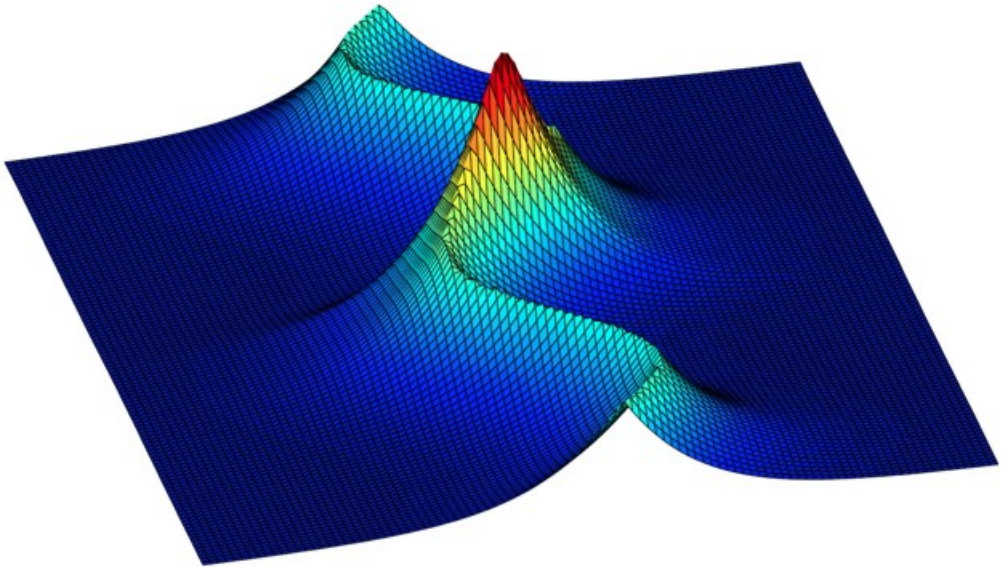
COMPUTATIONAL METHODS FOR SCIENTISTS

INTRODUCTION TO MATLAB



Ross L. Spencer and Michael J. Ware

Department of Physics and Astronomy
Brigham Young University

COMPUTATIONAL METHODS FOR SCIENTISTS

INTRODUCTION TO MATLAB

Ross L. Spencer and Michael J. Ware

Department of Physics and Astronomy
Brigham Young University

Last Revised: September 14, 2022

Preface

This is a tutorial to help you get started in Matlab. Examples of Matlab code are in monospaced font `like this`. You will learn best if you type and execute in Matlab all of the examples as you go rather than trying to read the text like a novel. In the beginning, you can just type the sample code at the ≫ command line prompt. Later, you'll write program files. Longer sections of code are set off and named. This code can be found as files on the Physics 330 web page at physics.byu.edu/courses/computational.

This booklet can also be used as a reference manual because it is short and has lots of examples. It is almost true that the basics of Matlab are in chapters 1-7 while the applications are in chapters 11–13. Please tell us about mistakes and make suggestions to improve the text (ware@byu.edu).

Contents

Chapter 1

Running Matlab

1.1 Getting Started

Start Matlab and locate the command window. As you read this book, type the commands that appear on their own line in `this font` at the ≫ prompt in this window, and then hit Enter to see how they work. You can get help at any time by pressing F1 or clicking the question mark at the top of the Matlab window.

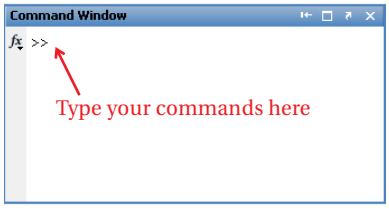


Figure 1.1 The command window allows you to directly issue commands to Matlab.

1.2 It's a Calculator

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

```
1+2
5/6
cos(pi)
```

Note that Matlab's standard trig functions are permanently set to radians mode, but it also provides degree versions of the trig functions:

```
sind(90)
```

The `ans` command returns the last result calculated.

```
2*2
ans+1
```

To enter numbers in scientific notation, like 1.23×10^{15}, use this syntax

```
1.23e15
```

Finally, note that the up-arrow key ↑ will display previous commands. And when you back up to a previous command, you can hit Enter and it will execute again. Or you can edit it and then execute the modified command. Do this now to re-execute and edit some of the commands you have already typed.

1.3 Variables

While Matlab has other types of variables, you will mostly just use two types: the matrix and the string. Variables are not declared before they are used, but are defined on the fly. The assignment command is the equal sign. For instance,

```
a=20
```

creates the variable a and assigns the value 20 to it, and then the command

```
a=a+1
```

adds one to the stored value. Note that *variable names in Matlab are case sensitive,* so watch your capitalization. If you want to see the value of a variable, just type its name like this

```
a
```

☞ An assign statement in programming is not a mathematical equation. It makes perfect sense to write a=a+1 as assign statement: it tells the computer to get the value of a and add 1 to it and then store the evaluated quantity back in the variable a. However, the mathematical equation $a = a + 1$ is clearly false.

String Variables

String variables contain a sequence of characters, like this

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

Some Matlab commands require options to be passed to them using strings. Make sure you enclose them in single quotes, as shown above.

Numerical Accuracy

All numbers in Matlab are stored as double-precision values which have 15 digits of accuracy. When you display numbers to the screen, like this

```
x=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 all the digits type

```
format long e
x
```

The e stands for exponential notation. The four most useful formats to set are

```
format short   % the default format
format long
format long e
format short e
```

Matlab knows the number π.

```
pi
```

Try displaying π under the control of a couple of the formats above.

⚠ Matlab will let you set the variable pi to anything you want, like this, pi=2; but please don't.

Clearing the Workspace

Matlab keeps all the variables you define in memory until you tell it to erase them. In Matlab lingo, the place where variables are stored is the *workspace*. To erase a single variable from the workspace, type `clear` and then the variable name. For instance, if you foolishly set the variable `pi` to something other than π, like this

```
pi=2
```

Then you would use the command

```
clear pi
```

to erase your user-defined value and restore the default value. To erase all the variables in the workspace, simply type `clear` without anything after it

```
clear
```

1.4 Matrix Variables

The "Mat" in Matlab stands for matrix, and it treats all numeric variables as matrices. For instance, Matlab thinks the number 2 is a 1×1 matrix:

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

You can enter a 1×4 row matrix using commas and braces

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

or a 4×1 column matrix with semicolons and braces

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

or a 3×3 matrix with columns separated by commas and rows separated by semicolons And the matrix

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

When you want to access the values of individual matrix elements, use the syntax `A(row,column)`. For example, to get the element of `A` in the 3rd row, 5th column 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:

```
b(end)
A(3,end)
```

The Colon (:) Command

⚠ Pay attention to this section. Understanding the colon command clearly will make your life much easier.

You can build large, evenly spaced arrays using the colon (:) command. To build an array x of values starting at $x = 0$, ending at $x = 10$, and having a step size of $dx = 0.5$ type this:

```
x=0:.5:10
```

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.

Selecting Rows and Columns

Sometimes you will want to select part of a matrix and load it into another array. This is also done with Matlab's all-purpose colon (:) command. To select just part of row or column, use commands like this:

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

The variable c now contains the first two elements of the second row of A. To load a column vector b with all of the entries of the third column of the matrix A, you can use the syntax

```
b=A(1:end,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. Since this type of operation is so common in Matlab, there is a shorthand for selecting all of the entries in a given dimension: just leave off the numbers around the colon, like this

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

And to load a row vector c with the contents of the second row of the matrix A use

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

Matlab treats strings as one-dimensional arrays of characters, so you to slice up string variables using this method

```
s='This is a string'
s(1:7)
```

1.5 Calculating

Matlab was built to crunch numbers, and it usually handles them much faster than symbolic programs like Maple or Mathematica. Here are the basics.

☞ 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."

Add and Subtract

Matlab knows how to add and subtract 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
```

Multiplication and Division

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:

```
.*
```

⚠ This "dot" operator stuff is important. Be patient if it is a little confusing now. When we start plotting and doing real calculations this will all become clear.

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.

Arithmetic with Array Elements

If you want to do some arithmetic with specific values stored individual elements of your arrays, you can just access them by index. 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 matrices in Matlab's mind), so Matlab will just treat this like the normal division of two numbers. We don't have to use the `./` command, although it wouldn't hurt if we did.

Sum the Elements

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)
```

If you want to add up all of the elements in the array, just nest the sum command like this

```
sum(sum(A))
```

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 by using `i` as a variable name, like this:

⚠ Don't use `i` as a variable name.

```
i=3
```

Don't do this, or you will no longer have access to imaginary numbers. 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

```
a=1+2i
b=2-3i

a+b
a-b
a*b
a/b
```

If you are in a discipline where j is used for the imaginary number, Matlab can be your friend too. The variables `i` and `j` have the same meaning in Matlab, and everything we say about `i` works the same with `j`. 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)
```

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, hyperbolic, and Bessel functions.

cos(x)	sin(x)	tan(x)	acos(x)	asin(x)	atan(x)
atan2(y,x)	cosd(x)	sind(x)	tand(x)	acosd(x)	asind(x)
atand	atan2d	exp(x)	log(x)	log10(x)	log2(x)
sqrt(x)	cosh(x)	sinh(x)	tanh(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)	erf(x)	erfc(x)	erfcx(x)	erfinv(x)
gamma(x)	expint(x)	legendre(n,x)	factorial(x)		

Table 1.1 A sampling of the mathematical functions available in Matlab.

1.6 Matlab Functions

Matlab knows all of the standard functions found on scientific calculators and even many of the special functions like Bessel functions. Table 1.1 shows a bunch of them. Note that the natural log function $\ln x$ is the Matlab function `log(x)`. To get the base-10 log, use `log10`.

All of the functions in Table 1.1 work like the "dot" operators discussed in the previous section (e.g. `.*` and `./`). This means, for example, that it makes sense to take the sine of an array: the answer is just an array of sine values. For example, type

```
sin([pi/4,pi/2,pi])
```

and see what you get. Likewise, you could use the degree versions of the trigonometry functions, like this:

```
cosd([0,45,90])
```

Housekeeping Functions

Matlab also has a bunch of other functions that don't really do math but are useful in programming. Two of the more useful housekeeping commands are `max` and `min`, which return the maximum and minimum values of an array. For example, create a couple of arrays like this

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

and then you can find the max and min like this

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

And with a slight change of syntax `max` and `min` will also return the indices in the array at which the maximum and minimum occur, like this

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

Look at the values of `imin` and `imax` and discuss them with your lab partner until you understand what they mean.

There are many more housekeeping functions. We've listed several of them in Table 1.2. Take a minute to familiarize yourself with the functions in the table; many of them will come in handy. Try

```
floor([1.5,2.7,-1.5])
```

to convince yourself that these functions operate on matrices and not just on single numbers.

`clc`	clears the command window; useful for beautifying printed output
`clear`	clears all assigned variables
`close all`	closes all figure windows;
`close 3`	Close figure 3
`length(a)`	the number of elements in a vector
`size(c)`	the dimensions of a matrix
`ceil(x)`	the nearest integer greater than x
`fix(x)`	the nearest integer to x looking toward zero
`floor(x)`	the nearest integer less than x
`round(x)`	the nearest integer to x
`sign(x)`	the sign of x and returns 0 if $x = 0$

Table 1.2 A sampling of "housekeeping" functions

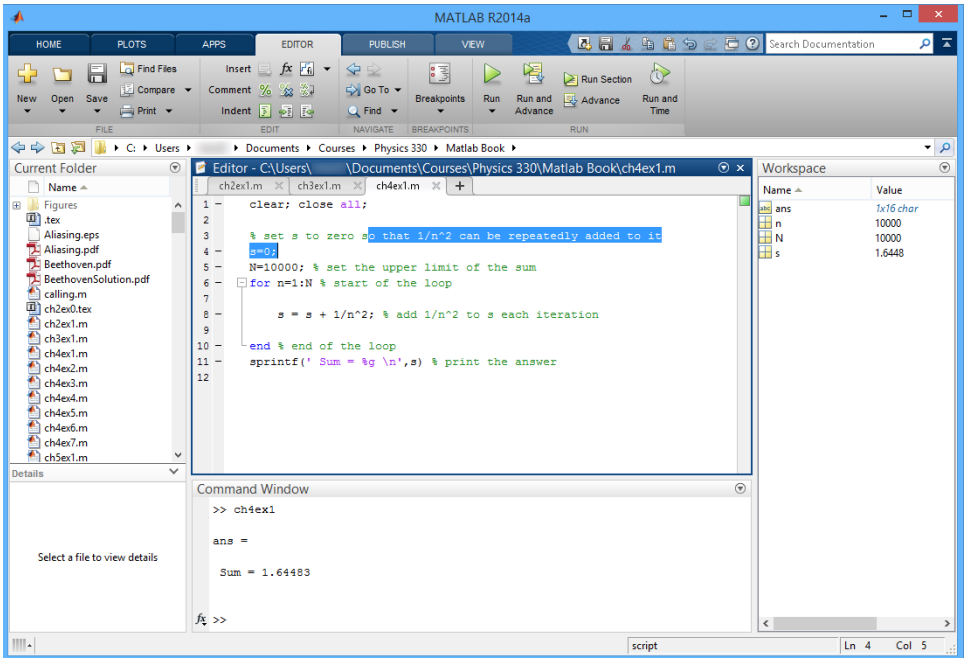
Chapter 2

Scripts

Typing in commands in the command window is just the tip of the iceberg of what Matlab can do for you. Most of the work you will do in Matlab will be stored in files called *scripts*, or m-files, containing sequences of Matlab commands to be executed over and over again. Let's walk through the process of writing a script.

2.1 The Matlab Desktop

First, let's take a tour of the Matlab desktop. The default layout of the desktop usually looks something like this:



The different panes provide the following functionality.

- The Command window at the bottom allows you issue commands directly to Matlab.
- The Editor window in the middle is where you write Matlab scripts. This is where we'll spend most of our time when using Matlab. If you've never opened a script file, this window may be missing, but it will appear when you create or edit a script.

- The Workspace window at the right displays all of the variables that Matlab currently has in memory. Double clicking a variable launches the Array Editor, which allows you to view and modify the values in your variables.
- The Current Folder window on the left displays the files in your current directory. Double click on script files (*.m) to open them in the editor, or use the navigation bar up above to navigate to a different folder.

You can rearrange the Matlab desktop using drag, drop, and docking, but we recommend that you wait until you've used it for a while before you start dragging things around.

2.2 Script Files

Let's start the scripting process. First, make a new script file in by clicking on the "New" command on the tool bar. Save this file as "test.m" (Matlab will automatically add the .m extension) in the directory where you want to store your Matlab scripts. After you've created the script file, enter the sample command `x=sin(2)` in the editor window, and save it.

☞ Make a folder where you can store all of the scripts for this class. Work out an arrangement to share these files with your lab partner.

Running a Script

When you run a script, the commands in the file are executed from top to bottom just as if you had typed them on the command screen. Before you can execute a script, you need to point Matlab's current folder to the place where your script is saved. Take a moment now to change the current folder (shown in the toolbar) to the directory where you saved the script file (use the buttons in the toolbar). Once your directory is set to the right place, execute your script by typing the name of your file without the `.m` extension in the command window, like this:

```
test
```

Always save changes to your script before executing it in the command window. Matlab loads scripts from the disk, not the editor window.

A convenient shortcut for running a script is to use the "Run" button on the toolbar (the green "play" icon), or simply pressing the 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 saved (if it isn't already pointed there), and then run the script for you. Use this method to run your `test.m` script again.

Script File Names

Before we continue, please note that *script file names cannot start with numbers*, like `330lab1a.m`. You execute scripts by typing their name, and when Matlab receives the start of a number in a command it thinks a calculation is coming.

Since something like `330lab1a` is not a valid calculation, Matlab will give you an error if you try and name your scripts like this. Also, *do not use a space or a period in the file name*. If you must separate words in your file name, you can use the underscore character (e.g. my_script.m), but it is easier just to use names without separators if you can.

⚠ Script names cannot begin with numbers or contain spaces or periods

Clear and Close All

You should nearly always begin your scripts with the following two commands:

```
clear;
close all;
```

The `clear` command clears all variables from Matlab's memory and makes sure that you don't have leftover junk active in Matlab that will interfere with your code. The `close all` command closes any figure windows that are open. The obvious exception to the rule of beginning your scripts with these commands is if you need to keep some data in memory for your script to run, or if you need to leave a plot window open. Add these two lines at the beginning of your `test.m` script, and run it again.

Making Matlab Be Quiet

Any line in a script that ends with a semicolon will execute without printing to the screen. For example, add these two lines of code to your test script

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

and then execute it. Look at the output in the command window. Even though the variable a didn't print, it is loaded with $\sin(5)$, as you can see by typing

```
a
```

in the command window, or looking in the workspace window. Printing output to the command window takes a lot of time, especially when dealing with big matrices, so your scripts will run much faster if you end your lines with semicolons.

⚠ Neglecting to end lines with semicolons can lead to slow execution

2.3 Input and Output

For many scripts, it is sufficient to type the input in the script file before you run it and view the output in the command window by selectively leaving off semicolons in your code. But sometimes you will need to have the program get input from the user and present formatted output on the screen.

2.4 Input

To have a script request and assign a value to the variable N from the keyboard, put the command

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

in your script and run it again. Note that Matlab is asking for input in the Command Window. 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 3×3 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.

2.5 Output

To display formatted results in the Command Window, you can use the fprintf command. Type the following examples in the command window to see what each one produces. (Hint: Use the up-arrow key between entering each command to save yourself a bunch of typing)

```
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 that 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 immediately after % is a format code. A g means use the "best" 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.) Note: \n is the command for a new line. If you want all the details on this stuff, see the Matlab help entry for fprintf.

Chapter 3

Debugging Code

Your programs will usually fail the first time you run them. This is normal. In this chapter we give you some guidance to help you quickly correct the problems and get your code to execute.

3.1 Make Your Code Readable

Begin by make your code readable by humans, most importantly by you. It takes a little more time up front, but pays big dividends in the long run. Here are a few tips to help you get started on the right path.

Add Comments

Document your scripts by including comment lines that begin with %, like this:

```
% 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
```

Get in the habit of documenting your code as you write it. If you don't do it then, you never will. Add a comment line to your test.m script now to start a good habit.

Wrap Long Lines

Make your code more readable by continuing long program lines onto successive lines by using the ... syntax, like this

```
a=sin(x)*exp(-y) + sqrt(b^2-4*a*c)/2/a + c1*d3 +...
  log(z) + sqrt(b);
```

Wrap any lines that are too long to read without horizontal scrolling. It is hard to find bugs you can't easily see.

Enter Matrix Constants Cleanly

When matrices become large the comma and semicolon 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 whenever possible.

Use Short Variable Names

Finally, when coding physics, don't follow the computer science convention of making your code more "readable" by using long variable names like this:

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

If you program like this, your code won't match the math, you'll drive yourself crazy with typos, and you'll end your programming career prematurely due to repetitive-stress injury. Do it this way

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

and then add comments when needed to clarify the meaning of your variables, like this

```
% r12 is the distance between the earth and the satellite
% m1 is the mass of the earth
```

3.2 Debugging

When your script fails you will need to examine the values stored in your variables 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 at the end of your script, just type

```
a
```

and its value will appear on the screen. You can also explore the values stored in memory using the Workspace window (the right pane in the desktop).

Breakpoints and Stepping

It is very helpful in this debugging process to watch what the script does as it runs. To help you do this Matlab comes with two important features: breakpoints and stepping. To help you see how these features work, make a script file with the code from Listing 3.1 in it, and open it in the editor window.

This script uses a loop and a logical test to step the variable n from 1 to 12 and at each value check if n evenly divisible by 3. We haven't introduced these loop

☞ The numbered code listings are available as individual files on the Physics 330 course website. You can download them to your script directory to save a bunch of cut and paste from this document.

Listing 3.1 (ch3ex1.m)

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

% N is the largest number to test
N=12;

% steps the value of n from 1 to N
for n=1:N
   % calculate the integer remainder of this value of n divided by 3
   r = mod(n,3);

   if (r==0)
       % indicate that 3 is a factor of this n
       fprintf(' 3 is a factor of %g \n', n);
   else
       % indicate that if 3 is not a factor of this n
       fprintf(' 3 is not a factor of %g \n', n);
   end
end
```

and logic commands yet, but they will be important, so let's use the debugging commands to explore how loops work.

To see what a breakpoint does, put the cursor on the line "`for n=1:N`" in the editor window and either click on Breakpoints on the tool bar and select Set/Clear, or press F12, or click on the small dash to the right of the line of code. 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 for setting 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 pause.

Set the breakpoint to "on", and execute the script by pressing F5. Notice that a green arrow appear on the line with the breakpoint. Look at the workspace window and note that `N` has been given a value, but that `n` has not. This is because the breakpoint stops execution just before the line on which it is set. You can also see the value of a variable by moving the cursor over the variable in the editor window and waiting for a tip window to pop up. Do this now for `N`.

The most common things to do next step through the code executing each line in turn using the Step button on the toolbar (or more commonly just F10) while watching what happens to your variables in the workspace window. Take a minute now and use F10 to step through the script while watching what happens to the variables in the workspace window. When you are done stepping, you can let the script continue to the end by clicking "Continue" on the toolbar (or more commonly, just use the shortcut F5). Use breakpoints and stepping to explore the this script until you can explain to your lab partner how the `for` loop works and

what how the `if` logic command works.

When you write a new script, you should almost 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.

Stopping Runaway Code

Sometimes you will accidentally write code that takes forever to run. Some common cases are defining a huge array and forgetting to put a semicolon after it to suppress the screen print, or writing an infinite loop. If you've done something like this, Ctrl-C will abort whatever Matlab is doing and return control to you.

3.3 Pause command

A `pause` command in a script causes execution to stop temporarily. To continue just hit Enter. This can be a useful way to view information at a certain point in execution before proceeding. Usually its good practice to give some indication in the command window that you are paused. Beginning students have been known to waste time searching for a "bug" that is causing their code to hang, only to find that they have a forgotten `pause` command in their code.

⚠ Make sure to give some indication to the user that the script is paused and waiting. Otherwise the user just waits for the program to do something.

You can also give `pause` a time argument like this

```
pause(.2)
```

which will cause the script to pause for 0.2 seconds, then continue. You can ask for a pause of any number or fractions of seconds, but 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.

Chapter 4

Loops and Logic

4.1 `for` Loops

A loop is a way of repeatedly executing a section of code. A `for` loop looks like this:

```
for n=1:N
   % Put code here
end
```

which tells Matlab to start `n` with a value of 1, then increment the value 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

$$\sum_{n=1}^{N} \frac{1}{n^2} \tag{4.1}$$

with N chosen to be a large number.

Listing 4.1 (ch4ex1.m)

```
clear; close all;

% set s to zero so that 1/n^2 can be repeatedly added to it
s=0;
N=10000; % set the upper limit of the sum
for n=1:N % start of the loop
    s = s + 1/n^2; % add 1/n^2 to s each iteration
end % end of the loop
fprintf(' Sum = %g \n',s) % print the answer
```

Create a breakpoint at the `s=0` line, run the code, and then step through the first several iterations of the `for` loop using F10. Look at the values of `n` and `s` in the Workspace window and see how they change as the loop iterates. Once you are confident you know how the loop works, press F5 to let the script finish executing.

You can also calculate the same sum using matrix operators like this

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

If your code will work equally well with either a loop or a matrix operator (like `sum`) with the colon command, use the colon command whenever possible. The matrix operators are pre-compiled and usually much faster. In modern versions, Matlab will check to see if it can automatically replace loops with precompiled matrix commands like `sum`, but it can't always tell if the replacement is proper.

Products with a `for` loop

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

Listing 4.2 (ch4ex2.m)

```
clear; close all;

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, 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)
```

⚠ Matlab's `factorial` command is a limited in that it won't act on arrays of numbers 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 using the command `gamma(x)`. you could also check the answer to your factorial loop this way: `gamma(21)`

Recursion relations with `for` loops

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

$$f(x) = \sum_{n=1}^{\infty} a_n x^n \tag{4.2}$$

and that we discovered that the coefficients a_n satisfy the recursion relation

$$a_1 = 1 \quad ; \quad a_{n+1} = \frac{2n-1}{2n+1} a_n. \tag{4.3}$$

To use these coefficients we need to load them into an array a so that $a(1) = a_1$, $a(2) = a_2$, etc. Let's load the array using a `for` loop:

Listing 4.3 (ch4ex3.m)

```
clear; close all;

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 we translated the recursion relation into Matlab code just as it appeared in the formula: `a(n+1)=(2*n-1)/(2*n+1)*a(n)`. Then we adjusted the counting in the loop to fit by starting at $n = 1$, which loads $a(1+1) = a(2)$ and ending at $n = 19$, which loads $a(19+1) = a(20)$. Always make your code fit the mathematics as closely as possible, then adjust the supporting variables and structures to fit. This will make your code easier to read and you will make fewer mistakes.

4.2 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` statement, which works like this:

Listing 4.4 (ch4ex4.m)

```
clear; close all;
a=1;
b=3;

if a>0
   c=1  % If a is positive set c to 1
else
   c=0  %if a is 0 or negative, set c to zero
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
```

Study each of the commands in the code above and make sure you understand what they do. You can build any logical condition you want if you just know the basic logic elements listed in Table 4.1.

Equal	==
Less than	<
Greater than	>
Less than or equal	<=
Greater than or equal	>=
Not equal	~=
And	&
Or	|
Not	~

Table 4.1 Matlab's logic elements

`while` loops

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

Listing 4.5 (ch4ex5.m)

```
clear; close all;

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` loops are awkward to use because you can get stuck in an infinite loop if your check condition is never false. The `break` command is designed to help you here. When `break` is executed in a loop the script jumps to just after the `end` at the bottom of the loop. The `break` command also works with `for` loops. Here is our sum loop rewritten with `break`

⚠ If you get stuck in an infinite loop, press Ctrl-C in the Command Window to manually stop the program and return control back to you.

Listing 4.6 (ch4ex6.m)

```
clear; close all;

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-100  % set a ridiculously small term.
                     % Don't really do this, as you
                     % only have 15 digits of precision.

   n=n+1; % add 1 to n so that it will count: 2,3,4,5,...
   term=1/n^2;
   s=s+term;

   if (n > 1000) % Break stop if it is taking too long
```

```
       disp('This is taking too long.  I''m out of here...')
       break
   end
end  % end of the loop

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

Chapter 5

Basic Plotting

One of the nicest features in Matlab is its wealth of visualization tools. In this chapter we'll learn how to use several common plots, but there are many more that Matlab can make for you besides these.

5.1 Linear Plots

Making a Grid

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

```
x=0:0.01:10;
```

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

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

☞ The semicolon at the end of the `x=0:.01:10;` line is crucial, unless you want to watch 1001 numbers scroll down your screen. If you make this mistake on a very large matrix and the screen print is taking forever, Ctrl-C will rescue you.

Plotting the Function

Once you have two arrays of the same size, you plot `y` vs. `x` like this

```
plot(x,y);
```

And what if you want to plot part of the `x` and `y` arrays? The `colon` and `end` commands can help. Try the following code to plot the first and second half separately:

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

Controlling the Axes

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

```
axis([0 10 -1.3 1.3])
```

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

```
plot(x,y)
xlim([0 10])
```

or `ylim`:

```
plot(x,y)
ylim([-1.3 1.3])
```

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

```
axis equal
```

Logarithmic Plots

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

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

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

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

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

3-D Line Plots

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.

Listing 5.1 (ch5ex1.m)

```
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)
axis equal
```

5.2 Plot Appearance

Line Color and Style

To specify the color and line style of your plot, use the following syntax

```
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. Many other colors and line styles are possible, and instead of connecting the dots you can plot symbols at the points with various line styles between the points. For instance, if you type

```
plot(x,y,'g.')
```

you get green dots at the data points with no connecting line. To see what the possibilities are type

```
help plot
```

in the command window.

Labeling your plots

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

```
xlabel('Distance (m)')
ylabel('Amplitude (mm)')
```

And to put a title on you can do this:

```
title('Oscillations on a String')
```

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 also 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('Oscillations with k=%g',5)
title(s)
```

In this example we hard-coded the number 5, but you can do the same thing with variables.

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. To print Greek letters just type their names preceded by a backslash, like this:

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

And to put a title on you can do this:

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

To force LaTeX symbols to come through correctly when using `sprintf` you have to use two backslashes instead of one.

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

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 `^` character: use `\theta^{10}` to print θ^{10}.

α	\alpha
β	\beta
γ	\gamma
δ	\delta
ϵ	\epsilon
ϕ	\phi
θ	\theta
κ	\kappa
λ	\lambda
μ	\mu
ν	\nu
π	\pi
ρ	\rho
σ	\sigma
τ	\tau
ξ	\xi
ζ	\zeta

Table 5.1 The lowercase Greek letters in LaTeX

5.3 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

```
close all;
```

```
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 the plot windows 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. If you want this to happen while Matlab is executing other code (say a long loop), you need to put in a short pause command, say `pause(0.1)` to give the computer a chance to switch from executing the code to drawing the plot.

Overlaying Plots

Often you will want to overlay two plots on the same set of axes. Here's the first way—just ask for multiple plots on the same plot line

```
plot(x,f1,'r-',x,f2,'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,f1,'r-')
hold on
plot(x,f2,'b-')
title('Second way')
hold off
```

The second way is convenient if you have lots of plots to put on the same figure. Remember to release the hold using the command `hold off` as shown in the example or every subsequent plot will be on the same axis.

Subplots

It is often helpful to put multiple plots in the same figure, but on separate axes. The command to produce plots like this is `subplot`, and the syntax is `subplot(rows,columns,plotNumber)` 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. For instance, to make two-row figure, do this

```
subplot(2,1,1)
plot(x,f1)
subplot(2,1,2)
plot(x,f2)
```

Chapter 6

Grids and Plots in Multiple Dimensions

Matlab will 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. You can also display functions like $\mathbf{F}(x, y)$, where $\mathbf{F}$ is a vector-valued function, using vector field plots.

6.1 Making 2-D Grids

Before we can display 2-dimensional data, we need to define arrays `X` and `Y` that span the region that you want to plot, then create the function $F(x, y)$ over the plane. First, you need 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`. Begin by executing the following example.

Listing 6.1 (ch6ex1.m)

```
clear; close all;

% Define the arrays x and y
% Don't make the step size too small or you will kill the
% system (you have a large, but finite amount of memory)
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);

% Plot the function
surf(X,Y,F);
xlabel('x');
ylabel('y');
```

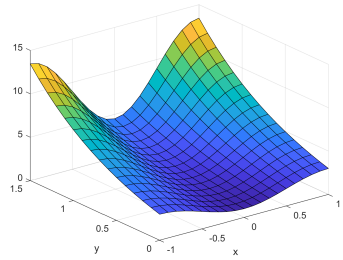


Figure 6.1 Output from Listing 6.1

The figure this code produces 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]
y=[4,5]
```

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

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \qquad Y = \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \end{bmatrix}. \tag{6.1}$$

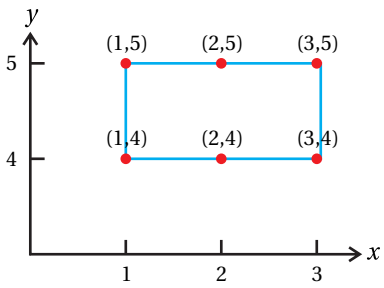


Figure 6.2

Compare these matrices to Fig. 6.2 to see how they correspond to the small area of x-y space that we are representing. Note that the y dimension entries are flipped from what you might expect because of the difference in conventions for writing matrices (left-to-right and top-to-bottom) versus plotting functions (left-to-right and bottom-to-top).

With `meshgrid`, the first index of both `X` and `Y` is a y-index, since matrix elements are indexed using the `X(row,column)` convention. For example, the elements `X(1,1)` and `X(2,1)` both have value 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 and y having index j, then the two-dimensional matrices have indices

$$X(j,i) \quad Y(j,i). \tag{6.2}$$

But in physics we often 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

$$X(i,j) \quad Y(i,j) \tag{6.3}$$

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

Matlab has a 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

$$X = \begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix} \qquad Y = \begin{bmatrix} 4 & 5 \\ 4 & 5 \\ 4 & 5 \end{bmatrix} \tag{6.4}$$

These matrices have the indexes in the $X(i,j)$, $Y(i,j)$ order, but lose the spatial correlation that `meshgrid` gives between Eq. (6.1) and Fig. 6.2.

Plots made either with `surf(X,Y,F)` or `contour(X,Y,F)` (discussed below) will look the same with either grid. However, `streamline` plots require your data to be in the format provided by `meshgrid`. You will need to be familiar with both methods of creating a grid. You will have need to do it both ways, depending on the circumstance.

6.2 Surface Plots

Now that we understand how to make two-dimensional grids, let's make some plots. Run the following example, read through the comments and watch what it does.

Listing 6.2 (ch6ex2.m)

```
clear; close all;

% Make the grid
x=-1:0.1:1;
y=0:0.1:1.5;
[X,Y]=ndgrid(x,y);
F=(2-cos(pi*X)).*exp(Y);

% Now make a surface plot of the function
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. You can also programmatically set the viewing angle with the `view` command. Here's a piece of code that flies around the surface plot by continually changing the viewing angles and using the pause command; we think you'll be impressed.

Listing 6.3 (ch6ex3.m)

```
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
```

The pause command in the loop allows Matlab the time to draw the plot. If you don't put it in, Matlab will not pause to draw the picture each iteration. This same trick will let you make animations of both xy and surface plots.

6.3 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.

Listing 6.4 (ch6ex4.m)

```
clear; close all;

x=-5.25:0.5:5.25;
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);

quiver(X,Y,Ex,Ey) % make the field arrow plot
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, but keeping direction)
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.
```

```
Bmin=min(min(B));
Bmax=max(max(B));
% s is the desired ratio between the longest arrow and the shortest one
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.

6.4 Streamlines

In addition to plotting little arrows at each point in the vector field, you can plot "streamlines." For fluid dynamics, streamlines show the path that a particle would follow if the arrows at each point represent the fluid velocity at that point. In electricity and magnetism, flow lines for the electric and magnetic fields are the field lines that you learned about in your introductory electricity and magnetism course.

As mentioned above, the streamline plotting routine assumes that the grid data was created in the `meshgrid` format rather than the `ndgrid` format.

Listing 6.5 (ch6ex5.m)

```
clear; close all;

%Define the position arrays x and y
[x,y] = meshgrid(0:0.1:1,0:0.1:1);

%Define the flow velocity arrays u and v
u = x;
v = -y;

%Create a quiver plot of the flow velocity
figure
quiver(x,y,u,v)

%Plot streamlines that start at different points along the line y=1.
startx = 0.1:0.1:1;
starty = ones(size(startx));
streamline(x,y,u,v,startx,starty);
```

Chapter 7

Make Your Own Functions

You will often need to build your own Matlab functions as you use Matlab to solve problems. You can do this either by putting simple expressions into your code by using Matlab's anonymous function syntax, or by defining function files with .m extensions called m-file functions.

7.1 Anonymous 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

$$f(x,y) = \frac{\sin(xy)}{x^2+y^2} \tag{7.1}$$

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

Listing 7.1 (ch7ex1.m)

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

x=-8:0.1:8;
y=x;

plot(x,f(x,2))

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

The second line in this listing creates a function `f` that is stored only in memory. The symbol `@` tells Matlab that the variable `f` contains a reference to a function rather than a numeric value, and the items in parentheses afterwards indicate that this function takes two arguments: `x` and `y`. The code

```
sin(x.*y)./(x.^2+y.^2)
```

defines how the input values are used to create output values.

Anonymous functions are convenient in many cases, but to use them you must be able to define your function in a single statement of Matlab code. Often this is not possible because your functions will require logic or loops. In these cases, you'll need to use the more flexible m-file functions.

7.2 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; for example, think about `sin(x)` or `plot(x,y,'r-')`. Here is an example of how you can code the function in (7.1) as an m-file function rather than anonymous function.

Listing 7.2 (trig.m)

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

f=sin(x.*y)./(x.^2+y.^2);
```

The first line of the function file is of the form

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

The word `function` is required, `output` is the name of the variable that the function passes back to whomever called it, `name` should match the filename of the script (e.g. "trig.m" above), and `input` is the argument list of the function. When the function is called, the arguments passed in must match in number and type defined in the definition. The m-file function must assign the output an appropriate value before finishing.

To call the function, you to write a separate script like this

Listing 7.3 (calling.m)

```
clear; close all;

h=0.1;
x=-8:h:8;
y=x;
plot(x,trig(x,2))

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

The names of the input and output variables of a function are local to the function, so you don't have to worry about overwriting variables in the file that called the function. However, this also means that *the variables inside Matlab functions are invisible in the command window or the calling m-file.* For example, you cannot reference the variable `f` in the calling.m script, nor can you access the variable `h` in the trig.m script.

If you want variables used outside the function to be visible inside the function and vice-versa, use Matlab's `global` command. This command declares 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 `trig.m` and in `trig.m` itself, then if you give `a`, `b`, and `c` values in the main script, they will also have these values inside `trig.m`. This construction will be especially useful when we use Matlab's differential equation solving routines in Chapter 9.

You should be aware that using global variables like this is regarded as something of a hack. It works, and is often an easy way to get code written but it can make your code harder to debug, especially when you start working on larger projects. Also, *using global variables is a performance bottleneck* that can slow your code down. If performance matters, you'll want to pass values in to functions as arguments rather than making variables global. We use globals here to simplify your life, but simple isn't always the best when coding.

⚠ Use global variables with care. They can make your code hard to debug, and can cause performance issues.

7.3 Functions With Multiple Outputs

Here is an example of a function that returns more than one output variable. This function takes as input an integer `n`, a width `a` in nanometers, and an integer `NPoints` and returns the energy level and wavefunction for a infinite square well of width `a`.

Listing 7.4 (SquareWell.m)

```
function [x,psi,E] = SquareWell(n,a,NPoints)
% Calculate the energy and wavefunction for an
% electron in an infinite square well
%
% Inputs: n is the energy level; must be a positive integer
%         a is the well width in nanometers
%         NPoints is the number of points in the x grid
%
% Outputs: x is the grid for the plot, measured in nanometers
%          psi is the normalized wave function
%          E is the energy of the state in electron-volts

% Make the x-grid
xmin = 0;
xmax = a;
x = linspace(xmin,xmax,NPoints);

% Wave number for this energy level
k = n * pi / a;

% Calculate the wave function
psi = sqrt(2/a) * sin(k*x);

global hbar m

% Calculate energy in electron-volts
E = n^2*pi^2*hbar^2 / (2*m*(a*1e-9)^2) / 1.6e-19;
```

Note that when a function returns more than one output variable, the names of these results are put in square brackets, as in `SquareWell.m`. When you call the function, you use the same type of format to receive the output, as in this example

Listing 7.5 (ch7ex5.m)

```
clear; close all;

global hbar m
% Constants in MKS units
hbar = 1.05e-34;
m=9.11e-31;

% remember that n must be a positive integer
n=3;

% Set the width to an Angstrom
a=0.1;

% Get the values
[x,psi,Energy] = SquareWell(n,a,100);

% Make the plot and label it
plot(x,psi)
s=sprintf('\\Psi_%g(x); a=%g nm; Energy=%g eV',n,a,Energy);
title(s);
xlabel('x (nm)');
ylabel('\Psi(x)');
```

Note that we didn't have to call the variables the same names when we used them outside the function in the main script (e.g. we used `Energy` instead of `E`). This is because all variables inside functions are local to these programs and Matlab doesn't even know about them in the command window. Confirm this by trying to display the value of `E` in the command window.

```
E
```

However, note that we did declare the variables `hbar` and `m` as global variables so that the values that we set in our calling script are available inside the function also.

Chapter 8

Derivatives and Integrals

In numerical physics we represent functions like $f(x)$ as discrete points on a grid. If you are careful, you can use these discrete values to quickly give you numerical approximations to the derivative $f'(x)$ and the integral $\int_a^b f(x)dx$.

8.1 Derivatives

The instructor of your first calculus class probably introduced the derivative with a formula like this:

$$\frac{df}{dx} = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}. \tag{8.1}$$

This is referred to as a *forward difference* derivative, since the fraction inside the limit looks forward to a point $x+h$ to calculate the slope and then considers what happens as h gets small. When doing numerics, it is usually better to use the following *centered difference* formula:

$$\frac{df}{dx} = \lim_{h\to 0} \frac{f(x+h) - f(x-h)}{2h}. \tag{8.2}$$

Eqs. (8.1) and (8.2) both converge to the same result in the mathematical limit $h \to 0$, but Eq. (8.2) usually converges faster. This means that for a given h you can get a more accurate approximation for the derivative with the centered difference formula than you can with the forward difference. To see the importance of centering, consider Fig. 8.1. In this figure we are trying to find the slope of the tangent line at $x = 0.4$. The forward difference formula uses the data points at $x = 0.4$ and $x = 0.5$, giving tangent line a. Notice that using the "centered" pair of points $x = 0.3$ and $x = 0.5$ to obtain tangent line b is a much better approximation.

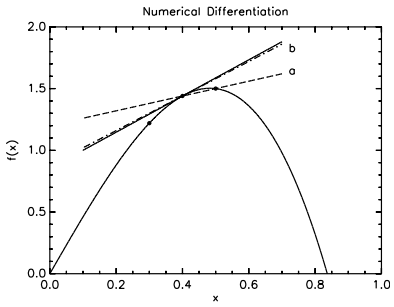


Figure 8.1 The centered derivative approximation works best.

As an example of the how good the centered difference formula can work, differentiate $\sin x$ this way:

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

Now 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)
```

You can also take the second derivative numerically using the centered difference formula

$$\frac{d^2f}{dx^2} = \lim_{h\to 0} \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}. \tag{8.3}$$

For example,

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

Again, we 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

$$f'(x_j) \approx \frac{f_{j+1} - f_{j-1}}{2h}. \tag{8.4}$$

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

Listing 8.1 (ch8ex1.m)

```
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
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')
```

Here is an example of a function that takes as inputs an 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.

Listing 8.2 (derivs.m)

```
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

% This is how you could do linear extrapolation
%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 extrapolation (better than linear)
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 see how to call this function, you can use the following script

Listing 8.3 (ch8ex3.m)

```
clear;
close all;

% 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);
```

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

Matlab also has its own routines for doing derivatives; look in online help for `diff` and `gradient`.

8.2 Integrals

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 method is shown in Fig. 8.2. Notice that this method should work pretty well over subintervals like [1.0,1.5] where $f(x)$ is nearly a straight line, but probably is lousy over subintervals like [0.5,1.0] where the function curves.

Listing 8.4 (ch8ex4.m)

```
clear; 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)
```

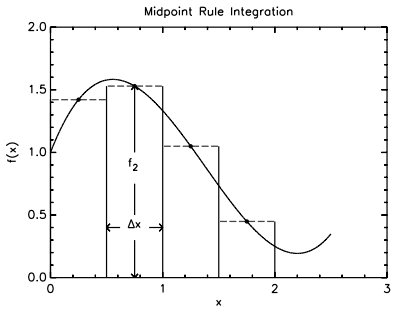


Figure 8.2 The midpoint rule works OK if the function is nearly a straight line across each interval.

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.

Indefinite integrals

And what if you need to find the indefinite integral? Remember that an indefinite integral seeks the function that is the "anti-derivative" of a known function. The fundamental theorem of calculus relates indefinite integrals to the cumulative area under a function, which Matlab can easily find numerically. If you have

arrays for x and $f(x)$, you can quickly approximate the indefinite integral function $\int_a^x f(x')dx'$ using Matlab's `cumtrapz` functions. This function takes an array of function values in `y` and an x-spacing `dx` and returns an approximate indefinite integral function $g(x) = \int_a^x y(x')dx'$ using the trapezoid rule. This 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:

$$\int_x^{x+h} y(x')dx' \approx \frac{y(x)+y(x+h)}{2}h \tag{8.5}$$

Because `cumtrapz` uses the trapezoid rule instead of the midpoint rule, the array of function values must start at $x = a$ and be defined at the edges of the subintervals of size h rather than at the centers. For example, if we store the values of $\cos(x)$ like this:

```
dx=pi/20;
x=0:dx:2*pi;
f=cos(x);
```

then you can calculate the indefinite integral $g(x) = \int \cos(x)dx$ like this:

```
g=cumtrapz(f)*dx;
```

where `dx` is the point spacing. You can compare the results with the exact indefinite integral ($g(x) = \sin(x)$ by plotting the two:

```
plot(x,g,x,sin(x),'*')
```

Here is a function that does essentially the same calculation as `cumtrapz`, but is coded to be readable. This function takes `dx` as an argument and multiplies it at each step of the calculation, whereas `cumtrapz` assumes a step size of one. This code is more intuitive, but less efficient. Study this code to see how it works, but use `cumtrapz` and multiply by `dx` as above when you actually do calculations, as it will be faster.

Listing 8.5 (indefint.m)

```
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
```

8.3 Matlab Integrators

For simple integrals of functions sampled at discrete points (e.g. measured data) it is usually fine to use `trapz` to calculate a definite integral, or `cumtrapz` to calculate an indefinite integral. Both functions treat the function as little line segments connecting the data points in your array. Performing integrals on raw measured data is usually a bad idea unless your data is very clean. If your data is noisy, you'll probably want to consider smoothing it or fitting it to a curve before trying to take an integral.

If you want to calculate the integral of a function rather than data stored in an array, you have more options because you can calculate function values at arbitrary points rather than having to interpolate between array elements. For this type of problem, use the command `integral` (or `integral2`, or `integral3` for multiple dimension integrals). These functions take references to a Matlab function in their arguments (like `fzero`, `fsolve`, and `fminsearch` did) and adjust the grid step size to optimize accuracy.

The Matlab routine `integral` adaptively approximates the function with parabolas (as shown in Fig. 8.3) or other functions instead of the rectangles of the midpoint method. The parabolic method is called Simpson's Rule. As you can see from Fig. 8.3, parabolas do a much better job, making `quad` a standard Matlab choice for integration. To use the `integral` command, you need to define your function using an anonymous function. For example, to define an anonymous function to represent the function $f(x) = \cos(x)e^{-x}$, you use this syntax:

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

After you have defined `f` this way, you can use it just like built-in Matlab functions like `sin` and `cos`. For instance, try these commands after defining `f` as above:

```
f(1)
```

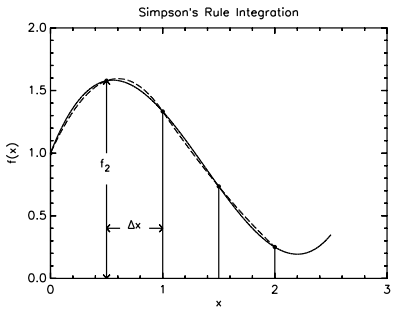


Figure 8.3 Fitting parabolas (Simpson's Rule) works better than the midpoint method.

```
f(1:10)
```

You can use the anonymous function syntax to integrate $\cos(x)e^{-x}$ from 0 to 2 using the `integral` function like this:

Listing 8.6 (ch8ex6.m)

```
clear; close all;

f=@(x) exp(-x).*cos(x)
integral(f,0,2,'AbsTol',1e-8)
```

Matlab also has a command `integral2` that does double integrals. Here's how to use it to integrate the function $f(x,y) = \cos(xy)$.

Listing 8.7 (f2int.m)

```
clear; close all;

f = @(x,y) cos(x.*y);
integral2(f,0,2,0,2,'AbsTol',1e-10)
```

Chapter 9

Ordinary Differential Equations

Matlab provides some powerful numerical solvers and gives you a lot of control over how they work. It is often more efficient to use Matlab for hard differential equations than to use Mathematica. In this chapter we review the methods used by numerical differential equation solvers and write a couple of crude solvers to see how they work. Then we introduce you to the solvers that Matlab provides.

9.1 General form of Ordinary Differential Equations

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

$$\frac{d\mathbf{u}}{dt} = \mathbf{F}(\mathbf{u}) \tag{9.1}$$

where $\mathbf{u}$ is a vector of unknown functions of the parameter t (often, but not always, t is 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 functions x, y, and z we would write

$$\frac{d}{dt}\begin{pmatrix} u_1(t) \\ u_2(t) \\ u_3(t) \end{pmatrix} = \begin{pmatrix} f_1(u_1,u_2,u_3) \\ f_2(u_1,u_2,u_3) \\ f_3(u_1,u_2,u_3) \end{pmatrix}. \tag{9.2}$$

where

$$\begin{aligned} u_1 &\equiv x \\ u_2 &\equiv y \\ u_3 &\equiv z \end{aligned} \tag{9.3}$$

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

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

$$\frac{dN}{dt} = -\gamma N \tag{9.4}$$

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

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

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

Simple Harmonic Oscillator

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

$$\frac{d^2x}{dt^2} = -\omega_0^2 x \quad \text{where} \quad \omega_0 = \sqrt{\frac{k}{m}} \tag{9.5}$$

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:

$$\begin{aligned} \frac{dx}{dt} &= v \\ \frac{dv}{dt} &= -\omega_0^2 x\,. \end{aligned} \tag{9.6}$$

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 **u** from Eq. (9.1) is

$$\mathbf{u} = \begin{pmatrix} x \\ v \end{pmatrix}. \tag{9.7}$$

In this notation, our system of equations becomes

$$\begin{aligned} \frac{du_1}{dt} &= u_2 \\ \frac{du_2}{dt} &= -\omega_0^2 u_1 \end{aligned} \tag{9.8}$$

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

$$\mathbf{F}(\mathbf{u}) = \begin{pmatrix} u_2 \\ -\omega_0^2 u_1 \end{pmatrix} \tag{9.9}$$

9.2 Solving ODEs numerically

So let's assume that we have a first order set. How can we solve it? 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 which are better, but you will better appreciate what they can do if you learn the ideas they are based on by studying these two methods.

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

$$\frac{\mathbf{u}_{n+1} - \mathbf{u}_n}{\tau} = \mathbf{F}(\mathbf{u}_n, t_n) \ . \tag{9.10}$$

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 $t_{n+1} = t_n + \tau$:

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \mathbf{F}(\mathbf{u}_n, t_n)\tau \ . \tag{9.11}$$

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. (9.6) as a first order system until you can see that this is true.) Here's a script that uses Euler's method to solve the harmonic oscillator equation.

Listing 9.1 (ch9ex1.m)

```
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 - ')

% Pre-allocate the arrays to make the code faster
t=zeros(1,N+1);
x=zeros(1,N+1);
v=zeros(1,N+1);

% 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 run this code 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 $N = 500{,}000$.

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:

$$\frac{\mathbf{u}_{n+1} - \mathbf{u}_n}{\tau} = \mathbf{F}(\mathbf{u}_n, t_n) \,. \tag{9.12}$$

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

$$\frac{\mathbf{u}_{n+1} - \mathbf{u}_n}{\tau} = \mathbf{F}(\mathbf{u}_{n+1/2}, t_{n+1/2}) \,. \tag{9.13}$$

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. 9.1, 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.

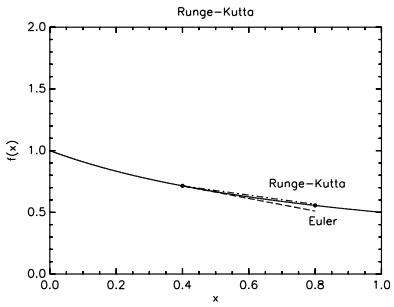


Figure 9.1 Runge-Kutta anticipates curving and beats Euler.

Listing 9.2 (ch9ex2.m)

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

% 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-')
```

9.3 Matlab's Differential Equation Solvers

Matlab has its own differential equation solvers and they are more accurate than the simple methods discussed above. Table 9.1 shows a list of these Matlab functions and what they do. All of these functions work in the same basic way:

1. You define the right-hand side function for your set of first order differential equations in an m-file, say `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. The differential equation solver then 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 `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);
```

   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.

An important point that sometimes hangs students up is that the data points that Matlab returns will not be equally spaced in time. If you just want the solution at certain pre-set times $[t_n]$, replace the 2-element array `[tstart,tfinal]` with an array of the times that you want: `[t1,t2,...,tN]`. For example, you could replace `[tstart,tfinal]` with the equally spaced array of times given by `tstart:dt:tfinal`. Alternately, you can use Matlab's interpolation functions to interpolate the output of `ode45` onto an equally spaced time grid. We'll illustrate this method in the next chapter.

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

`ode23`	An explicit, one-step Runge-Kutta low-order (2-3) solver. (Like the second-order Runge-Kutta method predictor-corrector discussed above.) 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 9.1 List from *Mastering Matlab 6* of some of Matlab's differential equation solvers.

equation. Notice that in `rhs.m` we have a bunch of comments to remind ourselves what each slot in the u vector represents. We recommend you get in this habit, or you will drive yourself crazy trying to debug your code.

Listing 9.3 (ch9ex3.m)

```
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);

% plot the position vs. time
plot(t,x)
title('Position vs. Time')

% make a "phase-space" plot of v vs. x
figure
plot(x,v)
title('Phase Space Plot (v vs. x)')
```

Listing 9.4 (rhs.m)

```
function F=rhs(t,u)
% right-hand side function for Matlab's ODE solver,
% simple harmonic oscillator example

% Write comments to remind yourself how the variables are arranged in u.
% In our case we will use:
%     u(1) -> x
%     u(2) -> v

% 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);
```

You can write the `rhs` function more compactly using anonymous functions. For the example above, you could do this as

```
rhs = @(t,u)[u(2);-w0^2*u(1)];

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

rather than writing a separate m-file function. The anonymous function method has the feature that there are no global variables to cause interesting debugging problems. It is also much more compact. But for more complicated systems of differential equations, it can become hard to read the code.

9.4 Event Finding with 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 we 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`.

Listing 9.5 (ch9ex5.m)

```
% 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), u(:,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
```

Listing 9.6 (eventrhs.m)

```
function rhs=eventrhs(t,u,omega)
%  eventrhs.m, Matlab function to compute [du(1)/dt;du(2)/dt]

% 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);
```

Listing 9.7 (events.m)

```
function [value,isterminal,direction] = events(t,u,omega)
% function to control event finding by Matlab's ode solvers
% NOTE: the argument list for the events function, i.e. (t,u,omega), must
% be exactly the same as the argument list for the right-hand side function.
% If they don't match you will get the error "Too many input arguments"


% 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

% load value(1) with the expression which,
% when it is zero, defines the event, u(1)=0 in this case.
value(1,1) = u(1);

% 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.
value(2,1)=u(2);


% 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
isterminal = [0 ; 0];


% 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
direction = [1 ; -1];
```

Chapter 10

Interpolation and Extrapolation

In computational physics we usually represent functions as arrays of values at discrete points in time and space. But we often want to be able to find 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.

10.1 Manual Interpolation and Extrapolation

Linear approximation

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 the two-point formula for a line is

$$y = y_1 + \frac{(y_2 - y_1)}{(x_2 - x_1)}(x - x_1) \tag{10.1}$$

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. 10.1, 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

$$f_{N+1} = 2f_N - f_{N-1} \tag{10.2}$$

You must be careful here as well: segment "d" in Fig. 10.1 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 $x = 2$.

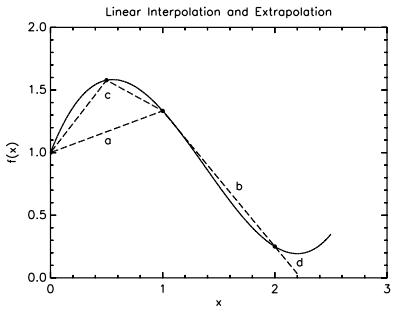


Figure 10.1 Linear interpolation only works well over intervals where the function is straight.

Quadratic approximation

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. 10.2. 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.

To derive the quadratic interpolation and extrapolation function, we assume that we have three known points, (x_1, y_1), (x_2, y_2), and (x_3, y_3). If our parabola $y = ax^2 + bx + c$ is to pass through these three points, then the following set of equations must hold

$$\begin{aligned} y_1 &= ax_1^2 + bx_1 + c \\ y_2 &= ax_2^2 + bx_2 + c \\ y_3 &= ax_3^2 + bx_3 + c \end{aligned} \tag{10.3}$$

Unfortunately, when you solve this set of equations for a, b, and c, the formulas are ugly. If we simplify to the case where the three points are part of an equally spaced grid, things are prettier. Let's assume equally spaced points spaced by h, so that $x_1 = x_2 - h$ and $x_3 = x_2 + h$. In this case, the solutions are[1]

$$\begin{aligned} a &= \frac{y_1 - 2y_2 + y_3}{2h^2} \\ b &= \frac{y_3 - y_1}{2h} - 2x_2 a \\ c &= y_2 + x_2 \frac{y_1 - y_3}{2h} + x_2^2 a \end{aligned} \tag{10.4}$$

With these coefficients, we can quickly find approximate y values near our three points using $y = ax^2 + bx + c$. This formula is very useful for getting function values that aren't in the array. For instance, we can use this formula to obtain the interpolation approximation for a point half way between two known points, i.e. $y_{n+1/2} \equiv y(x_n + h/2)$

$$y_{n+1/2} = -\frac{1}{8} y_{n-1} + \frac{3}{4} y_n + \frac{3}{8} y_{n+1} \tag{10.5}$$

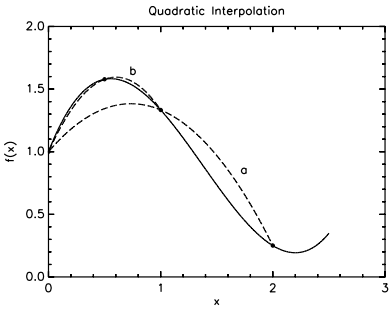


Figure 10.2 Quadratic interpolation follows the curves better if the curvature doesn't change sign.

[1] It is common in numerical analysis to derive this result using Taylor's theorem, which approximates the function $y(x)$ near the point $x = a$ as

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

If we ignore all terms beyond the quadratic term in $(x-a)$) near a point (x_n, y_n), use an array of equally spaced x values, and employ numerical derivatives as discussed in Chapter 8, the Taylor's series becomes

$$y(x) \approx y_n + \frac{y_{n+1} - y_{n-1}}{2h}(x - x_n) + \frac{y_{n-1} - 2y_n + y_{n+1}}{2h^2}(x - x_n)^2 .$$

This can be solved to find a, b, and c.

and also to find the quadratic extrapolation rule for a data point one grid spacing beyond the last point, i.e. $y_{N+1} \equiv y(x_N + h)$

$$y_{N+1} = 3y_N - 3y_{N-1} + y_{N-2} \,. \tag{10.6}$$

10.2 Matlab interpolaters

Interp1

Matlab has its own interpolation routine `interp1` which does the things discussed in the previous two sections 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,'pchip')
yi=interp1(x,y,xi,'spline')
```

We haven't talked about spline interpolation yet. It is a piece-wise polynomial fit that typically does an excellent job of matching smooth functions.

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

Listing 10.1 (ch10ex1.m)

```
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,'pchip');

% 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')
```

10.3 Two-dimensional 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 $\{x, y\}$. In the completely general situation where your data points $\{x, y, z\}$ don't fall on a regular grid, you can use the command `TriScatteredInterp` to interpolate your function onto an arbitrary new set of points $\{x_i, y_i\}$, such as an evenly spaced 2-dimensional grid for plotting. Examine the code below to see how `TriScatteredInterp` works, and play with the value of `N` and see how the interpolation quality depends on the number of points.

Listing 10.2 (ch10ex2.m)

```
clear; close all;

% Make some "data" at random points x,y points
N=200;
x = (rand(N,1)-0.5)*6;
y = (rand(N,1)-0.5)*6;
z = cos((x.^2+y.^2)/2);

% Create an interpolating function named F
F = TriScatteredInterp(x,y,z,'natural');

% Create an evenly spaced grid to interpolate onto
xe = -3:.1:3;
ye = xe;
[XE,YE] = ndgrid(xe,ye);

% Evaluate the interpolation function on the even grid
ZE = F(XE,YE);

% plot the interpolated surface
surf(XE,YE,ZE);

% overlay the "data" as dots
hold on;
plot3(x,y,z,'.');
axis equal
```

The `TriScatteredInterp` command is very powerful in the sense that you can ask it to estimate $z(x,y)$ for arbitrary x and y (within your data range). However, for large data sets it can be slow. In the case that your data set is already on a regular grid, its much faster to use the `interpn` command, like this:

Listing 10.3 (ch10ex3.m)

```
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]=ndgrid(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.

% Because the interpolation is linear the mesh is finer
% but the crude corners are still there
xf=-3:.1:3;
yf=xf;
[XF,YF]=ndgrid(xf,yf);
ZF=interpn(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, so interpolation doesn't work well

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

In this example our grids were created using `ndgrid`. If you choose to use the `meshgrid` command to create your grids, you'll need to use the command `interp2` instead of `interpn`.

Chapter 11

Linear Algebra and Polynomials

Since Matlab is convinced that every variable in the world is a matrix, you won't be surprised that it has about every matrix function you can imagine. 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.

11.1 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

$$\begin{aligned} x + z &= 4 \\ -x + y + z &= 4 \\ x - y + z &= 2\,, \end{aligned} \tag{11.1}$$

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 like this

```
A=[ 1, 0,1
   -1, 1,1
    1,-1,1 ];
b=[4
   4
   2];
```

Then using the backslash symbol \ (sort of "backwards divide") Matlab will use Gaussian elimination to solve this system of equations, like this:

```
x=A\b
```

11.2 Matrix Operations

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 load `C` with the inverse of `A` like this

```
C=inv(A)
```

We can verify by matrix multiplication that A*C is the identity matrix

```
A*C
```

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]'
```

Flipping Matrices

Sometimes you want to flip a matrix along the horizontal or vertical directions. The command to do this are

```
fliplr(A)   % flip A, left column becomes right, etc.
```

and

```
flipud(A)  % flip A, top row becomes bottom, etc.
```

Determinant

Find the determinant of a square matrix this way

```
det(A)
```

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
```

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
```

Special Matrices

Matlab will let you load several special matrices. A few of the more useful ones are the identity matrix

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

the zeros matrix

```
Z=zeros(5,5) % load Z with a 5x5 matrix full of zeros
```

the ones matrix

```
X=ones(3,3) % load X with a 3x3 matrix full of ones
```

and the random matrix

```
% load Y with a 4x6 matrix of random numbers between 0 and 1
% The random numbers are uniformly distributed on [0,1]
Y=rand(4,6)
```

```
% load Y with a 4x6 matrix of random numbers with a Gaussian
% distribution with zero mean and a variance of 1
Y=randn(4,6)

% And to load a single random number just use
r=rand
```

11.3 Vector Operations

Dot and Cross Products

Matlab will do vector 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. Note that the `dot` function is not the same thing as the "dot times" operator (`.*`). Type

```
dot(a,b)
a.*b
```

and compare the output. Explain the difference to someone sitting nearby.

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)
```

11.4 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.

Roots of a Polynomial

The following command will find the roots of a polynomial:

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

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)
```

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.

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

$$\frac{x^2+x+1}{x+1} = x + \frac{1}{x+1}. \tag{11.2}$$

First Derivative

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

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

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)
```

Chapter 12

Fitting Functions to Data

A common problem that physicists 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`. If you want to fit to an arbitrary functions containing sines, cosines, exponentials, logs, etc., then the process is a little more complicated. Pay attention to this section; it is useful.

12.1 Fitting Data to a Polynomial

If you have some data in the form of arrays (x, y) (perhaps read in with Matlab's `load` command), 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.

Listing 12.1 (ch12ex1.m)

```
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-')
```

Interpolating With `polyfit` and `polyval`

You can also use Matlab's polynomial commands 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.

Listing 12.2 (ch12ex2.m)

```
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);

% 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')
```

12.2 General Fits with `fminsearch`

If you want to fit data to a more general function than a polynomial, you need to work a little harder. Suppose we have a set of data points (x_j, y_j) 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

$$f(x, a_1, a_2) = a_1 e^{a_2 x} \ . \tag{12.1}$$

The first step in the fitting process is to make a m-file function, call it `funcfit.m`, that evaluates the function you want to fit to, like this The input `a` is a matrix containing the as-yet unknown parameters and `x` is a matrix with the independent variable for your fit. Make sure the `funcfit.m` function works properly on matrices, so that if `x` is a matrix input then `f` is a matrix of function values.

Our goal is to write some code to 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

$$S = \sum_{j=1}^{N} (f(x_j) - y_j)^2 \ . \tag{12.2}$$

We need to make another 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 you stored in the Matlab m-file `funcfit.m` above. Here is the form of the `leastsq.m` function.

Listing 12.3 (leastsq.m)

```
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);
```

With these two functions built and sitting in your Matlab directory we are ready to do the fit.

Matlab has a 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. The basic idea is that you pass a reference to your `leastsq.m` function to `fminsearch`, and it varies the fit parameters in the variable `a` in a systematic way to find the values that minimizes the error function S (calculated by `leastsq.m` from the (x,y) data and the a_n's).

Note, however, that `fminsearch` is a *minimizer*, not a zero finder. So it may find a local minimum of the error function which is not a good fit. If it fails in this way you need to make another initial guess so `fminsearch` can take another crack at the problem.

Here is a piece of code that performs the fitting functions. First, it loads the data from a file. For this to work, 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
```

Then the program asks 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 save it for future use.

It's a little more work to make three files to get this job done, but we suggest you learn how to use `fminsearch` this way. Function fitting comes up all the time. Once you get the hang of it, you can choose to fit to any function you like just by changing the definition in `funcfit.m`. The other two scripts usually don't need to be modified.

Chapter 13

Fast Fourier Transform (FFT)

In physics, we often need to see what frequency components comprise a signal. The mathematical method for finding the spectrum of a signal $f(t)$ is the Fourier transform, given by the integral

$$g(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{i\omega t} dt. \tag{13.1}$$

In chapter 8 we learned how to do integrals numerically, but the Fourier transform is hard because we usually want to calculate the integral in Eq. (13.1) for a lot of values of ω—often 100,000 or more. To calculate Eq. (13.1) this many times using regular integrating techniques is very time consuming, even for modern computers. Fortunately, some clever scientists came up with a much more efficient way to calculate a Fourier transform using the aptly-named Fast Fourier Transform (FFT) algorithm. Matlab knows this algorithm, and can perform it with the `fft` command.

13.1 Matlab's FFT

Suppose you have a signal stored as a series of N data points (t_j, f_j) equally spaced in time by time interval τ from $t = 0$ to $t = t_{\text{final}} = (N-1)\tau$. We'll assume that the data is in two arrays, an array `t` with the times and a corresponding array `f` with the signal strength at each time. The Matlab function `fft` will convert the array `f` into an array `g` 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. They are complex because they store the phase relationship between frequency components as well as amplitude information, just like the Fourier transform. 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.

If we don't care about the phase information contained in $g(\omega)$, we can work instead with the power spectrum $P(\omega) = |g(\omega)|^2$, obtained this way:

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

The `fft` routing runs fastest if you 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.

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 to plot $f(t)$. Later we'll derive the values for the frequency array that corresponds to g. For now we'll just tell you that the frequency interval from one point in g to the next is related to the time step τ from one point in f to the next via

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

The "regular" frequency ν (also called cyclic frequency) is measured in Hz, or cycles/second, and angular frequency $\omega = 2\pi\nu$ is measured in radians/second. 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);

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

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

Here is an example to show how this whole process works.

Listing 13.1 (ch13ex1.m)

```
clear; close all;

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

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

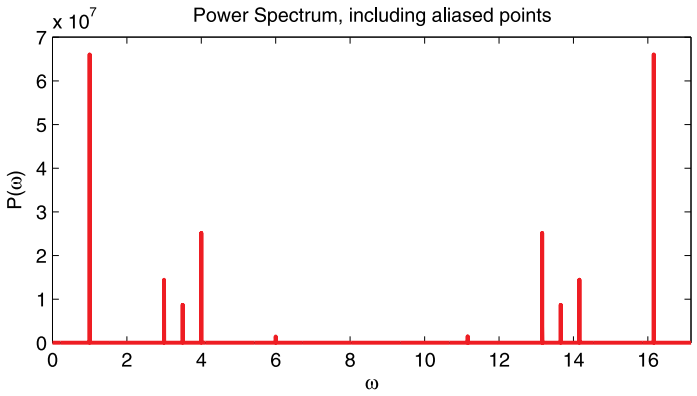


Figure 13.1 Plot of the example power spectrum

The power spectrum produced by Listing 13.1 is plotted in Fig. 13.1. It has peaks at $\omega = 1, 3, 3.5, 4, 6$ as we would expect, but there are some extra peaks on the right side. These extra peaks are due to a phenomenon called aliasing, which we'll discuss next. (This next material is hard, so read carefully.)

13.2 Aliasing and the Critical Frequency

In old western movies, stagecoach wheels sometimes appear to be turning backwards. This is because movies are made by taking a bunch of pictures separated by a time interval τ. We know the wheel rotates between pictures, but since all the spokes look the same, there are infinite possible rotations that are consistent with the change in spoke angles between sequential pictures (see Fig. 13.2). Each possible rotation corresponds to either a positive or negative rotation frequency. Our brains resolve the ambiguity by picking the frequency with the smallest magnitude, whether the sign is positive or negative, which accounts for the backwards stagecoach wheel effect. 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.

Imagine that we paint one of the spokes of a wagon wheel bright red (to remove multi-spoke effects), spin the wheel, and then sample the red spoke's position at $t = 0$ (dashed line) and $t = \tau$ (solid line) as illustrated in Fig. 13.3. The figure depicts three rotation angles that are consistent with what might have happened between our two measurements: the most "obvious" angle θ_0 that our brain would see in a movie; or it might have rotated a full positive revolution plus θ_0 to give $\theta_1 = \theta_0 + 2\pi$; or maybe the wheel was actually rotating backwards so that the rotation angle was $\theta_{-1} = \theta_0 - 2\pi$. These three angles of rotation correspond to

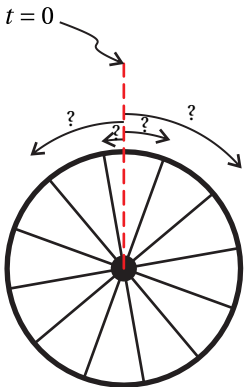


Figure 13.2 The stagecoach effect occurs when a spoke in an initial frame (indicated by the dashed line) could have rotated to many indistinguishable orientations in the next frame. The ambiguity of the wheel rotation is due in part the many spokes.

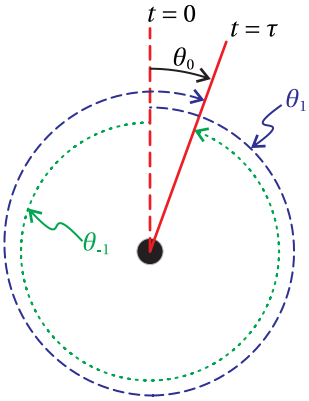


Figure 13.3 Even with one spoke, the ambiguity remains. In this figure, we show three of the possible rotation angles between frames.

three angular frequencies that are consistent with our data:

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

We can expand this thought process and write a general expression for all frequencies that are consistent with the wheel orientation at the two measured times as

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

Here n is any integer, with $n \geq 0$ corresponding to positive frequencies and $n < 0$ corresponding to negative frequencies. Notice that every possible frequency ω_n can be "aliased" by neighbor frequencies located $2\pi/\tau$ above or below the frequency of interest.

Now we are in a position to understand the extra peaks in Fig. 13.1. The Fourier transform of any real signal has an equal amount of positive and negative frequency content, i.e. $P(\omega) = P(-\omega)$. However, our original frequency window included only positive frequencies, so we didn't see the negative half of the spectrum. In Fig. 13.5 we've plotted the negative frequency content of the signal in Listing 13.1 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 is a result of an aliased negative-frequency peak located $2\pi/\tau$ below the peak. By convention, the `fft` function gives reports the negative frequency components as aliased positive frequencies as diagramed in Fig. 13.5. In the time domain, a negative frequency and its positive alias produce signals that have the same value at each sampling time, as shown in Fig. 13.4.

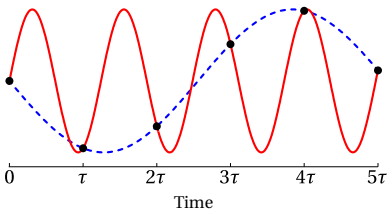


Figure 13.4 The sine of a negative frequency (dashed) and its aliased positive frequency (solid). In our wagon wheel example, these would be ω_{-1} and ω_0. The dots indicate times at which the signal is sampled. There are infinitely many other frequencies that also cross these sampled points, as specified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

Students are often bugged by the idea of a negative frequency. A negative frequency describes a wiggle with the same temporal regularity as its positive counterpart, but its phase evolves in a different manner.[1] For many problems you can safely ignore the negative part of the spectrum. But if you want to transform back to the time domain via the inverse Fourier transform

$$f(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(\omega) e^{-i\omega t} d\omega, \tag{13.6}$$

you will get the wrong answer if you don't properly account for the negative frequency components. Notice that the limits of integration in Eq. (13.6) extend to $\omega = -\infty$.

13.3 The Critical Frequency

Notice from Eq. (13.3) that the maximum frequency you can detect is controlled by the time step τ. If you want to see high frequencies you need a small time step

[1] A negative frequency component behaves just like a positive frequency for time-symmetric signals (i.e. $\cos(-\omega t) = \cos(\omega t)$), but has a π phase shift for antisymmetric signals (i.e. $\sin(-\omega t) = -\sin(\omega t)$).

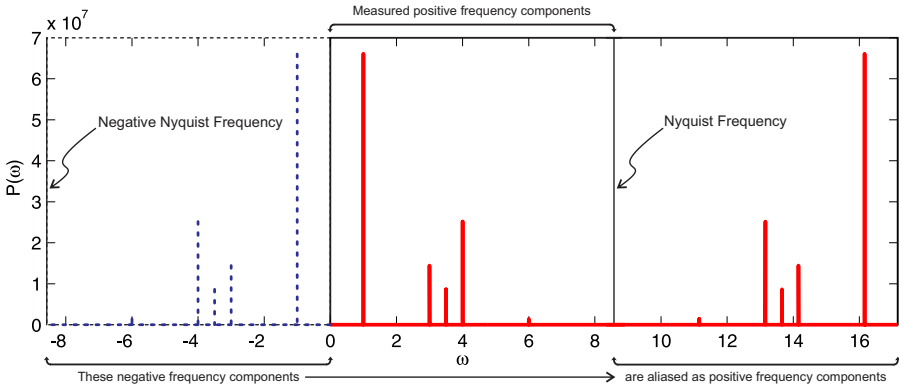


Figure 13.5 Plot of the power spectrum from Listing 13.1 showing the negative frequencies that are aliased as positive frequencies.

τ. Since the aliased negative frequencies will always be present in the right half of the `fft` window (for real signals), only the first half of the FFT frequency window has the peaks at the correct frequencies. Referring back to Eq. (13.3), this means we can only reliably detect frequency components with an absolute value less than

$$\nu_c = \frac{1}{2\tau} \qquad \text{or} \qquad \omega_c = \frac{\pi}{\tau} \tag{13.7}$$

This important limiting frequency is called the *critical frequency* or the *Nyquist frequency*. If the signal contains components higher than the critical frequency, those high frequencies intermingle with the aliased negative frequencies and you will be unable to distinguish between actual and aliased frequency peaks. For example, if we increase τ by using $\tau = 22{,}000/N$ while keeping $N = 2^{14}$ in our example code, the aliased peaks overlap the real peaks, as shown in Fig. 13.6. In this case it is impossible to tell which peaks are real and which are aliased throughout the whole range of ω.

Notice from the definition (13.2) that the frequency step size $\Delta\omega$ is controlled by $t_{\text{final}} = N\tau$. Thus, your minimum spectral resolution is inversely proportional to the amount of time that you record data. If you want to distinguish between two frequencies ω_1 and ω_2 in a spectrum, then you must take data long enough so that $\Delta\omega \ll |\omega_2 - \omega_1|$. Since the time step τ often needs to be tiny (so that ω_c is big enough), and the total data taking time t_{final} often needs to be long (so that $\Delta\omega$ is small enough) you usually need lots and lots of points. So you need to design your data-taking carefully to capture the features you want to see without requiring more data than your computer can hold.

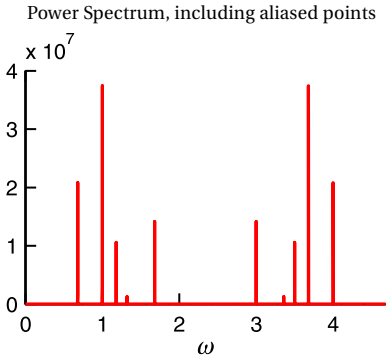


Figure 13.6 The `fft` of the same signal as Fig. 13.5 sampled with a slower rate (τ is about 4 times bigger) for the same amount of time. The "real" signal peaks are at $\omega = 1, 3, 3.5, 4, 6$; the rest of the peaks are aliased negative frequencies.

13.4 Windowing

The relative peak heights in Fig. 13.5 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 relative heights—power is proportional to amplitude squared,

so the peaks should be in the ratio [1,0.25,0.16,0.49,0.04]. To understand why the relative sizes are off, let's take the Fourier transform of one of the frequency components in our signal analytically:

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

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 correct. The solution to this amplitude problem relates to a concept called *windowing*.

When we numerically sample a waveform, there is always an implied "square window" around the data: the signal is zero before you start sampling, and immediately zero again after you stop. The square window artificially confines the signal in time. The *uncertainty principle* states that if we confine a signal to a short time, its frequency peaks will be broader. However, in our example we are broadening delta functions (infinitely narrow), and even the broadening due to the square window is not enough to resolve the peak.

The solution is to further narrow the time signal in a controlled fashion by multiplying it by a bell-shaped curve called a windowing function.[2] This narrowing in time further broadens the frequency peaks, and with broader peaks the height isn't as sensitive to where your data points fall in relation to the center of the peaks. However, if your windowing function is too narrow in time, your frequency peaks can overlap and your spectral resolution will be compromised (a phenomenon called *leakage*).

There are many types of windowing functions. See Matlab help on the `window` command for a list of ones that Matlab has built in.

13.5 The FFT vs. the Fourier Transform (Optional)

If you just want to know where the peaks in a spectrum occur, you can just take the `fft`, lop off the right half of the spectrum where the aliased negative frequencies are (after you have checked to make sure the aliased frequencies aren't spilling over), and plot the magnitude squared. However, there are many other uses for the `fft`. For instance, it is often useful to numerically calculate the Fourier transform of a signal, do work on the spectrum in the frequency domain, and then transform back into the time domain. When doing this type of calculation, you'll need to be careful to understand the relationship between Matlab's `fft` and what you think a Fourier transform should be.

[2] You may be wondering why we don't just use a shorter time window instead of multiplying by a windowing function. Well, you can, but then you reduce your resolution $\Delta\omega$ which is usually undesirable.

When a physicist refers to a Fourier transform, she usually means the operation defined by

$$g(\omega) = \mathscr{F}[f(t)] \equiv \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{i\omega t} dt \tag{13.9}$$

and when she refers to an inverse Fourier transform, she usually means

$$f(t) = \mathscr{F}^{-1}[g(\omega)] \equiv \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(\omega) e^{-i\omega t} d\omega, \tag{13.10}$$

However, these definitions of the Fourier transform are not universally used. For instance, you can put a factor of $1/2\pi$ on just one transform rather than a factor of $1/\sqrt{2\pi}$ on both.[3] 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, especially in engineering and acoustics. 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 formula Matlab calculates in the `fft` command is given by the sum

$$g(\omega_{k+1}) = \sum_{j=0}^{N-1} f(t_{j+1}) e^{-i2\pi jk/N}, \quad (k = 0, 1, 2, ..N-1) \tag{13.11}$$

and the `ifft` command computes the sum

$$f(t_{j+1}) = \frac{1}{N} \sum_{k=0}^{N-1} g(\omega_{k+1}) e^{i2\pi jk/N}, \quad (j = 0, 1, 2, ..N-1) \tag{13.12}$$

In these formulas, j enumerates the times t_j and k enumerates the frequencies ω_k. These expressions define the frequency spacing that we presented without justification in Eq. (13.2). To see how this works, use $j = t_j/\tau$ in the `fft` equation, so that 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 sums in Eqs. (13.11) and (13.12) are related to the integrals in Eqs. (13.9) and (13.10), but there are several differences that need to be addressed:

1. The `fft` defined in Eq. (13.11) is a sum with no normalization, so the height of your peaks scales with N, the number of points you sample. Thus, sampling the same signal with a different number of points will change the height of the result.

2. The `fft` has a negative exponent and (in our convention) the Fourier transform has a positive exponent. Thus, the `ifft` is closer to what we would call the Fourier transform than the `fft`.

[3] Physicists usually use the $1/\sqrt{2\pi}$ form because it makes an energy conservation theorem known as Parseval's theorem more transparent and symmetric.

3. The `fft` aliases negative frequency components to positive frequencies as discussed in the previous section.

To address the first issue, we just need to multiply by factors of N, $d\omega$ or dt, and $\sqrt{2\pi}$ at the appropriate places. The second issue can be addressed by using `ifft` to calculate the Fourier transform and `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. When we want to take the inverse Fourier transform, we put the negative frequencies back where the Matlab functions expect them to be.

When you put all of these modifications together, the function to calculate the "physicist's Fourier transform" in Eq. (13.9) becomes

Listing 13.2 (ft.m)

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

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

return
```

The `fftshift` function handles the requirement to put the negative frequencies back where they belong. The "physicist's inverse Fourier transform," i.e. Eq. (13.10) is then

Listing 13.3 (ift.m)

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

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

return
```

The nesting of functions in 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` takes in a time series and a dt (i.e. τ) and spits out a properly normalized (according to the $1/\sqrt{2\pi}$ convention) Fourier transform with the negative frequencies at the beginning of the array; `ift.m` takes in 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 matrices, the frequency matrix 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)
```

or this if you absolutely must use an odd `N` (puts $\omega = 0$ in the right place)

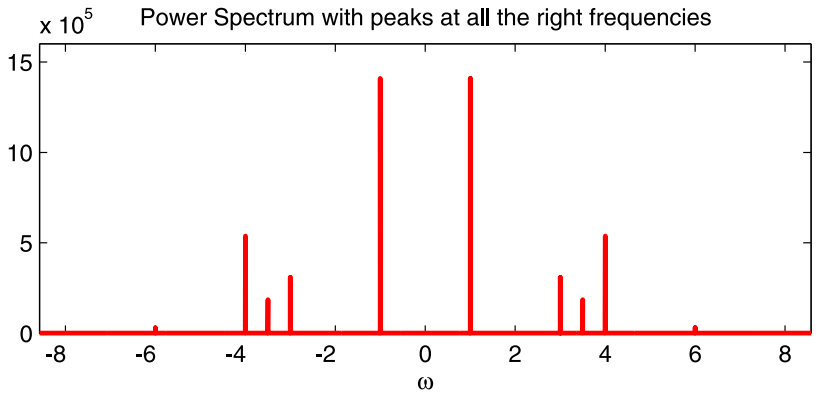


Figure 13.7 Plot of the example power spectrum

```
w=-((N-1)/2)*dw:dw:dw*((N-1)/2);
```

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

Listing 13.4 (ch13ex4.m)

```
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')
```

Run the example and note that we now have frequency peaks at all the right frequencies. Also vary N and note that the peak heights don't scale with N any more. The amplitude of peaks in a Fourier transform 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.

Chapter 14

Solving Nonlinear Equations

You will sometimes need to solve difficult equations of the form $f(x) = 0$ where $f(x)$ is a complicated, nonlinear expression. Other times you will need to solve complicated systems of the form $f(x,y,z) = 0$, $g(x,y,z) = 0$, and $h(x,y,z) = 0$, where f, g, and h are all complicated functions of their arguments. Matlab can help you solve these systems.

14.1 Solving Transcendental Equations

Matlab's `fzero` command solves equations like $f(x) = 0$ automatically. Before we see how to use `fzero`, let's learn the basics of how to find zeros by studying the secant method.[1]

The Secant Method

The first step in the secant algorithm is to make two guesses x_1 and x_2 that are near a solution of this equation. You can find reasonable guesses 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. This line is shown in Fig. 14.1 and its equation is

$$y - f_2 = m(x - x_2) \tag{14.1}$$

We can solve Eq. (14.1) for the value of x that makes $y = 0$. If we call this new value of x x_3, we have

$$x_3 = x_2 - \frac{f_2}{m} = x_2 - \frac{f_2(x_2 - x_1)}{f_2 - f_1} \tag{14.2}$$

as shown in Fig. 14.1. The value x_3 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 $\exp(-x) - x = 0$ using this method

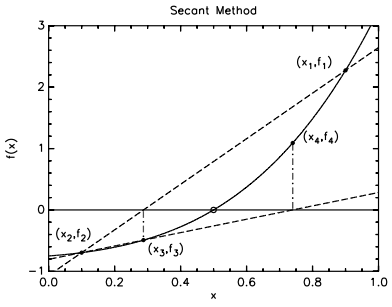


Figure 14.1 The sequence of approximate points in the secant method.

[1] The secant method 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.

Listing 14.1 (ch14ex1.m)

```
% Secant method to solve the equation exp(-x)-x = 0
clear; close all;

% Define the function as an anonymous function
func = @(x) exp(-x)-x;

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

% From the plot it looks like the solution is near x=0.6,
% so 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;

while chk>1e-8  % start the loop

   f2=func(x2); % find f(x2)

% find the new x from the straight line approximation
   xnew = x2 - f2*(x2-x1)/(f2-f1)

% find the error by seeing how closely f(x)=0 is approximated
   chk=abs(f2);

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

end % end of while loop
```

Matlab's Fzero

Matlab has its own zero-finder which internally does something similar to the secant method described above. To use it to solve the equation $f(x) = 0$ you must make an m-file function (called `fz.m` here) that evaluates the function $f(x)$. Here is an example function file for $f(x) = \exp(-x) - x = 0$:

Listing 14.2 (fz.m)

```
function f=fz(x)

% evaluate the function fz(x) whose roots are being sought
f=exp(-x)-x;
```

Once you have `fz.m` built, you call `fzero` and give it a reference to `fz.m` and an initial guess, and it does the solve for you. Here is the command to find a zero of $f(x) = 0$ near the guess $x = 0.7$

```
x=fzero(@fz,0.7)
```

Note that the function `fz.m` must be stored in the current directory. The @-sign syntax tells Matlab that you are passing in a reference to a function.

14.2 Systems of Nonlinear Equations

The `fzero` command will only work with a single equation. If you have a complicated system of nonlinear equations, you can use Matlab's `fsolve` to solve the system in a similar way.

Consider the following pretty-impossible-looking set of three equations in three unknowns (x, y, z).

$$\begin{aligned} \sin(xy) &= 0.95908 - \exp(-xz) \\ z\sqrt{x^2+y^2} &= 6.70820 \\ \tan(y/x) + \cos z &= 3.17503 \end{aligned} \tag{14.3}$$

The way to talk `fsolve` into solving this set is to first write the system with zeros on the right side of each equation, like this

$$\begin{aligned} \sin(xy) + \exp(-xz) - 0.95908 &= 0 \\ z\sqrt{x^2+y^2} - 6.70820 &= 0 \\ \tan(y/x) + \cos z - 3.17503 &= 0 \end{aligned} \tag{14.4}$$

Then we write a function file that accepts x, y, and z as arguments and returns the left sides of the equations. For the equations above, this would look like this:

Listing 14.3 (eqsystem.m)

```
% nonlinear system of equations routine for use with fsolve
function s=eqsystem(xn)

% Unpack the inputs into friendly names
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; Eq2; Eq3 ];
```

Here is a piece of Matlab code that uses `fsolve` to solve this system with the initial guess (1,2,2). `fsolve` uses a minimizer routine like `fminsearch` did when we were fitting data in chapter 12. Basically, it tries a bunch of input values to the function and searches for the inputs that make your `eqsystem.m` function return all zeros. If your initial guess is way off, it can get stuck in a local minimum, and if your system has multiple solutions, it will only find one.

Listing 14.4 (ch14ex4.m)

```
% Uses fsolve to look for solutions to the nonlinear system
% of equations defined in the file eqsystem.m
clear; close all;

x0 = [1; 2; 2];           % Make a starting guess at the solution
options=optimset('Display','iter');   % Option to display output
[x,fval] = fsolve(@eqsystem,x0,options)  % Call solver

fprintf(' The solution is x=%g, y=%g, z=%g\n',x(1),x(2),x(3));
fprintf(' Final values of the function file = %g \n',fval)
disp(' (Make sure they are close to zero)')
```

You could do the same thing a little more compactly using anonymous functions, but you lose some readability. Here is an example using this approach:

Listing 14.5 (ch14ex5.m)

```
clear;close all;
eqsystem = @(x) [sin(x(1)*x(2))+exp(-x(1)*x(3))-0.95908; ...
                 x(3)*sqrt(x(1)^2+x(2)^2)-6.70820; ...
                 tan(x(2)/x(1))+cos(x(3))+3.17503];

x0 = [1; 2; 2];
options=optimset('Display','iter');
[x,fval] = fsolve(eqsystem,x0,options)

fprintf(' The solution is x=%g, y=%g, z=%g\n',x(1),x(2),x(3));
fprintf(' Final values of the function file = %g \n',fval)
disp(' (Make sure they are close to zero)')
```

Chapter 15

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). In contrast, *vector* image formats store pictures as mathematical formulas describing the lines and curves, and let the renderer (e.g. the printer or the computer screen) draw the picture the best it can. Fonts are stored in a vector format so that they can be drawn well both on the screen and on paper. The most common format used to store vector graphics in physics is encapsulated postscript (EPS).

Raster graphics are well-suited for on-screen viewing. However, they are often not a good choice for figures destined for the printer (especially line plots and diagrams). They often 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, this approach makes for very large file sizes.

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, and then later we will go over some tips for making nice raster graphics for a presentation.

15.1 Creating an EPS File

Matlab can create 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.

Listing 15.1 (ch15ex1.m)

```
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--');
```

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.

While this is a pretty simple process, the Matlab defaults have some issues. We have included the EPS file generated this way as Fig. 15.1, shown at Matlab’s default size. This EPS has a few problems. The axes Matlab chose on which

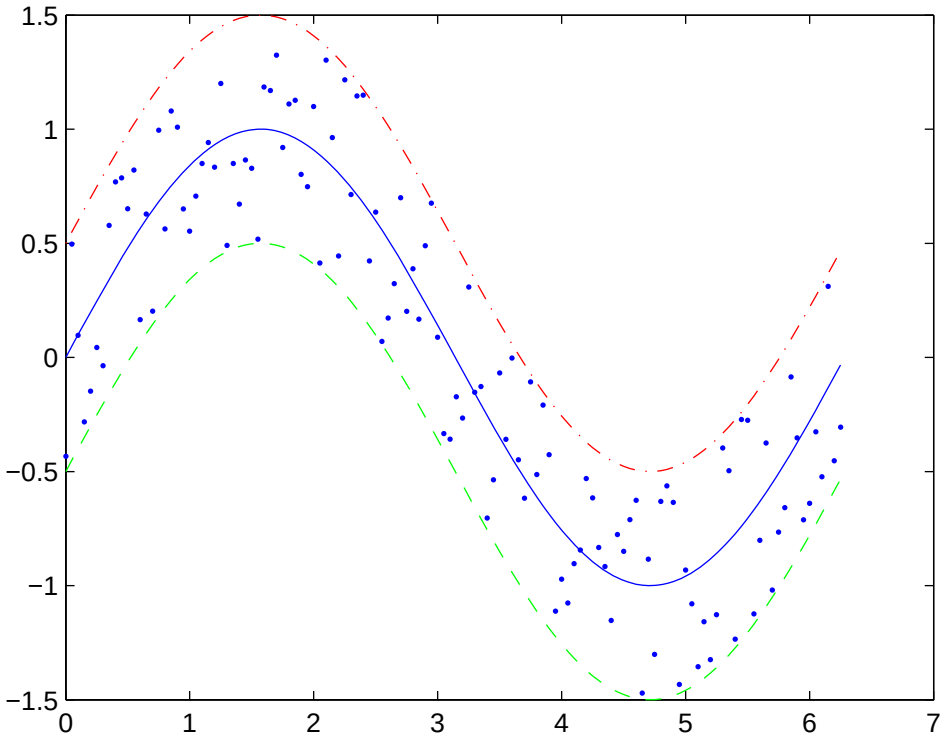


Figure 15.1 Default plot output from Matlab

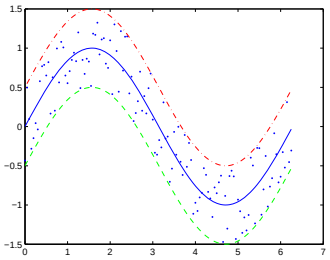


Figure 15.2 Figure 15.1 scaled to a size similar to journal requirements.

to plot the function aren’t the ones we would pick, but we can fix that with the `axis` command (see section 5.1). The large size that Matlab chose for the figure is also an issue. A common practice in senior thesis is to just scale Fig. 15.1 to be smaller, as in Fig. 15.2. But this figure has unreadably small text and almost invisible lines. It would 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 legible size and that the lines be visible. Journals

will not fix your figures for you.

While you may not immediately publish in journals, you will almost certainly include plots in your senior thesis. You will want to create something that looks nice when scaled to a reasonable size. 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.

15.2 Controlling the Appearance of Figures

The Matlab GUI interface

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 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 exporting an EPS file. The .fig file stores all of your adjustments, so you can come back later and modify something and then 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 your format commands so that all your figures have the same size and style. In the long run, you will save yourself time by learning to control figure properties from the m-file.

To help you 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 not get you what you want. The commands have to be put in the right place and refer to the right objects. The next section will teach you the basics of how these commands work.

Formatting figures with script commands

Matlab treats a figure as a collection of visual objects. Each object has an internal label called a *handle* to allow you 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');
```

creates the x-label and also stores a handle for the label object in the variable `tt`.

Once you have a handle to an object, you can specify the visual properties using the `set` command, like this

```
set(tt,'PropertyName1','PropertyValue1',...)
```

This command tells Matlab to take the object with handle `tt` and set its Property-Name 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 code to add an x-axis label and change its font size to 8 point.

One of the most frequently referenced objects on a plot 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 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 does it for you when you make a plot. 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`. Then, to set the font to 12 point Symbol for that axes object, you would use

```
set(aa,'FontSize',12,'FontName','Symbol');
```

Note that you need to use `gca` to store the current handle in a variable *before* you open another set of axes by using another `figure` or `plot` command, otherwise the axes you want to refer to will no longer be the current axes. 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 example code 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 have to choose the right index of the handle array. To get the hang of this, modify the example code 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 already learned many of these. 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

```
pp=plot(x,f,'b',x,data,'r*',x,err_hi,'r-.',x,err_low,'g--','LineWidth',2);
```

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.

Controlling the Size of Exported Graphics

Controlling the size of the exported figure is tricky. The basic parameters are 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. Probably the best way to learn how to do this is to study an example. Execute the following code see what the plot looks like.

Listing 15.2 (ch15ex2.m)

```
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 with the correct size
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;

% Set the outer dimensions of the axes to be the same as the
% figure.  The 'OuterPosition' property describes the
% boundary of the whole figure.
set(aa,'Units',Units,'OuterPosition',[0 0 figWidth figHeight])

% Calculate where the axes box should be placed inside the
% 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 rectangle
% around the plotted data
set(aa, 'Position', newPos);
```

The EPS produced using “Save As" is included as Fig. 15.3 in this document so you can see what was affected by these commands (compare with Fig. 15.1 which shows the output without the sizing commands).

Making an EPS Suitable for Publication

The sizing commands 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

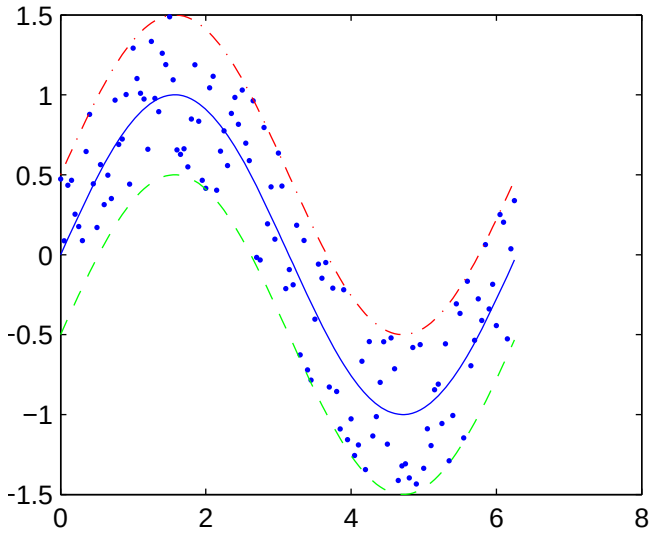


Figure 15.3 Plot made in Example 17.3a (no scaling).

to provide a legend that tells what the lines and dots on this plot mean. In the following example code 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.

Listing 15.3 (ch15ex3.m)

```
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
set(pp(3),'LineWidth',1); % Make the error bound lines heavier
set(pp(4),'LineWidth',1); % Make the error bound lines 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;

% Make the ticks a little longer, put the symbol for pi in the
% number labels using the symbol font (LaTeX won't work there),
% and set the minor ticks to display.
set(aa,'LineWidth',1,...
       'TickLength',get(aa,'TickLength')*2,...
       'FontSize',8,...        % Set the font for the axes
       'FontName','Symbol',... % to get the pi symbol in labels
       'XTick',[0  pi/2 pi 3*pi/2 2*pi],...
       'XTickLabel',{'0';'p/2';'p';'3p/2';'2p'},...
       '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 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
newPos = get(aa, 'OuterPosition') - ...
   get(aa,'TightInset')*[-1 0 1 0; 0 -1 0 1; 0 0 1 0; 0 0 0 1];

% Set the position of the axes box within the figure
set(aa, 'Position', newPos);
```

The EPS output (made using “Save as") produced by this example is included as Fig. 15.4. 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.

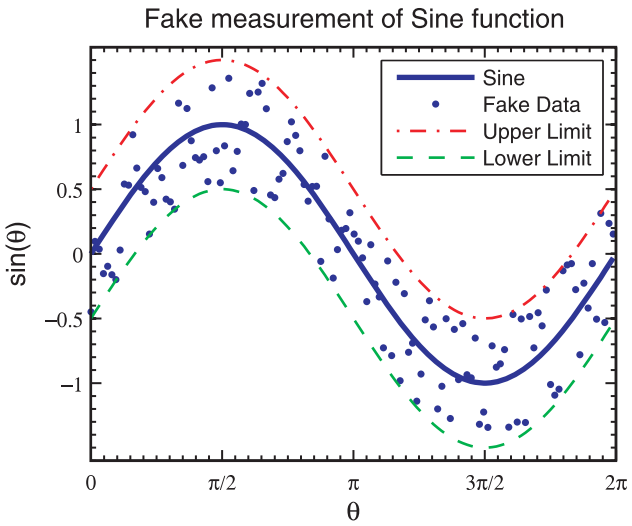


Figure 15.4 Plot made in Example 17.2b (no scaling).

Subplots

There are a few tricks to controlling the size and appearance of the `subplot` figures bound 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.

Listing 15.4 (ch15ex4.m)

```
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
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);

% Create the second set of axes in this figure
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.  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);
```

Figure 15.5 shows the plot produced by this script.

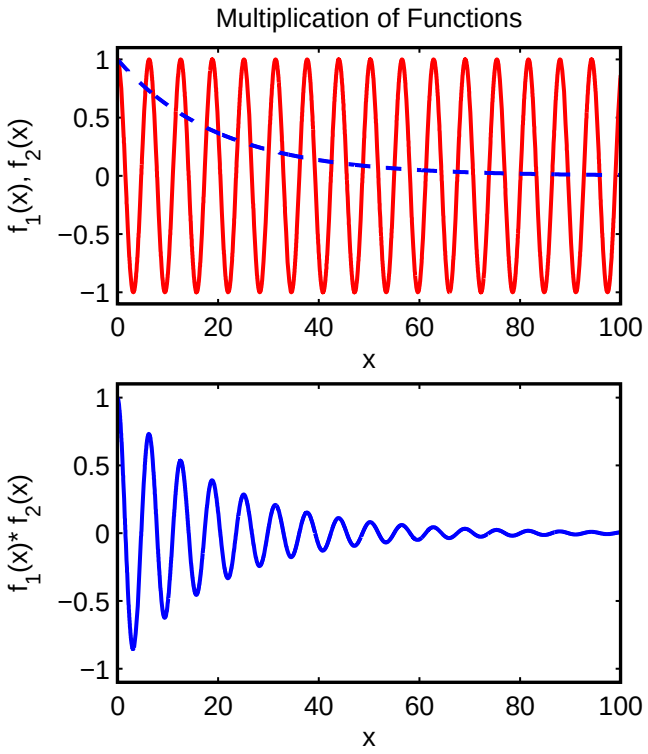


Figure 15.5 An example of a set of plots produced using `subplot`.

15.3 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

```
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 when destined 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.

[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.

Index