```
str1='Matlab for the behavioral
          sciences.';

str2='by Ione Fine and Geoffrey
          Boynton';

       disp([str1 str2])
```

Dedications

For Robert M. Boynton who bought Geoff an Apple II in 1979.

And also for Anne Fine.

Neither of whom will ever read this book.

PRAISE FOR PREVIOUS WORK BY THE AUTHORS

"We do not doubt the technical quality of your study or its interest to others working in this and related areas of research. However, after consideration, we are not persuaded that your findings represent a sufficiently outstanding scientific advance to justify publication in Nature." Editor, *Nature, 2006*

"While the work is adequate in execution it lacks originality" *Reviewer 1, NIH CVP Study Section, Autumn* 2007

"Unfortunately there are conceptual flaws in the design of the experiments which make this work unsuitable for publication" *Reviewer 3, Journal of Vision, 2009*

"While interesting, this work is incremental in scope, and as such is not suitable for a high profile journal" *Reviewer 1, Nature Neuroscience, 2011*

"Your qualities overshadow your weaknesses." *Chinese restaurant fortune cookie. 2013*

CONTENTS

Contents

CHAPTER 1: WHAT IS PROGRAMMING AND WHY LEARN IT?

1.1 How to use this book

Be careful how you skip through this book. Unless you are fluent in another programming language, please don't skip. A lot of important stuff happens even in Chapter 2. We've noticed that people who skip early chapters and exercises tend to get horribly stuck later because they are missing crucial skills.

Find a buddy
It's great if you can persuade a buddy to learn with you (not your boyfriend if you want the relationship to survive). But you don't want to be sitting watching while the 'better' buddy writes code, see the next tip on gurus.

Find a guru
Don't hesitate to ask a better programmer to be your guru – like good deeds, one of the hidden expectations of the Matlab brotherhood is that you must help pass on the knowledge.
It's even ok to get your guru to write the code for an exercise you are struggling with and explain it to you. Spend as long as you want looking at your guru's code. But then, close the file containing their code, take a 2 minute break away from the computer, and try to re-write the program *without looking at their code*. If you can do that successfully you have encoded what your guru has taught you, and you aren't relying on short term memory.

For bugs, ask your guru to give you a hint rather than just telling you what the bug is.

Don't go programming blind.
If you've been staring at the same bug for more than an hour, take a 15 minute break. Preferably, go outside and take a walk. If you've been staring at the same bug for more than four hours then give up for the day. You are likely to find that the bug is easier to solve in the morning.

Don't become a crazy-Frito-eating-insomniac
It's easy to code till 4 am and then lie half-awake till 6 am dreaming of 'for loops'. Set a time for when you are going to stop programming. Do something else for at least ½ an hour before you go to bed.

Remember the 100:10:1 rule
A very small simple program might take Geoff and I about 1 hour to write because we have more than 10 years of programming experience. The same program took us about 10 hours to write when we had 1 year of experience. It took 100 hours to write when we had 1 week of experience. You *will* get faster.

1.2 What is programming?

Programming is telling a computer what to do. There are only 2 tricky things about this:

Computers are very stupid – you have to tell them EXACTLY what to do.

Computers don't speak English.

Any programming language is a compromise between the computer's native language (0010001) and your native language (English). High level programming languages are closer to English, low level languages are closer to computer-ese. The closer a language is to English, the easier and faster it tends to be to program, but the slower it is for the computer to interpret it and the more constrained it is on what it can

do. Low level languages tend to be hard to program but very fast to run, and less constrained. Matlab is a mid- to high level language.
The metaphor with language goes further. Like real languages some things are easier to say in one language rather than another (Italian is the language of love etc. etc.). Some programming languages are better for computations, others for graphics. Like any language, programming languages have grammar. Unlike people who speak real languages (with the exception of the French) computers are very fussy about grammatical errors. Like learning a real language, at first it will be very boring and very hard to say the simplest thing, but it will get easier, fast. Finally, like learning languages, with even basic fluency you can do a lot of things you couldn't do otherwise.

1.3 Why program?

The following is a summary of the conversations I've had with students who have been wondering whether or not to learn to program.

"My field has lots of specialized prewritten software packages. Why can't I just use those?"
You can, but your experiments/analyses will be limited to what these packages can do. So a lot of really creative things aren't possible. What's more, we've noticed in some senior colleagues that spending many years thinking "inside the box" - only designing programs that are within the technical capacities of their software packages - can also (though not always) limit *theoretical* creativity."

"But surely these specialized software programs are way better than anything I can write?"

"Yes and no. These software packages are written by experienced or professional programmers and are used by lots of people. So they tend to be pretty reliable, and have fewer bugs. But they are definitely not bug free – no program is. If you can program you will have better insight into where the bugs are and will be able pinpoint issues with technical

support personnel much more quickly. With open source software – which is becoming increasingly common - you will be able to fix the bugs yourself.

"My lab has a professional programmer, and I plan to do the same thing when I run my own lab."

Hiring a programmer has some big advantages: you don't need to spend a lot of time programming yourself or teaching your students to program. However, good programmers are expensive and they can be hard to find. Computer scientist students may cheap and available, but they are not always good. And unless you yourself can program it's hard to evaluate how good a job they are doing. Even if you hire a programmer, being able to program yourself means you will be better able to describe what you want in a program and what kind of flexibilities you will want in terms of future experiments. Even more importantly, you will know how to bug-check the program carefully once it is written."

"I've tried to learn before, and programming is too hard for me."

"Every year some students in my class **love** the class, find the exercises ridiculously easy and are writing programs that predict the next winner of American Idol by week 7. (Before you make snide comments, yes some of them are uber-geeks, but others are relatively cool by graduate student standards.) Other, equally bright, students hate the class, feel totally inferior, and end up in my office in tears. **But … they all learn to program competently in the end if they put their time in**. Like real languages, some people have an 'ear for language' and others don't. I (Ione) have no ear for language whatsoever and I still speak reasonable French after 1000 hours of patient high school instruction. I'm not a natural programmer either, and yet I'm one of the people writing this book. If you find this book really difficult, relax, it doesn't mean you are stupid. Accept that you are not a natural programmer and it's going to take you a while. Or possibly we've written a bad book.

1.4 Hardware

Hardware is the physical presence of the computer. The monitor, the

hard drive, the CPU (central processing unit, i.e. the “brain”). Hardware is anything you can destroy by poking it with a screwdriver.

1.5 Software

Software = programs. Programs are instructions to your computer to behave in a particular way. So a software program like Microsoft Office gets all machines to behave in a particular way – the instructions are slightly different for different computers (different hardware), but the program makes all computers behave (almost) the same.

All software is written in a programming language. Some programs, (like Matlab) are there to help you write new programs. Matlab will run on Macs, PC, and UNIX. You may find that a few of the commands only run on some computers.

NOTE – these days it is almost impossible to damage your computer by running a program that you wrote in Matlab. There are built in safeguards which make it very hard for you to do anything that will damage the computer. In this book you won’t be using any commands that can do any permanent damage. So crash your computer hard. Have fun!

1.6 Getting started

You need:

A PC running Windows or a Mac computer running OSX.

You need to install Matlab on your computer.

Start Matlab. A window will appear that’s divided into a number of sub-windows. Close all the sub-windows (Later you may find these other windows helpful, but we will ignore them for now) except the one that called the "command window” which has a little prompt:

>>

This is where you should type your commands. In this book the text in green is what you should type into the command window. The text in blue shows what the command window should spit back out at you.

```
str1='I have no clue what I am doing'
```

```
str1 =
I have no clue what I am doing
```
Congratulations! You have just done your first bit of programming.

CHAPTER 2: STRINGS AND VECTORS

2.1 variables, who, whos & disp

```
str1 = 'this is all new to me'
str1 =
this is all new to me

str2='I have no clue what I am doing'
str2 =
I have no clue what I am doing
```

You just told Matlab to create two *strings* of letters and to name those *strings* **`str1`** and **`str2`**.

The single quotes tell Matlab that **`str1`** is a *string* of letters (not numbers, more on that later). A string that only contains a single letter is generally called a *character* or *char*.

`str1` and `str2` are both *variables*. If you want to see what's in the variables `str1` and `str2` you simply type their names.

```
str1
str1 =
this is all new to me


str2
str2 =
I have no clue what I am doing
```

The **`who`** command asks your computer to give you a list of all the variables you have in memory. At the moment the only variables you

have are **str1** and **str2**

```
who
Your variables are:
SPM5Path str1  str2
```

The command **whos** gives you a little more information about those variables.

```
whos
 Name   Size  Bytes Class  Attributes

 SPM5Path 1x34  68   char
 str1   1x21  42 char
 str2   1x30  60 char
```

As well as the name of each variable, **whos** tells you how long a string or vector it is (second column, more on vectors later), how much computer memory (`Bytes`) the list uses (third column, more on that later too) and the `class` of the variable (whether it is a list of letters (`char`) or numbers (`double`) or something else). We'll get into more technical detail about exactly what a double is later on. For now, just think of it as a number.

Typing the name of a variable without a semicolon asks your computer to tell you what is contained within that variable.

```
str1
str1 =
this is all new to me

str3='is it all going to be this boring?'
str3 =
```

```
is it all going to be this boring?
```

Compare the commands above, with the one below, where we add a semi-colon at the end.

```
str4='because this is very boring indeed.';
```

The semi-colon tells Matlab that you don't want it to display the output of each command. The technical expression is that the semi-colon "suppresses" output to the command window.

Using `disp` also displays what a variable is, but in a slightly more concise form, where the command window only shows the contents of the variable instead of also repeating the name of the variable

```
disp(str1)
this is all new to me
```

2.2 Creating an m-file

Typing everything in the command window won't work in the long term. Instead, you can open a file that contains a list of commands to run – a program! Matlab's files are called ‘m-files'.

Make sure the command window is at the front. Now go to the menu bar and choose **File->New->Script**.

You’ll see a blank document in a new editor window.

Every program should begin with a few lines of documentation. This is called a *header*. Good headers contain the following information:

The name of the program

A description of what it does

Who wrote it, and when

Any changes about the program you make after initially writing the program, and the date of those changes.

```
% BreakfastCereal.m
%
% this program provides examples of fun
things you can do
% with cereal boxes
% inspired by Geoff's misspent childhood
% at the breakfast table
%
% written by IF 3/2007
% minor modifications 9/2011
```

Make sure every new line begins with a %. The % tells Matlab to ignore that line – these comments aren't commands for Matlab, they're information about the program for you. Commented text will show up as being green.

If a line is longer than the width of the m-file window you can break it using three dots … These three dots tell Matlab to treat that line as a continuous line of code. In the Kindle our break lines may not appear in quite the right place.

Headers are important. You may think that you will remember the programs you write – but trust me, you won't! Getting in the habit of having good up-to-date headers is like cleaning your teeth – it's boring but it will save you a lot of pain in the long run.

Now create a folder somewhere on your computer called **LearningMatlab** (or something like that). Don't put it inside the Matlab program folder since anything inside that folder is liable to be deleted if you do a major upgrade of Matlab.

Don't let the name of this folder have any '/' symbols or spaces or funny characters in it since Matlab doesn't like that (the same is true of the titles of m-files- **Breakfast Cereal.m** is a name Matlab won't like at all.).

Save the new m-file as **BreakfastCereal.m** inside your **LearningMatlab** folder.

2.3 Setting the path

Now type:

```
help BreakfastCereal.m
```

```
BreakfastCereal.m not found.
Use the Help browser search field to search
the documentation, or
type "help help" for help command options,
such as help for methods.
```

When Matlab says that a file is "not found" that means Matlab can't find an m-file that has that particular name. Sometimes the reason you can't find a file is that it doesn't exist (e.g. you've got a typo in the name.) The other reason Matlab can't find your new file is because Matlab is only allowed to look for files in certain places.

One place that Matlab always looks for files is the *current directory* or *working directory*. In fact this is the first place that Matlab looks. When you start Matlab it automatically chooses a particular folder to be the working directory. The default setting is made by Matlab and is the bin directory - the Program Files/Matlab/R2011A/bin directory on my computer. If you don't tell Matlab otherwise it will save files to that default folder. You can see what folder Matlab thinks is the current directory using the print working directory command - **pwd**:

```
pwd
ans =
C:\Program Files (x86)\MATLAB\R2011a\bin
```

The other places that Matlab can find files are those folders that are in Matlab's search *path*. This path is a simply a list of folders that Matlab is allowed to look in whenever it is trying to find a file. Again Matlab comes with a default set of paths. You can get a list of the current folders in the path very easily:

```
path
MATLABPATH
C:\Documents and toolbox\emlcoder\emlcoder
C:\Program
Files\MATLAB\R2007b\toolbox\emlcoder\emlcode
rmex
C:\Program
Files\MATLAB\R2007b\toolbox\exlink
C:\Program
Files\MATLAB\R2007b\toolbox\filterdesign\fil
terdesign
C:\Program
Files\MATLAB\R2007b\toolbox\filterdesign\qua
ntization
C:\Program
Files\MATLAB\R2007b\toolbox\filterdesign\fil
tdesdemos
C:\Program
Files\MATLAB\R2007b\toolbox\finance\finance
```

We need to add your new folder **LearningMatlab** to the path. Make sure the command window is at the front, and go to: **File->Set Path** in the menu bar. A pop-up window will appear Choose **Add With Subfolders**, choose your **LearningMatlab** folder, and click OK. Then

choose **Save and Close**.
Matlab should be able to find `BreakfastCereal.m` now:

```
help BreakfastCereal
BreakfastCereal.m

This program provides examples of
fun things you can do with cereal boxes
inspired by Geoff's misspent childhood
at the breakfast table

written by IF 3/2007
minor modifications 9/2011
```

2.4 Redefining variables, indexing and subscripting

Now we are going to start typing things into your m-file instead of the command window. Because this code is going in an m-file we're using the same formatting as Matlab will use when you type it into an m-file (remember that green text still means you should write things in the command window).

```
frosted='FROSTED FLAKES';
cheerios='CHEERIOS';
```

Now imagine we are trying to find out what the ninth letter in `frosted` was. We would do this as follows:

```
frosted(9)
ans =
F
```

The value 9 is called the *subscript* or the *index* into `frosted`. With a 1-D string or matrix there is no difference between an index and a

subscript so we'll discuss the difference between these two terms a little later. You can see from this example that Matlab is counting spaces.

Now let's create a new variable called **frosted_scrambled** and tell Matlab to make **frosted_scrambled** the same as **frosted.** In the m-file:

```
frosted_scrambled=frosted;
```

And in the command window:

```
frosted
frosted =
FROSTED FLAKES

frosted_scrambled
frosted_scrambled =
FROSTED FLAKES
```

The variables **frosted** and **frosted_scrambled** are exactly the same. Now let's replace the 9th letter ('F') in the variable **frosted_scrambled** with an **'E'** (like you might do with a black marker on a cereal box.)

```
frosted_scrambled(9)='E'
frosted_scrambled =
FROSTED ELAKES
```

Now let's tell Matlab to make the first letter in **frosted_scrambled** the same as the sixth letter in **frosted.**

```
frosted_scrambled(1) = frosted(6)
frosted_scrambled =
EROSTED ELAKES
```

What is the difference between an array and a vector?

Technically, in Matlab there are only vectors so the terms tend to be used interchangeably. In some other languages there is a difference: the size of a vector can change but the size of an array is fixed when you first define it.

This would be a good time to go do Exercise 2.1 at the end of this chapter.

2.5 Vectors

You can also create lists of numbers. These are called *vectors* (or sometimes *arrays*). Here's four different ways of creating a vector list that goes from two to nine in steps of one.

```
vect1=[2 3 4 5 6 7 8 9]
vect1 =
  2  3  4  5  6  7  8  9
```

In this first example the square brackets tell Matlab that you are creating a single vector (list of numbers).

```
vect2=linspace(2, 9, 8)
vect2 =
  2  3  4  5  6  7  8  9
```

What is an Argument?

An argument is the term that is used to describe the value or values that are supplied to a procedure such as `linspace` (procedures are also known as commands or functions, more on that later).

In this second example, you are using a command **`linspace`** which has three *arguments*. The first argument (the number 2) gives you the first value in the vector, the second argument (the number 9) gives you the last value in the vector, and the third argument (the number 8) specifies that you want the list of 8 numbers that are evenly spaced between 2 and 9. You can imagine that this command would be useful if you had collected eight pieces of data at time points evenly spaced between two and nine seconds.

```
vect3=2:1:9
vect3 =
  2  3  4  5  6  7  8  9
```

In this third example you are asking for a list of numbers that goes from 2 to 9 with a step-size between each number of 1. You can imagine that this command would be useful if you collected a piece of data every second, starting after two seconds and continued collecting that data until nine seconds had gone by.

```
vect4=2:9
vect4 =
  2  3  4  5  6  7  8  9
```

In this fourth example we are using the fact that Matlab assumes a default step-size of 1, so we are simply skipping that part of defining the vector.

Here are three ways of creating a list of numbers that goes from 1 to 17 in steps of two.

```
vect5=[1 3 5 7 9 11 13 15 17]
vect5 =
  1  3  5  7  9 11 13 15 17

vect6=linspace(1, 17, 9)
vect6 =
  1  3  5  7  9 11 13 15 17

vect7=1:2:17
vect7 =
  1  3  5  7  9 11 13 15 17
```

Here are some other examples of vectors.

```
vect8=3:1.5:9
vect8 =
 3.0000 4.5000 6.0000 7.5000 9.0000

vect9=9:-1:-3
vect9 =
  9  8  7  6  5  4  3  2  1  0 -1 -2 -3

vect10=linspace(2, 8, 12)
vect10 =
Columns 1 through 9
  2.0000  2.5455  3.0909  3.6364  4.1818
4.7273  5.2727  5.8182  6.3636
 Columns 10 through 12
```

```
  6.9091  7.4545  8.0000
```

```
vect11=linspace(8, 2, 12)
vect11 =
Columns 1 through 9
  8.0000  7.4545  6.9091  6.3636  5.8182
5.2727  4.7273  4.1818  3.6364
 Columns 10 through 12
  3.0909  2.5455  2.0000
```

2.6 Back to indexing

You can index in to vectors as well as strings. Here we are going to index the second integer in vect3

```
vect3(2)
ans =
  3
```

You can also index more than one number in a vector or string. Or even use a vector as an index into a string. This is an important skill, so be sure you understand the examples below and do the exercises at the end of the chapter.

```
vect4(2:1:4)
ans =
  3  4  5
```

```
frosted(3:8)
ans =
OSTED
```

```
frosted(vect4)
```

```
ans =
ROSTED F

x=3:12
x =
  3  4  5  6  7  8  9 10 11 12

y=x(11)
??? Index exceeds matrix dimensions.
```

Index exceeds matrix dimensions?

This is an error message that you will see on a regular basis. It means you are trying to index a value in x that doesn't exist. In this case it doesn't exist because x only has ten numbers in it and you are asking for the eleventh number in x.

```
 x=-3:10
x =
-3 -2 -1  0  1  2  3  4  5  6  7  8  9 10

y=1:2:8
y =
  1  3  5  7
x(y)
ans =
 -3 -1  1  3

y(x)
??? Subscript indices must either be real
positive integers or logicals.
```

Subscript indices must either be real positive integers or logicals?

Why does `x(y)` work while `y(x)` doesn't?

This is an error message that you will see on a regular basis. It means you are trying to index a value in `x` that doesn't exist.

Because `x` contains negative values you can't use it as an index. It doesn't make sense to ask for the -3rd, -2nd or the 0th value of `y`. In the second example you are getting this error because you are asking for the 3.5th value of `x`.

```
x(3.5)
??? Subscript indices must either be real
positive integers or logicals.

ff=[6 13]
ff =
  6 13

frosted_scrambled(ff)=['F' 'F']
frosted_scrambled =
EROSTFD ELAKFS
```

2.7 clear

Finally we're going to teach you how to clear all your variables out of memory and start with a fresh slate. Here we are going to use **`who`** to see what's in your workspace, and then we will get rid of the single variable **`cheerios`**.

```
who
Your variables are:
ans    resp   vect11  vect4   vect7   x
x3
cheerios vect1   vect2   vect5   vect8   x1
y
frosted  vect10  vect3   vect6   vect9   x2

clear cheerios
who
Your variables are:
ans   resp   vect10  vect2  vect4  vect6
vect8  x    x2    y
frosted vect1  vect11  vect3  vect5  vect7
vect9  x1    x3
```

cheerios should be gone. Another way of seeing whether a variable still exists is simply to type its name into the command window.

```
cheerios
??? Undefined function or variable
'cheerios'.
```

Undefined function or variable 'cheerios'?

This error means that Matlab is looking for a variable or a function that it can't find. Often this is because of one of the following three issues:

(1) `cheerios` is a variable that hasn't been defined yet. Sometimes finding this bug is harder than you might expect because `cheerios` was defined at some point when you were writing the code, and then sat in memory after that little chunk of code disappeared. So the code

ran ok for a while, until you cleared the variables in memory, and then whoops! Suddenly there's a problem.

(2) A similar thing can happen when there is a typo in your code. For example you might define a variable `cheerios` and then refer to it as `cheereos` later in the code.

(3) You are trying to do a calculation on `cheerios` but you've been moving code around, so `cheerios` is actually defined later in the code then the place you are trying to use it.

(4) You are actually trying to refer to a function `cheerios` and you have a typo. For example, you might refer to the function as `cheerios` when you are calling it, but the actual name of the function is `cheerio`.

(5) You are referring to a function, but that function is no longer in the path for some reason. So as far as Matlab is concerned that function doesn't exist. The solution is to fix your path.

This error means that x no longer exists. In this case the error message is a good thing since it showed that **`clear`** worked, but read the side bar carefully, because this can be a tricky error to deal with.

To clear everything you use **`clear all`**

```
who
Your variables are:
ans   resp   vect10  vect2  vect4  vect6
vect8  x    x2    y
frosted vect1  vect11  vect3  vect5  vect7
vect9  x1    x3
```

```
clear all
who
```

Questions for Chapter 2

Make sure you do these in an m-file and not the command line since many of the questions build upon the answers to earlier questions.

Q 2.1: Replacing letters in strings

a) Start with a string containing CHEERIOS and replace the 'C' with an 'O' and the 'R' with a 'P' to spell OHEEPIOS.

b) Start with a string containing CHEMISTRY and replace the 'C' with 'O' and the 'R' with 'B' to spell OHEMISTBY. Try to do it in a single line.

c) Start with string containing MACARONI AND CHEESE and replace all 'C's with 'O's and 'I's with 'R's to spell MAOARONR AND OHEESE.

Q 2.2: More replacing letters in strings.

Starting with the string:

```
str = 'MACARONI AND CHEESE'
```

a) Create a vector called `id1` so that when you type:

```
str(id1)
```

you get

```
MAN CHEESE
```

(Hint, write the phrase 'MACARONI AND CHEESE' on a piece of paper and number each letter.)

b) Create another vector `id2` so that

```
str(id2)
```

gives you

```
HER MIND IS CHINESE
```

Q 2.3: Creating vectors.

Create the following vectors using both `linspace` and the colon '`:`' technique

a) `[1 2 3 4 5 6 7 8 9 10]`

b) `[10 12 14 16 18]`

c) `[19 18 17 16 15]`

d) `[10 8 6 4 2 0 -2 -4]`

e) `[0 3.1416 6.2832 9.4248 12.5664 15.7080]`

f) Type the command `pi` in the command window. Now do exercise e) again but using the predefined variable `pi`.

Q 2.4: More indexing into strings.

Start with the following string of 20 'a's:

```
str = 'aaaaaaaaaaaaaaaaaaaa';
```

a) Use indexing to make every third letter in the string 'c'

b) Then use indexing again to turn the string into: `'abcabcabcabcabcabcab'`

c) Demonstrate that the third letter in the string is a 'c'

d) Now use indexing to turn this string into: `'abcdefabcabcabcabcab'`

e) Use indexing to turn this string into: `'abcdefabcdefabcdefab'.` Try to do it in one line of code.

f) Demonstrate that the 6th, 12th and 18th letters in the string are 'f's

Q 2.5: Indexing into vectors

You ran an experiment where you took 40 measurements every 1.23 seconds starting 12 seconds into the experiment.

Create a vector that describes the moments in time that these measurements were taken.

b) When was the fifth measurement taken?

c) Display the last measurement, using the command `end`.

Q 2.6: Still more indexing

Imagine you are running an experiment where stimuli flash onto the screen. Subjects are asked to hit the 'r' key if they see a face that looks like Russell Crowe and 'e' if they see a face that looks like Eddie Izzard. While you are piloting the experiment you make the faces alternate on every trial, so you expect the results of 40 trials to look as follows:

```
resp='rerererererererererererererererererererere
re';
```

a) Now imagine that while you were piloting you were distracted on what you thought was the 5th trial and hit the 'k' key. So `resp` now looks like this:

```
resp='rerekererererererererererererererererererere
re';
```

How would you check that it was the 5th trial that contained the 'k'?

b) Replace the 'k' with an 'r'

c) How would you check to see that you always pressed an 'e' for Eddie Izzard?

Q 2.7: Checking your understanding

Make sure you understand what's happening in the following examples.

```
vect=3:10
vect =

  3  4  5  6  7  8  9 10
```

```
vect(5:-1:2)
ans =

  7  6  5  4

vect(vect(1:3))
ans =

  5  6  7
```

Now, create the following vector:

```
vect = 12:-1:1
vect =
12 11 10 9 8 7 6 5 4 3 2 1
```

Before typing anything at the Matlab prompt, guess what the output will be for:

a) `vect(1:12)`

b) `vect(12:-1:1)`

c) `vect([10 12 9 12 8 4])`

d) `vect(1:2)`

e) `vect(vect(1:2))`

f) `vect(vect)`

g) `vect(vect(vect))`

CHAPTER 3: MATRICES AND BASIC CALCULATIONS

3.1 Matrices

As mentioned in the earlier chapter, you can have lists of numbers as well as of letters. These lists of numbers can be one-dimensional, in which case they are called a *vector*. When they are two, three, or more-dimensional they are called a *matrix*. Here we'll define a variable **mat1**to be a 2-dimensional matrix with 4 rows and 3 columns (called a 4x3 matrix).

```
mat1=[1 54 3; 2 1 5; 7 9 0; 0 1 0]
mat1 =
   1  54   3
   2   1   5
   7   9   0
   0   1   0

mat2=[1 54 3
 2 1 5
 7 9 0
 0 1 0]
mat2 =
   1  54   3
   2   1   5
   7   9   0
   0   1   0
```

Take a look at **mat1** and **mat2**. As you can see there is more than one way of entering a matrix. **mat1** and **mat2** were entered differently,

but both have 4 rows and 3 columns. When entering matrices a semi-colon is the equivalent of a new line.

You can find the size of matrices using the command **size**.

```
size(mat1)
ans =
  4  3
```

For a two dimensional matrix the first value in **size** is the number of rows. The second value of size is the number of columns. Now try:

```
vect1=[1 2 4 6 3]
vect1 =
  1  2  4  6  3

vect2=vect1'
vect2 =
     1
     2
     4
     6
     3
```

Vectors can be tall instead of long, ' (the little symbol below the double quote on your keyboard) is a *transpose*, and allows you to swop rows and columns of a vector or a matrix.

There is also command whos which will tell you the size of all your variables at once? Use **whos** to look at the size of **mat1**, **mat2**, **vect1** and **vect2**.

```
whos
Name      Size      Bytes Class   Attributes
```

```
M1          2x3          48 double
SPM5Path    1x34          68 char
V1          1x4          32 double
V2          1x4          32 double
ans         1x2          16 double
mat         3x5          120 double
mat1        4x3          96 double
mat2        4x3          96 double
vect1       1x5          40 double
vect2       5x1          40 double
```

Scalar?

In physics a scalar is a value that only has magnitude (and not direction). In programming a scalar is defined as a quantity that can only hold a single value at a time, i.e. a single number or character. Generally the expression *scalar* tends to be used to refer to numbers rather than characters.

3.2 Addition and subtraction

You can perform various calculations on matrices and arrays. You can add a single number (generally called a *scalar*) to a vector.

```
vect1+3
ans =
  4   5   7   9   6
```

You can subtract a scalar.

```
vect2-3
ans =
  -2
```

```
-1
 1
 3
 0
```

You can also add two vectors as long as they are the same size. This adds the numbers 'point-wise', meaning that each element from one vector is added to its corresponding element in the other vector. For example, you can add a vector onto itself:

```
vect1+vect1
ans =
  2  4  8 12  6
```

You can't add **vect1** and **vect2** together since they are different sizes.

```
vect1+vect2
??? Error using ==> plus
Matrix dimensions must agree.
```

Matrix dimensions must agree?

You get this error if you try to add vectors of inappropriate sizes. To successfully add two vectors they must be the same orientation as well as the same length – i.e. you can't add a tall thin vector to a short fat vector. The same goes for other operations (subtraction, point-wise multiplication & division and matrix multiplication & division) and applies to matrices as well as vectors.

When you get this error the first thing you should do is check the size of all the variables that you are trying to manipulate, and try to work out why Matlab thinks they are mismatched in size or shape. Often it's the

case that simply transposing one of the variables is all you need to do. This is an important thing to understand, so play with a few examples of your own and make sure you understand this.

```
vect3= [1 2 3 4];
vect4=[1 3 5 2]';
vect3+vect4;
??? Error using ==> plus
Matrix dimensions must agree.

mat1=[1 2 3; 4 5 6];
mat2=[1 2; 3 4; 5 6];
mat1-mat2'
ans =
  0 -1 -2
  2  1  0

mat1-mat2
??? Error using ==> minus
Matrix dimensions must agree.
```

3.3 Scalar multiplication and division

For scalar multiplication and division you use the symbols '*****' and '**/**'. You can a multiply a scalar with another scalar, with a vector, or with a matrix.

```
2*3
ans =
  6
```

```
2/3
ans =
  0.6667

vect1*3
ans =
   3   6  12  18   9

vect1/2
ans =
  0.5000  1.0000  2.0000  3.0000  1.5000

mat1*0.5
ans =
  0.5000  1.0000  1.5000
  2.0000  2.5000  3.0000

3+1*4
ans =
   7

3*4+1
ans =
  13
```

Multiplication & division has priority over addition & subtraction. So the statements above are the equivalent of 3+(1*4) and (3*4)+1. If you want to do addition or subtraction before multiplication or division you need to use brackets.

```
(3+1)*4
ans =
```

```
 16
```

```
3*(4+1)
ans =
 15
```

Vector multiplication and division

There are three ways to multiply and divide two vectors. These are *point-wise*, *outer-product* and *inner-product* multiplication respectively.

3.4 Point-wise vector multiplication and division

Point-wise (or *element by element*) multiplication and division is the simplest. In Matlab these are computed using `.*` and `./` (notice the period character). Each element in the first vector is multiplied by the corresponding element in the second vector (Figure 3.1). As with addition and subtraction, the vectors must be the same shape.

```
V1=[1 2 3 4];
V2=[2 3 4 5];
V1.*V2
ans =
  2  6 12 20
```

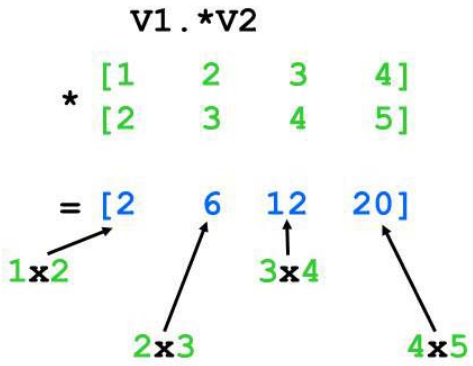


Figure 3.1. Point-wise multiplication of two vectors.

```
V2.*V1
ans =
  2  6 12 20

V1./V2
ans =
 0.5000 0.6667 0.7500 0.8000


V2./V1
ans =
 2.0000 1.5000 1.3333 1.2500
```

Note what happens if we make one of the two vectors tall and thin (using the **' transpose**). We get an error stating that we are trying to multiply two things of different shapes.

```
V1.*V2'
??? Error using ==> times
Matrix dimensions must agree.


V1./V2'
??? Error using ==> rdivide
Matrix dimensions must agree.
```

Try reorienting V1 so they multiply successfully. Now do it by

reorienting **V2**.
Here's another example.

```
M1=[1 2 3; 4 5 6]
M1 =
  1  2  3
  4  5  6
V1.*M1
??? Error using ==> times
Matrix dimensions must agree.
```

This multiplication is never going to happen for you, regardless of how you transpose **M1** and **V1** because these two variables will never be the same size.

3.5 Inner product (dot product)

The second kind of multiplication, the *inner-product* or *dot product* is a bit more complicated. For two vectors, the inner product is equal to the sum of the point-wise multiplication, but the first vector must be a row vector and the second vector must be a column vector (of the same length).

```
V1 = [1 2 4]
V1 =
  1  2  4

V2 = [1.1 2.2 3.3]'
V2 =
  1.1
  2.2
  3.3

V1*V2
```

```
ans =
 18.7000
```

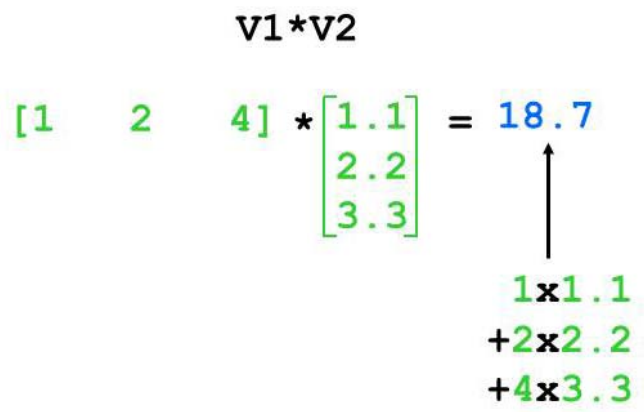


Figure 3.2. The inner product of two vectors.

To verify that this is the sum of the point-wise product, we need to transpose V2 (or V1) to do the point-wise multiplication:

```
V1.*V2'
ans =
  1.1000 4.4000 13.2000

sum(V1.*V2')
ans =
 18.7000
```

3.5 Outer product

The outer product of two vectors is a matrix. The shapes of the vectors you need to calculate outer products are the opposite of inner products. The first vector must be a column vector and the second must be a row vector. We can demonstrate the outer product by multiplying V2 with V1:

```
V2*V1
ans =
 1.1000 2.2000 4.4000
 2.2000 4.4000 8.8000
 3.3000 6.6000 13.2000
```

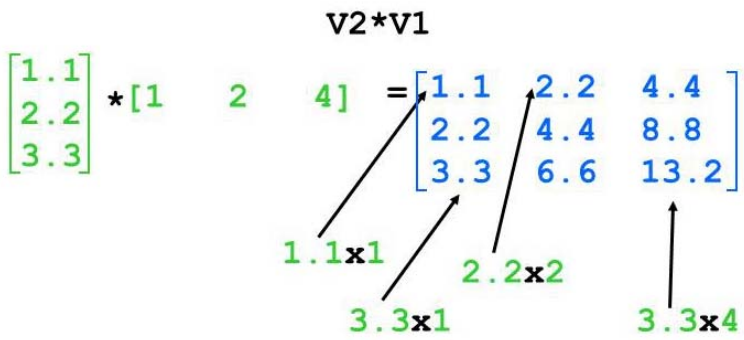


Figure 3.3. The outer product of two vectors.

Remember, in this example V2 is a column vector and V1 is a row vector. The result is a 3x3 matrix. For an element in row *i* and column *j*, the value is the product of the *ith* element in the column vector and the *jth* element in the row vector (figure 3.2).

Inner products and outer products use the same symbol '*'. Matlab knows whether to take the inner or outer product by the shapes of the vectors:

```
V2*V1
ans =
 1.1000 2.2000 4.4000
 2.2000 4.4000 8.8000
 3.3000 6.6000 13.2000

V1*V2
ans =
 18.7000
```

Here's another example of an outer product. Notice that they don't have to be the same length. The outer product of a 3-element column vector with a 4 element row vector will be a 3x4 matrix:

```
V3 = [1,2,3]';
V4 = [2,3,4,5];
V3*V4
ans =
  2  3  4  5
  4  6  8 10
  6  9 12 15
```

Think about the following examples and see if they make sense to you:

```
V4*V3
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

```
V4'*V3'
ans =
  2  4  6
  3  6  9
  4  8  12
  5  10 15
```

3.7 Matrix multiplication and division

An analogous type of multiplication and division occurs when A and B are matrices.

Point-wise multiplication for matrices is just the same as for vectors:

```
A = [1, 2; 3, 4]
A =
```

```
  1  2
  3  4

B = [0,1;1,0]
B =
  0  1
  1  0

A.*B
ans =
  0  2
  3  0
```

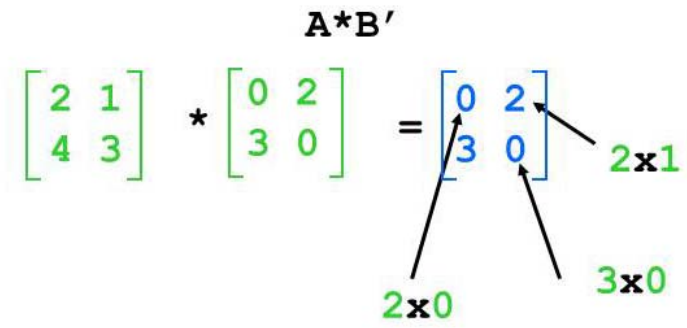


Figure 3.4. Point-wise matrix multiplication.

It is also possible to calculate the **inner product** for matrices, C=A*B. The number of columns in A still needs to match the number of rows in B. The size of the output depends on how you multiply the output. If A is m-by-p and B is p-by-n, their product C is m-by-n. I.e. **C has the same number of rows as A and the same number of columns as B**.

```
M1=[0,1,0; 1,0,1; 1,1,0]
M1 =
  0  1  0
  1  0  1
  1  1  0

M2=[1 2 3; 4 5 6'; 7 8 9; 1 2 3]
```

```
M2 =
  1   2   3
  4   5   6
  7   8   9
  1   2   3
```

M1 has 3 rows and 3 columns. M2 has 4 rows and 3 columns. That means we can calculate the following matrix multiplications:

```
M1*M2'
ans =
     2   5   8   2
    4  10  16   4
    3   9  15   3
```

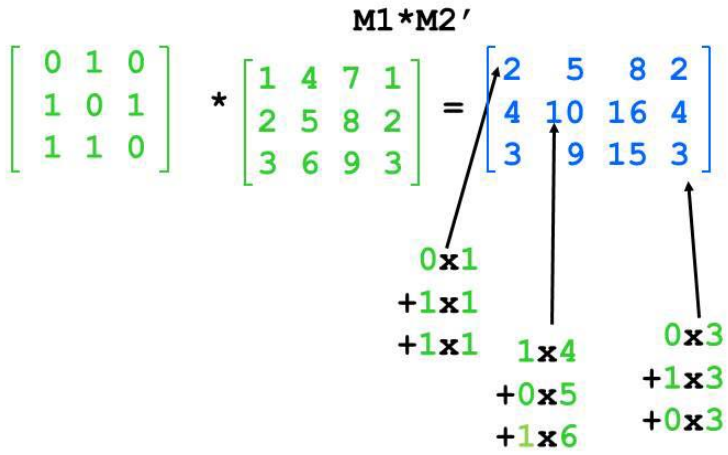


Figure 3.5 Inner product matrix multiplication.

```
M1'*M2'
ans =
  5  11  17   5
  4  10  16   4
  2   5   8   2
```

```
M2*M1
ans =
  5  4  2
 11 10  5
 17 16  8
  5  4  2

M2*M1'
ans =
  2  4  3
  5 10  9
  8 16 15
  2  4  3
```

The following multiplications aren't allowed because the number of columns in M1 doesn't match the number of rows in M2.

```
M1*M2
??? Error using ==> mtimes
Inner matrix dimensions must agree.

M2'*M1
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

3.8 More Calculation Stuff

To raise 10 to the 5th power (10^5) you simply do the following:

```
y=10^5
y =
```

```
  100000
```

To do this with vectors you need to add the period. You need to use the period regardless of whether you are taking a vector and raising it to a single number, taking a single number and raising it to a vector, or taking a vector and raising it to another vector (which will need to be of the same size and shape).

```
y=3;
y^2
ans =
  9

[1:5].^2
ans =
  1  4  9 16 25

y=2.^[1:5]
y =
  2  4  8 16 32

y=[2:6].^[1:5]
y =
  2  9  64  625  7776
```

Raising a number to the exponent of 0.5 is the same as taking the square root, which has its own function, **sqrt**, in Matlab:

```
y=4.^.5
y =
  2

y=sqrt(4)
y =
  2
```

For logarithms, you can take either the natural or the base 10 log of a number. It's important to remember that the default is the natural log.

```
y=log(10)
y =
 2.3026

y=log10(10)
y =
  1

y=log([5 6 7 8])
y =
1.6094 1.7918 1.9459 2.0794
```

The exponential function e^x is simply:

```
y=exp(1:3)
y =
2.7183 7.3891 20.0855
```

Other useful commands are **round**, **min** and **max**.

```
round(3.14)
ans =
  3


x=1:5; min(x)
ans =
  1


max(x)
ans =
```

```
  5
```

When **max** and **min** are used on a matrix they give you the minimum along each column, not the minimum of the entire matrix. The command max works in a similar way.

```
mat=[1 2 3 4 5; 1.1 2.2 3.3 4.4 5.5];
min(mat)
ans =
  1   2   3   4   5
```

mod is command that computes modulo arithmetic. **mod(a,b)** returns the remainder when a is divided by b. For example, try:

```
mod(12,10)
ans =
  2

mod(pi,3)
ans =
 0.1416
```

Modulo arithmetic is useful for making periodic functions. For example, starting with a vector that counts from 0 to 9:

```
y = 0:9
y =
  0   1   2   3   4   5   6   7   8   9
```

a 'sawtooth' can be made by taking the vector modulo 5.

```
mod(y,5)
ans =
  0   1   2   3   4   0   1   2   3   4
```

This is a good time to begin to start using **doc** (the Matlab help). Try looking up some of the commands you already know using **doc** and reading them to get a sense of how Matlab describes them. E.g.

```
>> doc round
```

Once the doc window is open you can just search for commands by typing them directly into the search line there.

Questions for Chapter 3

Q 3.1: Basic calculations

Create the following matrix **mat**:

```
mat=[ 1 2 3 4; 4 5 6 7; 8 9 10 11];
```

If you look at the size of **mat** you will see that it has 3 rows and 4 columns.

```
size(mat)
ans =
  3  4
```

a) Before typing anything in, guess what will result from:

```
mat + 1
```

b) Guess what you get when you type:

```
10-mat
```

c) Use the command **mim** to calculate the minimum of each column of **mat**.

d) Use `min` and the transpose (`'`) to calculate the minimum of each row of **mat**.

e) Use **min** twice to calculate the minimum of the entire matrix.

Q 3.2: Calculations with vectors

Define the following vectors:

```
v1 =[1 2 3 4];
v2 =[1 0 1 0];
```

Before typing anything in, guess what will result from:

a) `v1 + v2`
b) `v1 .* v2`
c) `sum(v1.*v2)`
d) `v1 * v2'`

Q 3.3: More calculations with vectors

Calculate the sum of all odd numbers from 1 to 99

Q 3.4: Estimating pi

The number π can be expressed as an infinite series:

$$\pi = \sqrt{\frac{6}{1^2} + \frac{6}{2^2} + \frac{6}{3^2} + \frac{6}{4^2} + \frac{6}{5^2} + \cdots}$$

We'll approximate this in the following steps:

a) Generate a vector a that counts in steps of 1 from 1 to 10,000 (don't forget the semicolon to suppress the output!)

b) Square every element of that vector and call this vector b.

c) Create a new vector, `c`, that is 6 divided by every element of vector `b`

d) Create a scalar, `d`, that is the sum of vector `c`

e) Take the square root of scalar `d`.

f) For extra credit, try to do to this all in a single line!

Q 3.5: Inner and Outer Products

Let the vector `heights` be a list of heights in inches:

```
heights = [66 68 65 70 65]';
```

a) Find the average of these heights by adding the heights and dividing by 5.

b) Use Matlab's `mean` function to get the same number.

Let the vector `w` be: `w = [1 1 1 1 1]/5;`

c) Show that the inner product of `w` and `heights` is the same as the mean. Do you see why?

d) Now let `w = [1 1 1 0 1]/4;`

Show that the inner product of `w` and `heights` is the mean height after taking out the fourth value.

CHAPTER 4: MULTIDIMENSIONAL MATRICES, INDEXING, LOOPS AND LOGICAL OPERATIONS

4.1 Multidimensional matrices

Matrices can be 3, 4 or more dimensions, though some commands won't work for matrices with more than 2 or 3 dimensions, as you will discover later in this book.

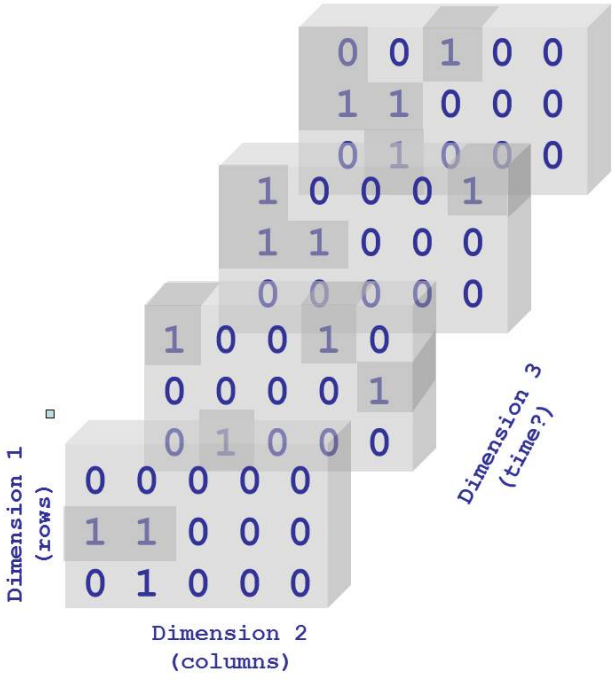


Figure 4.1 A three dimensional matrix.

```
clear all
mat(:, :,1)=[0 1 1 0 ; 0 0 0 0 ; 0 0 0 1; 0
1 0 0 ; 0 0 1 0 ];
mat(:, :, 2)=[ 1 0 1 1; 1 0 1 1 ; 0 0 0 0 ;
0 1 0 0 ;0 1 1 1];
mat(:, :, 3)=[0 0 0 0 ; 1 1 0 1; 0 0 0 0 ; 0
0 0 0 ; 0 0 0 0 ];
mat
mat(:,:,1) =
  0  1  1  0
  0  0  0  0
  0  0  0  1
  0  1  0  0
  0  0  1  0
mat(:,:,2) =
  1  0  1  1
  1  0  1  1
  0  0  0  0
  0  1  0  0
  0  1  1  1
mat(:,:,3) =
  0  0  0  0
  1  1  0  1
  0  0  0  0
  0  0  0  0
  0  0  0  0

size(mat)
ans =
  5  4  3
```

There are many circumstances where a three-dimensional matrix is useful in the behavioral sciences. One of the most common examples in the behavioral sciences is when you want to present subjects with a

series of images over time. In that case it is very natural to use the first and second dimensions to represent each image in space, and the third dimension to represent time. In that case, if you wanted to look at the first image presented in time you would reference it as follows:

```
mat(:, :, 1)
ans =
  0  1  1  0
  0  0  0  0
  0  0  0  1
  0  1  0  0
  0  0  1  0
```

The third image in time would be:

```
mat(:, :, 3)
ans =
  0  0  0  0
  1  1  0  1
  0  0  0  0
  0  0  0  0
  0  0  0  0
```

If you wanted to look at what was happening in the top left corner of the screen over time, you would reference it as follows:

```
mat(1, 1, : )
ans(:,:,1) =
  0
ans(:,:,2) =
  1
ans(:,:,3) =
  0
```

If you wanted to know the values in the bottom line of the screen in the first image you would reference it as follows:

```
mat(3, :, 1)
ans =
  0  0  0  1
```

Finally, if you wanted to know what was happening along the bottom line of the screen over time …

```
mat(3, :, : )
ans(:,:,1) =
  0  0  0  1
ans(:,:,2) =
  0  0  0  0
ans(:,:,3) =
  0  0  0  0
```

Now, here's an example of how a four dimensional matrix might be used to represent data. Functional magnetic resonance data is generally represented in four dimensions. The first two dimensions each represent a single image (slice) through the brain. The third dimension represents all the slices needed to represent the entire brain. The fourth dimension represents the time course of the experiment. You can think of it as multiple cubes over time. Let's create a fake fMRI dataset with 64x64 voxels in the slice, 28 slices and 80 time points. We'll begin by filling this matrix with random noise using the Matlab command `randn`. This is a command that generates normally distributed random numbers with a mean of zero and a standard deviation of 1. In the case below, `randn` creates a 4d matrix, 64 x 64 x 28 x 80 in size, filled with random numbers.

```
fakefmri=randn(64,64, 28, 80);
```

Now let's create a vector of the points in time when the stimulus was presented.

```
ontimepts=[1:10, 21:30, 31:40, 41:50,
61:70];
```

The vectors 5:10, 7:17 and 5:8 refer to the spatial position of a rectangular chunk of our 'brain'. The `ontimepts` vector refers to points in time. We are going to add the number 1 to that chunk of the brain during those time points.

```
fakefmri(5:10, 7:17, 5:8,ontimepts
)=fakefmri(5:10, 7:17, 5:8,ontimepts)+1;
```

Let's start by looking at a single voxel over time.

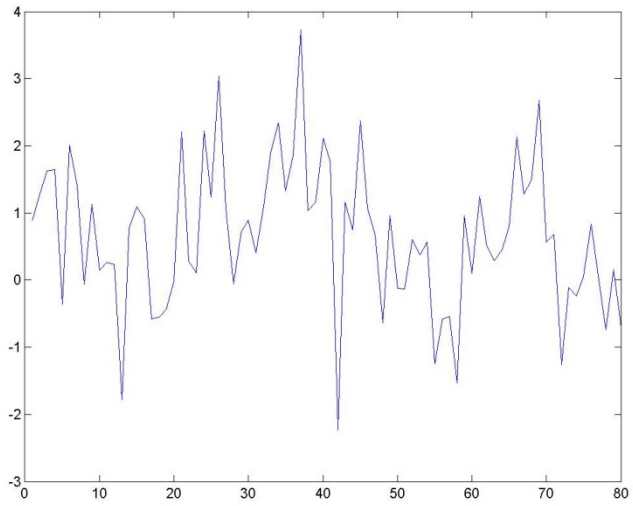
```
size(fakefmri(6, 8, 7,: ))
ans =
  1   1   1 80
```

So what we really have here is a vector of 80 time points, but because it was pulled from a 4d matrix, Matlab thinks of it as a 1 x1 x 1 by 80 matrix. `squeeze` is a command that removes these singleton dimensions and turns it into a standard vector.

```
timepts=squeeze(fakefmri(6, 6, 7,: ))
size(timepts)
ans =
 80   1
```

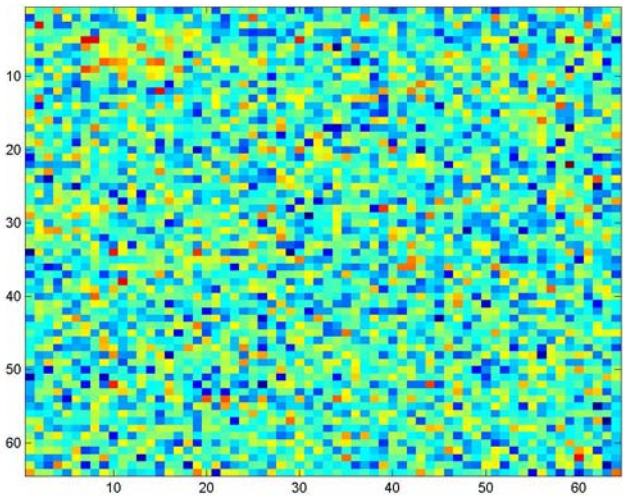
There will be a great deal about `plot` later, but what we're doing here should be pretty self-evident. Your plot will look very slightly different because `randn` generates a different set of random numbers every time it is called.

```
plot(1:80, timepts)
```



Now let's look at a single slice at a single time point (again, you'll learn more about `imagesc` later).

```
imagesc(fakefmri(:, :, 7, 22))
```

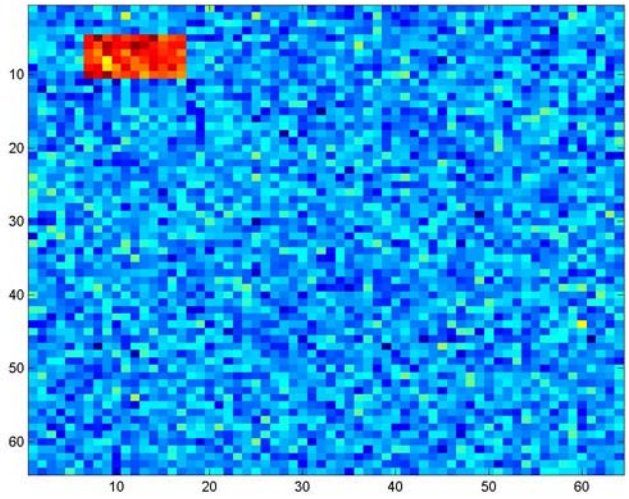


Now let's look at a single slice averaged across the `ontimepts`.

```
mnfakefmri=mean(fakefmri(:, :, :,
ontimepts), 4);
```

Note that `mnfakefmri` is a 3d vector the size of the brain, since we have averaged over time.

```
size(mnfakefmri)
ans =

 64 64 28
imagesc(mnfakefmri(:, :, 7))
```



4.2 for loops

MakingMatrices.m

You should create these matrices in an m-file for convenience sake, since you might want to modify them for the questions for this chapter. Save the m-file as MakingMatrices.m.

```
% MakingMatrices.m
% This program provides examples of cunning
% ways to create matrices
% Written IF 3/2007

mat1=zeros(5, 4);
```

The command `zeros(5, 4)` creates a matrix containing all zeros with five rows and four columns. Remember that the first dimension in a matrix is the rows, and the second dimension is the columns. Also try using the command `ones` (which works in a similar way) to make a matrix with 7 rows and 3 columns. Other commands that work in a similar way are `rand` (which fills the matrix with random values varying between 0-1) and `randn` (which fills the matrix with normally distributed noise with zero mean and unity variance). In this next example you are setting all the rows in the third column to be 1.

```
mat2=mat1;
mat2(1:5, 3)=1
mat2 =
  0  0  1  0
  0  0  1  0
  0  0  1  0
  0  0  1  0
  0  0  1  0
```

Another way of writing this command would be as follows. Instead of specifying that `mat2` has five rows, we use the expression `1:end` to tell Matlab to use the vector that goes from the first row to the last row.

```
mat2=mat1;
mat2(1:end,3)=6
mat2 =
```

```
  0   0   6   0
  0   0   6   0
  0   0   6   0
  0   0   6   0
  0   0   6   0
```

Finally, here's an even more succinct shortcut where we simply use a colon to tell Matlab to include all rows within that column.

```
mat2=mat1;
mat2(:, 3)=7
mat2 =
  0   0   7   0
  0   0   7   0
  0   0   7   0
  0   0   7   0
  0   0   7   0
```

Now let's set all the columns in the fourth row to be 1. The colon now means to include the entire column.

```
mat3=mat1;
mat3(4, 1:4)=1
mat3 =
  0   0   0   0
  0   0   0   0
  0   0   0   0
  1   1   1   1
  0   0   0   0
```

Other ways of writing this command would be:

```
mat3(4, 1:end)=1;
mat3(4, :)=1
```

```
mat3 =
  0  0  0  0
  0  0  0  0
  0  0  0  0
  1  1  1  1
  0  0  0  0
```

Here we create a matrix that is four by four zeros (it is a weird convention of the `zeros` command that simply giving a single argument (4) makes a 4x4 square matrix).

```
mat4=zeros(4)
mat4 =
  0  0  0  0
  0  0  0  0
  0  0  0  0
  0  0  0  0
```

Then we go through each row of the matrix, and replace the place in the matrix that is the i-th row and i-th column with the number i. We are doing this using a `for` loop. Matlab will go through the loop four times.

Each time Matlab goes through the loop it will wait at the `pause` command for you to press a key. The first time it goes through the loop i will be equal to 1, so the command `mat4(i, i)=i;` will be the equivalent of `mat4(1, 1)=1`. The second time it goes through the loop i will be 2, so `mat4(2, 2)=2`, and so on up to `i=4`, `mat(4, 4)=4`. I've left the semicolon off so you can watch `mat4` gradually change. In the m-file:

```
mat4=zeros(4);
for i=1:4
 mat4(i, i)=i
 pause
end
```

```
mat4 =
  1  0  0  0
  0  0  0  0
  0  0  0  0
  0  0  0  0
mat4 =
  1  0  0  0
  0  2  0  0
  0  0  0  0
  0  0  0  0
mat4 =
  1  0  0  0
  0  2  0  0
  0  0  3  0
  0  0  0  0
mat4 =
  1  0  0  0
  0  2  0  0
  0  0  3  0
  0  0  0  4
```

In this next example we sequentially go through five rows of a matrix filling the columns with a vector.

```
mat5=zeros(6);
for i=1:6
mat5(i,:)=[-2 0 -1 1 2 3]
 pause
end

mat5 =
  -2   0  -1   1   2   3
   0   0   0   0   0   0
```

```
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0

mat5 =
    -2     0    -1     1     2     3
    -2     0    -1     1     2     3
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0

mat5 =
    -2     0    -1     1     2     3
    -2     0    -1     1     2     3
    -2     0    -1     1     2     3
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0

mat5 =
    -2     0    -1     1     2     3
    -2     0    -1     1     2     3
    -2     0    -1     1     2     3
    -2     0    -1     1     2     3
     0     0     0     0     0     0
     0     0     0     0     0     0

mat5 =
    -2     0    -1     1     2     3
    -2     0    -1     1     2     3
    -2     0    -1     1     2     3
    -2     0    -1     1     2     3
    -2     0    -1     1     2     3
     0     0     0     0     0     0
```

```
mat5 =
  -2   0  -1   1   2   3
  -2   0  -1   1   2   3
  -2   0  -1   1   2   3
  -2   0  -1   1   2   3
  -2   0  -1   1   2   3
  -2   0  -1   1   2   3
```

Now try filling up the rows of a matrix using the vector `[-1 -2 0 -1 1 2 3].`

```
mat5=zeros(6);
for i=1:6
 mat5(i,:)=[-1 -2 0 -1 1 2 3]
  pause
end
??? Subscripted assignment dimension
mismatch.
```

You get an error! This is because you are trying to squeeze a vector that is 1 row and seven columns ([-1 -2 0 -1 1 2 3]) into the row of a matrix that only has six columns.

Subscripted assignment dimension mismatch?

This error is because you are trying to squeeze a vector that is 1 row and 7 columns into the row of a matrix that only has six columns. You often get this error because one of your vectors or matrices is the wrong size and/or shape, as in the examples here. As described above, the subscripted assignment dimension error means you are trying to fit a square peg into a round hole.

Here are some other examples of dimension mismatch errors.

```
x=zeros(4)
```

```
x =
  0  0  0  0
  0  0  0  0
  0  0  0  0
  0  0  0  0

x(2, :) = 1:5
??? Subscripted assignment dimension
mismatch.

x=zeros(2, 4)
x =
  0  0  0  0
  0  0  0  0

y=ones(4, 2)
y =
  1  1
  1  1
  1  1
  1  1

x(1, :)= y(1, :)
??? Subscripted assignment dimension
mismatch.

x=zeros(2, 4)
x =
  0  0  0  0
  0  0  0  0

y=ones(5, 5)
y =
  1  1  1  1  1
```

```
  1   1   1   1   1
  1   1   1   1   1
  1   1   1   1   1
  1   1   1   1   1
```

```
x(:, :)=y
??? Subscripted assignment dimension
mismatch.
```

Weirdly, the following command, which seems identical, **will** work. Because you specified that you wanted rows 1-5 and columns 1-5 Matlab assumed you knew what you were doing and expanded the matrix for you.

```
x=zeros(2, 4);
y=ones(5, 5);
x(1:5, 1:5)=y
x =
   1   1   1   1   1
   1   1   1   1   1
   1   1   1   1   1
   1   1   1   1   1
   1   1   1   1   1
```

Here's yet another matrix. Here we are saying we want to place ones in the matrix in the positions where the rows are between two through five *and* the columns are between one and three.

```
mat6=zeros(6);
mat6(2:5, 1:3)=1
mat6 =
   0   0   0   0   0   0
   1   1   1   0   0   0
   1   1   1   0   0   0
```

```
   1   1   1   0   0   0
   1   1   1   0   0   0
   0   0   0   0   0   0
```

In the next example, what we put into the matrix on each iteration of the loop depends on the value of `i`. When `i` is one, then the vector `[0 0 1 1 0]+1=[1 1 2 2 1]` is put into the first row of `mat7`. When `i` is two, then the vector `[0 0 1 1 0]+2=[2 2 3 3 2]` is put into the second row of `mat7` and so on.

```
mat7=zeros(5);
for i=1:5
 mat7(i, :)= [0 0 1 1 0]+i;
end
```

```
mat7
mat7 =
   1   1   2   2   1
   2   2   3   3   2
   3   3   4   4   3
   4   4   5   5   4
   5   5   6   6   5
```

Now let's do the same thing but we'll put the vectors in along the columns instead of the rows. All we need to do is change `mat7(i, :)` to `mat7(:, i)`.

```
mat7=zeros(5);
for i=1:5
 mat7(:, i)= [0 0 1 1 0]+i;
end
```

```
mat7
mat7 =
   1   2   3   4   5
```

```
   1    2    3    4    5
   2    3    4    5    6
   2    3    4    5    6
   1    2    3    4    5
```

Technically we're trying to put a vector that has one row and five columns into a space that has five rows and one columns. So if Matlab wanted to be pedantic it would give the *'Subscripted assignment dimension mismatch error'*. But Matlab, rather tolerantly, will let this one go. It would be better style to make the two vectors match in shape by transposing the right hand side as follows:

```
mat7(:, i)=[0 0 1 1 0]'+i
```

In the next example we are using a *nested* loop. Basically it is one `for` loop inside another. So in this example we use `i` to go down each row. For each value of `i` we then use `j` to go through each column in turn.

```
mat8=zeros(3, 4);
for i=1:3
 for j=1:4
  mat8(i, j)=i+j
 end
end
```

```
mat8 =
   2   0   0   0
   0   0   0   0
   0   0   0   0
mat8 =
   2   3   0   0
   0   0   0   0
   0   0   0   0
mat8 =
```

```
    2    3    4    0
    0    0    0    0
    0    0    0    0
mat8 =
    2    3    4    5
    0    0    0    0
    0    0    0    0
mat8 =
    2    3    4    5
    3    0    0    0
    0    0    0    0
mat8 =
    2    3    4    5
    3    4    0    0
    0    0    0    0
mat8 =
    2    3    4    5
    3    4    5    0
    0    0    0    0
mat8 =
    2    3    4    5
    3    4    5    6
    0    0    0    0
mat8 =
    2    3    4    5
    3    4    5    6
    4    0    0    0
mat8 =
    2    3    4    5
    3    4    5    6
    4    5    0    0
mat8 =
    2    3    4    5
    3    4    5    6
```

```
    4    5    6    0
mat8 =
    2    3    4    5
    3    4    5    6
    4    5    6    7
```

Here's a tricky one. The expression `((i-1)*4)+j` means that when we are on the first row, the columns are labeled as 1,2,3,4. When we are on the second row the columns are labeled as 5,6,7,8 and so on … When `i=1` then `((i-1)*4)=0`, so the columns are filled with the value of `j`. When `i=2` then `((i-1)*4)=4`, so the columns are filled with the value of `j+4`, and so on. Spend some time on this since you need to use a similar trick in the questions following this chapter.

```
mat9=zeros(3,4);
for i=1:3
 for j=1:4
  mat9(i, j)=((i-1)*4)+j
 end
end
```

```
mat9 =
    1    0    0    0
    0    0    0    0
    0    0    0    0
mat9 =
    1    2    0    0
    0    0    0    0
    0    0    0    0
mat9 =
    1    2    3    0
    0    0    0    0
    0    0    0    0
mat9 =
```

```
   1   2   3   4
   0   0   0   0
   0   0   0   0
mat9 =
   1   2   3   4
   5   0   0   0
   0   0   0   0
mat9 =
   1   2   3   4
   5   6   0   0
   0   0   0   0
mat9 =
   1   2   3   4
   5   6   7   0
   0   0   0   0
mat9 =
   1   2   3   4
   5   6   7   8
   0   0   0   0
mat9 =
   1   2   3   4
   5   6   7   8
   9   0   0   0
mat9 =
   1   2   3   4
   5   6   7   8
   9  10   0   0
mat9 =
   1   2   3   4
   5   6   7   8
   9  10  11   0
mat9 =
   1   2   3   4
   5   6   7   8
```

```
    9   10   11   12
```

In fact, you don't do a nested loop to do this – here's how to do this with a single loop.

```
mat9=zeros(3,4);
for i=1:3
 mat9(i, :)= ((i-1)*4)+[1:4]
end
```

```
mat9 =
    1    2    3    4
    0    0    0    0
    0    0    0    0
mat9 =
    1    2    3    4
    5    6    7    8
    0    0    0    0
mat9 =
    1    2    3    4
    5    6    7    8
    9   10   11   12
```

Being able to do these sorts of manipulations is one of the keys to being able to write good code in Matlab, so take your time and make sure you understand how these examples work. Also, make sure you can do all the exercises before you move onto the next chapter.

4.3 repmat

`repmat` is a command for replicating arrays. `x` is a vector with 1 row and 5 columns.

```
x=[1 5 3 2 5];
size(x)
```

```
ans =
      1   5
```

First, let's use `repmat` to create a matrix that is `x` replicated 3 times along the row dimension, and 2 times along the column dimension.

```
X=repmat(x, 3, 2);
size(X)
ans =
     3  10

X
X =
1   5   3   2   5   1   5   3   2   5
1   5   3   2   5   1   5   3   2   5
1   5   3   2   5   1   5   3   2   5
```

We can also replicate matrices:

```
X=[2 3 4; 5 6 7];
size(X)
ans =
  2  3

Y=repmat(X, 3, 2);
size(Y)
ans =
  6  6

Y
Y =
   2   3   4   2   3   4
   5   6   7   5   6   7
   2   3   4   2   3   4
   5   6   7   5   6   7
   2   3   4   2   3   4
   5   6   7   5   6   7
```

4.4 sub2ind and ind2sub

Sometimes it's useful to convert back and forth between vectors and matrices.

```
mat=[1 2 3; 4 5 6; 7 8 9];
vect=mat( : );
mat=[1 2 3; 4 5 6; 7 8 9]
vect=mat(:) ;
mat =
   1   2   3
   4   5   6
   7   8   9
```

The colon (the colon has many purposes in Matlab) tells Matlab to unwrap the matrix `mat` out to be a vector. Note that the unwrapped vector first lists all the rows in the first column, then lists all the rows in the second column, and so on …

Suppose you wanted to know where the number eight would be in the vector? Well, the number eight appears in the third row and the second column of the matrix. The 3rd row and 2nd column are known as the row and column *subscripts* of the matrix mat. The `sub2ind` command below calculates the *index* in `vect` corresponding to the 3rd row and 2nd column (the subscripts) of `mat`. You simply need to tell it the size of `mat`, and the row and columns subscripts.

```
ind=sub2ind(size(mat), 3, 2);
ind
ind =
  6
```

So the 3rd row and 2nd column in the matrix corresponds to the 6th *index*

into the vector. So both of the following should give you the number eight.

```
vect(ind)
ans =
   8

mat(3, 2)
ans =
   8
```

You can also go the other way using `ind2sub`. Once again you need to provide the size of the matrix and the position in the vector that you want to find the matrix subscripts for.

```
[sub_row, sub_col]=ind2sub(size(mat), 6);

sub_row
sub_row =
  3

sub_col
sub_col =
  2
```

4.5 Logical operations: equal to, greater than

Logical operations are statements checking the truth of statements. In programming truth is expressed as being 1 and falsity is 0. This is sometimes called logical 1 and logical 0 to express the fact that 1 and 0 are being used to express truth statements. Note that you use a single = to assign a value to a variable. You use a double equal == to determine whether the value of the left hand side variable is equal to that of the right hand side variable. The expression below gives you a logical 0 because the statement is false.

```
n1=2; n2=4;
n1==3.2
ans =
   0

n1==n2
ans =
   0
```

This expression is true.

```
n1=4; n1==n2
ans =
   1
```

You can also see whether one number is *not* equal to another. It is false that `n1` is not equal to `n2`.

```
n1=4; n2=4;
n1~=n2
ans =
  0
```

This is true.

```
n2=5; n1~=n2
ans =
   1
```

`and` (`&&`) and `or` (`||`) operators are used when the truth of a statement depends on more than one condition being true (&&) or on either of two conditions being true (**||**). Try the following:

```
clear all;
n1=1; n2=2; n3=3;
n1<n2 && n1>n3
ans =
   0
```

The first part of the condition statement is true (`n1>n2`), `n1` is greater than `n2`. The second part isn't (`n1>=n3`), `n1` is not greater or equal to `n3`. So the statement as a whole is false because the `and` operator (&&) requires that both condition statements are true.

```
n1<n2 || n1>n3
ans =
   1
```

Once again, the first part of the condition statement is true (`n1>n2`), but the second part isn't (`n1>n3`). However this time the statement as a whole is true because the `or` operator (`||`) only requires that either the first statement or the second is true.

4.6 find

One way in which logical operations are often used is in the command `find`, which is used to find the index for a particular value in a vector or matrix. So if you want to know where the 'F's are in 'FROSTED FLAKES' you would use `find` as follows.

```
ff=find('FROSTED FLAKES' =='F')
ff =
  1  9
```

or the equivalent:

```
frosted='FROSTED FLAKES';
ff=find(frosted =='F')
ff =
  1  9
```

You can also use `find` for vectors:

```
A=[ 5 6 7 9 3 4 ];
F=find(A==3)
F
F =
  5
```

You can also find the index of a value in a matrix using find, but this time you need the command to return two values, because the found value has both a row and a column index.

```
mat2=[1 54 3
 2 1 5
 7 9 0
 0 1 0]
[rr, cc]=find(mat2==0)
[rr, cc]
ans =
   4   1
   3   3
   4   3
```

In `mat2` there are 3 zeros. The first one is in the 4th row and the 1st column, the second is in the 3rd row and the 3rd column and the last one is in the 4th row and the 3rd column.

4.7 if/else statements

Another common way of using logical truth operations is in an

`if` statement. An if statement checks whether the statement following the `if` is true or not. If the statement is true, Matlab carries out the operations between the `if` and the `end`. If the statement is false Matlab doesn't carry out those operations.

Try these logical truth statements in a little junk m-file (much easier than typing them into a command window).

```
n1=3.2
if round(n1)==n1
disp('n is a round number');
end
```

```
n1 =
 3.2000
```

You can also tell Matlab what to do if the statement is false, using an `else`.

```
n=3.2
if round(n1)==n1
 disp('n is a round number');
else
 disp('n is not a round number');
end
```

```
n =
 3.2000
n is not a round number
```

Traditionally the statement that is evaluated by an `if` statement is either a 1 or a 0 (true or false) but in Matlab commands following an `if` statement *will* always be executed unless the condition following the `if` results in a 0.

```
n=-1
```

```
if n
 disp('hi there cutie-pants')
else
 disp('bye-bye darling')
end

n =
   -1
hi there cutie-pants
```

You can even have more than 2 alternatives.

```
n=randn;
if n<=-.6
 disp('hi there cutie-pants')
elseif n<=0
 disp('bye-bye darling')
elseif n>0 && n<=.6
 disp('smoocherooo')
else
 disp('snuggle-puppy!')
end
```

If you have lots of alternatives (more than 3) it's a little tidier to use `switch/case`. That's beyond the scope of our book, but Matlab's documentation is pretty clear.

4.8 while statements

The way `while` loops work is that the statements in the loop are carried out repeatedly while the `while` statement is true.

```
n=0;
while n<1
 n=randn;
 disp(['n = ', num2str(n)])
```

```
end


n = 0.66863
n = -0.011588
n = -0.15838
n = 1.0704
```

`while` loops are often used for timing. Here's an example using the commands `tic` - which starts a stopwatch, and `toc` which collects how many seconds have gone by since the stopwatch started.

```
tic
while toc<3
;
end
```

The program pauses while Matlab spins around the loop checking whether it has been 3 seconds since the `tic` stopwatch started.

4.9 Inf and NaN

Try dividing 1 by zero with your calculator and the screen will show something like ERROR. But in a programing language, you typically want to keep things running if you happen to do something like this, rather than grinding your program to a halt. In Matlab if you divide by zero you'll get a special number: `Inf`. Try it:

```
1/0
ans =
 Inf
```

`Inf` is a valid number to assign to a variable. You can set a variable to it:

```
x = Inf;
x =
```

```
 Inf
```

Or it'll show up if you divide a variable that is set to zero. For example:

```
x = ones(1,5)
x =
1   1   1   1   1

y = 0:4
y =
0   1   2   3   4

z = x./y
z =
Inf  1.0000  0.5000  0.3333  0.2500
```

The first element of the new variable `z` is set to `Inf` because the first element of `y` is zero. Having one or more `Infs` when doing math does some strange things. Try adding up the values in the vector `z`:

```
sum(z)
ans =
 Inf
```

Adding, subtracting multiplying or dividing anything with one or more `Infs` in it will return a value of `Inf`. This can be maddening for data analysis because one little `Inf` in your data set will make all of your statistics turn to `Inf`. It therefore helps to be able to find your `Infs` and work around them. Matlab's `isinf` function helps with this:

```
isinf(z)
ans =
1  0  0  0  0
```

`isinf` returns a logical 1 if the value is `inf`, and zero otherwise. We can then use the output of `isinf` as an index. For example, if we want

to refer to the finite values of z we can do this:

```
z(~isinf(z))
ans =
1.0000 0.5000 0.3333 0.2500
```

To add up all the finite values in z, we can do this all in one line:

```
sum(z(~isinf(z)))
ans =
  2.0833
```

Notice how commands can almost be read like English: This line says "find the values of z that are not `inf` and sum them up". Be careful, of course, about throwing values away in your analysis just because they're `Inf`. An `Inf` in your data is probably a symptom of something horribly wrong.

A close relative to `Inf` is the value `NaN`, which stands for 'not a number'. Like `Inf`, math with a `NaN` in it will return a value of `NaN`.

Various calculations will return a value of `NaN`, but most are variants of trying to divide 0 by 0 which is undefined:

```
w=0/0
w =
 NaN
```

`NaN`'s are useful for missing data. Suppose you recorded 5 reaction times, but on the third trial the subject fell asleep. Your data vector might look something like this:

```
rt = [1.2, 1.1, NaN, 2.4, 1.5];
```

Calculating the mean of this vector will return a `NaN`:

```
mean(rt)
ans =
 NaN
```

To refer to the values of `rt` without the `NaN`'s you can use the function `isnan`, which acts just like `isinf`:

```
~isnan(rt)
ans =
  1  1  0  1  1
```

To calculate the mean of these values, you can do this:

```
meanRT = sum(rt(~isnan(rt))/sum(~isnan(rt)))
meanRT =
 1.5500
```

Instead of using `~isnan` or `~isinf` to find the finite values in a vector, you can use `isfinite`, which returns a logical 0 for values that are either `NaN` or `inf`.

```
isfinite(z)
ans =
  0  1  1  1  1

isfinite(rt)
ans =
  1  1  0  1  1

isfinite(rt+z)
ans =
  0  1  0  1  1
```

Questions for Chapter 4

Q 4.1: Making matrices

Create the following matrices without typing in the numbers by hand:

```
A =
   1   1   1   1   1
   1   1   1   1   1
   2   2   2   2   2
   2   2   2   2   2
   2   2   2   2   2

B =
   1   1   0   0   1
   1   1   0   0   1
   1   1   0   0   1
   1   1   0   0   1

C =
   1   1   1   1   1
   1   0   0   1   1
   1   0   0   1   1
   1   0   0   1   1
   1   1   1   1   1
   1   1   1   1   1

D =
   1   1   1   1   1
   2   2   2   2   2
   3   3   3   3   3
   4   4   4   4   4
   5   5   5   5   5
```

```
E =
   2   2   2   2   2   2
   1   2   2   2   2   2
   1   1   2   2   2   2
   1   1   1   2   2   2
   1   1   1   1   2   2
   1   1   1   1   1   2
```

(This one has a Hint if you get stuck, see Hints section at the end of the book.)

```
F =
   0   5  10  15  20
   0   5  10  15  20
   0   5  10  15  20
   0   5  10  15  20
   0   5  10  15  20
```

(This one has a Hint too.)

```
G =
   1   6  11  16  21
   2   7  12  17  22
   3   8  13  18  23
   4   9  14  19  24
   5  10  15  20  25

H =
   1   0   1   0   1   0   1   0
   1   0   1   0   1   0   1   0
   1   0   1   0   1   0   1   0
   1   0   1   0   1   0   1   0
   1   0   1   0   1   0   1   0
   1   0   1   0   1   0   1   0
   1   0   1   0   1   0   1   0
   1   0   1   0   1   0   1   0

I =
```

```
   1   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   1   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   1   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   1   0
   0   0   0   0   0   0   0   0

J =
   0   1   2   3   4
   1   2   3   4   5
   2   3   4   5   6
   3   4   5   6   7
   4   5   6   7   8

K =
   1   2   3   4   5
   2   4   6   8  10
   3   6   9  12  15
   4   8  12  16  20
   5  10  15  20  25

L =
   1   2   3   4   5
   6   7   8   9  10
  11  12  13  14  15
  16  17  18  19  20
  21  22  23  24  25
```

Q 4.2: 3d matrices

a) Create a 3x3x3 matrix of zeros and set the very middle of the 3-d matrix to be a value of 1.

b) Create a 5x5x5 matrix of zeros and set the middle 3x3x3 cube to 1.

Q 4.3: sub2ind and ind2sub

a) Suppose you have a 4x3 matrix (4 rows and 3 columns). What is the index into the element that is in the 3rd row and 2nd column?

b) For the same size matrix, what is the row and column for the element with an index of 7?

Q 4.4: logical operations

a) Write a script so that if a variable x is positive it prints the string
'x is positive'

And if x is negative the script prints the string
'x is negative'

b) Write a statement that is true if the variable x is either less than 2 or greater than pi.

c) Write a statement that is true if either x is greater than 2 and y is less than 4, or if z is equal to zero.

Q 4.5: while loops

Write a script that repeatedly rolls two dice using this command:

```
roll = ceil(rand(1,2));
```

and counts the number of rolls until [1,1] ('snake eyes') comes up.

CHAPTER 5: BASIC GRAPHICS

5.1 paintpots.m

This chapter will show you two simple ways to draw images on the screen. Create a new m-file called `PaintPots.m`

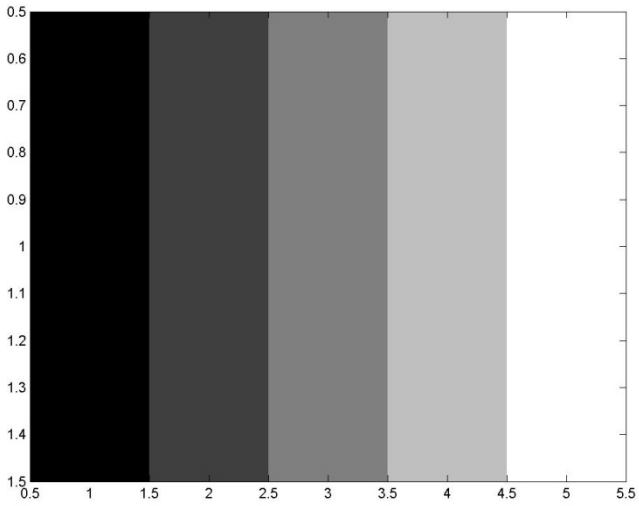
Begin with the following lines:

```
img = 1:5
img =
   1   2   3   4   5

paintpots1 = [0 0 0; 0.25 0.25 0.25; 0.5 0.5
0.5; 0.75 0.75 0.75; 1 1 1]
paintpots1 =
     0     0     0
  0.2500  0.2500  0.2500
  0.5000  0.5000  0.5000
  0.7500  0.7500  0.7500
  1.0000  1.0000  1.0000

paintpots2 = [0 0 1; 1 0 0; 0 1 0; 0.5 0 1;
1 0 1]
paintpots2 =
     0     0  1.0000
  1.0000     0     0
     0  1.0000     0
  0.5000     0  1.0000
  1.0000     0  1.0000
```

You've done this before; `img` is simply a vector of numbers going from 1 to 10. `paintpots1` and `paintpots2` are simply two matrices with five rows and three columns.

Now type the following into the program and run the whole thing.

```
figure(1)
image(img)
colormap(paintpots1);
```



The command `image(img)` opens a window (if one isn't already open) draws a picture of the matrix `img`. In this case, `img` is a list of 5 numbers that is displayed as a row of vertical bars that gradually change in grayness as the value along the x-axis changes from 1 to 5.

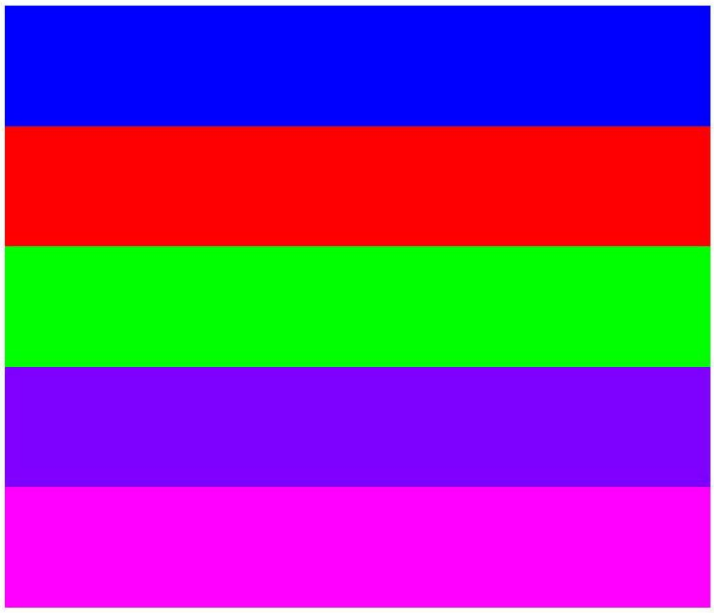
For the command `image` the rows go along the x-axis from left to right and the columns go along the y-axis from the top to the bottom. So try this:

```
image(img')
axis off
```

```
axis square
```

Now the bars will be horizontal and will increase in brightness as you go from the top to the bottom. The commands `axis off` and `axis square` get rid of the axes labels and make the image square.
Now try adding the following:

```
colormap(paintpots2)
```



Whoah! Why does the image suddenly change color?

Do you remember painting-by-numbers as a child? 1 meant pink, 2 meant orange and so on. `img` basically creates an image like the paint-by-numbers image – with indices that represent color (paint pot) 1, 2, 3, 4 and 5.

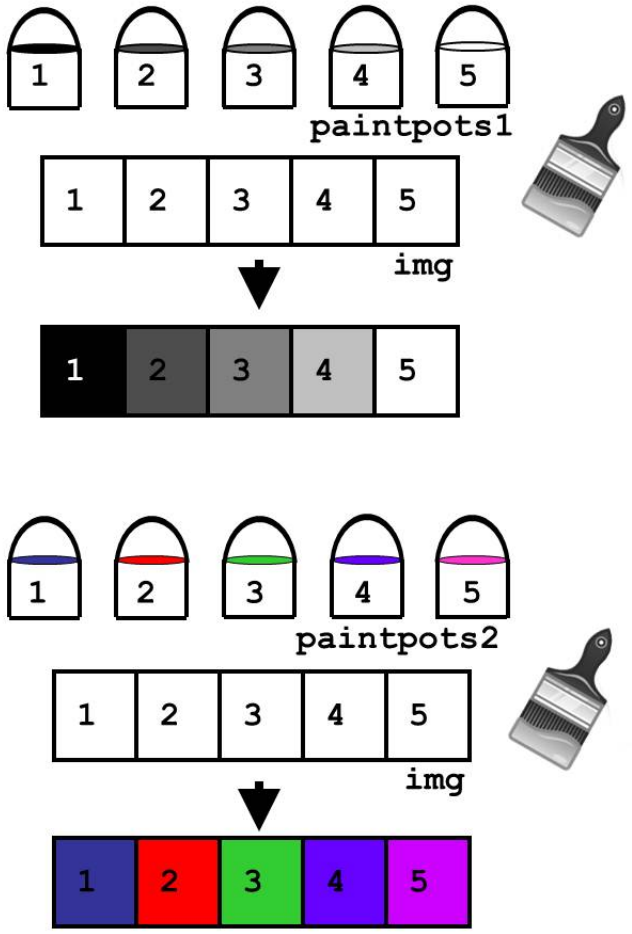


Figure 5.1 How colormaps work

`paintpots1` and `paintpots2` represent two different sets of paint pots.

In `paintpots1` and `paintpots2` the first column represents the output of the monitors' red gun, the second column represents the output of the green gun and the third column represents the output of the blue gun. You can think of each row as a separate paint pot. The collection of paintpots is known in Matlab as a colormap.

The command to tell Matlab to paint the image `img` using `paintpots1` was done using `colormap(paintpots1).` In `paintpots1` you can see that the output of each gun is always equal. Equal values of red green and blue make gray colors ranging from black [0,0,0] to white [1,1,1]. The five rows therefore represent different levels of gray going from black to white. So the image was grayscale

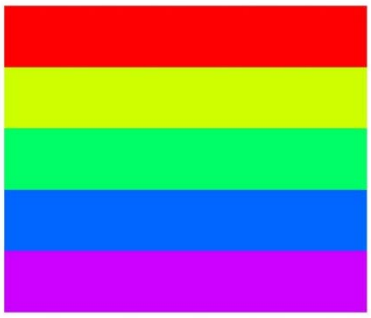
In `paintpots2` the first row contains the values [0 0 1]. So only the blue gun is used, and it has the maximum possible value. This means that the region of the `img` represented by 1 will be a saturated blue. The fourth row of paintpots2 contains [0.5 0 1]. This means that the red gun has half the maximum value and the blue gun has the maximum, giving you pinky-purple. So when you paint img using paintpots2 the values of 1-5 in the matrix now index a set of funky colors.

Basically a colormap is a mapping between a set of indices in an image (your paint-by-numbers picture) and a set of colors (the paint pots).

One cute thing about Matlab is that it has a lot of pre-made colormaps. Check out the following.

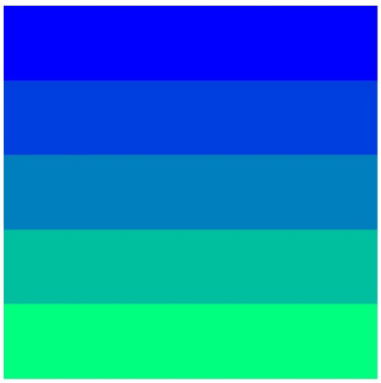
```
paintpots3=hot(5)
paintpots3 =
  1.0000     0     0
  1.0000  1.0000     0
  1.0000  1.0000  0.3333
  1.0000  1.0000  0.6667
  1.0000  1.0000  1.0000
colormap(paintpots3);
```

```
paintpots4=hsv(5)
paintpots4 =
  1.0000     0     0
  0.8000  1.0000     0
     0  1.0000  0.4000
     0  0.4000  1.0000
  0.8000     0  1.0000

colormap(paintpots4);
```



```
winter(5)
ans =
     0     0  1.0000
     0  0.2500  0.8750
     0  0.5000  0.7500
     0  0.7500  0.6250
     0  1.0000  0.5000
```

```
colormap(winter(5))
```



You can find a full list of Matlab colormaps using:

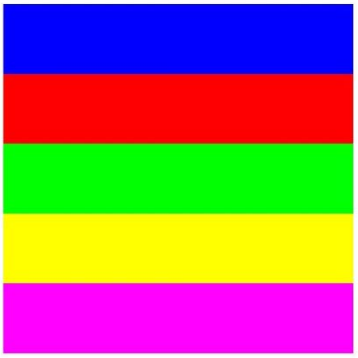
```
doc graph3d
```

Spend some time playing with colormaps. For example, see if you can change a single row of a colormap. For example, here we will replace purple with yellow in `paintpots2`.

```
paintpots2=[0 0 1; 1 0 0; 0 1 0; 0.5 0 1; 1
0 1];
paintpots2b=paintpots2;
paintpots2b(4, :)=[1 1 0]
paintpots2b =
   0   0   1
   1   0   0
   0   1   0
   1   1   0
   1   0   1
```

You should be able to guess what will happen if you use `paintpots2b` as your colormap.

```
colormap(paintpots2b);
```



5.2 UsingColormaps.m

OK, now we are going to make a little m-file which will allow you to gradually change an image by changing the colormap. Go to the menu bar and choose, **File-> New ->Script** and save it in your **LearningMatlab** folder as `UsingColormaps.m`.

```
% UsingColormaps.m
%
% a little program that manipulates colormaps
%
% written by IF and GMB 3/2005

clear all
close all
img=1:10;
figure(1)
paintpots = ones(10,3);
image(img);
colormap(paintpots)
axis off;

for i=1:10
 paintpots (i,:)= (i/10);
```

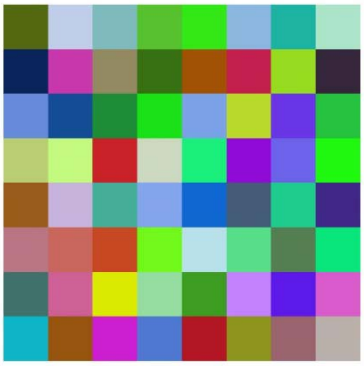
```
 colormap(paintpots);
 pause
 % The pause makes the program wait for a
 % key press so you can observe
 % each step.
end
```

As the program loops from `i = 1` to 10, it gradually replaces the 1's in `paintpots` with grays that go from 0.1 (very dark) to 1 (white). Then it replaces the old colormap with the new colormap, and gradually the white in the image is replaced by grays.

Now we are going to write a new program called `ExcitingColormaps.m`. You might want to just save a copy of `UsingColormaps.m` under a new name and modify it, since only a few lines will be changing.

```
% ExcitingColormaps.m
%
% another little program that manipulates
colormaps
%
% written by IF and GMB 3/2005


clear all
close all
```

```
img = reshape(1:64,8,8);
image(img);colormap(gray(64))
axis square % removes the axis labels
axis off % makes the image square
drawnow
pause

for i=1:200
 paintpots = rand(64,3);
 colormap(paintpots);
 drawnow
end
```



Was that was at least just a little bit exciting?

The command `reshape` allows you to reshape a vector or matrix into a different shape. The first argument is the vector or matrix that you want to reshape. In this case we gave reshape a vector that went from [1 2 3 …64]. The second argument is the number of rows we want the new matrix to have, and the second argument is the number of columns we want the new matrix to have. So in this case reshape turns a vector with 1 row and 64 columns into a matrix with 8 rows and 8 columns.

Of course this only works because 8x8=64. If we try to reshape a vector into a matrix of an inappropriate size we you would get an error.

```
img = reshape(1:64,8,7);

??? Error using ==> reshape
To RESHAPE the number of elements must not
change.
```

Here are some other examples of using reshape.

```
y1=reshape(1:8, 2, 4)
y1 =
   1   3   5   7
   2   4   6   8

y2=reshape(0:2:14,4, 2)
y2 =
   0   8
   2  10
   4  12
   6  14

v=reshape(y2, 2, 4)
v =
   0   4   8  12
   2   6  10  14
```

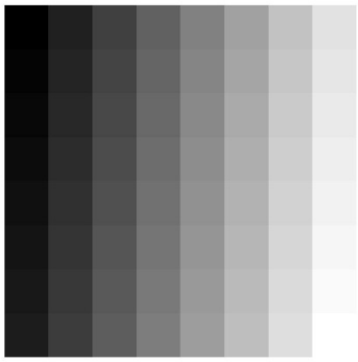
So, back to `ExcitingColormaps.m`. Take a look at `img`. You see that it creates a matrix where values gradually increase along each row and column.

```
img
img =
1   9  17  25  33  41  49  57
2  10  18  26  34  42  50  58
3  11  19  27  35  43  51  59
```

```
4   12   20   28   36   44   52   60
5   13   21   29   37   45   53   61
6   14   22   30   38   46   54   62
7   15   23   31   39   47   55   63
8   16   24   32   40   48   56   64
```

We can image `img` using a colormap that gradually fades from black to white. Notice how each column gets gradually lighter along the rows from top to bottom, as well as the image getting lighter across the columns.

```
image(img);colormap(gray(64)); axis off;
axis equal;
```



Now let's look more closely at the `for` loop above where i steps from 1, 2, 3 …200. If you type the command `rand`

```
rand
ans =
 0.6976
```

Matlab will give you a single random number. As well as getting a single random number you can also describe the size of the matrix of random numbers that you want, as is done in the program above. Try the following examples.

```
rand(3, 3)
ans =
  0.3558  0.3842  0.5911
  0.5556  0.7513  0.0717
  0.5310  0.6798  0.1502
```

Your random numbers will be different, because `rand` creates different random numbers every time it is called.

```
rand(2, 3)
ans =
  0.3214  0.7899  0.4002
  0.8732  0.6734  0.9538
```

```
paintpots = rand(64,3);
```

This line creates a matrix with 64 rows and 3 columns where the values are assigned randomly. The random numbers produced by rand vary between 0-1. So this gives us a colormap where the paintpots contain random colors.

We can look at the colors in the paintpots using code very similar to `UsingColormaps.m`. Note that the exact colors will be different from yours, because `rand` generates different random numbers every time it is called.

```
x=1:64;
image(x);
paintpots = rand(64,3);
colormap(paintpots)
axis off
```

You can also look at the actual values in `paintpots`, e.g.

```
paintpots(1:5,: )
ans =
  0.7819  0.4353  0.3267
  0.7389  0.0074  0.6113
  0.9209  0.9075  0.9158
  0.6931  0.0338  0.0096
  0.7140  0.3353  0.4582
```

Compare these values in `paintpots` to the first five stripes of the image.

```
x=1:5;
image(x);
colormap(paintpots)
axis off
```

Now run these lines of code again and do the same thing. The values in `paintpots` and the resulting images will change every time you run it.

Note that `img` never changes. It's always the same matrix, and it is a very orderly matrix, with values increasing along every row and every column. The way we get the crazy colors is by "painting" `img` using a colormap (a set of paint pots) that contain crazy random colors in no order. This set of crazy paint pots is replaced with a new set of crazy paintpots each time Matlab runs through the `for` loop. So we end up with a crazy pattern whose colors quickly change 200 times.

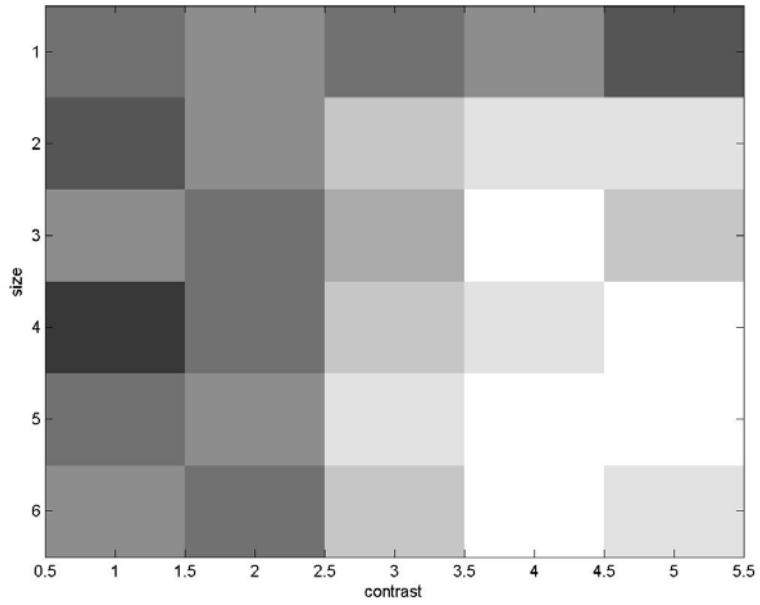
`drawnow` tells tells Matlab to update the figure. Normally updating figures have a pretty low priority for Matlab and will often be postponed in favor of other operations. `drawnow` tells Matlab to put other calculations on hold until the figure is updated.

5.2 Imaging 2D data

Suppose you ran an experiment where you varied both the size and the

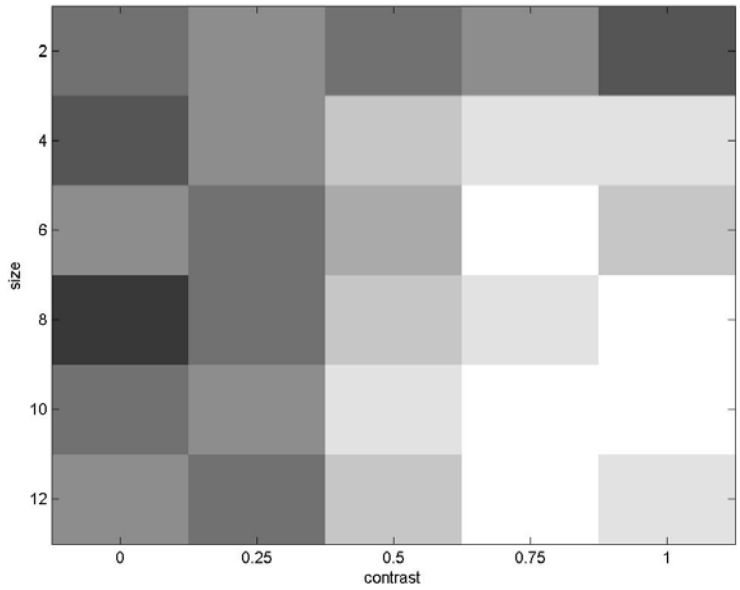
contrast (brightness) of a target and you measured the number of times (out of 10 trials) that the subject identified the stimulus correctly. You might end up with a matrix that looks like as follows:

```
contrast=linspace(0,1, 5); % 5 contrast
levels between 0 and 1
size=[2:2:12];% 6 sizes between 2 and 12
degrees of visual angle
pc=[5 6 5 6 4; % percent correct for each
size and contrast
  4 6 8 9 9;
  6 5 7 10 8;
  3 5 8 9 10;
  5 6 9 10 10
  6 5 8 10 9];
image(pc)
colormap(gray(10))
xlabel('contrast')
ylabel('size')
```



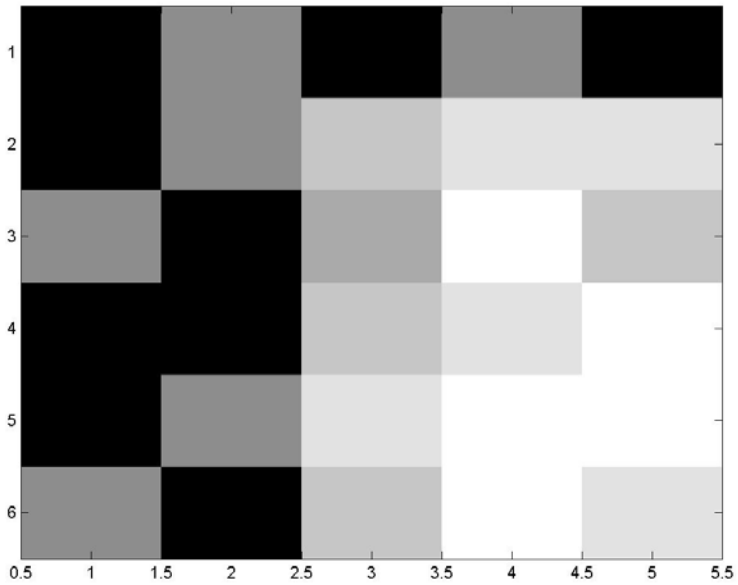
Now let’s add some cool stuff, we'll discuss this more in later chapters.

```
set(gca,'YTick', 1:6);
set(gca,'YTickLabel', size);
set(gca, 'XTick', 1:5)
set(gca, 'XTickLabel', contrast)
```



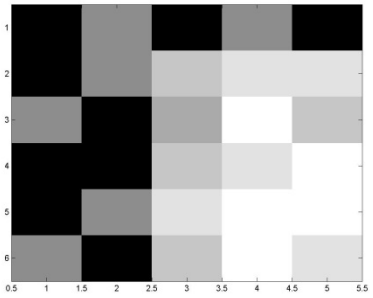
So, you know that chance performance is 50%, so you want to show any values at or lower than 50% as black. There are two ways of doing this. The first is by changing the data matrix.

```
pc_chance=pc;
pc_chance(pc_chance<=5)=0;
image(pc_chance);
colormap(gray(10));
```



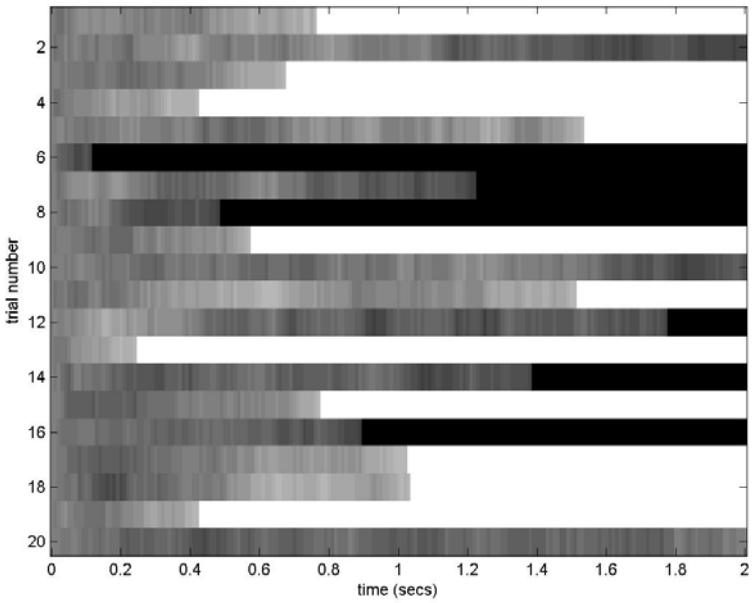
But a more conceptually elegant way is to change the colormap. The first 5 values of the colormap represents getting less than 50% correct, so we can just set those first 5 values to black. The result is the identical image, but this time it was done by changing the colormap instead of by changing the data matrix.

```
gmap=colormap(gray(10));
gmap(1:5, :)=0;
image(pc)
colormap(gmap)
```



5.2 Random rat

Here's another very similar example of using colormaps. Here we are faking some data. This is a very simple random walk model. Imagine on each trial a rat is internally gathering information about which direction to turn in a 2 armed maze.

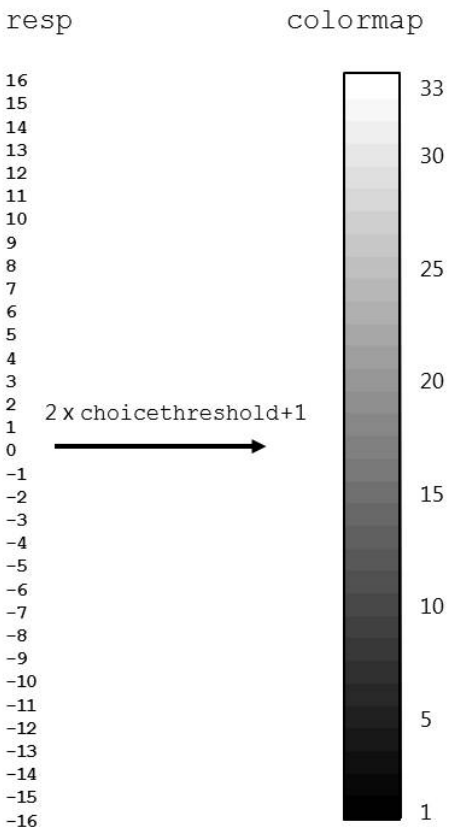


We're going to create a plot that represents the internal information of the rat that is determining whether s/he chooses the left or right arm of the maze. The x-axis represents time in seconds and the y-axis represents separate trials. Once the rat has chosen, the color goes black if the rat chose the wrong arm, and white if the rat chose the right arm.

```
% RandomRat.m
% A very simple example of a random walk model
%
% written IF 3/2013

ntrials=20; % number of trials
timepts=0:.01:2; % time
```

```
signal=.03*rand(ntrials, length(timepts));
% the internal signal that is added at each time
point
% note that the signal is small but always
positive. Because the
% amount of signal added at each time point is
independent of
% the signal at the previous time point/trial we
can fill this
% matrix outside of the loop even though
conceptually the signal
% is being added at each time point. It is better
to do it this
% way (even though it’s conceptually weird)
because % it’s
% better to keep as much as possible out of loops
for speed
% reasons.

noise=0.8*randn(ntrials, length(timepts));
% the noise added at each time point, much larger
and has a mean
% of zero

choicethreshold=8;
% the animal makes a choice when the internal
response reaches
% this number

for n=1:ntrials % go through each of the trials
 resp(n, 1)=0; % start with an internal response
of 0
 for t=2:length(timepts) %for each subsequent
moment in time
  resp(n, t)=resp(n, t-1)+signal(n,
t)+noise(n,t);
```

```
  % the response at that moment in time is the
same as the
  % previous time + the signal that was added to
the
  % internal representation at that time point +
some
  % random noise

  if resp(n, t)>=choicethreshold
   % if the response hits the positive threshold
   resp(n,t:length(timepts))=(2*choicethreshold);
   % set the response for the rest of that trial
to
   % 2x the choicethreshold
   t=length(timepts+1);
   % skip to the end of the trial,
   % so the loop moves to the next trial
  elseif resp(n, t)<=-choicethreshold
   resp(n,t:length(timepts))=-
(2*choicethreshold);
   t=length(timepts+1); % skip to the end
  end
 end % end of time loop
end % for each trial


cmap=gray((4*choicethreshold)+1);
colormap(cmap);
% see notes below
image(timepts, 1:ntrials,
resp+(2*choicethreshold)+1);
% this time using image we tell it the units for
the x and y axes.
ylabel('trial number')
xlabel('time')
```



So `resp` is a number that goes between +/- 2 x `choicethreshold`. We are going to make each integer value of `choicethreshold` a different level of gray. So if `choicethreshold` is 8 that means that we need a colormap that can represent -16:1:16 values – that's actually (4 x `choicethreshold`)+1 = 33 values. You can check this using `size(-16:16)` (Or by counting on your fingers.)

Remember that colormaps are indexed from 1 to how ever many paintpots there are (33).

So we need -2 x `choicethreshold` to map onto the first paintpot, and we need 2 x `choicethreshold` to map onto the 33rd paintpot and we need the value 0 to map onto the 17th paintpot. That's done by adding (2 x `choicethreshold`)+1 to resp.

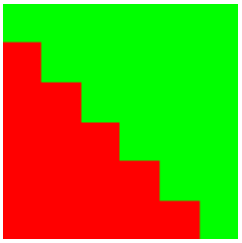
Questions for Chapter 5

Q 5.1 Images of matrices

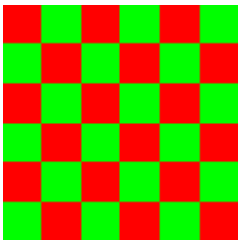
a) Use a nested `for` loop to create a matrix `M` that looks like this:

```
M= 2   2   2   2   2   2
   1   2   2   2   2   2
   1   1   2   2   2   2
   1   1   1   2   2   2
   1   1   1   1   2   2
   1   1   1   1   1   2
```

b) Create a color map with two colors, red and green. Use the `image` command with the matrix `M` and the `colormap` command with this new color map to generate an image that looks like this:



c) Create a new matrix `N` so that you get this image using the same colormap as in the example above.



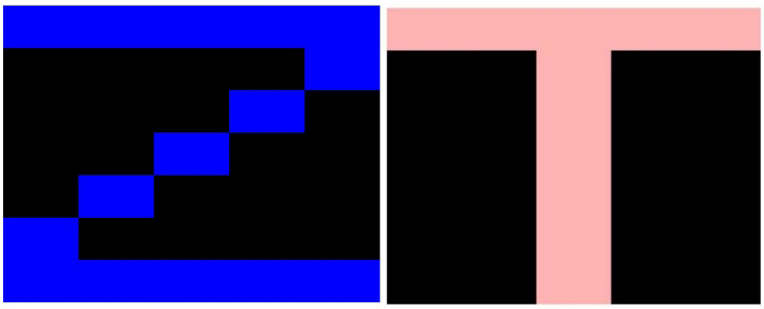
Q 5.2 Magic Letters

Starting with these two matrices:

```
Z=[ 1 1 1 1 1; ...
  0 0 0 0 1; ...
  0 0 0 1 0; ...
  0 0 1 0 0; ...
  0 1 0 0 0; ...
  1 0 0 0 0; ...
  1 1 1 1 1];

T=[ 1 1 1 1 1; ...
  0 0 1 0 0; ...
  0 0 1 0 0; ...
  0 0 1 0 0; ...
  0 0 1 0 0; ...
  0 0 1 0 0; ...
  0 0 1 0 0];
```

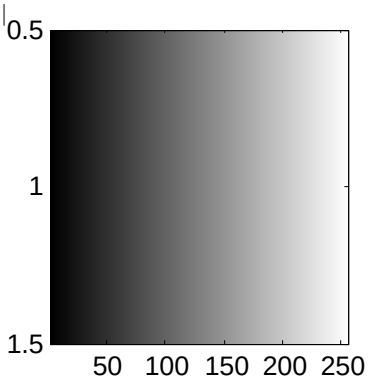
Combine Z and T to create a matrix ZT and create two colormaps (which will need to have 4 rows): `cmapZ` and `cmapT`, such that the following commands create the following two images.

```
image(ZT); axis off
colormap(cmapZ);
colormap(cmapT)
```

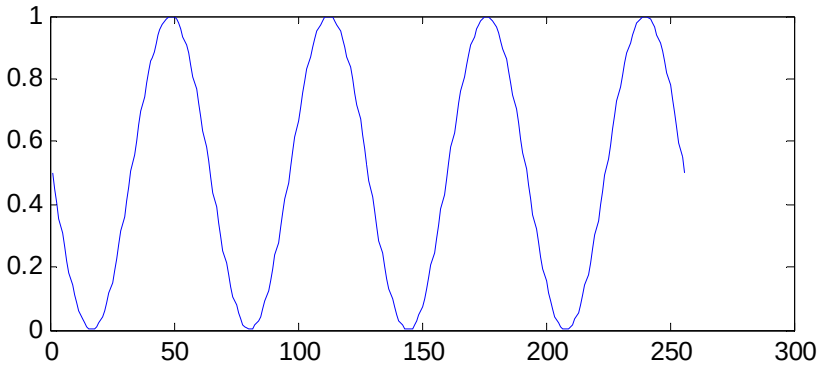


Q 5.3 Making a moving sinusoidal grating with a color map

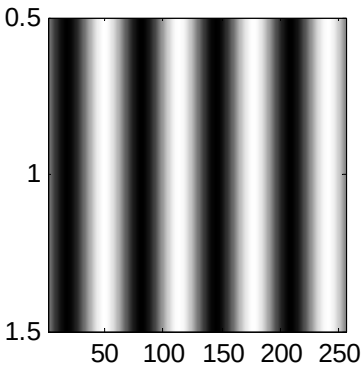
A moving sinusoidal grating is one of the classic stimuli of vision research. Just as a pure tone is a fundamental auditory stimulus, a moving grating is a fundamental stimulus for visual motion. This problem works through one way of making a moving grating.



a) Make a simple grayscale ramp by using `image` on a matrix `M=1:256` and `colormap(gray(256))`. It should look like this:b) Make a new color map of size 256x3 with each of the three columns (r, g and b) modulating sinusoidally from 0 to 1 for four cycles with a phase of pi. (If you're rusty on your trigonometry, see the Hints section). A plot of each



column of the color map should look like this:
Apply this color map to the ramp image. You should get this: A sinusoidal grating! Think about why this happens using the 'paint pots' analogy.



c) Make the grating move or 'drift' rightward by changing the phase in a loop, resetting the color map and using the `drawnow` command. You can make the grating drift through 4 cycles over 100 frames by setting the phase with a loop like this:

```
for phase = linspace(0,8*pi,100)
    .
    .
    .
end
```

Q 5.4 Altering the rat random walk model

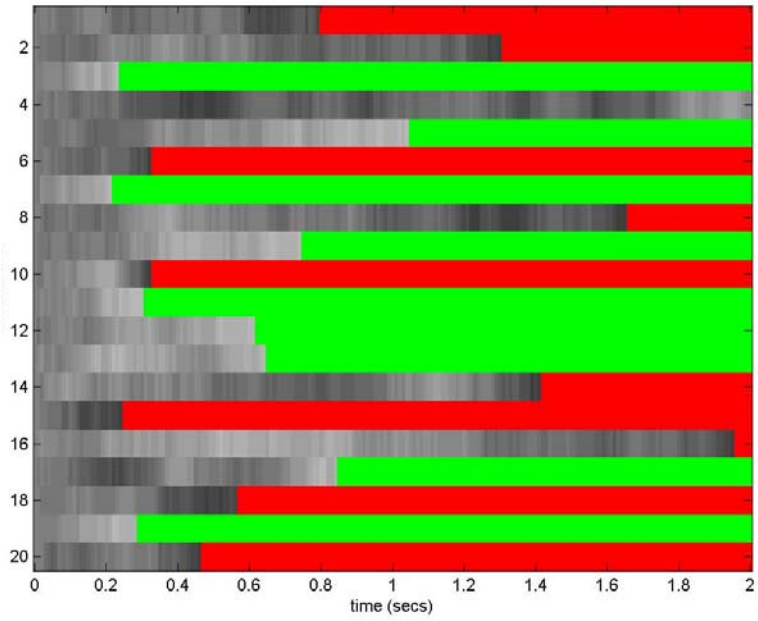
Alter the rat random walk model to do the following

a) The rat makes the decision more quickly. (See Hints if you get stuck.)

b) The rewarded arm of the maze alternates from trial to trial.

c) When the rat chooses one arm of the maze the image is green and when s/he chooses the other the image is green. (See Hints if you get stuck.)

The following image incorporates all these changes.


2
4
6
8
10
12
14
16
18
20
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
time (secs)

CHAPTER 6 – MORE LOGICAL OPERATIONS AND SOME FUNKY VISUAL STIMULI

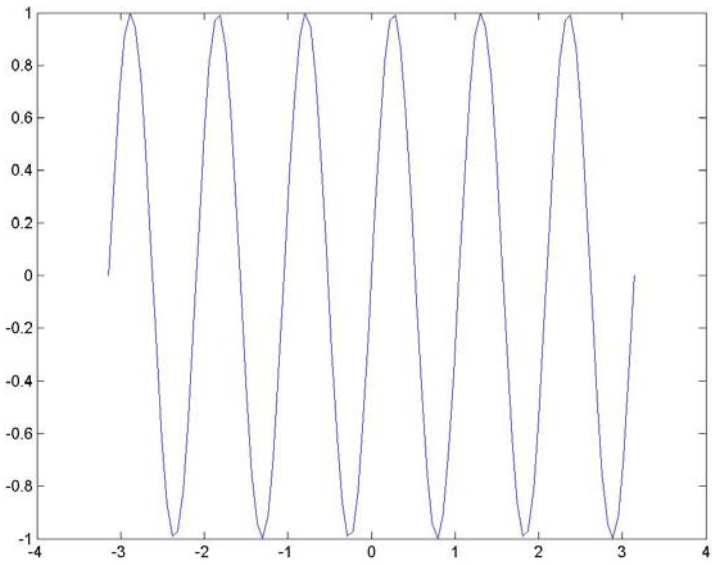
You have actually now learned the basic commands you need to program 90% of most experiments. Here we are going to use some of the commands we have learned to make some funky visual stimuli. This chapter is mostly revision, showing you how the commands you have learned in the previous chapter can be used. We will begin by making a sinusoidal grating.

Note that in this chapter we will make this sinusoidal grating in a different way than was done in chapter question 5.2. For the chapter question we used a linear ramp as the image and a sinusoid as a color map. Here we'll do the opposite – we'll use a sinusoidal image and a grayscale ramp for a color map.

6.1 SineInAperture.m

Create an m-file called `SineInAperture.m`

```
clear all
close all
x=linspace(-pi, pi, 100);
sf=6; % spatial freq in cycles per image
sinewave=sin(x*sf);
plot(x, sinewave);
```



This gives us a 1-dimensional sinusoidal grating with a frequency of 6 cycles in the distance between `-pi` and `pi`. As we described earlier, `pi` is a *reserved variable* in Matlab which means that it represents the number `3.1416` without you having to define it as such. Other reserved variables are `i` and `j`, which both represent the `sqrt(-1)` weirdly enough (see box below). To create a 2D version of this sinusoid we will calculate the outer product of the one-dimensional sinusoid with a vector of ones.

`i` and `j` are weird …

Compare these two errors. In both cases the bug is due to the same mistake on your part – you're doing a loop on `j`, but you've used another variable (`t` or `i`) by mistake within the loop. But Matlab gives you completely different error messages. Why?

```
clear all
vect=20:30
for j=1:10
 disp(vect(t))
end
Undefined function or variable 't'.
```

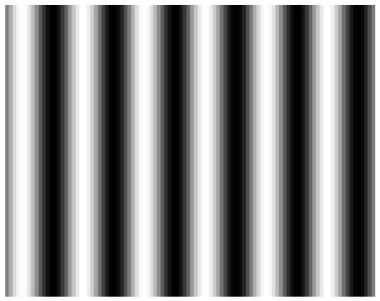
Matlab knows that you are trying to use a variable `t` that hasn't been defined yet. This message points you to the problem pretty clearly.

```
clear all
vect=20:30
for j=1:10
 disp(vect(i))
end
Subscript indices must either be real positive integers or
logicals.
```

The reason you get a different error is because `i` is a 'reserved variable whos default value is `sqrt(-1)`. The variable is defined, but it isn't a sensible index into a vector, and so Matlab spits out a different error message. You can imagine that it might be a little harder to realize that this error is actually due to using the wrong variable name.

The command **close all** closes all the figure windows.

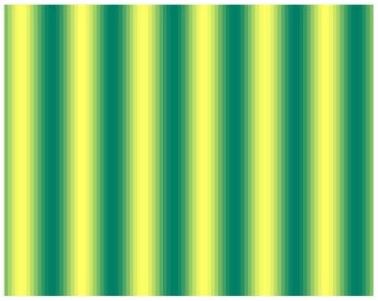
```
close all
onematrix=ones(size(sinewave));
sinewave2D=(onematrix'*sinewave);
colormap(gray)
imagesc(sinewave2D)
axis off;
```



Gratings like these are commonly used to study the visual system, partly because they are good stimuli for stimulating cells at early stages of the visual system.

You can also look at what this sinusoid looks like with a different kind of colormap.

```
colormap(summer);
```



Go ahead and try creating a horizontal rather than a vertical grating.

6.2 Scaling images

The variable `sinewave2D` varies between -1 and 1. The command `imagesc` scales the values in `sinewave2D` before plotting so as to match the colormap you are using. Default color maps have 64 rows, so `imagesc` rescales `sinewave2D` to range between 1 and 64.

If we want, we can scale the matrix ourselves and use `image` instead of using `imagesc`. One advantage of this is that we don't automatically use the entire contrast range of the colormap. So we can create a grating that is low contrast (modulates between shades of gray) instead of one that varies between black and white.

On normal monitors, we can use up to 256 different colormap values, as is described in more detail later. So to get maximum chromatic resolution we would probably want to use a map with 256 values, and rescale `sinewave2D` so it varies between 1 and 256.

```
close all
scaled_sinewave2D=((sinewave2D+1)*127.5)+1;
```

Those parts of `sinewave2D` that equal -1 will be scaled to equal 1.

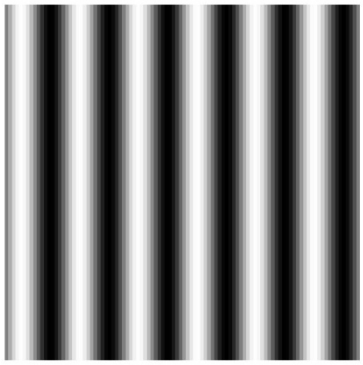
```
((-1+1)*127.5)+1
ans =
  1
```

Those parts of `sinewave2D` that equal 0 will become 128.5

```
((0+1)*127.5)+1
ans =
 128.5000
```

And those parts of `sinewave2D` that equal 1 will become 256

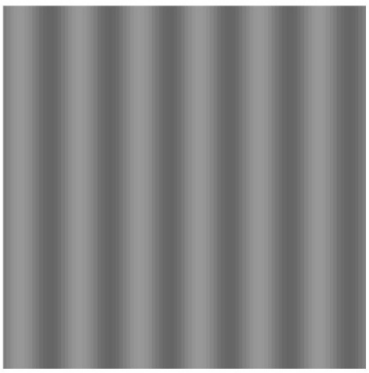
```
((1+1)*127.5)+1
ans =
  256

image(scaled_sinewave2D)
% rescales numbers between -1 and 1 to lie
between 1 and 256
colormap(gray(256))
axis equal
axis off
```



This image looks identical, but this time we are scaling it ourselves rather than letting Matlab do it for us. Now let's make the image low contrast. If we wanted the sinewave to be 1/5 of the full contrast range we would do the following. (Here we're assuming that we have a linear monitor which is almost certainly false, see Brainard, D.H. (1989). Calibration of a computer controlled color monitor. Color Research and Application, 14, 23-34 or hopefully a later kindle book of ours for details on monitor calibration.)

```
contrast =1/5;
scaled_sinewave2D=(((contrast.*sinewave2D)+1
)*127.5)+1;
image(scaled_sinewave2D)
% rescales numbers between -1 and 1 to lie
```

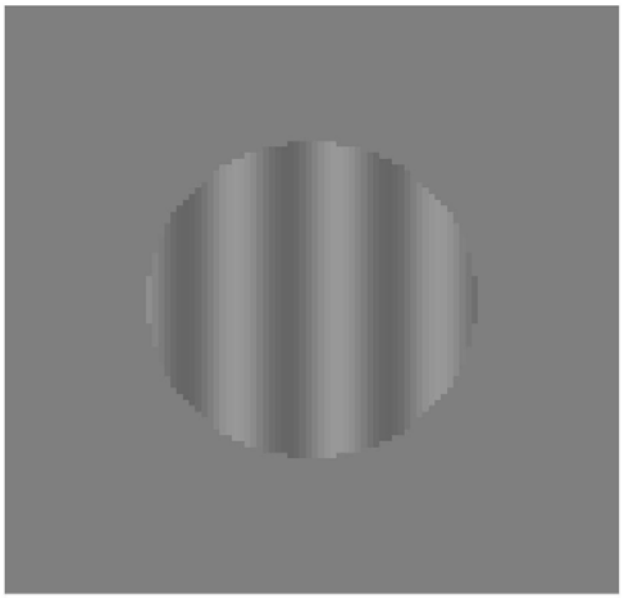
```
between 1 and 256
colormap(gray(256))
axis equal
axis off
```



6.3 Apertures and tiling

Now let's window this 2D sinusoid inside a circular aperture.

```
rad=1.7;
for r=1:length(x)
 for c=1:length(x)
  if x(r)^2+x(c)^2>rad^2
    sinewave2D(r, c)=0;
  end
 end
end
scaled_sinewave2D=(((contrast.*sinewave2D)+1
)*127.5)+1;
image(scaled_sinewave2D);
colormap(gray(256))
axis off;
```



What's happening in this loop is that we are marching our way through every row and column in `scaled_sinewave2D`. The line `x(r).^2+x(c).^2)>rad^2` is based on the equation to calculate the radius of a circle. For every point in `sinewave2D` we are calculating the distance to the center of the matrix to see whether that distance is greater than the radius of the desired aperture. If it is, we make that point in the image zero, which ends up mid-gray on our color map.

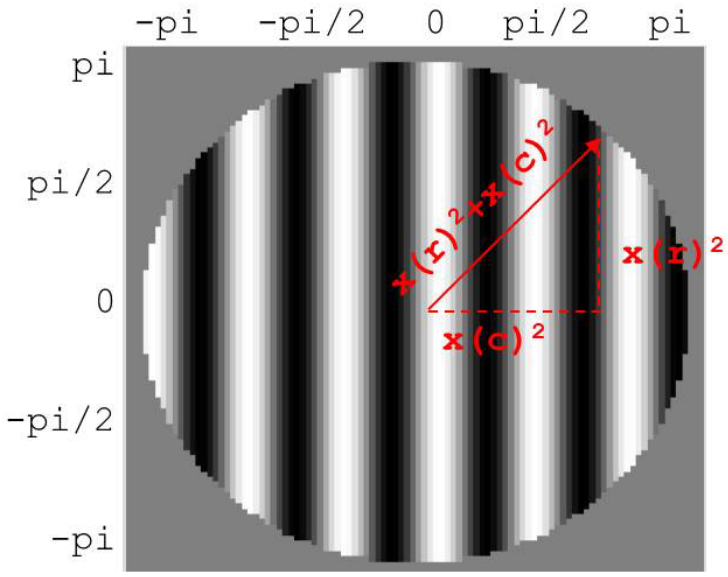


Figure 6.1 Creating a circular aperture.

Now let's create a glorious tiling of these windowed sinusoids.

```
imgsize=length(scaled_sinewave2D);
ntiles=2;
sep=50;
```

`sinewave2D` is 100 rows by 100 columns (`imgsize`). Let's suppose we wanted to create an image of 2 x 2 of these apertures (`ntiles`), with a 50 pixel separation between them (`sep`).

```
tilesize=(ntiles*(imgsize+sep))+sep;
tilematrix=zeros(tilesize);
```

The entire tile matrix will therefore need to be 350 pixels by 350 pixels (`tilesize`). We begin by creating a matrix that size (`tilematrix`) which is filled with zeros.

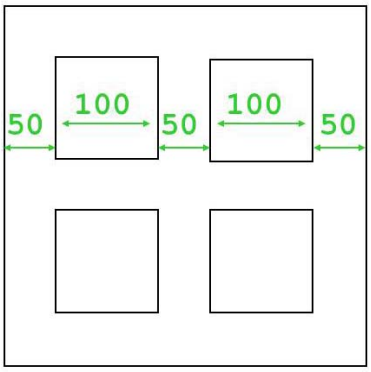


Figure 6.2 Creating a tiling.

```
startpos=sep:sep+imgsiz
e:length(tilematrix)-1;
```

We will then insert the apertures in four places: with the top left corner in the 50th row 50th column, 50th row 200th column, 200th row 50th column and 200th row 200th column. Because the matrix is symmetric along the rows and columns these locations can be specified with a single vector (`startpos`). We're going to put in the original `sinewave2D` matrix that varies between -1 and 1, and scale it later.

```
for rtile=1:ntiles
 for ctile=1:ntiles
  tilematrix(startpos(rtile):startpos(rtile)
+ …
  imgsize-1,startpos(ctile):startpos(ctile)+
…
  imgsize-1)=sinewave2D;
 end
end
```

We then go through the rows and the columns inserting `sinewave2D` into `tilematrix`. Let's think about what happens when `r=1` and `c=1`. We will insert `sinewave2D` into the following position in `tilematrix`:

```
tilematrix(startpos(1):startpos(1)+imgsize-
1, startpos(1):startpos(1)+imgsize-1)
```

Looking at this more closely you can see that this is the equivalent of

```
tilematrix(50:149,50:149)
```

```
startpos(1)
ans =
 50
```

```
startpos(1)+imgsize-1
ans =
 149
```

If you look at the length of 50:149 you will see that you have assigned a space that has 100 rows and 100 columns as being the place in `tilematrix` that should be overwritten by `sinewave2D.`

```
length(50:149)
ans =
 100
```

If you take out the -1, you will get an error since you are trying to put a matrix that has 100 rows and 100 columns into a space that has 101 rows and 101 columns.

```
r=1; c=1;
tilematrix(startpos(rtile):startpos(rtile)+imgsize,
startpos(ctile):startpos(ctile)+imgsize)=sinewave2D;
??? Subscripted assignment dimension mismatch.
```

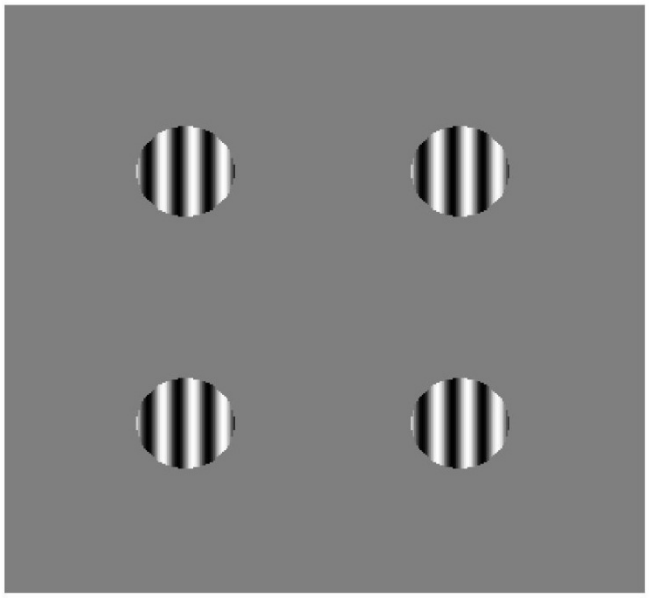
Similarly, you can't squeeze `sinewave2D` into a space with less than 100 rows and columns.

```
tilematrix(1:50, 50:149)=sinewave2D;
```

```
??? Subscripted assignment dimension
mismatch.
```

Let's look at this final tilematrix:

```
scaled_tilematrix=((tilematrix+1)*127.5)+1;
image(scaled_tilematrix);
colormap(gray(256))
axis off;
```

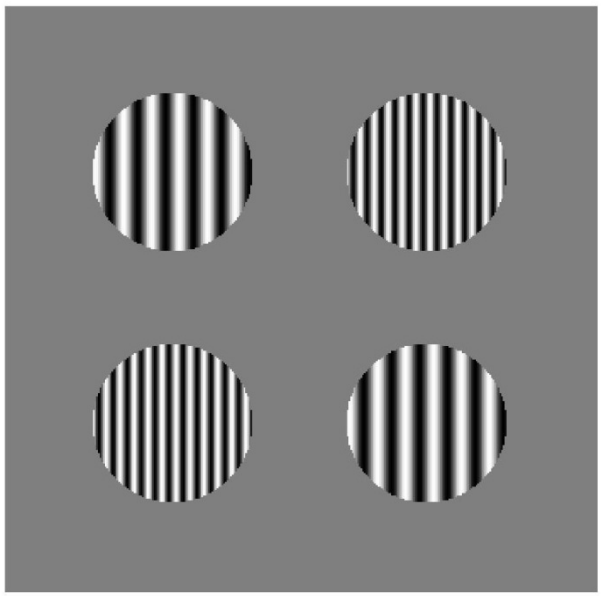


Finally, let's create a version where the apertures vary in spatial frequency.

```
% SineInApertureFreq.m
%
% tiled gratings that vary in spatial frequency
clear all
close all

% information about the sinusoids
x=linspace(-pi, pi, 100);
```

```
sf=[6 12]; % spatial freq in cycles per image
rad=3; % radius of the aperture

% creates a 100x100x2 matrix containing two 2D
sinusoids
% in apertures, of different spatial frequencies.
for s=1:length(sf)
 sinewave=sin(x*sf(s));
 onematrix=ones(size(sinewave));
 sinewave2D(:, :, s)=(onematrix'*sinewave);
 for r=1:length(x)
  for c=1:length(x)
   if x(r).^2+x(c)^2>rad^2
    sinewave2D(r, c, s)=0;
   end
  end
 end
end

% initialize the tilematrix by filling it with
zeros
imgsize=length(sinewave);
ntiles=2; % number of tiles in each direction
sep=50; % separation between the tiles
matrixsize=(ntiles*(imgsize+sep))+sep;
tilematrix=zeros(matrixsize);
startpos=sep:sep+imgsize:length(tilematrix)-1;

% do the tiling
for rtile=1:ntiles
 for ctile=1:ntiles
  if rtile==ctile
% if the 1st tile along both the row and column
% direction,
% or the 2nd tile along both the row and the
column
% direction
```

```
% then use the first spatial frequency (6 cycles
per image)
   tilematrix(startpos(rtile): …
   startpos(rtile)+imgsize-1, …
   startpos(ctile):startpos(ctile)+imgsize-1)= …
   sinewave2D(:, :, 1);
  else
   % otherwise use the second spatial frequency
(12
            % cycles per image)
   tilematrix(startpos(rtile): …
   startpos(rtile)+imgsize-1, …
   startpos(ctile):startpos(ctile)+imgsize-1)=…
   sinewave2D(:, :, 2);
  end
 end
end

% image the final tilematrix
scaled_tilematrix=((tilematrix+1)*127.5)+1;
image(scaled_tilematrix);
colormap(gray(256))
axis off;
```

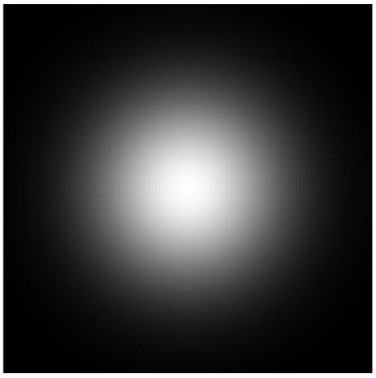


6.4 Gaussian windows

We can create a more sophisticated image using `for` loop to determine the value of a pixel rather than simply turning it into a 0. For example, if we want to calculate an image of a 2-dimensional Gaussian, we can define the values in the following nested loop:

```
% Gaussian1.m
%
% Method 1 of windowing a sinusoid in a Gaussian
% using a nested for loop


n = 101; % size of the nxn image of a Gaussian
Gaussian = zeros(n); % define a n x n matrix of
zeros
x = linspace(-2,2,n); % define the values for x
and y axes
y = linspace(-2,2,n);
for j=1:n % columns
```

```
 for i=1:n % rows
  % as i and j range from 1 to n,
  % x(i) and y(j) will range from -2 to 2.
  % We can then do math on x(i) and y(j) for each
pixel:
  Gaussian(i,j) = exp(-x(i).^2-y(j).^2);
 end
end

% Do the usual graphics stuff to show the image
figure(1)
clf
image((255*Gaussian)+1);
colormap(gray(256));
axis equal
axis off
```



As the variables `i` and `j` range from 1 to 101, `x(j)` and `y(i)` range from -2 to 2. When `i=1` and `j=1`, which is the top left of the image, `x(j)` and `y(i)` are both equal to one. So in this case `exp(-y(i).^2-x(j).^2) = 0.1353.`

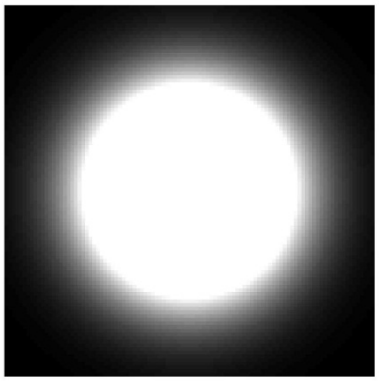
When `i=51` and `j=51`, which is the center of the image, `x(j)` and `y(i)` are both equal to zero. So in this case `exp(-y(i).^2-x(j).^2) = 1.`

So `Gaussian` will have a maximum value of 1 at the center, and drop off from there.

We've defined the variable `n` to be the size of the image. We could have typed in the number 101 wherever `n` appears. But by defining `n` as a variable instead using the number 101, we can easily change the resolution of our picture. Try changing `n` to 10 and seeing what the picture looks like. Defining numbers as variables is called *soft coding* (instead of using the actual value, which is called *hard coding*). Soft coding is convenient because it allows you easily change the parameters of a program by only changing a single line. It's also easier to read provided you give your variables sensible names. A good rule is that if you use the same number more than once in your program, it's a good idea to use soft coding and replace that number with a variable (actually it's usually better to use a variable even if you only use a variable once, just so the variable name makes it clear what the number represents. While you are a beginner it's sometimes easier to begin a program using hard coding, and then replace the hard coding with soft coding.

Because `Gaussian` goes between 0 and 1, when rescaling `gaussian` for `image` we need to multiply by 255 (to scale scale the image between 0 and 255) and then add 1 (so that the image goes between 1 and 256 to match the colormap).

Now try running the program again, but this time use a colormap containing 64 values: `colormap(gray(64));`



What's happening here is that every value in `Gaussian` that is greater than 64 is being assigned to the highest value of the colormap, which is white.

```
cmap=colormap(gray(64));
cmap(end, :)
```

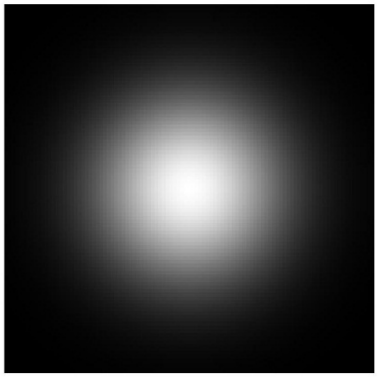
One of the peculiarities of Matlab (as compared to languages like C or Fortran) is that `for` loops are actually quite slow and should be avoided when possible. *Nested for* loops (one for loop inside another) end up being particularly slow – in the example above Matlab actually has to iterate around the second for loop 101 x 101 = 10201 times.

6.5 meshgrid

Luckily in this case it's easy to replace the nested for loop with a very cool command called `meshgrid`.

```
% Gaussian2.m
%
% A better method of windowing a sinusoid in
a
% Gaussian using meshgrid

n = 101;
```

```
[X,Y] = meshgrid(linspace(-2,2,n));
Gaussian = exp(-X.^2-Y.^2);
image((255*Gaussian)+1);
colormap(gray(256));
axis equal
axis off
```



All this in a seven line program! Notice that there aren't any `for` loops either. The trick is the `meshgrid` command that takes in the vector ranging from -2 to 2, and returns two matrices which we've called `X` and `Y`.

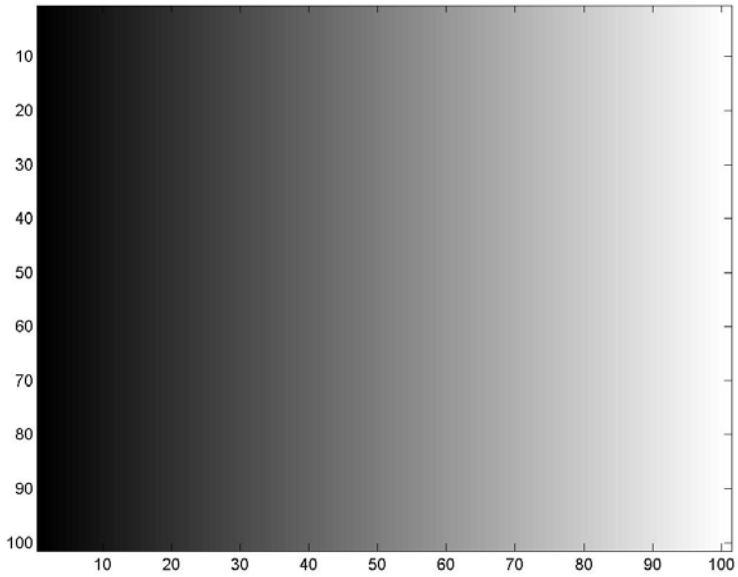
Let's take a look at sizes of these two matrices **`X`** and **`Y`**.

```
size(X)
ans =
 101 101
```

```
size(Y)
ans =
 101 101
```

They're both 101 by 101 matrices, where 101 is the length of the vector that was sent into `meshgrid`. Let's look more closely at these matrices.

```
figure(1); clf
imagesc(X);
colormap(gray(256))
```



`clf` clears the current figure window. You could also close it. The advantage of clearing it is that if you've moved the figure window to a convenient place in the monitor, it will stay in that place.

The matrix X goes from dark on the left to white on the right, with each column containing the same value. We can't see the actual values in this image, so let's let look at these values directly.

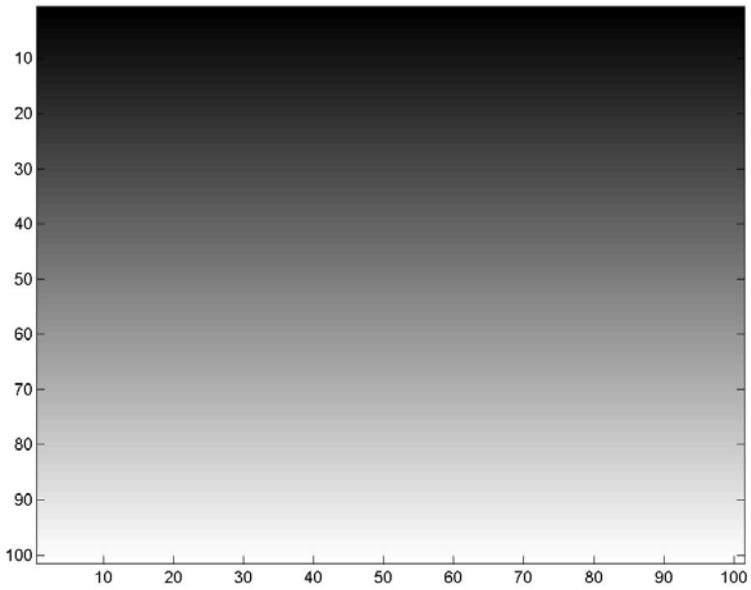
```
X(1, 1)
ans =
 -2

X(101, 1)
ans =
 -2
```

The first column of X is always -2.

```
X(1, 101)
ans =
   2


X(1, 101)
ans =
   2
```

The last column of X is always 2. X contains columns that range from -2 to 2 as we move from left to right. Now let's do the same for the matrix **Y**.

```
figure(2); clf
imagesc(Y);
colormap(gray(256))
```




```
Y(1, [1 25 50 75 101])
ans =
   -2   -2   -2   -2   -2
```

```
Y(101, [1 25 50 75 101])
ans =
   2   2   2   2   2

Y([1 25 50 75 101], 1)
ans =
  -2.0000
  -1.0400
  -0.0400
   0.9600
   2.0000

Y([1 25 50 75 101], 101)
ans =
  -2.0000
  -1.0400
  -0.0400
   0.9600
   2.0000
```

Each row in the matrix Y has the same value and the rows range in value from -2 to 2 as we move from top to bottom.

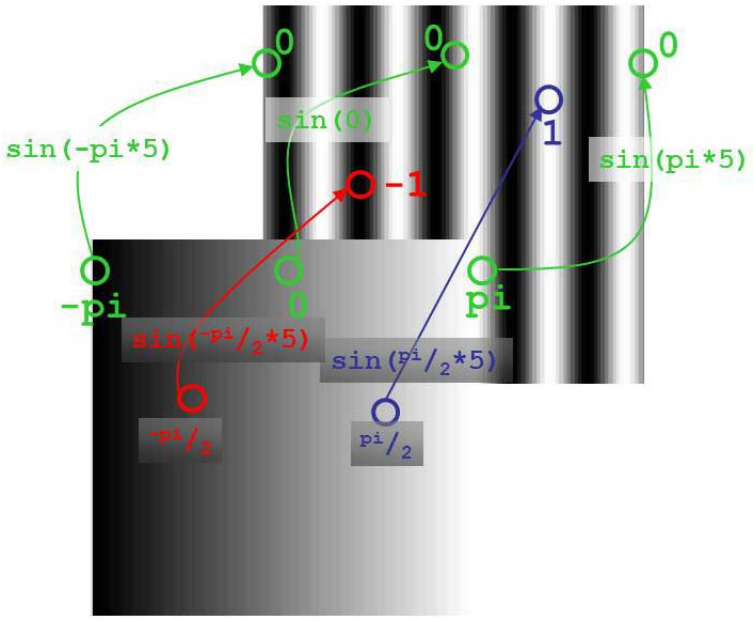


Figure 6.3 Using `meshgrid` to create a 2D sinusoid.

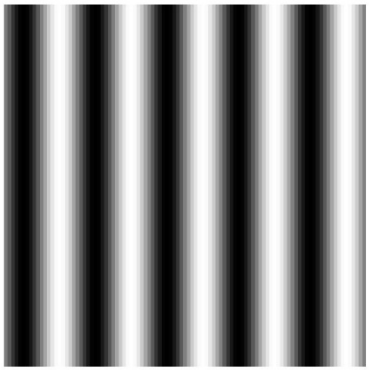
If you think about for a bit, you'll realize that these matrices X and Y are very useful for making mathematically defined images because we can now just do the math directly on these matrices without nested `for` loops. The values in these matrices contain the values that would be obtained by the following `for` loop:

```
for j=1:101 % columns
 for i=1:101 % rows
```

For example, here's a way of making a sinusoid using the X matrix of `meshgrid`.

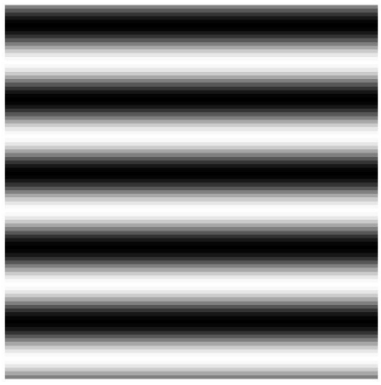
```
clear all; close all;
n = 101;
[X,Y] = meshgrid(linspace(-pi,pi,n));
sinewave2D = sin(5*X);
figure(1)
imagesc(sinewave2D)
```

```
axis equal; axis off; colormap(gray(256));
```



Each element of the matrix `sinewave2D` is defined as the sin of 5 times each element of the matrix X. This makes 5 cycles in the horizontal direction (see figure 6.3). We can make a horizontally oriented grating using Y:

```
horizontalGrating = sin(5*Y);
imagesc(horizontalGrating)
axis equal;axis off;colormap(gray(256));
```



Finally, you can make a Gaussian as an element-by-element function on the matrices X and X:

```
Gaussian = exp(-X.^2-Y.^2);
```

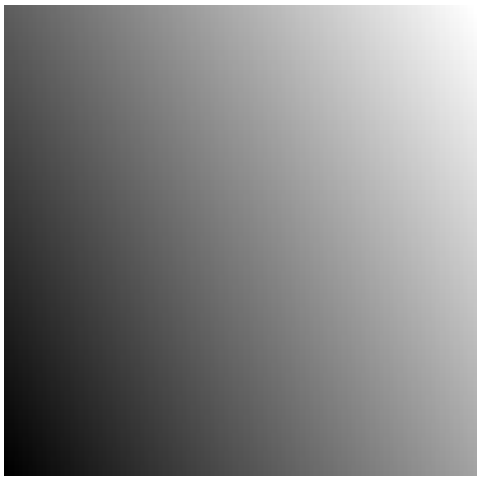
We can also use both X and Y to make a grating of any orientation. This is done by first combining X and Y to make gradient in a new direction.

For example, for a grating oriented 30 degrees off horizontal, we first make a new matrix that's a gradient that increases along this direction.

Any combination of the X and Y matrices produces a third matrix that is also a gradient. The combination that makes a gradient in our desired direction takes some trig:

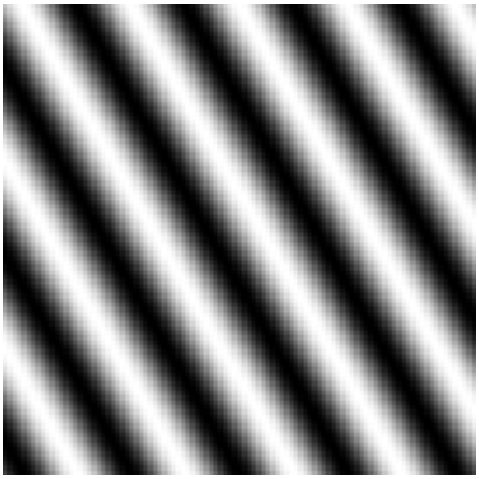
```
angle = 30; %degrees
ramp = cos(angle*pi/180)*X - sin(angle*pi/180)*Y;
imagesc(ramp)
axis equal;axis off;colormap(gray(256));
```

See how it increases from dark (low) to light (high) at an angle of 30 degrees counterclockwise from horizontal. The matrix `ramp` is the sum of the X and Y matrices where the X matrix is weighted by the cosine of the angle, and the Y matrix is weighted by the (negative) sin of the angle. `pi/180` converts from degrees to radians (degrees go from 0 to



360, radians to from 0 to pi*2).
Now all we have to do is take the `sin` of `ramp`:

```
orientedGrating = sin(5*ramp);
imagesc(orientedGrating);
axis equal; axis off; colormap(gray(256));
```



The `ramp` matrix is in the same units as the `X` and `Y` matrices, so multiplying ramp by 5 gives the same spatial frequency as before (5 cycles per image).

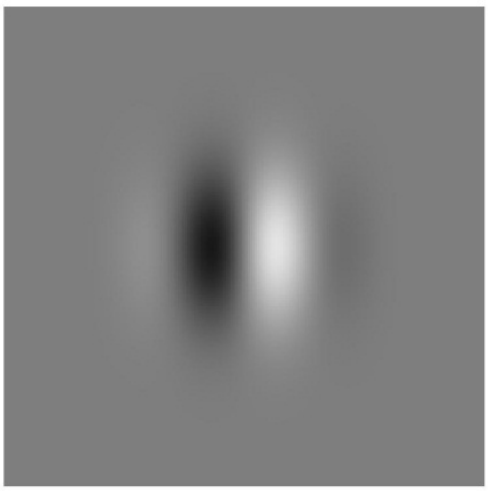
6.6 Gabors

Now that we've made a sinusoidal grating and a Gaussian, we're ready to write a program `MakeGabor.m` that creates a Gabor by multiplying a sinusoidal grating with a Gaussian. Gabors are one of the fundamental stimuli for vision research because they are a good model for a V1 receptive field.

We've added two new parameters: the parameter `width` determines how wide the Gaussian is, and the parameter `spatialFrequency` determines how many cycles the sinusoid makes across the whole image. Notice how the `meshgrid` command makes `X` and `Y` matrices that have values that range from `-pi` to `pi`. This way `sin(spatialFrequency*x)` makes a grating that has a number of cycles equal to the parameter `spatialFrequency`.

```
% MakeGabor.m
%
% Creates a grating windowed by a Gaussian.
```

```
clear all;
close all;

% define the Gabor's parameters:
n = 201; %resolution of the image
width = 1; %1/e half width of Gaussian
spatialFrequency = 3;
%spatial frequency of the sinewave carrier
(cycles/image)

% Use meshgrid to create matrices X and Y that
range from -pi to pi;
[X,Y] = meshgrid(linspace(-pi,pi,n));

% Sinusoid on the matrix X makes a vertical
grating.
sinusoid = sin(spatialFrequency*X);

% Gaussian aperture:
Gaussian = exp( -(X.^2+Y.^2)/width^2);

% A Gabor is the product of the sinusoid and the
Gaussian
Gabor = sinusoid.*Gaussian;

% Show the image using the usual commands:
figure(1)
image( (Gabor+1)*127.5); %scale Gabor between 0
and 255
axis equal
axis off
colormap(gray(256))
```



The following program, `OrientedGabor.m`, makes a Gabor stimulus with full flexibility over where the Gabor is centered in the image, as well as size, contrast, spatial frequency, orientation and phase. Play with the parameters defined at the beginning of the program to see what each one does.

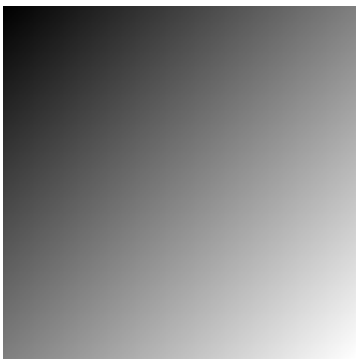
```
% OrientedGabor.m
%
% Creates a grating windowed by a Gaussian where
the following
% parameters can vary:
% size
% contrast
% spatial frequency
% orientation
% phase


% define the Gabor's parameters:


center = [1,1];
%[0,0] is the middle of the image, [pi,pi] is the
lower right
orientation = pi/4; %radians (pi/4 = 45 degrees)
```

```
width = .5; %1/e half width of Gaussian
spatialFrequency = 6; %spatial frequency of
Sinewave carrier (cycles/image)
phase = -pi/2; %spatial phase of sinewave carrier
(radians)
contrast = 0.75; %contrast ranges from 0 to 1;


n = 201; %resolution of the image


% Use meshgrid to define matrices X and Y
% that range from -pi to pi;
[X,Y] = meshgrid(linspace(-pi,pi,n));


% Create an oriented 'ramp' matrix as a linear
% combination of X and Y. For
% example, when orientation = 0, cos = 1 and sin
= 0
% so ramp = X.
% When orientation is pi/2 then cos = 0;
% sin = 1 and ramp = Y.


ramp = cos(orientation)*(X-center(1)) +
sin(orientation)*(Y-center(2));
% Sinusoid on the matrix X makes a vertical
grating.
Gabor = sinusoid.*Gaussian;


% Gaussian aperture:
Gaussian = exp( -((X-center(1)).^2+(Y-
center(2)).^2)/width^2);
sinusoid = contrast*sin(spatialFrequency*ramp-
phase);

% A Gabor is the product of the sinusoid and the
Gaussian
```

```
Gabor = sinusoid.*Gaussian;


image((Gabor+1)*127.5); %scale Gabor between 0
and 255
axis equal
axis off
colormap(gray(256));
```

Let's walk through each step of the program. After making the X and Y matrices with meshgrid, an oriented ramp is made just like before.

```
ramp = cos(orientation)*(X-center(1)) +
sin(orientation)*(Y-center(2));
imagesc(ramp); axis equal;  axis off
colormap(gray(256));
```

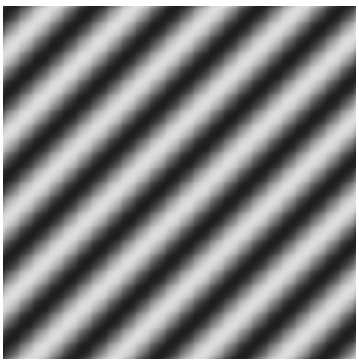


The only thing new is the use of `center(1)` and `center(2)` in our definition of the ramp. This makes the value of ramp at the center point equal to zero.

Then a new matrix `sinusoid` is the sinusoid of the ramp. Notice how we use the variables `contrast` and `phase`.

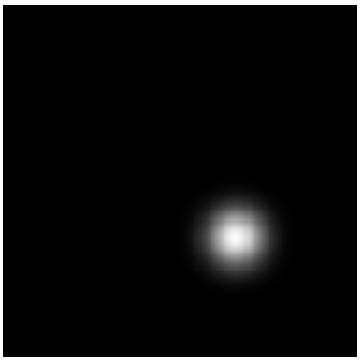
```
sinusoid = contrast*sin(spatialFrequency*ramp-
phase);
```

```
image((sinusoid+1)*127.5); %scale between 0 and
255
axis equal
axis off
colormap(gray(256));
```



We next make a Gaussian centered at (`center(1), center(2)`). This is done as before, except that we incorporate center into our definition of the Gaussian

```
Gaussian = exp(-((X-center(1)).^2+(Y-
center(2)).^2)/width^2);
image(Gaussian*255); %scale between 0 and 255
axis equal
axis off
colormap(gray(256));
```



Finally, the matrix `Gabor` is the element-wise product of the Gabor and the sinusoid:

```
Gabor = sinusoid.*Gaussian;
image((Gabor+1)*127.5); %scale Gabor between 0
and 255
axis equal
axis off
colormap(gray(256));
```

To show this stimulus properly you need to 'calibrate' your display so that you know exactly how much light is coming off the monitor for each value in the color map. We'll get to calibration in a later book.

6.7 Polar coordinates

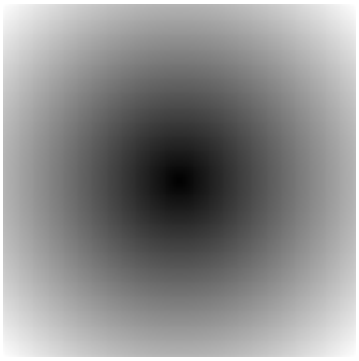
Sometimes it is useful to work in polar coordinates instead of Cartesian (X,Y) coordinates. Polar coordinates come in useful when making images that vary systematically either around a circle or with distance from the origin. A point in polar coordinates is defined by its distance from the origin (`rad`) and its angle counterclockwise from horizontal (`theta`). Images can be computed in polar coordinates by converting our `X`, `Y` matrices made from `meshgrid` into two new matrices (`rad`, `ang`):

```
[X,Y] = meshgrid(linspace(-pi,pi,n));
rad = sqrt(X.^2+Y.^2);
```

```
ang = atan2(Y,X);
```

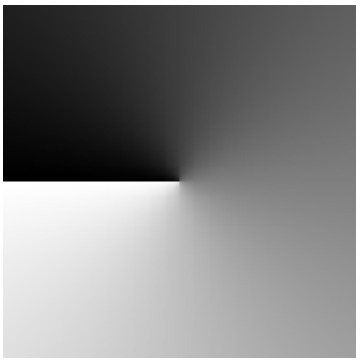
The matrix `rad` is zero at the origin and increases outward. An image of `rad` looks like this:

```
imagesc(rad);
axis equal;axis off;colormap(gray(256));
```



The matrix `ang` looks like this:

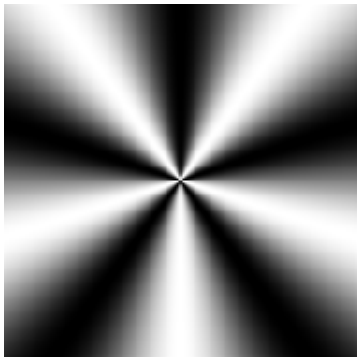
```
imagesc(ang);
axis equal;axis off;colormap(gray(256));
```



Doing math on these two matrices can make some cool looking images. A radial grating modulates sinusoidally around the clock:

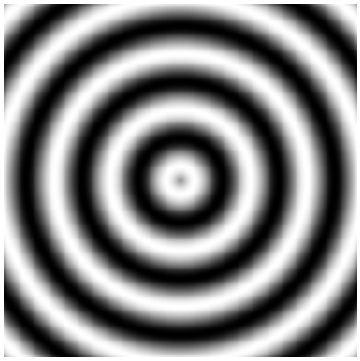
```
radialGrating = sin(5*ang);
imagesc(radialGrating);
```

```
axis equal;axis off;colormap(gray(256));
```



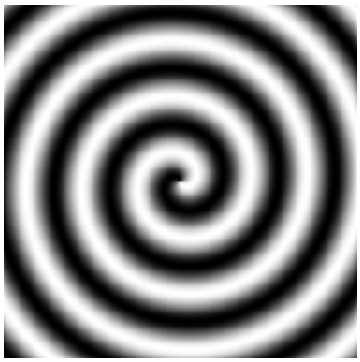
Take the sin of the rad matrix and you get a circular grating:

```
circularGrating = sin(2*pi*rad);
imagesc(circularGrating);
axis equal;axis off;colormap(gray(256));
```



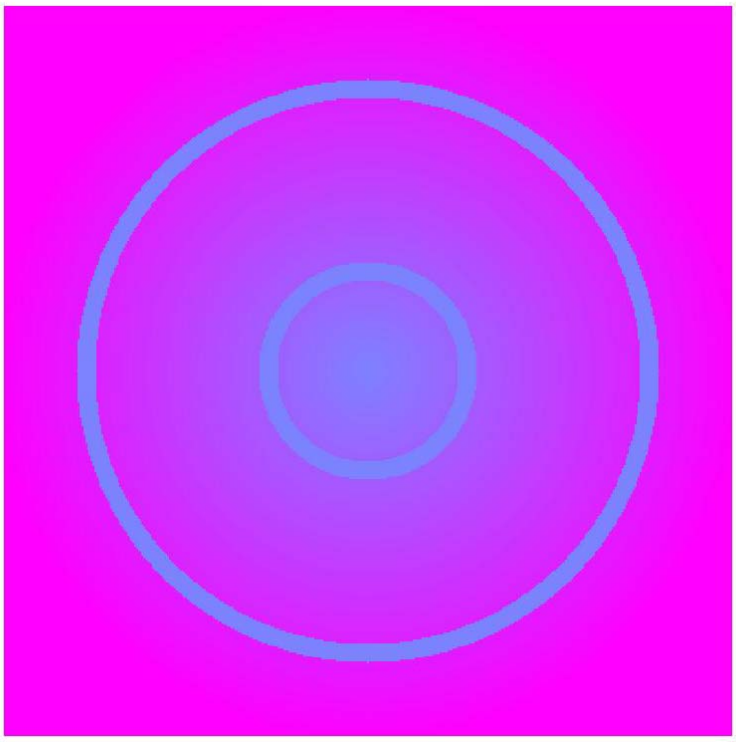
Take the sin of a combination of `ang` and `rad` and you get a spiral:

```
spiral = sin(2*pi*rad+ang);
imagesc(spiral);
axis equal;axis off;colormap(gray(256));
```



6.8 ColorIllusion.m

Finally, here's an example of using `meshgrid` to make a cool illusion.



```
% ColorIllusion.m
% creates the image of a color illusion
% the gray circles are the same
% gray, but don't look the same

close all
[x,y] = meshgrid(linspace(-1,1,401));
r = sqrt(x.^2+y.^2);
r(r>=.25 & r<=.3) = 0;
```

```
r(r>=.75 & r<=.8) = 0;


figure(1); clf
image((r+1)*127.5);
axis equal
axis off
colormap([[.7 .7 .7]; cool(256)]);
saveas(gcf,'figure6_16.jpg')
```

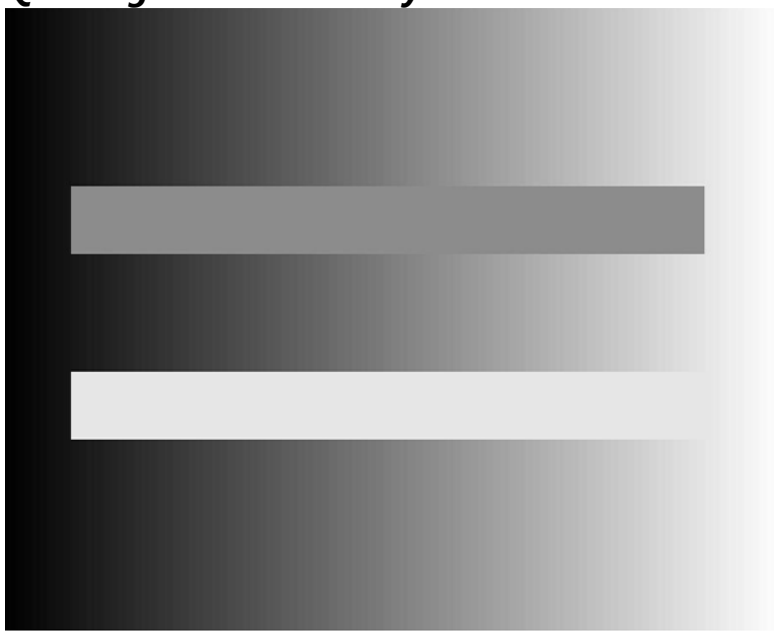
`meshgrid` creates two matrices - the rows of `x` are copies of the vector `linspace(-1,1,401).` The columns of `y` are copies of `linspace(-1,1,401)`.

We then find points where the radius is greater than 0.25 and less than 0.3 and make those points in the matrix 0. We then scale the matrix `r` between 0-255 and `image` it.

Remember that the inner and outer rings were set to be 0, which means that they will take the first value of the `colormap`. Here we fix the first value of the `colormap` to be gray, `[0.7 0.7 0.7]` and then allow the rest of the colormap to take nice funky colors. You can also try out this illusion using other colormaps like `spring` and `hsv` etc. etc. Or create your own colormap.

Questions for Chapter 6

Q 6.1 Lightness constancy



Both bars are uniform but appear to vary in luminance because their ratios with the background vary (this particular version of the illusion is courtesy of Mike Harris, Birmingham, UK)

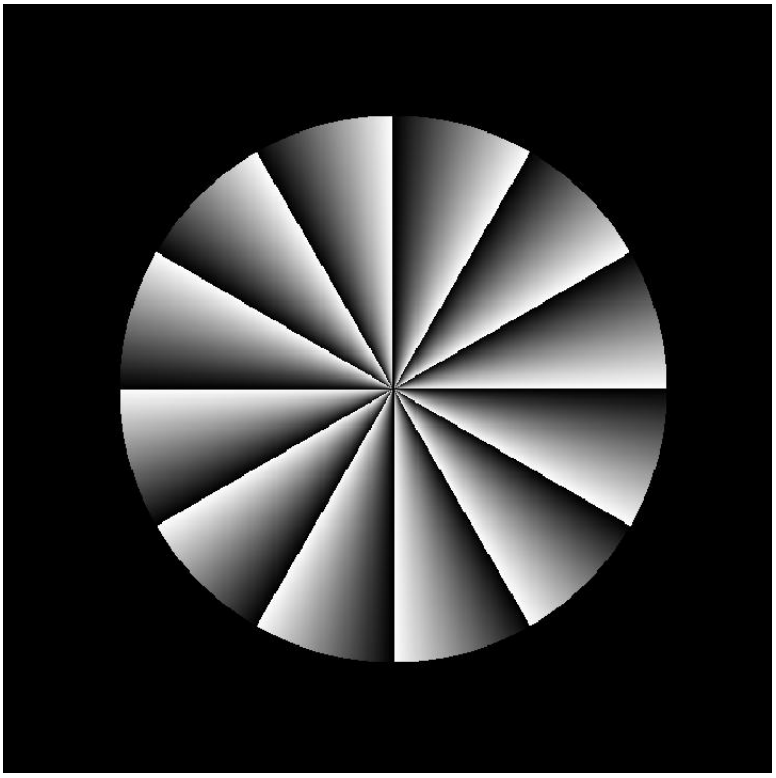
Create this illusion using Matlab.

For fun, check out what it looks like with different colormaps.

Q 6.2 Peripheral drift illusion

This is the peripheral drift illusion: patches containing sawtooth

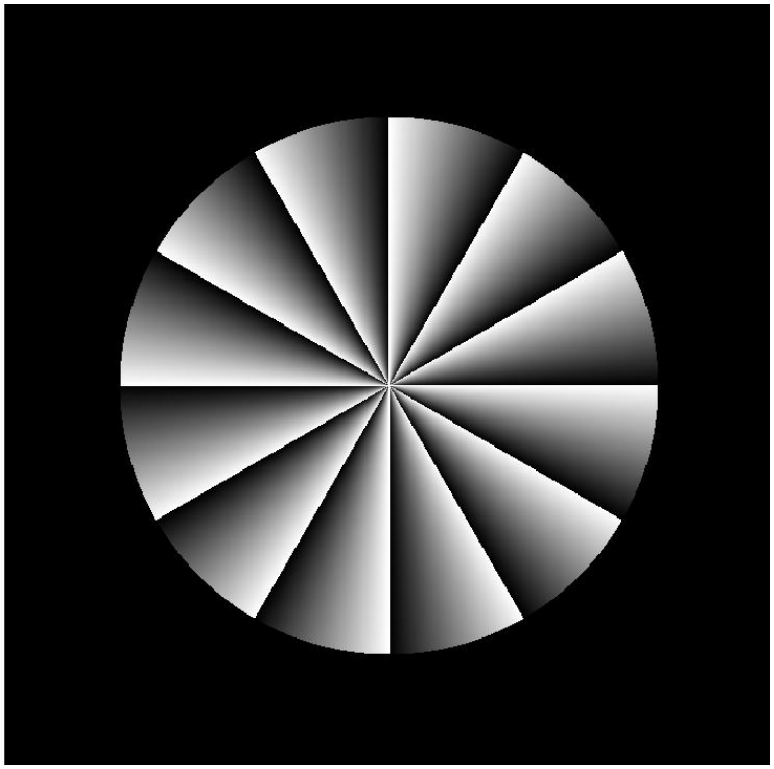
luminance profiles produce a sensation of motion when viewed in the periphery (when you don't look directly at it). In this example the direction of apparent motion is clockwise.



Faubert, J and Herbert, A. (1999). The peripheral drift illusion: A motion illusion in the visual periphery. Perception, 28, 617-622.

a) Make this illusion. For fun, check out what it looks like with different colormaps.

b) Now modify your code to make this version. What direction is the apparent motion now?



CHAPTER 7 – VARIABLE TYPES (CLASSES)

7.1 Variable classes

This chapter is about how variables come in all sorts of flavors, called 'variable classes' that allow you to organize your data in a more efficient and intuitive manner.
So far we've talked about three kinds of variables: strings, vectors and matrices. A string holds a list of characters (usually letters), a vector holds a list of numbers and a matrix holds a n-dimensional list of numbers. Recall that the way you define the variable tells Matlab what type of variable it is. For example:

```
clear all
str = 'No good deed goes unpunished.';
```

Defines the variable `str` as a string of characters, while

```
x= 1:5:100;
```

Defines the variable `x` as a list of numbers. You can see what class and size variables are using `whos`.

```
whos
 Name   Size      Bytes Class   Attributes

 str    1x29        58 char
 x     1x20       160 double
```

The variable x is of class 'double'. Matlab says that it uses up 160 Bytes. What does that mean?

A variable that is of class 'double' contains numbers that are 'double-precision floating point numbers'. This means that each number in the matrix uses up 8 'bytes' of memory.

The smallest piece of memory in a computer is a bit, where each bit represents a zero or one. A byte is a group of eight 'bits' of memory. Because each bit can either be 0 or 1 a single byte can take `2^8 =256` possible values. Matlab counts memory in bytes not bits (because bits are just too tiny).

A variable of class `double` consists of 8 bytes. So a `double` is enough memory represent `256^8,` which is more than 18,447,000,000,000,000,000 possible values.

You can see above that the variable x uses up 160 bytes of memory. That's because it contains 20 numbers that each use up 8 bytes. A double generally isn't used to represent values from 0-1.8447e+19. Instead it represents a smaller range of numbers with greater than integer precision (i.e. it can represent numbers to several decimal places). The important thing to know is that a variable of class double can represent a wide range of values with excellent precision.

The variable `str` has 25 characters and uses up 50 Bytes of memory. So that means that a single character only uses up two bytes (65536 possible values) of memory. The default size of memory assigned to characters is a single. That's because even in Lithuanian, there's still only a limited number of possible characters.

The default class of numbers in Matlab is a double – whenever you define a variable that contains numbers, the variable will be of class double unless you specify otherwise. This is usually a good thing; it is generally useful to be using the most precise type of variable because then you don't have to worry much about rounding errors and other sorts of imprecisions in your calculations. But the drawback is that

`doubles` use up lots of memory.

For example, if I want to generate two huge matrices of random numbers:

```
clear all
x = rand(10000,10000);
y = rand(10000,10000);
??? Error using ==> rand
Out of memory. Type HELP MEMORY for your options.
```

I run into memory problems on my computer, even though I cleared all the variables in memory beforehand with the `clear all` command (if you don't run into memory problems, just increase the size of the matrices and feel smug that you have a better computer than I do).

Fortunately, there are other classes of variables that don't use up as much memory – with the limitation that they also can't represent as many unique numbers and therefore don't carry as much precision. Often that's OK, and the loss of precision is acceptable. But you still need to tell Matlab to use a different kind of variable.

One commonly used class of variable that uses a single byte for each element (as compared to 8 bytes for a double) is called an `int8` or an `uint8`, for 'Unsigned 8-bit integer'. Variables of class `int8` contain integer values between -127 and 127. Variables of class `uint8` contain positive (unsigned) integers between 0 and 255. You can define a matrix of zeros with class `uint8` as follows (don't forget the semicolon!).

```
img = zeros(1000,1000,'uint8');
```

You can also turn the class of any variable into a uint8 with the command of the same name:

```
x = uint8(1:10)
```

```
class(x)
x =
 1 2 3 4 5 6 7 8 9 10
ans =
uint8
```

Be careful with `uint8`. Matlab doesn't allow certain calculations with them:

```
sin(x)
??? Undefined function or method 'sin' for input arguments of type
'uint8'.
```

Some operations are allowed, like addition and subtraction, but screwy things can happen:

```
x - 5
ans =
  0  0  0  0  0  1  2  3  4  5
```

Because variable of class `uint8` cannot be negative, anything below zero gets truncated to zero. This sort of problem is tough to debug, so use uint8s carefully!

There is a whole host of classes for variables. You can see a list with the command:

```
help class
CLASS Return class name of object.
  S = CLASS(OBJ) returns the name of the class of
object OBJ.

Possibilities are:
double     -- Double precision floating point
number array
            (this is the traditional MATLAB
matrix or array)
single     -- Single precision floating point
```

```
number array
logical     -- Logical array
char       -- Character array
cell       -- Cell array
struct      -- Structure array
function_handle -- Function Handle
int8       -- 8-bit signed integer array
uint8       -- 8-bit unsigned integer array
int16       -- 16-bit signed integer array
uint16      -- 16-bit unsigned integer array
int32       -- 32-bit signed integer array
uint32      -- 32-bit unsigned integer array
int64       -- 64-bit signed integer array
uint64      -- 64-bit unsigned integer array
<class_name>  -- MATLAB class name for MATLAB
objects
<java_class>  -- Java class name for java objects
```

We'll now discuss some classes that you might find useful.

7.2 Logical variables

The tiniest variable is of class logical. A logical is a single bit, so it contains either a zero or 1. These are often used in statements with 'logical operators' such as AND (&&) and or (||) where zero is false and 1 is true.

```
a = logical(1);
b = logical(0);

a || b
ans =
   1

a && b
ans =
```

```
   0
```

By the way, English can be ambiguous with the word 'or'. For example, if you say 'Bring me your tired or hungry', you'd probably include those who are both tired *and* hungry. But if you say 'Eat or be eaten.' you probably don't mean both eat *and* be eaten. Computer languages hate ambiguity, so to deal with this there are two kinds of 'or'. The standard `or` operator is true if either statement is true. In contrast, xor ('exclusive or') is true if either `a` or `b` is true but is false if both are true.

```
tired = logical(1);
hungry = logical(1);

tired | hungry
ans =
   1

xor(tired,hungry)
ans =
   0
```

Another way to define a logical variable using with the reserved words `true` (which is reserved to be logical 1) and `false` (which is reserved to be logical 0):

```
x = logical(1);
y = true;
x == y
ans =
   1
```

Here we set `x` and `y` both to 1 with both having class `logical` using two different methods. The last line compares the two variables. The result of the comparison is another `logical` variable set to 1 because the comparison is `true`. Let's set a third variable to `logical(0)` using the reserved word `false`, and compare it to `not x`.

```
z = false;
z == ~x;

ans =
   1
```

This also returns a 1 (`true`) because `z` is `false` and x is `true`. So comparing `z` to not `x` (`~x`) gives us a `true` result.

7.3 Structures

Sometimes it is useful to clump related variables together. For example, suppose that you want to organize information about your experimental subjects, such as id number, age, handedness, and gender into a single variable. This can be done using a *structure*, (the class of variables is called `struct`) . A structure is simply a variable with sub-variables called *fields*. Here's how to define the variable of class `struct`:

```
subject.id = 213;
subject.age = 28;
subject.hand = 0;
subject.gender = 1;
```

We've defined a single variable subject that has four fields called `id`, `age`, `hand` and `gender`. Here we'll define handedness as 0 or 1 for right and left, and gender as 0 or 1 for male and female. These fields were defined using the period '.' To see the whole structure, type the name of the variable without a semicolon:

```
subject =

    id: 213
    age: 28
```

```
   hand: 0
 gender: 1
```

To see the value of a single field, type the name of the structure and field without a semicolon:

```
subject.age
ans =
  28
```

One of the really cute things is that you can make a vector (or even an array) of structures. So additional subjects can be entered as follows:

```
subject(2).id = 301;
subject(2).age = 25;
subject(2).hand = 0;
subject(2).gender = 0;

subject(3).age = 43;
subject(3).id = 200;
subject(3).hand = 1;
subject(3).gender = 0;
```

We can associate more data with each subject by adding fields. These fields can be vectors or arrays:

```
subject(1).data = rand(1,5);
subject(2).data = rand(1,7);
subject(3).data = rand(1,4);
```

Once the subject data is entered this way (we don't recommend that you make up and publish your own randomly generated data like this), you can do things like calculate the data means for just the right-handed subjects:

```
count = 0;
```

```
for i=1:length(subject)
  if subject(i).hand == 0
    count = count+1;
    meanData(count) = mean(subject(i).data);
  end
end

meanData
meanData =
  0.6786  0.5676
```

You can go crazy and have fields of structures. Let's set up a new structure to hold data for two animal subjects:

```
animalSubject(1).id = 23;
animalSubject(1).data = rand(1,5);
animalSubject(2).id = 46;
animalSubject(2).data = rand(1,4);
```

We can create a new structure that has the human and animal structures as fields:

```
allSubjects.human = subject;
allSubjects.animal = animalSubject;

allSubjects =
   human: [1x3 struct]
   animal: [1x2 struct]
```

The structure `allSubjects` now has two fields, one is a field containing the structure `human` that holds information for the human subjects, and the other is a structure `animal` that holds information for the animal subjects. We can access the third human subject's gender like this:

```
allSubjects.human(3).gender
ans =
```

```
    0
```

7.4 Cell Arrays

Another way to group together variables is with a cell array. Cell arrays are matrices with elements that can have any mix of classes. For example, to put a string of characters in the first element of a cell array, we can do this:

```
myCell{1} = 'this is a string of characters.';
```

What's so cool about cells is that other elements of this cell array can be of a different type. Let's make the second element a vector of numbers:

```
myCell{2} = 2.^(1:5);
```

The third element could be a structure, like the one from the last section:

```
myCell{3} = allSubjects;
```

Just as we defined the elements of the cell array with curly brackets, we use curly brackets to access them too:

```
myCell{3}
```

```
ans =
    human: [1x3 struct]
    animal: [1x2 struct]
```

Suppose we want to know the third value of the second element of this cell array. One way is to pull out the second element as a new variable, and then access its third value:

```
newVector = myCell{2};
```

```
newVector(3)

ans =
   8
```

Another way which is shorter but a little weird until you get your head around it, is to use curly brackets followed by round brackets. Think of it as accessing the third element in the vector that is the second element of `myCell`.

```
myCell{2}(3)

ans =
   8
```

Cell arrays have lots of cool uses. One of the first ways you will probably use them is for holding lists of character strings. While it is possible to create a matrix of characters, that has the limitation that each row has to have the same number of characters.

```
days(1, :) = 'Monday';
days(2, :) = 'Tueday';
days(3, :) = 'Wedday';
```

If you try using different length strings in a matrix you get an error:

```
days(1, :) = 'Monday';
days(2, :) = 'Tuesday';
Subscripted assignment dimension mismatch.
```

Using a cell array allows each row to have a different length:

```
days = {'Monday','Tuesday','Wednesday','Thursday','Friday'};
days
'Monday'  'Tuesday'  'Wednesday'  'Thursday'
```

```
'Friday'

days{3}
ans =
Wednesday
```

Cell arrays are useful, but can be confusing at first. Confusion and delay often comes from deciding whether to refer to cell array elements with curly {} or round ( ) brackets. If you use round brackets to refer to elements, it returns a new cell array containing those elements. Here's an example. We'll define a new cell array with some arbitrary things:

```
clear all
myCell{1} = 10;
myCell{2} = 5;
myCell{3} = 'not a number';
myCell{4} = true;
```

If we want a new cell array that contains only a subset of these things, we use round brackets, and we use the usual square brackets to create a vector of indices.

```
newCell = myCell([1,3])

ans =
[10]  'not a number'

Whos

 Name      Size            Bytes  Class  Attributes

 myCell    1x4               489  cell
 newCell   1x2               256  cell
```

`newCell` is a new cell array containing the first and third members of the original cell array.

Since using round brackets keeps things as cell arrays, we can't do things like math on the elements. This won't work:

```
myCell(1) + myCell(2)
```

```
??? Undefined function or method 'plus' for input
arguments of type 'cell'.
```

In other words, you can't add two cell arrays together. But if you refer to elements using curly brackets, it takes the elements out of the cell arrays and gives you variables with their original class (the class they were when you poked them into the cell array). This does work:

```
myCell{1} + myCell{2}
```

```
ans =
  15
```

This works because `myCell{1}` and `myCell{2}` are the actual numbers 10 and 5 (of class `double`). So this command is exactly the same as `10+5`.

Sometimes you might want to pull out elements of a cell array that have the same class and put them into a vector. For example, you might want to pull out the first two elements of `myCell` to make the vector `[10 5]`. Remember, round brackets return a cell array and we want a vector. The trick is to turn the cell array into a vector (or matrix) using the command `cell2mat`:

```
myDouble = cell2mat(myCell([1,2]))
```

```
myDouble =
  10   5
```

Remember from above, `myCell([1,2])` returns a cell array

containing two elements. The command `cell2mat` turns that into a two element vector.

Another less obvious way to do the same thing is to put square brackets *after* referring to the elements with curly brackets:

```
[myCell{[1,2]}]

myDouble =
  10   5
```

I know this is weird. What's happening is that using curly brackets with `myCell{[1,2]}` returns the actual values of `myCell`, but as two 'answers':

```
myCell{[1,2]}

ans =

  10
ans =

   5
```

The square brackets around `myCell{[1,2]}` concatenates these two answers into a single vector. It's kind of ugly, but you should know about it because you'll find it in Matlab code sometimes.

Up to here we've defined the elements of a cell array one at a time. This always works, but it can be inefficient. For example, suppose you have two strings that are labels for conditions: 'test' and 'control', and you have a 10x2 matrix of numbers that correspond to 10 measures for each of the two conditions. One way to make a 11 x 2 cell array that contains everything is with a `for` loop:

First, define the conditions and the data (random again, not recommended for publication)

```
condname = {'test','control'};
data = rand(10,2);
```

To place the condition names into our cell array called `dataCell`, we simply make `dataCell` equal to `condname`, since `condname` is already a cell array:

```
dataCell = condname;
```

To insert the values of the matrix `data` into the subsequent rows of `dataCell`, we will do a nested `for` loop. Note that the indexing is `i+1` and `j`, The +1 places the data in `dataCell` one row below data, so we don't overwrite the first row containing `condname`.

```
condname = {'test','control'};
data = rand(10,2);
dataCell = condname;
or i=1:10
  for j=1:2
    dataCell(i+1,j) = {data(i,j)};
  end
end
for i=1:10
  for j=1:2
    dataCell(i+1,j) = {data(i,j)};
  end
end
```

The new cell array `dataCell` should look something like:

```
dataCell =
  'test'    'control'
  [0.3517]  [ 0.0759]
  [0.8308]  [ 0.0540]
```

```
[0.5853]  [ 0.5308]
[0.5497]  [ 0.7792]
[0.9172]  [ 0.9340]
[0.2858]  [ 0.1299]
[0.7572]  [ 0.5688]
[0.7537]  [ 0.4694]
[0.3804]  [ 0.0119]
[0.5678]  [ 0.3371]
```

`for` loops are slow in Matlab, but as usual there's a faster command that avoids them called `num2cell`:

```
dataCell = [conds; num2cell(data)];
```

Here we've simply concatenated vertically along the rows (using ';') the cell array `conds` with the matrix `data` that has been converted into its own cell array.

We appreciate that cell arrays can be confusing. Don't feel stupid – this stuff takes practice. Your authors will confess to occasionally fiddling with round and curly brackets until things work.

Questions for Chapter 7

Q7.1 Structures containing arrays, and an array of structures

Given the vector:

```
vect = 1:10;
```

a) Make a structure `x` that has a field called `vect` that contains the vector above so that `x.vect` contains:

```
x.vect
ans =
   1   2   3   4   5   6   7   8   9  10
```

b) Make an array of structures `y`, each containing a field `vect` with a single number so that when you type y you get:

```
y
y =

1x10 struct array with fields:
  vect
```

and

```
y(1)
ans =
  vect: 1

y(2)
```

```
ans =
  vect: 2
```

etc.

Q 7.2 Cell arrays

a) Make a cell array called `bigCell` that contains in order:

the vector `vect` from the problem 7.1

the structure `x` from problem 7.1a

the structure `y` from problem 7.1b

the vector `11:20`

the string `'This is the fifth element of bigCell'`

the string `'Matlab is great!'`

b) Make a new cell array called `lessBigCell` that contains the 1st, 4th, and 6th element of `bigCell`

c) Add together the 1st and 4th element of `bigCell`

d) Concatenate the 1st and 4th elements of `bigCell` to create the matrix:

```
 1   2  3   4   5   6   7   8   9  10
11  12  13  14  15  16  17  18  19  20
```

(This one has a Hint.)

e) Make a new string `str` that contains the string ‘This is strings’, pulled from the 5th and 6th elements of `bigCell`.

CHAPTER 8 FUNCTIONS

'Functions' are little Matlab programs that you can call from other Matlab programs. Proper use of functions will help to make your programs much easier to read and modify.

A function is a self-contained block of commands that performs a coherent task of some kind. It's like outsourcing in business - sending a task away to be performed somewhere else. We send materials to Singapore, they make the sneakers, they send back the sneakers and we really don't know much about what happened in Singapore. When you send a task to a function, you give the function all the variables it needs to know, and it returns the variables that you want. The calculations and variables within the function are hidden.

8.1 Why use functions?

There are three main reasons for writing functions.

- To make your code readable – hiding pieces of code in functions makes the overall structure of your code clearer and easier to read.
- To shorten your code – if you do the same thing repeatedly putting it in a function allows you to call the function repeatedly instead of repeating the same lines of code again and again.
- To give you a library of routines. You will do many things again, and again in your career writing code. Writing a function to do it means that the next time you need to do that particular manipulation you don't need to rewrite that piece of code.

Whenever you write code, think about whether you will ever need that particular piece of code again in a later program, or whether you will be using that code more than twice in your current program. If either of those things are the case, the "correct" thing to do as far as programming style is concerned is to make it a function. Even if it's just

two lines – it will be easier finding a function then rummaging through old code for those precious two lines.

Beginning an m-file with the word `function` tells Matlab that you want a function m-file.

```
function out=SimpleFunction(in)
% a very simple function
% written by if 4/2007

out=10*in;
```

The first line of your m-file defines this piece of code as a function. The terminology is that any variables you want to send into your function (the raw materials for sneakers) go inside the brackets after the function name (`in`). Any variable you want to return from the function (the finished sneakers) comes before the equals sign (`out`). Variables sent in and out of functions are also called input and output arguments.

Inside this particular function all that is happening is that your input is being multiplied by 10, and that value is your output.

This is how you use SimpleFunction.m. Type the following into your command line.

```
my_out=SimpleFunction(3)
my_out =
  30
```

Provided SimpleFunction.m is in your path, you should now have a variable `my_out` which will be 30.

Note that the names of the variables returned by SimpleFunction don't have to match the names of the variables within SimpleFunction. In this example the output of the function is called `out` inside the function and `my_out` at the command window.

The analogy would be that the outsourcing factory might call the sneaker the *no.34532* while the company sending the materials and receiving the sneaker might call it the *SuperDuperRebox™.*

The same is true for sending in variables, as you'll see from the example below.

```
inval=5;
my_out=SimpleFunction(inval);
my_out
my_out =
  50
```

Functions can take more than one variable in, and send more than one variable out.
Create a new m-file:

```
function [output1,
output2]=SimpleFunction2(input1, input2)
%
% another very simple function
%
% written by if 4/2007


output1=input1.*input2;
output2=input1./input2;
```

Then in the command window try the following:

```
myinput1=3;
```

```
myinput2=4;
[myoutput1, myoutput2]=SimpleFunction2(myinput1,
myinput2)
myoutput1 =
  12
myoutput2 =
  0.7500
```

Because we used the `.*` and `./` commands for multiplying and dividing, SimpleFunction2 can take vectors as well as inputs:

```
vect1 = [1:5];
vect2 = [2:2:10];
[myoutput1, myoutput2]=SimpleFunction2(vect1,
vect2)
myoutput1 =
   2   8  18  32  50
myoutput2 =
  0.5000  0.5000  0.5000  0.5000  0.5000
```

You have been actually using functions already - most of the commands you have been using already are functions written by the programmers at Mathworks. For example, `round` – you give the command round any number as the input argument and it returns the closest integer as the output argument.

```
a=3.45;
b=round(a)
b =
   3
```

8.2 scaleMat

Next we'll write a nice little utility function that will scale the values of a matrix to range between to preset values. The first argument sent to

the function is the unscaled matrix. The next two values are the min and max values for the scaled matrix that is returned. If no min and max values are sent, then the matrix is scaled by default to be between 0 and 1.

```
function outMat = scaleMat(inMat,range)
%outMat = scaleMat(inMat,[range])
%
%Scales a matrix to a new range of values.
%
%Inputs:
%  inMat:    unscaled matrix
%  range:   1x2 vector containing the low and
high values
%          for the scaled matrix. Default is
[0,1].
%
%Output:    scaled matrix with minimum value
range(1), and
%          max value range(2)
% Written 9/08 by G.M. Boynton and I. Fine


%Deal with default values for range if none are
provided
if ~exist('range','var')
  range = [0,1];
end


%minimum value of input matrix
minVal = min(inMat(:));


%max value of input matrix
maxVal = max(inMat(:));
```

```
%scale the matrix to range between zero and 1.
outMat =  (inMat-minVal)/(maxVal-minVal);

%scale this new matrix to values in 'range'

outMat = (range(2)-range(1))*outMat+range(1);
```

We can call this function in two ways. First, by defining the new range:

```
mat = [1,2;3,4]
mat =
   1   2
   3   4

scaleMat(mat,[0,12])
ans =
   0   4
   8  12
```

Or by using the default range of zero to 1

```
scaleMat(mat)
ans =
     0  0.3333
  0.6667  1.0000
```

This example illustrates how to set default values for parameters if the user does not explicitly provide them. It relies on the function `exist`. When the second argument into `exist` is the string var, the function returns a logical variable (0 or 1) depending on whether the first argument sent into `exist` is a variable that exists in memory. For example:

```
clear all
```

```
x = pi;
exist('x','var')
ans =
   1
```

Here we are asking whether `x` is a variable that exists in memory. It is, so `exist` returns `true` (remember `true` is identical to `logical 1`).

```
exist('y','var')
ans =
   0
```

The `exist` function returned `false` for the variable `y` because it does not exist in memory.

Note that in the help documentation for the function `scaleMat`, the input variable range is placed in square brackets. This is a convention for documentation to let the user know that this variable is optional and will be set to a default value if not provided.

8.3 SineInAperture

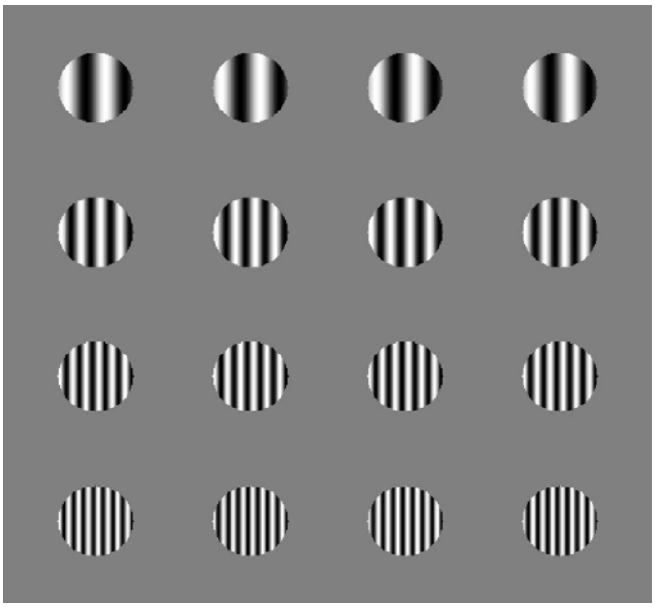
Now let's create a new version of `SineInAperture.m` that uses a function called MakeSineInAperture.m to create the windowed sinusoids. We'll use the vector `x`, the spatial frequency (`sf`), and the radius of the aperture (`rad`) as input arguments. The 2D image of the sinusoid is the output.

```
% SineInAperture
%
% creates an image of apertured vertical
sinusoids in a
% tiled array
```

```
% information about the sinusoids
clear all
close all
x=linspace(-pi, pi, 100);
sf=[3 6 9 12]; % spatial freq in cycles per image
rad=2;


% creates a 100x100xlength(sf) matrix containing
two 2D apertured sinusoids,
% of different spatial frequencies.


for s=1:length(sf)
 sinewave2D(:, :, s)=MakeSineAperture(x, sf(s),
rad);
end


% initialize the tilematrix by filling it with
zeros
imgsize=length(x);
ntiles=4;
sep=30;
tilesize=(ntiles*(imgsize+sep))+sep;
tilematrix=zeros(tilesize);
startpos=sep:sep+imgsize:length(tilematrix)-1;


for rtile=1:ntiles
 for ctile=1:ntiles
tilematrix(startpos(rtile):startpos(rtile)+imgsiz
e-1, startpos(ctile):startpos(ctile)+imgsize-
1)=sinewave2D(:, :, rtile);
 end
end
imagesc(tilematrix);
colormap(gray(256))
axis off;
```

This won't work until we create the function MakeSineAperture.m:

```
function sinewave2D = MakeSineAperture(xval,
sf, radius)

% function that creates a vertical
% sinewave windowed by a circular aperture
%
% takes as input arguments:
% xval: for which the sinusoid is computed
% sf: the spatial frequency of the sinusoid
% radius: the radius of the aperture in
pixels
% returns as output the 2D sinusoidal
grating image

onematrix=ones(size(xval));
sinewave=sin(xval*sf);
sinewave2D=(onematrix'*sinewave);
for r=1:length(xval)
 for c=1:length(xval)
  if xval(r).^2+xval(c)^2>radius^2
  sinewave2D(r, c)=0;
  end
 end
end
```



8.4 insertGabor

Structures are useful for organizing and grouping variables for sending into functions. If a function takes in a lot of variables the command can be unwieldy. But if you clump variables together into structures, the function calls can be much simpler. For example, in the last chapter we wrote a program that creates an oriented Gabor. Let's modify that code so that it adds a Gabor into an existing image, and we'll put all the parameters that define the Gabor in a single structure. Here's the function - much of it should look familiar:

```
function outImg = insertGabor(params,inImg)

%Use meshgrid to define matrices x and y that
range from -pi to pi;
[x,y] = meshgrid(linspace(-pi,pi,params.n));

if ~exist('inImg','var')
 inImg = zeros(params.n);
```

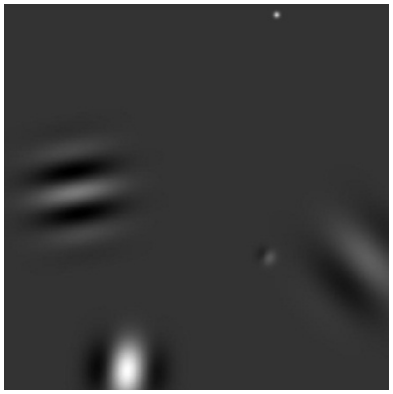
```
end

%Create an oriented 'ramp' matrix as a linear
combination of x and y. For example,
% when params.orientation = 0, cos = 1 and sin = 0
% so ramp = x. When
% params.orientation is pi/2 then
% cos = 0; sin = 1 and ramp = y.
ramp = cos(params.orientation)*(x-params.center(1)) +
sin(params.orientation)*(y-params.center(2));

%Sinusoidal carrier is a sinusoid on the matrix
'ramp'
sinusoid = params.contrast*sin(params.sf*ramp-
params.phase);

%Gaussian envelope:

Gaussian = exp( -((x-params.center(1)).^2+(y-
params.center(2)).^2)/params.width^2);

%A Gabor is the product of the sinusoid and the
Gaussian
Gabor = sinusoid.*Gaussian;
outImg = inImg + Gabor;
```

Here's some code that calls the function insertGabor that we just wrote. Note how we define all of the parameters for the Gabor in a single structure, which is passed into the function insertGabor as a single variable.

```
params.n = 400; %size of image (pixels)
img = zeros(params.n); %start with blank
image
```

Loop through five times, adding a new Gabor to the image each time:

```
for i=1:5
  %Pick random parameters for each Gabor
  params.center = 2*pi*(rand(1,2)-.5);
  params.orientation = pi*rand(1); %radians
(pi/4 = 45 degrees)
  params.width = rand(1);
  %1/e half params.width of Gaussian
  params.sf = 12*rand(1);
  %spatial frequency of Sinewave carrier
(cycles/image)
  params.phase = 2*pi*rand(1);
  %spatial params.phase of sinewave carrier
(radians)
  params.contrast = rand(1);
  % params.contrast ranges from 0 to 1;
  img =insertGabor(params,img);
end

%Show the image using the usual commands:
figure(1)
clf

%scale Gabor between 0 and 255
img = scaleMat(img,[0,255]);
image( img);
axis equal
axis off
colormap(gray(256));
```



Questions for Chapter 8

Q 8.1 nonanStats

Write a function that calculates the mean, standard deviation and the standard error of the mean of the values in a vector `x` which can deal with `NaN`. The function should carry out the following steps:

a) Make a new vector `y` that contains the values of `x` that are not `NaN`s.

b) Set the variable `m` to the mean of `y` using the `mean` function.

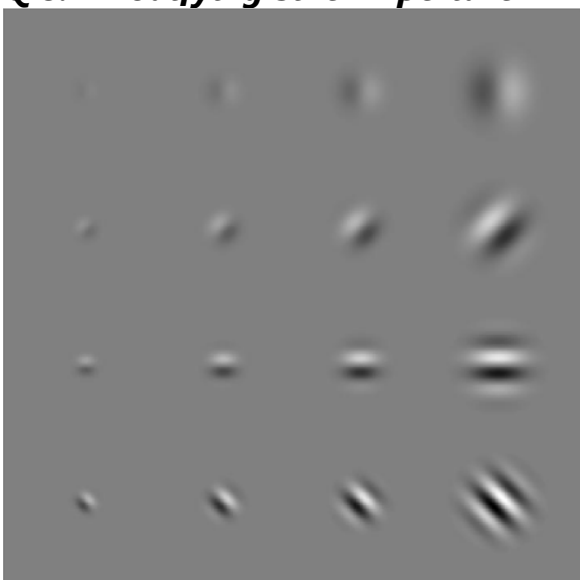
c) Set the variable `s` to the standard deviation of `y` using the `std` function.

d) Set the variable `sem` to the standard error of the mean of `y` by dividing `s` by the square root of the length of `y`.

e) Turn these steps in to a function called `nonanStats` that should start like this:

```
[m,s,sem] = nonanStats(x)
```

Q 8.2 Modifying SineInAperture



Modify the files `SineInAperture`, and the function `MakeSineAperture` to make this image. The parameters you need to know are these:

```
sflist=[1 2 3 4];
% spatial frequency in cycles per image
radlist=[1 1.5 2 3 ];
phaselist=[0 pi/2 pi 3*pi/4];
orientationlist=[0 0.7854 1.5708 2.3562];
```

The contrast of the Gabors are:

```
 0.2000 0.3333 0.4667 0.6000
 0.3333 0.4667 0.6000 0.7333
 0.4667 0.6000 0.7333 0.8667
 0.6000 0.7333 0.8667 1.0000
```

The Gaussian width is the radius/2.

(Hint: The Gabors are windowed with both a Gaussian and a circular aperture.)

CHAPTER 9: BASIC PLOTTING

9.1 The 6 stages of plotting data

1. Prepare your data in variable structures that make data easy to plot. If you don’ t think about how you are going to analyze your data when you are writing your experimental program then analysis will be a lot more complicated. One REALLY good idea is to create a fake data set and analyze it before collecting real data. This is also often a good way of realizing that there may be issues in your experimental design that you hadn't thought of. It's also a great way of checking for bugs in your analysis code. Making this part of your standard procedure will make you a much better scientist.

2. Decide which figures you want, and how you want them to be organized.

3. Plot the data.

4. Set axis limits, tick marks etc.

5. Annotate the graph with x and y labels, titles and legends explaining each curve. Labels should be informative ("water-deprived" rather than "condition 1").

6. Print the figure, hold it out at arms’ length and see whether someone with presbyopia (your supervisor) will be able to read it.

90% of students omit stages 5-6 a significant proportion of the time. The easiest way to make your supervisor love you is to not forget those two stages.

9.2 Why use Matlab rather than Excel?

There are two reasons to plot data in Matlab rather than Excel.

1. *Matlab is more flexible.* Matlab makes it easy to explore data by plotting it in a wide variety of ways. This can help you get a greater insight into your data than you could by simply carrying out a standard set of tests. While you could easily use Excel to create many of the plots that we create in this chapter, some examples of plotting data we describe here would be very difficult or impossible in Excel.

2. *It is easier to replot data using Matlab.* You will find that you re-plot data repeatedly either as you collect more data, or as you (or your supervisor) think of new ways to look at it. A well written Matlab function will easily allow you to make modifications (e.g. throwing out outliers, only looking at male subjects) without having to entirely recreate the figure.

3. A Matlab program provides a record of the analyses you carried out and the resulting plots. Another piece of important advice: whenever you submit a paper, carefully check and save a record of both the data and the Matlab files you made to make the final figures. (Read the Matlab documentation on *Identifying Dependencies* to find out how to identify exactly which m-files to save.) Make sure it's all backed up outside your computer. This simple step means you can recreate the exact figures that you submitted if the reviewers request a minor modification (or if your lab is audited for some reason, it does happen). It can be surprisingly laborious to work out exactly what Matlab files you used four months later.

Most of the information in this chapter (and more interesting stuff) can be found in the Matlab help documentation in the section *Basic Plotting Commands*.

9.3 figure windows

Here we are simply closing any windows that might already be present,

then putting up a figure window with no data in it so we can look at its properties.

```
close all
figure(1);
get(gcf)
```

```
    Alphamap = [ (1 by 64) double array]
    CloseRequestFcn = closereq
    Color = [0.855388 0.855388 0.855388]
    Colormap = [ (64 by 3) double array]
    CurrentAxes = []
    CurrentCharacter =
    CurrentObject = []
    CurrentPoint = [0 0]
    DockControls = on
    DoubleBuffer = on
    FileName =
    FixedColors = [(3 by 3) double array]
    IntegerHandle = on
    InvertHardcopy = off
    KeyPressFcn =
    KeyReleaseFcn =
    MenuBar = figure
      Etc …
```

`get(gcf)` means "*get me the properties of gcf*". `gcf` is shorthand for "get current figure". So the `gcf` command retrieves the current figure (the one that was most recently modified) and `get` lists its properties.

(For people with experience with other programming languages, `gcf` is a handle (which is a generalized term for a pointer) to the last figure that Matlab created.)

As you can see there are a lot of properties.

Suppose we want to find out how to change the mouse pointer shape as it hovers over the window. We would find out what the possible alternative mouse shapes are as follows.

```
set(gcf, 'Pointer')
[ crosshair | fullcrosshair | {arrow} | ibeam | watch
| topl | topr | botl | botr | left | top | right |
bottom | circle | cross | fleur | custom | hand ]
```

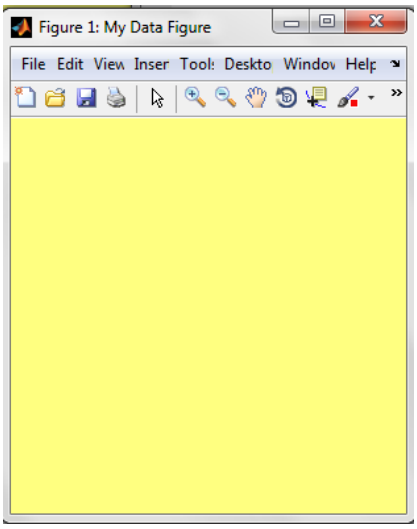
To change the properties of the mouse pointer we specify what sort of pointer we want.

```
set(gcf, 'Pointer', 'fleur')
```

Now the mouse arrow should change into a rather nice fleur shape when it hovers over the figure.

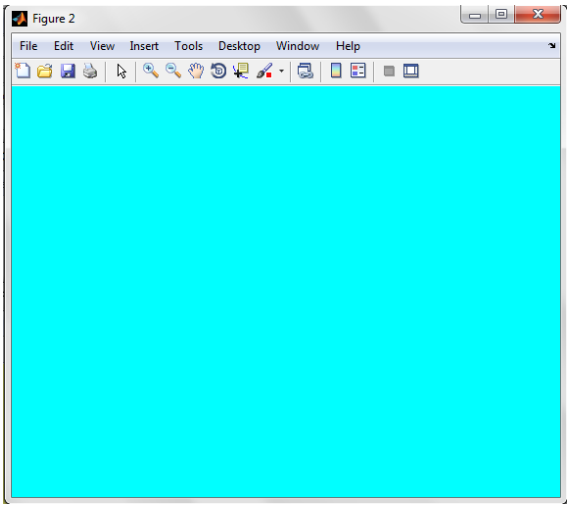
You can change a very wide variety of figure properties using the `set` command.

```
set(gcf, 'Color', [1 1 .5])
set(gcf, 'Position', [100 100 300 300])
set(gcf, 'PaperOrientation', 'landscape');
set(gcf, 'Name', 'My Data Figure');
```



Sometimes you will have more than one figure and want to be able to access and change properties on one that isn't the most recent figure. In that case you need to create a handle to each figure.

```
close all
f1=figure(1);
f2=figure(2);
set(f1, 'Color', [1 1 .5]);
set(f2, 'Color',[0 1 1]);
```



You can also put figure handles into an array as if they were a list of

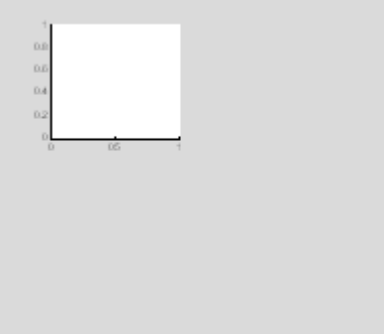
numbers. At this point it’s probably easier to write these commands in a junk m file.

```
clear all
close all
clist=[1 1 .5; .5 1 1];
for i=1:2;
 f(i)=figure(i);
 set(f(i), 'Color', clist(i, :));
end
```

9.4 axes

Any data plot is contained within an axis. A figure by default will contain a single axis, but it's possible to create a set of `subplots` (axes) within a single figure window. The following code creates a figure window organized so as to contain two rows and two columns of subplots, and calls up the first of those subplots.

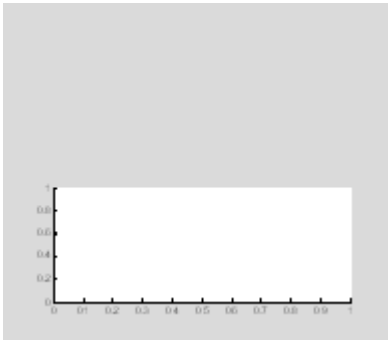
```
close all
figure(1)
subplot(2,2,1)
```



Here's another example.

```
close all
```

```
figure(1)
subplot(2, 1, 2)
```



Like figures, subplots have various properties which are accessed in a very similar way as figure properties. Here gca stands for get current axis, and allows you to get the properties of the most recently modified subplot.

```
get(gca)
    ActivePositionProperty = position
    ALim = [0 1]
    ALimMode = auto
    AmbientLightColor = [1 1 1]
    Box = off
    CameraPosition = [0.5 0.5 9.16025]
    CameraPositionMode = auto
    CameraTarget = [0.5 0.5 0.5]
    CameraTargetMode = auto
    CameraUpVector = [0 1 0]
    CameraUpVectorMode = auto
    CameraViewAngle = [6.60861]
    CameraViewAngleMode = auto
    CLim = [0 1]
    CLimMode = auto
    Color = [1 1 1]
    CurrentPoint = [ (2 by 3) double array]
```

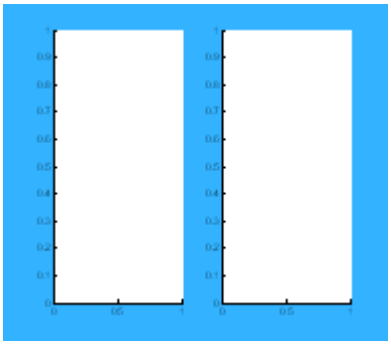
```
ColorOrder = [ (7 by 3) double array]
DataAspectRatio = [1 1 1]
Etc. …
```

Note that an axis has different properties from those of a figure. Once again you can look at the various options you have for a given axis properties using `set`.

```
set(gca, 'FontWeight')
[ light | {normal} | demi | bold ]
```

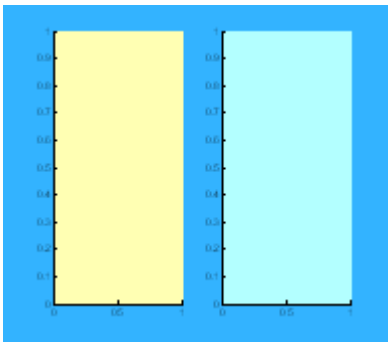
Here's a list of some of the most useful properties of axes that you might want to change. As you go through these commands look at how the figure changes. But you can also find and modify lots of other properties. You can find out what properties are available to change using `get(gca)` and `set(gca)`.

```
clear all
close all
figure(1)
set(gcf, 'Color', [.2 .7 1]);
% set background color to blue
a1=subplot(1, 2, 1);
a2=subplot(1, 2, 2);
```



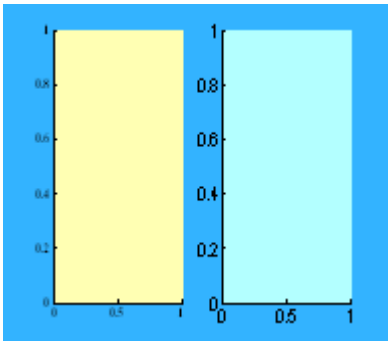
axis color properties

```
set(a1, 'Color', [1 1 .7]);
% Sets the color inside the subplot.
set(a2, 'Color', [.7 1 1]);
```



axis font properties

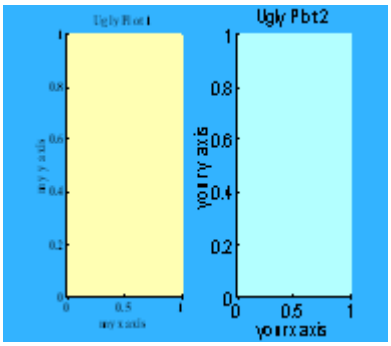
```
set(a1, 'FontName', 'Times');
set(a1, 'FontSize', 12);
set(a1, 'FontWeight', 'bold')
set(a2, 'FontName', 'Arial');
set(a2, 'FontSize', 16);
```



axis labeling

Note that you need to use a slight variant of terminology for title and axis labeling.

```
title(a1, 'Ugly Plot 1');
xlabel(a1, 'my x axis');
ylabel(a1,'my y axis');
title(a2, 'Ugly Plot 2');
xlabel(a2,'your x axis');
ylabel(a2,'your y axis');
```

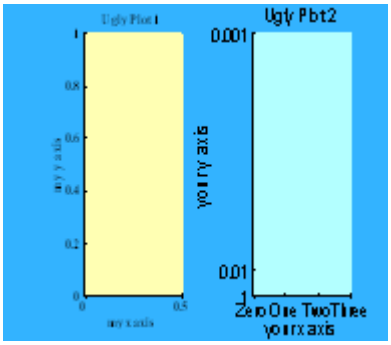


axis limits, tick placement and tick labeling

Currently the plot has ticks between 0-1. You can use either text or a vector to replace default tick labels with whatever you want. First you need to specify where you want the ticks to be, and then you need to specify how they are labeled. Here we are using a cell array to label our axes.

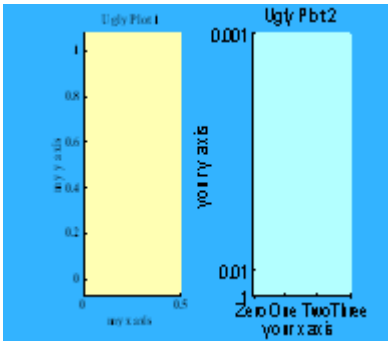
```
set(a1, 'XLim', [0 .5]);
set(a2, 'XLim', [0 3]);
set(a2, 'YLim', [1 1000]);

% set the limits of the axes to be different
% from the default of 0-1
set(a2, 'XTick', 0:1:3);
set(a2, 'YTick', [1 100 1000]);
% put tick marks exactly where you want them

set(gca,'XTickLabel',{'Zero';'One';'Two';'Three'}
```

```
)
set(gca, 'YTickLabel', [1 1/100 1/1000]);
% the tick mark labels don’t actually have to
match their values
% you can specify whatever you want as tick
labels
```

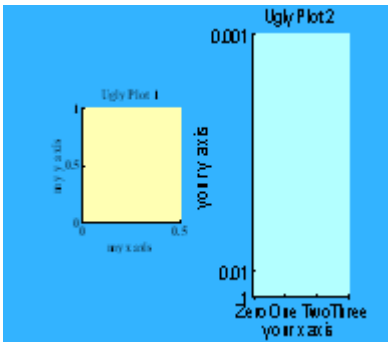


Other useful commands

```
axis(a1, 'equal');
% Sets the axis so tick marks are equally spaced
on x and y axes
```



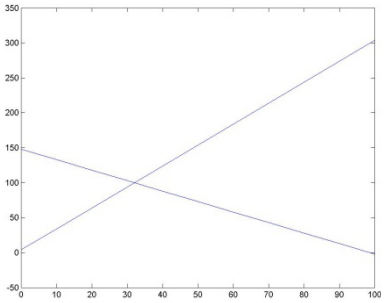

```
axis(a1, 'square');
% makes the current axis square
```



9.5 plotting

Now we are going to plot data on the axes. Once again, plots have properties that you can alter. The kinds of properties you can change depend on the kind of plot you are making.
Let's start with a line plot. The command `hold` tells Matlab to include both plot lines in the figure rather than simply overwriting the first plot line with the second.

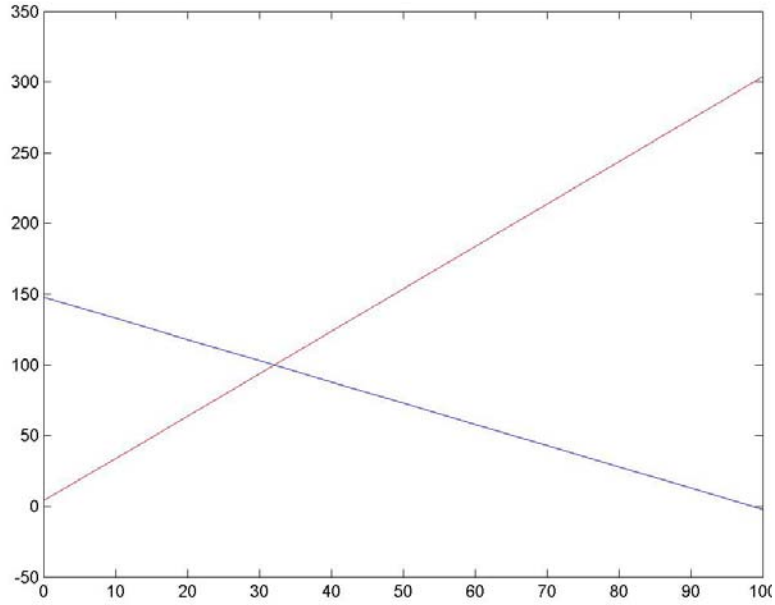
```
clear all
close all
figure(1)
x=0:10:100;
y1=3*x+4;
y2=150-(y1/2);
p1=plot(x, y1);
hold on
p2=plot(x, y2);
```



Color

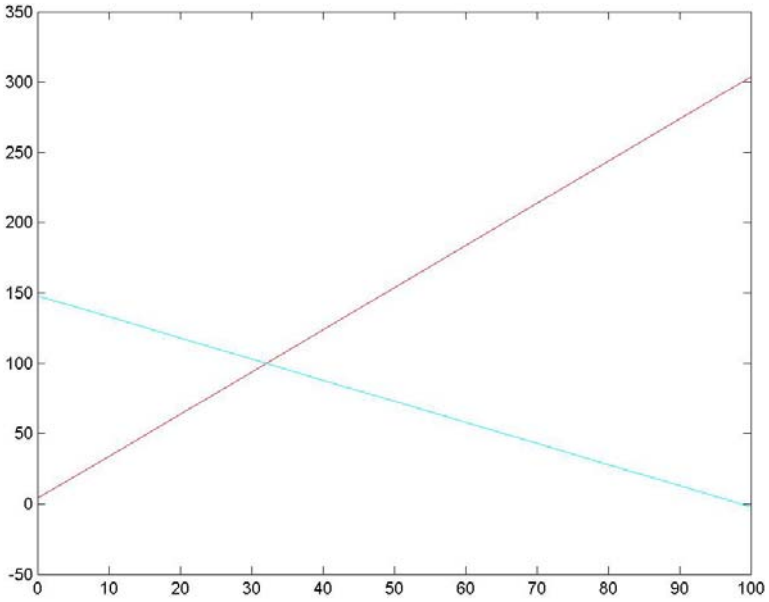
First we'll define the color of these plot lines. One way to do this is to specify the level of the red, green or blue guns, using values between 0-1.

```
set(p1, 'Color', [ 1 0 0]);
```



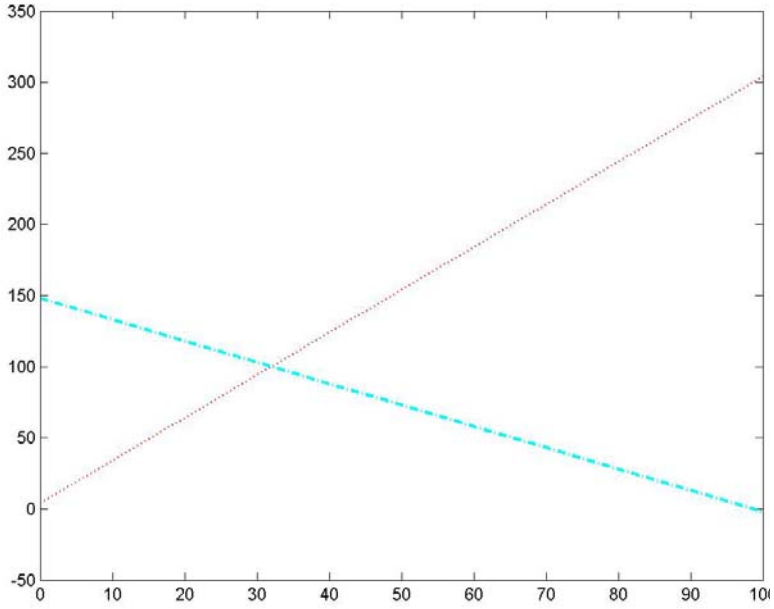
Alternatively you can use shorthand for some specific colors. Options include: `'b', 'g', 'r', 'c', 'm', 'y', 'k'` (blue, green, red, cyan, magenta, yellow black).

```
set(p2, 'Color', 'c');
```



Line style and markers

```
set(p1, 'LineWidth', 1)
set(p1, 'LineStyle', ':');
set(p2, 'LineWidth', 2)
set(p2, 'LineStyle', '-.');
```



Options include:

- = solid line

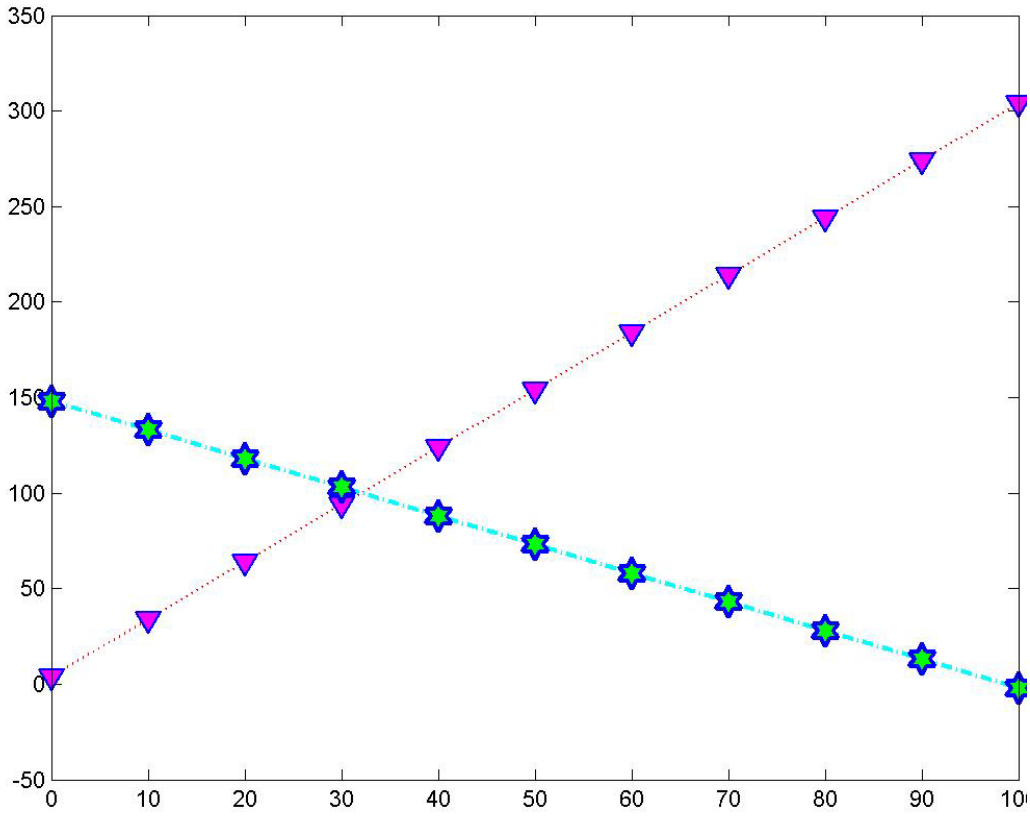
-- = dashed line

: = dotted line

-. = dash-dot line

none = no line

```
set(p1, 'MarkerSize', 10);
set(p1,'MarkerEdgeColor' , [0 0 1])
set(p1, 'MarkerFaceColor', [1 0 1])
set(p1, 'Marker', 'v');

set(p2, 'MarkerSize', 15);
set(p2,'MarkerEdgeColor' , [0 0 1])
set(p2, 'MarkerFaceColor', [0 1 0])
set(p2, 'Marker', 'h');
```

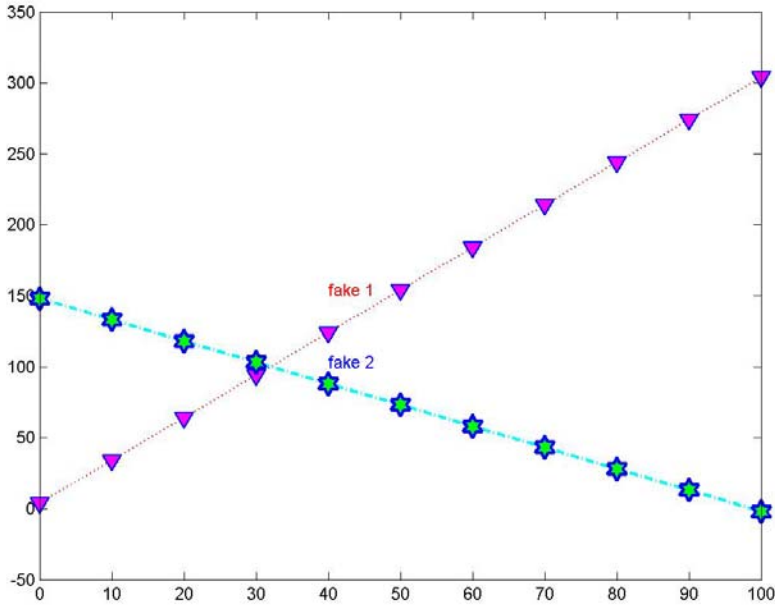


Options for Markers include '+', 'o ', '* ', '.', 'x', 's', 'd', '|', 'v ', '^', '> ', '<' ,'p', 'h', 'none'. The 'v', '<', '>', '^' symbols represent triangles of various orientations

9.6 text

Text objects are manipulated using handles in a very similar way. `text` takes as input the x and y position of the string you want to position, and the text string itself. These text objects can also be assigned handles and you can change their properties. To manipulate properties you once again use `get` and `set`.

```
t1=text(x(5), y1(6), 'fake 1');
t2=text(x(5), y2(4), 'fake 2');
set(t1,'Color', 'r');
set(t2,'Color', 'b');
```



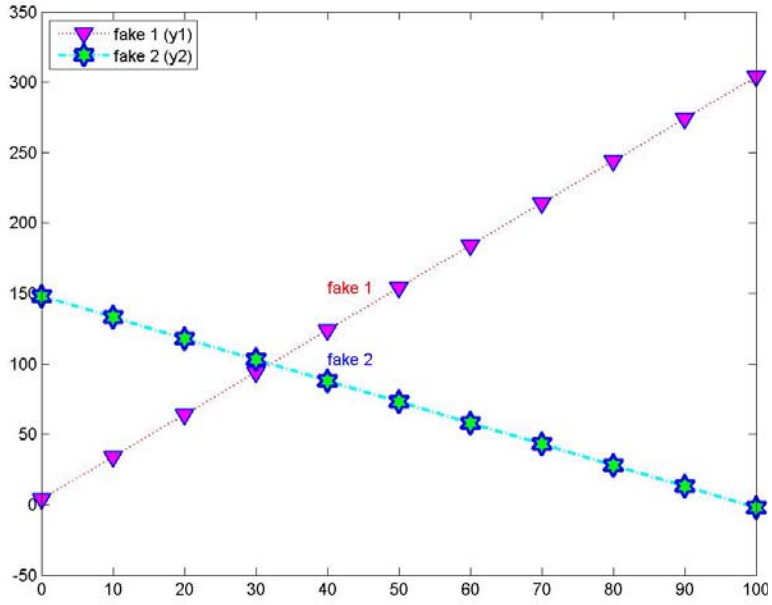

```
get(t1)
```

```
    Annotation = [ (1 by 1) hg.Annotation array]
    BackgroundColor = none
    Color = [1 0 0]
    DisplayName =
    EdgeColor = none
    EraseMode = normal
    Editing = off
    Extent = [39.5402 142.42 8.50575 20.9913]
    FontAngle = normal
    FontName = Helvetica
    FontSize = [10]
    FontUnits = points
    FontWeight = normal
    Etc ...
```

9.7 legend

`legend` has a very similar format. It takes in a list of the plot handles for which you want to create legend entries, and a list of the strings that will be the legend labels.

```
lh=legend([p1 p2], {'fake 1 (y1)'; 'fake 2 (y2)'}, …
 'Location', 'NorthWest');
set(lh, 'FontName', 'Arial', 'FontSize', 10);
```



And once again, legends have a variety of properties that can be altered.

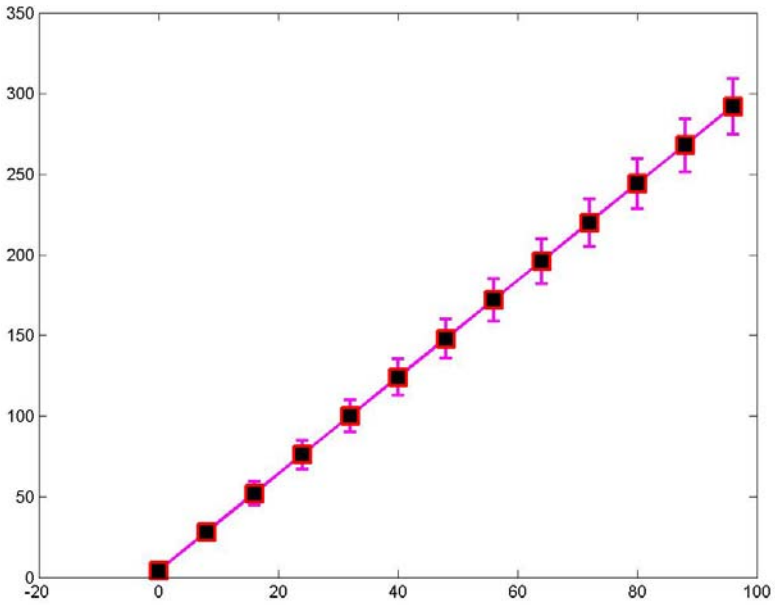
```
get(lh)
ActivePositionProperty: 'position'
         ALim: [0 1]
       ALimMode: 'auto'
   AmbientLightColor: [1 1 1]
          Box: 'on'
```

```
    CameraPosition: [0.5000 0.5000 17.3205]
  CameraPositionMode: 'auto'
     CameraTarget: [0.5000 0.5000 0]
   CameraTargetMode: 'auto'
    CameraUpVector: [0 1 0]
  CameraUpVectorMode: 'auto'
    CameraViewAngle: 6.6086
  CameraViewAngleMode: 'auto'
         CLim: [0 1]
          Etc …
```

You have now created the world's ugliest figure.

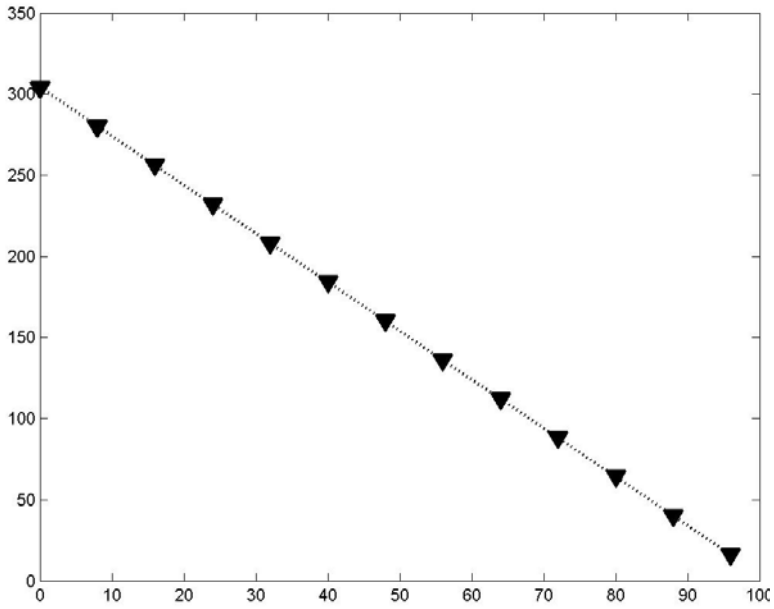
9.8 Errorbar plots

```
clear all; close all
x=0:8:100;
y=3*x+4;
err=sqrt(y);
ph=errorbar(x, y, err);
set(ph, 'Marker', 's', 'MarkerSize', 13, ...
 'MarkerFaceColor', 'k', 'MarkerEdgeColor', 'r',
'Color', 'm', 'LineWidth', 2)
```



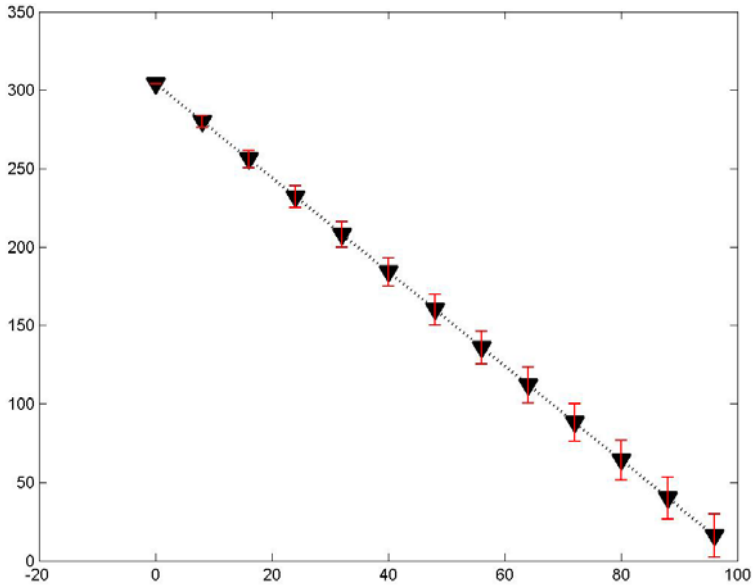
One inconvenient thing about the errorbar plots automatically generated by Matlab is that the error bars always have same properties (e.g. color and line thickness) as the main line. Sometimes you want the error bars to have different properties. The way to do this is to draw the main line and the error bar as overlapping plots, i.e. use the error bars as an overlay. Here we start by creating a plot with black dotted lines and triangular symbols. Notice how we use a single set command to define multiple properties.

```
clear all ; close all
x=0:8:100;
y=300-3*x+4;
err=sqrt(2*x);
p=plot(x, y);
set(p, 'Color', 'k', 'LineStyle', ':',
'LineWidth', 2, 'Marker', 'v', 'MarkerSize',
10,'MarkerEdgeColor' , 'k',
'MarkerFaceColor','k');
```



Then we add red error bars. We define this plot as having no line and no marker so only the actual error bars are visible.

```
hold on
e=errorbar(x, y, err);
set(e, 'Color', [1 0 0 ], …
'LineStyle', 'none', 'LineWidth', 1, 'Marker',
'none');
```



9.9 Log plots

Matlab has functions for plotting on log axes, including `semilogx` and `semilogy`. But these functions don't provide much control over the axes, and the resulting plots are kind of ugly. Instead, in our lab we use a couple of simple functions that we wrote called `logx2raw` and `logy2raw`. Here's the function `logx2raw`:

```
function logx2raw(base,precision)

% logx2raw([base],[precision])
%
% Converts X-axis labels from log to raw values.
% base: base of log transform; default base is e.
% precision: number of decimal places, or a
FORMAT string.

if ~exist('base','var')
 base=exp(1);
```

```
end

origXTick = get(gca,'XTick')';
newXTick = base.^(origXTick);

if exist('precision','var')

set(gca,'XTickLabel',num2str(newXTick,precision))
;
else
 set(gca,'XTickLabel',num2str(newXTick));
end
```

The function doesn't actually plot anything. Instead you use it by first plotting the data with the regular `plot` command after converting your numbers to log vales. `logx2raw` re-labels the x-axis with new numbers corresponding to the original 'raw' values.

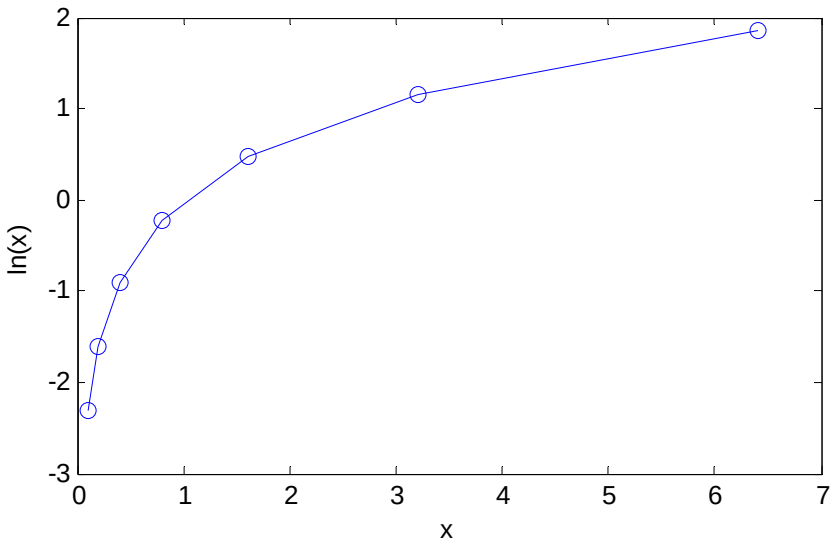
Create `logxraw` and put it in your Matlab path.

Here's an example of how it works. First we'll define our x and y values to plot:

```
x=[.1,.2,.4,.8,1.6,3.2,6.4];
y=log(x);
```

Here is a plot y as a function of x on regular (linear) axes:

```
figure(1)plot(x,y,'o-')
plot(log(x),y);
xLabel('x');
yLabel('ln(x)');
```



To plot the same points on a log-x axis, we first use the regular `plot` command, but use `log(x)` instead of `x`:

```
figure(2)
clf
plot(log(x),y,'o-');
```

Then we'll set the x-ticks to correspond to the data points

```
set(gca,'XTick',log(x))
```

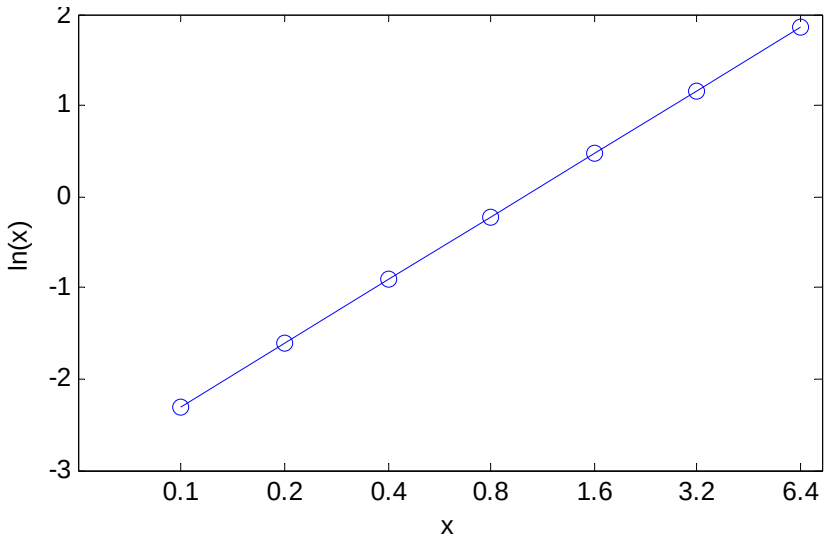
Finally, we'll use our `logx2raw` to re-label the x-axis values.

```
logx2raw
```

And give the graph some labels:

```
xLabel('x');
yLabel('ln(x)');
```

Your new plot should look like this:



It's a line because if `y=log(x)`, then plotting on a log-x axis is simply plotting `log(y)` as a function of `log(x)`, which is like `y=x`.

You might learn some new tricks by looking at the code you typed in for `logx2raw`. The function works by using the get command to find the values for the `XTick` marks, and resetting the `XTickLabel` to hold the logs of the `XTick` values.

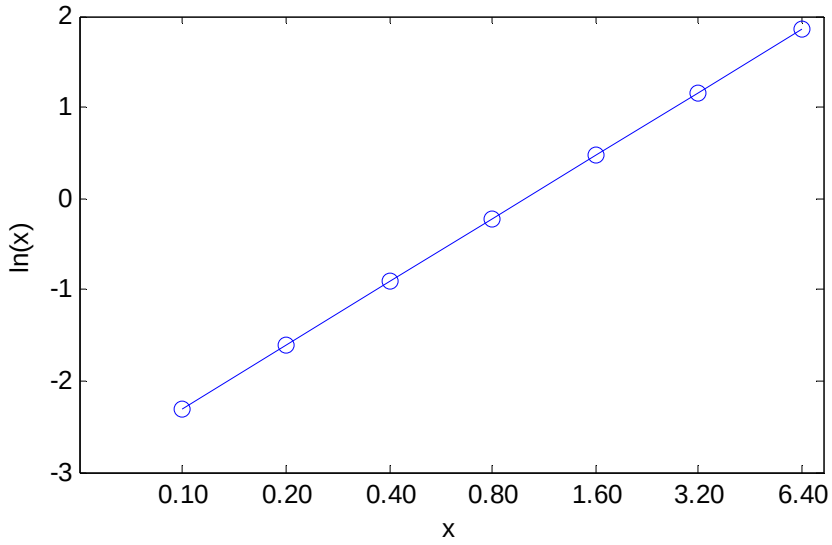
Notice the use of the `exist` function to determine if the user sent in any parameters. The first parameter is the base of the logarithm. See how by default the base is set to 'exp(1)', which is that crazy number 2.7182818284590455…, the base of the natural logarithm.

The second parameter is the 'precision'. You can see how precision works by looking at the help for num2str. If the precision is a number, it is the number of digits shown. Another option for the precision is a FORMAT string which gives you a little more control. FORMAT strings are a throwback to lower level languages like C and FORTRAN.

With figure 2 as your current figure, try this:

```
logx2raw(exp(1),'%5.2f')
```

The new plot should look like this:



Note that we need '`exp(1)`' as the first input into `logx2raw` to set the base to e. The second input is '`%5.2f`' which means 'give me a total of five digits with two on the right side of the decimal point'. See how there are now two decimal points for the numbers on the x-axis.

As a homework question for chapter 9 you will be asked to write `logy2raw` and use it.

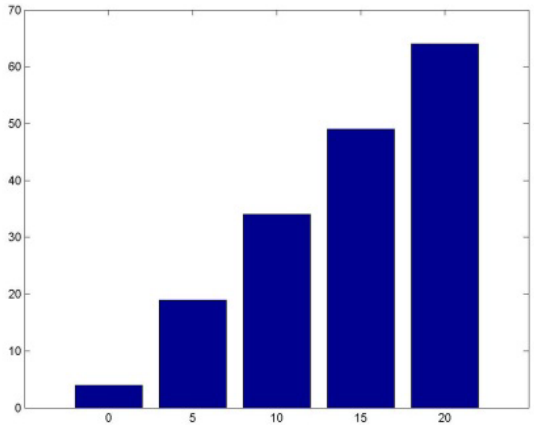
Another nice thing about `logx2raw` and `logy2raw` is that since they simply re-label the axes, you can use it on any kind of plot, including bar plots and 3-d plots.

9.10 Bar plots

Here's how to create a simple bar plot, and change some basic properties using handles.

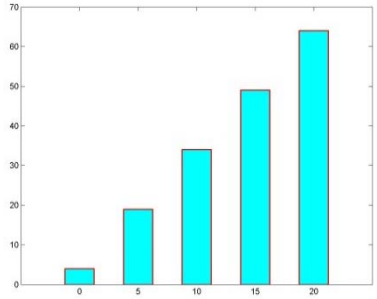
```
clear all ; close all
```

```
x=0:5:20;
y=[3*x+4];
ph=bar(x, y);
```



Bar properties

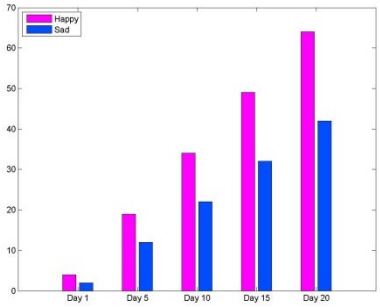
```
set(ph, 'BarWidth', .5);
set(ph, 'EdgeColor', 'r')
set(ph, 'LineWidth', 1.5);
set(ph, 'FaceColor',[0 1 1 ]);
```



More complex bar plots

With bar plots using more than one set of y-values you will get a handle for each set of data (similarly to plotting line data using plot). Here we have two sets of y data for every x-value

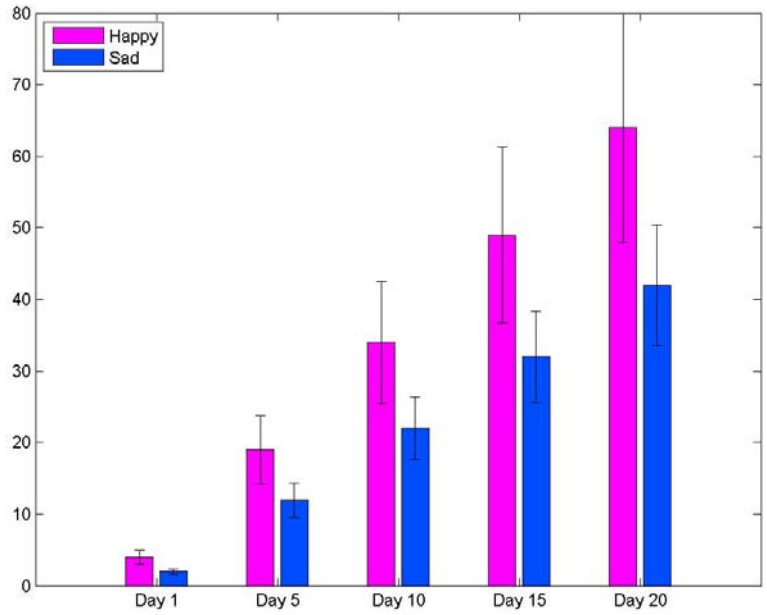
```
clear all; close all
x=0:5:20;
y=[3*x+4;2*x+2]';
bh=bar(x, y);
set(bh(1), 'FaceColor',[1 0 1 ])
set(bh(2), 'FaceColor',[0 .3 1 ])
set(gca, 'XTickLabel',...
 {'Day 1', 'Day 5', 'Day 10', 'Day 15','Day 20'})
legend([bh(1) bh(2)], 'Happy', 'Sad', 'Location',
'NorthWest')
```



9.11 Bar plots with error bars

Matlab doesn't have a function to do this, so you need to use a work around. You add errorbars to bar plots in a very similar way as plotting them as an overlay to line plots. The real nuisance is you have to fiddle with the parameter that controls the width of the bars to make it look right.

```
hold on
width=get(bh(1), 'BarWidth')-.1;
eh1=errorbar(x-width, y(:, 1), y(:, 1)/4, ...
 'Marker', 'none','Color', 'k', 'LineStyle',
'none');
eh2=errorbar(x+width, y(:, 2), y(:, 2)/5, ...
 'Marker', 'none','Color', 'k', 'LineStyle',
'none');
```



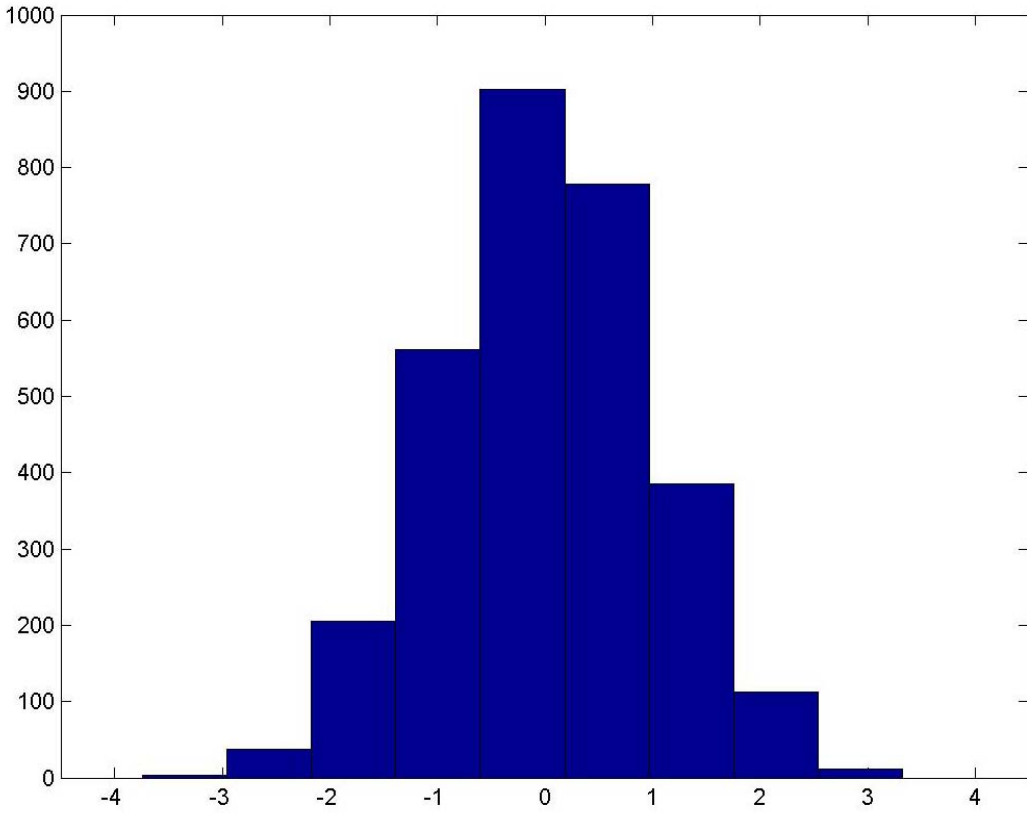
Fiddling with the bar width is very annoying and fiddly. Luckily Bolu Ajiboye (currently an assistant professor at Case Western) has posted a very nice function which deals with this for you on the Mathworks website (where they have a file exchange website). If you go to Mathworks and type `barweb` you can download his function, which is called as follows:

```
handles=barweb(y, err, [], …
{'Day 1', 'Day 5', 'Day 10', 'Day 15','Day 20'},…
 'barweb plot', 'x values', ' y values', [1 0 0 ;
0 1 0 ],…
 [],{'happy', 'sad'})
set(handles.legend, 'Location',
'SouthWestOutside');
% put the legend somewhere sensible
```

The function returns handles to the various components of the plot so you can alter then further if you want. Here we've just moved the legend.

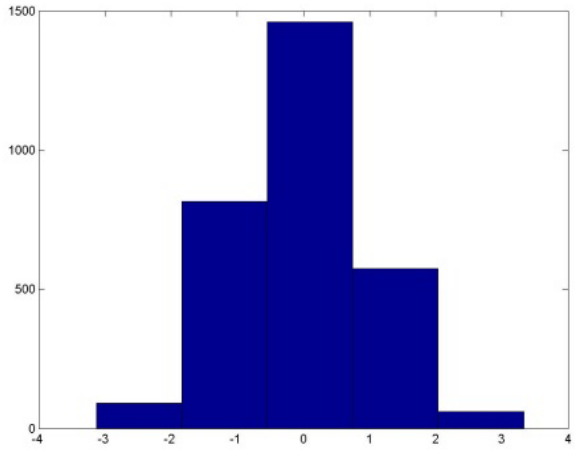
9.12 Histograms

```
clear all; close all
data=randn(3000, 1);
hist(data, 9);
set(gca, 'XLim', [-4.5 4.5])
```



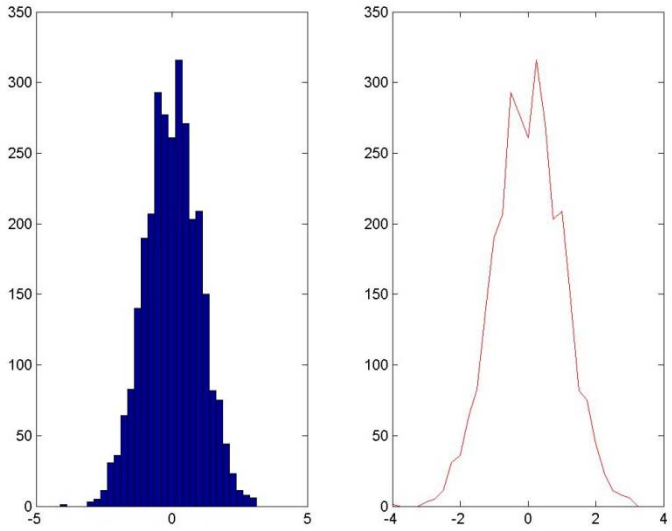
Here you have asked for your data to be divided into 9 bins. If you want to know what the center of the bins are you can ask `hist` to return the number of data points in each bin (`n1`) and the centers of the bins (`x1`) instead of plotting the data. If you want to then plot the histogram then you will need to call `hist` twice.

```
[n1, x1]=hist(data, 5);
n1
n1 = 53 627 1556 692 72

x1
x1 = -2.8111 -1.4280 -0.0450 1.3381 2.7212

figure(1); clf; hist(data, 5);
```



You can also give `hist` a vector of the bin centers. Here we only collect the return argument that tells us the number of data points that fell within each bin. We already know where the bin centers are, because we defined them. We can then plot the histogram using bar.

```
bins=-4:.25:4;
subplot(1, 3,2);
n1=hist(data, bins);
figure(1); clf;
subplot(1, 2, 1)
hist(data, bins)
subplot(1, 2, 2)
plot(bins, n1, 'r')
```



`histc` is a very similar command that takes in a vector that contains the upper and lower limits of each bin. It returns the number of data points that fell within each bin. Note that the bins need no longer be equally spaced.

9.13 Other useful 2d plot commands

Use doc (e.g. doc polar) to learn about these commands.

```
plotyy
polar
stairs
stem
```

9.14 An example of plotting

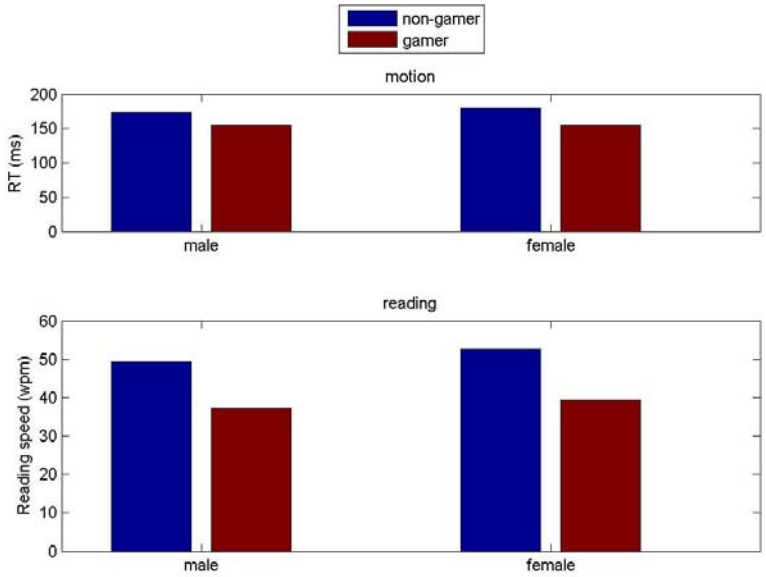
Let's imagine you've collected data measuring students' reaction times to moving stimuli and their reading speed. You've done this for both males and females, and you've recorded whether the subjects play video games vs. non-gamers. So you have the following subject break down:

10 males, 10 females
6 of the 10 males are gamers
4 of the 10 females are gamers.

In this program, `GamersAnon.m`, we'll begin by faking our data, and then we'll plot it as both a bar graph and a scatter plot.

```
% GamersAnon.m
%
% A fake experiment examining the effects of
% gaming on reading speed
% and motion RTs

clear all; close all;


% fake data
data=zeros(4, 20);
data(1, 1:10)=1; % 0 = male, 1 = female
data(2, [[1:6], [11:14]])=1; % 0 = non-gamer,
1=gamer
data(3, :)=180; % default RT speed is 180ms
data(3, [[1:6], [11:14]])=data(3, [[1:6],
[11:14]])-30;
% gamers are 30ms faster
data(3, :)=round(data(3, :)+8*randn(size(data(3,
:))));
% add random noise
data(4, :)=50; % default wpm
data(4, [[1:6], [11:14]])=40;
% gamers are slower
data(4, :)=round(data(4, :)+ 4*randn(size(data(4,
:))));
% add random noise


gender={'male', 'female'};
gamer={'non-gamer', 'gamer'};
```

```
task={'motion', 'reading'};


%% bar graph
figure(1); clf;
for k=1:2 % task
 for i=1:2 % male / female
  for j=1:2 % gamer / non-gamer
   ind= find(data(1, :)==i-1 & data(2, :)==j-1);
   tmp=data(2+k,ind);
   MeanData(k, i, j)= mean(tmp);
   StdData(k, i, j)= std(tmp);
   SemData(k, i, j)=std(tmp)/sqrt(length(tmp));
  end
 end
 subplot(2, 1,k)
 b1=bar(squeeze(MeanData(k, :, :))); hold on
 title(task{k});
 set(gca, 'XTickLabel', gender);
 if k==1
  ylabel('RT (ms)');
  h=legend(gamer);
  set(h, 'Location', 'NorthOutside');
 else
  ylabel('Reading speed (wpm)');
 end
end
```



So, `data` has 4 rows and 20 columns. Each column represents a subject. The first row represents whether the subject was male or female, the second whether the subject was a non-gamer or a gamer, the third row represents their reaction times and the fourth row represents their reading speed.

We then used a combination of nested for loops and `find` to extract the data for each condition.

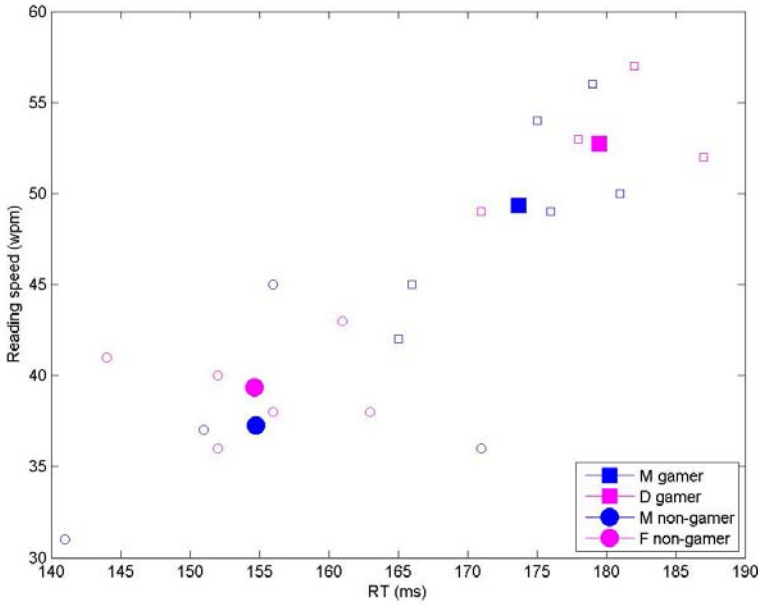
```
size(data)
ans = 4 20
```

As you may recall from Chapter 4, `squeeze` is a funny command. Let's say you have a 3D matrix, but you only want two dimensions from it (a slice from the cube). If you take the slice along anything but the last dimension then Matlab still considers this 2D slice as being 3D – it's a 3D matrix of size 1 x n x m. `squeeze` "squeezes" this 3D slice into an m x n 2D matrix.

Now let's look at our data in a scatter plot and plot the means with

bigger filled in symbols.

```
colorlist=[0 0 1; 1 0 1];
symlist=['s', 'o'];


figure(2); clf;


 for i=1:2 % male / female
 for j=1:2 % gamer / non-gamer
  ind=find(data(1, :)==i-1 & data(2, :)==j-1);
  tmp1=data(3,ind);
  tmp2=data(4,ind);
  plot(tmp1, tmp2, '.', 'MarkerSize', 6, …
           'Color', colorlist(i ,:), 'Marker',
      symlist(j), …
           'MarkerFaceColor', [1 1 1]);
  hold on


% overlay larger solid color symbols for the mean
response
% for male gamers, female gamers, etc. etc.
  h(i, j)=plot(MeanData(1, i, j), MeanData(2, i,
j), …
'MarkerSize', 12, 'Color', colorlist(i ,:), …
'Marker', symlist(j), 'MarkerFaceColor',
colorlist(i ,:));
 end
end


xlabel('RT (ms)');
ylabel('Reading speed (wpm)');
legend(h(:), 'M gamer', 'D gamer','M non-gamer',
'F non-gamer','Location', 'SouthEast');
```



9.15 mesh and surf

These commands are useful when you have two independent and one dependent variable.

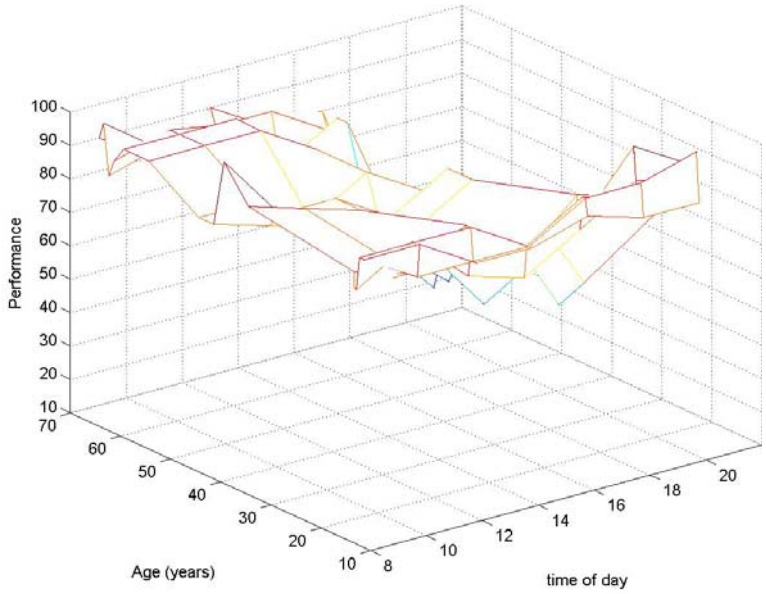
Suppose you were looking at how age affects your ability to perform a task as a function of the time of day. So in this example, `Sleepyhead.m`, you have two independent variables, the time of day and the age of the subject, and one dependent variable, which is performance. (This example was inspired by an eminent faculty member who was taking my class and fell asleep just before I choose an experiment example. Luckily he didn't wake up before I was finished.) Your data might look like this:

```
clear all
age=[17 17.5 18 18.5 19 19 40 45 47 50 60 65 67
68 69 70];
tod=[9 11  1  3  5  7  9];
tod(3:end)=tod(3:end)+12; % move to the 24 hour
```

```
clock!
perf=[90 90 80 80 85 80 80 ;
      90 80 80 80 90 90 95 ;
      90 90 90 80 90 85 90 ;
      80 90 75 70 80 90 85 ;
      80 80 80 70 80 100 90 ;
      85 80 90 75 90 85 85 ;
      90 85 80 70 80 67 40 ;
      100 80 80 70 80 58 30 ;
      80 75 70 65 75 60 36 ;
      80 80 70 80 70 60 40 ;
      90 90 90 80 50 40 20 ;
      90 90 90 70 80 40 30 ;
      85 90 87 70 80 35 22 ;
      80 85 80 75 65 45 22 ;
      95 83 85 85 75 38 25 ;
      90 80 90 80 80 36 18];
```

We are going to plot our data using `mesh(x,y,Z)`. The rule for mesh is that first two vectors must have `length(x) = n` and `length(y) = m` where `[m,n] = size(Z)`.

```
figure(1); clf;
sh=mesh(tod, age, perf);
xlabel('time of day');
ylabel('Age (years)');
zlabel('Performance');
```



You can now play with various aspects of the figure:

```
shading flat
shading interp
colormap(gray)
```

One useful command is `view`. You can rotate your view of this three-D graph manually using the mouse if you hit the little rotation icon on the top of the menu bar figure window. If you rotate the graph to a good view, you might then want to save the values that represent that view so you don't have to manipulate it by hand the next time you plot the data. `view` allows you to save that viewpoint.

```
v=view;
```

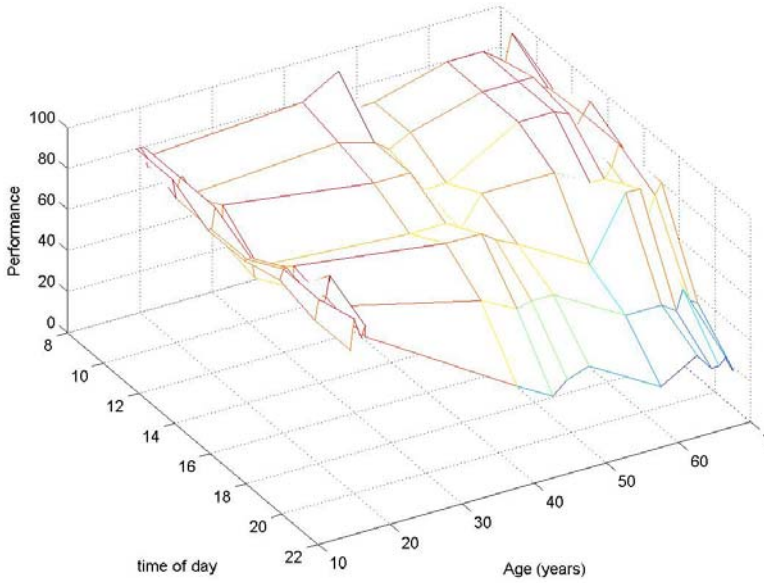
You can then set the view to the desired view which you previously saved by using the saved viewmatrix, v, as an input argument.

```
view(v);
```

Try saving a viewpoint, then rotate the graph manually, and use view to restore the graph to the saved viewpoint.

Alternatively you can provide view with the azimuth and elevation you desire. Azimuth and elevation are basically simpler forms of the 4x4 matrix contained in V.

```
view(60, 50)
```

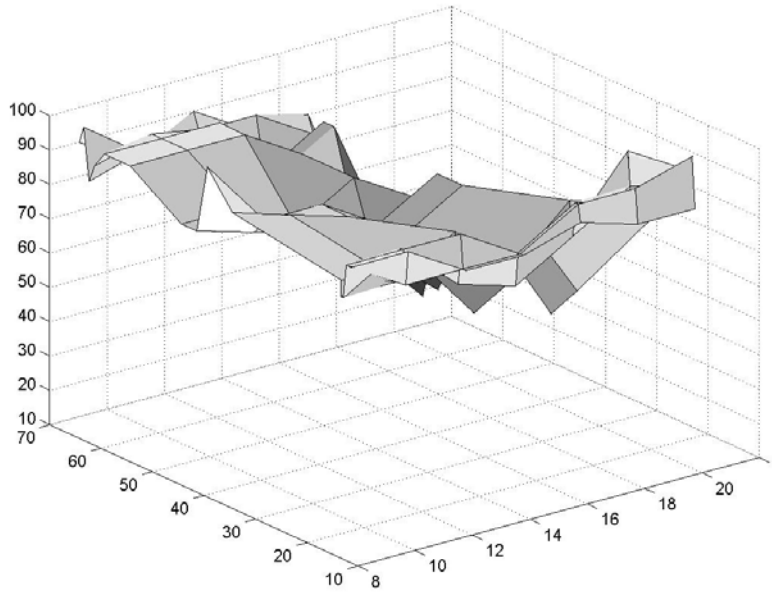


There are also three default views.

```
view
view(2);
view(3);
```

Also try:

```
surf(tod, age, perf);
```



`surf` and `mesh` can deal with missing data values as long as they are set to be `NaN` (not a number). `surf` and `mesh` will simply interpolate between the missing values.

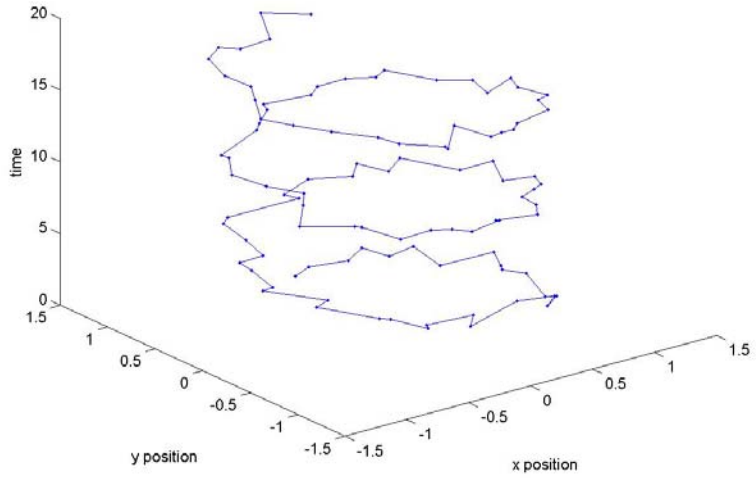
```
perf(3,1)=NaN;
perf(5,3)=NaN;
perf(7,2)=NaN;
sh=surf(tod, age, perf);
```

9.16 plot3

This is often useful when you have one independent and two dependent variables. One example would be if you were tracking the x and y position of a rat over time, as in this example `Ratty.m`.

```
% Ratty.m
%
% tracks the x and y position of a rat moving in
space over time.
```

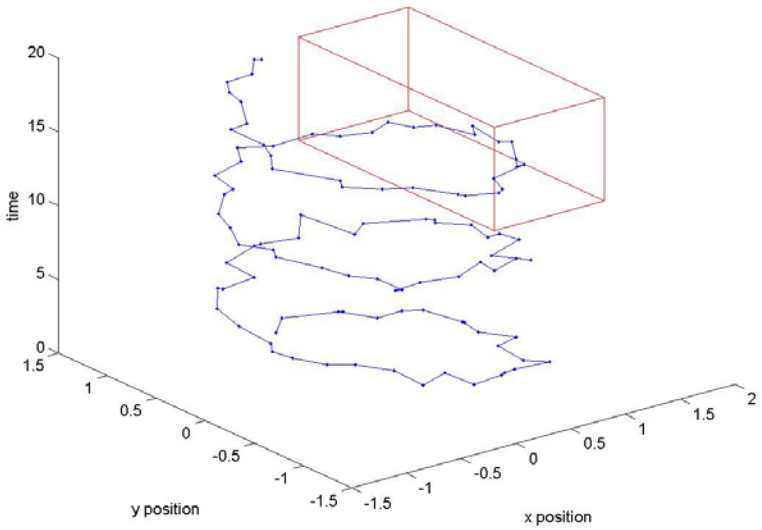
```
clear all
figure(1); clf;
t = linspace(0,6*pi,101);

% fake the data
x = sin(t)+.1*randn(size(t));
y = cos(t)+.1*randn(size(t));

% plot x & y position over time
plot3(x, y, t, '.-');
xlabel('x position');
ylabel('y position');
zlabel('time')
```



This rat is a little confused (are you?). Let's suppose you were interested in seeing whether the rat reached a certain location (food or a swim platform) within a certain space-time window. First let's plot the position of this target on the graph.

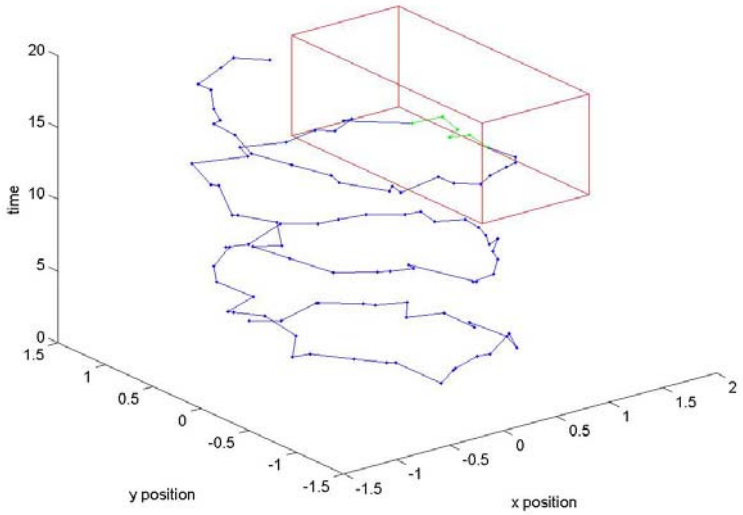
```
twin=[10 17]; % target time is 10-17 seconds
xwin=[0.7 1.7]; % target platform lies between
these x co-ordinates
ywin=[-.5 1.5]; % target platform lies between
```

```
these y co-ordinates


edgeoftarget=[1 2 1 1 1 1; ...
  1 2  1 1  2 2; ...
  1 2  2 2  1 1 ; ...
  1 2  2 2  2 2; ...
  1 1  1 2  1 1; ...
  1 1  1 2  2 2; ...
  2 2  1 2  1 1 ; ...
  2 2  1 2  2 2; ...
  1 1  1 1  1 2; ...
  1 1  2 2  1 2; ...
  2 2  1 1  1 2; ...
  2 2  2 2  1 2];


for i=1:size(edgeoftarget, 1)
 lh=line([xwin(edgeoftarget(i, 1))
xwin(edgeoftarget(i, 2))], ...
  [ywin(edgeoftarget(i, 3)) ywin(edgeoftarget(i,
4))], ...
  [twin(edgeoftarget(i, 5)) twin(edgeoftarget(i,
6))]);
 set(lh, 'Color', 'r');
end


ind=find(t>=twin(1) & t<twin(2) & x>=xwin(1) &
x<xwin(2) & ...
 y>=ywin(1) & y<ywin(2));
hold on
```



Now re-plot those points in the rat's history when she is inside the target box in green

```
plot3(x(ind), y(ind), t(ind), 'g.-');
```



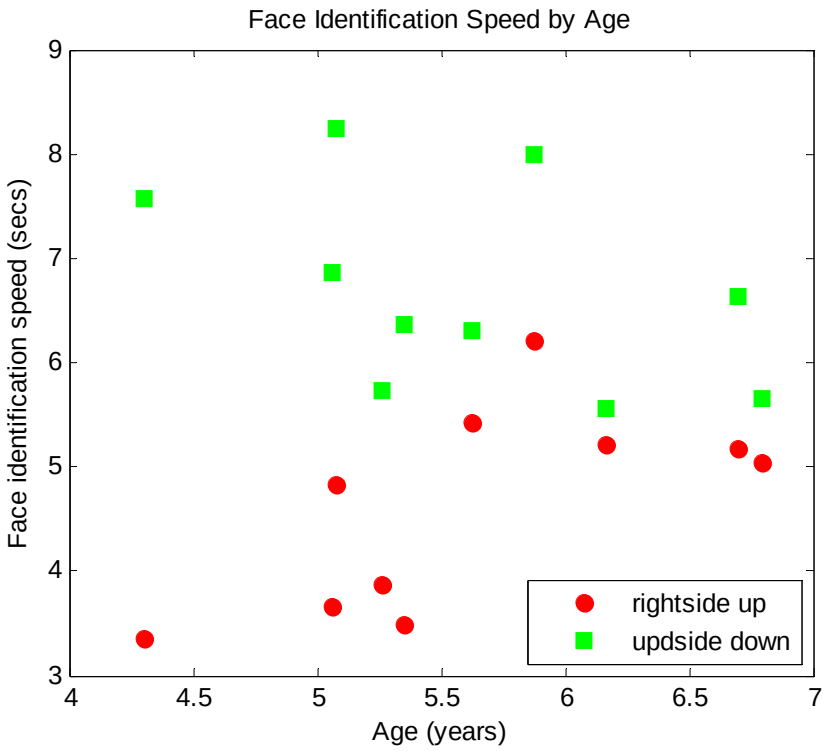
Questions for Chapter 9

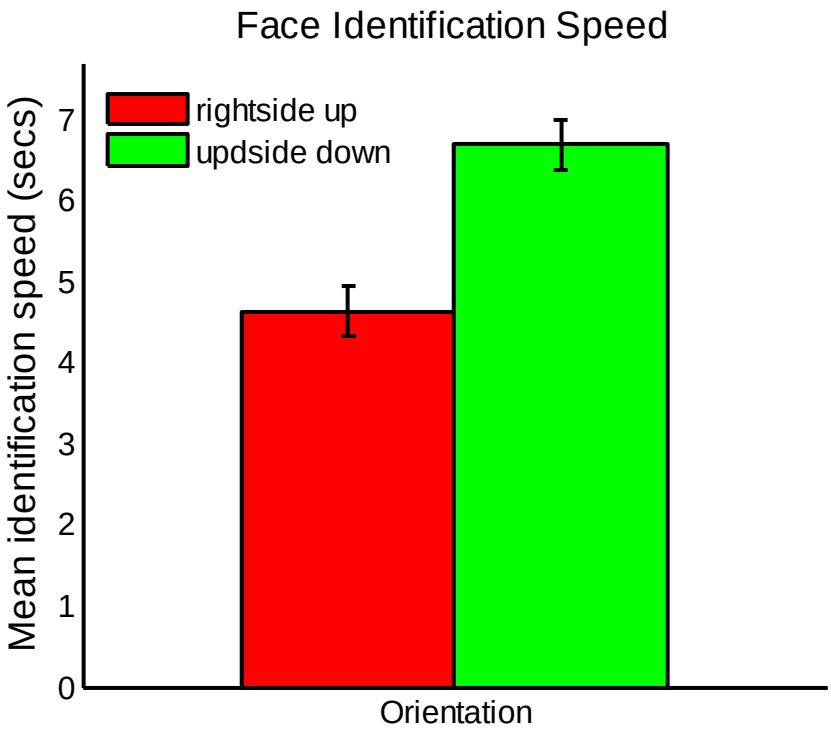
Q 9.1 Create a scatter plot from either faked data or real data

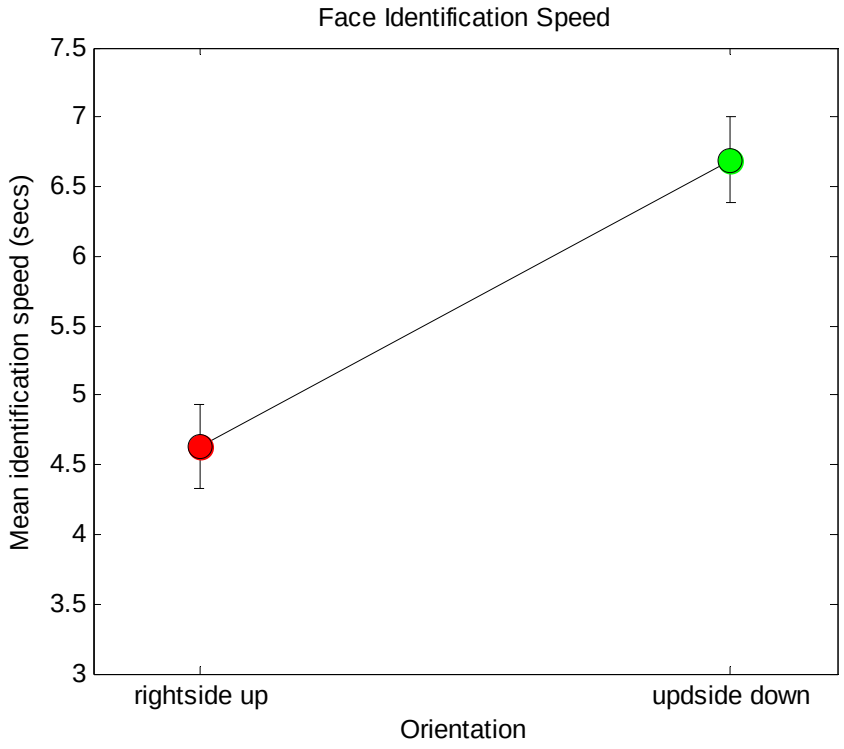
The following code creates some fake data for reaction times to identifying faces for young subjects of various ages. RTup represents reaction times for right side up faces and RTdown is for upside down faces.

```
randn('seed',0);
age=5+randn(10, 1);
RTdown=7+randn(10, 1);
RTup=age-1-randn(10,1);
```

Make the following figures from the fake data. Error bars should represent the standard error of the mean collapsed across age. (Standard error of the mean is the standard deviation divided by the square root of the number of subjects, which is 10 in this case).


Face Identification Speed by Age
Face identification speed (secs)
Age (years)
9
8
7
6
5
4
3
4
4.5
5
5.5
6
6.5
7
rightside up
updside down


Face Identification Speed
rightside up
updside down
Mean identification speed (secs)
0
1
2
3
4
5
6
7
Orientation



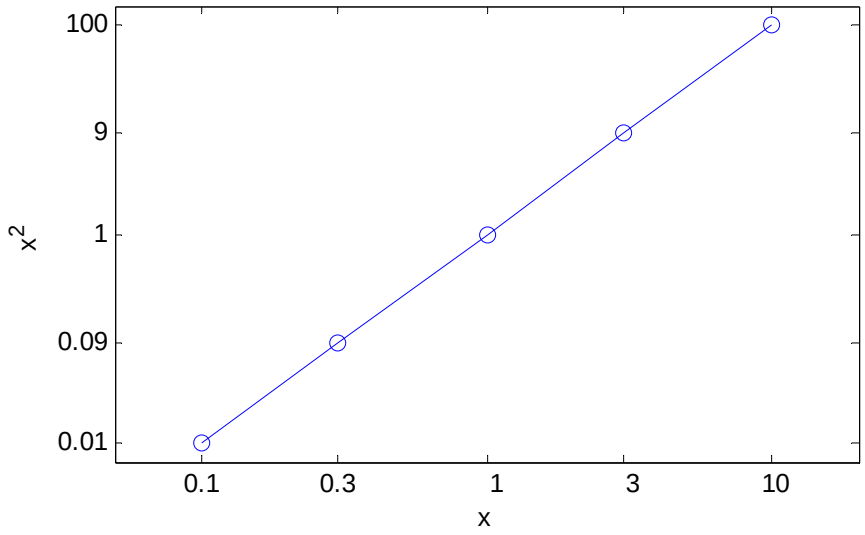
Q 9.2 logy2raw and log-log plots

a) Edit the function `logx2raw` to create the function `logy2raw` (it's as easy as you think).

b) Set up the x and y values so that y is equal to x^2, where x is:

```
x = [.1, .3, 1, 3, 10];
```

Plot y as a function of x on a log-log axis. Use logx2raw and logy2raw to generate the graph below:


100
9
1
0.09
0.01
x^2
0.1
0.3
1
3
10
x

CHAPTER 10: DATA IN, DATA OUT

Here we are going to show you how to load and save data in three file formats – a native Matlab "mat" file, an Excel spreadsheet and a text file.

In some cases you can just use the Import Data function by going to the command window and choosing File->Import Data. But sometimes when you are reading in a lot of files it's more convenient to program this instead of doing it manually.

10.1 .mat format

The traditional way to save data in Matlab is in something called a mat file, which will show up with the extension *.mat*.

To save variables you simply need to specify the filename you want to save under, and the variables you want to save. If you don't specify which variables you want to save Matlab will save all variables that are currently in the workspace. By default, Matlab saves in the current directory.

```
clear all
x=1:10; y=10:-1:1; z=x(1)+y(4);
save mymatfile x y
disp(['variables saved in directory ', pwd])
variables saved in directory C:\Program
Files\MATLAB\R2008a
```

You can reload these variables using the command `load`.

```
clear all
who
```

```
load mymatfile
who
Your variables are:
x y
```

Here's another way to use `save` and `load` that is useful when the name you want to save your variables under is defined as a Matlab string.

```
filename='mysecondmatfile';
save(filename, 'x');

clear all
filename='mysecondmatfile';
load(filename);
who
 Your variables are:
filename x
```

You can also specify which variable you want to load from a mat file. Here, we specifying that we only want to load the variable y.

```
clear all
load mymatfile x
who
Your variables are:
y
```

The bug that took me a week to figure out …

It begins with saving some data. Suppose x is really what you were interested in saving, but you didn't specify that you only wanted `x` when you saved the mat file. Maybe `y` was just some junk variable hanging about in memory …

```
y=rand(3);
x=rand(10, 1);
save('myvar');
```

Then later you loaded `myvar`. You never noticed that `y` was hanging about when you saved `myvar` so it's easy to be completely baffled by why `y` is mysteriously metamorphosing inside the loop for no apparent reason.

```
z=rand(10, 1)
y=rand(10, 1);
load('myvar')
x+y+z
```

The way to avoid this is to specify which variable you want to save and load when dealing with mat files, for example:

```
save('myvar','x');
```

10.2 Reading in and out of Excel

The ability to read data in and of excel is currently very limited on Macs. Basically your only option is to read in entire sheets, not subsets as described here.

Here's some example code `MatlabExcelFile.m` showing how to write and read data into Excel for Windows. This code won't work on a

Mac.

```
% MatlabExcelFile.m
%
% Example of how to read and write data into
Excel


clear all
xlsfilename='MatlabExcelFile.xls';
for d=1:4
 sheetname=['Day ', num2str(d)];
 xlswrite(xlsfilename, {'Subject', 'Resp 1',
'Resp2', 'Resp3'}, sheetname, 'a1:d1')
 xlswrite(xlsfilename, [1:18]', sheetname,
'a2:a19');
 xlswrite(xlsfilename, rand(18, 3), sheetname,
'b2:d19');
end
disp(['Excel spreadsheet created in ', pwd]);
```

The first argument to `xlswrite` is the name of the Excel file you want to write into (or create and write into), the second argument is the data you want to write, the third (optional) argument is the same of the Excel worksheet you want to write into and the last argument is a list of the cells into which the data should be written. You should now be able to find a file called MatlabExcelFile.xls which contains that data.

BEWARE – `xlswrite` will quite happily write over data. When dealing with real data, make sure you have backed it up.

Now let's read Excel data into Matlab using `xlsread`:

```
for d=1:4
 sheetname=['Day ', num2str(d)];
 [junk, datainfo]=xlsread(xlsfilename,sheetname,
'a1:d1');
```

```
 subnum=xlsread(xlsfilename, sheetname,
'a2:a19');
 data=xlsread(xlsfilename, sheetname, 'b2:d19');
end

who
Your variables are:
d data datainfo junk sheetname subnum xlsfilename
```

`xlsread` is generally pretty straightforward when reading in numeric values. When reading in text, or a combination of numeric values and text things are a little more complicated. The first argument returned from xlsread is any numeric data contained within the range of cells syou are reading from. The second argument is any text. So, in the line `[junk, datainfo]=xlsread(xlsfilename,sheetname, 'a1:d1');` because we are reading a set of text, `junk` will be empty and it is the second argument, datainfo, that will contain the text information.

10.3 Reading in and out of a text file

This program MatlabTextFile.m writes and reads data into a standard text file.

```
% MatlabTextFile.m
%
% Writes and reads data from a text file.

clear all

txtfilename='MatlabTextFile.txt';
colStr={'Subject', 'Resp1', 'Resp2', 'Resp3'};
nSub=18;

% open file
```

```
fp=fopen(txtfilename, 'wt');
```

We begin by opening the text file to be read, and returning a file pointer to that file. In this case we specify that we want to *read* to the file using `'r'`. Other commonly used options include writing to a text file `'wt'` or appending data to a file `'a'`. doc fopen will give you a list of all the possible options. Some confusion occurs if you just use 'w' to write to a file instead of 'wt'. 'w' alone writes to a binary file and not a text file, so things like return characters don't work when you view the file with some text editors.

We then check to see that the file opened successfully. There are lots of reasons a file may not open, for example, if you already have that file open using a text editor, or the file isn't in your Matlab path.

```
if fp==-1
 disp('sorry could not open file')
else
 for d=1:4
  fprintf(fp, '%s\n', ['Day ', num2str(d)]);

  for i=1:length(colStr)
   fprintf(fp,'%s\t', colStr{i});
  end
  fprintf(fp,'\n');
  for i=1:nSub
   fprintf(fp, '%d\t', i);
   fprintf(fp, '%.3f', rand(1, 3));
   fprintf(fp,'\n');
  end
 end
end
disp(['Text spreadsheet created in ', pwd]);
fclose(fp);
```

`fprintf` takes in the following arguments:

The file pointer (created by `fopen`) specifying the file you want to read from.

The type of data you want to write into from the file (`‘%s’` = string, `‘%d’` = integer. `‘%f’` means double, use `doc` fprintf for more options). When writing a double you can either write using `'%f'` or specify the precision and after the decimal place: `'%.3f'` will round data to 3 decimal places.

An optional argument describing how many pieces of data you want to write.

Remember to close the file at the end of the program. If your program crashes before the end you would be wise to call `fclose(fp)` in the command line before re-running the program.

`'\t'` adds a tab, `'\n'` adds a new line.

Now let's read the data back in again.

```
fp=fopen(txtfilename, 'r');

if fp==-1
 disp('sorry could not open file')
else
 for d=1:4
  dayStr{d}=fscanf(fp, '%s', 2);
  % Note how we read two strings into a single
cell

  for i=1:4
   colStr{i}=fscanf(fp,'%s', 1);
```

```
  end

  for i=1:nSub
   subNum(i)=fscanf(fp, '%d', 1);
   respVals(i, :)=fscanf(fp, '%f', 3);
   % Note how we read three floats in at a time
  end
 end
end
fclose(fp);
```

Working with `fscanf` tends to be a little fiddly since if you read the wrong thing in early, you get out of sync with the file, and every variable read in after that point gets messed up. If you try to read a float as if it's a string you will end up with nonsense. One trick for debugging is to make sure you close your file and clear all the read in variables (e.g. `dayStr`, colStr, `respVals`). Then go through the program line by line, checking that the right variable is being read in.

Questions for Chapter 10

Q 10.1 Reading and writing Excel files

a) Create an empty 1x2 cell array using the command:

```
dataCell = cell(4,2);
```

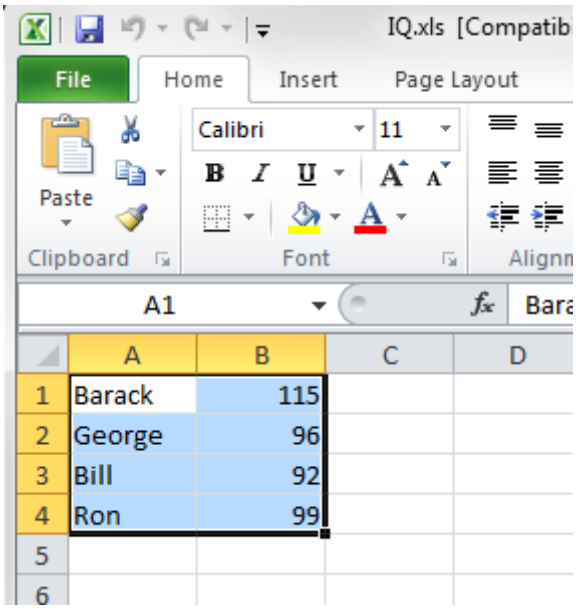
b) Fill the first column with the following names: Barack, George, Bill, Ron

c) Fill the second column of `dataCell` with four integers generated with the following lines:

```
randn('seed',1);
IQ = round(100+15*randn(4,1));
```

(See Hint.)

d) Use `xlswrite` to save `dataCell` as an excel spreadsheet called 'IQ'. If you open the spreadsheet, it should look something like this:



e) Add a fifth row in the spreadsheet containing the name 'Jimmy' and any value you'd like, and save it.

f) Load the modified spreadsheet back in to Matlab using `xlsread` and see if the new row has been added. Note, that `xlsread` can return up to three variables. The first contain numbers, the second strings, and the third is a cell array containing everything.

CHAPTER 11: IMAGES AND MOVIES

11.1 image – true color and colormaps

Let's begin by reading in and displaying an image.

```
clear all;
close all;
rgb = imread('ngc6543a.jpg');
```

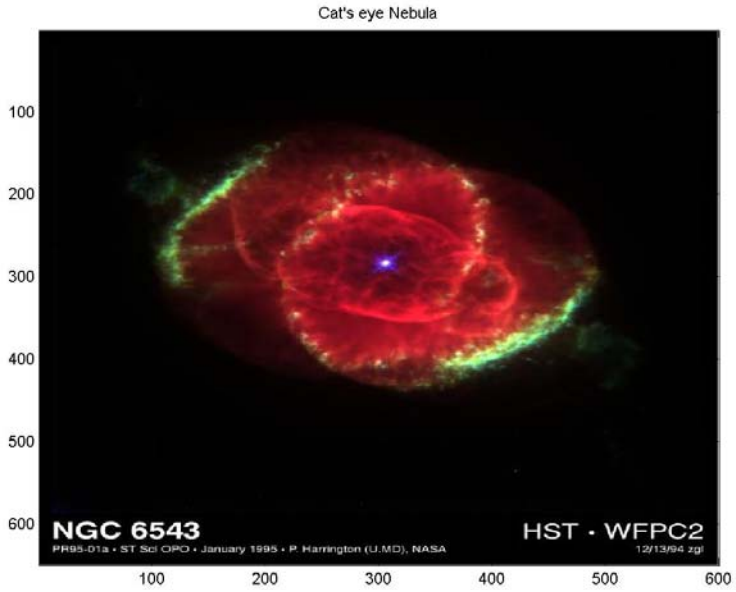
`imread` loads an image file. In this case it's reading in an example image that Matlab has stored called ngc6543a.jpg.

Sometimes you may need to also specify the file type that you want to read if it doesn't have an extension.

```
rgb=imread('ngc6543a', 'jpg');
```

Note that rgb is a 3D matrix, 650 x 300 x 3, with the 3rd dimension representing the red green and blue monitor values. Let's now image this matrix.

```
figure(1);
image(rgb);
title('Cat''s eye Nebula');
```



Note how to create the title *Cat's eye Nebula* we used two single quotes to produce a single quote on the title, this is to get around the issue that a single quote is used to mark the end of a string.

When we previously used the command `image` (in Chapter 5), the input into `image` was a 2D matrix, where each value represented an entry into a colormap that specified the monitor RGB values. This time we are directly specifying the RGB values. This method of representing images is called *true color*.

Images can be stored using either true color or the colormap format.

If you want to read in an image that is stored using the colormap convention, it is done as follows:

```
[img, map]=imread('nofile.tiff');
```

In this case `img` will be a 2D matrix, and `map` will be an m x 3 array containing the colormap. Some file formats, like tiff can contain either colormap or true color representation of data. You really need to just

load it in, and look to see whether the image is 2D or 3D. If it's 2D you'll also need to read in the colormap.

One other thing you need to know about `image` is that it will represent images differently depending on whether the image matrix is stored as doubles or as uint8s (images are often saved as uint8 in order to save space). If you are having trouble using image after reading in a stored jpg or tiff image it's worth looking to see whether your image is actually in uint8 format.

The following table explains the options.

	double	*uint8*
colormap	Image matrix must be 2D and contain values in the range of 1:length(colormap) (usually 1:256). The colormap must be an m x 3 array with values that range between 0-1.	Image matrix must be 2D and contain values in the range of 0:255. The colormap must be an m x 3 array with values that range between 0-1.
true color	Image matrix must be 3D (p x q x 3) with values that range between 0-1	Image matrix must be 3D (p x q x 3) with values that range between 0-255.

11.2 Saving images

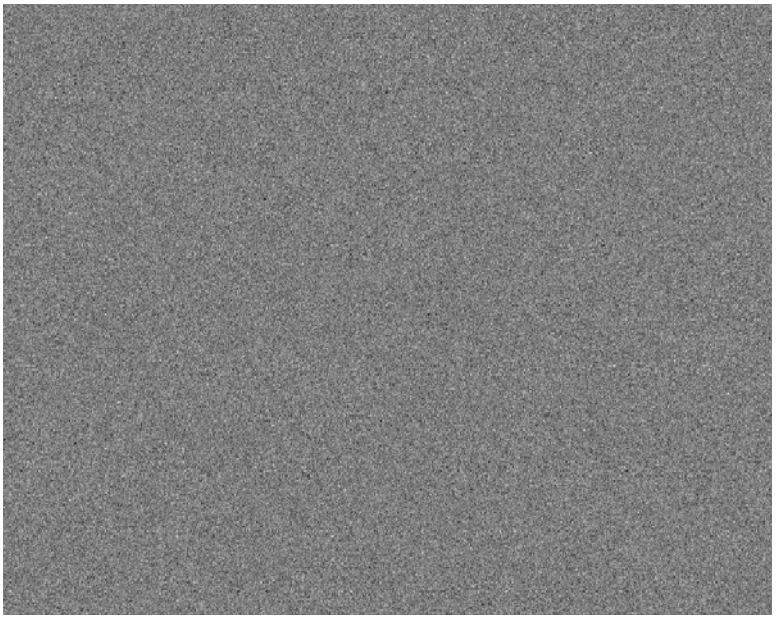
You can also write either colormap or true color images to a graphics file. Here, in `NoiseImages.m` we are going to create and save some random noise images. First we will create and save grayscale noise, using both colormap and true color techniques. Then we will create and save colored noise using just the true color technique (think about why

we can't use the colormap technique to create colored noise). Note that we are using that useful little function `scaleMat` that you created earlier in Chapter 8.

First we're going to create and save an image using the colormap technique.

```
% NoiseImages.m
%
% creates and saves random noise using a
combination of
% colormap and true color techniques.


% grayscale colormap
img=scaleMat(randn(600, 800), [1 256]); % image
must range between 1-255
map=repmat(linspace(0, 1, 256)', 1, 3);
image(img);
colormap(map);
axis off
% gets rid of the axis labels
imwrite(img, map, 'grayscaleImg_cmap.tiff',
'tiff');
disp(['image file saved in Dir ', pwd]);
```

```
image file saved in Dir C:\Users\Ione
Fine\Documents\ Matlab book 2013\matlab code and
figures 2013
```



Now we're going to do the same thing using true color.

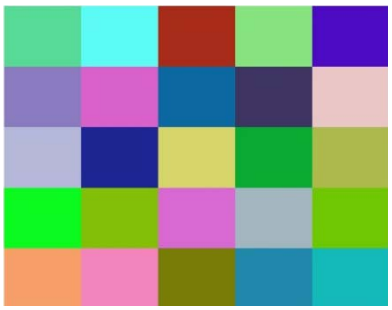
```
img=scaleMat(randn(600, 800), [0 1]); % images
must range between 0-1
rgb=repmat(img, [1, 1,3]); % make image 3D
image(rgb);
imwrite(img, 'grayscaleImg_truecolor.tiff',
'tiff');
disp(['image file saved in Dir ', pwd]);
```

The figure will look identical. But if you look at the two tiff files you will notice they are different in size because they are saving different sets of numbers. The true color tiff file is much larger because it needs to save 600 x800 x 3 = 1, 440, 000 values whereas the colormap version only needs to save (600 x 800)+(256 x3) = 480,768 values. This smaller size comes at the cost of not being able to represent colors quite so accurately since only 256 unique colors can be represented using the colormap technique.

11.3 Printing figures

Images can also be printed to either a physical printer or a graphics file as follows. Here we are saving to a jpeg image, but the print command without the second argument `'djpeg'` would send the figure to your computers' default printer.

```
filenames={'img 1', 'img 2', 'img3'};
for i=1:3
 image(rand(5, 5, 3));
 axis off
 print(filenames{i}, '-djpeg');
end
```



11.4 Movies

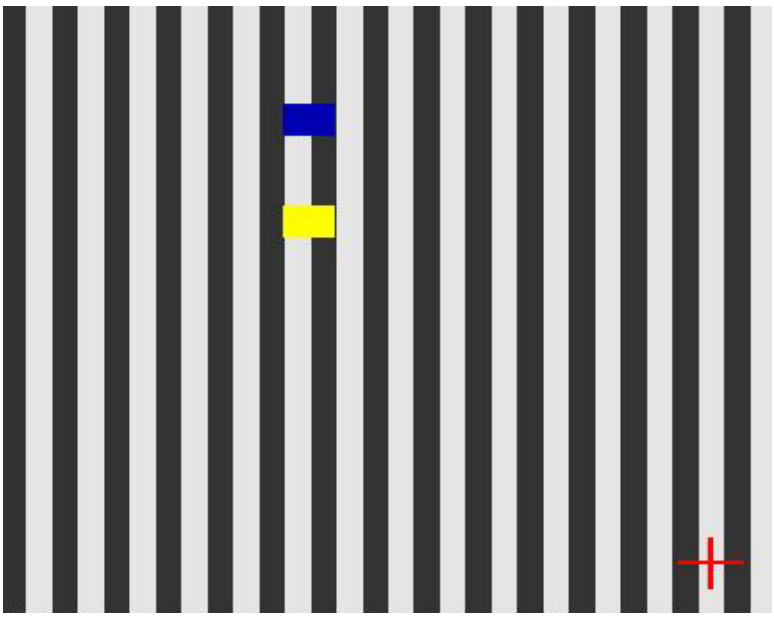
Now let's create and save a movie. Matlab has a movie function, but it only allows you to play movies in Matlab, so it's usually best to carry out one additional step and save the file as an avi. There's lots of cheap software out there (e.g. Quicktime) which you can use to convert avi to any other file type you like. In Footsteps.m we'll create a rather funky illusion discovered by Stuart Anstis of UCSD. The illusion is most compelling if you look at the red cross.

```
% Footsteps
%
```

```
% Fixate on the red cross and observe the
% blue and yellow patches.
% Whenever the grid is visible,
% the boxes seem to step out of phase,
% while in reality their movement is always
parallel.
%
% Anstis SM (2003) Moving objects appear to
% slow down at low contrasts.
% Neural Netw 16:933-938;
% Anstis SM (2004) Factors affecting footsteps.
% Vis Res 44:2171-2178;
% Thompson P (1982) Perceived rate of movement
% depends on contrast. Vis Res 22:377-380


writerObj = VideoWriter('footsteps.avi');
open(writerObj);


% timing parameters
tSteps=500; % must be less than imgSize(1)-
(3*stripeWidth)
altRate=200;
% alternate between gray and stripes every 200
frames
altProp=.75; % proportion of the time use the
striped background


% background
imgSz=[600, 600];
blank=ones(imgSz); % create the gray background
stripeWidth=20;% create the background stripes
stripes=repmat([zeros(imgSz(1), stripeWidth),
ones(imgSz(1), stripeWidth)], ...
 1, imgSz(2)/(2*stripeWidth));
stripes=stripes+2;
```

```
% add fixation cross
stripes(549:551, 525:575)=6;
stripes(525:575, 549:551)=6;
blank(549:551, 525:575)=6;
blank(525:575, 549:551)=6;


% determine position and size of the blue and
yellow squares
boxWidth = 2*stripeWidth;
boxHeight = 30;
% Top, Bottom, Left, Right
bpos=[100 100+boxHeight 20 20+boxWidth];
ypos=[200 200+boxHeight 20 20+boxWidth];


% colormap
cmap=[ .5 .5 .5; ... % blank
 .2 .2 .2; ... % stripes
 .9 .9 .9; ... % stripes
 .0 0 .7; ... % blue box
 1 1 0; ... % yellow box
 1 0 0 ]; % fixation cross color


for t=1:tSteps
 % recreate the background
 if mod(t, altRate)<(altRate*altProp)
  img=stripes;
 else
  img=blank;
 end
 % reposition the boxes
 img(bpos(1):bpos(2), bpos(3):bpos(4))=4;
 img(ypos(1):ypos(2), ypos(3):ypos(4))=5;
 bpos(3:4)=bpos(3:4)+1;
 ypos(3:4)=ypos(3:4)+1;
```

```
 %image, and save a movie frame
 image(img);
 colormap(cmap);
 axis off
 F=getframe;
 writeVideo(writerObj,F);
end
close all
close(writerObj);
```



We begin by creating a video object (it's like a structure, but it has a very specific set of fields), using the commands:

```
writerObj = VideoWriter('newfootsteps.avi');
open(writerObj);
```

This video object has a variety of properties or fields:

```
writerObj
```

```
writerObj =
 VideoWriter

 General Properties:
  Filename:     'newfootsteps.avi'
  Path: 'C:\Users\Ione Fine\Documents\ Matlab
book 2013\matlab code and figures 2013'
  FileFormat:    'avi'
  Duration:     6.6333

 Video Properties:

  ColorChannels:   3
  Height:      281
  Width:       338
  FrameCount:    199
  FrameRate:     30
  VideoBitsPerPixel: 24
  VideoFormat:    'RGB24'
  VideoCompressionMethod: 'Motion JPEG'
  Quality:      75
```

On each timestep we save the figure window as a movie frame into the video object using the command `getframe`. `F` is a structure containing two fields, `cdata` (the matrix image) and `colormap`.

```
F
F =
1x500 struct array with fields:
 cdata
 colormap
```

In this example the video object is an avi file. But you can also save in other formats such as Motion JPEG 2000 files and MPEG-4.

11.5 DoDaFunky

```
% Do da funky
% a program that puts together bits and
% pieces of what you've learned to make a fun
% movie
%
% IF 2/2013


cmap_str={'cool', 'spring', 'summer', 'autumn'};
funkystr='Do da FUNKY!!';


% open the video object
writerObj = VideoWriter('dodafunky.avi');
open(writerObj);


% create spiral, see chapter 6
n=100;
[X,Y] = meshgrid(linspace(-pi,pi,n));
rad = sqrt(X.^2+Y.^2);
ang = atan2(Y,X);
spiral{1}.img = sin(2*pi*rad+ang);
spiral{2}.img = sin(2*pi*-rad+ang);


posx=0;
for ff=1:length(funkystr)
 f=figure(1); clf;
 set(f, 'Color', rand(3, 1));


 % background image
 % choose a spiral randomly
 imagesc(spiral{round(rand)+1}.img); hold on
 axis off; axis equal
 % choose a colormap randomly, see Chapter 3
```

```
 colormap(cmap_str{mod(ff, length(cmap_str))+1});

 % put up text in a nested loop.
 posx=0;
 for f=1:ff
  posx=posx+(n/15)+(randn*2);
  h=text(posx, (n/2)+randn*2, funkystr(f));
  set(h, 'FontSize', 27, 'FontName', 'Showcard
Gothic', 'Color', rand(1, 3));
  drawnow
 end
 F=getframe;
  for f=1:5
% controls how fast the movie plays by adding
multiple frames
% of the same image
  writeVideo(writerObj,F);
 end
end


% close everything
close all
close(writerObj);
```

Questions for Chapter 11

Q 11.1 DoDaFunky

Create your own version of DoDaFunky.m

(For the class we teach, this is usually the most fun assignment. Submissions range from minor modifications to some very clever animations. We'd love to see some of your ideas!

CHAPTER 12: MENU BARS, BUTTON BOXES ETC.

Sometimes we want our subjects (or lab assistants) to be able to interact with our program without having to deal with the code itself. The way to do this is via user interfaces. These are called `uis` (user interfaces) and `guis` (graphical user interfaces).

12.1 input and ginput

The simplest way of getting user input is via the command window and the commands `input` and `ginput`. (These are `uis` not `guis`, since they don't bring up figure windows.)

```
reply = input('How old are you?: ');
if isempty(reply)
  reply = NaN;
end
```

This will print out the text `'How old are you?: '` into the command window and will wait for a response. If the subject just hits enter, then a `NaN` is put into the variable reply. Otherwise the variable reply is the number the person types in.

If you want to ask subjects to input a string or character, you need to specify that the response will be a string.

```
reply = input('Do you want to run another trial?
Y/N [Y]?: ', 's');
if isempty(reply)
  reply = 'Y';
end
```

This will print out the text `'Do you want to run another trial? Y/N [Y]: '` into the command window and will wait for a response. If the subject just hits enter, then input returns an empty matrix [ ] in the variable reply, and you specify in the code that if reply is empty then you want to assume that the subject would like another trial.

12.2 ginput

This reports where the subject clicks the mouse in a figure window. So you might use to select which of 4 colors the subjects thinks is prettiest, as in the code below.

```
cmap=[1 0 0; .9 .1 0; .8 .2 0; .7 .3 0];
img=[1 2 ; 3 4]
image(img); colormap(cmap);
[r, c]=ginput(1);
colorval=round(r)+(2*(round(c)-1));
colorval
```

Note that the returned values from `ginput` are in terms of the *figure* co-ordinates, not pixels on the screen.

12.3 predefined dialog boxes

It's also possible to do pop-up windows (graphical user interfaces, GUIs) in Matlab. Certain kinds of stereotypes dialog boxes can be made pretty easily – these are called predefined dialog boxes (look at Matlab documentation for a full list of these).

errordlg

This puts up an error box. The first argument is the error statement, the second is the title of the error box.

```
errordlg('File not found', 'A pointless error
box');
```

questdlg

This puts up a question box. The first argument is the question statement, the second is the title of the question box, the following strings are the possible answers, and the default answer. The command `questdlg` returns the string that was the selected choice. So in this example `button` will be `'No'`, `'Sort of'` or `'Yes'` depending on what the user chooses.

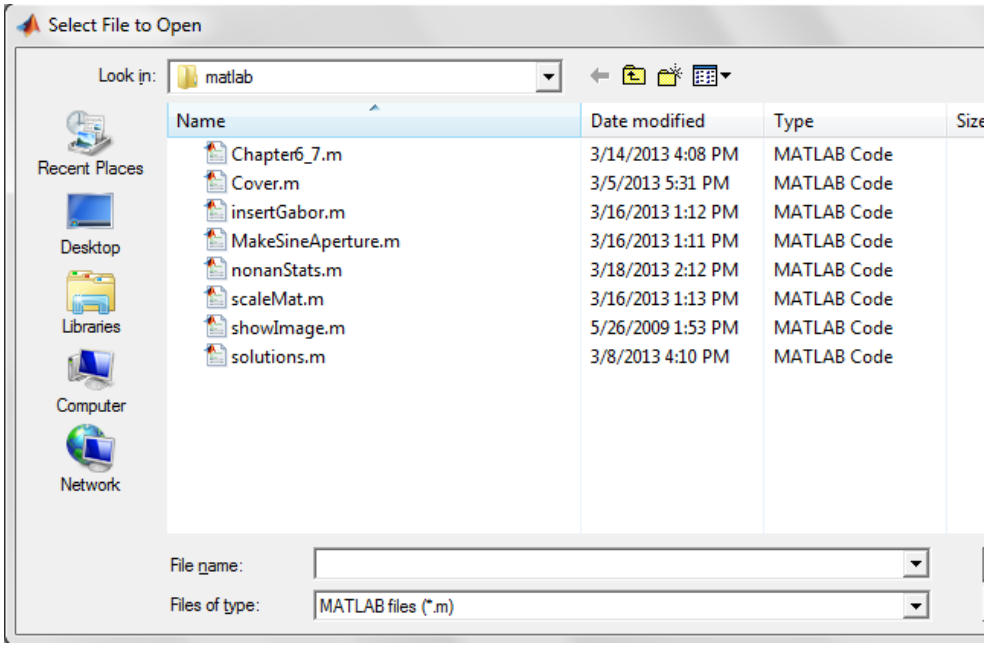
```
qstring = 'Are you having fun yet?'
titlestr = 'pointless questdlg box'
str1 = 'No'; str2='Sort of'; str3 = 'Yes';
default=str2;
button =
questdlg(qstring,titlestr,str1,str2,str3,default)
;
```

uigetfile

This is a useful ui function that opens up a standard file navigator window so you can select a file. For example, if you want the user to choose a Matlab file ending with '.m' in the current directory you can do this:

```
[FileName, PathName] = uigetfile('*.m')
```

The command will open a window that will look something like this:



The `'*.m'` restricts the list of files to those that end with `'*.m'`. Those familiar with unix will recognize the asterisk (*) as a wildcard which tells Matlab to list all files with .m as part of their filename. Selecting mat files can be done with `'*.mat'`. You can also select files that begin a certain way e.g. `'mydatafile*.mat'`.

If you want to allow for more than one type of file, you can make a list separated by semicolons. For example, if you want to filter for both 'xls' and 'xlsx' files, you can do this:

```
[FileName, PathName] = uigetfile('*.xls;*.xlsx')
```

Note that `uigetfile` doesn't actually open a file but instead returns a file name and a path to that file. You can then use that information to actually open the file.

12. 4 uicontrol

Matlab's `uicontrol` function is a generic function that can make a huge variety of `guis`. It returns a handle to a graphical user interface object, which can be have one of a variety of styles, such as a slider, checkbox or popupmenu. To see all of the properties that you can control with `uicontrol`, open up a `uicontrol` and use 'set':

```
h = uicontrol;
set(h)
```

```
    BackgroundColor
    Callback: string -or- function handle -or-
cell array
    CData
    Enable: [ {on} | off | inactive ]
    FontAngle: [ {normal} | italic | oblique ]
    FontName
    FontSize
    FontUnits: [ inches | centimeters |
normalized | {points} | pixels ]
    FontWeight: [ light | {normal} | demi | bold
]
    ForegroundColor
    HorizontalAlignment: [ left | {center} |
right ]
    KeyPressFcn: string -or- function handle -or-
cell array
    ListboxTop
    Max
    Min
    Position
    String
    Style: [ {pushbutton} | togglebutton |
radiobutton | checkbox | edit | text | slider |
frame | listbox | popupmenu ]
    SliderStep
    TooltipString
    Units: [ inches | centimeters | normalized |
```

```
points | {pixels} | characters ]
   Value

   ButtonDownFcn: string -or- function handle -
or- cell array
   Children
   Clipping: [ {on} | off ]
   CreateFcn: string -or- function handle -or-
cell array
   DeleteFcn: string -or- function handle -or-
cell array
   BusyAction: [ {queue} | cancel ]
   HandleVisibility: [ {on} | callback | off ]
   HitTest: [ {on} | off ]
   Interruptible: [ {on} | off ]
   Parent
   Selected: [ on | off ]
   SelectionHighlight: [ {on} | off ]
   Tag
   UIContextMenu
   UserData
   Visible: [ {on} | off ]
```

The most important properties are the `Style`, `Position` and `Callback`. As for other handles, you can either set the properties when you define the handle with `uicontrol`, or you can set properties using the handle afterward. Here's an example of a slider that sets the variable `x` to the current value on the slider whenever the slider is moved:

```
figure(1)
clf
h =
uicontrol('Style','Slider','Position',[20,20,100,
20]);
set(h,'Min',0,'Max',100);
set(h,'CallBack','x = get(h,''Value'')');
```

This defines a slider UI in the bottom left corner of the figure (20 pixels up and 20 pixels over, 100 pixels wide and 20 pixels tall). The lower and upper values of the slider are set to 0 and 100.

We'll label the slider above it with a uicontrol that's just contains text:

```
uicontrol('Style','Text','Position',[20,40,100,20
],'String','Set x by slider');
```

The bottom corner of your figure should look like this:

Adjust the slider and look at the command window. You should see stuff like this:

```
x =
   70
x =
   60
```

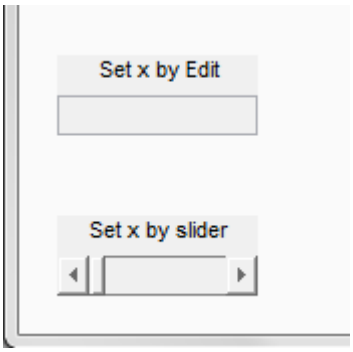
What's happening here is that every time you adjust the slider, the command defined in the `'CallBack'` property is executed. In this case, it sets the variable `x` to the current value of the slider, which is obtained with the `get` command `get(h,'Value')` The double quotes in the actual command are to deal with the fact that you're inserting a string into a string. (We've deliberately left off the semicolon on the callback string so it dumps the output to the screen. You can add the semi-colon so that it'll set the variable `x` without spitting out to the command window).

Now let's add a new uicontrol that lets the user set the variable `x` with

an ‘Edit’ box:

```
hEdit =
uicontrol('Style','Edit','Position',[20,100,100,2
0]);
set(hEdit,'CallBack','x =
str2num(get(hEdit,''String''))');

uicontrol('Style','Text','Position',[20,120,100,2
0],'String','My Editor');
```

The bottom of the figure should look like this:



The callback for this UI is a little different. The text inside the edit UI can be obtained using the `get` command for the ‘`String`’ property. This returns a string, not a number. To convert it to a number we use Matlab’s `str2num` command which does exactly what you think it should do. Try typing numbers in to the editor and hit Enter. The variable `x` should be redefined.

Here’s something kind of cool: str2num can convert strings containing mathematical expressions as well as numbers. Try this in the command line:

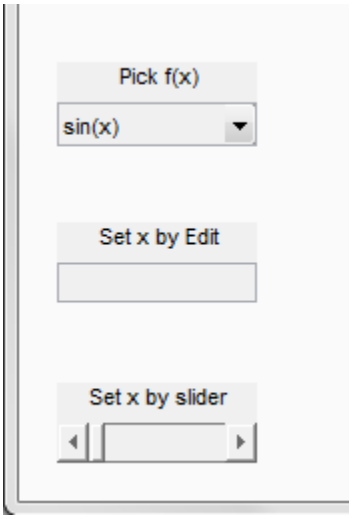
```
str2num('exp(1)')
ans =
  2.7183
```

This means that you can type any mathematical expression in your new editor box and x will be set to the evaluated expression. You've just built a little calculator!

Let's add one more UI that is a pull down menu that performs some function on the value of x:

```
functionsStr =
{'sin(x)','cos(x)','exp(x)','log(x)','exp(i*x)','
mod(x,pi)'};

hPulldown =
uicontrol('Style','popupmenu','Position',[20,180,
100,20],'String',functionsStr);
set(hPulldown,'Callback','eval(functionsStr{get(h
Pulldown,''Value'')})');
uicontrol('Style','Text','Position',[20,200,100,2
0],'String','Pick f(x)');
```

Your figure should look like this:



The `'String'` field is a cell array of strings. Check out the

`'CallBack'` command. It uses the `get` command to pull out the value of the pull down menu. This provides an integer value that is an index into the list of options. The string corresponding to this member of the list is then pulled out of the cell array and evaluated.

Try it out. Pull down one of the functions and watch it execute that string.

This is a lot for one callback string to achieve. For complicated callbacks, it is recommended that you write your own function or script that can run a series of commands.

For even more stuff on uicontrol, you can download an example called `CosmoQuiz.m` from our website at www.finelab.org. But beware. GUIs can eat up a lot of programming time very quickly. Don't get obsessed with them.

CHAPTER 13: CONCLUSION

Congratulations for reaching the end of this book (assuming you didn't just skip to here). Please email us (ionefine@uw.edu, gboynton@uw.edu) if you find errors or have any suggestions.

As you may have noticed, lots of topics weren't covered in this book for reasons of space/exhaustion. We do plan to cover some topics in 2 additional Kindle books that should be coming out shortly. If you've got suggestions for additional topics you would like us to include, do let us know.

`Happy Programming!`

Ione & Geoff

Stay tuned for these exciting upcoming titles from the same authors …

Matlab for the behavioral sciences: Book II. Stimulus generation and presentation

Topics will include:

- Psychtoolbox
- Sound stimuli
- Collecting keypresses
- Timing
- Calibration

Matlab for the behavioral sciences: Book III. Advanced analysis methods

Topics will include:

- Fitting Psychometric functions
- Bootstrap analyses
- Signal detection theory and ROC curves
- Linear filters and deconvolution
- 1D and 2D Fourier analysis
- Independent components analysis
- Parallelization

HINTS

Hint for Q4.1

matrix E

If you let `i` correspond to the rows and `j` be the columns, then you can set `M(i,j) = 2` if `i<=j`

matrix F

```
for i=1:5
  (i-1)*5;
end
```

Hint for Q5.2

Hint 1:
How do you combine Z and T to create ZT?

ZT should range from 1-4. One sensible way of making ZT (there are others) is that values in ZT that are 1 should represent positions in the matrix that are black when either 'Z' or 'T' is visible, ZT=2 should represent positions that are only visible in the letter 'Z', ZT=3 should represent positions only visible in 'T' and 4 should represent positions visible for both 'Z' and 'T'.

Hint 2:
If you've created ZT as describe above then look at what happens with this cmap and then modify it to create `cmapT` and `cmapZ`.

```
cmap=[0 0 0 ; 0 1 0 ; 1 .7 .7; 1 .7 .7;];
```

Hint for Q5.3

To make a sinusoidal grayscale color map, let:

```
nCycles = 4;
phase = pi;
```

Set each of the three columns of the colormap matrix to a sinusoid that modulates at the desired number of cycles and phase;

```
for i=1:3
     cmap(:,i) =
(sin(linspace(0,2*pi*nCycles,256)'-phase)+1)/2;
end
```

Sinusoids range from -1 to 1, so adding 1 and dividing by 2 makes the new sinusoid range from 0 to 1.

Hint for Q5.4

a) Change the size of the signal.

c) `cmap=gray(20); cmap(1, :) =[1 0 0]; cmap(20, :)=[ 0 0 1];`

Hint for Q7.2d

You may have guessed that you'll be using the command `cell2mat`. You may also have discovered that `cell2mat(bigCell([1,4]))` gives you a 1x20 element vector. This is because `bigCell([1,4])` is a 1x2 cell array

each containing a 1x10 element vector. `cell2mat` will concatenate these two vectors horizontally creating a 1x20 vector.

The trick is to transpose `bigCell([1,4])` which will create a 2x1 cell array. `cell2mat` will then concatenate the two 1x10 vectors vertically creating the desired 2x10 vector.

Hint for Q10.1c

To put a column of numbers into a cell array you could either use a `for` loop to convert each number into a cell one at a time:

```
dataCell = cell(4,2);
y= [1,2,3,4]';
for i=1:4
   dataCell(i,2) = {y(i)};
end
```

Or you can use the Matlab command `num2cell` which does the same thing all at once:

```
dataCell = cell(4,2);
y= [1,2,3,4]';
dataCell(:,2) = num2cell(y);
```

SOLUTIONS

Chapter 2

Q 2.1

a)

```
str = 'CHEERIOS';
str(1)='O'
str(5)='P'
str =

OHEERIOS


str =

OHEEPIOS
```

b)

```
str = 'CHEMISTRY';
str([1 8])='OB'
str =

OHEMISTBY
```

c)

```
str = 'MACARONI AND CHEESE';
str([3 14 8])='OOR'
str =

MAOARONR AND OHEESE
```

Q 2.2

a)

```
str = 'MACARONI AND CHEESE'
```

```
id1 = [1 2 7 13:19];
str(id1)
str =

MACARONI AND CHEESE


ans =

MAN CHEESE
```

b)

```
id2 = [15 16 5 9 1 8 7 12 9 8 18 9 14 15 8 7 17 18
17];
str(id2)
ans =

HER MIND IS CHINESE
```

Q 2.3

a)

```
linspace(1, 10, 10)
1:1:10
1:10 % the convention is that if you don't specify
the step size it's 1
ans =

     1     2     3     4     5     6     7     8
9    10


ans =

     1     2     3     4     5     6     7     8
9    10


ans =

     1     2     3     4     5     6     7     8
9    10
```

b)

```
linspace(10, 18, 5)
10:2:18
ans =

    10    12    14    16    18


ans =

    10    12    14    16    18
```

c)

```
linspace(19, 15, 5)
19:-1:15
ans =

    19    18    17    16    15


ans =

    19    18    17    16    15
```

d)

```
linspace(10, -4, 8)
10:-2:-4
ans =

    10     8     6     4     2     0    -2    -4


ans =

    10     8     6     4     2     0    -2    -4
```

e)

```
linspace(0, 15.7080, 6)
0:3.1416:15.7080
0:pi:15.7080
0:pi:5*pi % another solution
ans =
```

```
         0     3.1416     6.2832     9.4248    12.5664
15.7080


ans =

         0     3.1416     6.2832     9.4248    12.5664
15.7080


ans =

         0     3.1416     6.2832     9.4248    12.5664
15.7080


ans =

         0     3.1416     6.2832     9.4248    12.5664
15.7080
```

Q 2.4

```
str = 'aaaaaaaaaaaaaaaaaaaa';
```

a)

```
str(3:3:end)='c'
str =

aacaacaacaacaacaacaa
```

b)

```
str(2:3:end)='b'
str =

abcabcabcabcabcabcab
```

c)

```
disp(str(3:3:end))
cccccc
```

d)

```
str(4:6)='def'
str =

abcdefabcabcabcabcab
```

e)

```
str([4:6 10:12 16:18])='defdefdef'
str =

abcdefabcdefabcdefab
```

f)

```
str([6 12 18])
ans =

fff
```

Q 2.5

a)

```
timepts=12:1.23:1000;
timepts=timepts(1:40)
timepts =

  Columns 1 through 7

   12.0000    13.2300    14.4600    15.6900    16.9200
18.1500   19.3800

  Columns 8 through 14

   20.6100    21.8400    23.0700    24.3000    25.5300
26.7600   27.9900

  Columns 15 through 21

   29.2200    30.4500    31.6800    32.9100    34.1400
35.3700   36.6000

  Columns 22 through 28
```

```
   37.8300    39.0600    40.2900    41.5200    42.7500
43.9800   45.2100

  Columns 29 through 35

   46.4400    47.6700    48.9000    50.1300    51.3600
52.5900   53.8200

  Columns 36 through 40

   55.0500   56.2800   57.5100   58.7400   59.9700
```

b)

```
timepts(5)
ans =

   16.9200
```

c)

```
timepts(end)
ans =

   59.9700
```

Q 2.6

```
resp='rerererererererererererererererererererere';
```

a)

```
resp(5)='k';
disp(resp(5))
k
```

b)

```
resp(5)='r'
resp =

rerererererererererererererererererererere
```

c)

```
disp(resp(2:2:end))
eeeeeeeeeeeeeeeeeee
```

Q 2.7

```
vect = 12:-1:1
vect =

    12     11     10      9      8      7      6      5
4     3     2     1


a)
vect(1:12)
ans =

    12     11     10      9      8      7      6      5
4     3     2     1


b)
vect(12:-1:1)
ans =

     1      2      3      4      5      6      7      8
9    10    11    12


c)
vect([10 12 9 12 8 4])
ans =

     3     1     4     1     5     9


d)
vect(1:2)
ans =

    12    11


e)
vect(vect(1:2))
```

```
ans =

     1     2
```

f)

```
vect(vect)
ans =

     1      2      3      4      5      6      7      8
9    10    11    12
```

g)

```
vect(vect(vect))
ans =

    12     11     10      9      8      7      6      5
4     3      2      1
```

Chapter 3

Q 3.1

```
mat=[ 1 2 3 4; 4 5 6 7; 8 9 10 11];
```

a)

```
mat+1
ans =

     2     3     4     5
     5     6     7     8
     9    10    11    12
```

b)

```
10-mat
ans =

     9     8     7     6
     6     5     4     3
     2     1     0    -1
```

c)
```
min(mat)
ans =

     1     2     3     4
```

d)
```
min(mat')
ans =

     1     4     8
```

e)
```
min(min(mat))
ans =

     1
```

Q 3.2

```
v1 =[1 2 3 4];
v2 =[1 0 1 0];
```

a)
```
v1 + v2
ans =

     2     2     4     4
```

b)
```
v1 .* v2
ans =

     1     0     3     0
```

```
%c)
sum(v1.*v2)
ans =

     4
```

```
%d)
v1 * v2'
ans =

     4
```

Q 3.3

```
sum(1:2:99)
ans =

        2500
```

Q 3.4

a)
```
a=1:10000;
```

b)
```
b=a.^2;
```

c)
```
c=6./b;
```

d)
```
d=sum(c);
```

e)
```
e=sqrt(d)
e =

    3.1415
```

f)
```
sqrt(sum(6./([1:10000].^2)))
ans =

    3.1415
```

Q 3.5

```
heights = [66 68 65 70 65]';
```

a)

```
sum(heights)/5
ans =

   66.8000
```

b)

```
mean(heights)
ans =

   66.8000
```

c)

```
w = [1 1 1 1 1]/5;
w*heights
ans =

   66.8000
```

d)

```
w = [1 1 1 0 1]/4;
w*heights
ans =

    66
```

is the same as

```
mean(heights([1,2,3,5]))
ans =

    66
```

Chapter 4

Q 4.1

```
A=ones(5);
A(3:5, :)=2;
A
A =

     1     1     1     1     1
     1     1     1     1     1
     2     2     2     2     2
     2     2     2     2     2
     2     2     2     2     2

B=ones(4, 5);
B(:, 3:4)=0;
B
B =

     1     1     0     0     1
     1     1     0     0     1
     1     1     0     0     1
     1     1     0     0     1

C=ones(6, 5);
C(2:4, 2:3)=0;
C
C =

     1     1     1     1     1
     1     0     0     1     1
     1     0     0     1     1
     1     0     0     1     1
     1     1     1     1     1
     1     1     1     1     1

D=zeros(5);
for i=1:5
    D(:, i)=[1 2 3 4 5];
end
D
D =

     1     1     1     1     1
```

```
     2     2     2     2     2
     3     3     3     3     3
     4     4     4     4     4
     5     5     5     5     5

E=ones(6);
for i=1:6
    for j=1:6
        if i<=j
            E(i, j)=2;
        end
    end
end
E
E =

     2     2     2     2     2     2
     1     2     2     2     2     2
     1     1     2     2     2     2
     1     1     1     2     2     2
     1     1     1     1     2     2
     1     1     1     1     1     2

F=ones(5);
for i=1:5
    F(:, i)=(i-1)*5;
end
F
F =

     0     5    10    15    20
     0     5    10    15    20
     0     5    10    15    20
     0     5    10    15    20
     0     5    10    15    20

for i=1:5
    G(:, i)=[1 2 3 4 5]+((i-1)*5);
end
G
G =

     1     6    11    16    21
     2     7    12    17    22
     3     8    13    18    23
     4     9    14    19    24
     5    10    15    20    25
```

```
H=zeros(8);
H(:, 1:2:end)=1;
H
H =

     1     0     1     0     1     0     1     0
     1     0     1     0     1     0     1     0
     1     0     1     0     1     0     1     0
     1     0     1     0     1     0     1     0
     1     0     1     0     1     0     1     0
     1     0     1     0     1     0     1     0
     1     0     1     0     1     0     1     0
     1     0     1     0     1     0     1     0

I=zeros(8);
for i=1:2:8
    for j=1:2:8
        if i==j
            I(i, j)=1;
        end
    end
end
I
I =

     1     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     1     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     1     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     1     0
     0     0     0     0     0     0     0     0

J=ones(5);
for i=1:5
    J(:, i)=[0 1 2 3 4]+(i-1);
end
J
J =

     0     1     2     3     4
     1     2     3     4     5
     2     3     4     5     6
     3     4     5     6     7
     4     5     6     7     8
```

```
for i=1:5
    K(i, :)=i:i:5*i;
end
K
K =

     1     2     3     4     5
     2     4     6     8    10
     3     6     9    12    15
     4     8    12    16    20
     5    10    15    20    25

for i=1:5
    L( i, :)=[1:5]+((i-1)*5);
end
L
L =

     1     2     3     4     5
     6     7     8     9    10
    11    12    13    14    15
    16    17    18    19    20
    21    22    23    24    25
```

or

```
L=G'
L =

     1     2     3     4     5
     6     7     8     9    10
    11    12    13    14    15
    16    17    18    19    20
    21    22    23    24    25
```

Q 4.2

a)

```
M=zeros(3, 3, 3);
M(2, 2, 2)=1
M(:,:,1) =

     0     0     0
     0     0     0
```

```
     0     0     0


M(:,:,2) =

     0     0     0
     0     1     0
     0     0     0


M(:,:,3) =

     0     0     0
     0     0     0
     0     0     0
```

b)

```
M=zeros(5, 5, 5);
M(2:4, 2:4, 2:4)=1
M(:,:,1) =

     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0


M(:,:,2) =

     0     0     0     0     0
     0     1     1     1     0
     0     1     1     1     0
     0     1     1     1     0
     0     0     0     0     0


M(:,:,3) =

     0     0     0     0     0
     0     1     1     1     0
     0     1     1     1     0
     0     1     1     1     0
     0     0     0     0     0
```

```
M(:,:,4) =

     0     0     0     0     0
     0     1     1     1     0
     0     1     1     1     0
     0     1     1     1     0
     0     0     0     0     0


M(:,:,5) =

     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
```

Q 4.3

a)

```
sub2ind([4,3],3,2)
ans =

     7
```

b)

```
[i,j] =ind2sub([4,3],7)
i =

     3


j =

     2
```

Q 4.4

example values:

```
x = 1;
y = 0;
z = 0;
```

a)

```
if x> 0
    disp('x is positive')
elseif x< 0
    disp('x is negative')
end
x is positive
```

b)

```
x<2 || x>pi
ans =

     1
```

c)

```
(x>2 && y<4) || z==0
ans =

     1
```

Q 4.5

```
count = 1;
roll = ceil(rand(1,2)*6);
while sum(roll) >2   %while not snake eyes (total
greater than 2)
    count = count+1; %no snake eyes, add 1 to the
count
    roll = ceil(rand(1,2)*6); %roll again.
end
disp(['Snake eyes after ',num2str(count),' rolls.']);
Snake eyes after 58 rolls.
```

Chapter 5

Q 5.1

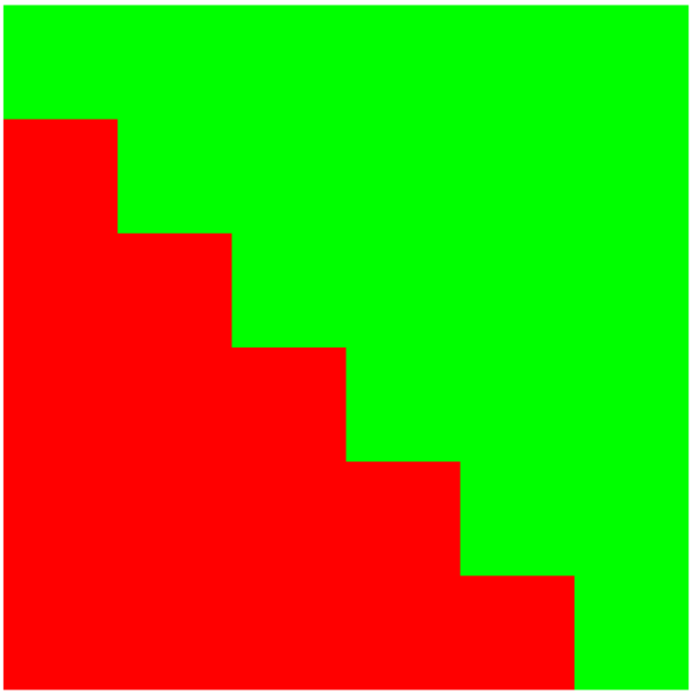
a)

```
M = ones(6,6);
for i=1:6
    for j=1:6
```

```
        if i<=j
            M(i,j) = 2;
        end
    end
end

M
M =

     2     2     2     2     2     2
     1     2     2     2     2     2
     1     1     2     2     2     2
     1     1     1     2     2     2
     1     1     1     1     2     2
     1     1     1     1     1     2
```

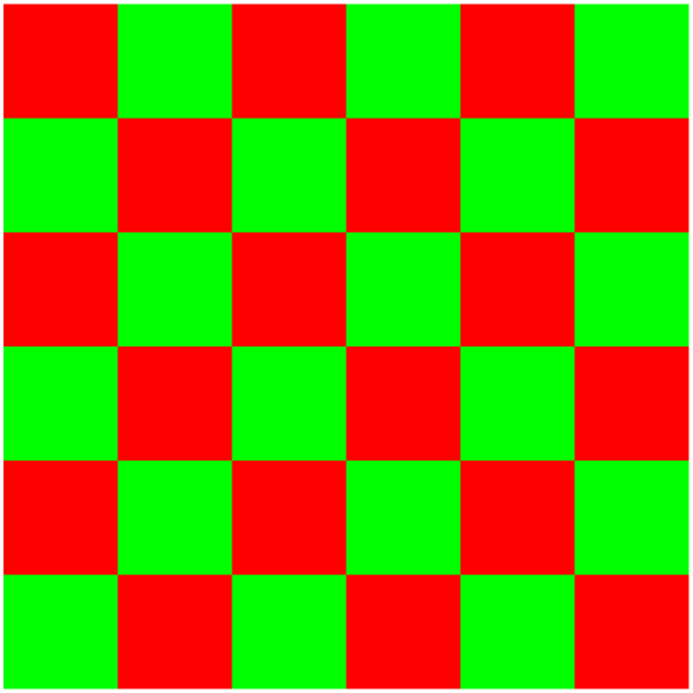
b)

```
cmap = [1 0 0 ; 0 1 0];
figure(1)
clf
image(M)
colormap(cmap)
axis equal
axis off
```



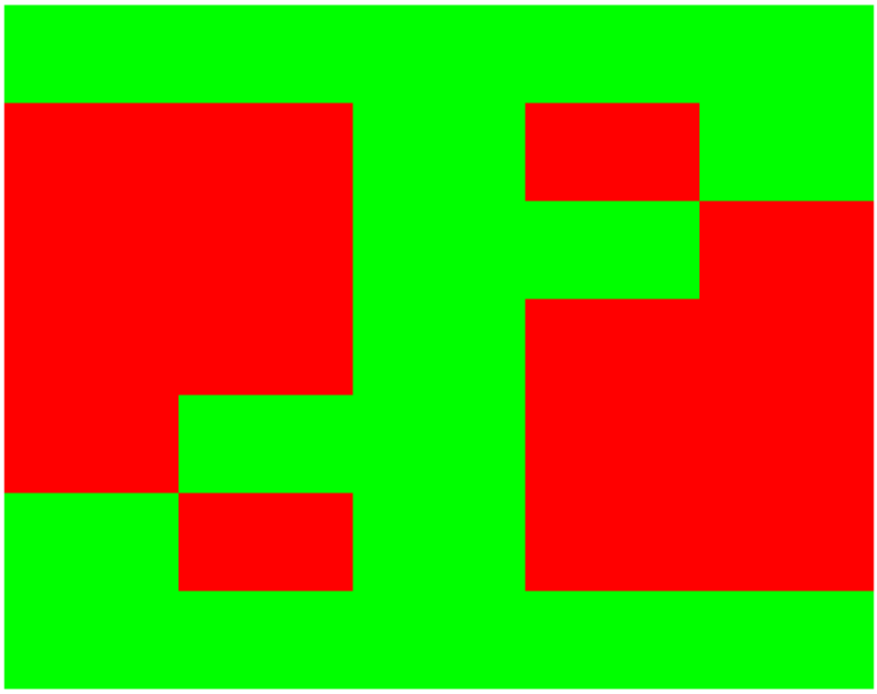
c)

```
N = ones(6,6);
for i=1:6
    for j=1:6
        if round((i+j)/2) ~= (i+j)/2
            N(i,j) = 2;
        end
    end
end

figure(2)
clf
image(N)
colormap(cmap)
axis equal
axis off
```

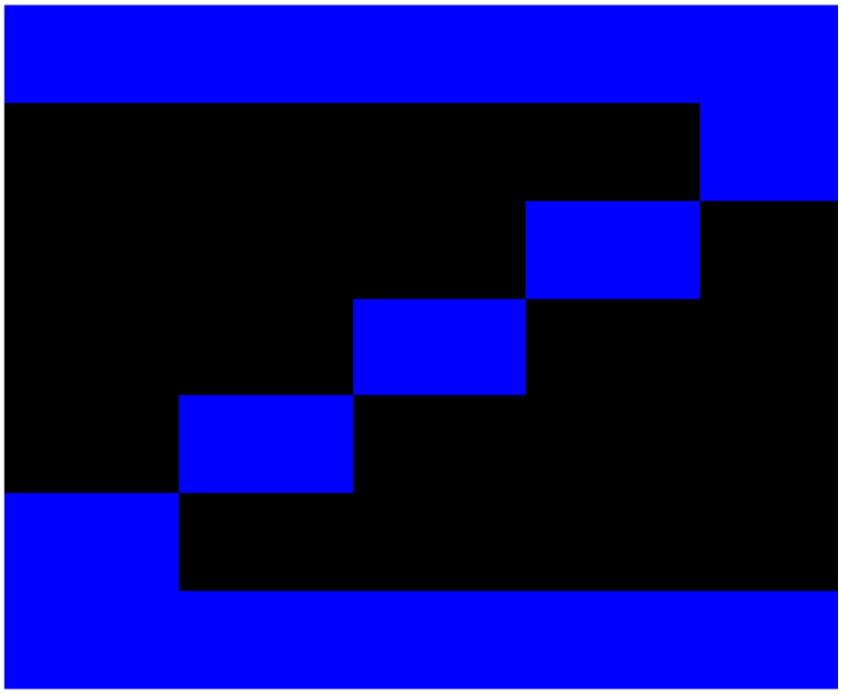


Q 5.2

```
Z=[ 1 1 1 1 1 ; 0 0 0 0 1; 0 0 0 1 0; 0 0 1 0 0; ...
    0 1 0 0 0; 1 0 0 0 0; 1 1 1 1 1];
T=[ 1 1 1 1 1; 0 0 1 0 0; 0 0 1 0 0; 0 0 1 0 0; ...
    0 0 1 0 0; 0 0 1 0 0; 0 0 1 0 0];

ZT=1+Z+(T*2);
cmapT=[0 0 0 ; 0 0 0 ; 1 .7 .7; 1 .7 .7;];
cmapZ=[0 0 0 ; 0 0 1 ;0 0 0 ; 0 0 1];
image(ZT); axis off
```



```
colormap(cmapZ)
saveas(gcf,'_5_2_a.jpg')
```

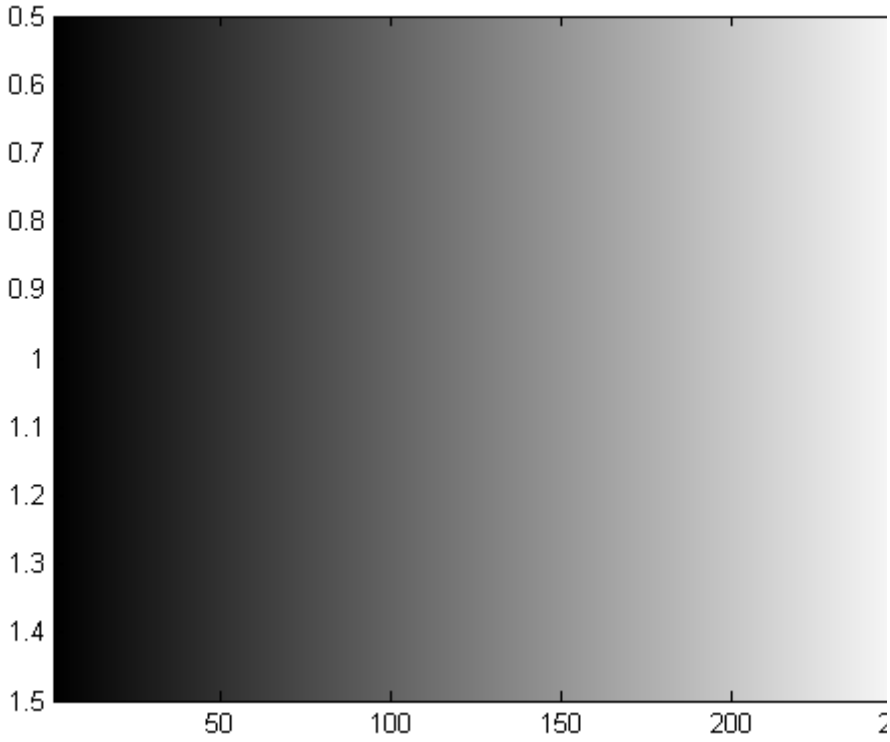


```
colormap(cmapT)
saveas(gcf,'_5_2_b.jpg')
```

Q 5.3

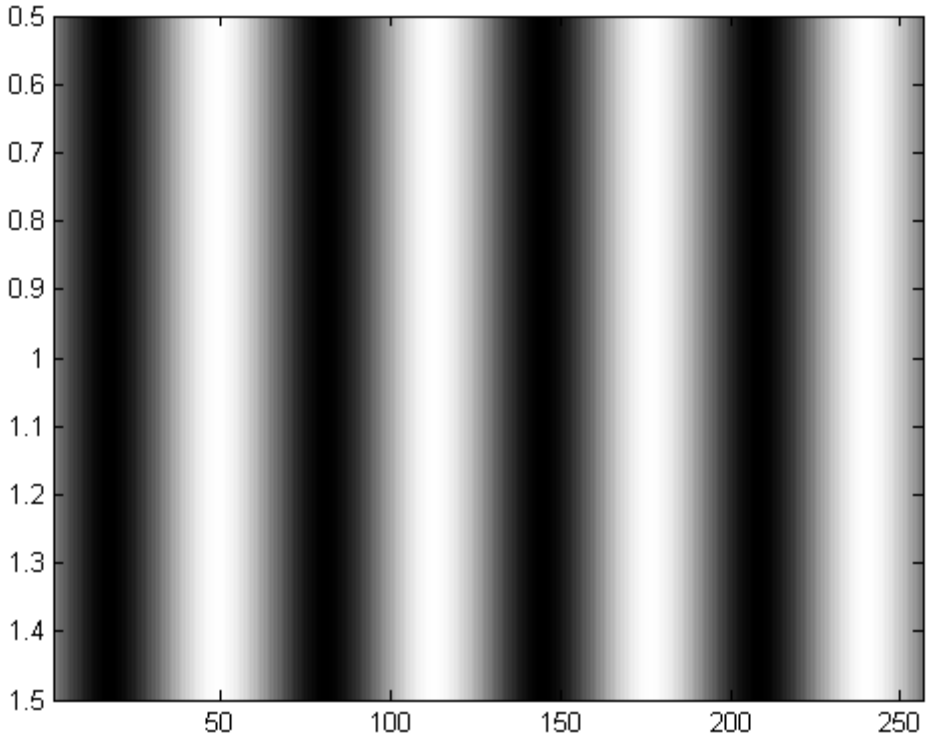
a)

```
M = 1:256;
figure(1)
clf
image(M)
colormap(gray(256));
```



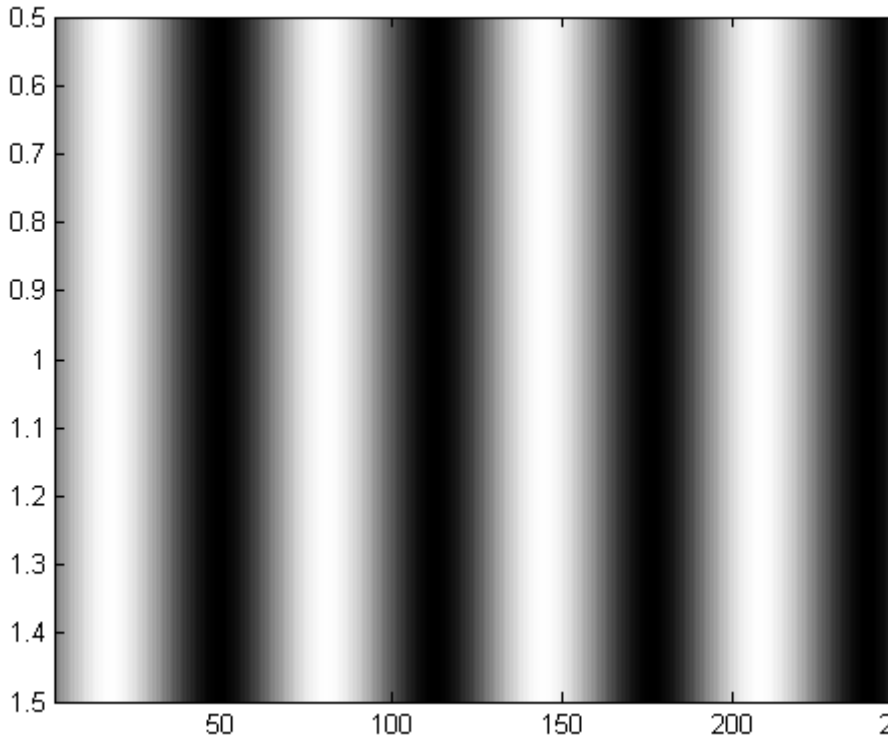
b)

```
cmap = zeros(256,3);
nCycles = 4;
phase = pi;
for i=1:3
    cmap(:,i)   =   (sin(linspace(0,2*pi*nCycles,256)'-
phase)+1)/2;
end
colormap(cmap);
```



c)

```
for phase = linspace(0,8*pi,100)
    for i=1:3
        cmap(:,i)                                          =
(sin(linspace(0,2*pi*nCycles,256)'-phase)+1)/2;
    end
    colormap(cmap);
    drawnow

end
```



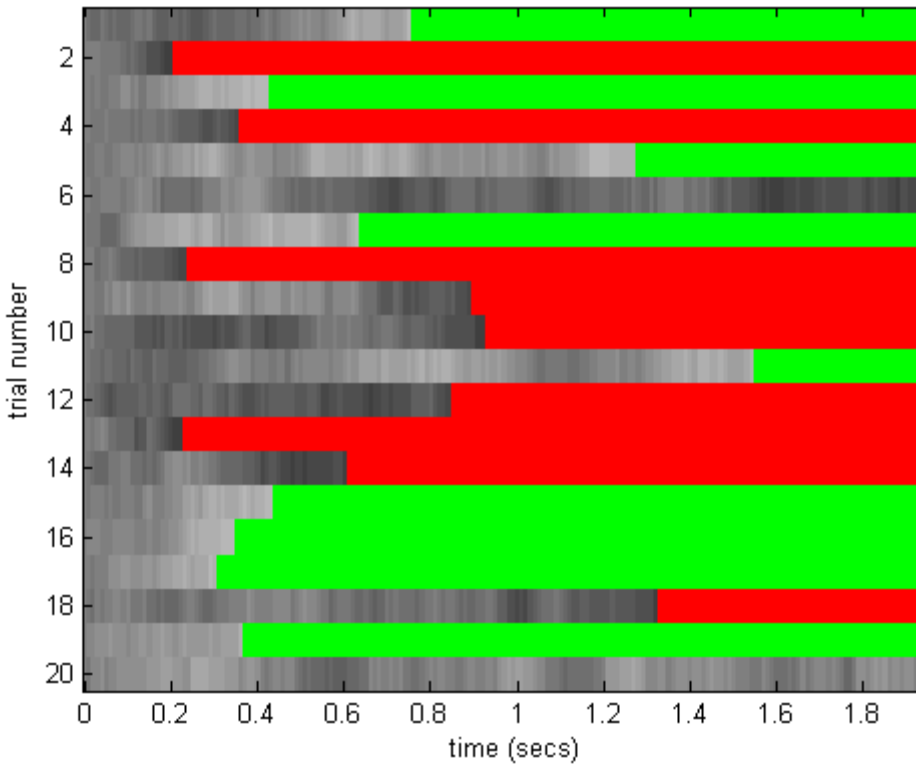
Q 5.4

look for the CAPS to see where things have changed

```
% random rat
% A very simple example of a random walk model
%
% written IF 3/2013

ntrials=20; % number of trials
timepts=0:.01:2; % time

signal=.15*rand(ntrials, length(timepts)); % SPEED UP
THE DECISION
signal(2:2:end,  :)=-signal(1:2:end,  :);  %  SWITCH
REWARDED ARM OF MAZE
% the internal signal that is added at each time
point
```

```
% note that the signal is small but always positive. Because the
% amount of signal added at each time point is independent of the signal
% at the previous time point/trial we can fill this matrix outside of the
% loop even though conceptually the signal is being added at each time point.
% It's better to do it this way (even though it's conceptually weird) because it's better to keep as much as possible out of loops for speed reasons.

noise=0.8*randn(ntrials, length(timepts));
% the noise added at each time point, much larger and has a mean of zero

choicethreshold=8;
% the animal makes a choice when the internal response reaches this number
clear resp
for n=1:ntrials % go through each of the trials
    resp(n, 1)=0; % start with an internal response of 0
    for t=2:length(timepts) %for each subsequent moment in time

        resp(n, t)=resp(n, t-1)+signal(n, t)+noise(n,t);

        % the response at that moment in time is the same as the
        % previous time + the signal that was added to the internal
        % representation at that time point + some random noise

        if resp(n, t)>=choicethreshold
            % if the response hits the positive threshold

resp(n,t:length(timepts))=(2*choicethreshold);
            % set the response for the rest of that trial to
            % 2x the choicethreshold

            t=length(timepts+1);
            % skip t to the end of the trial,
```

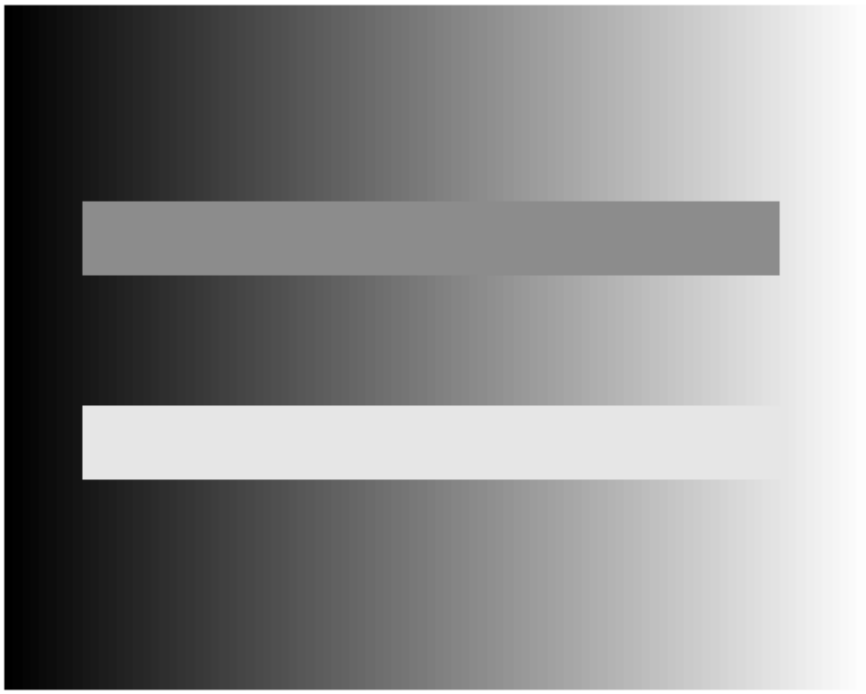
```
            % so the loop moves to the next trial
        elseif resp(n, t)<=-choicethreshold
            resp(n,t:length(timepts))=-
(2*choicethreshold);
            t=length(timepts+1);
        end
    end % end of time loop
end % for each trial
cmap=gray((4*choicethreshold)+1);
cmap(1,  :)=[1  0  0];  %  ADD  RED  AND  GREEN  TO  THE
COLORMAPS
cmap(end, :)=[ 0 1 0 ];
colormap(cmap)
image(timepts,                                  1:ntrials,
resp+(2*choicethreshold)+1);
ylabel('trial number')
xlabel('time (secs)')
saveas(gcf,'randomrat_2.jpg')
```



Chapter 6

Q 6.1

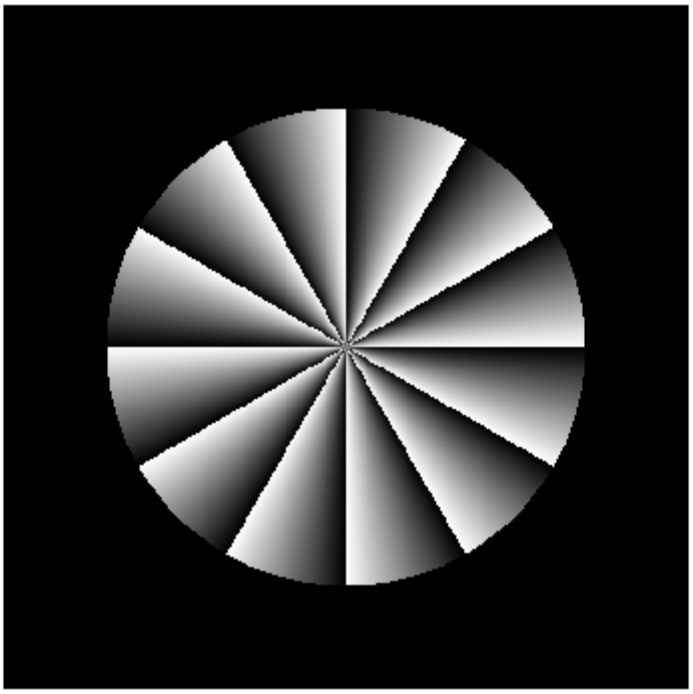
```
n = 101; % size of the nxn image of a Gaussian

[X,Y] = meshgrid(linspace(-1,1,n));
X(30:40, 10:90)=.1;
X(60:70, 10:90)=.8;

imagesc(X); axis off
colormap(gray(256))
saveas(gcf,'Illusion_ColorConstancy_1.jpg');
```



Q 6.2

```
% create mesh
n = 701; % size of the nxn image of a Gaussian
```

```
nseg=6; % number of segments'
radius=.7; % radius of the aperture

[X,Y] = meshgrid(linspace(-1,1,n));

% create segment pattern

theta = atan2(Y,X)./pi;
theta=mod(theta*nseg, 1);
%theta=max(theta(:))-theta; % TO FLIP THE DIRECTION

% create aperture

radiusimage = sqrt(X.^2+Y.^2);
aperture=NaN(size(radiusimage));
aperture(radiusimage<radius)=1;
aperture(radiusimage>=radius)=0;

illusion=theta.*aperture;

imagesc(illusion);
axis square
axis off
colormap(gray(256))
saveas(gcf,'peripheraldrift1.jpg')
```



Chapter 7

Q 7.1

```
clear x y
vect = 1:10;

% a)
x.vect = vect

% b)
for i=1:length(vect)
    y(i).vect = vect(i);
end

y(1).vect
y(2).vect
```

```
x =

    vect: [1 2 3 4 5 6 7 8 9 10]


ans =

     1


ans =

     2
```

Q 7.2

a)

```
bigCell{1} = vect;
bigCell{2} = x;
bigCelll{3} = y;
bigCell{4} = 11:20;
bigCell{5} = 'This is the fifth element of bigCell';
bigCell{6} = 'Matlab is great!';
```

b)

```
lessBigCell = bigCell([1,4,6])
lessBigCell =

    [1x10  double]      [1x10  double]      'Matlab  is
great!'
```

c)

```
bigCell{1}+bigCell{4}
ans =

    12     14     16     18     20     22     24     26
28    30
```

d)

```
cell2mat(bigCell([1,4])')
ans =
```

```
     1      2      3      4      5      6      7      8
9    10
    11     12     13     14     15     16     17     18
19    20
```

e)

```
str = [bigCell{5}(1:7) bigCell{6}(10:end)]
str =

This is great!
```

Chapter 8

Q 8.1

See included function 'nonanStats.m'

```
type nonanStats
function [m,s,sem] = nonanStats(x)
%function [m,s,sem] = nonanStats(x)
%Calculates the mean, standard deviation and standard
error of the mean of
%a vector x

%y contains values of x that are not NaNs
y = x(~isnan(x));

%calculate the mean, standard deviation and standard
error of the mean of y
m = mean(y);
s = std(y);
sem = s/sqrt(length(y));
```

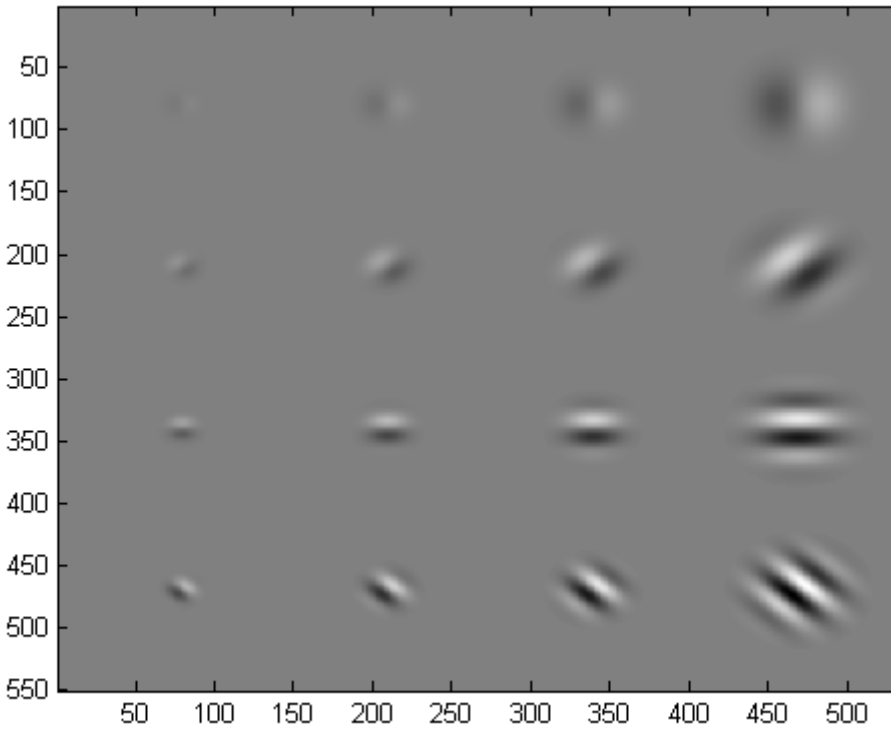
Q 8.2

See included file 'SineInApertureHW.m and function MakeSineApertureHW.m

```
type MakeSineApertureHW
type SineInAptertureHW

SineInAptertureHW
function sinewave2D=MakeSineApertureHW(g)
%
```

```
% function that creates a vertical sinusoid windowed
by a circular aperture
%
% takes as input
% x - the values of x for which the sinusoid is
computer
% sf - the spatial frequency of the sinusoid
% radius  0 the radius of the sinusoid
%
% returns as output the 2D sinusoidal grating image
%
% written by matlab class 2011

%% create a grating that is length(x) x length(c)
center=[0,0];
width= (g.rad/2);
%% create an aperture
[X, Y]=meshgrid(g.x);
ramp=cos(g.orientation)*(X-
center(1))+sin(g.orientation)*(Y-center(2));
sinewave= g.contrast*sin(g.sf*(ramp-g.phase));
Gaussian=exp(-((X-center(1)).^2+(Y-
center(2)).^2)/width^2);
sinewave2D=sinewave.*Gaussian;
RAD=sqrt(X.^2+Y.^2);
sinewave2D(RAD>g.rad)=0;


% SineInApertureHW
% creates an image of apertured vertical sinusoids in
tiled array

clear all
close all
% info
sflist=[1 2 3 4];% spatial frequency in cycles per
image
radlist=[1 1.5 2 3 ];
phaselist=[0 pi/2 pi 3*pi/4];
orientationlist=[0 0.7854 1.5708 2.3562];
contrastlist=[ 0.2000 0.3333 0.4667 0.6000; 0.3333
0.4667 0.6000 0.7333;...
    0.4667 0.6000 0.7333 0.8667; 0.6000 0.7333 0.8667
1.0000];
for r=1:4
    for c=1:4
        g(r,c).x=linspace(-pi, pi, 100);
```

```
        g(r,c).sf=sflist(r);
        g(r,c).rad=radlist(c);   %   radius   of   the
aperture
        g(r,c).phase=phaselist(r);
        g(r,c).orientation=orientationlist(r);
        g(r,c).contrast=contrastlist(r,c);
        %   creates   a   100x100xlength(sf)   matrix
containing two 2d
        % apertures  sinusoids  of  different  spatial
frequencies
        g(r,   c).sinewave2D=   MakeSineApertureHW(g(r,
c));
    end
end

% initialize a tilematrix by filling with zeros
imgsize=length(g(1).x);
ntiles=length(sflist);
sep=30;
tilesize=(ntiles *(imgsize+sep))+sep;
tilematrix=zeros(tilesize);
startpos=sep:sep+imgsize:length(tilematrix-1);

% poke in the tiles
for rtile=1:ntiles
    for ctile=1:ntiles

tilematrix(startpos(rtile):startpos(rtile)+imgsize-1,
...
            startpos(ctile):startpos(ctile)+imgsize-
1)= ...
            g(rtile, ctile).sinewave2D;
    end
end
axis off
axis square
imagesc(tilematrix);
colormap(gray(256));
```



Chapter 9

Q 9.1

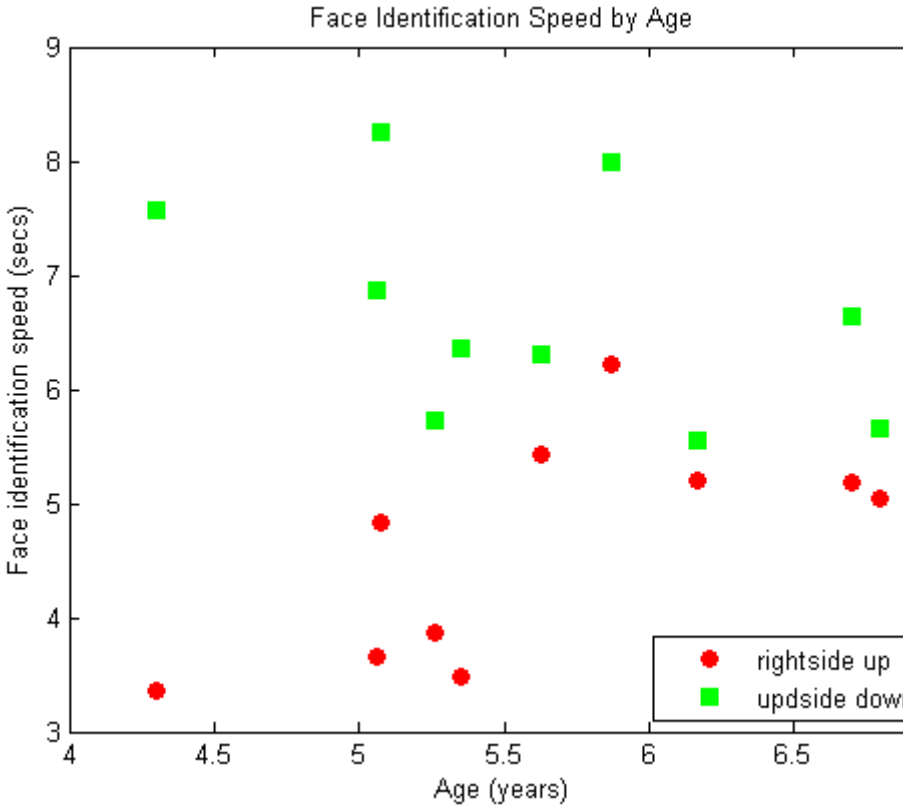
fake data

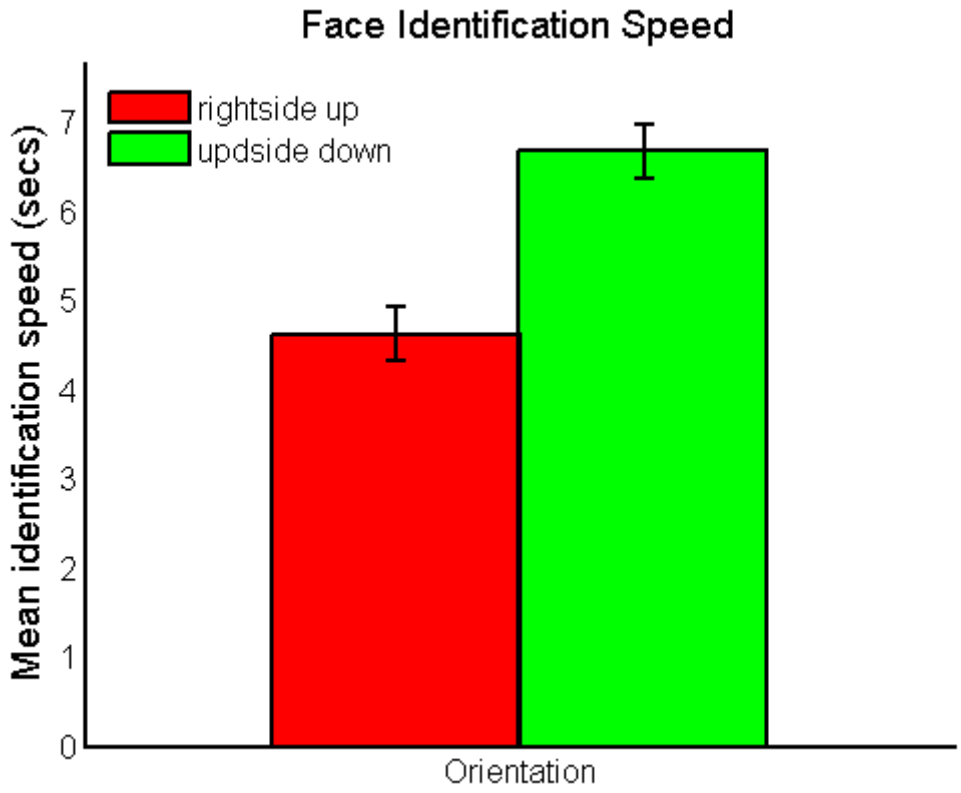
```
randn('seed',0);
age=5+randn(10, 1);
RTdown=7+randn(10, 1);
RTup=age-1-randn(10,1);
```

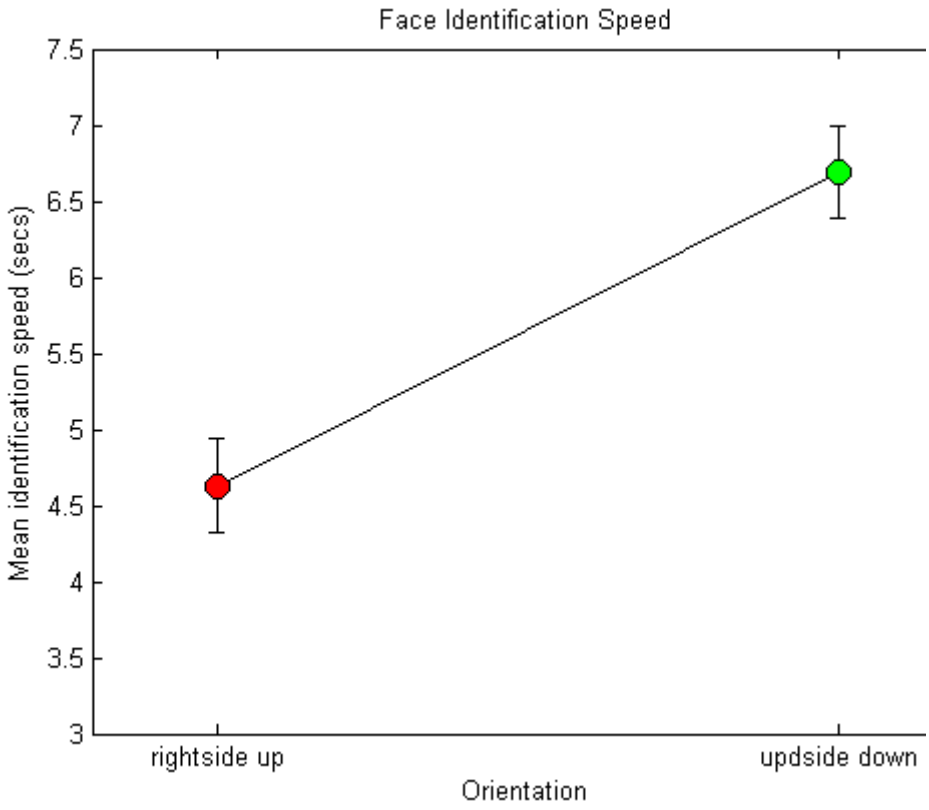
plot data

```
figure(1);
clf
p1=plot(age, RTup, 'ro', 'MarkerFaceColor', 'r');
hold on
p2=plot(age, RTdown, 'gs', 'MarkerFaceColor', 'g');
```

```
xlabel('Age (years)')
ylabel('Face identification speed (secs)')
legend([p1    p2],    {'rightside    up',    'updside
down'},'Location','SouthEast')
title('Face Identification Speed by Age');

figure(2);
clf
handles=barweb([mean(RTup); mean(RTdown)], ...
    [std(RTup)./sqrt(length(RTup));
std(RTdown)./sqrt(length(RTdown))], ...
    [],      {'Orientation'},   'Face   Identification
Speed', ...
    [], 'Mean identification speed (secs)', [1 0 0; 0
1 0 ],[],  {'rightside up', 'updside down'})
set(handles.legend, 'Location', 'NorthWest')

figure(3);
clf
errorbar([1    2],    [mean(RTup);    mean(RTdown)],
[std(RTup)./sqrt(length(RTup));
std(RTdown)./sqrt(length(RTdown))], ...
    'k-');
hold on
plot([1  ],  [mean(RTup)],  'ko',  'MarkerSize',  10,
'MarkerFaceColor', 'r');
plot([2],  [mean(RTdown)],  'ko',  'MarkerSize',  10,
'MarkerFaceColor', 'g');
set(gca, 'XTick', [1 2])
set(gca,'YLim',[3,7.5]);
set(gca,  'XTickLabel',    {'rightside  up',  'updside
down'})
ylabel('Mean identification speed (secs)');
xlabel('Orientation')
title('Face Identification Speed');
handles =

      bars: [402.0292 405.0287]
    legend: 411.0287
    errors: [459.0287 462.0287]
        ax: 401.0287
```


Face Identification Speed by Age
Face identification speed (secs)
Age (years)
9
8
7
6
5
4
3
4
4.5
5
5.5
6
6.5
rightside up
updside dow


Face Identification Speed
rightside up
updside down
Mean identification speed (secs)
0
1
2
3
4
5
6
7
Orientation



Q 9.2

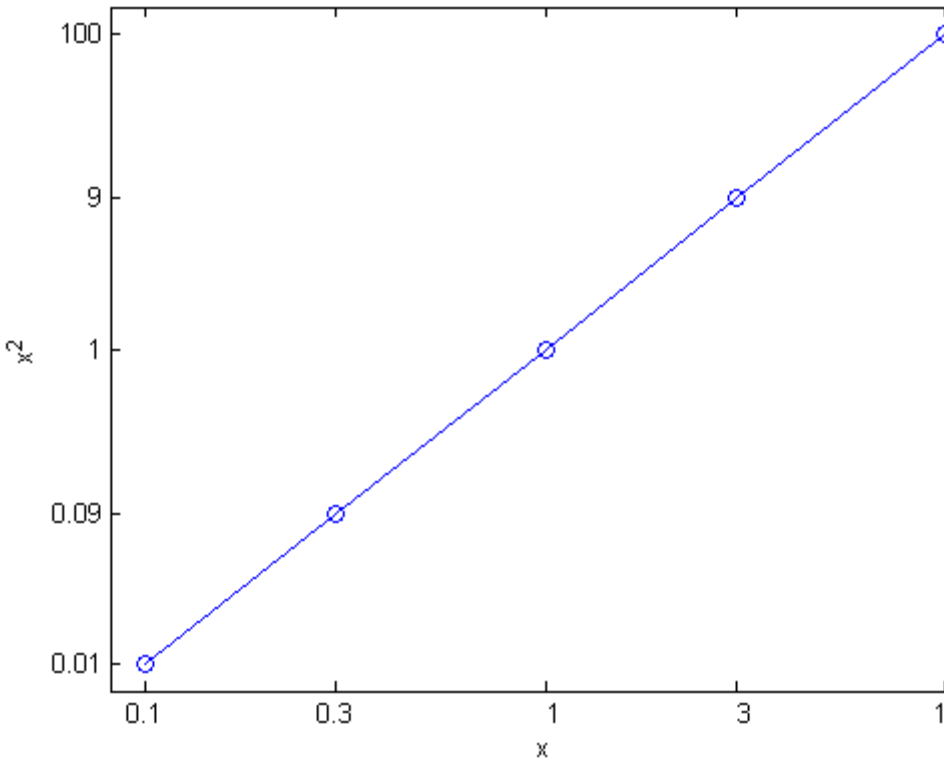
a) See included function 'logy2raw.m'

```
type logy2raw
function logy2raw(base,precision)

%logy2raw([base],[precision])
%
%Converts Y-axis labels from log to raw values.
%base:          base of log transform; default base is
e.
%precision:     number of decimal places, or a FORMAT
string.

if ~exist('base','var')
    base=exp(1);
end
```

```
origYTick = get(gca,'YTick')';
newYTick = base.^(origYTick);

if exist('precision','var')

set(gca,'YTickLabel',num2str(newYTick,precision));
else
    set(gca,'YTickLabel',num2str(newYTick));
end
```

b)

```
x = [.1,.3,1,3,10];
y = x.^2;

figure(1)
clf
plot(log(x),log(y),'o-')
set(gca,'XTick',log(x));
set(gca,'YTick',log([.01,.03,.1,.3,1,3,10,30,100]));
set(gca,'YTick',log(y));
logx2raw
logy2raw;
xlabel('x');
ylabel('x^2');
```



Chapter 10

Q 10.1

a)
```
dataCell = cell(4,2);
```

b)
```
dataCell(:,1) = {'Barack','George','Bill','Ron'};
```

c)
```
randn('seed',1);
IQ = round(100+15*randn(4,1))
dataCell(:,2) = num2cell(IQ);
IQ =
```

```
   115
    96
    92
    99
```

d)

```
xlswrite('IQ',dataCell);
```

e) * after adding a fifth row *

```
[IQ,names,newDataCell] = xlsread('IQ');
```