisting of angular frequencies
% w=1, 3, 3.5, 4, and 6
f=cos(t)+.5*cos(3*t)+.4*cos(3.5*t)+.7*cos(4*t)+.2*cos(6*t);

% Use our new function to calculate the fourier transform
% which needs to be saved as ft.m
g = ft(f,tau);
P = abs(g).^2;

figure
plot(w,P)
xlabel('\omega')
ylabel('P(\omega)')
title('Power Spectrum with peaks at all the right frequencies')

% Now lets plot a normalized spectrum to compare the relative heights
% of the various peaks

figure
plot(w,P./max(P))
xlabel('\omega')
ylabel('Normalized P(\omega)')
title('Normalized Power Spectrum to compare peak height')
```

Now we have frequency peaks at all the right frequencies. Also note that the peak heights don't scale with N any more. The amplitude of peaks in a Fourier transform still tends to be large, however, because instead of amplitude, it is amplitude density (amplitude per unit frequency, or amplitude squared per unit frequency in the case of a power spectrum). So if the signal is confined to a tiny range in ω, its density will be huge.

The relative peak heights are similar to what they should be, but zoom in closely on the normalized power spectrum and you will see that they are not exactly the right size. (Power is proportional to amplitude squared, so the peaks should be in the ratio [1,0.25,0.16,0.49,0.04].) This is due to the discrete sampling nature of the data in an `fft`.

To understand why the relative sizes are off, lets take the Fourier transform of one of

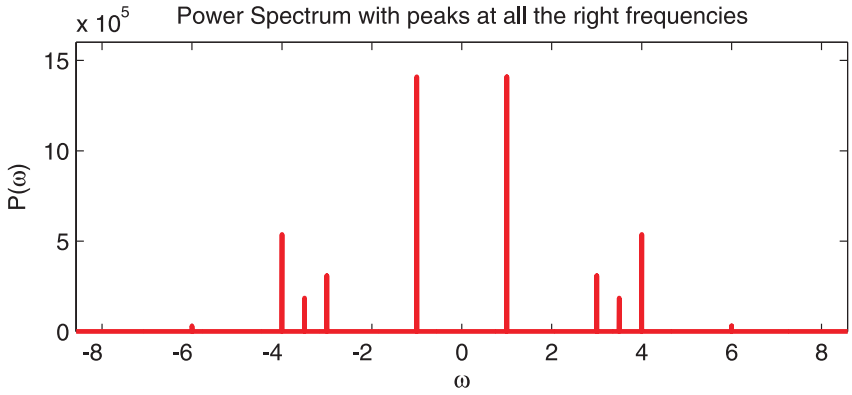


Figure 13.5 Plot of the power spectrum from Example 13.4c

the frequency components in our signal analytically:

$$\mathcal{F}\left[\cos(\omega_0 t)\right] = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \cos(\omega_0 t) e^{i\omega t} dt = \sqrt{\frac{2}{\pi}} \left(\delta(\omega + \omega_0) + \delta(\omega - \omega_0)\right) \ . \tag{13.8}$$

Yes, those are delta-functions located at $\omega = \pm\omega_0$, and they are infinite (but they have finite area.) So when Matlab does the `fft` on periodic data, the result is a bunch of approximate delta functions with very narrow widths and large amplitudes. However, since our frequency array doesn't have, for example, a point exactly at $\omega = 3$ where our signal should have a delta function, the heights of the approximate delta function peaks are not exactly correct. Another way of stating this is that only if the total sampling time happens to be an exact multiple of the periods involved will you get correct amplitudes for periodic signals. You can get around this problem using a technique called *windowing*, where you artificially confine your signal in the time domain which makes your peaks broader in the frequency domain. With broader peaks the height isn't as sensitive to where your data points fall in relation to the exact center of the peaks.

We've only scratched the surface of what you can do with numerical Fourier transforms, but its time to move on. For more details see online help or *Mastering Matlab 6*, Chapter 21.

Chapter 14

Fitting Functions to Data

A common problem that physicists often encounter is to find the best fit of a function (with a few variable parameters) to a set of data points. If you are fitting to a polynomial, then the easiest way to do this is with `polyfit`. But if you want to fit to a sine, a cosine, an exponential, a log, etc., then there is no simple command available. Pay attention to this section; it is useful.

14.1 `fminsearch`

Matlab has a very nice multidimensional minimizer routine called `fminsearch` that will do fits to a general function if you give it a half-decent initial guess for the fitting parameters. To see how to use `fminsearch` generally you can use online help, but here we will just show you how to use it for function fitting.

Suppose we have a set of data points (x_j, y_j) (perhaps read in with Matlab's `load` command as discussed in the Fast Fourier Transform section) and a proposed fitting function of the form $y = f(x, a_1, a_2, a_3, ...)$. For example, we could try to fit to an exponential function with two adjustable parameters a_1 and a_2 as is done in the example in `leastsq.m` below:

$$f(x, a_1, a_2) = a_1 e^{a_2 x} \quad . \tag{14.1}$$

Or you could fit to a cubic polynomial in x^2 with four adjustable parameters a_1, a_2, a_3, a_4 with this f:

$$f(x, a_1, a_2, a_3, a_4) = a_1 + a_2 x^2 + a_3 x^4 + a_4 x^6 \tag{14.2}$$

which is what `polyfit` does.

In any case, what we want to do is choose the parameters $(a_1, a_2, a_3, ...)$ in such a way that the sum of the squares of the differences between the function and the data is minimized, or in mathematical notation we want to minimize the quantity

$$S = \sum_{j=1}^{N} (f(x_j) - y_j)^2 \ . \tag{14.3}$$

The first thing you need to do is to make a Matlab M-file called `leastsq.m` which evaluates the least-squares sum you are trying to minimize. It needs access to your fitting function

$f(x, a)$, which is stored in the Matlab M-file `funcfit.m`. Here are examples of these two files.

This function `leastsq.m` is the one that you give to `fminsearch`. It allows `fminsearch` to talk to your fitting function `funcfit.m` by calculating S from the (x, y) data and the a_n's; it looks like this:

Example 14.1a (leastsq.m)

```
% Example 14.1a (Physics 330)

function s=leastsq(a,x,y)

%************************************************************
% leastsq can be passed to fminsearch to do a
% non-linear least squares fit of the function
% funcfit(a,x) to the data set (x,y).
% funcfit.m is built by the user as described here

% a is a vector of variable parameters; x and y
% are the arrays of data points
%************************************************************

% find s, the sum of the squares of the differences
% between the fitting function and the data

s=sum((y-funcfit(a,x)).^2);
```

Example 14.1b (funcfit.m)

```
% Example 14.1b (Physics 330)

function f=funcfit(a,x)

%************************************************************
% this function evaluates the fitting
% function f(x,a1,a2,a3,...) to be fit to
% data.  It is called by leastsq.

% a is a vector of variable fitting parameters a1, a2, ...
% that are used in the function and x is a
% vector of x-values

% the function returns a vector f=f(x,a1,a2,...)
%************************************************************

% sample exponential function with 2 variable
% parameters

f = a(1)*exp(a(2)*x);
```

With these two functions built and sitting in your Matlab directory we are ready to do the fit. Here is a piece of code that allows you to enter an initial guess for the fitting parameters, plot the initial guess against the data, then tell `fminsearch` to do the least squares fit. The behavior of `fminsearch` can be controlled by setting options with Matlab's `optimset` command. In the code below this command is used to set the Matlab variable `TolX`, which tells `fminsearch` to keep refining the parameter search until the parameters are determined to a relative accuracy of `TolX`. Finally, it plots the best fit against the data. We suggest you give it a name (perhaps `datafit.m`) and save it for future use. The data needs to be sitting in the file `data.fil` as two columns of (x, y) pairs, like this

```
0.0  1.10
0.2  1.20
0.4  1.52
0.6  1.84
0.8  2.20
1.0  2.70
```

Let's try to fit this data to the function $f(x) = a_1 \exp(a_2 x)$. Here's the fitting code

Example 14.1c (ch14ex1c.m)

```
% Example 14.1c (Physics 330)

%*********************************************************
% Uses fminsearch to least squares fit
% a function defined in funcfit.m to
% data read in from data.fil
%*********************************************************

clear;close

% read the data file and load x and y

load data.fil;
x=data(:,1);
y=data(:,2);

% set up for the plot of the fitting function

xmin=min(x);
xmax=max(x);
npts=1001;
dx=(xmax-xmin)/(npts-1);
xplot=xmin:dx:xmax;

% set ifit to 0 and don't continue on to the fit until
% the user sets it to 1

ifit=0;
```

```
while ifit==0

disp(' Enter an initial guess for the function ')
a=input('parameters [a1,a2,...] in vector form [...]- ')

% plot the data and the initial function guess

yplot=funcfit(a,xplot);

plot(x,y,'b*',xplot,yplot,'r-')
xlabel('x')
ylabel('y')
title('Initial Guess and Data')

ifit=input(' Enter 0 to guess again, 1 to try to fit with this guess - ')

end

%*********************************************************
% Do the fit with the option TolX set; fminsearch will adjust a
% until each of its elements is determined to within TolX.
% If you think fminsearch could do better than it did, reduce TolX.
%*********************************************************

option=optimset('TolX',1e-5);
a=fminsearch(@leastsq,a,option,x,y)

% plot the data and the final function fit

yplot=funcfit(a,xplot);

% Plot the final fit and the data

plot(x,y,'b*',xplot,yplot,'r-')
xlabel('x')
ylabel('y')
title('Final Fit and Data')
```

It's a little painful to have to make three files to get this job done, but we suggest you learn how to use `fminsearch` this way. It comes up all the time.

Chapter 15

Systems of Nonlinear Equations

15.1

Matlab's `fminsearch` can also be used to solve systems of nonlinear equations. Consider the following pretty-impossible-looking set of three equations in three unknowns (x, y, z).

$$\begin{aligned} \sin(xy) + \exp(-xz) - 0.95908 &= 0 \\ z\sqrt{x^2+y^2} - 6.70820 &= 0 \\ \tan(y/x) + \cos z + 3.17503 &= 0 \end{aligned} \tag{15.1}$$

The way to talk `fminsearch` into solving this set is to invent a scalar function $S(x, y, z)$ which consists of the sum of the squares of the three functions given above:

$$\begin{aligned} S(x,y,z) &= [\sin(xy) + \exp(-xz) - 0.95908]^2 \\ &\quad + \left[z\sqrt{x^2+y^2} - 6.70820\right]^2 + [\tan(y/x) + \cos z + 3.17503]^2 \ . \end{aligned} \tag{15.2}$$

If you look at $S(x, y, z)$ for a bit you will see that (1) it is always positive, (2) its smallest possible value is zero, and (3) if you can somehow find values of (x, y, z) that make it zero, then you have solved the system of three equations.

Note, however, that `fminsearch` is a *minimizer*, not a zero finder. So it may find a local minimum of $S(x, y, z)$ which does not satisfy $S = 0$. If it fails in this way you need to (1) know about it and (2) make another initial guess so `fminsearch` can take another crack at the problem.

Here are two pieces of Matlab code that will use `fminsearch` to solve systems like this. The first one calls `fminsearch` and the second one, which is a Matlab function (`eqsystem.m`) contains the equations to be solved.

Example 15.1a (ch15ex1a.m)

```
% Example 15.1a (Physics 330)

%*********************************************************
% Uses fminsearch to look for solutions to the
% nonlinear system of equations defined in the
```

```
% file eqsystem.m
%*********************************************************

clear;

itry=0;

while itry ==0

disp(' Enter an initial guess for the solution')
x=input('of the system of equations in the form [x1,x2,...] - ');

% evaluate the scalar function S(x1,x2,...) so the
% user can see how good the guess is

s=eqsystem(x);
fprintf(' For this guess: S(x1,x2,...) = %g \n',s)

itry=input(' Enter 0 to guess again, 1 to try to solve with this guess- ')

end

x=fminsearch(@eqsystem,x)


s=eqsystem(x);
fprintf(' Final value of S(x1,x2,...) = %g \n',s)
disp(' (Make sure it is close to zero)')
```

Example 15.1b (eqsystem.m)

```
% Example 15.1b (Physics 330)

function s=eqsystem(xn)

% nonlinear system of equations routine for
% use with zerofinder and fminsearch

x=xn(1);y=xn(2);z=xn(3);

Eq1 = sin(x*y)+exp(-x*z)-0.95908;
Eq2 = z*sqrt(x^2+y^2) -6.70820;
Eq3 = tan(y/x)+cos(z)+3.17503;

s=Eq1^2+Eq2^2+Eq3^2;
```

When you solve this system and you are asked for an initial guess, try something near (0.8,1.8,2.5).

Chapter 16

Ordinary Differential Equations

The standard way to write a differential equation, or a system of differential equations, in numerical work is as a first order system, like this:

$$\frac{d\mathbf{u}}{dt} = \mathbf{F}(\mathbf{u}) \tag{16.1}$$

where $\mathbf{u}$ is a vector of unknown functions of the parameter t (often time in physics problems) and where $\mathbf{F}(\mathbf{u})$ is a vector-valued function of a vector argument. For a system with three unknown function x, y, and z we would write

$$\frac{d}{dt}\begin{pmatrix} u_1(t) \\ u_2(t) \\ u_3(t) \end{pmatrix} = \begin{pmatrix} f_1(u_1,u_2,u_3) \\ f_2(u_1,u_2,u_3) \\ f_3(u_1,u_2,u_3) \end{pmatrix}. \tag{16.2}$$

where

$$\begin{aligned} u_1 &\equiv x \\ u_2 &\equiv y \\ u_3 &\equiv z \end{aligned} \tag{16.3}$$

This makes perfect sense to a mathematician, but physicists usually need examples. Here are a couple.

16.1 Decay of a Radioactive Sample

If there are N atoms of an unstable element with an exponential decay rate of γ then the differential equation describing how N decreases in time is

$$\frac{dN}{dt} = -\gamma N \tag{16.4}$$

which is just a single first order differential equation whose solution is

$$N(t) = N(0)e^{\gamma t} \quad .$$

In this case $\mathbf{u}$ is the one-element vector with $u_1 = N$ and $\mathbf{F}$ is the one element vector $\mathbf{F}(\mathbf{u}) = -\gamma u_1$

16.2 Simple Harmonic Oscillator

The equation of motion of a mass m bouncing in a weightless environment on a spring with spring constant k is

$$\frac{d^2x}{dt^2} = -\omega_0^2 x \quad \text{where} \quad \omega_0 = \sqrt{\frac{k}{m}} \tag{16.5}$$

which has the fundamental solution

$$x(t) = A\cos\omega_0 t + B\sin\omega_0 t$$

This is a second order differential equation rather than a first order system, so we need to change its form to fit Matlab's format. This is done by using position $x(t)$ and velocity $v(t) = dx/dt$ as two unknown functions of time. The first order set consists of the definition of $v(t)$ in terms of $x(t)$ and the second order differential equation with d^2x/dt^2 replaced by dv/dt:

$$\begin{aligned} \frac{dx}{dt} &= v \\ \frac{dv}{dt} &= -\omega_0^2 x\ . \end{aligned} \tag{16.6}$$

It is always possible to use this trick of defining new functions in terms of derivatives of other functions to convert a high order differential equation to a first order set.

In this case, the vector $\mathbf{u}$ from Eq. (16.1) is

$$\mathbf{u} = \begin{pmatrix} x \\ v \end{pmatrix}. \tag{16.7}$$

In this notation, our system of equations becomes

$$\begin{aligned} \frac{du_1}{dt} &= u_2 \\ \frac{du_2}{dt} &= -\omega_0^2 u_1 \end{aligned} \tag{16.8}$$

So the vector $\mathbf{F}(\mathbf{u})$ is

$$\mathbf{F}(\mathbf{u}) = \begin{pmatrix} u_2 \\ -\omega_0^2 u_1 \end{pmatrix} \tag{16.9}$$

So let's assume that we have a first order set. How can we solve it? A symbolic math program like Mathematica will often give you an analytic expression for the solution; Matlab can only give you an array which is a numerical approximation. But Matlab will find this approximation very quickly and you have control over how it does it, so it is often more efficient to use Matlab for hard differential equations than to use Mathematica.

We are going to show you two methods. The first is simple, intuitive, and inaccurate. The second is a little more complicated, not terribly intuitive, but pretty accurate. There are many ways to numerically solve differential equations and the two we will show you are rather crude; Matlab has its own solvers (discussed in Sec. 16.5) which are better, but you will learn a bit about the ideas they are based on by studying these two methods.

16.3 Euler's Method

The first method is called Euler's Method (say "Oiler's Method"), and even though it's pretty bad, it is the basis for many better methods. Here's the idea.

First, quit thinking about time as a continuously flowing quantity. Instead we will seek the solution at specific times t_n separated by small time steps τ. Hence, instead of $\mathbf{u}(t)$ we will try to find $\mathbf{u}_n = \mathbf{u}(t_n)$. The hope is that as we make τ smaller and smaller we will come closer and closer to the true solution.

Since we are going to use discrete times and since the initial conditions tell us where to start, what we need is a rule that tells us how to advance from $\mathbf{u}_n$ to $\mathbf{u}_{n+1}$. To find this rule let's approximate the differential equation $d\mathbf{u}/dt = \mathbf{F}$ this way

$$\frac{\mathbf{u}_{n+1} - \mathbf{u}_n}{\tau} = \mathbf{F}(\mathbf{u}_n, t_n) \ . \tag{16.10}$$

In doing this we are assuming that our solution is represented as an array of values of both t and $\mathbf{u}$, which is the best that Matlab can do. If we already know $\mathbf{u}_n$, the solution at the present time t_n, then the equation above can give us $\mathbf{u}$ one time step into the future at time $t_{n+1} = t_n + \tau$:

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \mathbf{F}(\mathbf{u}_n, t_n)\tau \ . \tag{16.11}$$

This is a little abstract, so let's use it to approximately solve the harmonic oscillator equation. For this case Matlab would use for $\mathbf{u}$ the vector `[x,v]` and for $\mathbf{F}$ the vector `[v,-w^2*x]`. (Stare at the harmonic oscillator equation given in Eq. (16.6) as a first order system until you can see that this is true.) Here's a script that uses this method to solve the harmonic oscillator equation.

Example 16.3a (ch16ex3a.m)

```
% Example 16.3a (Physics 330)

clear; close all;

% Use Euler's method to solve the harmonic oscillator equation

% set the angular frequency
w=1;

% decide how long to follow the motion, 10 periods in this case
tfinal=2*pi/w*10;

% choose the number of time steps to take
N=input(' Enter the number of time steps to take - ')

% t=zeros(1,N+1);x=zeros(1,N+1);v=zeros(1,N+1); % uncomment this line to make
                                                % the code run faster

% calculate the time step
tau=tfinal/N;
```

```
% initialize the time array
t(1)=0;

% set the initial values of position and velocity
x(1)=1;v(1)=0;

% Do Euler's method for N time steps
for n=1:N
  t(n+1)=n*tau;
  x(n+1)=x(n) + v(n)*tau;
  v(n+1)=v(n) - w^2*x(n)*tau;
end

% plot the result and compare it with the exact solution
% which is x(t)=cos(w*t)
plot(t,x,'r-',t,cos(w*t),'b-')
```

When you copy this code into a file and run it you will see that even if you take 1000 steps, the solution is not very good. No matter how small `tau` is, if you run long enough Euler's method will blow up.

Also note that if you try to run this script for many steps ($N = 50,000$, for instance) it runs slow. The reason is that you keep making the t, x, and v arrays longer and longer in the loop, so Matlab has to allocate additional memory for them in each step. But if you define them ahead of time to be big enough (see the commented line just after the line `N=input...` in the code above), the arrays are defined to be big enough before you start the loop and no time will be wasted increasing the array sizes. Run this script again with the line of code beginning with `t=zeros(1,N+1)...` uncommented and watch how fast it runs, even if you choose $N = 500,000$.

16.4 Second-order Runge-Kutta

Here is a method which is still quite simple, but works a lot better than Euler. But it is, in fact, just a modification of Euler. If you go back and look at how we approximated $d\mathbf{u}/dt$ in Euler's method you can see one thing that's wrong with it: the derivative is not centered in time:

$$\frac{\mathbf{u}_{n+1} - \mathbf{u}_n}{\tau} = \mathbf{F}(\mathbf{u}_n, t_n) \ . \tag{16.12}$$

The left side of this equation is a good approximation to the derivative halfway between t_n and t_{n+1}, but the right side is evaluated at t_n. This mismatch is one reason why Euler is so bad.

Runge-Kutta attempts to solve this centering problem by what looks like a cheat: (1) Do an Euler step, but only by $\tau/2$ so that we have an approximation to `[x,v]` at $t_{n+1/2}$. These half-step predictions will be called `[xhalf,vhalf]`. (2) Then evaluate the function $\mathbf{F}(\mathbf{u}, t)$ at these predicted values to center the derivative

$$\frac{\mathbf{u}_{n+1} - \mathbf{u}_n}{\tau} = \mathbf{F}(\mathbf{u}_{n+1/2}, t_{n+1/2}) \ . \tag{16.13}$$

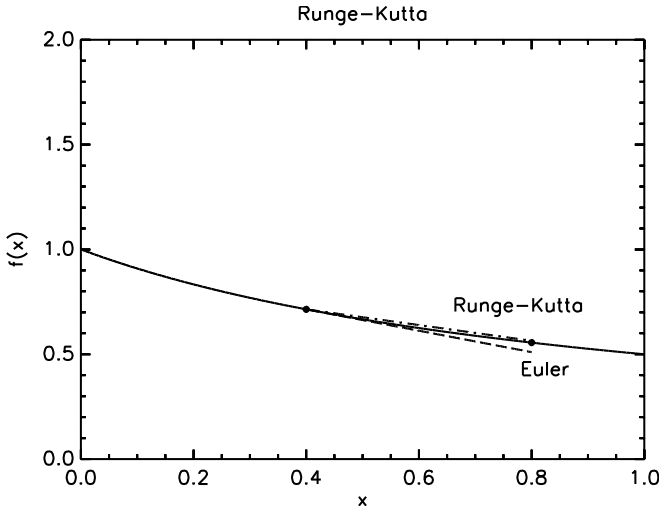


Figure 16.1 Runge-Kutta anticipates curving and beats Euler.

This is the simplest example of a *predictor-corrector* method and it works lots better than Euler, as you can see by running the code given below. (Step (1) above is the predictor; step (2) is the corrector.)

You can see this difference between Runge-Kutta and Euler in Fig. 16.1, where the upward curve of the solution makes Euler miss below, while Runge-Kutta's half-way-out correction to the slope allows it to do a much better job.

Example 16.4a (ch16ex4a.m)

```
% Example 16.4a (Physics 330)

% Runge-Kutta second order approximate solution to the harmonic oscillator
clear;close all;

% set the angular frequency
w=1;

% decide how long to follow the motion, 10 periods in this case
tfinal=2*pi/w*10;

% choose the number of time steps to take
N=input(' Enter the number of time steps to take - ')

% calculate the time step
tau=tfinal/N;

% initialize the time array
t(1)=0;

% set the initial values of position and velocity
x(1)=1;v(1)=0;
```

```
% Do Runge-Kutta for N time steps
for n=1:N
   t(n+1)=n*tau;

   % Predictor step .5*tau into the future
   xhalf=x(n) + v(n)*tau*.5;
   vhalf=v(n) - w^2*x(n)*tau*.5;

   % Corrector step
   x(n+1)=x(n) + vhalf*tau;
   v(n+1)=v(n) - w^2*xhalf*tau;

end

% plot the result and compare it with the exact solution
% x(t)=cos(w*t) and v(t)=-w*sin(w*t)

plot(t,x,'r-',t,cos(w*t),'b-')
```

16.5 Matlab's Differential Equation Solvers

Matlab also has its own differential equation solvers and they are more accurate than the simple methods discussed in Secs. 16.3 and 16.4. Table 16.1 shows a list borrowed from *Mastering Matlab 6* of these Matlab functions and what they do.

Below you will find two sample scripts `odetest` and `rhs` which can use any of these Matlab solvers to solve and plot the solution of the harmonic oscillator equation. Note that they all work in this way:

1. You define the right-hand side function for your set of first order differential equations in the M-file `rhs.m`.

2. You choose the beginning and ending times to pass into the Matlab ode function.

3. You put the initial column vector **u** in the Matlab variable `u0` to define the initial conditions for your problem.

4. You choose the ode solver control options by using Matlab's `odeset` function.

5. You ask Matlab to give you a column of times t and a matrix of **u**-values by calling one of the ode solvers like this

   ```
   [t,u]=ode45(@rhs,[tstart,tfinal],u0,options);
   ```

6. This command returns a column vector `t` of the discrete times between `tstart` and `tfinal` which `ode45` chose to make the solution be as accurate as you asked it to be when you used `odeset`. You will also receive a matrix `u` with as many columns as you have unknowns in your set of ode's and with as many rows as you have times in

`ode23`	An explicit, one-step Runge-Kutta low-order (2-3) solver. (Like the second-order Runge-Kutta method predictor-corrector discussed here in Sec. 16.4.) Suitable for problems that exhibit mild stiffness, problems where lower accuracy is acceptable, or problems where $\mathbf{F}(t, \mathbf{x})$ is not smooth (e.g. discontinuous).
`ode45`	An explicit, one-step Runge-Kutta medium-order (4-5) solver. Suitable for non-stiff problems that require moderate accuracy. *This is typically the first solver to try on a new problem.*
`ode113`	A multi-step Adams-Bashforth-Moulton PECE solver of varying order (1-13). Suitable for non-stiff problems that require moderate to high accuracy involving problems where $\mathbf{F}(t, \mathbf{x})$ is expensive to compute. Not suitable for problems where $\mathbf{F}(t, \mathbf{x})$ is discontinuous or has discontinuous lower-order derivatives.
`ode23s`	An implicit, one-step modified Rosenbrock solver of order 2. Suitable for stiff problems where lower accuracy is acceptable, or where $\mathbf{F}(t, \mathbf{x})$ is discontinuous. *Stiff problems are those in which there are several different rates of change involved whose sizes differ by several orders of magnitude, or more.*
`ode15s`	An implicit, multi-step solver of varying order (1-5). Suitable for stiff problems that require moderate accuracy. *This is typically the solver to try if* `ode45` *fails or is too inefficient.*

Table 16.1 Some of Matlab's differential equation solvers.

`t`. If you make the required accuracy smaller, you will receive more data points. If the position x is called `u(1)` in your set of equations then you can obtain an array containing these positions by extracting the first column, like this

```
x=u(:,1);
```

Important note: The data points will not be equally spaced in time. If you just want the solution at certain pre-set times $[t_n]$, just replace the 2-element array `[tstart,tfinal` with an array of the times that you want: `[t1,t2,t3,t4,...,tN]`. For example, you could replace `[tstart,tfinal]` with the equally spaced array of times `tstart:dt:tfinal`.

Once you have extracted the different components of your solution from `u`, i.e., x, v_x, y, v_y, z, v_z, etc., you can use Matlab's plotting and data analysis capabilities to slice and dice the data anyway you want.

Oh, and what if you want change your data to be equally spaced in time because, for example, you wanted to send the solution off to `fft` to find a spectrum? Just use `interp1` to interpolate the output of `ode45` onto an equally spaced time grid and away you go.

Here's some sample differential equation solving code that solves the simple harmonic oscillator.

Example 16.5a (ch16ex5a.m)

```
% Example 16.5a (Physics 330)

%*********************************************************
% ordinary differential equation solver using
% Matlab's ode solvers and the M-file rhs.m to
% specify F(t,u)
%*********************************************************

clear;close all;

% declare the oscillator frequency to be global and set it
global w0;
w0=1;

% set the initial and final times
tstart=0;tfinal=200;

% set the initial conditions in the y0 column vector
u0=zeros(2,1);
u0(1)=.1; % initial position
u0(2)=0; % initial velocity

% set the solve options
options=odeset('RelTol',1e-8);

% do the solve
   [t,u]=ode45(@rhs,[tstart,tfinal],u0,options);

% unload the solution that comes back in y into x and v arrays
x=u(:,1);v=u(:,2);

%*********************************************************
% because Matlab's ode solvers don't use equally spaced
% time steps, and because you might want equal spacing,
% here's how you convert from Matlab's unequally-spaced (t,x,v)
% to equally spaced data (te,xe,ve)
%*********************************************************

N=length(t);
taue=(tfinal-tstart)/(N-1);
te=tstart + (0:taue:(N-1)*taue) ;
te=te';  % convert te to a column vector, to match t
xe=interp1(t,x,te,'spline');
ve=interp1(t,v,te,'spline');

%*********************************************************
% Note that you could have obtained equally-spaced points by
% telling ode45 to give you the solutions at times you specify.
% For instance, suppose you wanted 1024 points between t=0 and
```

```
% t=200.  You could build them like this (the code is commented):

% N=1024;
% taue=(tfinal-tstart)/(N-1);
% te=tstart + (0:taue:(N-1)*taue) ;
% [t,u]=ode45(@rhs,te,u,options);
% xe=u(:,1);ve=u(:,2);
%*************************************************************


% plot the position vs. time

plot(te,xe)
title('Position vs. Time')

% make a "phase-space" plot of v vs. x
figure
plot(xe,ve)
title('Phase Space Plot (v vs. x)')
```

Example 16.5b (rhs.m)

```
%*************************************************************
% right-hand side function for Matlab's ordinary
% differential equation solvers: simple harmonic
% oscillator example:
%
% It is a good idea to write a comment to remind yourself
% how the variables are arranged in the vector u.
%
% In our case we will use:
%     u(1) -> x
%     u(2) -> v
%
%*************************************************************

function F=rhs(t,u)

% declare the frequency to be global so its value
% set in the main script can be used here
global w0;

% make the column vector F filled
% with zeros
F=zeros(length(u),1);

% Now build the elements of F

% Recall that in our ordering of the vector u we have:
%
```

```
%     du(1)                  dx
%     ----  = F(1)   ->     -- = v
%      dt                   dt
%
% so the equation dx/dt=v means that F(1)=u(2)
F(1)=u(2);

% Again, in our ordering we have:
%
%     du(2)                  dv
%     ----  = F(2)   ->     -- = -w0^2*x
%      dt                   dt
%
% so the equation dv/dt=-w0^2*x means that F(2)=-w0^2*u(1)
F(2)=-w0^2*u(1);
```

16.6 Event Finding with Matlab's Differential Equation Solvers

Something you will want to do with differential equation solvers is to find times and variable values when certain events occur. For instance, suppose we are solving the simple harmonic oscillator and we want to know when the position of the oscillator goes through zero with positive velocity, as well as when the velocity is zero and decreasing. We have good news and bad news. The good news is that Matlab knows a way to do this. The bad news is that the way is a little involved. If you try to figure out how it works from online help you will be confused for a while, so we suggest that you use the example files given below. Read them carefully because I have put all of the explanations about how things work in the codes as comments. The main file is `eventode.m` and its right-hand side function is `eventrhs.m`. There is also an additional M-file to control the event-finding called `events.m`.

Example 16.6a (ch16ex6a.m)

```
% Example 16.6a (Physics 330)

% eventode: example of event finding in Matlab's ode solvers

clear;close;

dt=.01;  % set the time step
u0=[0;1];  % put initial conditions in the [x;vx] column vector

% turn the eventfinder on by specifying the name of the M-file
% where the event information will be processed (events.m)
options=odeset('Events',@events,'RelTol',1e-6);

% call ode45 with event finding on and a parameter omega passed in
omega=1;
```

```
[t,u,te,ue,ie]=ode45(@eventrhs,[0,20],u0,options,omega);

%*********************************************************
% Here's what the output from the ode solver means:
% t: array of solution times
% u: solution vector, u(:,1) is x(t), y(:,2) is vx(t)
% te: array of event times
% ue: solution vector at the event times in te
% ie: index for the event which occurred, useful when you
%     have an array of events you are watching instead of
%     just a single type of event.  In this example ie=1
%     for the x=0 crossings, with x increasing, and ie=2
%     for the vx=0 crossings, with vx decreasing.

% separate the x=0 events from the vx=0 events
% by loading x1 and v1 with the x-positions and
% v-velocities when x=0 and by loading x2 and v2
% with the positions and velocities when v=0
%*********************************************************

n1=0;n2=0;
for m=1:length(ie)

   if ie(m)==1
      n1=n1+1;
%  load event 1: x,v,t
      x1(n1)=ue(m,1);v1(n1)=ue(m,2);t1(n1)=te(m);
   end

% load event 2: x,v,t
   if ie(m)==2
      n2=n2+1;
      x2(n2)=ue(m,1);v2(n2)=ue(m,2);t2(n2)=te(m);
   end
end

% plot the harmonic oscillator position vs time
plot(t,u(:,1),'g-')
hold on

% plot the x=0 crossings with red asterisks and the v=0
% crossings with blue asterisks
plot(t1,x1,'r*')
plot(t2,x2,'b*')

hold off
```

Example 16.6b (eventrhs.m)

```
% Example 16.6c (Physics 330)

%  eventrhs.m, Matlab function to compute [du(1)/dt;du(2)/dt]

function rhs=eventrhs(t,u,omega)

% right-hand side for the simple harmonic oscillator
% make sure rhs is a column vector

rhs(1,1)=u(2);
rhs(2,1)=-omega^2*u(1);
```

Example 16.6c (events.m)

```
% Example 16.6c (Physics 330)

%   events.m, Matlab function to control event finding by
%             Matlab's ode solvers)

function [value,isterminal,direction] = events(t,u,omega)

%*********************************************************
% Locate the time and velocity when x=0 and x is increasing

% value array: same dimension as the solution u.  An event is defined
%              by having some combination of the variables be zero.
%              Since value has the same size as u (2 in this case) we
%              can event find on two conditions.  Should be a column vector
%*********************************************************

value(1,1) = u(1);  % load value(1) with the expression which,
                    % when it is zero, defines the event, u(1)=0 in this case.

value(2,1)=u(2);    % load value(2) with a second event condition, vx=0
                    % (u(2)=0) in this case.  If you don't want a second
                    % event just set value(2)=1 so it is never 0.

isterminal = [0 ; 0];  % this vector tells the integrator whether
                       % to stop or not when the event occurs.
                       % 1 means stop, 0 means keep going.  isterminal
                       % must have the same length as y (2 in this case).
                       % Should be a column vector

direction = [1 ; -1];  % direction modifier on the event:
                      % 1 means value=0 and is increasing;
                      % -1 means value=0 and is decreasing;
                      % 0 means value is zero and you don't care
                      % whether it is increasing or decreasing.
```

```
                    % direction must have the same length as y.
                    % should be a column vector
```

Chapter 17

Publication Quality Plots

The default settings which Matlab uses to plot functions are usually fine for looking at plots on a computer screen, but they are generally pretty bad for generating graphics for a thesis or for articles to be published in a journal. However, with a bit of coaxing Matlab can make plots that will look good in print. Your documents will look much more professional if you will take some time to learn how to produce nice graphics files. This chapter will show you some of the tricks to do this. This material owes a lot to the work of some former students: Tom Jenkins and Nathan Woods.

Before getting started, let's review a little about graphics file formats. *Raster* formats (e.g. jpeg, bmp, png) are stored as a grid of dots (like a digital photograph). Raster graphics are well-suited for on-screen viewing, and they are usually the best choice for graphics intended for a presentation that will be viewed using a computer projector. However, raster graphics are usually not a good choice for figures destined for the printer (especially line plots and diagrams). They usually look blurry and pixelated on paper because of the mismatch between the image resolution and the printer resolution. Although it is possible to just make really high resolution raster graphics for printing, it is usually better to use vector formats for printed line plots and drawings.

Vector formats store pictures as mathematical formulas describing the lines and curves, and let the renderer (e.g. the printer) draw the picture the best it can. The most common format used to store vector graphics in physics is encapsulated postscript (EPS). Although some programs don't display EPS graphics very nicely on screen (Word does a particularly bad job with on-screen EPS), the figures look great in the printed copy or an exported PDF. We will first learn how to make nice EPS graphics files for printing, and then later we will go over some tips for making nice raster graphics for a presentation.

17.1 Creating an EPS File

Matlab can create a vector EPS files for you. To see how this works, let's make a simple plot with some fake data that looks like something you might publish. Run this example and then select "Save as..." in the figure window, and Matlab will let you choose to save your plot in the EPS format. We have included the EPS file generated this way as Fig. 17.1.

Example 17.1a (ch17ex1a.m)

```
%Example 17.1a (Physics 330)
clear;close all;

% Creates some fake data so we have something to plot.
x=0:0.05:2*pi;
f=sin(x);
data = f + rand(1,length(x))-0.5;
err_hi = f + 0.5;
err_low = f - 0.5;

% Plot the data
plot(x,f,'b',x,data,'b.',x,err_hi,'r-.',x,err_low,'g--');
```

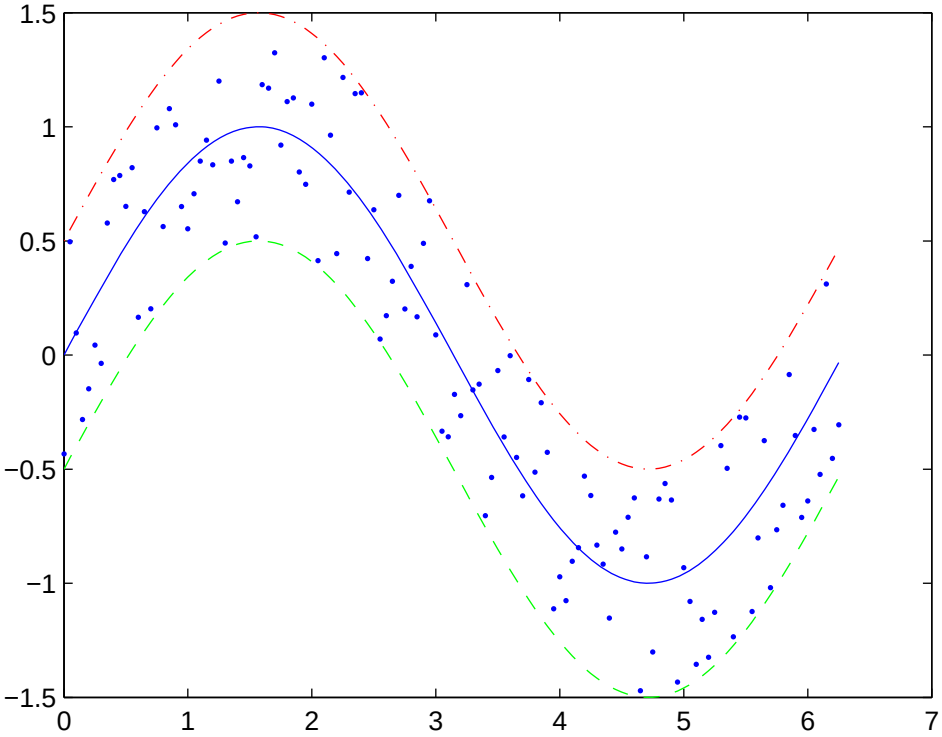


Figure 17.1 Default plot output from Matlab

While this is a pretty simple process, the EPS shown in Fig. 17.1 has a few problems. The axes Matlab chose on which to plot the function aren't necessarily the ones we would pick, but back in section 4.7 we learned how to set the limits of a plot, so that can be addressed (if you don't recall how, go read section 4.7).

A problem that is a little harder to address is that the Matlab default is for figures that are intended to take up a good fraction of the page. We could just scale Fig. 17.1 to to be smaller (and we often see this in theses), but the resulting figure often has unreadably small text and almost invisible lines. This type of figure will not be acceptable for physics journals. These journals have strict requirements on how wide a figure can be in its final form because of the two-column format (8.5 cm width) they often use. But at the same time they require that the lettering be a certain size (for example, American Institute of Physics journals require that in the final reduced size the lettering be at least 2.8 mm high). The journals often require that the figures be submitted in Encapsulated Postscript format (.EPS) that are the right size without any fussing on their part.

While you may not immediately publish in journals, you will almost certainly include plots in your senior thesis. The rules for figures in your thesis aren't as strict as those for journals, but you will probably still want to create something that looks nice when scaled so that the plot doesn't take half a sheet of paper. Fortunately, Matlab allows us to change the visual properties of a plot. Once you have learned the basics, you can use Matlab to make suitable figures for your thesis and journal articles.

17.2 Controlling the Appearance of Figures

You can control the visual properties of a figure from the GUI of the figure window. This capability is great for quick one-time adjustments and to get a feel for what can be done. To get started, click on the "show plot tools" button on the toolbar to display the interface. If you haven't used the GUI formatting tools yet, you should take some time to get familiar with their capabilities. Once you have formatted a figure to your liking, you can save export an EPS for use in your paper. If you want to control the size of your exported plot through the GUI, you will need to use the "Export Setup..." option in the File menu before making the EPS. It is a good idea to save your doctored figure as a .fig file (in addition to EPS file). The .fig file stores all of your adjustments, so you can come back later and modify something and the re-export without having to start from scratch.

As convenient as the GUI interface can be, it has its limitations. Some properties are buried pretty deep in the interface, and it can get tedious to manually format a large number of graphs (and then reformat them all when you decide something needs to change). Fortunately, you can also control the visual appearance of your figures using m-file commands. With the m-file approach, your plot gets the formatting applied each time you run your script. You can also cut and paste commands to format all your figures at once after you've decided on a size and style for your graphics. In the long run, you will save yourself time by learning to control figure properties from the m-file.

To help you with learn the m-file commands, Matlab allows you to export all of the adjustments you make to a figure in the GUI to m-file commands using "Generate m-file..." in the figure's File menu. You can then paste this code into your files to get this figure formatting each time you run the script. However, before you can make the m-file formatting commands work as you expect, you need to take the time to understand a few concepts—just blindly pasting the Matlab-generated code without understanding what it does will usually not get you what you want. The commands have to be put in the right place and refer to the right objects. The rest of this section will teach you the basics of how

this works.

Matlab treats a figure as a collection of visual objects. Each object has a *handle* (a kind of label) to refer to objects in a figure. A handle is simply a number that Matlab has associated with an object. (You can look at the number of a handle, but it won't really tell you anything—it just references a place in the computer's memory associated with the object.) For most objects, you can get the handle when you create them. For instance, the code

```
tt = xlabel('My Label');
```

stores a handle to the x-axis label object in the variable `tt`.

Once you have a handle to an object, you can specify the visual properties using the `set`, using the following syntax (assuming you have already stored the object's handle in `tt`):

```
set(tt,'PropertyName1','PropertyValue1',...)}
```

This command tells Matlab to take the object with handle `tt` and set its PropertyName to PropertyValue. The last comma and dots are not part of the syntax, but indicate that you can set as many property Name-Value pairs as you want in the same `set` command. For instance, to make the x-axis label 20 point Arial font, you would use the command

```
set(tt,'FontSize',20,'FontName','Arial');
```

Take a moment now and modify Example 17.1a to add an x-axis label and change its font size to 8 point.

One of the most frequently referenced objects is the *axes* object. This object includes the box surrounding the plot, and it also includes all the labels and titles as *child objects*. When you set many of the properties of the parent axes object (e.g. the font size), the child objects also inherit this setting. This feature makes the axes object a useful way to set a bunch of things at once. Getting a handle to an axes object is a little different because you don't usually create axes objects manually—Matlab usually does it for you (for instance, when you use the figure command Matlab makes an axes object to put in the figure). To get a handle to an axes object, you use `gca` command (which stands for Get Current Axes). For instance, the command

```
aa = gca;
```

stores the handle for the current axes object in the variable `aa`. You need to use `gca` to store the current handle in a variable *before* you open another set of axes (e.g. by using the `figure` command), otherwise the axes you want to refer to will no longer be the current axes. If, for example, to want to set the font to 12 point Symbol for a figure, you would use

```
set(aa,'FontSize',12,'FontName','Symbol');
```

See "Axes Properties" in the online help for a list of properties you can set for the axes.

Another frequently used object is the *lineseries* object, which refers to the lines or symbols displayed inside an axes object to represent the data. Matlab can have multiple lineseries plotted on the same set of axes, so we need a way to reference an individual

lineseries independent from the axes on which they are displayed. Take a moment to modify the code in Example 17.1a to get handles to the individual lineseries objects using the following syntax:

```
pp = plot(x,f,'b',x,data,'b.',x,err_hi,'r-.',x,err_low,'g--');
```

This syntax stores an array of handles referring to the lineseries objects displayed by the `plot` command in the variable `pp`. The first element, `pp(1)`, refers to the first lineseries (the plot of `f`), the second element, `pp(2)`, refers to the second lineseries (the plot of `data`), and so forth.

The syntax for setting the properties of the lineseries object is essentially the same as the axes, except you use the handle to the lineseries:

```
set(pp(1),'PropertyName1','PropertyValue1',...)}
```

Again, you can set as many lineseries properties as you want in the same `set` command. To get the hang of this, modify Example 17.1a again to change the plot of the `data` variable to red stars rather than blue dots using the following command:

```
set(pp(2),'LineStyle','none','Marker','*','Color',[1 0 0])
```

Note that here we have chosen to set the color with an RGB value rather than a preset color (an RGB value is a matrix of three numbers between 0 and 1 which specify a color).

Because we often need to control the visual styles of the lineseries data, Matlab gives us shortcuts to set many of the visual properties of the plot data right in the plot command. You have have already learned many of these (and in fact we used some in our example). You could have gotten the red star effect simply by changing your plot command to

```
pp = plot(x,f,'b',x,data,'r*',x,err_hi,'r-.',x,err_low,'g--');
```

You can also set properties that apply to every lineseries in the plot by putting name-value pairs at the end of a plot command. For example, change your plot command in Example 17.1a to

```
pp = plot(x,f,'b',x,data,'r*',x,err_hi,'r-.',x,err_low,'g--','LineWidth',2);
```

Note that this changes the line thickness for the plots to a heavy 2 point line (the default width is 0.5 point). However, the stars are also drawn with heavy lines which looks kind of awkward. If you want to control the properties of the lines individually, you have to go back to the longer syntax with handles. For example

```
pp = plot(x,f,'b',x,data,'r*',x,err_hi,'r-.',x,err_low,'g--');
set(pp(1),'LineWidth',2);
```

makes the plot of `f` heavy, but leaves the rest at the default width. See "lineseries properties" in the online help for a list of properties you can set for a lineseries.

17.3 Controlling the Size of Exported Graphics

Controlling the size of the exported figure is tricky. The basic idea in controlling size is that you have the OuterPosition property which specifies the extent of the entire figure, the Position property which specifies the position of the axes box within the figure, and the TightInset property that describes the size of the labels around the axes box. If you want to learn about these properties, see "axes properties" in the Matlab help. Probably the best way to learn how to do this is to study an example. Execute Example 17.2a and see what the plot looks like. The EPS produced using "Save As" is included as Fig. 17.2 in this document so you can see what was affected by these commands (compare with Fig. 17.1 which shows the output without the sizing commands).

Example 17.3a (ch17ex3a.m)

```
%Example 17.3a (Physics 330)
clear;close all;

x=0:0.05:2*pi;
f=sin(x);
data = f + rand(1,length(x))-0.5;
err_hi = f + 0.5;
err_low = f - 0.5;

% Store our target size in variables. Using these variables
% whenever you reference size will help keep things cleaner.
Units = 'Centimeters';
figWidth = 8.5;
figHeight = 7;

% Create a figure window.
% You can specify general properties right in the figure command.
% Later we'll see how to modify things with a handle to the entire figure.
figure('Units',Units,'Position',[10 10 figWidth figHeight])

% Plot the data
plot(x,f,'b',x,data,'b.',x,err_hi,'r-.',x,err_low,'g--');

% Get a handle to the newly created axes
aa = gca;

% First set the outer dimensions of the axes the same as the figure.
% The 'OuterPosition' property describes the boundary of the whole figure.
set(aa,'Units',Units,'OuterPosition',[0 0 figWidth figHeight])
% Then calculate where the axes box should be placed inside the overall
% figure (using information from 'TightInset').
newPos = get(aa, 'OuterPosition') - ...
   get(aa, 'TightInset') * [-1 0 1 0; 0 -1 0 1; 0 0 1 0; 0 0 0 1];
% The 'Position' property describes the the rectangle around the plotted data
set(aa, 'Position', newPos);
```

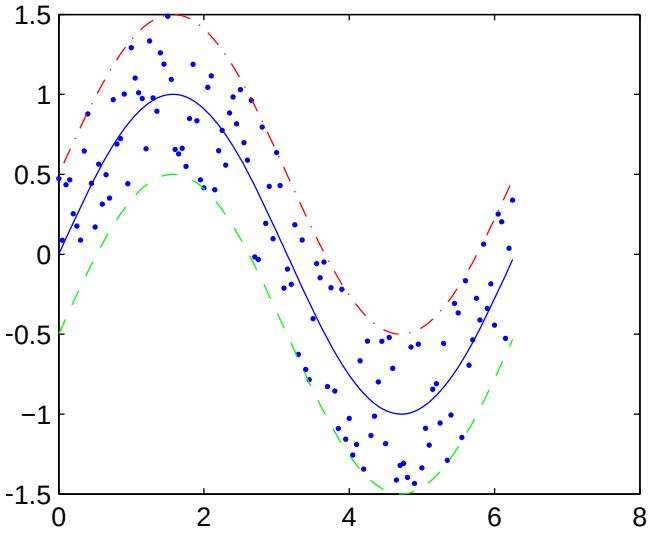


Figure 17.2 Plot made in Example 17.3a (no scaling).

17.4 Making an EPS Suitable for Publication

The sizing commands in Example 17.3a fixed our scaling problem, but the figure still needs a lot of improvement before it would be suitable for a thesis or journal. For instance, we still need to fix the axes limits and put on labels. The lines are still sort of "spidery," and the x-axis is labeled with integers rather than fractions of π. We also need to provide a legend that tells what the lines and dots on this plot mean. In Example 17.4a we show how to address all of these issues by setting the visual properties of the objects on the figure. Run this example and then study the comments in it. The EPS output (made using "Save as") produced by this example is included as Fig. 17.3.

Example 17.4a (ch17ex4a.m)

```
%Example 17.4a (Physics 330)
clear;close all;

x=0:0.05:2*pi;
f=sin(x);
data = f + rand(1,length(x))-0.5;
err_hi = f + 0.5;
err_low = f - 0.5;

% Choose what size the final figure should be
Units = 'Centimeters';
figWidth = 8.5;
figHeight = 7;

% Create a figure window of a specific size.
% Note that we also get a handle to the entire figure (ff) for later use
ff = figure('Units',Units,'Position',[10 10 figWidth figHeight])

% Plot the data and get handles to the lineseries objects.
```

```
pp=plot(x,f,'b',x,data,'b.',x,err_hi,'r-.',x,err_low,'g--');

% Set the lineseries visual properties.
set(pp(1),'LineWidth',2);   % Make the main sine a heavy line
set(pp(2),'MarkerSize',8);  % Make the dots a bit bigger than default
set(pp(3),'LineWidth',1);   % Make the error bound lines slightly heavier
set(pp(4),'LineWidth',1);   % Make the error bound lines slightly heavier

% Set the plot limits and put on labels
axis([0 2*pi -1.6 1.6])
xlabel('\theta')
ylabel('sin(\theta)')
title('Fake measurement of Sine function')

% Get a handle to the axes and set the axes visual properties
aa=gca;
set(aa,'FontSize',8,...   % Set the font for the axes
       'FontName','Symbol',... % to get pi in labels (p is pi in symbol font)
       'LineWidth',1,...  % Make the border around the figure heavier
       'TickLength',get(aa,'TickLength')*2,...
       'XTick',[0  pi/2 pi 3*pi/2 2*pi],... % Specify where the ticks go
       'XTickLabel',{'0';'p/2';'p';'3p/2';'2p'},... % custom tick labels
       'XMinorTick','On',...
       'YTick',[-1 -.5 0 .5 1],...
       'YMinorTick','On')

% Put in a legend.  We have to specify the font back to Helvetica (default)
% because we changed to the symbol font above for the pi tick labels.
ll=legend('Sine','Fake Data','Upper Limit','Lower Limit');
set(ll,'FontName','Helvetica')

% Set the output size for the figure.
% DO THIS LAST because the margins will depend on font size, etc.

% Set the outside dimensions of the figure.
set(aa,'Units',Units,'OuterPosition',[0 0 figWidth figHeight])
% Calculate where the axes box should be placed inside the overall figure.
newPos = get(aa, 'OuterPosition') - ...
   get(aa, 'TightInset') * [-1 0 1 0; 0 -1 0 1; 0 0 1 0; 0 0 0 1];
% Now set the position of the axes box within the figure
set(aa, 'Position', newPos);

% As a finishing touch, draw a proper rectangle around the plot area
% (The box matlab draws has lines that don't join in sharp corners...)
Bound = annotation(ff,'Rectangle',[0 0 1 1]);
set(Bound,'Units',Units,...
          'Position',get(aa,'Position'),...
          'LineWidth',1);
```

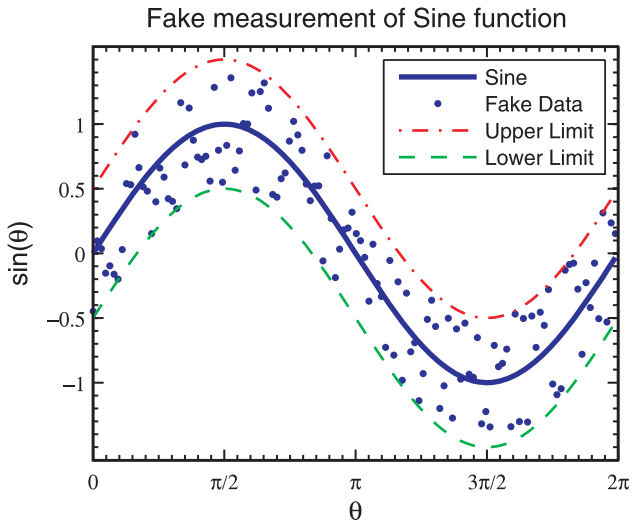


Figure 17.3 Plot made in Example 17.2b (no scaling).

Although the code is (of course) more complicated, it does make a graph that's suitable for publication. The FontName business can be removed if you are not trying to get symbols as tick labels (unfortunately you can't use Matlab's TEX capabilities for tick labels). You may have also noticed that the example used the `get` command, which allows you to read the current value of a property from one of the objects that you are controlling.

17.5 Subplots

In technical writing, it is often desirable to put multiple plots in the same figure. The command to produce plots like this is `subplot`, and the syntax is:

```
subplot(rows,columns,plot number)
```

This command splits a single figure window into an array of subwindows, the array having `rows` rows and `columns` columns. The last argument tells Matlab which one of the windows you are drawing in, numbered from plot number 1 in the upper left corner to plot number `rows*columns` in the lower right corner, just as printed English is read. See online help for more information on subplots.

You have probably already used `subplot`, but there are a few tricks to controlling the size and appearance of the exported figure for publication. Here is an example of how to produce a two-axis plot, formatted to fit in a single column of a journal. Notice that in such a figure, there are multiple sets of axes, so it is important to be clear which set you are setting properties for. Figure 17.4 shows the plot produced by this script.

Example 17.5a (ch17ex5a.m)

```
% Example 17.5a (Physics 330)
clear;close all;

% Make up some data to plot
```

```
x=0:.01:100;
f1=cos(x);
f2=exp(-x/20);

% Choose what size the entire final figure should be
Units = 'Centimeters';
figWidth = 8.5;
figHeight = 10;

% Create a figure window of a specific size.
ff = figure('Units',Units,'Position',[10 10 figWidth figHeight])

% Make the top frame: 2 rows, 1 column, 1st axes
subplot(2,1,1)

% Make the plot--in this case, we'll just set the lineseries
% properties right in the plot command.
plot(x,f1,'r-',x,f2,'b--','LineWidth',1.5)

% set the plot limits
axis([0 100 -1.1 1.1])

% Make the labels.
xlabel('x')
ylabel('f_1(x), f_2(x)')
title('Multiplication of Functions')

% Get a handle to the top axes and set the properties of this set of axes
aa = gca;
set(aa,'FontSize',10,...
        'LineWidth',0.75,...
        'XTick',[0 20 40 60 80 100],...
        'YTick',[-1 -.5 0 .5 1])

% Set this axis to take up the top half of the figure
set(aa,'Units',Units,'OuterPosition',[0 figHeight/2 figWidth figHeight/2])

% Now adjust the axes box position to make it fit tightly
newPos = get(aa, 'OuterPosition') - ...
   get(aa, 'TightInset') * [-1 0 1 0; 0 -1 0 1; 0 0 1 0; 0 0 0 1];
set(aa, 'Position', newPos);

% As a finishing touch, draw a proper rectangle around the plot area
Bound = annotation(ff,'Rectangle',[0 0 1 1]);
set(Bound,'Units',Units,...
          'Position',get(aa,'Position'),...
          'LineWidth',1);


% Create the second set of axes in this figure: 2 rows, 1 column, 2nd axes
subplot(2,1,2)
```

```
% Make the second plot
plot(x,f1.*f2,'b-','LineWidth',1.5)

% Set labels for second axes
xlabel('x')
ylabel('f_1(x)* f_2(x)')

% Set limits
axis([0 100 -1.1 1.1]);

% Get a handle for the second axes.  Note that we are overwriting
% the handle for the first axes, but we're done modifying them, so it's ok
aa=gca;
% Set properties for the second set of axes
set(aa,'FontSize',10,...
       'LineWidth',0.75,...
       'XTick',[0 20 40 60 80 100],...
       'YTick',[-1 -.5 0 .5 1])

% Set this axis to take up the bottom half of the figure
set(aa,'Units',Units,'OuterPosition',[0 0 figWidth figHeight/2])

% Now adjust the axes box position to make it fit tightly
newPos = get(aa, 'OuterPosition') - ...
   get(aa, 'TightInset') * [-1 0 1 0; 0 -1 0 1; 0 0 1 0; 0 0 0 1];
set(aa, 'Position', newPos);

% As a finishing touch, draw a proper rectangle around the plot area
Bound = annotation(ff,'Rectangle',[0 0 1 1]);
set(Bound,'Units',Units,...
          'Position',get(aa,'Position'),...
          'LineWidth',1);
```

17.6 Making Raster Versions of Figures

While EPS figures are great for printing, the predominant method for presenting information in a talk is with a computer projector, usually with something like PowerPoint. Unfortunately, PowerPoint does a lousy job of rendering EPS files, so you may prefer to make a raster version of your figure to use in a presentation. In principle, you can just do this by changing output resolution in the "Export Setup" dialog and then choosing a raster format in the "Save as..." dialog. However, this can sometimes give mixed results.

We have found better results by exporting via the Matlab `print` command. (See Matlab help for details on `print`.) To use this method, make sure to get a handle to the figure window when it is created using the

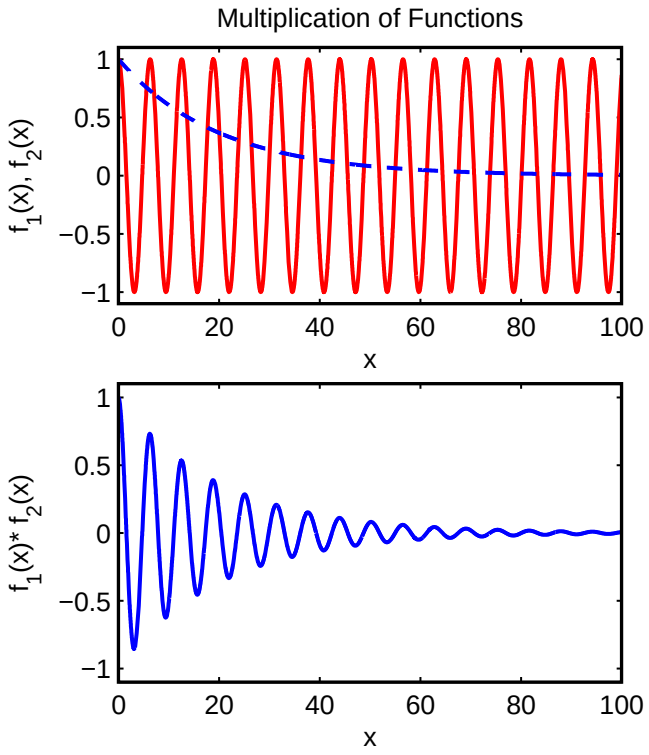


Figure 17.4 An example of a set of plots produced using `subplot`.

```
ff=figure
```

syntax. Then control the size and appearance as we discussed above for making EPS figures. Then once your figure looks right, you can use the following code:

```
set(ff,'PaperUnits',Units,...
       'PaperSize',[figWidth figHeight],...
       'PaperPosition',[0 0 figWidth figHeight]);
print -djpeg -r600 'Test.jpg'
```

to make a jpeg image with good resolution (600 dpi). This code assumes you put the size in the variables `Units`, `figWidth`, and `figHeight` as before. The raster images that Matlab produces sometimes get rendering oddities in them, and they don't do anti-aliasing to smooth the lines. This can sometimes be helped by increasing resolution or changing what rendering method Matlab uses (see the Renderer property in "Figure Properties" in Matlab help).[1]

Another situation where raster graphics may be called for is for 3-D surface plots with lighting, etc. These are hard to render in vector graphics formats, so even in when destined

[1]You can often get better and more reliable results in making raster figures for presentations by creating the EPS figure and then converting the EPS file directly using a good raster imaging program. However, this requires a good raster imaging program which we don't have available in the department labs. The Matlab renderer usually makes figures that work just fine, however.

for printing you may be better off making a raster figure file. Just control the resolution as shown in the example code above to make sure your printed versions look OK. (We don't recommend getting into the habit of doing this for print figures. Matlab's vector rendering engines usually do a better job than its raster rendering engines.)

Index