Part 1 – Matlab

What is Matlab?

Matlab is a program that performs various numerical operations. Like a big calculator. Computer languages are generally divided into *low-level languages*, that interact with the specific hardware directly and need to be both written and compiled for the specific setting you are using. This is very powerful, because it allows you to use the resources of your machine in whatever way you choose. *High-level languages*, on the other hand, can be transferred from machine to machine (and, in some cases, from operating system to operating system), but often will need to be compiled for a specific setting. Matlab functions as a *scripting language.* Scripting languages are high-level computer languages. However, above and beyond the portable nature of most high-level languages, a system-specific interpreter interprets them online, as they run. Therefore, you will not need to compile the programs you write on Matlab. Scripting languages are relatively easy to learn. However, they do not retain the same level of flexibility as low-level languages. Moreover, because they need to be interpreted as they run, they are often slower than the equivalent program written in a compiled high-level language.

Starting to work with Matlab:

You will need a computer running Matlab. Matlab runs on Macs and PCs running Windows or Unix The student version of Matlab doesn't cost that much ($95 at TSW) and can be used for everything we will be doing here, but notice that it can't do everything that you can do with the standard version of Matlab.

Start Matlab. A window will appear that's divided into a number of sub-windows. The "command window" is the one which has a little prompt '>>'. The prompt is your interface with Matlab, for now. When you type things at the prompt and press Enter, your commands are processed by Matlab.

Type at the prompt:

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

You just told the computer to create a list of letters 'I have no idea what I'm doing' and to name that list of letters str1. str1 is a ***variable*** - The single quotes tell the computer that str1 is a list of letters (not numbers, more on that later). Any list of letters is called a ***string***. Now type:

```
>>who
```

This command asks your computer to give you a list of all the variables you have. At the moment the only variable you have is str1.

```
Text that looks like this is stuff
that is happening in the command
window. Either stuff you are typing
in at the prompt, or stuff that the
command window is spitting back out.
```

Text that looks like this is me telling you what's happening in Matlab for a particular command.

Text that looks like this is me giving you a general overview.

Typing the name of a variable asks your computer to tell you what is contained within that variable.

```
>>str1
```

The computer should show you what's in str1

```
str1 =

I have no clue what I am doing
```

Using disp also displays what a variable is, but only shows the contents, instead of repeating the name of the variable

```
>>disp(str1)
```

So all the computer does is display the contents of the variable as follows:

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

```
>>str2='Is it all going to be boring?';
>>str2='Is it all going to be boring?'
```

The first time you typed this you added a semi-colon at the end. The second time you didn't. The semi-colon tells the computer whether or not you want it to display the output of each command. This will be useful later when you write scripts. When you debug your programs, sometimes you want the result of a line to be output to the command window, as a sanity check. You can control that by adding or removing semi-colons.

```
>>who
```

Now you have both str1 and str2

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

Now you are re-defining str1 by making it represent a slightly different list of letters

```
>>str1
```

See - the list of letters contained within str1 has changed

```
>>str1(3)
```

You've asked the computer to display the third letter in str1. This is called ***indexing*** or ***subscripting***. 3 is an index (or subscript) into the third character in str1. Now try:

```
>>str1(6)
```

You can see from this that the computer is counting spaces

```
>>mixstr=str1;
```

Now you've created a new variable called mixstr. You've told the computer to make mixstr the same as str1

```
>>mixstr
```

See - they are exactly the same

```
>>mixstr(3)=str1(1);
```

Now you are telling the computer to make the 3rd letter in mixstr the same as the 1st letter in str1

```
>>mixstr(1)=str1(3);
```

Now make the 1st letter in mixstr the same as the 3rd letter in str1

```
>>mixstr
```

You should now have:

```
'S Itill have no clue what I am doing'
```

You can also create lists of numbers. These are called ***arrays*** or ***vectors***. Here's four different ways of creating a vector list that goes from 2 to 9 in steps of 1. A ***variable*** is a generic term that can be used to describe a ***character*** (a single letter), a ***string*** (a string of characters), a ***double*** (a single number, more on that later), a ***vector***, or a ***matrix*** (a two or more dimensional set of numbers) and other types of data structures.

```
>>array1=[2 3 4 5 6 7 8 9]
>>array1=linspace(2, 9, 8)
```
Here you are saying you want a 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 8 pieces of data evenly spaced between two and 9 seconds.

```
>>array1=2:1:9
```
Here you are saying that you want 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 data that went from 2 seconds, to 9 seconds, and had collected data every second.

```
>>array1=2:9
```
Matlab assumes a default step-size of 1, so you can simply skip it for this particular way of creating vectors.

You can also index vectors. 2 indexes the second integer in array1
```
>>array1(2)
```

You can also index more than one number in an array or string.
```
>>array1(2:4)
```

```
>>disp(array1)
```

`disp` can also be used to display numbers

Matrices and Calculations

The real power of Matlab comes from using matrix computations (Matlab actually stands for ‘Matrix Lab’!). As mentioned earlier, you can have lists of numbers as well as of letters. These list of numbers can be either one-dimensional (vectors, or arrays), or n - dimensional (for n>1; matrices). Let’s give this a shot:

```
>>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]
```
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)
```
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]
>>vect2=vect1’
```
Vectors can be tall instead of long, ' (the little symbol below the double quote on your keyboard) is a ***transpose***, and allows you to swap rows and columns.
Remember there is also the command `whos,` which will tell you the size of all your variables.
Use `whos` to look at the size of mat1, mat2, vect1 and vect2.
```
>>whos
```

What does `mat1'` look like?
You can perform various calculations on matrices and arrays.

You can add a single number (also called a scalar) to a vector
```
>>vect1+3
```
You can subtract
```
>> vect2-3
```
You can add a vector onto itself
```
>>vect1+vect1
```
You can also add two vectors as long as they are the same size. You can't add vect1 and vect2 together since they are different sizes.
```
>>vect1+vect2
??? Error using ==> plus
Matrix dimensions must agree.
```
The reason you got an error is that you can't add something with 1 row and 4 columns to something else that has 4 rows and 1 columns.
```
>>vect3=[2 4 8 12 6]
>>vect1+vect3
```
These two vectors are the same size, so you can add them together.

Matrix multiplication and division

When you want to add two vectors or matrices to each other you need to know that there are two sorts of multiplication and two sorts of division. The simple kind of multiplication and division is called array multiplication (also known as **scalar multiplication**). If you are multiplying/dividing a vector/matrix by a single number, you can do that as you did before:

```
>>vect1*3
>>mat1*0.5
>>vect1/2
```

However, if you are multiplying a vector/matrix by another vector matrix, you need to tell matlab what kind of multiplication to do (as you will see below, there is more than one way to multiply matrices). Scalar multiplication will proceed element by element: each element in the first vector is multiplied by the corresponding element in the second vector. You do this using the symbols `.*` and `./`
Note that the vectors must be the same shape and size:

```
>>vect1.*vect3
>>vect1./vect3
>>vect3./vect1
```

Watch out for the transpose in the next 2 examples:

```
>>vect1.*vect2'
>>vect1.*vect3'
??? Error using ==> times
Matrix dimensions must agree.
```

You will get this error a lot. What an error like this means is that the two vectors or matrices that you are trying to perform an operation on (such as multiplication) aren't the right size or shape to do what you are doing. Lots of operations are fussy about making sure the sizes of the vectors or matrices are consistent. So when you get this error the first thing you should do is check the `size` of all the variables that you are tying to manipulate, and try to work out whether they might be different sizes. 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 don't rush through the examples above. Make sure you understand what's going on.

```
>> mat1./vect1
??? Error using ==> rdivide
Matrix dimensions must agree.
```

This is not going to happen for you, regardless of how you transpose mat1 and vect1. vect1 and mat1 will never be the same size, no matter what you do.
So, with point-wise multiplication and division (which is what you will be doing most of the time) you can:
1) multiply or divide a matrix or vector by a single number (Notice that in this case either `*` or `.*` will work)
2) multiply or divide a matrix or vector by a matrix/vector that is the same size (make sure you use `.*`).
It's best to get into the habit of using .* all the time, unless you are specifically using matrix multiplication (which you will only use rarely)

The second kind of multiplication and division is matrix multiplication and matrix division. In general matrix multiplication and division occurs through the following formula:

For the m-by-n matrix A and the n-by-p matrix B, the m-by-p matrix of multiplying the both will be defined by :

$$(A \cdot B)_{i,j} = \text{Sum}_{r=1\ldots n}\ \{A_{i,r} \cdot B_{r,i}\}$$

where i goes from 1 to m and j goes from 1 to p. Notice that you can use this in order to multiply any two matrices for which the inner dimensions agree, so a 1-by-n matrix can multiply an n-by-n matrix (how large will this product be?), but an n-by-n matrix cannot multiply a 1-by-n matrix. Conclusion – matrix multiplication is not commutative – so be careful!

This produces two general types of matrix multiplication.

Outer product:

```
>>B = [1; 2; 3; 4; 5]
B =
     1
     2
     3
     4
     5

>>C = [2 3 4 3 2]
C =
     2     3     4     3     2


>> whos
  Name       Size                    Bytes  Class

  B          5x1                        40  double array
  C          1x5                        40  double array
```

The first vector is tall and thin, and the second vector is short and fat. Calculating the ***outer product*** of the two vectors:

```
>>B*C

ans =
      2      3      4      3      2
     4      6      8      6      4
     6      9     12      9      6
     8     12     16     12      8
    10     15     20     15     10
```

If we transpose both B and C and reverse the multiplication order, then the number of rows in C' again matches the number of columns in B', and we can calculate another outer product, which is in fact the transpose of the previous outer product:

```
> C'*B'

ans =

     2      4      6      8     10
     3      6      9     12     15
     4      8     12     16     20
     3      6      9     12     15
     2      4      6      8     10
```

Inner product:

Here the first vector is short & fat, and the second vector is tall & thin.

```
>>C*B
```

With inner products you can again transpose both vectors and swap the order of the multiplication. This time you get the same answer – 42.

Creating Programs

By now you should be getting a little irritated with having to type in every command one at a time. We are therefore going to create a ***program*** – a program is simply a document containing a sequence of commands.
In Matlab programs are written in documents called ***m-files***. So now we are going to put the commands you just did in a m-file.

Make sure the command window is at the front. Now go to the menu bar and choose File->New->M-file.
You'll get a blank document in a new editor window in which you can write your program.
Let's look at a program that uses some of the things we've learned so far:

```
Green is comments
Purple is text that is part of a string
Blue is commands that start and end loops
```

You’ll notice in the code below that some of the lines are rather long. When you are writing code and a line is longer than your page you can break it using three dots : ‘ . . . ‘ . Without those three dots Matlab will treat each part of the line as a separate command and give you an error message.

```
% Calculations.m
%
% Example program that carries out a series of
% calculations on two numbers and two matrices
%
% written by IF and GMB 4/2005
%
% modified by ASR 7/2007: added matrix operations

clear;
n1=input('choose the first number ... ');
n2=input('choose the second number ... ');

% spit responses out onto the command line
disp(['first number is ', num2str(n1)])
disp(['second number is ', num2str(n2)])
disp([num2str(n1),'+', num2str(n2), ...
 '=', num2str(n1+n2)]);
disp([num2str(n1), '-', num2str(n2), ...
'=', num2str(n1-n2)]);
disp([num2str(n1), '*', num2str(n2), ...
 '=', num2str(n1.*n2)]);
disp([num2str(n1), '/', num2str(n2), ...
 '=', num2str(n1./n2)]);
if (round(n1)==n1)
   disp(['n1 is an integer, n2 rounded = ', ...
       num2str(round(n2))]);
end
if (round(n2)==n2)
   disp(['n2 is an integer, n1 rounded = ', ...
       num2str(round(n1))]);
end;
disp([num2str(n1), ' to the power of ', ...
   num2str(n2), ' is ' ,num2str(n1.^n2)]);
disp(['the smallest number of ',num2str(n1), ...
    ' and ',num2str(n2), ' is ', num2str(min(n1, n2))]);
if n1>0 & n2>0
   disp('Both n1 and n2 are greater than zero');
end

A = [0:n1:n1*10 ; 0:n2:n2*10].*n1;
B = ([0:n2:n2*10 ; 0:n1:n1*10]./n2)';

disp('The matrix A is:');
disp(A);
disp('The matrix B is:');
disp(B);

disp ('A * B = ');
disp(A*B);
disp('B * A =');
disp(B*A);
```

```
54      disp ('A.*n1 = ')
55      disp(A.*n1);
56      disp('B./n2 = ')
57      disp(B./n1);
```

Lines 1-8: Every program should begin with a few lines of documentation. This is called a ***header***. Good headers contain the following information:
1) The name of the program
2) A description of what it does
3) Who wrote it, and when
4) If you are changing an existing program, add a comment about that. The time you did it and what you changed

You can run Calculations.m in two ways. One is to type the name of the program into the command window:
```
>Calculations
```
Alternatively you can go to the menu bar and choose Debug-Run (can also be invoked by pressing F5).

Lines 11-12. `input` prints a string onto the command window and waits for a response. The expected response needs by default to be a number. If you try running the program and type a letter in (a *character*) rather than a number – you will get an error:

```
??? Error using ==> input
Undefined function or variable 'r'.
```

]It's possible to get input to accept characters and strings instead of numbers by explicitly telling it that the input won't be a number. Try this at the command window:
```
>> yourname=input('What is your name? ', 's')
```
Here the `'s'` tells `input` that the input will be a string instead of a number.

Lines 14-23. num2str converts the number n1 into a string. So:
```
['first number is ', num2str(n1)]
```
is one long string, which is then displayed by disp.

Line 25. The == checks to see if round(n1) is the same number as n1. Try
```
>3==3
```
This will give you a 1 since it is true
```
>3==3.4
```
This will give you a 0 because it is untrue.
It's important to remember the difference between == and =,
The single = tells Matlab to make the variable x be 3.
```
>x=3

x =

      3
```

The double == asks Matlab to check whether or not x is equal to 3, and returns and answer of 1 if x *does* equal 3 and an answer of 0 otherwise.
```
>x=3.4;
>x==3
ans =

      0
```

Remember how we explained that numbers could either be integers (e.g. 3) or be doubles (3.12). On Line 22 we check to see whether n1 and n2 are round numbers by seeing if the rounded version of the number is the same as the number itself. If this is confusing try:

```
>round(3.15)
>round(3)
>round(3.14)==3
>round(3.14)==4
```

Remember that the way == works is that you get an answer of 1 if the numbers on either side of the == are the same, and a 0 otherwise.
`if` loops work in the following way: The if statement checks the condition that follows the if. For this if statement the condition is (round(n1)==n1). If the output of that condition is a 0 (the condition is false) then the statement between the `if` and the `end` (line 23-24) is **not** executed. If the output of the condition is anything but 0 (the condition is true), then the statement after the `if` is executed. Traditionally a true condition is represented by a 1, but in Matlab an `if` loop will be executed unless the condition following the `if` results in a 0.
Line 37 The `&` operator is used when you only want to carry out a loop only when more than one condition are both true. `&` checks to see if *both* the statement before and after the & are true. I.e. `Condition 1 & Condition 2` gives you a 1 if both conditions are true, and gives you a 0 otherwise. Try the following:

```
> 3>0 & 2>0
> 3>0 & -1>0
> -1>0 & 3>0
```

Lines 41- 56 Demonstrate the use of matrices and calculations done on matrices. Notice that in order for the multiplication to work, one of the matrices needs to be transposed, so that the inner dimensions of the matrices match in both multiplications (outer and inner).

Now we need to save the file. Make a folder called “PTBTutorial” somewhere. Create a subfolder called “Misc”. **Don’t** put these folders inside the Matlab application folder or you will lose everything if you reinstall Matlab.

Save the file as Calculations.m (the same as the header) in the “Misc” folder.

Now go back to the command window and type

```
>>help Calculations
```

You will almost certainly get an error message:

```
Calculations.m not found.
```

When the computer says that a file is “not found” that means the computer can’t find an m-file that has that particular name. The reason the computer can’t find the file even though you saved it in the folder ‘Misc’ is because the computer is only allowed to look for files in certain places.

One place that the computer always looks for files is the ***current directory*** or ***working directory***. In fact this is the first place that the computer looks. When Matlab opens it automatically links to a particular folder (the default setting is made by Matlab). If you don’t tell it otherwise it will save files to that folder. You can see what folder Matlab thinks is the current directory using the ***print working directory*** command:

```
>>pwd
```

The other places that the computer can find files are in folders that are in Matlab’s ***search path***. This ***path*** is a simply a list of folders that the computer 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
```

To tell the computer where to look, you ***set the path*** …

Setting the path via the menu bar

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 the "Misc" folder, and click OK. Then choose Save and Close.

Now type

```
>> help Calculations
```

You should see the information entered in the header.

The command `help` tells the computer to display the header you wrote. That's why you always need to write headers for every m-file that describe clearly what the program does and how to use it.

An important thing to remember about setting your path is that if there are two files with the same name, and your path allows the computer to see both of them you won't get a warning, the computer will just use the first one it finds!
You can find out which file the computer is using by typing

```
>>which Calculations
```

Matlab should spit out something like the following depending on where you put Calculations.m on your computer.

```
/Users/ariel/PTBTutorial/Misc/Calculations.m
```

You only have one version of Calculations. If you had multiple versions, Matlab would print out the path for each version that was within the path. The one at the top of the list is the version that Matlab will use by default at that current moment.

But be careful – which version that is used depends on what directory is the current working directory for Matlab. If the current directory changes it is possible that the version of MixStrings that is used will also change. This can lead to some really weird ***bugs*** (a bug is any time a program doesn't do what you want it to do and you have no idea why). One really confusing thing that can happen if more than one file of the same name is in the path is that you can make changes to a file, but the changes don't affect what the computer does. What's happening is that the computer is not actually using the file that you are changing – it is using a different file of the same name somewhere else in the path.

So you need to be careful about two things:
1) Don't be sloppy about having multiple m-files with the same name sitting in different directories
2) Be careful about your path.
Make sure that old directories with out-of-date files in them aren't still in your path. One good technique for path management is not to simply put folders in your default path so

they are permanently in the Matlab path. Instead, when you write a program you simply add the paths you will need for that program at the beginning of the program using Matlab commands (instead of the Menu bar). This technique lets you have different paths depending on which programs you are running, instead of having one mega-path. Using this technique to manage your path minimizes the probability of having out-of-date folders in your path.

Changing the path within Matlab

Play with the following commands and use them to move to your MatlabClass/Misc directory.

```
>>pwd
```

This tells you the current directory

```
>>cd ..
```

Moves up one directory

```
>>ls
```

Lists the files in that directory. You can move to any of the files listed in that directory:

```
>>cd Misc
```

Moves to folder Misc (provided you are in a folder that contained the subfolder Misc). You should be able use these commands to navigate to your Misc folder. Then you can add Misc to your path as follows:

```
>>addpath(pwd)
```

What you are doing here is telling the computer to add the folder you are currently in (`pwd`) to the path.

Remember how in my computer I used which to find that Calculationss lived in the folder /Users/PTBTutorial/Misc
Well I can now use that information at the beginning of Calculations to make sure that the folder Misc is in my path by just adding the line
addpath('/Users/ariel/PTBTutorial/Misc') at the beginning of the program

Help – your new bestest of friends

One of the advantages of working with Matlab is that there exist built-in functions to do most of the numerical operations you can think of (and a whole lot of numerical operations you have never thought of). How would you know if the operation you want to do is already built-in? One way to do this is through the exstensive help module. Open the help module and type in the search bar your favorite mathematical operation. For example, try typing in 'cross-correlation'. You will get several pages of documentation, among them is a help page for the function 'xcorr', that actually computes the cross-correlation function of two vectors.
If you know the name of the built-in script you want to get help about, another way to access the documentation on this script is to type:

```
>>help xcorr
```

This types the header of the script to the command window. If the programmer writing this script has done a good job, this will provide you with enough information to run the script. Don't be shy to use help – you will probably use this feature of Matlab more often than any other feature of the program.

Functions

A function is a self-contained block of code that performs a coherent task of some kind. 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 within the function are hidden.

Why Functions?
There are three main reasons for writing functions.
1) To make your code readable – hiding pieces of code in functions makes the overall structure of your code clearer and easier to read.
2) 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.
3) 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.
4) To save memory. When we create a function, we are only interested in the output of that function. All the other variables that are created within a function are only there in order to help us get to that answer. The memory taken by these variables will be freed once the function has terminated.

Whenever you write code, think about whether you will ever need that particular piece of code again – if you will, 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.)

Here is a very simple function.

```
function output=SimpleFunction(input)
%
% a very simple function
%
% written by if 4/2007

output=10*input;

```

The first line of your m-file defines this piece of code as a function. The terminology is that any variables you want to ***return*** from the function come ***before*** the equals sign. Variables you want to ***send*** into your function go inside the brackets ***after*** the function name.

You can send more than one variable in, and get more than one variable out.

```
function [output1, output2]=SimpleFunction2(input1, input2)
%
```

```
% Another very simple function
%
% written by if 4/2007

output1=input1.*input2;
output2=input1./input2;
```

Actually you have been using functions already, many of the commands you have been using already are functions – e.g. `round` – you give the command `round` a number as input and it returns the closest integer as the output.

We will see extensive use of functions next time, when we start programming an experiment.

Plotting

For completeness' sake, I add this section on plotting. One of the uses of matlab is creating visualizations of data. This can be done through a rather sophisticated system of plotting functions. We will use some of these when we turn to analysis of data from psychophysical experiments. You can get more information about plotting by using Matlab's interactive help.

Let's start by plotting a linear function:

```
>>figure(1)
>>x=0:5:100;
>>y=3*x+4;
>>plot(x, y)
```

The figure window has various properties. You can access them with the command:

```
>>get(gcf)
```

`gcf` stands for ***get current figure***. It basically provides a handle through which we can access the properties of the last figure that was referred to. It will tell you a lot about the current properties of the figure.

You can find out about the possible options for a figure using
```
>>set(gcf)
```

You can also change various figure properties using the set command
```
>>set(gcf, 'Color', [1 1 .5])
>>set(gcf, 'Position', [100  100  300 300])
>>set(gcf, 'Pointer', 'ibeam')
>>set(gcf, 'PaperOrientation' , 'landscape');
>>set(gcf, 'Name', 'My Data Figure');
```

If you have more than one figure and you want to be able to access and change properties on more than the most recent figure. In that case you need to create a handle to each figure.

```
close all
fh1=figure(1);
fh2=figure(2);
set(fh1, 'Color', [1 1 .5])
set(fh2, 'Color', [.5 1 1])
```

You can also put figure handles into an array

```
>>close all
>>clist=[1 1 .5; .5 1 1]
>>for i=1:2; fh(i)=figure(i); set(fh(i), 'Color', clist(i, :)); end
```

Subplots

You might want the figure to contain subplots. In Matlab each subplot is called an ***axis***. If you don't define the number of subplots then Matlab assumes you have a single axis (the whole figure).

```
>close all
>figure(1)
>subplot(2,2,1)
```

Like figures, subplots have various properties. You access them very similarly to the way that you access figure properties

```
>get(gca)
```

Here `gca` stands for ***get current axis***. It provides a handle to the last axis that Matlab created. It will tell you a lot about the current properties of the axis. Note that an axis has different properties from those of a figure. Once again you can look at the various options you have for axis properties using

Here's a table of the most useful properties of axes that you might want to change. But you can find and modify lots of other properties by using `get(gca)` and `set(gca)`

`>>help plot` is also a really good way of getting information about plotting

***** These axis properties can also be set as part of plot properties.

****** These axis properties are defined using a slightly different command structure

Axis property	***How to set the property***	***Comments***
Color	`set(gca, 'Color', [1 1 .7])`	Sets the color inside the subplot. Takes as input a single color.
*ColorOrder	`set(gca, 'ColorOrder', gray(8))`	Defines the order of colors used in plot commands. Takes a colormap as input
FontName	`set(gca, 'FontName', 'Times')`	Defines the font, uses normal computer fonts, e.g. 'Helvetica' , or 'Arial'
FontSize	`set(gca, 'FontSize', 12)`	Defines font size
FontWeight	`set(gca, 'FontWeight', 'bold')`	Thickness of the font, can take the properties of [light | normal | demi | bold]
*LineStyleOrder	`set(gca, 'LineStyleOrder', … {'-'; '--'; '-'})`	Defines the order of line styles used in plotting the figures. Takes a structure defining the line styles
LineWidth	`set(gca, 'LineWidth', 2)`	Sets the line width of the axis lines
** title	`title('my subplot ')` *or*	Title

	`title(gca, 'My Title')`	
** xlabel / ylabel	`xlabel('my X axis') / ylabel('my Y axis')` *or* `xlabel(gca, 'My x label')`	Labels x and y axis
** XLim / YLim *or* axis	`set(gca, 'XLim', [0 3])` `axis([0 3 0 1])`	Sets the X or Y limits of the plot. Two ways of setting the axes
XTick / YTick	`set(gca, 'XTick', 0:1.5:3)`	Decides where you want the tick values, takes as input a vector of where you want the tick values
XTickLabel / YTickLabel	`set(gca,'XTickLabel',… [5 35 100])` `set(gca,'XTickLabel', … {'One';'Two';'Three'})`	Decides what you want to label the ticks as instead of using the numbers in XTick. Can either take a vector, or a structure containing the strings you want to use as axis labels

Other useful commands are:

```
>>axis equal
```
sets the axis so tick marks are equally spaced on x and y axes

```
>>axis square
```
makes the current axis square

```
>>axis off
```
turns off all axis labeling, tick marks etc.

Like figures, if you have more than one subplot and you want to be able to access and change properties on more than the most recent subplot, you need to create a handle to each subplot, and use that to change the properties

```
>>close all
>>fh=figure(1)
>>set(fh, 'Color', [1 .6 1])
>>sp1=subplot(1, 2, 1);
>>sp2=subplot(1, 2, 2)
>>set(sp1, 'Color', [ 1 1 .6]);
>>set(sp2, 'Color', [.6 1 1]);
```

Line plots

Once again, plots have properties that you can alter. But for plots there is no handy command like gcf and gca that allow you to reference the most recent plot (it's possible, but tricky). So you should work on defining handles from the beginning if you are planning to change plot properties. The kinds of properties you can change actually depend on the kind of plot you are doing.

Here are some of the properties you may often want to change for a **line plot** (like the linear function we plotted before)

Line plot property	***How to set the property***	***Comments***
Color	`set(ph, 'Color', [ 1 0 0]) or` `set(ph, 'Color', 'r')`	Sets the color of the plot line. Color can either define red, green or blue guns or you can use a shorthand for some specific colors. Options include: 'b', 'g', 'r', 'c', 'm', 'y', k' {blue, green,

		red, cyan, magenta, yellow black}
LineStyle	`set(ph, 'LineStyle’, '--')`	Sets the style of the line. Options include: [{-} | -- | : | -. | none] (solid line, dashed line, dotted line, dash-dot line, no line)
LineWidth	`set(ph, 'LineWidth', 1)`	The thickness of the line
Marker	`set(ph, 'Marker', 'v')`	What the plot marker is. Options include: [+ | o | * | . | x | square | diamond | v | ^ | > | < | pentagram | hexagram | {none}] The v< >^ symbols represent triangles of various orientations
MarkerSize	`set(ph, 'MarkerSize', 5)`	The size of the markers
MarkerEdgeColor	`set(ph,'MarkerEdgeColor', [0 0 1])`	The color of the marker edging
MarkerFaceColor	`set(ph, 'MarkerFaceColor', [0 0 1])`	The color of marker fill

Legend and Text

In much the same way as we have manipulated figures, axes and plots using handles, legends and texts can also be manipulated:

`text` takes as input the x and y position of the string you want to use, and the string. These text objects can also be assigned handles and you can change their properties. To see what those properties can be, you simply use `get` or `set`.

```
>>th1 = text (10,20, 'This is a piece of text')
>>set(th1)
>>get(th1)
```

`legend` takes in a list of the plot handles for which you want to create the legend, and a cell array containing the strings that will be the legend labels.
Again, you can use `get` and `set` to look at properties of the legend. You can change things like the location or the font size or type.

```
>>lh=legend(ph1, {'x'})
>>set(lh, 'Location','West', 'FontName', 'Arial',  'FontSize',
10)
```

```
% plotExample.m
%
% Demonstrates the use of plotting commands and of  %
annotation of plots and figures
%
% ASR made it 7/2007, using code from IF

close all
x=0:2:100;
y1=2*x;
y2=3*x;
y3=3.5*x;

```

```
14      ph1=plot(x,y1, 'r-');
15      hold on
16      ph2=plot(x,y2,'g--');
17      ph3=plot(x,y3, 'b:');
18      title('fake data')
19      xlabel('my x data');
20      xlabel('my y data');
21      th1=text(x(13), y1(13), 'red line');
22      th2=text(x(15), y2(15), 'green line');
23      th3=text(x(17), y3(17), 'blue line');
24      set(th1,'Color', 'r')
25      set(th2,'Color', 'g')
26      set(th3,'Color', 'b')
27      lh=legend([ph1 ph2 ph3], {'x2'; 'x3'; 'x3.5'})
```

Line 15: 'hold on' tells matlab not to remove the old plots from this figure, when new plots are added. This can be reversed using 'hold off'

More plotting in 2D

As I mentioned before, Matlab plotting is rather sophisticated and enables you to create rather complicated plots. We continue with some more plotting commands in 2-dimensional plots

Errorbar plots

This type of plot presents errorbars.

```
close all
x=0:8:100;
y=3*x+4;
err=sqrt(y/4);
ph=errorbar(x, y, err)
```

In order to achieve maximal flexibility in determining the properties of the error bars separately from the properties of the plotting of the data, plot each of these separately (using '`hold on`' when adding a new plot to the figure).

Bar plots

You can also make barplots.

```
>close all
>x=0:5:20;
>y=[3*x+4]
>ph=bar(x, y)
```

Here are some of the properties you may often want to change for a simple bar plot

Line plot property	***How to set the property***	***Comments***
BarWidth	`set(ph, 'BarWidth', .5)`	Width of bars
LineWidth	`set(ph, 'LineWidth', 1.5)`	Thickness of lines surrounding bars
EdgeColor	`set(ph, 'EdgeColor',[1 0 0 ])`	Color of the lines surrounding bars

FaceColor	`set(ph, 'FaceColor',[0 0 1 ])`	Color of inside of bars

With more complex bar plots you get a handle for each set of data. Here we have two sets of y data for every x-value

```
close all
x=0:5:20;
y=[3*x+4;2*x+2]'
ph=bar(x, y)
set(ph(1), 'FaceColor',[1 0 1 ])
set(ph(2), 'FaceColor',[0 .3 1 ])
set(gca, 'XTickLabel', {'Day 1', 'Day 5', ...
    'Day 10', 'Day 15','Day 20'})
legend([ph(1) ph(2)],  'Happy', 'Sad', 'Location', 'NorthWest')
```

Bar plots with error bars

Matlab doesn't have a function to do this, so you need to use a work around. In the same way as I showed you how to have more control over you error bar plots by plotting the error bars on top of the original data plot, you can plot error bars on top of a bar graph. You may have to fiddle with the parameter width to make it look right.

```
hold on
width=get(ph(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')
```

Hist

`Hist` creates histograms. One way to use `hist` is simply to tell it how many bins you want

```
clear all
close all
data=randn(3000, 1);
subplot(1, 3,1)
hist(data, 9);
set(gca, 'XLim', [-4.5 4.5])
```

If you want to know what the center of the bins are you can ask `hist` to return a list of the number of data points in the bin (`n1`) and the centers of the bins (`x1`) instead of plotting the data. If you want to both know the centers of the bins and plot the histogram then you will need to call `hist` twice.

```
[n1, x1]=hist(data, 9)
```

You can also give `hist` a vector of the bin centers. Here we are simply returning 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=-4:1:4
subplot(1, 3,2)
n1=hist(data, bins);
bar(bins, n1)
set(gca, 'XLim', [-4.5 4.5]);
```

`histc` 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. We can plot this again, using bar

```
subplot(1, 3,3)
n2=histc(data, bins);
bar(bins,n2,'histc')
set(gca, 'XLim', [-4.5 4.5])
```

Note that the number of bins is 9. That means we have 9 bins in the second subplot with the bin centers of -4, -3, -2 … 2, 3, 4. In the third subplot we only have 8 bins, The first bin contains values from -4->-3, the second bin contains values from -3->-2, the third contains values from -2->-1 and so on …

Plotting in 3d

Mesh

Let's say you have an experiment in which the data is 3 dimensional. Suppose you were looking at how age affects your ability to perform a task as a function of the time of day. So you have two independent variables, the time of day and the age of the subject, and one dependent variable, which is performance. Your data might look like this

					Time of day			
		9	**11**	**1**	**3**	**5**	**7**	**9**
Age of subject	**17**	90	90	80	80	85	80	80
	17.5	90	80	80	80	90	90	95
	18	90	90	90	80	90	85	90
	18.5	80	90	75	70	80	90	85
	19	80	80	80	70	80	100	90
	19	85	80	90	75	90	85	85
	40	90	85	80	70	80	67	40
	45	100	80	80	70	80	58	30
	47	80	75	70	65	75	60	36
	50	80	80	70	80	70	60	40
	60	90	90	90	80	50	40	20
	65	90	90	90	70	80	40	30
	67	85	90	87	70	80	35	22
	68	80	85	80	75	65	45	22
	69	95	83	85	85	75	38	25

This gives us the following data in Matlab

```
>>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;
```

Note that that last line of code transforms the time variable into the 24 hour clock!

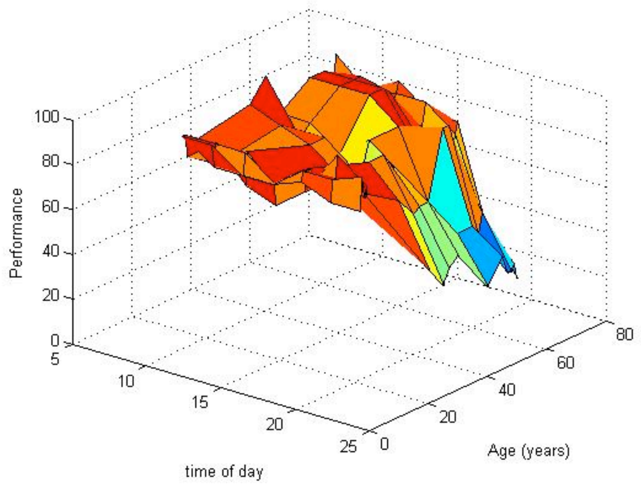
```
>>perf=[90    90    80    80    85    80    80
    90    80    80    80    90    90    95
    90    90    90    80    90    85    90
    80    90    75    70    80    90    85
```

```
     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];

  >whos
Name        Size                     Bytes  Class

  age         1x16                     128  double array
  ans         1x16                     128  double array
  perf       16x7                      896  double array
  tod         1x7                       56  double array
```

We are going to use mesh(x,y,Z). The first two vectors must have length(x) = n and length(y) = m where [m,n] = size(Z).

```
>> sh=surf(tod, age, perf)
>> xlabel('time of day')
>> ylabel('Age (years)')
>> zlabel('Performance')
```

You can rotate the view of this three-D graph using the mouse if you hit the little rotation icon on the top of the menu bar. You can now play with various aspects of the figure



Try
```
>> shading flat
>> shading interp
>> colormap(gray)
```

A useful command is `view`. If you rotate the graph to a good view, you might want to save the values that represent that view in a matrix `v` (a 4x4 matrix) so you don't have to manipulate it by hand the next time.

```
>>v=view
```

You can set the view to the desired view which you previously saved by using

```
>view(v)
```

Try saving a view, then rotate the graph manually, then use view to restore the graph to the view that you saved/
Or you can provide the graph with the azimuth an elevation you desire (the azimuth and elevation are basically simpler forms of the 4x4 matrix describing the view contained in `v`)

```
>view(60, 50)
```

There are also three default views

```
>view(1)
>view(2)
>view(3)
```

Also try

```
> mesh(tod, age, perf)
```

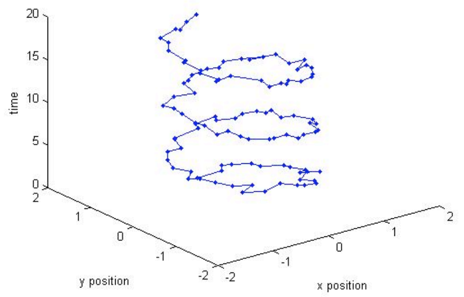
Surf and mesh can deal with missing data values as long as they are set to be `NaN.` It will simply interpolate between the missing values.

```
> perf(3,1)=NaN; > perf(5,3)=NaN; perf(7,2)=NaN;
> sh=surf(tod, age, perf)
```

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.

```
> t = linspace(0,6*pi,101);
> x = sin(t)+.1*randn(size(t));
> y = cos(t)+.1*randn(size(t));
> plot3(x, y, t, '.-');
> xlabel('x position');
> ylabel('y position');
> zlabel('time')
```



This rat is a little confused.

Let's suppose you were interested in seeing whether the rat reached a certain location (the food or the swim platform?) within a certain time window. First let's plot the position of this target on the time graph.

```
%plotExample2.m
%
%Example of plotting 3 dimensional data and locating a
certain area in the
```

```
%plot
%
%Plots the trajectory of a random process through space-time
(space 2d,
%time 1d).
%
%Written by IF (?) and commented by ASR 07/2007
%

close all
t= linspace(0,6*pi,101);
x= sin(t)+.1*randn(size(t));
y = cos(t)+.1*randn(size(t));
plot3(x, y, t, '.-');
xlabel('x position');
ylabel('y position');
zlabel('time');

twin=[10 17];
xwin=[ 1 2];
ywin=[-.5 1.5];

%line commands make a box in red around the time of interest
and location
%of interest:
lh=line([xwin(1) xwin(2)],[ywin(1) ywin(1)], [twin(xwin(1))
twin(xwin(1))]);
set(lh, 'Color', 'r')
lh=line([xwin(1) xwin(2)],[ ywin(1) ywin(1)], [twin(2)
twin(2)]);
set(lh, 'Color', 'r')
lh=line([xwin(1) xwin(2)],[ywin(2) ywin(2)], [twin(xwin(1))
twin(xwin(1))]);
set(lh, 'Color', 'r')
lh=line([xwin(1) xwin(2)],[ ywin(2) ywin(2)], [twin(2)
twin(2)]);
set(lh, 'Color', 'r')

lh=line([xwin(1) xwin(1)],[ywin(1) ywin(2)], [twin(xwin(1))
twin(xwin(1))]);
set(lh, 'Color', 'r')
lh=line([xwin(1) xwin(1)],[ ywin(1) ywin(2)], [twin(2)
twin(2)]);
set(lh, 'Color', 'r')
lh=line([xwin(2) xwin(2)],[ywin(1) ywin(2)], [twin(xwin(1))
twin(xwin(1))]);
set(lh, 'Color', 'r')
lh=line([xwin(2) xwin(2)],[ ywin(1) ywin(2)], [twin(2)
twin(2)]);
set(lh, 'Color', 'r')


lh=line([xwin(1) xwin(1)],[ywin(1) ywin(1)], [twin(xwin(1))
twin(xwin(2))]);
set(lh, 'Color', 'r')
lh=line([xwin(1) xwin(1)],[ ywin(2) ywin(2)], [twin(1)
twin(2)]);
set(lh, 'Color', 'r')
```

```
lh=line([xwin(2) xwin(2)],[ywin(1) ywin(1)], [twin(xwin(1))
twin(xwin(2))]);
set(lh, 'Color', 'r')
lh=line([xwin(2) xwin(2)],[ ywin(2) ywin(2)], [twin(1)
twin(2)]);
set(lh, 'Color', 'r')
```

This is clearly a little “wordy”. Here’s a way to shorten it up.

```
%plotExample3.m
%
%Same example as plotExample2.m, but shorter and less
"wordy".
%
%Made by IF (?) and commented by ASR 07/2007


close all
t= linspace(0,6*pi,101);
x= sin(t)+.1*randn(size(t));
y = cos(t)+.1*randn(size(t));
plot3(x, y, t, '.-');
xlabel('x position');
ylabel('y position');
zlabel('time');

twin=[10 17];
xwin=[0.7 1.7];
ywin=[-.5 1.5];

%'mat' represents the facets of the area of interest
mat=[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];

 %The lines are drawn with a 'for' loop:

 for i=1:size(mat, 1)
       lh=line([xwin(mat(i, 1)) xwin(mat(i, 2))], ...
           [ywin(mat(i, 3)) ywin(mat(i, 4))], ...
           [twin(mat(i, 5)) twin(mat(i, 6))]);
     set(lh, 'Color', 'r');
 end


ind=find(t>=twin(1) & t<twin(2) & ...
    x>=xwin(1) & x<xwin(2) & ...
47          y>=ywin(1) & y<ywin(2));
48
49      hold on
50      plot3(x(ind), y(ind), t(ind), 'g.-');
```

Lines 45-50. We’ve added some line finding out whether the rat made it into the box at the specified time, and we’ve re-plotted those points as green

Part 2 – Introducing Psychtoolbox

What is Psychtoolbox?

Psychtoolbox is a set of Matlab functions (m-files) and Matlab executables (.mex files) written in C, which allow you to write simple code in order to generate scripts that will run psychological experiments.
Psychtoolbox has several advantages over other software used for this purpose: It is very flexible - you can create almost any stimulus you can imagine (and describe mathematically!). It has amassed a large and quite active community of users and developers. It is open source and is given free of charge. The support of the community is also free of charge and is, in most cases, quite superb. It is currently the only package of its kind that is documented in peer-reviewed publications:

Brainard, D. H. (1997) The Psychophysics Toolbox, Spatial Vision , 10:443-446.
Pelli, D. G. (1997) The VideoToolbox software for visual psychophysics: Transforming numbers into movies, Spatial Vision 10:437-442.

Don't forget to cite these good people when you publish results collected using PTB.
Nevertheless, PTB does have some distinct disadvantages: all the code is written by people just like you, in the middle of trying to do real science. This means that commands may not work as stated, help files may be out of date; commands may not even exist. Also, PTB runs on a commercial package (Matlab) and thus also suffers some of the problems of using commercial packages (closed code, price, etc…).
We will discuss alternative options at the end of the tutorial.

In order to start futzing with PTB, you need to download Psychtoolbox from here:

http://psychtoolbox.org/

You'll have to download Psychtoolbox from the web site. Make sure you download the right version (for MacOSX or PC). Once you have installed Matlab, then you can carry on with the chapter.

Basic graphics

Like I said before, Psychtoolbox enables you to display any stimulus you can describe mathematically. In this section we will review how to describe basic graphics and go over some of the tools that Matlab has in order to create graphics.

Colormaps

Begin with the following lines:

```
>>img = 1:5;
```

This is simply a vector with five elements.
Then put up a figure window in matlab:

```
>>figure(1)
>> 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]
>>colormap(paintpots1);
```

I'll explain this colormap command in just a minute.

```
>>image(img)
```

The command `image(img)`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.

In 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')
```

Now the bars will be horizontal and will increase in brightness as you go from the top to the bottom.

```
>>axis off
>>axis square
```

This will get rid of the axes

```
>>paintpots2=[0 0 1; 1 0 0; 0 1 0; 0.5 0 1; 1 0 1];
>>colormap(paintpots2)
```

Whoah! Why does the image suddenly change color?

`paintpots` represent two different sets of paints.

In `paintpots1` the values 1-5 was represented by different levels of gray. The command to tell Matlab to paint the image img using paintpots1 was `colormap(paintpots1)` . Basically a ***`colormap`*** is a *mapping* between a set of indices in an image (your paint-by-numbers picture) and a set of *colors* (your picture). You may also hear the word ***palette*** to describe a colormap – the two words mean the same thing.

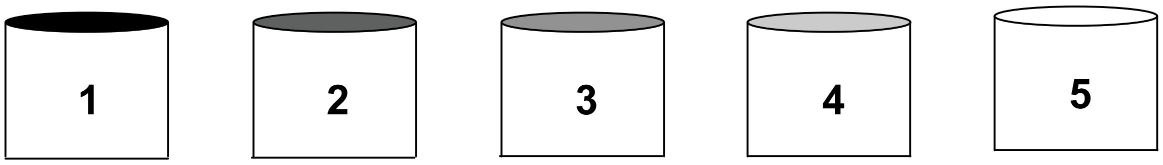
If you look at paintpots1 you will see that it is a 5x3 matrix.

```
>>size(paintpots1)
```

The 5 rows refer to the fact that paintpots has five paint pots (indices into the colormap).
The 3 rows refer to the intensity of the red, green a blue gun. These gun intensities go between 0 (black) and 1 (white).

```
>>paintpots1
```

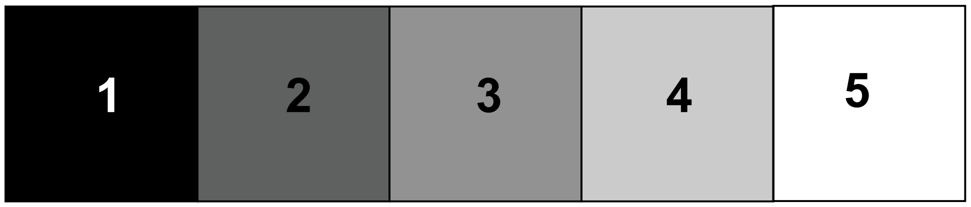
So the first paint pot contains black, the fifth paint pot contains white, and the intermediate ones contain gradually lighter colors of gray.



This is why

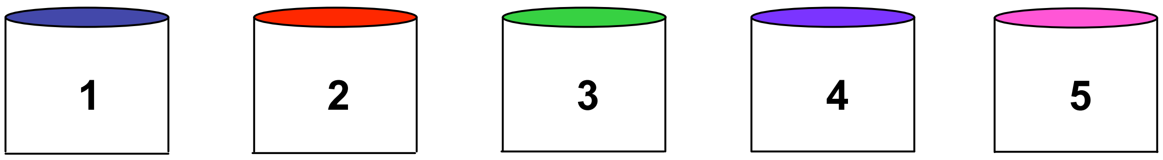
```
>image(img)
>colormap(paintpots1)
```

produces an image that gradually goes from black to white.

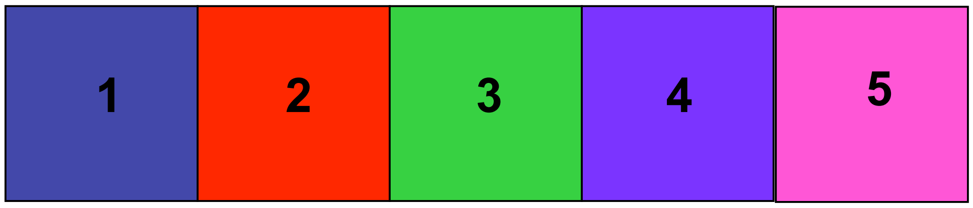


Now lets look at paintpots2. Once again, paintpots2 represents 5 different colors.

```
>paintpots2

paintpots2 =
        0      0     1   % only the blue gun, at maximum, this makes blue
        1      0     0   % only the red gun, at maximum, this makes red
        0      1     0   % only the green gun, at maximum, this makes green
        0.5    0     1   % some red and lots of blue, makes purple
        1      0     1   % red and blue, makes pink
```



So when you paint the img using this set of paintpots (a new colormap) the image changes.



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

```
>>paintpots3=hot(5);
>>colormap(paintpots3);
>>paintpots4=hsv(5);
>>colormap(paintpots4);
```

Also check out `winter, spring, cool, jet.`
You can find a full list of Matlab colormaps using

```
>>help graph3d
```

Spend some time playing with colormaps. For example, see if you can change a single row of a colormap. For example if you want to replace purple with yellow:

```
>>paintpots2b(4, :)=[1 1 0]
```

You should be able to guess what will happen if you use paintpots2b as your colormap.
Now see if you can remove all the red from paintpots1.
Now we are going to look at a little m-file, which will allow you to gradually change an image by changing the colormap:

```
1       % UsingColormaps.m
2       %
3       % a little program that manipulates colormaps
4       %
5       % written by IF and GMB 3/2005
6
7       clear all
8       close all
9       img=1:10;
10      figure(1)
11      paintpots = ones(10,3);
12      colormap(paintpots)
13      image(img);
14      axis off;
15      for i=1:10
16          paintpots (i,:)= (i/10);
17          colormap(paintpots);
18          pause
19      end
```

Lines 15-19. As you go from i = 1 to 10, you are gradually replacing the 1's in paintpots with grays that go from 0.1 (very dark) to 1 (white). Then you replace the old colormap with the new colormap, and gradually the white in the image is replaced by grays. .

Line 18. The pause makes the program wait for a key press so you can observe each step.
Now we are going to replace lines 3-13 in UsingColormaps.m

```
1       % UsingColormaps2.m
2       %
3       % Another little program that manipulates colormaps
4       %
5       % written by IF and GMB 3/2005
6
7       clear all
8       close all
9
10      colormap(gray(256))
11      img = reshape(1:256,16,16);
12      image(img);
13      axis square
14      axis off
15      pause
16      for i=1:200
17          paintpots = rand(256,3);
```

```
18          colormap(paintpots);
19          drawnow
20      end
```

I hope that was at least just a little bit exciting!

Line 11: The command ***reshape*** takes 3 arguments and 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 give reshape a vector that goes [1 2 3 …256]. The second argument is the number of rows you want the new matrix to have, and the second argument is the number of columns you want the new matrix to have. So in this case reshape turns a 1x256 vector into a matrix with 16 rows and 16 columns. Of course this only works because 16x16=256. Otherwise you would get the following error

```
>>img = reshape(1:254,16,16);
??? Error using ==> reshape
To RESHAPE the number of elements must not change.
```

Take a look at img

```
>>img
```

Here's another example of using reshape:

```
>>tmp=[3 4 4 5 1 2; 6 7 1 2 2 3]
>>tmp2=reshape(tmp, 3, 4)
>>tmp3=reshape(tmp, 12, 1)
```

Lines 12-14: simply draws the matrix img, removes the axis and makes the image square. Waits for a keypress.
Lines 16-20: i will step from 1, 2, 3 …200. Each time, lines 17-19 will be executed. You remember how a `colormap` is like a collection of paint pots – with each paint pot having a number label. Here we have 256 paint pots, and the colors inside the paint pot are being assigned randomly. `rand(256,3)` creates a 256 x 3 matrix of random numbers. The random numbers vary between 0-1, so that allows us to define 256 random colors to go into the paint pots.
(A brief digression: if you type just the command `rand` then Matlab will just give you a single randomly chosen number from a uniform distribution. Otherwise you have to describe the size of the matrix of random numbers that you want, as is done here. Try `help rand` for more details)
Line 19: Tells Matlab to update the figure. Normally updating figures have a pretty low priority for Matlab, `drawnow` tells Matlab to put the other calculations on hold until it's updated the figure.

Making a Gabor

Here we are going to use some of the things we have learned to make a Gabor filter. These are regularly used for image processing. They are also popular as stimuli among vision scientists.

```
>>sd=.3;
>>x=linspace(-1,1,100);
>>y=(1/sqrt(2*pi*sd)).*exp(-.5*((x/sd).^2));
>>y=y./max(y);
```

So `y` is a 1-d Gaussian with a standard deviation of `sd=0.3.` It has a minimum value of 0 and a maximum value of 1. We can `plot` this and see what it looks like.

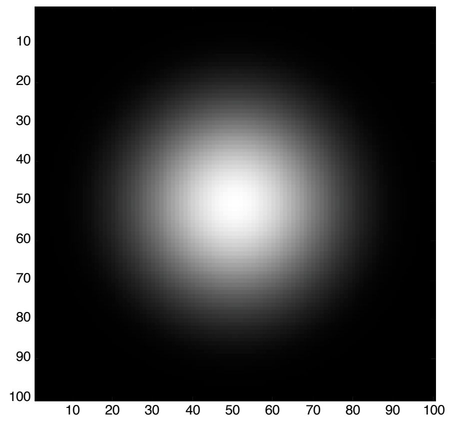
```
>>plot(x, y);
```

Now we use the outer product to create a two-dimensional Gaussian filter, with a maximum value of 1

```
>>filt=(y'*y);
>>max(filt(:))
```

And then we can look what that two dimensional Gaussian looks like. We are going to allow the colormap to take 256 possible values of gray. That means we want filt to vary between 1 and 256.

```
>>colormap(gray(256));
>>scaled_filt=scaleif(filt, 1,256);
>>image(scaled_filt)
```

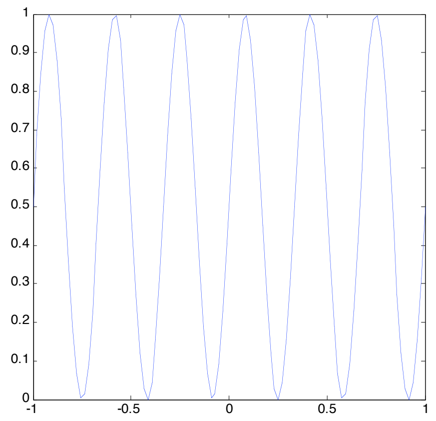


We can then use this Gaussian window to filter any image we want. Let's say we want to filter a sinusoidal grating. We actually use much the same set of tricks.

```
>>sf=6; % spatial freq in cycles per image
>>y2=sin(x*pi*sf);
>>y2=scaleif(y2, 0, 1);
```

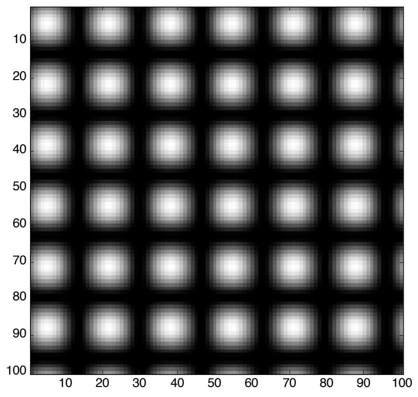
scaleif is a custom-made function, which you can download from the tutorial website.Wonder what it does? Type `'help scaleif'`

```
>>plot(x, y2);
```



This now gives us a 1-dimensional sinusoidal grating with a frequency of 6 cycles per image. We've scaled it so it has a minimum value of 0 and a maximum value of 1. If we calculate the outer product of this sinusoid with itself we get a weird checkerboard.

```
>>img=(y2'*y2);
>>colormap(gray(256))
>>scaled_img=scaleif(img, 1, 256)
>>image(scaled_img)
```

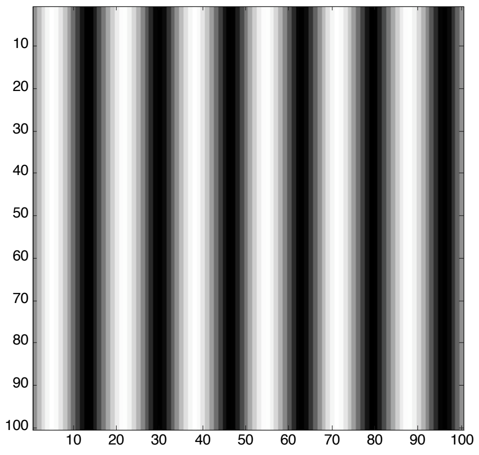


What we want to do is calculate the outer product of the one-dimensional sinusoid with a vector of ones.

```
>>y3=ones(size(y2));
>>img=(y3'*y2);
>>colormap(gray(256))
>>scaled_img=scaleif(img, 1, 256)
>>image(scaled_img)
```

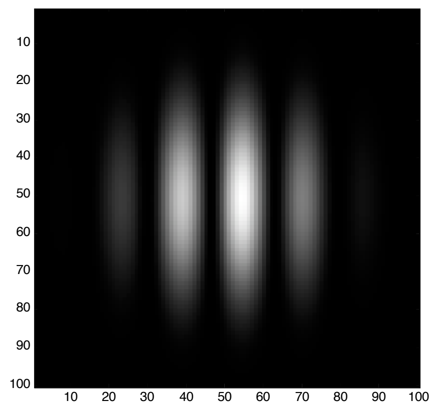
Once again, img has a minimum of 0 and a maximum of 1

```
>>max(img(:))
```



Now to create a Gabor (a sinusoid windowed by a Gaussian) becomes a piece of cake. We simply multiply the two-dimensional Gaussian window by the two-dimensional Gabor.

```
>>gabor=img.*filt;
>>scaled_gabor=scaleif(gabor, 1, 256);
>>colormap(gray(256))
>>image(scaled_gabor);
```



Note that throughout this process we have worked with two versions of the Gaussian and the Gabor. The original versions (`filt, img`) were scaled between 0 and 1. We did this to keep the nice property that when the Gaussian filter was at its maximum, the brightness of the grating didn't change, and when the Gaussian filter was at 0 the Gabor was also at 0. But when we used `image` to look at these matrices we converted them to range between 1-256 so as to match the colormap.

This gives you some basic stuff to start with.
Again – the limit of making visual stimuli is your imagination and your ability to describe what you imagine in numbers.

Getting started with PsychToolbox Screen.m

In order to know if PTB is properly installed try running the following:

```
>>ScreenTest
```

If you get the screen going blank and then something like the following then `screenTest` worked

```
***** ScreenTest: Testing Screen 0 *****


PTB-INFO: This is the OpenGL-Psychtoolbox version 3.0.8. Type
'PsychtoolboxVersion' for more detailed version information.
PTB-INFO: Psychtoolbox is licensed to you under terms of the GNU
General Public License (GPL). See file 'License.txt' in the
PTB-INFO: Psychtoolbox root folder for a copy of the GPL license.


PTB-INFO: OpenGL-Renderer is NVIDIA Corporation :: GeForce Go
7400/PCI/SSE2 :: 2.0.1
PTB-Info: VBL startline = 768 , VBL Endline = -1
PTB-Info: Measured monitor refresh interval from VBLsync =
16.712593 ms [59.835118 Hz]. (50 valid samples taken,
stddev=0.044112 ms.)
PTB-Info: Reported monitor refresh interval from operating system
= 16.666667 ms [60.000000 Hz].
```

```
PTB-Info: Small deviations between reported values are normal and
no reason to worry.
PTB-INFO: Using NVidia's GL_TEXTURE_RECTANGLE_NV extension for
efficient high-performance texture mapping...

***** ScreenTest: Done With Screen 0 *****
```

If not, then Psychtoolbox isn't working right on your computer

Writing code using Screen.m

For now, let's go back to the basics. We are simply going to open an experimental window, making it black, making it white, and then closing it again.

The script below uses a command called `Screen`, which is one of the core functions of Psychtoolbox. Screen is actually not an m file, like the ones you have probably been writing in Matlab. It is a mex file. This means that it is written in C (or C++) or some other programming language, and is then compiled to run in Matlab. The reason this was done is because Screen does some pretty funky stuff that would be impossible in Matlab.

```
% DarkScreen.m
%
% opens a window using Psychtoolbox,
% makes the window black, then white, and then closes
% the window again
%
% written for Psychtoolbox 3 on the PC by IF 3/2007

screenNum=0;
res=[1280 1024];
clrdepth=32;
[wPtr,rect]=Screen('OpenWindow',screenNum,0,...
[0 0 res(1) res(2)], clrdepth);
black=BlackIndex(wPtr);
white=WhiteIndex(wPtr);
Screen('FillRect',wPtr,black);
Screen(wPtr, 'Flip');

HideCursor;
tic
while toc<3
    ;
end

Screen('FillRect',wPtr,white);
Screen(wPtr, 'Flip');
HideCursor;
tic
while toc<3
    ;
end


Screen('CloseAll');
ShowCursor
```

Line 9. If you are running more than one monitor on your computer then Screen tells you which one to use. 0 means the monitor with the menu bar (or in Mac OSX, the dock) 1 means the other monitor. For now, if you have any difficulty I would make sure you are only using one monitor. If you are cloning your monitor then Screen 1 becomes the display screen.
Line 10-11. Check the resolution of your screen and the using. As far as the resolution of the monitor in pixels is concerned the first number is the width of the screen, the second number is the height of the screen. The resolution of your screen in terms of color depth will be 8, 16 or 32 bits, depending on the age of your monitor or computer. Set it to the highest setting the computer will allow.
Line 12. This calls a function called Screen, which takes 5 arguments.
Argument 1. a command telling Screen what to do – in this case you want Screen to open a window. *Argument 2.* which monitor you want the window opened inside. In this case you want the Screen opened in monitor 0 – the one with the menu bar.
Argument 3. The color you want to fill the window with. This currently doesn't seem to work in Psychtoolbox.
Argument 4. This tells Screen how big you want the window to be – in this case you want it to be a rectangle the size of the entire screen. The order in which you describe this rectangle can be remembered as **Le**T**te**RB**ox** (Left, Top, Right, Bottom). We want the rectangle to start 0 pixels from the Left, and 0 pixels from the Top, and go to 1280 pixels towards the Right and 1024 pixels towards the Bottom.
(Why does it start with 0 instead of 1? Because Screen is a mex file and was written in C which is a 0-based language.) Currently Psychtoolbox always uses the whole screen, no matter what you enter into rect. *Argument 5* is the color depth of the monitor.
Arguments 3-5 don't actually need to be specified – Matlab will default to certain values: a random background color, the whole screen and the color depth of the monitor as described in the control panel
Lines 14 - 15 Find the colormap values that will give you black and white.
Line 16 Draw a black rectangle the size of the screen. Psychtoolbox automatically assumes that you have an offscreen window and an onscreen window. Every time you use a drawing command like DrawRect it will automatically draw on the offscreen window. So the rectangle won't yet be visible. The advantage of this approach (as you will see later) is that you can spend some time drawing several things offscreen without them showing up one by one on the screen. Once you have finished drawing you can move the offscreen window to the front.
Line 17 You need to `Flip` the screen so the offscreen window you drew the black rectangle on comes to the onscreen window – the one that is actually on the monitor.
Monitors have a ***refresh*** rate (of something between 60Hz-120Hz for a standard monitor). Every refresh begins at the top of the screen, and moves quickly down the screen in a matter of a few milliseconds. The flip command simply tells Matlab to refresh the screen with the image you have drawn offscreen.

Lines 19-23 Hide the cursor and wait 3 seconds
Lines 25-31 Draw a white rectangle on the offscreen window, flip it to the front and wait 3 seconds
Lines 34-35. Close the screens and re-appear the cursor

One weird thing about Screen is that you don't get help about it's commands in the normal way.

If you type:

```
>>help Screen
```

You get a lot of info, but it doesn't really tell you how to use `Screen`.
If you try:

```
>>Screen
```

You will get a list of all the subcommands that are contained within Screen. To get more information about a particular command you do:

```
>Screen OpenWindow?
>Screen DrawText?
```

****

NOTE FOR IF YOUR SCREEN FREEZES
If you display stimuli on the main screen, as we often do, then the Screen window will hide the main menu bar and obscure Matlab's command window. That can be a problem if your program stops (perhaps due to an error) before closing the window. The keyboard will seem to be dead because its output is directed to the front-most window, which belongs to Screen not Matlab, so Matlab won't be aware of your typing.

Remain calm.

Typing Ctrl-C will stop your program if hasn't stopped already. Typing:

command-zero (on the Mac)
Alt-Tab (on Windows)

Will bring Matlab's command window forward. That will restore keyboard input.
Typing:

```
>>sca
```

will call Screen('CloseAll'). This will close all the open screens and will cause the cursor to reappear (if you hid it)
The screen might still be hard to make out, if you've been playing with the lookup table.
Typing:

```
>>clear Screen
```

Should flush all the changes made by Screen from memory and restore your screen to normal. If that doesn't work, do the usual dance – quit Matlab and restart it. If all else fails, throw away your pc and get a mac.
****

Here's another more elaborate example of how you can use `Screen`

```
% FunkyScreen.m
%
% opens a window using psychtoolbox,
% makes the window do some funky things
%
% written for Psychtoolbox 3 on the PC by IF 3/2007

screenNum=0;
flipSpd=13; %  a flip every 13 frames

[wPtr,rect]=Screen('OpenWindow',screenNum);

monitorFlipInterval=Screen('GetFlipInterval', wPtr);
% 1/monitorFlipInterval is the frame rate of the monitor

```

```
black=BlackIndex(wPtr);
white=WhiteIndex(wPtr);

% blank the Screen and wait a second
Screen('FillRect',wPtr,black);
Screen(wPtr, 'Flip');
HideCursor;
tic
while toc<1
    ;
end

% make a rectangle in the middle of the screen; flip colors and size
Screen('FillRect',wPtr,black);
vbl=Screen(wPtr, 'Flip'); % collect the time for the first flip with vbl
for i=1:10
    Screen('FillRect',wPtr,[0 0 255], [100 150 200 250]);
    vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
% flip 13 frames after vbl
    Screen('FillRect',wPtr,[255 0 0], [100 150 400 450]);
    vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
end

% blank the screen and wait a second
Screen('FillRect',wPtr,black);
vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
tic
while toc<1
    ;
end

% make circles flip colors & size
Screen('FillRect',wPtr,black);
vbl=Screen(wPtr, 'Flip');
for i=1:10
    Screen('FillOval',wPtr,[0 180 255], [ 500 500 600 600]);
    vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
    Screen('FillOval',wPtr,[0 255 0], [ 400 400 900 700]);
    vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
end

% blank the Screen and wait a second
Screen('FillRect',wPtr,black);
vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
tic
while toc<1
    ;
end

% make lines that flip colors size  & position
Screen('FillRect',wPtr,black);
vbl=Screen(wPtr, 'Flip');
for i=1:10
    Screen('DrawLine',wPtr,[0 255 255], 500, 200, 700 ,600, 5);
    vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
    Screen('DrawLine',wPtr,[255 255 0], 100, 600, 600 ,100, 5);
```

```
74       vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
75   end
76
77   % blank the Screen and wait a second
78   Screen('FillRect',wPtr,black);
79   vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
80   tic
81   while toc<1
82       ;
83   end
84
85   % combine the stimuli
86   Screen('FillRect',wPtr,black);
87   vbl=Screen(wPtr, 'Flip');
88   for i=1:10
89       Screen('FillRect',wPtr,[0 0 255], [100 150 200 250]);
90       Screen('DrawLine',wPtr,[0 255 255], 500, 200, 700 ,600, 5);
91       Screen('FillOval',wPtr,[0 180 255], [ 500 500 600 600]);
92       Screen('TextSize', wPtr , 150);
93       Screen('DrawText', wPtr, 'FUNKY!!', 200, 20, [255 50 255]);
94       vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
95
96
97       Screen('FillRect',wPtr,[255 0 0], [100 150 400 450]);
98       Screen('FillOval',wPtr,[0 255 0], [ 400 400 900 700]);
99       Screen('DrawLine',wPtr,[255 255 0], 100, 600, 600 ,100, 5);
100      vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
101  end
102
103  % blank the screen and wait a second
104  Screen('FillRect',wPtr,black);
105  vbl=Screen(wPtr, 'Flip', vbl+(flipSpd*monitorFlipInterval));
106  tic
107  while toc<1
108      ;
109  end
110
111  Screen('CloseAll');
112  ShowCursor
113
114
```

Line 9. Here you define `flipSpd`, you are going to make the display flip every 13 frames.
Line 11. This time when you open the monitor you let the monitor choose the resolution and the color depth. The monitor actually returns the resolution in rect.
Line 13 You can find out how fast your monitor flips using `'GetFlipInterval'`
Line 31. This time when you flipped the window you made `Screen` return the time (according to the computer clock) that it did the flip. That time is saved as `vbl`. VBL stands for “Vertical Blanking”, which is a synonym of the vertical retrace of the screen by the electron beam that draws the screen display.
Line 33. Here we are using `FillRect` again, but this time we are defining the rect as only being a subset of the entire screen. Remember that the `rect` is defined as LeTteRBox. The other difference is that this time, instead of sending it a single number that is an index into the colormap, we are sending it 3 values - for the red, green and blue guns. There is a weirdness here that the monitor thinks that it is a 32 bit monitor, but these values are on an 8 bit scale. This is because allowing the red gun to take any number between 0-255 takes up 8 bits, allowing the green gun to take any number between 0-255 takes up 8 bits, allowing the blue gun to take any number between 0-255 takes up 8 bits. 3x8 is 24. The other 8 bits are padding – don’t ask me what they are used for.

Line 34: We `flip` the screen, but this time we are telling it to flip at time vbl + (13 * monitorFlipInterval). This means it will flip 13 frames after the flip on line 31. Once again the time that the monitor was flipped is saved as vbl.
Lines 36-37. We do the same thing again, but this time `vbl` refers to the flip that happened on line 34. So once again there is a 13 frame wait before the flip.
Line 53. `FillOval` works just like `FillRect` except that it draws ovals instead of squares.
Line 71. `DrawLine` doesn't take a `rect` as an argument. Instead it takes in the starting horizontal position, starting vertical position, ending horizontal position, ending horizontal position. I guess the mnemonic for that would be **H**um**V**ee
Lines 89-92 Here we are drawing more than one thing on the offscreen window before we flip them to the front
Lines 93-94. Here we are drawing text on the screen. First we define the size of the text as having a font size of 150. Then we draw it on the screen. 200, 20 refer to the starting horizontal and vertical positions of where you want the text placed (**H**um**V**ee again). `[255 50 255]` refers to the color of the text.

PsychDemos

If you are wondering what kind of thing you can do with PTB and how some of your ideas can be coded up, look at the demos that ship with PTB. Try typing:

```
>>help PsychDemos
```

This will give you a list of available demos and a short description of what they do. If you are curious what a certain demo does you can inquire further. For example, type:

```
>>help MandelbrotDemo
```

This will tell you what this script does. If you are curious how this is implemented, type:

```
>>edit MandelbrotDemo
```

This will open the file MandelbrotDemo.m in an editor window. Don't edit this file! You might cause some damage. Instead, save the file under a new name. For example, 'myMandelbrotDemo.m'. Now you can twiddle things in the file and try to see what effect these changes have on the execution of the program. But before you start doing that, let's get acquainted with the single most important function in Psychtoolbox.

Key Presses

In most experiments that we do presenting the stimulus is only half of the action. Collecting responses from our subjects is key (har, har)! Here we will go through the various ways that you can collect keypresses and discuss their strengths and weaknesses. Different techniques are useful for different purposes. This table should help you keep these commands straight in your head.

	Psychtoolbox windows	Timing	What was pressed	Waits for keypress	NOTES
`Pause`	NO	NO	NO	YES	Just waits for a specified period of time or for any key to be pressed on the keyboard
`Input`	NO	NO	YES (Matlab	YES	Just waits for a key to be pressed on the

			expression)		keyboard and returns the character or number of that key
`CharAvail`	YES	NO	NO	NO	Simply checks whether there is a key press in the event queue
`GetChar`	YES	Dubious accuracy	YES (character)	YES	Checks or waits for a key press in the event queue. Returns what the key was, and when it was pressed
`KbCheck`	YES	Good	YES (key)	NO	Tests whether a key has been pressed at that moment in time
`KbWait`	YES	Good	NO	YES	Waits for a key press to occur

Collecting keypresses using pause and input

These techniques are useful when you are not using Psychtoolbox. Remember that none of these techniques have good timing. It's easiest practicing these commands in an m-file, since many of these commands will accept Return as a key press (if you type them into the command line, the Return you use at the end of the line will also be used as the input into the key press command).

pause

This is the simplest command you can use to collect a key-press. It simply pauses the computer until a key (any key) is pressed on the computer and doesn't collect what that response is.
Bear in mind that the command window needs to be in front for the keypress to be available to Matlab, so pause doesn't work well if you are using Psychtoolbox. In that case use `GetChar` instead.
Pause can also be used to enforce a delay –

```
>>pause(3)
```

pauses for 3 seconds.

input

This command again pauses the computer until a key is pressed but it allows you to collect the response. By default the response is a number

```
>>resp_num=input('press a number key ...')
```

But you can also specify that `input` will accept a string, as shown below. In that case if the subject inputs a number key the computer will assume that the subject chose the character '3' rather than the double 3.

```
>>resp_char=input('press a number key ...', 's')
>>whos
```

Note that `resp_num` is a double and `resp_char` is a character.

If in a program you want to only accept a certain type of response, then you need to write a loop like this one:

```
% PracticeKeyPresses
% a program to practice different ways of collecting
% key press responses
%
% written if 4/2007

% using input
disp('Using input command')
resp='x';
while resp~='a' & resp~='b'
    resp=input('press a or b ... ', 's');
end
resp

```

Bear in mind that the command window needs to be in front for the key-press to be available to Matlab, so `input` doesn't work well if you are using Psychtoolbox. In that case you will need to use `GetChar`, `CharAvail` or `KbCheck` instead.

Collecting keypresses using GetChar & CharAvail

When you type into the keyboard (or any input device) the key presses get saved into an event queue. This is why sometimes when you type really fast and the computer is also busy with other tasks there will be a pause before the letters suddenly appear in your document. `CharAvail` and `GetChar` commands read from the event queue. It's therefore important to empty events from the event queue before collecting responses using these two commands

```
>>FlushEvents
```

Before using any of the following commands.

```
>>CharAvail
```

looks to see if there is anything in the event queue.

```
[avail, numChars] = CharAvail;
```

`avail` will be 1 if characters are available, 0 otherwise. `numChars` may hold the current number of characters in the event queue, but in some system configurations it is just empty, so do not rely on `numChars` providing meaningful results, unless you've tested it on your specific setup. If `avail` is 1 then you need to call `GetChar` to find out what the actual key press is – `CharAvail` just tells you whether there was a key press.

```
% CharAvail
WaitSecs(1)
disp('Using CharAvail command')
disp('Wait 2 sec to see if you pressed a key during that
time');
FlushEvents
tic
while toc<2
 ;
end
resp = CharAvail;
```

```
resp
```

CharAvail is useful when you want to see whether your subject pressed a key while you were doing something else (such as displaying images) in the meantime. It's especially useful if you are doing something like fmri where you don't want to wait indefinitely for a subject response.

GetChar

If there is something already in the event queue then `GetChar` retrieves it. If there isn't something in the Event queue then GetChar waits for a typed character and lets you save the response.

It's used as follows:
`[resp, when] = GetChar([getExtendedData], [getRawCode]);`

`char` is the character that was typed
`when` is a structure. It returns the time of the key press, the "adb" address of the input device, and the state of all the modifier keys (shift, control, command, option, alphaLock) and the mouse button. If you have multiple keyboards connected, address may allow you to distinguish which is responsible for the key press. `when.secs` is an estimate of the time when the key press happened. However the timing of `GetChar` is not necessarily reliable, the reported values can be off by multiple dozen or even hundreds of milliseconds. If you are interested in precise timing you should check the timing of `GetChar` on the particular system that you are using or use `KbWait` or `KbCheck` instead.

`getExtendedData` tells `GetChar` whether or not to collect when data, if set to 0 then `GetChar` will be a tiny bit faster. `getRawCode` determines whether you want to return the character information in ascii or char (the default) format. Normally you won't bother using either of these input arguments.

Add the following lines to your `PracticeKeyPresses` m-file. If you press a key GetChar behaves very like CharAvail except that it retrieves what the character is. If you don't then GetChar waits

```
%Using GetChar
WaitSecs(1)
disp('Using GetChar command just on its own')
FlushEvents
tic
while toc<1
    ;
end
[resp_GC, when_GC]=GetChar;

disp('now you have pressed a key')
resp_GC

```

Note that even though the command line may appear, the program doesn't actually continue to print `'now you have pressed a key'` until you press a key – if there's nothing in the event queue then `GetChar` will wait for a response.

What if you only want to accept certain responses (e.g. the 'm' or 'f' keys)?

```
%GetChar - only accepting certain responses
WaitSecs(1)
```

```
disp('Using GetChar command - only accepting certain
responses')
FlushEvents
resp_GC2='x';
while resp_GC2~='m' & resp_GC2~='f'
    disp('press m or f ... ');
    resp_GC2=GetChar;
end
resp_GC2
```

What if you want to combine the useful property of CharAvail that it doesn't wait indefinitely for a response with the useful property of GetChar that you can find out what the response was?

```
% Combining GetChar and Char Avail
WaitSecs(1)
disp('Combining GetChar and Char Avail')
disp('Wait 2 sec to see if you pressed a key during that
time');
FlushEvents
tic
while toc<2
    ;
end
resp_CA3= CharAvail;
if resp_CA3==1
    resp_GC3=GetChar;
    disp(['You pressed key', resp_GC3])
else
    disp('You did not press a key');
end
```

One more thing to remember

KbCheck and KbWait are MEX files, which take time to load when they're first called. They'll then stay loaded until you flush them (e.g. by changing directory or calling CLEAR MEX). So your timing on the first trial will be more accurate if you just call them at the beginning of the program.

Collecting keypresses using KbCheck & KbWait

`KbCheck` and `KbWait` don't pull key presses out of the event queue. Instead they monitor the state of the keyboard at that very moment. The advantage of this is that they tend to have more accurate timing on some systems, and there is no lag – the minute the key press happens the `KbCheck` and `KbWait` command knows that the key press has happened. The disadvantage is that it's hard to do something else (e.g. display images) while constantly monitoring the state of the keyboard.

One thing that is a little weird about `KbCheck` and `KbWait` is that they actually refer to the key that was pressed on the keyboard rather than the character represented by that key. This is because they are lower level commands. What character a key represents actually varies across computer systems, which means you need to convert your key press information into character information.

While most keynames are shared between Windows and Macintosh, not all are. Some key names are used only on Windows, and other key names are used only on Macintosh. For a lists of key names common to both platforms and unique to each see the comments in the body of KbName.m.

KbName will try to use a mostly shared name mapping if you add the command KbName('UnifyKeyNames'); at the top of your experiment script. In fact, **KbName often won't work unless you add this line to the beginning of your scripts!**

Psychtoolbox has a great demo for KbCheck and KbWait which is called KbDemo. You can try it by just typing KbDemo at the command window:
>KbDemo
This is also a good way of making sure that KbCheck is working. If not, try replacing the version of KbCheck.m that you have with a revised version found here:

http://en.wikibooks.org/wiki/Psychtoolbox:KbCheck

The examples below are simplified versions of taken from KbDemo

KbCheck

Checks to see whether a key on the keyboard is pressed down at that very moment in time.

```
 [keyIsDown,secs,keyCode] = KbCheck;
```

`keyIsDown` is 1 if *any* key, including modifiers such as <shift>,<control> or <caps lock> is down. 0 otherwise.

`secs` is the time of the key press as returned by `GetSecs`. This is an accurate way of getting timing but you should still read the help file for `KbCheck` and `GetSecs` carefully if your experiment depends on accurate timing.

`keyCode` On Macs this is a 128-element array, on PCs it is a 256 . Each number in the array represents a keyboard key. If a key is pressed that index in the array is set to 1, otherwise it will be set to 0. To convert a `keyCode` to a vector of key numbers that were pressed use `find(keyCode).` To find out what actual key that was use `KbName`.

KbWait

Just like KbCheck, but it waits until a key on the keyboard is pressed down and simply returns the time that the key was pressed.

```
secs = KbWait;
```

`secs` is the time of the key press as returned by `GetSecs`.

KbName

KbName maps between KbCheck-style keyscan codes and the character that key represents. Use KbName to make your scripts more readable and portable, since keycodes, are cryptic and vary between Mac and Windows computers.

```
kbNameResult = KbName(arg)
```

KbName actually will let you go either way from keycodes to characters, or vice versa. If you send in as input a string designating a character then KbName returns the keycode for that character.
```
>KbName('t')
```

If on the other hand you send in a number representing the keycode, then KbName will return the character of that keycode
```
>KbName(84)
```

The numbering of these practice examples matches their order in KbDemo, but I'm going through them in order of how complicated they are. That's why the naming has a funny order (we start with KbPractice3)

```
% exampleKb
% Wait for a key with KbWait.
% a simplified version of KbDemo by IF 4/2007

WaitSecs(0.5);
```

```
6       disp('Testing KbWait: hit any key.  Just once.');
7
8       startSecs = GetSecs;
9       timeSecs = KbWait;
10
11      [keyIsDown, t, keyCode ] = KbCheck;
12
13      str=[KbName(keyCode), ' typed at time ', …
14      num2str(timeSecs - startSecs), ' seconds'];
15      disp(str);
16
```

Line 8-10. KbWait returns the time in seconds (with high precision) since the computer started. Usually we don't want to know the absolute time in seconds, but the time since some other event (e.g. since you put an image on the screen). So here we are calculating the time between the key press and an arbitrary start time.

```
%exampleKb2.m
% Displays the key number when the user presses a key.
% a simplified version of KbDemo by IF 4/2007

WaitSecs(0.5);
disp('Testing KbCheck and KbName: press a key to see its
number');
disp('Press the escape key to exit.');
escapeKey = KbName('ESCAPE');

while KbCheck
;
end % Wait until all keys are released.

while 1
    % Check the state of the keyboard.
    [ keyIsDown, seconds, keyCode ] = KbCheck;

% If the user is pressing a key,
% then display its code number and name.
    if keyIsDown

% Note that we use find(keyCode) because keyCode is an array.
        str=['You pressed key ', find(keyCode), …
' which is ', KbName(keyCode)];
        disp(str)

        if keyCode(escapeKey)
            break;
        end

% If the user holds down a key,
% KbCheck will report multiple events.
% To condense multiple 'keyDown' events into a single event,
% once a key has been pressed
% we wait until all keys have been released
% before going through the loop again
        while KbCheck; end
    end
```

```
end
```

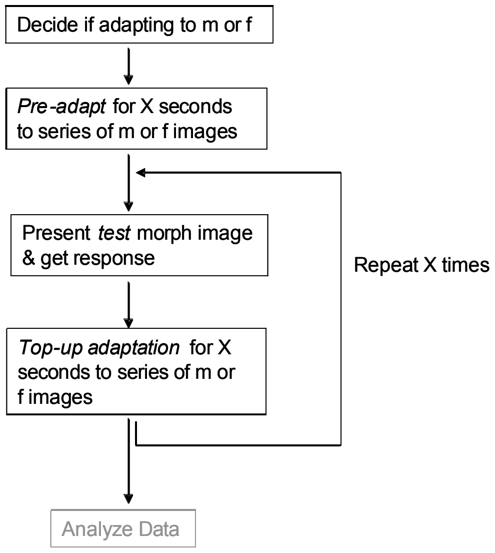
Part 3 – Making an Experiment

In this part we are actually going to begin writing real code for a real experiment. We are going to do a version of a cool illusion first demonstrated by Mike Webster (Webster, Kaping, Mizokami, and Duhamel, 2004, *Adaptation to natural facial categories.* Nature 428, 558-561.)

The illusion works like this – after staring at a series of male faces for a long time, a face that previously appeared gender neutral looks female. Similar aftereffects are very famous in other domains (look on the web for color adaptation effects and the waterfall illusion) but it's still pretty surprising to see such strong adaptation for human faces.

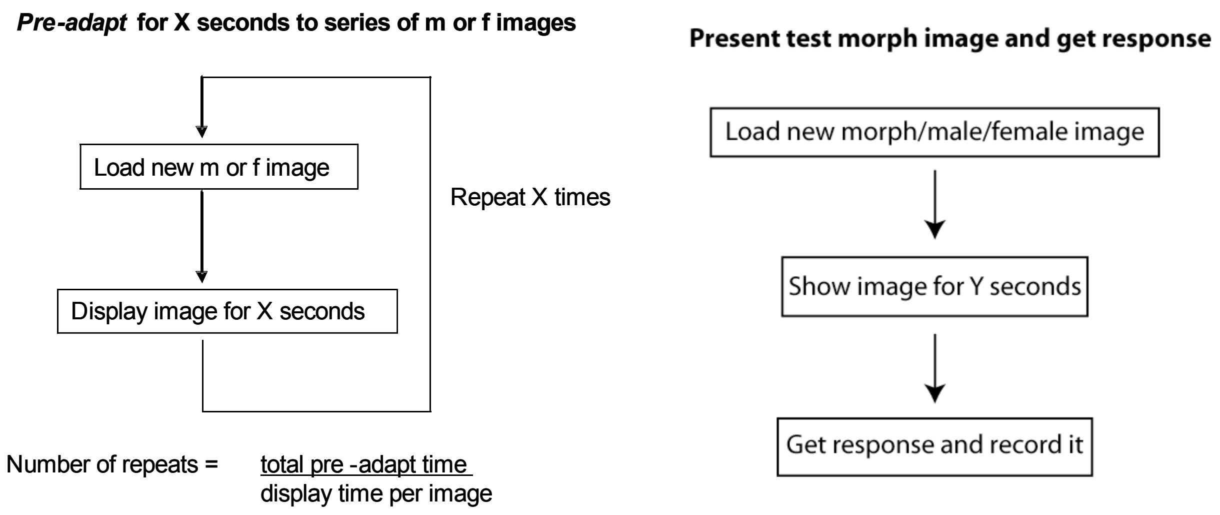
What we are going to do is write a program where subjects adapt for a minute to a set of male or female images, and then carry out a series of trials where they respond whether a morph image (morphed between male and female) is perceived as being male or female. Between each trial will be a 5 second adaptation top-up period. The goal is to measure how adaptation to male or female images influences your perception of the morphed images.

Let's think about the structure of the experiment. You can think of it as having 5 parts, as shown below.


Decide if adapting to m or f
Pre-adapt for X seconds to series of m or f images
Present test morph image & get response
Repeat X times
Top-up adaptation for X seconds to series of m or f images
Analyze Data

I've put the Analyze Data part in gray since hopefully you already all know how to do that. We will see in the end what the data obtained looks like and how it can be accessed for analysis.

Each sub-part of this program can also be diagrammed. For example, compare these two sub-parts of the above flow diagram:


Pre-adapt for X seconds to series of m or f images
Load new m or f image
Display image for X seconds
Repeat X times
Number of repeats = total pre -adapt time / display time per image
Present test morph image and get response
Load new morph/male/female image
Show image for Y seconds
Get response and record it

Notice that there are some similarities between both pieces of the experiment and some differences. For example, they both require loading and presentation of images. We can save a lot of work by making a separate function that executes each these tasks. Then, we will be able to reuse that function in both parts of the experiment. However, notice that in the different parts of the experiment different images are loaded and they may need to be presented for different durations. Therefore, we require a function that loads and presents images. Let's start from that. The function should be able to flexibly receive input about the duration of presentation and about which image is to be presented. Since loading an image requires only one line of code, let's perform both of these operations in the same function:

```
function showImage(imName, wPtr, dur)

%function [adaptcount, stim]=showImage(imName, wPtr, dur)
%
%Part of the FaceAdaptation experiment. This function presents a jpg image
%for a given duration
%<imName> The name of the image file.
%<wPtr> Pointer to a screen
%<dur> duration of presentation in seconds
%
% 8/2007 ASR made it

%Read image from file:
img=imread(imName, 'JPG');

%Present image:
textureIndex=Screen('MakeTexture', wPtr, double(img));
Screen('DrawTexture', wPtr, textureIndex);
Screen(wPtr, 'Flip');

%Wait for dur seconds before before breaking out of the function

WaitSecs(dur)

```

Line 14: `imread` is a built-in function of Matlab. Notice the flag for the file type. This function can read in image files from many different file formats.
Lines 17-19: These lines are the very core of our program. This is where the magic happens. First of all, notice that the output of line 14, `img`, is in machine format (uint8, to be precise) so it needs to be **casted**. That is, we need to convert it into something that Screen can then present. That's what `double(img)` does. It turns it into another type of variable called **double.** Now – as for the magic: `MakeTexture` takes a matrix of numbers and turns it into an OpenGL texture. `DrawTexture`, takes this texture and puts it onto the current screen. Nothing will actually appear on the screen until `Flip` is issued. This way the timing of the presentation can be controlled very precisely.
Line *23:* `WaitSecs` is a very precise way to wait. Yes, just wait. For as long as you need.

Before we put all this in context, let's deal with another little thing happening within the above mentioned processes. Getting the response from the subject. As we saw above, there are many ways of collecting responses. For now, we will use getChar:

```
function response=getResponse(params)

%function response=getResponse(stim, response)
%
%Part of the FaceAdapt experiment. This function gets a button press
%response from the subject and records it.
%<params> is a struct that contains a field params.beep, which defines the
%sound to be played before collecting the button press
%{response} is a character - the response button press
%
%8/2007 ASR made it.

FlushEvents;
sound(params.beep);
response=GetChar;

```

This looks so simple that you are probably asking yourself why bother have a function in order to do this. Now imagine that you want to make a program that will run a completely different experiment. Notice that there is nothing about this function that relates it to this experiment. So, if the function is properly written, you will never have to bother thinking about the mechanics of getting responses again! Moreover, you might want to try running your experiment in several different setups. For example, you might want to port over a behavioral experiment for the scanner, and collect responses from the MRI-compatible button box, without having to change all the little details each time you move to and from the magnet. Next meeting we will talk about how this can be done by making a slightly more sophisticated version of `getResponse`

Let's move on. What other functions do we need? Notice that there is another operation that occurs more than once in the flow of the program. This operation is adapting the subjects to faces. In the beginning of the experiment we have a long pre adaptation period and then, each trial, we readapt the subjects by exposing them to a period of top-up adaptation. Let's just make one function that takes care of adaptation and can be used for both of these:

```
function doFaceAdapt(params, wPtr, dur, time)

```

```
3   %function doFaceAdapt(params, wPtr, dur, time )
4   %
5   %Part of the face adaptation experiment
6   %Executes the adaptation phases of the experiment (both preadaptation and
7   %readaptation!).
8   %<params> the params struct
9   %<dur> duration of presentation of each adaptation image
10  %<time> total duration of adaptation
11  %
12  %ASR made it 8/2007
13
14  lastImage=0; %Makes sure that we don't show the same image twice in a row
15  for repeat=1:(time/dur)
16      whichImage=ceil(rand*params.numAdaptImages);
17      while whichImage==lastImage %ibid
18          whichImage=ceil(rand*params.numAdaptImages);
19      end
20      imName=[params.adaptstr,num2str(whichImage)];
21      showImage(imName, wPtr, dur);
22      lastImage=whichImage;
23  end
24
25
```

Line 15: We loop over the number of images we will show, `(time/dur)`. Notice that we don't use the counting variable of this loop for anything except to tell the loop when to stop.
Line 14,22 and Lines 16-19: These lines are here just to make sure that each adaptation image is different from the one before it.
Line 20: Choose the file that will be loaded.
Line 21: Here is our function `showImage` from before. This is not the last time we will see it.

Now we are ready to start piecing things together. Let's zoom out for a couple of minutes and look at the whole experiment from beginning to end:

```
%faceAdaptationMain.m
%
%This is the main script for the face adaptation experiment
%Images of faces are shown. First, a pre-adaptation period of either male
%or female faces. Then, judgments are made on images of either male, female
%or morph
%
%08/2007 ASR made it, adapted from code by IF

%Start by removing anything you had left over in the memory:
clear all; close all;

%Initialize the path:
cd('/Users/ariel/PTBTutorial/FaceExperiment/')
addpath(pwd);
addpath('morph');

%Get the params of the experiment:
params = getParams;

```

```
21  %Intialize response struct into which you will record the subject's
22  %responses:
23  response = {};
24
25  %Set DebugLevel to 3, so that you don't get all kinds of messages flashed
26  %at you each time you start the experiment:
27  Screen('Preference', 'VisualDebuglevel', 3)
28
29  %Open display window:
30  [wPtr rect] = openFaceAdapt;
31
32  %Pre-adaptation:
33  doFaceAdapt (params, wPtr, params.preAdaptImageDuration, params.preAdaptTime);
34
35  %Start trial loop
36  for trialCounter=1:params.nTrials
37
38      response{trialCounter} = doFaceAdaptTrial (wPtr, params);
39
40      if trialCounter<params.nTrials %Top Up the adaptation
41
42          doFaceAdapt(params, wPtr, params.reAdaptImageDuration,
43  params.reAdaptTime);
44      end
45
46  end %Ends trial loop
47
48  % close display window
49  Screen('CloseAll');
50  ShowCursor
```

Line 19: `getParams` does exactly that. It gets parameters for running the experiment. It is not a bad idea to have all you parameters in one easy to read file. We will look at this file a bit later. For now all you need to know is that the output of this function, `params` is a **struct**. This is a data type that can contain other variables. This is very useful if you have a lot of variables that you want to pass on to a function and don't want your function inputs to be impossibly long.
Let's look at an example of how that works.

```
>>exampleStruct.number=4

exampleStruct =

    number: 4
```

the period is used in order to reference different **fields** of the struct. In this case, it is used in order to create the field `number`.

```
>> exampleStruct.matrix=[0 5 5 ; 5 5 4 ; 4 3 2]

exampleStruct =

    number: 4
    matrix: [3x3 double]
```

Now our struct already contains more than one field. You can add more and more fields of different types of variables. This is almost like object orientation (for those of you into that), but not quite as powerful.
Line 23: this initializes a variable that will eventually hold the results of the experiment. This variable is a **cell array**, yet another useful Matlab data type. Cell arrays are useful because they are like matrices that can hold together variables of different kinds. For example:

```
 >> exampleCellArray= {'Ariel', 2 , [1 0 0; 0 1 0; 0 0 1], 'what?'}

exampleCellArray = 

    'Ariel'    [2]    [3x3 double]    'what?'
```

Cell array are indexed just like vectors or matrices, but with curly braces:

```
>> exampleCellArray{3}

ans =

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

Line 27: depending on this parameter, Psychtoolbox may present some parameters of the display and the setup as it initializes the screen. This turns all those bells and whistles off.
Line 30: This opens the display and gives you a pointer to the screen. The pointer is simply a number, but you will need this number each time you want to control things that are going on on the screen. So take good care of it. We will see what goes on within `openFaceAdapt`.
Line 33: This calls `doFaceAdapt`, which we saw before, with the timing parameters of the preadaptation.
Lines 36-45: We loop over this portion of code for as many times as there are trials, as defined in the `params` struct.
Line 38: `doFaceAdaptTrial` executes the nitty gritty of running the trial. We will look at it next. For now, just notice an important detail – this function has as one of its inputs `response` and its output is into `response`
Line 40-43 : If this is not the last trial we need to readapt the subject. We call `doFaceAdapt` again, this time with the top-up adaptation timing parameters. Reuse, recycle, relax.
Line 48-49 : Some housekeeping. When the experiment is over, close all the screens you opened and give me back my cursor.

That was the big picture. Now let's zoom in and look at the components we haven't looked at yet. We'll start by examining `getParams`:

```
function params=getParams

%function params = getParams
%Part of the face adaptation experiment
%This function makes the struct that holds the parameters for the
%presentation of the stimuli and such.
%
%8/2007 ASR made it.

%Experimenter defined params:

```

```
12  params.preAdaptTime=10; %Pre-adapt time (in seconds)
13  params.preAdaptImageDuration=2; %Presentation time (secs) for each adaptation
14  image
15  params.numAdaptImages=10; %There are 10 images in each adaptation set of images
16  params.numTestImages=49; %There are 49 morph images
17  params.testImageDuration=0.5; %Presentation of the test image (in secs)
18  params.reAdaptTime=6; %Top up adaptation
19  params.reAdaptImageDuration=2; %Presentation time for the readaptation images
20
21  params.beep=sin(1:0.5:100);
22  params.ISI=0.5; %ISI, in seconds
23
24  %User defined params:
25
26  params.adaptType = questdlg('Do you want to adapt to males (m) or females
27  (f)?',...
28      'Adaptation Type','m','f', 'f'); %Adapt to male or female images:
29  if params.adaptType == 'f'
30      addpath('female');
31      params.adaptstr='female';
32  elseif params.adaptType == 'm'
33      addpath('male');
34      params.adaptstr='male';
35  else
36      disp('Error 1. Sorry, response about adapt type is not correct');
37  end
38
39  nTrials = inputdlg('How many trials do you want to run?');
40  params.nTrials=str2num(nTrials{1});
```

In general, the function has two parts. One part initializes experimenter-defined parameters. These can be changed by the experimenter by editing this file. The second part are parameters that need to be changed each time you run the experiment. For example, in this case, we want to be able to adapt to either male or female images and we want to be able to change the number of trials in the experiment.

Line 26 : It is easy to make dialogue boxes in Matlab. `Questdlg` is a function that opens a dialogue box with up to three buttons. In this case we only have two.

Lines 39-40 : `inputdlg` is a function that opens a dialogue box which takes a string as its input. The output `nTrials` is a cell array containing a string. So it needs to be converted before it can be used as a number.

Next, let’s examine `openFaceAdapt`:

```
1   function [wPtr rect] = openFaceAdapt
2
3   %function [wPtr rect] = openFaceAdapt
4   %
5   %Part of the face adaptation experiment.
6   %
7   %Made by IF (when?). Commented by ASR 8/2007.
8
9   screenNum=0;
10  [wPtr,rect]=Screen('OpenWindow',screenNum);
11  HideCursor;
12  white=WhiteIndex(wPtr);
```

```
13  Screen('FillRect',wPtr,white);
```

Line 9: In a multi-screen setup, it is important to choose the right screen. 0 denotes the screen on which the Matlab command window appears (where the start menu is in Windows or the dock in Macs).
Line 10: This is the command that gets the screen. The wPtr from here is what we use in order to reference the screen in all the other functions. `Rect` holds the resolution (or size) of the screen

Finally, let's take a look at `doFaceAdaptTrial`:

```
1   function thisResponse = doFaceAdaptTrial (wPtr, params)
2
3   %function thisResponse = doFaceAdaptTrial (wPtr, params)
4   %
5   %Part of the face adaptation experiment. Executes the trials and outputs the
6   %response associated with this trial.
7   %<wPtr> is a pointer to the screen
8   %<params> is the params struct
9   %
10  %ASR made it 8/2007
11
12  chooseFrom = {'morph' , 'female' , 'male'};
13  chooser=ceil(rand*length(chooseFrom));
14  thisResponse.gender = chooseFrom(chooser);
15
16  if chooser==1
17      whichImage=ceil(rand*params.numTestImages);
18  else
19      whichImage=ceil(rand*params.numAdaptImages);
20  end
21
22  imName=strcat(thisResponse.gender,num2str(whichImage));
23
24  SmallBlank (wPtr, params);
25  showImage(imName{1}, wPtr, params.testImageDuration);
26
27  Screen('DrawText', wPtr, 'Press m for male, f for female ... ', ...
28      200, 20, [0 0 0]);
29  Screen(wPtr, 'Flip');
30
31  thisResponse.char=getResponse(params);
32
33  SmallBlank (wPtr, params);
```

Line 12-22: Each trial, randomly choose whether you want to test the subject with a male, a female or a morph. Then, randomly choose which of the images from the chosen class to present in this trial.

Line 24: `SmallBlank` clears the screen and waits for a time defined in `params.ISI`
This is what it looks like:

```
function SmallBlank(wPtr, params)
%function SmallBlank(wPtr, params)
```

```
%Pauses the program with a white screen for a pause
%<wPtr> pointer to the screen
%<params> params struct
%
% written if 5/2007
% modified ASR 8/2007 - put the ISI duration in the params
initialization
% script. Changed some more stuff.

white=WhiteIndex(wPtr);
Screen('FillRect',wPtr,white);
Screen(wPtr, 'Flip');

tic
while toc<params.ISI
    ;
end
```

Notice that we use `tic` and `toc` in order to wait here (instead of, for example, `WaitSecs)`
We use `SmallBlank` again in line 33

Line 25: Here we are, full circle again, by using `showImage` again. This time in order to show the test image.

Line 27-31: we use another psychtoolbox function, `drawText` in order to prompt the subject. Finally we assign the value of the output, `thisResponse`, by calling `getResponse`

When all is said and done we are left with the following variable in our workspace:

```
>> response

response =

    [1x1 struct]    [1x1 struct]
```

If we want to analyze the results, we can reference the different parts of the variable:

```
>> response{1}

ans =

    gender: {'morph'}
      char: 'f'

>> response{1}.gender

ans =

    'morph'

>> response{1}.char

ans =

f
```

Part 4 – Advanced Topics

This part will include several topics that have bearing on making successful experiments. We will start by building another experiment. This time we will a psychophysical staircase in order to measure thresholds. First, we will use a standard staircase. Psychophysical staircase methods are very useful, because they give you a relatively efficient method to estimate the threshold of your subjects in a certain task. You will not waste a lot of trials exploring parts of parameter space far away from the threshold, because the threshold will converge on the domain of interest, around the subject's threshold. Notice that you might not get a full psychometric curve this way. For example, you might not get any information about the level at which saturation occurs or the curvature of the psychometric curve around the point at which performance starts to rise. You might not even get any information about the slope of your psychometric curve. If all these things are important in your analysis of the behavior, you might want to use the method of constant stimuli in order to estimate your threshold. Alternatively, you might want to run several different psychophysical staircases, converging on different levels of performance.

The experiment we will run as an example is a target detection experiment. In this case, the targets will be red circles. In each trial, the subject will view the screen and will need to determine whether a red circle appeared among the green circles and red squares surrounding it. The time of presentation then changes from trial to trial according to the subject's performance. Let's start by charting the flow-chart of this experiment:

Determine whether to show targets on the left or on the right in this block of trials

Determine subjects name

{repeat the following as long as there aren't enough reversals }

Determine whether to show a target this trial

Show stimulus

Get response

Record data about the trial as you go (you'll see what that means soon)

If target was shown, determine the duration of the presentation in the next trial according to the subject response

{end of trial}

Analyze the data and determine the threshold

We will see as we go along that here too, we will want to make functions that will take care of different parts of the experiment. Let's start by looking at the main script:

```
%detectMain.m
%
%Runs the detection experiment. Target is a red circle among green circles
%and red squares. The staircase parameter is the duration of presentation.
%
%8/26/2007 ASR wrote it.

%Start by removing anything you had left over in the memory:
clear all; close all;
%Set DebugLevel to 3:
Screen('Preference', 'VisualDebuglevel', 3);

%Get the params of the experiment (this also opens the display):
params = getDetectParams;

%Initialize the struct where data will be recorded:
history = makeHistory(params);

%Start some local variables used to control the staircase:
nTrials=0;
reversals=0;

%Trial loop. This time the number of trials is determined by the occurence
%of reversals:
while reversals<params.nReversals
    nTrials=nTrials+1;
    history=doTrialDetect(params, history, nTrials);
    reversals=sum(history.isReversal);
end

%Save data (using save):

now=fix(clock);
%File name includes the subject id and the time of the end of the
%experiment:
fileName=[params.subjectID,'_',num2str(now(2)),num2str(now(3)),num2str(now(1)),'_
save(fileName,'history', 'params');

% close display window
Screen('CloseAll');
ShowCursor;

```

This should look more or less familiar to you. The logic is the same as in the face detection experiment. Initialization of the parameters occurs in a function of its own. Then, there is a main trial loop which calls a function which executes the details of the trial itself. Notice that the data is stored in a struct called "history" which is then saved (in line 37). This saves the variables "history" and "params" into a .mat file. These files can then be read (only!) by matlab. This simply creates in the workspace the variables that existed in the workspace when "save" was invoked.

Next Let's look at the function that gets the parameters:

```
1   function params=getDetectParams
2
3   %function params=getDetectParams
4   %
5   %Creates parameters and intitializes the display for the detection experiment
6   %
7   %8/27/2007 ASR made it
8
9   %Experimenter defined params:
10
11  params.stimulusSize=50; %pixels
12  params.startDuration=1;
13  params.nBeforeReversal=3; %3-> ~80% success 2->~70% success 1->~50% ssuccess
14  params.stairCaseDecrements = 0.1;
15  params.screenNum = 0;
16  params.fixationSize = 20; % in pixels
17  params.ITI = 1; %seconds
18  params.ISI=0.03; %seconds
19
20  params.beep=sin(1:0.5:100);
21
22
23  %User defined params:
24
25  params.testSide = questdlg('Do you want to test on right(R) or Left(L)?',...
26      'Test Side','R','L', 'R');
27  if params.testSide == 'R'
28      params.isLeft=0;
29  elseif params.testSide == 'L'
30      params.isLeft=1;
31  else
32      disp('Error 1. Sorry, response about test side is unrecognized');
33  end
34
35  nReversals = inputdlg('How many reversals do you want before stopping?');
36
37  params.nReversals=str2num(nReversals{1});
38
39  subjectID= inputdlg('What is the subject ID?');
40
41  params.subjectID=subjectID{1};
42
43  %Initialize display params and open the display:
44
45  [params.black params.wPtr params.rect] = getDisplay (params.screenNum);
```

This function is again divided into experimenter params, user params and here also screen parameters, initialized by the function getDisplay:

```matlab
function [black wPtr rect] = getDisplay (screenNum)

%function [wPtr rect] = getDisplay
%
%opens the display defined by the input <screenNum>, returns the size of
%the screen (in pixels!) and the pointer to that screen.

[wPtr,rect]=Screen('OpenWindow',screenNum);
HideCursor;
black=BlackIndex(wPtr);
Screen('FillRect',wPtr,black);

```

This is a rather generic function. In fact, it was very slightly adapted from the equivalent function in the face adaptation experiment.

Next Let’s look at what happens within a trial:

```matlab
function history=doTrialDetect(params,history,nTrials)

%function history=doTrialDetect(params,history,nTrials)
%
%Executes the trial of the detection experiment. Receives as input,
%<params>, a struct with various parameters,
%<history>, a struct with the history of the experiment so far, this same
%struct is then updated as the function proceeds and is the output of the
%function. <nTrials> is simply a counter that tells us what trial we are
%at.

%Put up the fixation:
rectWidth=params.rect(3);
rectHeight=params.rect(4);

fixationRect=[rectWidth/2-params.fixationSize/2 rectHeight/2-
params.fixationSize/2 ...
    rectWidth/2+params.fixationSize/2 rectHeight/2+params.fixationSize/2];

Screen('FillRect',params.wPtr, [255 255 255], fixationRect);
Screen('Flip', params.wPtr);

WaitSecs (params.ITI);

```

```
25  %Determine whether to show a target in this trial and record as you go:
26  
27  isTarget=round(rand);
28  history.isTarget=[history.isTarget isTarget];
29  
30  makeStim(params,isTarget)
31  Screen('FillRect',params.wPtr, [255 255 255], fixationRect);
32  sound(params.beep);
33  showTime=Screen('Flip', params.wPtr);
34  
35  while GetSecs<showTime+history.dur(nTrials)
36      ;
37  end
38  
39  Screen('FillRect',params.wPtr, [255 255 255], fixationRect);
40  Screen('Flip', params.wPtr);
41  sound(params.beep)
42  
43  thisCorrect=getResponse(isTarget);
44  history.correct=[history.correct thisCorrect];
45  
46  if thisCorrect
47      if history.nUp(nTrials) >= params.nBeforeReversal
48          history.isReversal = [history.isReversal 1];
49          history.nUp = [history.nUp 0];
50          nextDur=history.dur(nTrials)-params.stairCaseDecrements;
51  
52      else
53  
54          history.isReversal = [history.isReversal 0];
55          nextDur = history.dur(nTrials);
56          history.nUp=[history.nUp history.nUp(nTrials)+1];
57  
58      end
59  
60  
61  
62  else
63  
64      history.nUp = [history.nUp 0];
65      nextDur=history.dur(nTrials)+params.stairCaseDecrements;
66      history.isReversal = [history.isReversal 0];
67  
68  end
69  
70  history.dur=[history.dur nextDur];
71  
```

Notice one neat thing about the function definition. This function has as its output one of its inputs. This way we can update the struct that will eventually hold all the data about the experiment, as we go. Lines 46-68 are the lines that determine the progression of the staircase. Note that in this case the staircase is updated whether there is a target present or not. When designing staircase experiments, you may want to have the staircase updated only when a target is present. Lines 50 and 66 are the lines in which the staircase parameter is changed. Sometimes you may want to have several sizes of decrements. So that in the beginning of the staircase you converge more rapidly and towards the end, you converge in smaller steps, sampling the area of the threshold more thoroughly.

```matlab
function makeStim(params,isTarget)

%function makeStim(params,isTarget)
%
%This function makes the stimulus for the detection experiment. <params> is
%the struct with the params. <isTarget> is a variable whos value will be 1
%if there is a target in this trial.
%The stimulus is hard-coded rather unwisely into the size of the display
%used.
%Screen('FillOval') and Screen('FillRect') are used in order to make the
%stimuli themselves.


heightUnit=64;
lengthUnit=64;
stim_loc=zeros(10,4);

%Stimuli on the left side of the screen:
stim_loc(1,:)=[1*lengthUnit 5.5*heightUnit 2*lengthUnit 6.5*heightUnit];
stim_loc(2,:)=[3*lengthUnit 2.5*heightUnit 4*lengthUnit 3.5*heightUnit];
stim_loc(3,:)=[3*lengthUnit 7.5*heightUnit 4*lengthUnit 8.5*heightUnit];
stim_loc(4,:)=[6*lengthUnit 1.5*heightUnit 7*lengthUnit 2.5*heightUnit];
stim_loc(5,:)=[6*lengthUnit 9.5*heightUnit 7*lengthUnit 10.5*heightUnit];
%Stimuli on the right side of the screen:
stim_loc(6,:)=[9*lengthUnit 9.5*heightUnit 10*lengthUnit 10.5*heightUnit];
stim_loc(7,:)=[9*lengthUnit 1.5*heightUnit 10*lengthUnit 2.5*heightUnit];
stim_loc(8,:)=[12*lengthUnit 7.5*heightUnit 13*lengthUnit 8.5*heightUnit];
stim_loc(9,:)=[12*lengthUnit 2.5*heightUnit 13*lengthUnit 3.5*heightUnit];
stim_loc(10,:)=[14*lengthUnit 5.5*heightUnit 15*lengthUnit 6.5*heightUnit];

for i=1:10
    isCircle=rand;
    if isCircle>0.5
        Screen('FillOval', params.wPtr , [0 255 0], stim_loc(i,:));
    else
        Screen('FillRect', params.wPtr , [255 0 0], stim_loc(i,:));
    end
end

if params.isLeft && isTarget
    whichLeft=ceil(rand*5);

    Screen('FillOval', params.wPtr , [255 0 0], stim_loc(whichLeft,:));

elseif ~params.isLeft && isTarget

    whichRight=ceil(rand*5)+5;

    Screen('FillOval', params.wPtr , [255 0 0], stim_loc(whichRight,:))
end;

```

Notice that the stimulus is coded in terms of a certain display. To fit it to your own display, try playing with the heightUnit and lengthUnit variables. If we are lucky, you may be able to fit the stimuli nicely enough into your display by only changing these variables. Otherwise, try to change the stimulus array, so that it fits symmetrically around the fixation point. Bonus points to the participant who finds a way to make this code portable across different displays. Note that this only makes the stimulus. Screen ('Flip') is not called from this script but rather from the doTrial script that calls this function.

```
function history=makeHistory (params)

history.dur=[params.startDuration];
history.isTarget=[];
history.nUp=[0];
history.isReversal=[];
history.correct=[];
```

This simpy initializes the history struct. We need do that because doTrial assumes that these variables already exist in the struct. The staircase parameter is initialized to its intial value, set by the experimenter. Additionally, the nUp variable, which counts how many correct trials have occurred, in order to update the staircase, is initialized to 0, so that it can be evaluated in the doTrial function. They'll end up having one component more than the other variables in "history".

Using Quest

The quest algorithm is a very efficient way to conduct experiments using psychophysical staircases. At each point in the experiment, it calculates the maximum likelihood estimate of the threshold, given the data collected so far in the experiment and the prior we had on the threshold going into the experiment. Then, it proposes the intensity of the staircase parameter, for which a trial would result in the maximal information on the value of the threshold. Details exist in a paper by Watson and Pelli from 1983, to be found on the website.
One major disadvantage of the Quest algorithm is that it requires input on a log scale. This means that you will have to perform some transformations on your staircase parameter and eventually also on your data. Nicely enough, you don't have to change everything in your scripts in order to change your experiment to use the quest algorithm. Let's look at what the main script looks like now:

```
%detectMainQ.m
%
%Minimal changes introduced to the detection experiment, using the quest algorith
%in order to determine the threshold.
%
%8/27/2007 ASR wrote it.

```

```
8   %Start by removing anything you had left over in the memory:
9   clear all; close all;
10  %Set DebugLevel to 3:
11  Screen('Preference', 'VisualDebuglevel', 3);
12
13  %Get the params of the experiment (this also opens the display):
14  params = getDetectParamsQ;
15
16  %Initialize the struct where data will be recorded:
17  history = makeHistoryQ(params);
18
19  %Trial loop. This time the number of trials is determined in advance
20
21  for nTrials=1:params.totalTrials
22
23      history=doTrialDetectQ(params, history, nTrials);
24
25  end
26
27  %Save data (using save):
28
29  now=fix(clock);
30  %File name includes the subject id and the time of the end of the
31  %experiment:
32  fileName=['withQuest',
33  params.subjectID,'_',num2str(now(2)),num2str(now(3)),num2str(now(1)),'_',num2str(
34  save(fileName,'history', 'params');
35
36  % close display window
37  Screen('CloseAll');
38  ShowCursor;
```

One obvious change is in that lines 21-25 now have a for loop instead of the while loop that existed there in the earlier version. Quest gives us not only an estimate of the value of the threshold, but also the error on the estimate. So - Quest can be set up to run until it reaches an estimate of the threshold with a specific size of the confidence interval around it. Then, a while loop should be used.
Other than that, pretty much everything is the same.

So far, it doesn't look like we changed a lot. Let's look at the doTrial function. There are some more substantial changes there:

```
function history=doTrialDetectQ(params,history,nTrials)

%function history=doTrialDetect(params,history,nTrials)
%
%Executes the trial of the detection experiment (with Quest). Receives as
input,
%<params>, a struct with various parameters,
%<history>, a struct with the history of the experiment so far, this same
%struct is then updated as the function proceeds and is the output of the
%function. <nTrials> is simply a counter that tells us what trial we are
%at.
```

```
12
13  %Put up the fixation:
14  rectWidth=params.rect(3);
15  rectHeight=params.rect(4);
16
17  fixationRect=[rectWidth/2-params.fixationSize/2 rectHeight/2-
18  params.fixationSize/2 ...
19      rectWidth/2+params.fixationSize/2 rectHeight/2+params.fixationSize/2];
20
21  Screen('FillRect',params.wPtr, [255 255 255], fixationRect);
22  Screen('Flip', params.wPtr);
23
24  WaitSecs (params.ITI);
25
26  %Determine whether to show a target in this trial and record as you go:
27
28  isTarget=round(rand);
29  history.isTarget=[history.isTarget isTarget];
30
31  trialTheta=QuestQuantile(history.q);
32  dur=params.maxDur*10^trialTheta;
33  tDur=min(dur,params.maxDur);
34
35  makeStimQ(params,isTarget)
36  Screen('FillRect',params.wPtr, [255 255 255], fixationRect);
37  sound(params.beep);
38  showTime=Screen('Flip', params.wPtr);
39
40  while GetSecs<showTime+tDur
41      ;
42  end
43
44  Screen('FillRect',params.wPtr, [255 255 255], fixationRect);
45  Screen('Flip', params.wPtr);
46  sound(params.beep)
47
48
49  thisCorrect=getResponseQ(isTarget);
50  history.correct=[history.correct thisCorrect];
51  history.dur=[history.dur tDur];
52  history.q=QuestUpdate(history.q,log10(tDur/params.maxDur),thisCorrect);
```

Differences start to become apparent in lines 31-33. QuestQuantile is a function that receives as an input a quest struct (we'll see what that is soon) and gives as an output an estimate of the staircase parameter given the data stored in the quest struct. In this case, the staircase parameter is not the duration, but log10 (duration/maximal duration). This is the transformation I was warning you about before. In cases like duration (which can, in principal, stretch on forever) you don't need to set a maximal value, but when you are using other physical parameters of your stimulus as the staircase parameter, such as contrast or difference in angle, there is a natural maximum which you should be careful not to pass. This little procedure makes sure that you never give the stimulus presentation functions a value of the stimulus which cannot be produced. In order to get back the duration for this trial, we apply the opposite transformation (in line 32), then we make sure that the duration in this trial does not surpass a predefined maximal duration (line 33). Everything between line 28 and line 51 is the same as it was in the previous experiment, except that we have totally removed everything that has anything to do with the staircase. However, nothing has changed in the way in which the stimulus is presented and the responses collected. Then, line 52 updates the quest struct with the result of this trial.

Let’s look at the changes that are introduce when using this algorithm to the parameter and history initialization:

```matlab
function params=getDetectParamsQ

%function params=getDetectParams
%
%Creates parameters and intitializes the display for the detection experiment
%
%8/27/2007 ASR made it

%Experimenter defined params:

params.stimulusSize=50; %pixels
params.startDuration=1;

%for the quest algorithm:
params.startVariance=0.2;
params.maxDur = 2;
params.pThreshold=0.82; %0.82 is equivalent to a 3 up 1 down standard staircase
%

params.screenNum = 0;
params.fixationSize = 20; % in pixels
params.ITI = 1; %seconds
params.ISI=0.03; %seconds

params.totalTrials=40; %Rather standard for using quest (!)

params.beep=sin(1:0.5:100);


%User defined params:

params.testSide = questdlg('Do you want to test on right(R) or Left(L)?',...
    'Test Side','R','L', 'R');
if params.testSide == 'R'
    params.isLeft=0;
elseif params.testSide == 'L'
    params.isLeft=1;
else
    disp('Error 1. Sorry, response about test side is unrecognized');
end

subjectID= inputdlg('What is the subject ID?');

params.subjectID=subjectID{1};

%Initialize display params and open the display:

[params.black params.wPtr params.rect] = getDisplayQ (params.screenNum);
```

Major changes are in lines 14-17. We have introduced some parameters that are required for running Quest and we have omitted some variables that are no longer needed, pertaining to the staircase.

Finally, here is the intialization of the history struct. It includes the initialization of the q struct which is stored as part of the history struct:

```
1   function params=getDetectParamsQ
2
3   function history=makeHistoryQ(params)
4
5   %function history=makeHistoryQ(params)
6   %
7   %This function actually changed a lot. Notice that we are initializing the
8   %quest struct in this script. The quest struct will then be part of the the
9   %history struct - serve both as input as output to the doTrial function.
10  %
11  %8/27/2007 ASR made it.
12
13  history.dur=[params.startDuration];
14  history.isTarget=[];
15  history.correct=[];
16
17  %Initialize the quest struct and tack it on:
18
19  tGuess=log10(params.startDuration/params.maxDur);
20  tGuessSd=params.startVariance;
21
22  beta=3.5;delta=0.01;gamma=0.5;
23  history.q=QuestCreate(tGuess,tGuessSd,params.pThreshold,beta,delta,gamma);
24  history.q.normalizePdf=1;
25
26
```

We no longer need to keep track of the reversals or of how many trials ‘up’ we had so far. Instead, line 23 calls QuestCreate, which makes a quest struct. The input is our guess of the initial value of the staircase parameter (computed in line 19, from our initial guess of the duration), the standard deviation of this guess (in essence, how much weight we want to put on our prior beliefs about the parameter we are measuring). beta delta and gamma are parameters of the psychophysical function. These values are reasonable in a 2afc. Setting history.q.normalizePdf to 1 is important, but I don’t really know why.

Notice that the quest struct is saved as part of the history struct. So, we can analyze the data contained in that struct (the intensity of the staircase parameter at each trial, for example) from that.

File I/O

So far, we have seen use of the ‘save’ function in order to save variables into a .mat file. However, Matlab is also capable of more flexible forms of file I/O than that.
Let’s start by looking at a simple example:

```
1    %Simple example of File IO with fprintf and fscanf
2
3    fid = fopen ('data.txt', 'w');
4
5    A=magic(5);
6
7    fprintf(fid,'%2.2f \t %2.2f \t %2.2f \t %2.2f \t %2.2f \n', A)
8
9    clear all
10
11   fid = fopen ('data.txt', 'r');
12
13   B=fscanf(fid,'%g %g %g %g %g \n', [5 5])
```

Line 3 opens the file and creates a pointer to the file, ‘fid’. Line 5 creates a variable – A is a magic matrix (a matrix in which the sum of the rows, columns and diagonal are all equal). Line 7 prints the matrix A into the file defined by ‘fid’, according to the formatting string in the middle. This formatting string simply says: print the contents of the matrix A as tab separated floating point numbers with 2 digits on either side of the decimal point, break line every 5 components of the matrix. In lines 11-13, we simply retrieve back the matrix from the file.

```
1    %Adding a header:
2
3    fid = fopen ('data.txt', 'w');
4
5    A=magic(5);
6    fprintf(fid,'%s \n','This is the header');
7    fprintf(fid,'%2.2f \t %2.2f \t %2.2f \t %2.2f \t %2.2f \n', A)
8
9    clear all
10
11   fid = fopen ('data.txt', 'r');
12   header=fscanf(fid ,'%s \n', 4)
13   B=fscanf(fid,'%g %g %g %g %g \n', [5 5])
```

This example includes putting a header into the file and reading the file in, header separately from data.

Monitor Characterization: The final stages in setting up a visual stimulus

Here are the main things to think about before assuming that the visual stimulus you programmed is going to appear on the screen as you wish it to do.

These issues include calibration (measuring the relationship between gun values and the light emitted by the display) and various other issues.

The following table is a summary of the issues we are going to address in this chapter.

	time	**Space**	**luminance**	**color**
time	Missing frames Phosphor persistence	Protect your screen resolution from lab assistants Record your display settings and your viewing distance	Warming up Slow temporal changes Quantization errors	Warming up Slow temporal changes
space		Pixels not square	Stationarity Power drain	Stationarity Power drain
luminance			Variation between monitors Gamma correction	Chromatic variance for low/high gun values
color				Variation between monitors Gun drain

Always calibrate with the conditions as much as possible like the experiment as you can manage, same computer, same monitor, same screen resolution etc. etc.

Timing issues

1. Missing frames.

If you are putting up a series of images in quick succession (for example 1 flip for every monitor refresh) it's important to make sure that drawing the offscreen images isn't taking too long. If the offscreen drawing takes too long then you may be missing ***refreshes*** or ***frames***. This means that the program will be taking longer (and the stimulus will be moving more slowly, or present for longer) than you specified in the code. The way to check this is to record the times that the flips happen and make sure they match the frame rate of the monitor (as specified in the monitor control panel). If you are missing refreshes there are two main solutions to the problem (1) Try lowering the refresh rate of the monitor in the control panels to give you a little more time for each flip. (2) Try to do as much computational work as possible before you have to start quickly throwing up flips (e.g. if you are putting up a series of face images, load them all into memory beforehand rather than loading them from disk which is slow).

2. Phosphor persistence.

This is the phenomenon whereby even if you ask Psychtoolbox to only flash a gun for a single frame, that gun will still admit some energy after the frame is over. This is a particular issue for LCD and videoprojectors since these devices use relatively slow phosphors in comparison to good old fashioned CRT monitors. The issue of phosphor persistence is often an important issue for experiments where you are flashing up a stimulus briefly – imagine you have a 60Hz monitor and you are measuring how long a stimulus needs to be presented to be visible. When the stimulus is theoretically only up for a single frame (16.67ms) it is in reality presented for longer then that if it isn't immediately masked by some other stimulus. Of course this effect is most significant when a stimulus is only up for a very small number of frames. Another case where this is important is when you want to create a moving or flickering stimulus. The slow decline of the phosphors can result in a moving dot seeming to "streak" across the screen. For moving dot stimuli it is much better to use a CRT monitor if you can find one.

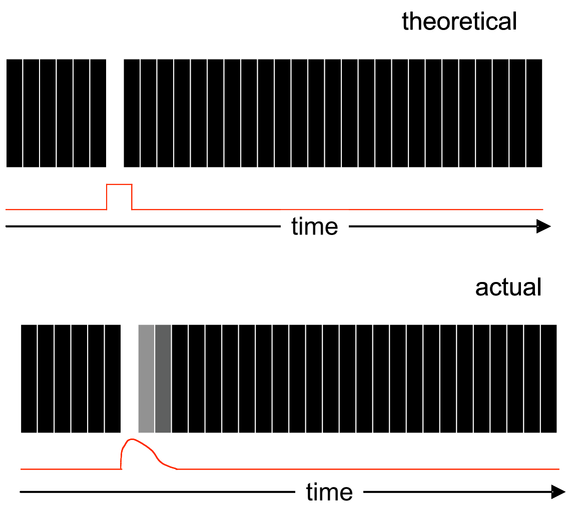
Space

1. Pixels are not square (isotropic)

The pixels in monitors are not perfectly square. So if you want a perfectly square stimulus you need to calculate the length and width of each pixel. The best way to do this is simply to measure the extent of the screen's width and height (remember there's often a black rim around the edge of the display which shouldn't be counted) and divide that by the number of pixels. You can then try to adjust the monitor to make the pixels more square, or modify the code for your stimulus to compensate for the pixels not being perfectly square.

2. Calculating visual angle

Generally visual stimuli are expressed in terms of visual angle – I.e. the size of the visual stimulus as subtended on the retina. This is because for most visual tasks it is the size of the image on the retina that is important rather than the size of the image on the screen.



To calculate the visual angle subtended by a stimulus involves two stages. (Make sure you use the same units, e.g. cm for both stages).

1) Calculate the width of a single pixel in cm.

(i) Measure the width and height of the screen (there is usually a black rim, don't measure that) in cm

(ii) Find out the number of pixels for that monitor horizontally and vertically

width of single pix in cm=width screen in cm/number of horizontal pixels (call this `pixH`)
height single pixel in cm=height screen in cm/number of vertical pixels (call this `pixV`)

It's possible that your pixels aren't square. It may be that you don't care. If you do, then you might wantto adjust the settings on your monitor to make the pixels squarer. If your pixels still aren't square then you may have to have different values for the number of pixels per degree in the horizontal and vertical directions. If the pixels are reasonably square you can average vertical and horizontal values together to give you a single value (pix)

(iii) Measure the viewing distance of your subject (from the middle of their two eyes) to the center of the monitor in cm. (call this `viewD`)

2) Calculate the visual angle subtended by a single pixel

```
degperpix=atan(pix/viewD)
```

It's often also useful to calculate the number of pixels within 1 degree of visual angle.

```
pixperdeg= 1/degperpix
```

Here's a program you can use to automatically calculate pixperdeg and degperpix

```
>params.res=[1024 768]
>params.sz=[30 24];
>params.vdist=57;
>[pixperdeg, degperpix]=VisAng(params)
```

Note that there are two values for pixperdeg and degperpix – the horizontal and the vertical values. If they are reasonably similar and your experiment doesn't depend critically on the stimulus being perfectly scaled along horizontal and vertical dimensions then you can simply average the two values of pixperdeg and the two values of degperpix to get an approximation of the visual angle subtended by each pixel

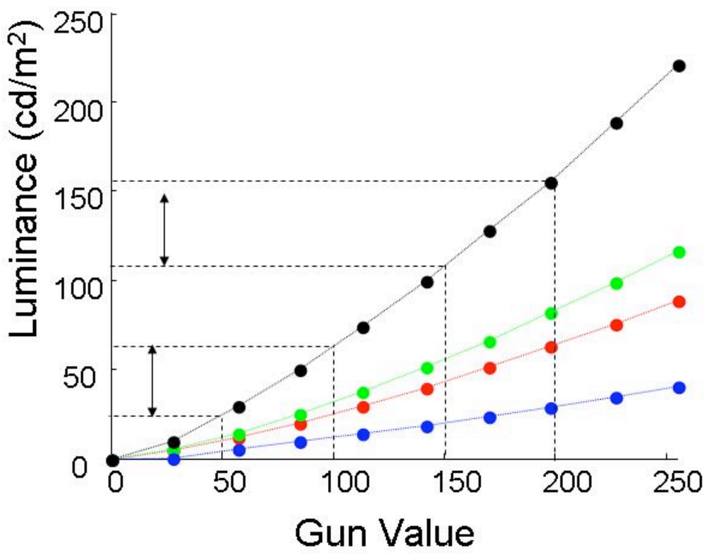
```
function [pixperdeg, degperpix]=VisAng(params)
% function [pixperdeg, degperpix]=VisAng(params)
%
%       Takes as input a structure containing:
% struct.res - the resolution of the monitor e.g. struct.res=[1024 768]
% struct.sz - the size of the monitor width then height
%      (needs to match the order you listed struct.res)
%      e.g. struct.sz=[30 24]
% struct.vdist - the viewing distance
%      e.g. struct.vdist=57;
%      Note that struct.vdist and struct.sz should be in the same
%      units (cm or inches)
%
%   Calculates the visual angle subtended by a single pixel
% Returns the pixels per degree
% and its reciprocal - the degrees per pixel (in degrees, not radians)
%
% written by IF 7/2000



degperpix=2*((atan(params.sz./(2*params.vdist))).*(180/pi))./params.res;
pixperdeg=1./degperpix;

```

Luminance

1. Different monitors vary a lot in their relationship between gun value and luminance

This is true even of two monitors of the same brand. The monitor output for a given gun value may also vary depending on what computer you are using to drive it. If your experiment depends in any way on the luminance of the stimuli then you need to think carefully about your choice of monitors. Either you need to calibrate your monitors, so each monitor is displaying the same luminance value, or for experiments where this is less of a concern you don't want to run everyone from condition A on monitor 1 and everyone from condition B on monitor 2. You also don't want to swop monitors half way through the experiment if you can possibly help it.

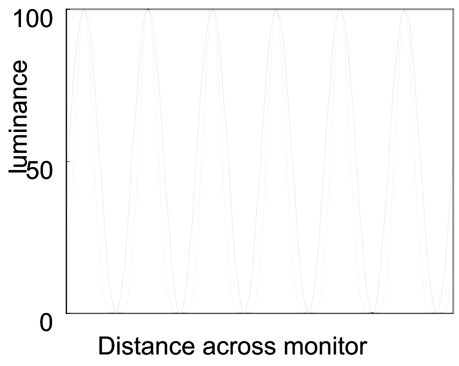
2. Gamma Correction: The relationship between gun value and luminance is nonlinear and different for each gun



Up until now you have expressed image intensity in terms of gun values. But you should be aware that monitors vary a surprising amount in how these gun values translate into actual luminance or chromaticity values, and the relationship is not linear. Here I show how luminance is related to the gun value for the red, green and blue guns, and for all three guns on with an equal value (black line).

One way of thinking about this is that doubling the gun value does **not** result in a doubling of the luminance. In the example shown here the gun value of 75 results in an image that is about 42 cd/m^2, while a gun value of 150 results in an image that is about 109 cd/m^2. Another way of thinking about this is that gun value 100 results in an image that is about 39 cd/m^2 higher in luminance than gun value 50. But gun value 200 results in an image that is about 51 cd/m^2 higher in luminance than gun value 150.

If you are using any stimuli that vary continuously in luminance, like Gabors or gratings, you will need to calibrate your monitor. Otherwise your stimulus will not actually be a grating (or whatever it is that it is meant to be). In the figure here, the black curves represent a true sinusoid. The gray curves represent how that sinusoid would be represented on the screen if you simply sent gun values to a typical monitor without calibrating. Notice how the bright parts of the sinusoid have gotten much narrower, and the dark parts of the sinusoid have gotten much broader. This is because the gamma curve relating gun value to luminance is much



steeper when the luminance is high.

A typical set of gamma correction measurement will look something like this:

GunValue	red Y	green Y	blue Y
0	0	0	0
28	4.4	5.3	.2
57	11.7	14.3	5.7
85	20.1	25.1	9.6
113	29.5	37.4	13.9
142	40.2	51.5	18.8
170	51.2	66.3	23.8
198	63.0	82.0	29.0
227	75.7	99.4	34.6
255	88.6	116.9	40.3

Rather than measuring the luminance output for every gun value we tend to fit these data using a gamma function of the form:

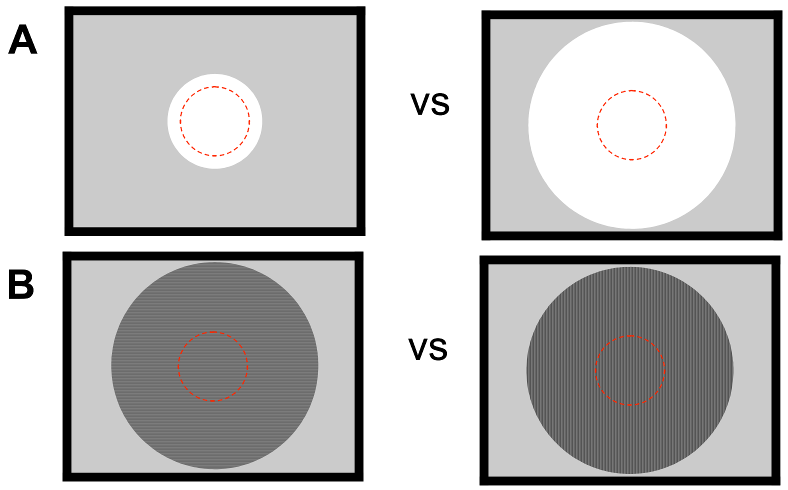
$$I=\alpha g^{\gamma}+\beta$$

Where I is the intensity of the gun (or guns if you are calibrating for grayscale with all thres guns on at an equal gun value), and g is the gun value.

3. Stationarity: The relationship between gun value and luminance/color actually changes over the spatial extent of the monitor

The luminance and color of a stimulus will depend on where it is on the monitor so you should always calibrate the region of the monitor that will actually be used to present the stimuli. If you are running an experiment where contrast is critical (e.g. comparing a contrast discrimination task where the stimulus is presented in different locations of the monitor) then it is worth measuring the relationship between gun value and luminance for the different locations, and making sure the monitor is reasonably homogenous over space. If not you may need different calibration tables for different locations.

4. Power drain – the intensity of a pixel may depend on the extent to which neighboring pixels are also being driven



Note that the amount of power drain tends to be worse between pixels on the same horizontal dimension then for pixels that are aligned vertically (this is certainly true for a CRT monitor, I don't know if LCD monitors have a similar issue). This can be an important issue if you are comparing (for example) contrast discrimination thresholds for vertical and horizontal gratings. More mutual power drain along the horizontal dimension might mean that your horizontal gratings are actually slightly lower contrast. Good monitors shouldn't suffer too badly from this problem, but it's still worth checking. The following would be sensible ways of checking for power drain. The dotted red line represents the part of the screen that the photometer is measuring. Make sure the test patch is big enough to cover the whole region measured by your photometer.

In (A) you check whether the luminance output of a bright patch depends on how much more of the screen is also being driven. You want the measured luminance of the central bright patch to remain constant. If not, then you have a power drain problem. This will be a real nightmare if your stimuli vary in a way that will affect power drain. If they don't then the best work around is to calibrate using calibration patches that are as similar as possible to the stimuli you will use in your experiment. (B) is specifically designed to check whether power drain differs depending on whether stimuli are oriented horizontally or spatially. If you did not see any evidence for power drain with test A you should be OK, but if you are comparing vertical and horizontal gratings it might still be worth double checking using B. Note that the lines have to be small compared to the aperture of the photometer or you will get slightly different values depending on how many lines are fitting into the apertures. You really want the lines to be 1 pixel in width/height.

5. Monitors take time to warm up

When calibrating a monitor or running an experiment you should turn on the monitor at least 5 minutes before the actual experiment begins (often experiments take 5 minutes to get started so this isn't something you have to worry a lot about). If the experiment depends heavily on luminance then you might want to measure the luminance of your monitor every minute after it's been started in the morning and measure how long it is before your monitor is stable.

6. Temporal stability: The relationship between gun value and luminance/chromaticity will change slowly over time

When you are running an experiment you should, before the experiment begins
1) ALWAYS tape over the buttons that allow you to change monitor settings and add a warning to people not to change these without consulting you.
2) Write down the exact settings in the monitor control panel (resolution, bit setting etc. And put a note on the computer warning people not to change those settings without consulting you.

It is amazing how often a new lab assistant will change these settings without telling anyone. This quickly leads to a very bad lab atmosphere especially when the monitor concerned belongs to a graduate student who is trying to finish up their thesis.

Even without someone fiddling with the controls, monitors still change gradually over time, so you should recalibrate periodically. If your experiment depends critically on contrast/color then recalibrating once a months should be fine. Otherwise you should probably recalibrate every couple of months, and certainly when you start a new experiment.

7. Quantization errors.

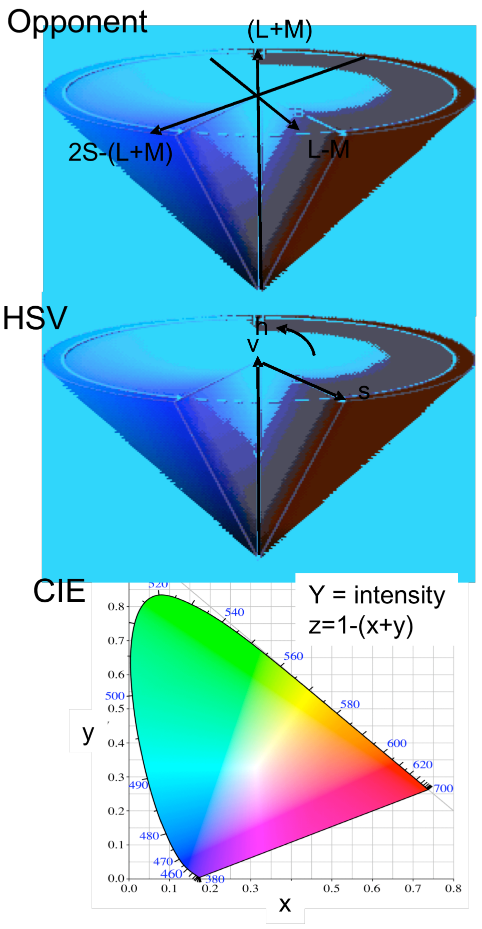
As you have learned, on most monitors the guns only take a certain number of discrete levels, generally 8 bits (256 different levels). When you display an image, you will generally provide a matrix that similarly contains integers between 0 and 255. If you send in non-integers you need to bear in mind that the gun value will simply be rounded to the nearest integer – sending in the gun value 145.3 will simply give you a gun value of 145. So be wary of this when calculating (for example) the rms contrast of a stimulus, or when setting the background to be the mean contrast of the stimulus.

For fine contrast discrimination tasks you may sometimes need to produce more discrete levels than 256. There are a variety of ways to do it, which won't be discussed here.

Color

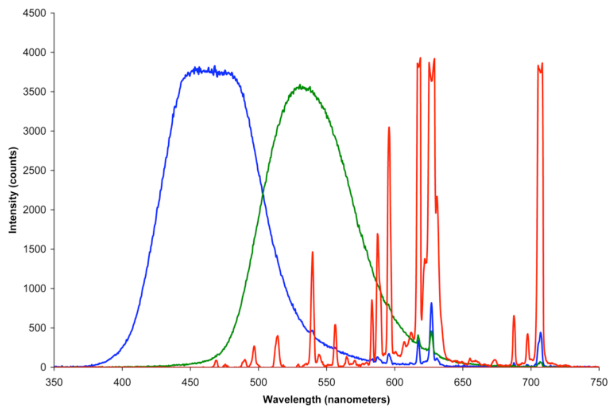
A color space is an abstract mathematical model describing how colors can be represented. Typically, because we have three cones, color spaces are represented in terms of three axes (since more are unnecessary).
Examples of color spaces include:

1. Cone space – color and intensity is represented in terms of the relative and absolute firing rate of human LM and S cones. This color space represents cone outputs, and tends to be mainly used by vision scientists.
2. Opponent color space – This contains three axes – a luminance axis, a red-green axis and a blue-yellow axis. This is again a color space that tends to be used by vision scientists. In this case the color space is thought to represent color processing in the later parts of the retina and the cortex.
3. RGB – color can be represented in terms of red, green and blue intensity
4. HSV – this represents color in terms of hue, saturation and brightness
5. CIE color – This is one of the earliest color spaces. It has the axis Y which represents luminance, X which represents redness, y which represents greenness and z which represents blueness (z=1-(x+y). CIE also uses the vectors X (X=Yx/y) and Z (Yz/y).



1. Different monitors vary in their relationship between gun value and chromaticity

Similar issues apply to color as to luminance – while equal gun values will tend to look gray, in fact that exact shade of gray will differ considerably across monitors. If your experiment depends in any way on the color of stimuli then you need to think carefully about your choice of monitors. Either you need to calibrate your monitors, so each monitor is displaying the same color values, or for experiments where this is less of a concern you don't want to run everyone from condition A on monitor 1 and everyone from condition B on monitor 2.



When you look at the wavelengths emitted by a typical CRT monitor you can see that (as would be expected) the blue gun mostly emits short wavelengths, the green gun emits longer wavelengths and the red gun emits mostly longer wavelengths.
With a high quality spectrometer you can measure the energy as a function of wavelength, usually in 10-30nm bins. These are becoming surprisingly cheap – the GregtagMacbeth Eye One photometer will run you about $250 (they used to run about $7-12k). Cheaper or older colormeters will give you the chromaticity of each phosphor in some sort of 3d color space.

2. Gun drain/power stealing

For some monitors there is power stealing across the three guns. I.e. The luminance output of the red gun is smaller when the green gun is also on at full intensity. The best way to test for this is to measure the output of each of the three guns independently at maximum intensity (gun value of 255) and then measure the output of the three guns on together for a stimulus of the same size. The overall luminance should be the sum of the maximum luminance of the three individual guns. In my experience CRT monitors show a lot of power stealing (up to 10% drop in luminance), so this is something you do need to compensate for when calibrating. If you are using a monochromatic stimulus (e.g. all guns on at equal gun value) then just make sure you use gamma correction tables based on all three guns being on at equal intensities (don't just sum the gamma tables for the individual guns, since you will underestimate the luminance of your stimuli). If you stimuli are varying in color (and aren't just red, green and blue) then compensating for power stealing gets a little more complicated. We'll go into that in more detail later since full chromatic calibration is mostly for the brave.

3. The chromaticity of a monitor is only likely to be stable for intermediate luminance values.

Over a wide range of luminances if you have a fixed ratio of red, green and blue guns, then the stimulus will stay roughly the same in chromaticity – e.g. when red, green and blue guns have the same value the stimulus will be close to gray. This tends to fall apart for very low gun values (of less than 50), and to a lesser extent for very high gun values of 225 or more. So if your stimuli are falling into that range, then you want to be wary of slight shifts in chromaticity for high and low luminance stimuli. This is probably slightly due to slight inaccuracies in the fit of the gamma function for very low gamma values, but is mostly due to the chromaticity of the guns actually changing slightly when they are on at very low or very high intensities.

A typical set of chromaticity values for a monitor might look like this in CIE space:
For red gun

GunValue	x	y	z
0	0.5695	0.1406	0.2899
28	0.5544	0.0899	0.3557
57	0.5182	0.2493	0.2325
85	0.5220	0.1741	0.3039
113	0.5722	0.2164	0.2114
142	0.5025	0.2117	0.2858
170	0.5891	0.2011	0.2099
198	0.5784	0.1498	0.2717
227	0.5089	0.1526	0.3384
255	0.5367	0.1813	0.2820

For a green gun

GunValue	x	y	z
0	0.2407	0.4564	0.3029
28	0.2472	0.4828	0.2700

57	0.3736	0.3916	0.2348
85	0.3028	0.4270	0.2702
113	0.2391	0.4369	0.3240
142	0.2979	0.3893	0.3127
170	0.2436	0.3840	0.3724
198	0.2325	0.4966	0.2709
227	0.2869	0.4506	0.2625
255	0.3477	0.4177	0.2346

Calibration

The goal is of course to be able to measure the luminance and chromaticity of a monitor, and then used those measured values to specify the exact color and luminance you want the monitor to produce. Here's code that will let you do that.

Grayscale calibration

If you don't care about the chromaticity of your stimulus then calibration is very simple. You simply find the best fitting gamma function that describes the transformation from luminance to intensity, when all the guns are on at equal value:

$I=\alpha g^{\gamma}+\beta$

Then you simply need to calculate the inverse of that gamma function, which is of course:

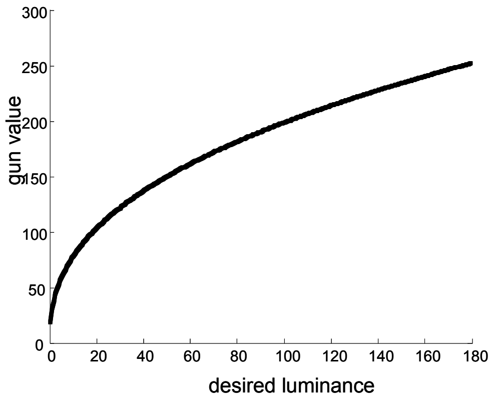
$g=((I-\beta)/\alpha)^{1/\gamma}$

so if we want to find the gun values for a linear set of intensity values we simply plug in a linear set of gun values from the minimum possible, to the maximum possible and calculate the corresponding gun value.

Suppose

```
>calib,gv=[0 28 57 85 113 142 170 198 227 255];
>calib.measuredgray= [0.3000 0.5000 4 10 20 35 70 110 140 180];
>calib.graygamma.alpha=0.0002141;
>calib.graygamma.beta=0.0023;
>calib.graygamma.gamma=2.4666;
```

```
We can calculate a linear
desired set of luminance values
that spans the full range
encompassed by the monitors:
>desiredlum=linspace(min(calib.m
easuredgray), ...
    max(calib.measuredgray), 256);
```



We can then calculate the inverse gamma table – i.e. the gun values you need to step linearly from the minimum luminance to the maximum luminance.

```
calib.inversegray=((desiredlum-calib.graygamma.beta) ...
     ./calib.graygamma.alpha) ...
     .^(1./calib.graygamma.gamma);
```

We can then associate our inverse gamma function with a psychtoolbox window so as to create a linearized window. To do this we first need to create a 256 rows x 3 columns matrix

```
>gtable=[calib.inversegray; calib.inversegray;
calib.inversegray]';
```

Note that because you can't actually get a luminance of 0 the inverse gamma table starts just above 0. You then need to scale the gtable so it goes between 0-1 instead of from 0-255.
>calib.graytable=scaleif(gtable, 0, 1)

Once you have done this, then there is a nice linear relationship between gun value and luminance. If you put in normalized gun value of 1 it will give you maxlum (180), a normalized gun value of 0 will give you minlum, and a normalized gun value of 0.5 it will give you a luminance value that is 90.15
(maxlum-minlum)/2+minlum.

Color calibration using spectral data and calculating LMS values

Let's say you want to produce a particular response in L, M and S cones. The spectral output of the monitor will be a weighted sum of the three phosphors, so the overall light at each wavelength will be:

T = SI – where T is the overall spectral output of the monitor
$I=[I_r ; I_g ; I_b]/(maxI_r + maxI_g + maxI_b)$ – this is a 1 x 3 matrix containing the relative luminance of each gun
$S=[S_r\ S_g\ S_b]$ - where S is a n x 3 matrix where n is the number of samples of wavelength

We can then go on to calculate cone outputs as follows:
$C= [L\ M\ S]^T$ – this describes the spectral absorption spectrum of L, M and S cones.
O=CSI - describes the output of the cones for a given set of luminances for each of the cone types

M=CS is called the calibration matrix of the monitor and should be a three by three matrix
[0.2732 0,9922 0.1466
0.1034 0.9971 0.2123
0.0117 0.1047 1.0]

To calculate the luminance needed on each gun to achieve a desired output for each of the three cones we just work our way backwards

$I=M^{-1}O$

Where O is the desired output for each of the three cone types.

This technique is pretty standard and doesn't take account of gun drain (the luminance when all three guns are on full being less that would be predicted by each gun being on separately at full)