Laboratory 1: Generating random inputs and evaluating some of their statistics

Goals

1. To become familiarized with the MatLab environment, including how to create and use simple functions;
2. To learn about generating random numbers, seeding the random number generator, and controlling random numbers with respect to mean and variance;
3. To introduce elementary statistical ideas about sample size and convergence of a sample average;
4. To produce a MatLab function that generates random normalized vectors that will serve as inputs to your neural networks in future laboratories; and
5. To produce template functions for your figures.

Note: You are not required to do a formal write-up for this laboratory, so don't worry about including an introduction, methods, and discussion section. Your grade will be based on your responses to the exercise questions and on the quality of your figures and figure legends. In order to receive full credit, your figures should be appropriately labeled and you should give a clear explanation of the significance of each figure. Be sure to include the MatLab commands that you used in each exercise in an appendix at the end of your homework.

**Students, please note the existence of four Appendices at the end of this chapter.*

Introduction

In studying the biological bases of cognition, we need more than just a model of the brain—we also need a model of the world that provides inputs to the brain. This is a tall order, too tall. Therefore, we study just one small part of the world and even then with as little detail as sensible. That is, the challenge is to recreate the paradigmatic essence of a cognitive/behavioral problem confronting a living organism. In other words, we need inputs, or an input environment, to present to a computational model of cognition, and this input environment will have certain explicit, and often statistical, characteristics.

It might seem desirable to use real world inputs, but this is usually not feasible. Using real world inputs would require networks that are too large and too complex for beginners (and often too complex for any research lab in the world). Instead, we will create artificial worlds that are, in some specific aspects, prototypical of the real world problems that humans actually solve. A *probability model* is often the preferred way to create an artificial world. A probability model assumes that the real world can be approximated by random variables with one or more underlying probability generators. Of course, the actual generators we use are themselves not truly random but are formed from a deterministic, but chaotic, process. Interestingly, chaotic processes might also be the basis of randomness in the real world.

In mathematics, the technical name for such a generating process is, somewhat confusingly, called a *random variable*. That is, a *random variable* is not a variable at all, but rather it is a function that assigns value to a variable.

Regardless of what we call it, such processes are difficult to characterize and can be extremely difficult to predict.

When a process generates events that are not perfectly predictable, we often use statistics to summarize these events and then use the value of these statistics to predict the value of future events. Statistical averages are useful in this way, and none is more useful than the *mean*. A mean can be calculated for anything that can be measured or counted.

Averages, particularly the mean, are useful to scientists, sports aficionados, politicians, and just about anybody else who wants a better understanding and, perhaps, better control of a quantifiable process. Averages are often used to summarize a set of observations. They are even useful as an incomplete but meaningful way of representing such a data set. Averages can also be used to generate predictions of future events. Because the brain generates predictions, it may be the case that individual neurons generate predictions. If so, we might suspect that individual neurons base their predictions on some kind of averaging process.

Consider two simple examples of quantifiable events, weight and height. The weight of a first-year university student is a variable that can be measured. If we calculate the mean weight of 1000 students, we can use this value to predict the average weight of a different sample of 1000 first-year students. Indeed, we can use this average to predict the weight of a single student. Although this prediction will not be perfect, by comparison to a completely naïve state of knowledge, we are likely to guess closer to the correct value than if we were to guess without knowing this average value.

Simple events without an obvious value can also be averaged in terms of their frequency over time. For example, we can't measure a person walking into the student union in the same sense that we can quantify the person's weight. But we can count the number of people walking into the union between 3 and 4 PM. Thus, if 126 people come into the union over a 60-minute period, then we can conclude that, on average, 2.1 people enter every minute. Such averages are also useful in neuroscience—for example, we can estimate the rate at which neurons fire action potentials, or spikes, by counting the number of spikes that occur during a particular time interval.

Simple Averages

In describing averages, we must distinguish between the *population mean* and the *sample mean*. Both values can be called means—we can think of the population mean as the "true mean" and the sample mean as an "empirically observed mean." For example, if there are three thousand students in the freshman class, the mean constructed by summing over all 3000 weights divided by 3000 is the population mean. Our earlier mean based on just 1000 students is a sample mean.

The mean of MatLab's primary random number generator approaches the value 0.5. The function is called by the command `rand()` and returns discrete values on the interval [0,1] in extremely small increments. As you will see below, a sample mean constructed by averaging a set of numbers produced by the random number generator is only converging towards the population mean and will almost certainly differ from the population mean of 0.5. This is unfortunate because it is the set of population statistics that would be most useful to know. Necessarily then, the existence of such convergence and the rate at which a sample mean approaches the underlying population mean (assuming this theoretical value is finite and the distribution is stationary) is of interest if we are going to use a sample mean for prediction.

When it comes to predicting future events, averages have optimal predictive properties. Because neurons are in the business of prediction, 1) we can hypothesize that synapses construct some kind of average as they incorporate information from the neuronal activity they experience, and 2) even if this is not the case, we can at least compare the behavior of a biological neuronal system to a model, computational system that does construct such averages.

Finally, there is one more reason that motivates our interest in averages. Empirical averages are single values that summarize many observed events from the past. In this sense, such averages are a form of memory, and they are an example—perhaps the simplest example—of abstraction. Memory and abstraction are surely cognitive processes worthy of study and understanding!

Before performing explicit exercises that calculate some averages, we must introduce MatLab.

In-Class Introduction To MatLab[1]

Section 1. Helpful MatLab commands

Here are some MatLab commands every beginner should know.

Command	Description
`>>help command`	Provides detailed information about the command. Examples: **>>**`help help` **>>**`help mean`
`>>CRTL-C`	Hitting Control-C will stop a process that is taking too long or is just not doing what you wanted it to do. This command kills whatever process is running, and then you will be given a new command prompt. Sometimes you must be patient.

[1] Special thanks to Per Sederberg, Adam Soroka, Aaron Shon, and Sean Polyn for their diligent efforts on parts of this section.

If you want to stop working before completing some subproject, it might be useful to save your work. You can use the point-and-click method that is standard for your operating system. MatLab also provides the line commands for saving and loading.

`>>save filename`	Use the save command to save all of your variables to the hard disk. Example: `>>save myvariables` places all the currently valued variables in the command window into a file named myvariables.mat
`>>load filename`	Use the load command to retrieve saved variables from the hard disk. Example: `>>load myvariables` loads the variables in myvariables.mat into the command window or into an .m file that is being executed

(These are line commands—the point and click interface can also be used to save your work.)

If you would like to save your work into a particular directory and file you will have to specify the path.

We will begin our familiarization with MatLab by working from the command line in the Command Window.

You need to get MatLab running on your computer and click onto the Command Window. In the Command Window is the command line prompt, >>.

Note to instructor: *Make sure that each student has MatLab open and running. Read the commands aloud for the students to try. Make sure that the students are able to follow the instructions—walk around, look for people who are having trouble.*

Section 2. Some commands in action

MatLab can be used as a calculator. At the command line type in a sequence of arithmetic operations followed by depressing the enter key. For example,

```
>>3-7*(1/12+3/4)
```

This calculator-like function combined with a vector expressing several data points lets you perform the same set of operations repeatedly. For example, let us subtract 0.5 from three different numbers and divide each difference by 2.

```
>>([0.5,1.4,2]-0.5)/2
```

Did MatLab give you the right answers?

Sometimes you will need more information about a command and its syntax. Usually, this information is readily available through the 'help' command (MatLab also has website help with more details.). At the command line, type in help help then hit return

```
>>help help
```

Try searching for help on something else. The use of punctuation is sometimes not obvious in MatLab, so enter

```
>>help punct
```

There is too much information here, but look at what is said about the semicolon. As for the rest, do not bother memorizing all this information; just remember where to find it.

Although you get too much information again, the following, nonobvious MatLab help category names might be useful later.

```
>>help arith
>>help colon
            %the colon is used often
>>help paren
            %parentheses and brackets will be used quite often and
            %they have very different syntactical implications
```

The most basic thing you will want to do when programming is to store a value in a variable. Example: to store the value 100, we create variable A valued at 100; enter

```
>>A = 100
```

Note that the symbol '=' does not mean equality; rather, it is an assignment function.

You will also, and quite often, perform a mathematical operation on a valued variable. For example, because the variable A now has a value, you can take its square root; enter:

```
>>sqrt(A)
            %note that the value of A is not changed by this
            %operation so the value of sqrt(A) is not saved
            %anywhere
>>A
            %A still has its original value
```

However, if a variable is not valued, an error will occur.

```
>>clear A
            %note the syntax: see help clear
>>A
            %check to see that A is cleared
>>sqrt(A)
```

This `sqrt( )` example also serves as a generic illustration: mathematical and plotting functions in MatLab operate on data and the valued variable holding this data is enclosed by a parenthetical expression that follows immediately (no spaces!) after the function name.

You should always feel free to play and explore at the MatLab command line. For example, after valuing A, you could enter the error-filled sequence of commands

```
>>A = 2
>>sqrt (A)
            %this is wrong
>>sqrt A
            %and so is this
```

and finally, let's get it right

```
>>sqrt(A)
```

Section 1.1. Doing arithmetic with vectors

Most of the basic commands in MatLab can operate on vectors. A vector is an ordered list, and a specific vector is defined by placing a sequence of numbers within square brackets. Here we construct a row vector using spaces to separate each scalar value (commas would also work):

```
>>v = [3 1 10 11]
v =
3 1 10 11
```

To see this valued vector, simply type its assigned label,

```
>>v
v =
3 1 10 11
```

Of course, you can now do arithmetic on each and every element of this vector all at once if the arithmetic operations are term specific. See what the following produces

```
>>(v - 0.5)/2
```

You can also select particular elements in this vector using parentheses. For example, to view the first entry of v, the vector, simply type

```
>>v(1)
ans =
     3
```

or to see the second

```
>> v(2)
ans =
1
```

or perhaps you want to see, or use, a consecutive subset of the vector's scalar values, e.g., the second, third, and fourth elements of vector v,

```
>>v(2:4)
1 10 11
```

and this use of the colon conforms to what `help colon` says.

Valuing a matrix is much like valuing a vector, but with an additional separator. One way to parameterize a matrix is to enter each successive row separated by a semicolon. Here we create a matrix made up of two, three-dimensional row vectors.

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

And here we create the same matrix as three, two-dimensional column vectors

```
>>b = [[1;3], [2;4], [3;5]]
```

We can select a single value from a matrix much like from a vector, but we must specify both; e.g., a particular row and a particular column

```
>>b(2,3)
```

which selects the scalar at row two , column three,

or we can select an entire row or column. Here, using the colon, we print out the entire first row

```
>>b(1,:)
```

which is equivalent to the more obvious

```
>>b(1,1:3)
```

We can also use colons to generate a sequence of values.

```
>>temp_vec = [0: 0.2: 1.9]
```

Note the syntax of this uniformly growing sequence of numbers – starting point (here it is 0), then increment (0.2), then maximum allowed value (1.9, which may not be reached).

Now let us look at simple addition and subtraction of vectors and also introduce the transpose operation. As noted above, we define our vectors by typing the variable name and setting it equal to a set of numbers listed in brackets. If we write them in a row with spaces or commas, we define a row vector; if we write them in a column, we define a column vector. We can easily switch between row and column vectors, however, by using the transpose command. The example shows how the variable defined as a row vector becomes a column vector by taking the

transpose. As you will later encounter, the column-versus-row distinction has a technical meaning in linear algebra.
The transpose command, which is an apostrophe ('), turns a row vector into a column vector or vice versa. We will define two column vectors.

```
>>columnvector1 = [1 2 3]'
columnvector1 =
     1
     2
     3
>>columnvector2 = [2;4;6]''
columnvector2 =
     2
     4
     6
```

When two vectors are of the same dimension they can be added. Addition occurs element by matching element.

```
>>columnvector1 + columnvector2
ans =
     3
     6
     9
```

Now subtract them, element by element,

```
>>columnvector1 - columnvector2
ans =
    -1
    -2
    -3
```

Thus, addition and subtraction of two vectors is performed on matching elements of the two vectors. This is not optional, it is required! The above operations are allowed (that is, they are mathematically defined) because the vectors above are both column vectors with three entries. You have to be careful when doing addition and subtraction of vectors and matrices. If the dimensions of the two objects do not agree, then an error will occur because the operation you request is undefined in a mathematical sense.

Two vectors or two matrices must agree exactly in form if you want to add them together (or subtract them). For example, using v from above,

```
>>v
            %v should be a four dimensional row vector from above
>>columnvector1 + v
```

gives the following error:

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

Let's try it again.

```
>>columnvector1 + v'
```

Still not right; now try

```
>> columnvector1 + v(2:4)'
```

Ok! Remember, you can only add two vectors if they have the same number of the same kind of elements. That is, the two vectors must both be row vectors or both be column vectors of the same dimension, otherwise the elements will not be perfectly matched.

Of course, there is a mathematically defined exception which you already know about: A scalar can operate on a vector (or a matrix for that matter). For example,

```
>>columnvector1 - 0.1
```

is interpreted as

```
>>columnvector1 - [0.1;0.1;0.1]
```

There are several types of multiplication involving vectors. However, linear algebra defines a subset of three of these, and all of these are easily implemented as MatLab commands. The easiest is a scalar times a vector—for example,

```
>>scalar = 10
>>scalar * columnvector1
```

Thus a scalar times a vector is just the scalar times each element of the vector.

We will encounter a second kind of vector multiplication before the end of this lab.

Generating random numbers

Let's use MatLab's random number generator to create a three-dimensional column vector that might be used as an input to a neural network. Enter the command

```
>>ColumnVectorA = rand(3,1)
```

This command generates an ordered list of three pseudo-random numbers. This ordered list, or vector, is specifically ordered with three rows and one column corresponding to the (3,1) of the `rand()` command.

`rand(3)` produces nine values; in general `rand(`**`scalar`**`)` produces a square matrix of size **scalar-by-scalar**.

You should memorize this row-by-column notation, (row, column). This is the standard of linear algebra, and it is important in this class because MatLab requires it. That is, if your notation is not consistent with the row-by-column notation, you will get errors when you try to implement commands that require the proper dimensions of input vectors and matrices. So remember the notational rule when creating or operating on a matrix: row-by-column.

Since we have assigned three values to `ColumnVectorA`, let us use it to be sure we understand the command `clear`.

```
>>ColumnVectorA
```

MatLab should have printed the three-dimensional column vector of random numbers to the screen.

`ColumnVectorA` will contain this set of numbers until you clear `ColumnVectorA`, until you set `ColumnVectorA` equal to something else, or until you exit MatLab.

Assume `v`, `columnvector1`, and `columnvectorA` are still valued. Enter each of them to be sure.

```
>>ColumnVectorA
>>v
```

Now let us unvalue one of them.

```
>>clear ColumnVectorA
            %unassign ColumnVectorA and only ColumnVectorA
>>ColumnVectorA
            %ColumnVectorA no longer implies any set of values
>>v
            %But v and ColumnVector1 still are valued
>>columnvector1
>>clear
            %not anymore
>>v
>>columnvector2
```

Every time you start a new series of calculations you should play it safe. First, save any variable values you might need. Then, to make sure there are no old values hanging around, use the clear command. Take a look at

```
>>help clear
```

and take a look at the appendix on how to program.

There are many different generating processes that depend on the probability distribution used to generate random numbers. MatLab's `rand( )` command picks numbers between zero and one with a seemingly uniform (i.e., with the same) probability for picking any one particular value.

You can, however, alter what this generator produces. You will learn how to shift and rescale the random number generator in later in section 3.

Section 1.2. Functions useful for calculating network statistics

First let's create a valued column vector of dimension four

```
>>randVec = rand(4,1)
```

Now let's average its elements and find the sample variance

```
>>mean(randVec)
>>var(randVec)
```

You can save keystrokes using the tab key for completions and the up and down arrow keys to skip up and down to commands you have entered recently. Also, if you type the first few letters of a command you have already entered and then press the up arrow, it will go to the nearest command matching those letters.

Table 1.1.

The variance calculated by the MatLab command var( ) is the sample variance, not the population variance.

Def: $\text{population\_variance} = \dfrac{\sum_{i=1}^{n}\left(x(i)-\text{pop\_mean}(x)\right)^2}{n}$,

where the population is finite with *n* members and where $x(i) = 1, \ldots, n$ is the complete list of all *n* members.

For the sample variance, the $x(i)$ are random samples taken from the population.

Def: $\text{sample\_variance} = \dfrac{\sum_{i=1}^{m}\left(x(i)-\text{sample\_mean}(x)\right)^2}{m-1}$

Def: $\text{sample\_mean}(x) = \dfrac{\sum_{i=1}^{m} x(i)}{m}$

You are about to encounter the first of the homework exercises. To make your task much easier (and to teach you one of the ultimately important methods of modern programming), let us learn about programming your own, custom MatLab function. MatLab functions are always stored as .m-files, so they are often called a "dot m file."

Software development

In MatLab, the user-programmed function is a superior technique for program development. The reason for this superiority may be difficult for novice programmers to accept; nevertheless, it is the result of hard-won knowledge coming at great expense. The expense I am referring to is the time wasted on computer programs that did not perform as intended.

It turns out that there are only a couple of universally accepted methods to reduce the enormous amount of time people spend correcting computer programs. These two methods are (1) reuse and (2) encapsulation.

Reuse means that you should always be building your bigger programs out of older and smaller programs, where such older programs have been extensively tested and are known to be correct.

Encapsulation refers to the techniques of passing data into and out of a calculation, i.e., into and out of a function, in a way that merely parameterizes, but does not change, the function. Thus, when you reuse a well-tested function, you do no damage to its functionality by the data that is the input, or output, of the function.

Constructing Your Own Functions

If already showing, click on the editor window with your mouse. But if no editor window is showing, then close all windows but your Command Window, and in the Command Window, click on File; then click on the menu item New; then click on submenu item M-File. (Or instead of all this mousing, if you prefer to work at the command line, just enter >>edit *yourfilename*)

You now have a blank editing window that can be used to write a MatLab function, among other possibilities. Therefore, let's inform the editor that this is a function by typing the obvious; i.e., type the word "function". If you typed it correctly, the word will be colored blue.

We will make a function out of the two sample statistics, so let's name the function stats. The input will be a vector of data called dataset and the output will be the sample mean and the sample variance. Here is what you should type into the editor, on the same line as the blue colored word function. While staying in the same line, add at least one space, and then enter

```
function [samplemean, samplevariance] = stats(dataset)
```

Note the syntax. The output appears in square brackets to the left of the assignment symbol (=); to the right of the assignment symbol comes the name of the function which is immediately followed by a parenthetical expression. Inside these parentheses are the inputs on which the function operates. In general, the first line takes the following form

```
function [output data] = function_name(inputdata)
```

Now let's program the function; typing somewhere below this first line,

```
Samplemean = mean(dataset);
Samplevariance = var(dataset);
```

This is the end of the function, so save it with the visual interface and return to the command line.

(Note the special use of the semicolon at the end of a line. It prevents screen printing when executing the line; refer to `help punct`.)

Test your program with something simple.

```
>>stats([7])
```

For this singleton data set, the mean is seven and the variance is zero.

Now annotate the function; that is, write what the function does (use the % symbol at the beginning of each annotation line), and write what the input must be. Then save this function by clicking on **File**. Note that MatLab automatically names the file stored on the hard disk, but just to be sure, always click on **Save As** the first time you save a function in the editing window (later you can use the diskette/floppy disk icon in the toolbar). Also, MatLab will try to write the filename as the function named in your first line, so watch out if you change the name of the function or write a derived function. That is, don't lose your editing!

Saving such an .m file is very important! The Command Window, as well as other functions, can only use functions that are saved. (A common error made by students is to edit an existing function, and then they run a program that calls this function although the newly edited function has not been saved. Much confusion results because MatLab runs the old, pre-editing function, not the new version. In fact, MatLab tries to tell you that saving is needed by an asterisk on the banner and by the shading of the floppy disk icon.)

So remember, save an edited function if you want to run this new version!

Now let's use the function that was just constructed. Click on the Command Window and construct a data vector.

```
>>unif_rand_data = rand(1000,1);
            %Here you really need to use the end-of-the-line
            %semicolon.
```

Now we call the function just programmed and assign the output to variables named `mn` and `vr`.

```
>>[mn, vr] = stats(unif_rand_data);
>>mn
>>vr
```

Note how we called the function, by its name `stats`, while also supplying a suitable input, `unif_rand_data`. Also note how we assigned the output of the function to variables `mn` and `vr`.

In general, a function is called by its name and placed on the right side of the assignment symbol (the equals sign, =). Following the function name in parentheses is the data and other information the function needs to do its computation. To the left side of the assignment symbol are square brackets; the variables inside these brackets receive the functions output.

What can go wrong? Lots of things, but here is one thought to keep in mind. When you call a dot m function in the Command Window or when a MatLab program or another MatLab function calls such a function, the system looks for your function in the current directory unless you inform the system otherwise. If you put your function elsewhere (which you must in the long-term if you are using a shared system), then you must give the path to that function as part of calling it. To see what such a path looks like, note what is listed when you run the Save command.

You are strongly encouraged to write and use functions for any of the exercises (i.e., you will be graded down if you don't). For the next in-class exercise create a function that returns the following three sample statistics: mean, variance, and standard deviation (std). Test it and keep it around for your homework.

Homework 1.2.1.

1. Calculate the mean, variance, and standard deviation for an arbitrary set of numbers in the form of a vector.
2. Test your function `stats`. Test two data points (pick convenient values besides zero and one). Continue testing, this time using ten randomly generated data points. Fix the function if it is not working correctly.

Creating plots and producing a reusable template

Let's just take a quick look at a command that generates a figure.

```
>>clear

>>data = rand(5,1)

>>plot(data)
            %the five random data points are plotted and a line connects each
            %successive point it is a pretty nonsensical plot
```

and a variant of this command

```
>>plot(data, 'o')
            %the same data are replotted and, by specifying a particular
            %symbol for each datum, the default line connecting successive
            %points is disabled
```

Lots just happened, but you must have noticed how the second plot destroyed the first one. You can do a little better job of controlling things.
(In what follows, we will call two of MatLab's trigonometric functions. These functions take values in radians. In the next chapter you will find radians convenient. Just remember 90 degrees is equal to $\frac{\pi}{2}$ radians.)

Critical but nonintuitive plotting commands:
The commands '`clf`', '`figure`', and '`hold on`' are very useful. Let's use them to plot two figures with slightly different sets of commands. Enter

```
>>clf
            %remove all figures
>>points = [0:2*pi/1000:2*pi];
            %this command creates 1001 values, from zero to 2π,
            %evenly spaced by an interval of 2π divided by 1000.
>>plot(sin(points))
            %MatLab trig functions take values in radians
```

You should now see a window that is called figure 1; there should be a sine wave plotted in this window. Note that it is valued zero at the origin and valued one at pi/2. Now enter

```
>>plot(cos(points))
```

Although you still have something called figure 1, it is a different figure 1. A cosine wave has replaced the sine wave.
Now let us do almost the same thing in two other ways. First we get rid of the last plot, and then create two distinct plots as figure 1 and figure 2.

```
>>clf
            %figure is cleared and will be reused
>>plot(sin(points))
>>figure
            %this is the critical command. Note that a new, blank
            %window appears labeled figure 2.
>>plot(cos(points))
```

You should now have a figure 1 and a figure 2.

Now let us start over.

```
>>clf
```

this time we put both plots on a single figure.

```
>>plot(sin(points))
>>hold on
            %this command keeps us from losing anything that is
            %already plotted in the current figure window
>>plot(cos(points))
```

You should have both the cosine and sine plots on figure 1.

And here is something fancier

```
>>figure
>>plot(cos(points), sin(points))
>>axis([-1.1,1.1,-1.1,1.1])
            %I wanted you to see a circle but it looks like an oval. The
            %problem is MatLab's default aspect ratio, called 'normal'.
```

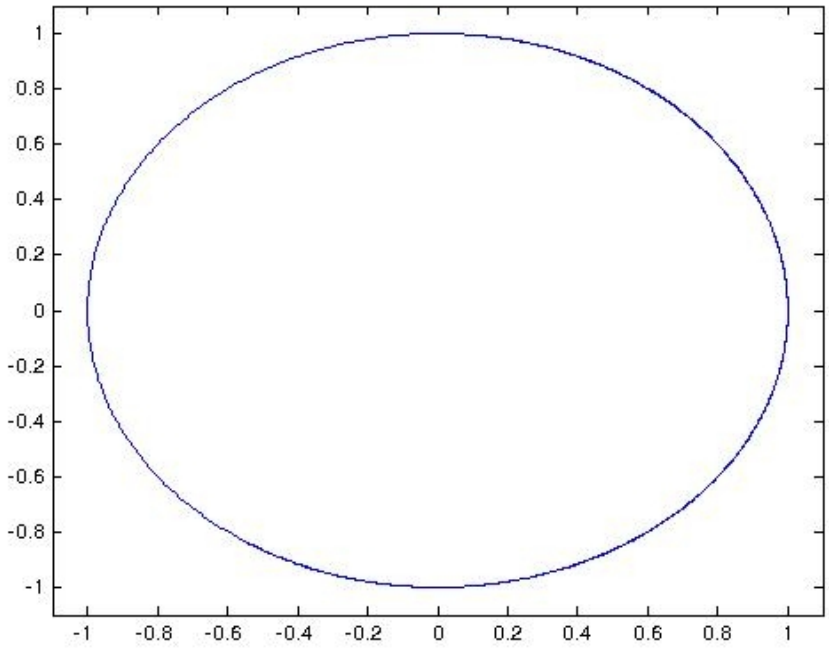


Fig. 1.1. Unlabeled plot that was supposed to be a circle. MatLabd has its own way of scaling the x and y axes. You will need to explicitly overwrite this default setting.

```
>>axis('square')
            %fix this by forcing the x and y axes to the same
            %length. You need to use this command whenever you are
            %trying to illustrate a two-dimensional plane
            %geometric space.
```

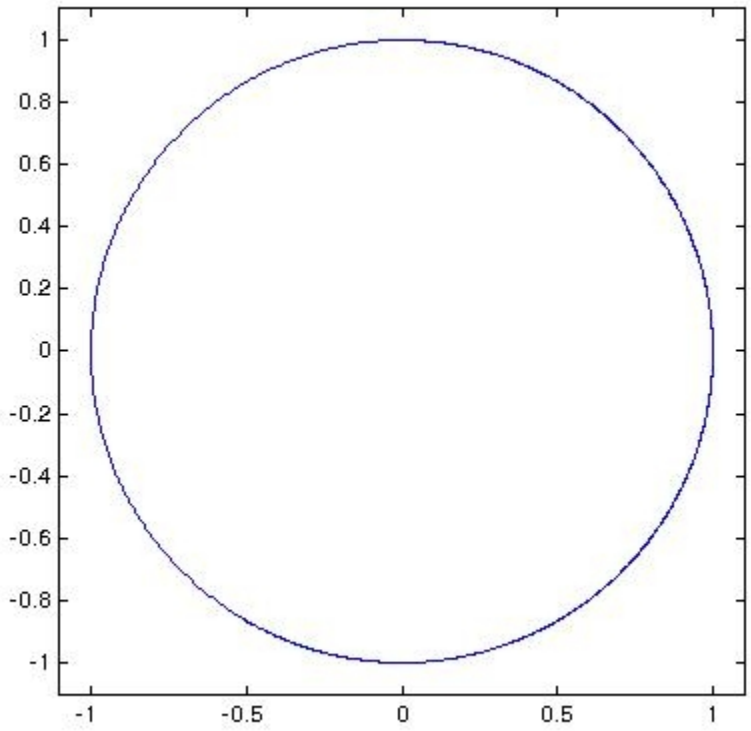


Fig. 1.2. Unlabeled circle. The same data in the previous figure replotted with the x and y axes squared. That is, the x and y axes have been set to the same scale.

Displaying data points in graphical form is required for your lab reports, but more is required. Your figures need to be properly labeled.

More Software Development – Automating Figure making.

Figure making used to be a very time consuming process. There are many details that comprise a proper figure; pleasantly however, MatLab goes a long way in relieving the tedium after you have worked out a format that you want to reuse.

First we need a figure to work with so we begin with the plot command. Here is the tedious but necessary list of facts and comments you will need for every figure.

1. your name
2. the date
3. the title of the figure
4. the labels for both axes
5. the scale of the axes

For a two-dimensional parameter plot, the x-axis values no longer default to 1,2,3,…n. Rather a n-by-2 matrix is used. This matrix with 2 columns and n rows produces a plot with n points. The first column of values are x-axis values and the second column of values are the corresponding y-axis values.

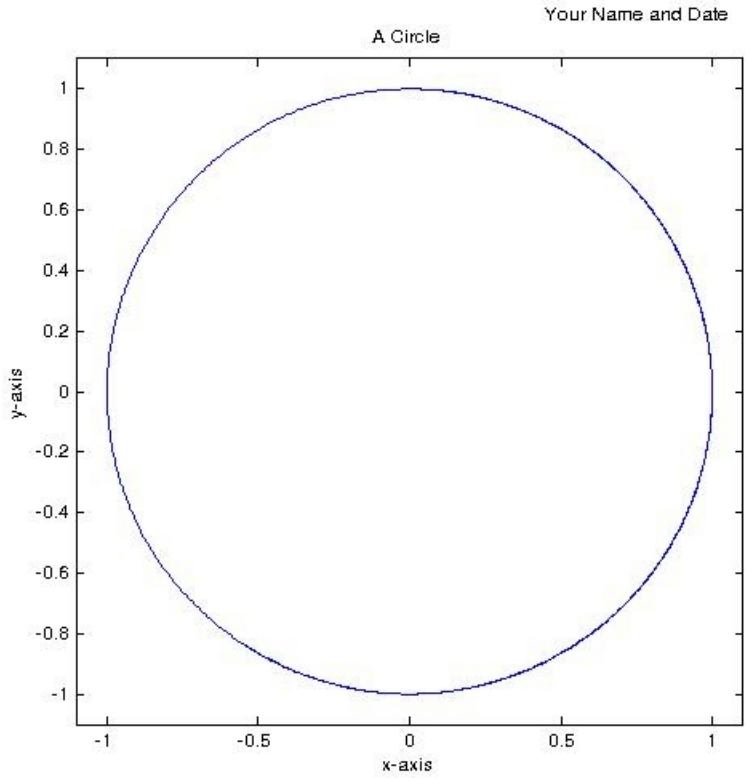


Fig. 1.3. Properly labeled circle. The previous figure is enhanced with the proper labels. These labels include: the figure title, your name and date, and labels on the axes.

To improve the aesthetics of your plot use the Property Editor which can be found under the tool bar item **View**, or click on the white arrow on the tool bar, then right click on the feature you want to edit.

As before, many details of the plot can be altered while working with the Property Editor. For example, move the cursor to the x-axis line and right click. You can change things such as line thickness and color (or gray scaling if you do not want to use a color printer). A variety of lines, arrows, boxes, and more can be added to your figure. After you have appropriately modified the figure, we will save the properties of this modified figure as a function.

Saving a figure and using it in your lab report

To save your figures, use the point and click method (i.e., the graphical user interface). At the figure you wish to save, under the **File** menu click on **Save As**. Here you can specify the file format for saving your figure; within the **Save As** menu choose "type." You will often want to save the same figure twice. First save it as a .fig so you can recall it for editing purposes. Then save it for insertion into your lab report. The format-type that you will want to use depends on what you will do with the figure and memory limitations. We have had satisfactory experiences with the JPEG format, saving the figure at full size, then importing into the Microsoft Word Editor and, there, resizing in the Word Editor. Even better quality can be obtained by using EPS or TIF but your word processor may find it difficult to handle these formats .

We can convert this figure into a function by virtue of a remarkably insightful feature of MatLab. Create the function `skeletonplot(xdata, ydata, newtitle, newlabelx, newlabely)`. Then go through the following steps:

In the current figure window, click on **File**, and find the option **`Generate M-file`**. Now when you now click on this option, MatLab produces a function called `createfunction` in the editing window. This function has all the information in your current figure except the data. To make your own personal skeleton plot function you need to edit createfunction, change its name and, add comments to this function. First, change the name of the function and adjust the inputs to the function. (Be sure you save this function – **Save As** – with the name you just used!) You will want to send new data into the function.

For the purposes of this course your name will not change, therefore make this a permanent feature of your skeleton plot. That is, enter your name into a textbox and place it in the upper right hand corner. Also in this same textbox, type in the word '`date`'. Finally, choose no line for the textbox using the line style.

Note: You can add some other commands to the function. See the skeletonplot.m function I made (Appendix 1.2).

Generating uniformly spaced points on an arc of a circle

This section will help you generate the points in two-dimensions that are needed in the next laboratory. Specifically, you will need to generate random points that are limited to a specified arc in the positive quadrant.

You have already seen how to generate evenly spaced points on a circle using trigonometric functions. We will continue with this method for random points localized to a specific arc.

Starting with a circle of radius one, the circumference is 2π. An arc that is one-quarter of a circle (90°) is $\pi/2$ radians.

Table 1.2.

The arc-length in radians subtended by angle alpha is $\alpha \text{ radians} = \frac{\alpha \text{ degrees}}{360 \text{ degrees}} * 2\pi$
because the circumference of a circle is 360° or, equivalently, 2π radians.
Thus, 1/4 circumference = $\frac{\pi}{2}$ radians.

Suppose we need to generate one hundred and one evenly spaced points within a particular arc. If the arc is a full circle, then the MatLab command is

```
points = [0:2*pi/100:2*pi]
            %note that MatLab knows a value for pi
```

If we want the one hundred evenly spaced points to be restricted to the positive orthant, then the MatLab command is

```
positive_points = [0:pi/2/100:pi/2]
```

Use the plot command to create a figure for each of these data sets.

For generating points uniformly spaced between 30° and 90°, the line command might be

```
>>thirty_to_ninety = [pi/6:(pi/2-pi/6)/100:pi/2]
```

Check this by plotting the sine and cosine values and be sure to square your axes.

Section 1.3. Relocating and rescaling the random number generator

The default range for the command `rand()` is [0,1), i.e., zero is possible but one is not as it is outside the range. However, [you might need a sample of random numbers with a different center point and a different range. By adding or subtracting a constant, you can move the range, and by multiplying or dividing, you can change the width of the range. The following is an example of shifting and rescaling the random number generator.
Here you will generate random numbers on intervals that are different from the [0,1) MatLab default. First, generate nine random numbers on the interval [–.5, +.5).

```
>>rand(9,1)-.5
            %subtract 0.5 from each random number to relocate
            %the default interval
```

Basically, what you are doing here is generating 9 random numbers between 0 and 1 and then subtracting .5 from each one to shift the scale.

Now let's generate five random numbers on the interval [0, .5).

```
>>(rand(5,1))/2
            %contract the interval by dividing by 2; this also
            %relocates the interval
```

Here, you are generating 5 random numbers between 0 and 1 and then dividing each one by 2 to reduce the range of the scale to [0, .5).

Now compare the estimated mean and variance for the two different ranges sampled above. For a *uniform distribution* with range [a,b) and an infinite number of samples, the mean is (a+b)/2 and the variance is $(b–a)^2/12$. Your observations will be a little different than you might expect because you are using a finite number of samples.

Now generate 100 random values from zero to 30°.

```
>>randpoints=rand(1,100)*pi/6;
            %generate 100 random points from 0
            %to π/6 and suppress
            %printing
```

Plot the sine and cosine values of these.

Note how we scaled the random values by pi/6. For comparison, generate 100 points that are uniformly distributed on the all-positive portion of the unit circle.

The use of sine and cosine to locate a point on the unit circle works fine in two dimensions but will not work with points that are three-dimensional or higher (i.e., points on the surface of a sphere or hypersphere). An alternative and more general way of generating random points on the surface of an *n*-dimensional hypersphere with radius one uses the method of normalization to the unit (of course, the unit hypersphere in two-dimensions is the unit circle). Normalization will be covered in section 1.4 of this lab.

Section 1.4. Seeding the random number generator

The reproduction of random sequences is very useful, but a truly random sequence (or even a pseudorandom sequence) is not reproducible on demand unless the sequence has been saved. However, saving a long sequence of random numbers is unnecessarily burdensome. Because computer-generated random numbers are, properly speaking, pseudo-random numbers, we can generate the identical sequence whenever we want so long as we start the generator the same way. Use a positive integer to initialize the generator (see >>help rng for details). The initialization is controlled by a seed. Here is an example that shows the repeatability of the sequence.

```
>>rng(1204513528) %1204513528 is an example of a seed value
>>rand(1,3)       %compare the vector generated to the next one
>>rand(1,3)       %this second vector differs from the first
>>rng(1204513528) %reseeding with 1204513528 allows us to repeat
>>rand(1,3)       %the first random vector
```

Important Tip: Remember that it is important to seed the random number generator at the beginning of the code for each homework assignment. Your seed must be unique to distinguish your work from all your colleagues—use a value like your phone number. In any case, do not lose your seed values—without them, you cannot go back and check your results**.** Because you need to seed your random generator for all homework, write, test, and save a function, such as this one:

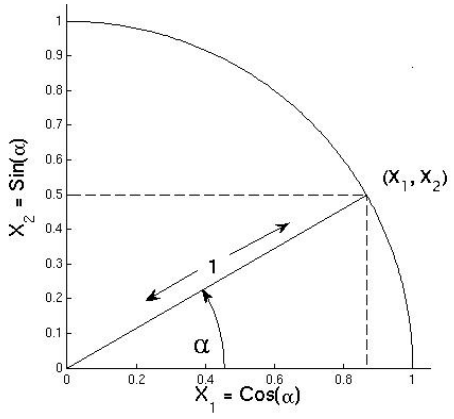
```
function    randseed(seed)
            %you can use this function with a new seed whenever you
            %want or your can use  your personal, standard seed
            %this standard seed will be used if you call this
            %function with a negative value of seed

disp('my seed is')
disp(seed)
            %then seed the default (twister) pseudorandom number generator
rng(seed)
            %this syntax is valid and preferred for the newest
            %version of MatLab (I am using version 2011b). Older
            %versions back to 5.0 or so will use rand('state',seedchoice)
            %The function ends here.
end
```

Alternatively, you can have a function `seedrand` that always seeds the generator with a fixed seed. For example:

```
function    seedrand()
%this function always reseeds the random number generator with the same seed
rng(12345678);
end
same
```

Homework 1.4.1.

Generate uniformly random points limited to the arc that goes from (π/6 to π/2 radians).
(Hint: Use the [0,1) uniform random number generator. For each random number generated, you must apply a rescaling and a shift. Instead of the interval being length one, the interval is length π/2 - π/6 = π/3. Therefore, rescale the length (π/3) (multiply). Instead of starting at zero, the interval starts at π/6. Therefore, shift the interval by π/6 (add).)



$$\sin(\alpha) = \frac{\text{side opposite}}{\text{hypotenuse}} = \frac{x_2}{1} = x_2$$

$$\cos(\alpha) = \frac{\text{side adjacent}}{\text{hypotenuse}} = \frac{x_1}{1} = x_1$$

Fig. 1.4. (figure name: skp11.jpg) The cosine and sine of a point at distance one from the origin locates its position in the same terms as the point's coordinates relative to the abscissa and ordinate. For a right triangle, remember that sin(α) equals the side opposite divided by the hypotenuse and that cos(α) equals the side adjacent divided by the hypotenuse.

Section 1.5. Vector normalization

A condition typically enforced on the inputs to a neural network is normalization of a multidimensional input or, in more realistic models, approximate normalization. Normalization makes pattern recognition much easier. The idea is that the absolute strength of a signal often says nothing about its meaning—it's like playing the radio at two different volumes. The songs, weather, and news reports are the same, but the signal is louder when the volume is higher. Similarly in the visual domain, if we view an object in dim light or in bright light, we should still perceive the same object. Thus, a pattern recognition system should be able to scale the amount of illumination or even the amount of contrast and get the same result. Indeed in terms of biology, there are neurons sensitive to global light levels (and even to global contrast) that make such scaling not only sensible but also feasible for the nervous system.

From a purely technical viewpoint, *Normalization* is a technique to standardize the length of a vector so that all vectors are of the same length. In particular, the length of one is often chosen for theoretical studies because it can be so easily rescaled and because of the relationship, via the cosine function, between such scaled points (see Appendix in Lab 2).

There are various (infinite) kinds of normalization. We will use the most common, *Euclidean normalization*. We call it Euclidean because we use Euclidean geometry to define the measure of length. The Euclidean distance is a straight line between two points. When orthogonal axes are used to define the position of a point relative to the origin (probably the only definition you have ever encountered), the point (or, equivalently, the vector) has a length that is the length of the straight line from the origin to this point. This length is calculated in terms of the position of the point as defined by the right triangle relationship, $a^2 + b^2 = c^2$; in particular, the length of c is $\sqrt{a^2 + b^2}$ where, in two dimensions, a and b are the values along each axis that locate relative to the origin. For a two-dimensional vector $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, the distance of this point from the origin is the Euclidean length of the vector, $\sqrt{x_1^2 + x_2^2}$. Normalization makes all vector lengths equal to one. Thus, we can always take our original vector and project it up or down to the circumference of the unit circle centered at the origin so that $\frac{x}{\sqrt{x_1^2 + x_2^2}} = \begin{bmatrix} \frac{x_1}{\sqrt{x_1^2 + x_2^2}} \\ \frac{x_2}{\sqrt{x_1^2 + x_2^2}} \end{bmatrix}$ is certain to be a vector length one so long as x_1 and x_2 are not both zero. This idea easily generalizes to any number of dimensions. For example, in three dimensions, $x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$, and the normalized vector is

$$\frac{x}{\sqrt{x_1^2 + x_2^2 + x_3^2}}.$$

Using MatLab, it is very easy to calculate $\sum_{i=1}^{n} x_i^2$. This sum of squares is the scalar product of a vector times itself. That is, the inner product of a vector with itself squares, then adds, each term. Using the notation of linear algebra,

$$x^T x = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = x_1 * x_1 + x_2 * x_2 + x_3 * x_3 = x_1^2 + x_2^2 + x_3^2$$

These are equivalent terms in linear algebra:
inner product ≡ scalar product ≡ dot product

Now if you take the square root, this is exactly what you need in the denominator when you normalize your vector, term by term.

Here is an example where a single three-dimensional input is generated then normalized to the unit circle:

```
>>x = rand(3,1)
            %generate three random numbers as a column vector
>>innerprod = x' * x
            %left multiply x by its transpose to get the inner
            %product, which is a scalar
>>normalizedx = x/sqrt(innerprod)
            %normalize the original vector by dividing each of the original
            %random numbers by the square root of this scalar
>>normalizedx' * normalizedx
            %check to see if the new values are on the unit circle
```

Now we can try plotting a normalized vector so that we know how to visualize such data. (Remember the vector we use is a point, not the line drawn from the origin to that point.)

Homework 1.5.1.

Create 200 random, two-dimensional vectors. Each two-dimensional vector is a point in the plane. Normalize each such point to the unit circle. Normalize one point at a time; you should end up with 200 normalized points. Plot these vectors, both before and after normalization, as points. Be sure to square the axes of your plot. (See Fig. 1.3. Figure 1.3 introduced squared axes; figures 1.5 and 1.6 illustrate output for this homework. You will need to use a for-loop. Be sure to read about for-loops in MatLab by invoking `>>help for`

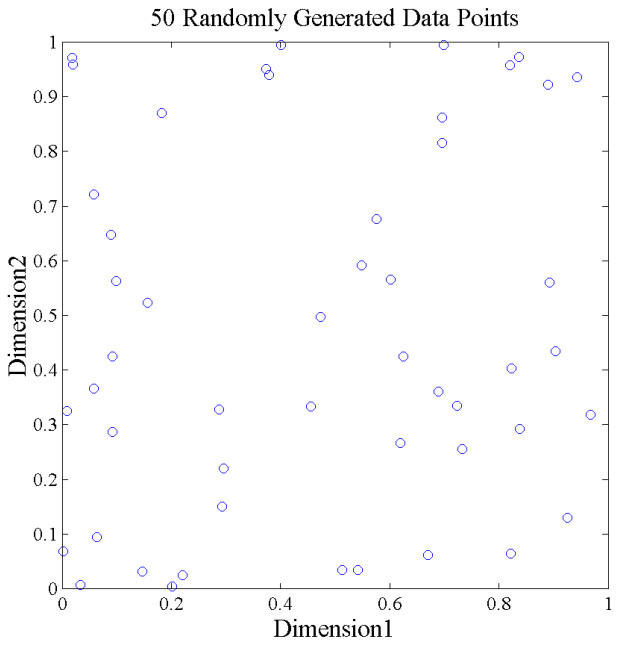


Fig. 1.5. Scatterplot of the pairs of 50 random numbers before normalization to the unit circle. Note how the points are scattered all over the graph. The vector named Dimension1 consists of n uniformly distributed random values Dimension1(1), Dimension1(2), ... , Dimension1(n). Similarly the vector Dimension2 also has n uniformly distributed random values, where n = 50. The data points plotted here are pairs of values (Dimension1(i), Dimension2(i)), where $1 \leq i \leq n$; e.g. the first point is (Dimension1(1),Dimension2(1)).

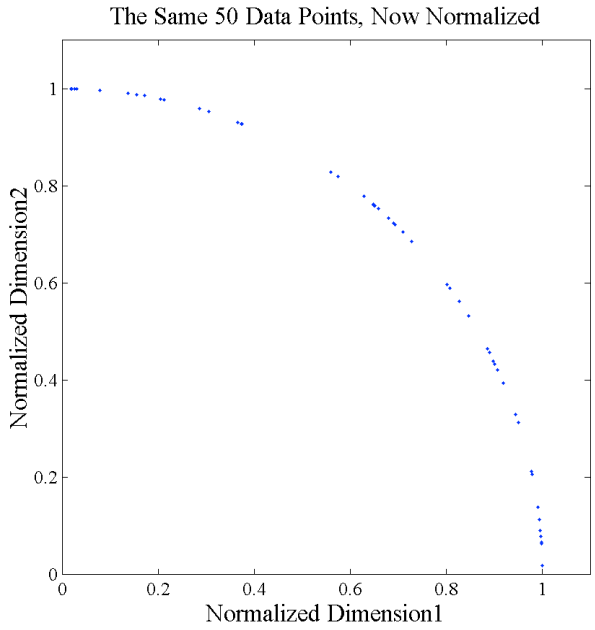


Fig. 1.6. The same 50 data points are normalized to the unit circle. The normalized vector, Dimension1, is called Normalized Dimension1; the normalized vector, Dimension2, is called Normalized Dimension2. Note the distribution of the data points along the arc of the unit circle.

Homework 1.5.2.

Program a function that creates random points uniformly distributed across an arc that is specified in terms of its center and length (in radians). Demonstrate the effectiveness of your program with plots. Calculate the mean and variance of the random points, both for the original random points on the interval [0,1) and the scaled and shifted points. Compare the two sets of statistics.

How to make a high quality histogram

First, create one thousand or more random values as a one-dimensional vector, and call this list 'data.' This is enough data points to justify a twenty-bin histogram. To make a good looking histogram, use a sequence of two commands. First, group (or, equivalently, bin) your data using the function `[y,x] = hist( ),` a function which MatLab has already programmed for you. The brackets on the left are part of the function and give you the function's output.

```
>>[bin_values, bin_position] = hist(data, 20);
```

(Go ahead and take a look at the vectors `bin_values` and `bin_position`, there are only twenty elements in each vector.)

Now create a basic histogram using the `bar( )` command which plots the bin values as a bar graph.

```
>>bar(bin_values)
```

The command `bar ( )` can take either one or two arguments. With only a single argument you can plot the height of each bar. Unfortunately, the x-axis is usually not labeled correctly. If you use bar with two arguments, the x-axis goes first; that is, you first specify the bin positions. The second argument is the bin values (i.e., heights). Be careful, the number of elements in each argument must match. Here is an example using two arguments where we change the x-axis of the previous figure.

```
>> bar(0.025: 0.05: 0.975, bin_values)
```

Here we have offset, by 0.025, the left and right most bin, and we increment successive bins by 0.05, The increment and the range produces the required twenty bars, one for each value in the list `bin_values`.

The bar graph you have created is an absolute value histogram, or for short, histogram. The same data can be normalized by the total counts.

```
>>total_count = sum(bin_values)
            %should equal 1000.

>>bar(bin_position, bin_values/total_count)
```

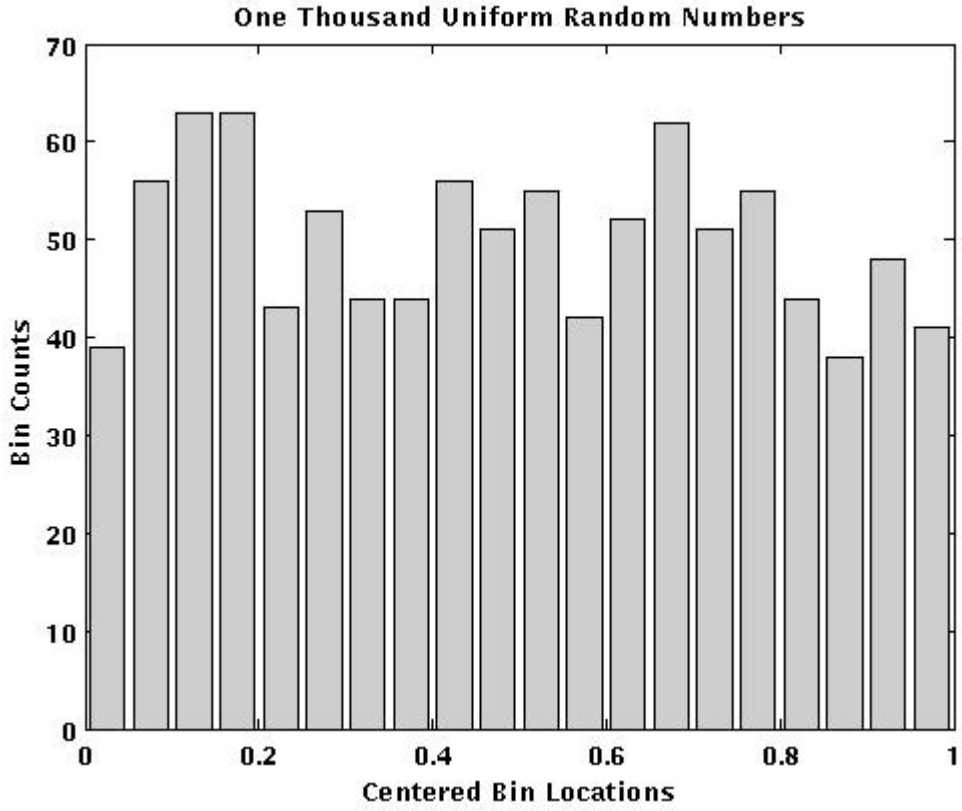


Fig. 1.7. A histogram with 20 bars. Note that this figure has a title, student name, date, axis labels, and calibrated intervals (figure file name: hist1000.jpg)

This is a relative frequency histogram of the same data. Because of this normalization (which implies that the bin sizes sum to one) and because these values are nonnegative, a relative frequency histogram qualifies as a probability distribution.

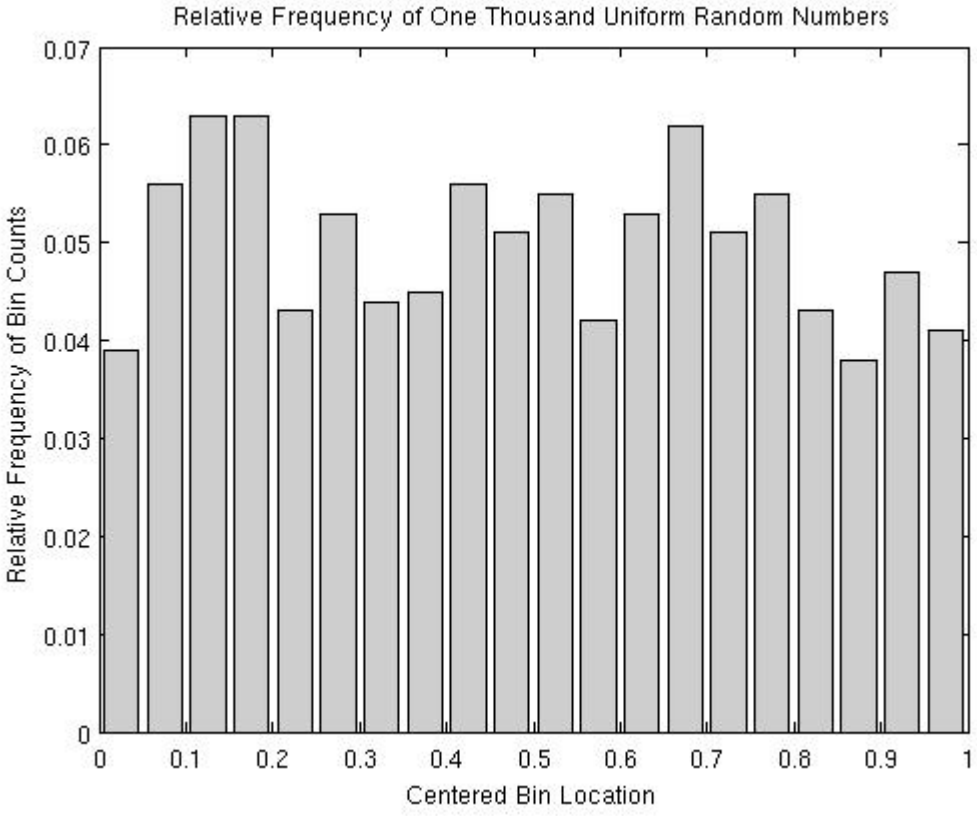


Fig. 1.8. A relative frequency histogram. These are the same data used to generate the previous relative frequency histogram (fig. 1.7). Here, however, the bin values are normalized so that they sum to one.

Fix the figure a bit with the graphical user interface, and then save the .m version of the figure. Date it; put your name on it; name the axes; put the correct value on the x-axis; and fix the colormap and boundary color. Save the configuration you like as a future template. See Appendix 1.3 and 1.4 for the two functions I made to use as templates.

Homework 1.5.3.

Compare the relative smoothness of the histograms and relative frequency plots when 10, 100, 1000, and 10,000 random numbers are used. How does bin width (i.e., number of bins) affect this smoothness? Be sure to consider absolute values of bin counts as well as relative values. Are your conclusions different?

The following section is intended to be used as an in-class exercise. Ask for help if you encounter difficulties.

Convergence of the Mean

Convergence of learning is a reoccurring issue in neuronal network theory. The homework assignment here illustrates an important insight into the convergence of an average.

If you have not already done so, program a function, I call mine stats to produce the mean, variance, and standard deviation from a vector of sample values.

```
function [mn, vr, sd] = stats(data_vector)

mn = mean(data_vector);

vr = var(data_vector);

sd = std(data_vector);
```

Now let's use this function to look at the convergence of each sample statistic as it moves toward its corresponding population statistic. First, let's compare sample sizes from two to thirty. Again, the for-loop will be useful.

```
>>for samp_size = 2:30

[mn(samp_size), vr(samp_size), sd(samp_size)] = …
            %use the ellipsis, …, and then Return key to break the
            %line and then continue a long line of code
stats(rand(1,samp_size))

end

>>figure

>>plot(mn)

>>figure

>>plot(vr)

>>figure

>>plot(sd)
```

If you have access to a printer, print your plots now or save them for later printing.

Now let's use lots of samples 5, 25, 625, …, 5^6.

```
>>for powr = 1:8

[mn(powr), vr(powr), sd(powr)] = stats(rand(1,5^powr))

end
```

Again plot each series to see how the sample statistics converge. That is,

```
>>figure

>>plot(mn)

>>figure

>>plot(vr)

>>figure

>>plot(sd)
```

Distribution of Averages

In an earlier exercise, you illustrated the convergence of sample statistics to population values. Theorems about such results are referred to as laws of large numbers.

Just like the issue of convergence, the rate of convergence is of interest in learning tasks. For sample statistics, we can get an appreciation of the rate at which the sample mean converges to the population mean by illustrating a result that falls within the rubric of a

central limit theorem. A central limit theorem describes the tendency for a sampling of sample averages to reproduce a Gaussian distribution with the chance of being far from the distributional mean going to zero at a particular rate. For the typical central limit theorem, this rate is the square root of the sample size.

You will need a function like the one below, in addition to your relative histogram plotting function.

```
function [vec_means, mnmn, vrmn, stdmn] = repeated_samples(samp_size)
            %this function creates 1000 means from a uniform random generator
            %set seed before using this function

vec_means = mean(rand(samp_size, 1000));
[mnmn, vrmn, stdmn] = stats(vec_means);
```

Homework 1.5.4.

For the following questions, be sure to keep the abscissa-scale constant.

1. Plot each of these statistics as a function the sample size. Write one to three sentences telling what these plotted statistics illustrate in terms of convergence.
2. Plot a histogram for each sample size. What do you see as sample size increases? How does bin width affect what you see?

Helpful hint: The uniform random number generator on the interval [0,1) has a mean of slightly less than 0.5, a variance of 1/12, and a standard deviation of sqrt(1/12).

Compare relative frequency histograms for means created from the average of 5^1, 5^2, 5^3, 5^4, 5^5, 5^6 data points. How do these histograms of sample averages differ from the previous histograms of raw samples.? What happens in these new histograms as the number of samples per mean increases? Print out three of these plots to illustrate what is happening.

Section 1.6. Functions Worth Saving

If you do all of the homework suggested in this chapter, at minimum, you should have created four functions:
A statistics function, a plotting function, a bar/histogram function, and a random seed function

Section 1.7. Possible Quiz Questions

If there is an in-class quiz next week, you should be able to answer the following questions:

- Explain in what sense the equals sign that you type into a MatLab command line is not like the equals sign in the mathematical expression 1+1=2.
- How would you use a 'for' loop in MatLab to update the current average as you iterate through the elements in a vector?
- How does a relative frequency histogram differ from an absolute frequency histogram?
- Why is it important that you seed the random number generator?
- How would you shift the range in which random numbers are generated in MatLab from the default [0,1) to [-1,-.5)?
- What does the following command produce:

```
>>size(rand(3,20))
```

- The idea of normalization is important. Most importantly, remember that normalization makes all vector lengths equal to one.
- Why does the following entry create a column vector?

```
>>columnvector2 = [2;4;6]''
columnvector2 =
2
4
6
```

Glossary Terms

{X}	the set X; the set of neuronal outputs is often an integer taken from the set {0,1}.
[0,1)	the real numbers from zero to one including zero but not including one.
average	something that is not extreme, but instead something in the middle; for our purposes, an *average* is the arithmetic mean
normalize	standardize the length of a vector. There are many ways to define length. We often use Euclidian length (square root of the sum of squares). Another common normalization for a vector made up of non-negative elements requires that the sum of the elements equals one.
random	The term *random* can be used in many ways: • Not deterministically describable or exactly knowable • Apparently not deterministically describable (even though, in fact, it is deterministically describable); often pseudorandom is the more precise term. • Maximally unpredictable or uncertain • Stochastic • There are others uses of this term, so be careful—if you use this word, be sure your reader knows what kind of random you mean.
random sample	A single output generated by a stochastic process is a random sample of the population (perhaps infinite) implied by the definition of the process.
relative frequency histogram	A normalized histogram in which the sum of all the bin values is equal to one.
vector	An ordered list or ordered set; a point in n-dimensional space; the line from the origin to such a point in n-dimensional space. (Note that all three definitions are equivalent, but for now it will be helpful to think in terms of the second definition.)

Appendix 1.1.

Little Things

```
%                       % Comment to end of line
;                       % Suppress output of line from printing to screen
~=                      % Does not equals sign
matrixA'                % "Tick", performs matrix transpose (i,j) → (j,i) for
                        each entry
matrixA(:,1)            % First column of matrixA
matrixA(2,:)            % Second row of matrixA

[0:0.1:5]               % Create 51 values, from 0 to 5 in 0.1 increments
1:N                     % Numbers 1-N.  Full colon has multiple uses.

help ____               % Get help with a matlab feature
clear var_name          % Clear variable
```

Useful Functions

```
randVec=rand(3,1)               % Random column vector, 3 entries
length(randVec)                 % Length of a vector, will return 3
round(randVec)                  % Round elements to nearest whole number
abs(randVec)                    % Absolute Value

mean(randVec(3,3))              % Mean of randVec columns, will output 1x3
matrix
var(randVec)                    % Variance (Population)
std(randVec)                    % Standard Deviation (Sample)

sqrt(num)                       % Square Root
sin(num)                        % Sine (radians)
cos(num)                        % Cosine (radians)
```

Figures

```
plot(data)                      % Plot data points in line
plot(data,'o')                  % Change data point markers, in this case to
"o"
axis('square')                  % Set to square axes
axis([xmin,xmax,ymin,ymax])     % Replace with values to set axes

[bin_values, bin_position]=hist(data,20); % Histogram, 20 bars
bar(bin_values)                           % Plot bin values as bar graphs
bar(0.05:0.05:1,bin_values)     % Assign 0.05,0.10,0.15 → 1 as x values for
bars
total_count = sum(bin_values)             % Total number of data points
bar(bin_position, bin_values/total_count) % Relative Frequncy Graph
```

User-Programmed Functions
The following should be saved as "stats.m" in a folder of other functions

```
function [samplemean,populationvariance]=stats(dataset)% First line of
stats.m

samplemean=mean(dataset);           % function will output sample mean value
populationvariance=var(dataset);    % function also outputs population
variance
```

Loops

```
b = 0                           % Set initial value before use
for a = 1:10,                   % For loop over variable a, values 1,2,3 → 10
      b = b + a;                % Add a to current value of b
end                             % End loop, now b=55 (1+2+3+4+5+6+7+8+9+10)
```

Creating Input Environments

```
rng(1204513528)                 % Use a specific seed for NEXT generated random
6*rand(1,5)-2                   % Create 5 random numbers on interval [-2,4)
x=rand(1,100)*pi/6              % Create 100 random numbers on interval
[0,pi/6)
plot(cos(x),sin(x))             % 100 random numbers on unit circle, angle [
                                0,pi/6)

innerprodx = x'*x               % Inner product of x (also dot prod., or
                                projection)
normalizedx = x/sqrt(innerprodx)    % Normalized x vector
normalizedx'*normalizedx        % Values of normalizedx should be on unit
circle
```

Custom Functions
These are just examples to get you going. Creating good functions makes it easier to set up more complicated networks. You'll also have fewer debugging headaches in the long run.

Basic Statistics (stats.m)
Calculate mean, variance, and standard deviation of a vector

```
function [mn, vr, sd] = stats(data_vector)
mn = mean(data_vector);
vr = var(data_vector);          % Calculated as if population variance
sd = std(data_vector);          % Calculated from sample variance (N-1)
```

Appendix 1.2. Programming technique and debugging[2]

Let's put things into perspective. Suppose you are building a house and a sloppy contractor will pour your foundation for $10,000 but it will probably be crooked. On the other hand, another, very careful foundation contractor who works a little slower want to charge you $20,000, but you will get a square and level foundation. To build the rest of the house on the square, level foundation cost $100,000 but to build the house properly on the crooked foundation will cost you $130,000. Now which contractor do you choose?

When it comes to programming, your time is your cost. For larger programs (and a large program is a relative thing), it is almost certain that you will spend more time debugging than the time spent on the initial programming. Thus, it makes sense to spend a little extra time writing the program so it is easy to debug.

Debugging is often the hardest part of completing a complex program. Small typos, misspelled variables, improper use of functions, and in fact just about anything can cause a well-conceived program to run incorrectly or not run at all. The trick is to find the errors quickly and get the program working with a minimum amount of rewriting. There are many ways to lessen the number of errors in your program code, and there are several ways to trace the errors once the code is written.

Here are some ways to avoid making errors when you are your writing code:

- *Write your code in small, reusable chunks*. Make your main program out of small functions. If you write your code piece-by-piece, it will be easier to locate errors and, once you have debugged these pieces, you can reuse them in future programs. For example, you can write a generic piece of code to calculate the internal excitation of a single neuron. Once it is debugged, it can be used for all networks that need such excitation calculated.
- *Write and comment your code for the long haul*. Write as if somebody else – someone who did not read your mind as you wrote the program – will modify your program. This will make it easier for you to modify the program. There are two mutually reinforcing paths to produce understandable and therefore easily reusable code.
- *Use descriptive variable names*. Choose variable names that describe what they represent. For example, if you have a variable that represents the number of neurons in your network, call it `number_of_neurons` instead of something generic, like `n`.
- *Write the code with lots of comments.* After each line, explain how any new variable is being used and what the calculation is going to do. This will make it much easier to debug later, and those who read your code will be able to understand what your program is doing. In the absence of such comments, far and away the hardest thing to do when you are trying to read code that you've written, or especially code written by someone else, is to figure out what the purpose of a certain process or variable is.
- *Reuse and modification of old code is much easier with error trapping routines built into the original code*. This helps prevent misuse when reusing.

[2] Thanks to Jason Meyers and Andrew Perez-Lopez for contributions to this appendix.

- *Do not write your program all at once and then try to run it*. Run the pieces as you write them. Try each function separately. Check the output at every step. Is it doing what you thought? Do the inputs and outputs make sense? Are they the size you would expect? It is much easier to debug small bits of code than an entire network. (OK, so this is the same as the first point, but it is important enough to write it twice!!)

- *Make your code legible*. Ideally, your code should as close to regular English as possible. For example, suppose you want to write a nested for loop to print each number in a synaptic weight matrix. The following code will work:

```
for i=1:nr
      for j=1:nc
            x(i,j)
      end
end
```

but who can understand what this code fragment is doing for a particular program? The following will be much more useful in the long run and leads to easier debugging.

```
            % go through each value in the weight matrix
            %and print it out
for rowIndex = 1:maxnumberOfRows
      for columnIndex = 1:maxnumberOfColumns
            weightMatrix(rowIndex,columnIndex)
      end
end
```

If the first set of code were positioned in the middle of a page, perhaps nested in another loop, it would be extremely confusing, while the second set of code would be clear. Using longer variable names can be taxing on the fingers, but with the copy, cut, and paste functions the editor provides, it's not so bad, and it makes much less work for your brain in the long run.

Make your code more generic to make reuse easier. If you are writing a program that will be reused with different parameters (such as numbers of neurons or fraction of connectivity), write the original program with variables like `number_of_neurons` and `number_of_synapses`, in place of what are called 'magic numbers,' like constants 256 or 0.05. If you do this, then you can just copy and paste whole blocks of code, changing only those lines that define the number of neurons and the number of synapses. But be careful. Many errors come from copying and pasting lines and then missing a few lines when making the changes to a new network.

- *Use the* `clear` *command*. If you frequently use some variables (e.g., `input(x)` or `output(y)`) the current implementation of the network can fail if a previous network had a different size input or output, or if the previous network left lots of data still in the matrices. To eliminate this problem, just include `clear`

`variable_name` in your code. This command will erase all data in the stated variables so that the next running of the network can start fresh.

Along with this, it is good to include a `clear all` statement at the very beginning of a long program, so that all of the variables are cleared off, the memory is wiped clean, and the program has a fresh slate. If you don't have to carry any variables over between network models, you can run `clear all` between each simulation of the network.

In the absence of proper `clear` statements, outputs from previous runs can mess up current runs. Here's a suggestion that may just kill two birds with one stone. At the beginning of your code, write a `clear` statement for each variable you are using and also state what the variable does. For example, the code used above might now look like this:

```
clear rowIndex
            %loop variable to specify the current row
clear numberOfRows
            %the number of rows in my weight matrix
clear columnIndex
            %loop variable to specify the current column
clear numberOfColumns
            %the number of columns in my weight matrix
clear weightMatrix
            %a matrix containing all the weights for my
            %network
            %……
            %the rest of your code goes in between the part above & the
            loop below
            %……
for rowIndex = 1:numberOfRows
      for columnIndex = 1:numberOfColumns
            weightMatrix(rowIndex,columnIndex)
      end
end
```

This method makes sure that all your variables are cleared out, and it provides a central location where you can find out what all your variables mean. You might be thinking, "I know what my own variables mean," but when the code gets long, and you come back to code written weeks before, it gets tricky!

Following our prescriptions will make your code more readable to yourself and any others who may feel inclined to read it.

- *For small debugging runs, leave out the end of line semicolons*. When trying some new code for the first time, it's really useful to see how the data are changing. If you leave out the ; at the end of each line, MatLab can display the variables as they are changing. This is usually counter productive for 100x100 matrices, but it is really useful for seeing how counters, excitations, etc. are changing in small simulations.

- *Test functions using extreme values*. If a variable can vary between 0 and 1, test at the extreme of 0 and then the other extreme of 1. Then put your functions together and rerun the same test values.

  All too common mistakes:
  - Typing in `function(column,row)` rather than `function(row,column).`
  - Forgetting the final parentheses in embedded commands. For example,

```
test=input(excitation(counter),output(neurons(counter)
)
```

    needs one more parenthesis at the end.
  - Leaving data in variables from previous runs of a program. For example, if you use the variable `output(x)` for a network with 6 outputs and then for a network with only 4 outputs, the variable will still have 2 outputs from the old network. This will mess up later calculations. Include a `clear output` line in the beginning of the program.
  - Trying to assign something of the wrong size to a variable. If a network is giving a column vector output, make sure that the variable assigned to that output can accept a column vector. To help with this problem, write all of your programs so that they use a column vector for the weights of a single neuron and a column vector for the input at one instant in time. Conventionally time goes from left to right so use time to index columns of successive inputs and just stick with it!

Above all, remember to keep things simple. When first testing ideas, do not use lots of deeply embedded loops. Write simply, and you can debug simply. Once you know that your code is working properly, expand it out into the loops through each neuron, variable, synapse, etc. Please take this suggestion to heart. Use all of our suggestions to write code that is easily understandable, easy to debug, and elegant. Lots of people can program but a good programmer is a rare bird indeed. If you faithfully follow the above prescriptions, you have taken a big step towards joining this exclusive group. Work hard!

Appendix 1.2. skeletonplot.m

```
function skeletonplot(xdata,ydata,newtitle,newlabelx,newlabely)
% xdata: column vector of xaxis data
% ydata: column vector of yaxis data; if no ydata use length(xdata) as ydata
input
% the default MatLab axis is aesthetically inferior because it plots points
% on the figure edges. Here we will force the axes beyond all data points.

% error trapping:
[xrows,xcols] = size(xdata);
[yrows,ycols] = size(ydata);
if(xcols ~= 1 || ycols ~= 1)
 disp('NOT COLUMN VECTORS');
end
if(yrows ~= xrows)
    disp('Data do not agree. They are not the same length');
    disp(['x is ',num2str(xrows),'. y is ',num2str(yrows),'.']);
end

xrange = max(xdata) - min(xdata); %max( ) and min( ) are MatLab functions.
yrange = max(ydata) - min(ydata);
todaysdate = date; %date is MatLab command

% Create figure
figure1 = figure('color',[1,1,1]);

%replace MatLab?s gray border with white
colormap gray

% Create axes
axes1 = axes('Parent',figure1);
axis(axes1, [min([xdata-0.05;0]) , max([xdata+0.05;1]), ...
             min([ydata-0.05;0]) , max([ydata+0.05;1])]);
%forcing the axis larger

title(axes1,newtitle);
xlabel(axes1,newlabelx);
ylabel(axes1,newlabely);
box(axes1,'on');
hold(axes1,'all');

% Create plot
plot1 = plot(xdata,ydata,'x');

% Create textbox
annotation1 = annotation(figure1,...
        'textbox'  ,...
        'Position' ,[0.6643 0.9405 0.308 0.07024],...
        'LineStyle','none',...
        'String'   ,{['Not John Smith ' todaysdate]},...
        'FitHeightToText','on');
%use your name
```

Appendix 1.3. histbar.m

```
function histbar(range_min,range_max,newdata,numbins,newtitle)

% histbar() will produce a histogram with number of bins
%   equal to numbin and using newdata
% also prints current date
% range_min and range_max are the left and right edge of the smallest and
% largest bin on the figure
% newdata is a vector
% numbins is an integer greater than one
% newtitle is a string
% example call from command line:
%   >> histbar(-.005,1.005,data,20,'The Title')
% where data is a vector and 20 specifies the number of bins

%% Create figure and color the surround white
figure1 = figure('color',[1,1,1]);  % both the colormap and the color [1,1,1]
colormap gray                       % must be specified

%% Create axes and labels

axes1 = axes(...
  'FontName','lucida',...
  'FontWeight','bold',...
              'Parent',figure1);
title(axes1,newtitle);
xlabel(axes1,'Bin Position');
ylabel(axes1,'Bin Count');
box(axes1,'on');
hold(axes1,'all');

%% Create histogram and get some facts about the bins

  [all_bin_values,jtt] = hist(newdata,numbins);
  largest_bin_value = max(all_bin_values);

  bar(jtt,all_bin_values)

% move vertical edges of fig away from end bars

fullrange=range_max-range_min;         % find out where the smallest and
leftvert=range_min-.02*fullrange;      % possible values could be and
rightvert=range_max+.02*fullrange;     % move 2 percent of the range away

% adjust the top of the box to be 10 percent above largest bin value

axis(axes1,...
    [leftvert,rightvert, 0, 1.1 * largest_bin_value]);

%% Create textbox of name and date
annotation1 = annotation(...
  figure1,'textbox',...
  'Position',[0.6643 0.9405 0.308 0.07024],...
  'LineStyle','none',...
```

```
  'String',{['Not Joy Arcangeli ' date]},...   %change to your name, not
Joy's
  'FitHeightToText','on');

%% shade the bars using the current colormap

caxis([0.2,1.1])
```

Appendix 1.4. cumulbar.m

```
function cumulbar(range_min,range_max,newdata,numbins,newtitle)

%cumulative distribution derived from

% histbardist() will produce a prob. distrib. with number of bins
%   equal to numbin and using newdata
% also prints current date
% range_min and range_max are the left and right edge of the smallest and
% largest bin on the figure
% newdata is a vector
% numbins is an integer greater than one
% newtitle is a string
% example call from command line:
%   >> histbar(-.005,1.005,data,20,'The Title')
% where data is a vector and 20 specifies the number of bins

totcount = length(newdata);

%% Create figure and color the surround white
figure1 = figure('color',[1,1,1]);  % both the colormap and the color [1,1,1]
colormap gray                       % must be specified

%% Create axes and labels

axes1 = axes(...
 'FontName','lucida',...
 'FontWeight','bold',...
              'Parent',figure1);
title(axes1,newtitle);
xlabel(axes1,'Bin Position');
ylabel(axes1,'Cumulative Relative Frequency');
box(axes1,'on');
hold(axes1,'all');

%% Create histogram and get some facts about the bins

  [all_bin_values,jtt] = hist(newdata,numbins);
  largest_bin_value = max(all_bin_values);

%% cumsum(1)=all_bin_values(1)/totcount;

  for index= 2 : length(all_bin_values)
      cumsum(index) = cumsum(index -1) + all_bin_values(index)/totcount;
  end

bar(jtt,cumsum,'BarWidth',1);

% move vertical edges of fig away from end bars

fullrange=range_max-range_min;            % find out where the smallest and
leftvert=range_min-.02*fullrange;         % possible values could be and
rightvert=range_max+.02*fullrange;        % move 2% of the range away

% adjust the top of the box to be 10% above largest bin value
```

```
axis(axes1,...
    [leftvert,rightvert, 0, 1.1  ]);

%% Create textbox of name and date
annotation1 = annotation(...
  figure1,'textbox',...
  'Position',[0.6643 0.9405 0.308 0.07024],...
  'LineStyle','none',...
  'String',{['Not Joy Arcangeli ' date]},...   %change to your name, not
Joy's
  'FitHeightToText','on');

%% shade the bars using the current colormap

caxis([0.2,1.1])
```