

# Appendix A

## Programming Primer and Reference

You may want to start with Matlab tutorial:

[http://www.mathworks.com/academia/student\\_center/tutorials/launchpad.html](http://www.mathworks.com/academia/student_center/tutorials/launchpad.html)

### Summary of MATLAB learned in this chapter

#### *Assignment and variables:*

equal sign (=): sets a variable equal to a value

semicolon (;): output display suppression:

#### *scalars, vectors, matrices:*

A scalar variable holds a single value. Created by assigning a value to a variable name, e.g.  $x = 5$

A vector holds more than one value along a single dimension. A matrix is similar to a vector, except that it has two dimensions referred to as rows and columns.

Vectors and matrices can be created by:

**Concatenation:**  $x = [1, 1, 2, 3, 5, 8, 13]$  for 1x8 vector or  $y = [1, 2, 3; 4, 5, 6]$  for 2x3 matrix.

**Data creation functions:** zeros(), ones(), rand(), randn(), linspace(), logspace(), : (colon)

**Implicitly:**  $x(2) = 5$  creates  $x$  with length 2, and sets the second value to 5

Elements of a vector are accessed by indexing using parentheses;  $x(1)$  thus refers to the first element of  $x$ ;  $y(2,3)$  is the element of  $y$  in the second row and third column

#### *operators:*

**relational:**  $<$ ,  $>$ ,  $<=$ ,  $>=$ ,  $=$ , and  $\sim$  compare 2 values and return TRUE (1) or FALSE (0)

**logical:**  $\sim$  ("NOT"),  $\&$  ("AND"),  $|$  ("OR"), and  $\text{xor}(\text{value1}, \text{value2})$  compare logical values for vectors;  $\&\&$  and  $||$  are short-circuit logical operations

**arithmetic:**  $+$ ,  $-$ ,  $*$ ,  $/$ , and  $^$  perform basic arithmetic operations,  $\text{rem}$  gives the integer remainder of division

#### *general functions*

**sum(x)** adds all elements of a matrix  $x$  column-wise

**mean(x)** computes the mean of elements in  $x$

**std(x)** computes the standard deviation of elements in  $x$

**plot(x,y)** creates a plot of  $y=f(x)$

**hist(x)** histogram of vector or matrix  $x$

**find(condition)** find indices of matrix elements that satisfy condition

**help('function\_name')** or **help function\_name** provides help on the function

**lookfor('topic')** might work if you don't know the exact function name

### *data creation*

**randsample(data, size, replace)** takes samples from “data”

**[]** concatenates all of its arguments into a single vector

**repmat(a, x, y)** replicates a scalar or matrix a in x-rows and y-columns

**:** creates a series of values incremented by 1 or specified size

**linspace(from, to, howmany)** is a more flexible version of the above

**logspace(from, to, howmany)** creates logarithmic scale

**randn(x,y)** creates x by y normally distributed random numbers

**rand(x,y)** creates uniform random numbers

**zeros(x,y)** creates x by y matrix of zeros

**ones(x,y)** creates x by y matrix of ones

### *control flow functions*

**for()** repeats one or more operations a specified number of times

**if() ... else()** performs one or more operations contingent on some condition

**while()** runs until some condition is met

### *data import and export*

**load()** reads a Matlab file or tab delimited text file containing data only

**dlmread()** does the same for any delimiter

**xlsread()** reads data from an excel sheet, including column names

**save()** writes a Matlab or –ascii file

**dlmwrite()** writes delimited formatted data

**xlswrite()** writes data into an excel file, possibly including column names

## **The meat and potatoes: assignment and variables**

A *variable* can be thought of as a container, referred to by the *variable name* that can hold any numeric *value*.

= (the equals sign) assigns a value (on the right) to a variable (on the left).

For example, after

```
x = 5
```

the variable x contains the value 5.

Importantly, if the variable already exists, it will be overwritten with the new value.

Seriously, if “x” contains all the data that the Air Force uses to drop bombs, and you type

```
x = 4
```

then too bad for the Air Force – x now contains the value 4 – no recovery, no exceptions, all bombs go to “4” (wherever that is).

One seeming exception to the “= destroys things” rule – and a very useful one – is as follows. Programming languages, unlike English, are read right-to-left by default. Thus

```
x = x + 1
```

will take the existing value of  $x$ , add one to it, and store the result back into  $x$ . The old value of  $x$  is gone forever, but it was used in an intermediate step to calculate the new value of  $x$ .

I think of variables like a mailbox. I don't have to go around to various places to collect my daily mail. Instead "today's mail" (the value) appears in the mailbox "Cormack" (the variable name). So all I have to do each day is query the variable "Cormack", which makes checking my mail easy.

## ***semicolon***

The semicolon is used at the end of an input line to suppress output to the console.

Typing

```
x = 1
```

causes MATLAB to print the values in  $x$  to the command window, whereas

```
x = 1;
```

creates  $x$  silently. This is valuable when working with large sets of numbers.

You will see the tremendous value in using variables when we start writing longer programs, especially those with loops.

## **Scalars, vectors and matrices**

### ***regular variables (a.k.a. scalars)***

Scalars are variables that hold a single value.

$x = 5$  is an example of creating a scalar variable.

### ***vectors***

A *vector* consists of multiple values, all referred to by the same variable name.

The individual values are referred to by the variable name and an integer *index*.

For example, after

```
x = [1, 2, 3, 5, 7]
```

(note those are square brackets - see "data creation" below),  $x$  contains the first five prime numbers.

Elements are accessed using regular parentheses, so the 4th value is obtained by  $x(4)$ , etc. The number inside the brackets is the *index* into the vector  $x$ .

You can also refer to ranges of values – so

```
x(2:4)
```

are the middle three values, and, as I mentioned above, “2:4” can be read as “2 through 4, inclusive”.

You can think of a vector as a row of mailboxes, like we have on rural Texas roads (where all the mailboxes for the houses down a particular turn-off are in a single row at the turn-off). So if there is a row of 5 mailboxes at the turnoff to Guntotin Rd., then we could name the entire row “guntotin” and guntotin[3] would then refer to the middle mailbox.

In addition to the above use of square brackets, there are some other ways to create vectors. One is to use the zeros() function:

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

which creates a vector elements long (5 rows and one column) and fills it with zeroes. The ones() function works the same, except it creates a vector filled with ones.

### ***matrices and arrays***

*Matrices* are the same as vectors except they have both rows and columns, i.e. they are 2 dimensional instead of 1 dimensional. MATLAB (=Matrix laboratory) can perform various matrix operations fast and easy, so you want to use them a lot. Scalar and vectors in MATLAB can be thought of as a special case of a matrix: scalars are matrices of size 1x1, vectors are matrices that have size of one dimension equal to one.

You can type in small arrays using square brackets as above, inserting a semicolon each time you want to start a new row:

```
y = [1, 2, 3; 5, 7, 11]
```

The variable y is now a matrix with 2 rows and 3 columns.

```
y =  
    1     2     3  
    5     7    11
```

Individual values in a matrix are set or obtained just as with vectors, except both a row index and a column index are needed. Thus

```
y(2, 3) = 2.718282
```

Sets the element at the second row and third column to e (roughly), and

```
my_e = y(2, 3)
```

grabs this value and places it in the variable my\_e.

You can think of matrix as being like the faculty or student mailboxes in a departmental mailroom. If the name “faculty” referred to the faculty mailboxes, then faculty[4, 5] would be my mailbox, since it is fourth from the top and fifth from the left.

You can extract chunks of data from a matrix using the colon (which, remember, can be read as “through.”). So

```
y(1:2, 2:3)
```

will give you the lower left four elements of  $y$ . Used by itself, the colon translates to “all rows” or “all columns”. For example:

```
y(:,3)
```

returns the entire third column of  $y$ .

If you are familiar with a spreadsheet program such as Microsoft Excel, then an analogy might be useful. A scalar variable is exactly like a cell in Excel, a vector is like a column (or row), and a matrix is like a single worksheet. In fact, because of correspondence, it is easy to read data into MATLAB from an excel spreadsheet and vice versa. This is covered below in “input and output.”

It is generally easier to create matrices with MATLAB commands designed for this purpose:

```
x = zeros(5, 3)
```

creates a matrix,  $x$ , with 5 rows and 3 columns filled with zeros. Note that this is the same thing we did to create a vector above, we simply changed the second number to tell MATLAB to create three columns rather than one. Note also that the new  $x$  has replaced or overwritten the one created above.

In addition to `zeros()`, the commands `ones()`, `rand()`, and `randn()` are available. The latter two create a matrices containing either uniform or Gaussian random numbers.

```
myrand = 10 + 3.*randn(1000, 1);
```

creates a variable `myrand` containing 1000 random numbers with mean 10 and standard deviation 3. Note that MATLAB automatically applies the multiplication and addition to every member of the matrix created by `randn()`, which is handy.

Matrices have 2 dimensions by definition, and are a special case of *arrays*, which can have any number of dimensions. If you are new to programming and you try to work with arrays with more than 3 dimensions, your brain will probably explode.

Another option how to introduce a variable for later use is to create an empty matrix. Matlab assumes that all variables represent two dimensional matrices. Therefore

```
a = []
```

will generate a variable  $a$  of size  $0 \times 0$ .

Creating an empty variable may be useful when we do not know the size of the variable up front and let the variable to grow as needed. In all other cases, we should initialize the variable using `zeros(x,y)` command as that will allow Matlab to allocate appropriate amount of memory and yields faster code execution.

## ***datasets***

The MATLAB statistics toolbox provides a type of matrix called a dataset. It is structured like the data are in almost all of the popular GUI-driven statistics packages, with named variables in the rows and observations in the columns.

## Operators

Operators usually take two inputs and produce one output. Plus and minus are probably the first operators you learned at elementary school.

### **Relational (comparison) operators**

*Relational (comparison) operators* compare two values (hence the name...).

Importantly, they do not destroy or overwrite anything, unless combined with an *assignment* operator. The comparison operators are:

<, >, <=, >=, ==, and ~=

These should be fairly obvious – “less than?”, “greater than?”, “less than or equal to?” etc. Note that the *double equals sign* is not a typo – “is equal to?” is a double equals sign; this is what distinguishes it from the *assignment operator*.

( ~ means “not” in MATLAB, so the last one is “not equal to?”)

### **Logical**

Logical operators combine logical values using AND, OR, NOT, and exclusive OR (XOR) (if you’ve taken logic out of a philosophy dept., congratulations! – it was not a completely wasted semester).

They are very useful when we need to determine if a value is within or outside a particular interval (as we do in hypothesis testing or to detect outliers).

The operators are:

~ (NOT), & (AND), | (OR), and xor(value1, value2) (XOR).

Here is an example.

```
x = randn(100, 1)
```

This puts 100 normally distributed random numbers (with mean 0 and standard deviation 1) in the 100x1 column vector x.

```
y = (x < -2) | (x > 2)
```

This finds each case in which the value is more than 2 standard deviations from the mean. In English, it reads “Set y to TRUE for each value of x that is *less than -2 OR greater than 2*. The parenthesis ensure that the operations are done in the right order (evaluate the *less than* and the *greater than*, then evaluate the *OR*), just like in high school algebra. The resulting variable y is a logical vector of the same size as x, containing 0 (false) wherever x is between -2 and 2 (i.e. within 2 standard deviations from the mean) and 1 (true) wherever the value of x is outside this interval.

If you need to combine single logical values (i.e. scalars as opposed to vectors), use double & (“&&”) or double | (“||”). && and || are called short-circuit operators, because

the second expression is evaluated only if necessary. For example, if you have only one value in a variable (i.e. t-value from a comparison of two groups), use

```
(t < - crit_value) || (t > crit_value)
```

to find if your t-value is significant. If the first expression is true, there is no need to evaluate the second expression, true OR anything is always true. This is useful if you want to avoid generating a warning when the second expression is not defined for some cases:

```
x = length(b)~=0 && mean(b)>100
```

If b is empty, second expression (“mean(b)>100”) is not evaluated and warning is “Divide by zero” is avoided.

## **Arithmetic**

+, -, \*, /, and ^ should be obvious for single values (also called scalars).

rem(x,y) gives the remainder of division x/y.

## **Colon**

The : (colon) isn’t an operator in the strict sense, but I’ll introduce it now anyway. It stands for “through” in that

```
1:5
```

generates the numbers 1 through 5.

More usefully,

```
x = 1:5
```

generates the integers 1 through 5 and stores them in the *vector* x.

I include it (the : ) here because you can think of it as an operator that keeps adding (a true operation) 1 to the first number until it gets to the second number.

You can also specify a different number to use as the increment. For example:

```
x = 1:0.5:5
```

generates the sequence

```
1.0, 1.5, 2.0 ... 5.0
```

and stores it in the variable x.

## **Functions**

*Functions* are fancy, black-box versions of operators. They take zero or more inputs, called *arguments*, and produce one or more outputs, called *return values*.

In MATLAB, sum() is a function version of + that, by default, operates on columns.

```
sum([2;2])
```

will produce the same output as

```
2+2.
```

Because “sum” and other functions operate on columns by default, they can be used as a fast method to generate summary statistics for various data sets. Let’s say that you have a data set with variables in columns and observations in rows. I have accuracy in the first column and reaction time in seconds in the second column, for four different participants (rows):

```
mydata = [.75 1.20; .88 1.310; .59 .89; .72 1.52]
```

```
mydata =  
    0.7500    1.2000  
    0.8800    1.3100  
    0.5900    0.8900  
    0.7200    1.5200
```

Now, to generate summary statistics for my data, I simply call “mean” and “std” function on the whole data set and Matlab knows what to do:

```
>> mean(mydata)  
  
ans =  
    0.7350    1.2300  
  
>> std(mydata)  
  
ans =  
    0.1190    0.2627
```

Although it is possible to organize your data in another way, using rows for individual observations and columns for different variables will save you a lot of headache. Column-oriented analysis of multivariate statistical data is default in both Matlab and other data analysis packages, like SPSS.

Functions are powerful and flexible, and they are nice in that we usually don’t care how they work internally. For example, let us put some data in a variable, y, and plot it:

```
somedata = randn(20,1)  
plot(somedata)
```

Look at the x axis – it seems that the function `plot()` automatically plotted our data as “y” against and “x” consisting of the integers 1 to 20. That won’t always be what we want, but it is a handy default.

Now lets create 20 more numbers:

```
moredata = logspace(1,2, 20)
```

This function creates 20 logarithmically spaced numbers between  $10^1$  (10) and  $10^2$  (100). Now let’s plot again:

```
plot(moredata, somedata)
```

This time, `plot()` made a graph of the second variable, `somedata`, as a function of the first variable, `moredata`. How does it “know” how to behave when it is given one vs. two input arguments, or how does it “know” how to deal with one input that is a row vector (20x1) and another that is a column vector (10x1)? **We don’t care!** All we care about is that it is a black box into which we can throw sensible input, and receive in return sensible output.

No matter what you want to do with MATLAB in the course, there is probably a function to do it.

Here’s how easy it is to do a bootstrap replication:

```
mydata = 50 + 10.*randn (100, 1);  
bootsamp = randsample(mydata, 100, true);
```

Here, `randn()` creates a data set of 100 normally distributed numbers that we scaled to a mean of 50 and a standard deviation of 10 (let’s say we want to simulate a sample of scores on a Wechsler scale). The function `randsample()` then grabs 100 values from `mydata` with replacement, creating one simulated data set.

Wow, `randsample ()` seems like it is doing a pretty hard and complicated job for us – how does it work?? Again, **you don’t** (have to) **care!** Think of it like your car; as long as it goes when you press the skinny thing on the right and stops when you press the fat thing on the left, you (or most people, at least) don’t care how it works. Cars can be driven by people who aren’t mechanics, and that makes cars useful for all of us.

## **Control flow**

“*Control flow*” refers to the programmers (i.e. your) ability to have your program do different things depending on the value of one or more variables. Say, for example, you collected a data set with  $n=20$  and computed the mean. Now you want to compute a *sampling distribution* for that mean using 200 bootstrap replications of the experiment. What you need to do (in English) is:

- collect - using `randsample()` - a bootstrap replication
- compute the mean
- store it

-keep doing the above three steps until 200 replications have been done

The key here is that you want your program to do something different when you have collected fewer than 200 replications – in this case “keep going” – than you do when you have finished collecting your 200 replications – in this case “stop”.

There are three basic “control structures” in MATLAB that should allow you to do everything you need to do for basic Monte Carlo and Bootstrap analysis. In fact, the first two we’ll discuss should probably suffice.

## **for()**

The for() statement is what is known as a “loop”, because it allows you to loop through (repeat) a series of steps for some specified number of times. For example, we might want to compute a bootstrapped sampling distribution of the mean using 200 bootstrapped replications of our experiment.

Let’s create some data to play with:

```
data = randn(100);
```

Now we have 100 measurements of IQ scores stored in the vector “data”. Next, we’ll make a vector to store each sample mean as we go through the loop:

```
mybootmeans = zeros(200, 1);
```

Here, we created a variable that will store all our 200 bootstrapped sample means. We don’t have to initialize (pre-generate) variables in Matlab, but it does speed up the code, especially when creating larger vectors or matrices. Initially, each element is set to zero.

Now we can use for() to do the bootstrapping:

```
for i = 1:200
    onesample = randsample(data, 100, true);
    mybootmeans(i) = mean(onesample);
end
```

The first line tells MATLAB that we are going to go through whatever is inside the braces 200 times, incrementing the value of the variable i each time. This is really convenient, because we can use i as an index into the vector mybootmeans in order to store each of our 200 bootstrapped means.

To plot the sampling distribution, we can just type

```
hist(mybootmeans)
```

Step back and think about this – we have barely learned to program, and we can already compute a bootstrapped sampling distribution of the mean! And all we’ve had to learn are the functions for(), sample(), mean() – whose function should be obvious – and randn() to create some fake, normally distributed data, and hist() to look at a histogram of the results! Wee haa!

## ***if()***

The `if()` statement behaves just like the word “if” behaves in our language. For example, you might think “*If* I have finished my stats assignment by 10 p.m., I’ll go out.” A fancier version is “*If* I have finished my stats by 10 p.m., I’ll go out, *else* I’ll stay home and get depressed”.

You will use `if()`, usually in connection with relational operators, when you need to evaluate a certain condition, for example when testing a hypothesis.

```
if mymean>criticalvalue
    h0 = false;
else
    h0 = true;
end
```

In Matlab, `if()` is often implicit in other commands, such as `while()`, `find()` or when using *logical indexing*. We will talk about these later.

## ***while()***

The `while()` control statement repeatedly executes a command or commands while some condition is true. Let’s say that you would like to generate a random sample of 40 IQ scores, but it is important to you that your sample mean and sample standard deviation will be approximately that of population, i.e. 100 and 15. Maybe you need your sample mean to be within 5 points from the population mean and your sample standard deviation within 2 points from the population standard deviation. You can automatically generate a sample like that by asking Matlab to keep generating samples until it finds a sample that is suitable:

```
% generate the initial IQ sample
iqsample = round(15*randn(40,1) + 100);
% specify tolerable difference between your sample statistics and
population statistics
okmeandiff = 5;
okstddiff = 2;
% test whether your sample statistics are within tolerable limits
% and keep generating new samples if not
while abs(mean(iqsample)-100)>okmeandiff || abs(std(iqsample)-
15)>okstddiff
    iqsample = round(15*randn(40,1)+100);
end
```

After the code finishes, the required sample will be in the `iqsample` variable.

## **find() and logical indexing**

### ***find()***

`find()` enables you to find indices of all observations that satisfy a condition and is a computationally highly efficient (fast) shortcut for a combination of `for()` and `if()`

```
% median reaction times in a complex cognitive task
myrts = [2.17, 1.31, 1.44, .98, 1.26, .85, .23, 1.57];
% it is thought that reaction time that involves decision making
% (i.e. different response to
% different kind of events) cannot be shorter than 300-400 ms
% because of
% the number of synapses involved.
tooshort = find(myrts < .3)

tooshort =
     7
```

I should have a closer look at data from participant number 7!

### ***logical indexing***

logical indexing is a great feature of Matlab that makes various common computations fast and elegant. It allows you to perform an operation on a subset of your data that satisfy some condition, without having to use `for()`, `if()` or even `find()`.

Let's say you measured spatial awareness in one class of high school students. Gender (1=female, 2=male) is coded in a variable "gender", spatial awareness score in variable "spaw". Beside population mean, you may wish to compute mean separately for males and females. In Matlab, you can do this easily in one line of code:

```
meanfemalescore = mean(spaw(gender == 1));
meanmalescore = mean(spaw(gender == 2));
```

Read "compute mean for those values of vector "spaw" where vector "gender" has a value of 1".

## Data creation

“Data creation” is somewhat of a misnomer. Generally, we use data creation functions to generate a convenient x-axis (or axes). An important exception is Monte Carlo analysis, in which we use data creation functions to make simulated data sets in order to compute a sampling distribution. We have already encountered one important data creation tool, which is the `:` operator, and is used to generate a sequence of integer values, as in:

```
x = 1:20
```

Now the variable (vector) `x` contains the integers 1 through 20. You can also specify the increment value in the middle:

```
x = 1:.5:20
```

which gives us 39 numbers running from 1 to 20 in steps of 0.5

Another option is the `linspace()` command, which allows you to specify the number of points you want, rather than the increment. So, for example, if we want 40 points between 1 and 20 (rather than an increment of exactly 0.5), we can enter:

```
x = linspace(1, 20, 40);
```

A very similar command, `logspace()`, generates logarithmically spaced numbers:

```
x = linspace(1, 20, 40);
```

We have already encountered a very simple data creation function, `[]`, which is the way to type in data using square brackets. So:

```
x = [0, 1, 1, 2, 3, 5, 8, 13]
```

concatenates (glues together) the 8 values – the first 8 numbers in the Fibonacci series – into the single vector `x`.

You can also concatenate vectors:

```
a = [1, 2]
```

```
b = [3, 4]
```

```
x = [a, b]
```

will set `x` to `[1, 2, 3, 4]`

This is useful for entering very small data sets. For larger data sets, the functions in “input and output” (below) are much more useful.

You can also repeat values, or whole matrices, using `repmat()` (for “replicate matrix”). So:

```
x = repmat(2.72, 20, 1)
```

is the same as `x = 2.72*ones(20,1)` and creates a vector of 20 rough approximations to  $e$ .

Alternatively,

```
x = repmat([1, 2, 3], 20, 3)
```

creates a 20 x 9 matrix containing 20 rows of the sequence 1, 2, 3, 1, 2, 3, 1, 2, 3.

For Monte Carlo analysis, MATLAB provides a wonderful corpus of random number generators, from which we can draw random samples from various common distributions. If we want 100 (say) random numbers from a standard normal distribution, all we have to do is enter:

```
x = randn(100,1)
```

and – poof – we have our sample. If we need 100 normally distributed numbers representing standard I.Q. scores, we can enter:

```
x = 100 + 15.*randn(100,1)
```

which generates 100 normally distributed random numbers with a mean of 100 and a standard deviation of 15.

Why be normal? For random numbers from a uniform distribution:

```
x = rand(100,1)
```

which generates a uniform (flat) distribution of numbers between 0 and 1. Or, if we are studying something like whether subjects are above or below some baseline, the binomial distribution might be helpful:

```
x = binornd(50, .1, 100, 1)
```

Which simulates how many times “heads” came up in 50 flips of a very unfair coin – a coin in which “heads” is expected to come up only 10% if the time – across 100 “experiments”.

Some handy distributions are described below. For a complete list of the distributions that MATLAB can generate, see “supported distributions” in MATLAB’s help.

## **Statistical distributions**

Statistics toolbox provides a number of build-in distributions for which you can generate random samples (‘rnd’), probability (‘pdf’) and cumulative probability (‘cdf’) density functions, inverse functions (‘inv’) or even fit these functions into your data (‘fit’).

To call the specific function for your distribution of choice, simply put together Matlab name of the distribution (e.g. “logn” for log-normal distribution or “bino” for binomial distribution) with one of the generic function name suffix listed above. So “lognrnd(0,1,5)” will give you 5 random numbers from a lognormal distribution with mean = 0 and sd = 1, and “binocdf(1,10,1/6)” will give you probability of maximum of one success (e.g. throwing 6 on a dice) out of 10 trials using an unbiased dice.

What are the most useful distributions depends largely on the field of study, but probably include the following:

**beta:** beta distribution; e.g. for Bayesian statistics

**gam:** gamma distribution; e.g. canonical hemodynamic response function in fMRI  
**exp:** exponential distribution; e.g. radioactive decay  
**norm:** normal distribution; sum of many independent events: “if it is complicated, it is Gaussian”  
**logn:** lognormal distribution; product of many independent events  
**wbl:** weibull distribution; e.g. in psychophysics  
**chi2:** chi-square distribution; independence of factors, goodness of fitting  
**f:** F distribution; e.g. analysis of variance  
**t:** Student’s t-distribution; estimating mean for small sample sizes  
**binom:** binomial distribution; e.g. die rolling and coin flipping  
**pois:** poisson distribution; e.g. rare events modeling

## input and output

### **DATA INPUT AND IMPORT**

#### **[]**

We have already met the concatenation operator. It can be used to manually input a small number of values like this:

```
x = [1, 2, 4, 8, 16, 32, 64]
```

#### **load()**

If you have your data, **numbers only**, in a tab delimited text file (from Excel or notepad) named “mydata.txt”, you can read it into MATLAB by typing:

```
x = load('mydata.txt')
```

and – poof – your data are in MATLAB, tucked away in the variable x (if the file isn’t in MATLAB’s current working directory, you have to specify the path in addition to the file within the quotes).

You can also use

```
load mydata.txt % no quotes
```

Your data will then be in variable “mydata”.

#### **dlmread()**

Same as above, but can be used for files delimited by coma, tab, or any other delimiter.

#### **xlsread()**

For Excel files that have variable names in the first row and numerical values elsewhere, use xlsread().

```
[mydata, variablenames]=xlsread('mydatawithheader.xls');
```

### ***data import wizard***

Matlab also comes with a GUI-based data import wizard that can handle all kinds of files and formats of data within those files. Click on File/Import Data and locate your file in the explorer window, just as you would if you wanted to open the file. In the next window, you can preview the resulting variables that will be imported. By default, you will have two variables: 'colheaders', a cell array of strings containing the column names, and 'data', a numerical matrix containing the data values. Additionally, you may have 'textdata' containing other text and information that was in your file. Highlighting the name of a variable will allow you to preview and make sure that all data are being imported correctly. Alternatively, you can import each original column as a separate variable, with the original column name being used as a variable name. Handy!

## **DATA OUTPUT AND EXPORT**

### ***save()***

When you close your Matlab session, all variables that you had in your workspace will be lost, unless you save them. The save() function can save all your workspace or selected variables from it. To save all your current variables (your workspace) at the end of the day to continue working with them next time, simply use "save *filename*"

```
save myworkspace
```

A file called 'myworkspace.mat' will be created in your current directory and you can load it using "load myworkspace" next time you open Matlab. Watch all your variables re-appear in your workspace.

Often, we only need to save a particular variable. For that, we use syntax "save *filename variablename*". It is a good idea to name your filename the same as your variable name. Here, I create and save a matrix "accrt" containing accuracy and reaction time data of four participants:

```
accrt = [.75 1.20; .88 1.310; .59 .89; .72 1.52];  
save accrt accrt  
clear    % nothing in my workspace now  
load accrt    % accrt loaded back into the workspace
```

The usage of save() above only preserves your data for further use within Matlab. If you want to be able to read your data in another program outside Matlab, the most general method is to save it as a simple text file.

```
save accrt.txt accrt -ascii
```

You have created a file called “accrt.txt” that contains your data and can be opened in any application that has a text processing capability, including any text editor, spreadsheet application (Excel), and of course Matlab. One drawback is that save() formatted your data into scientific notation while you would prefer some other format (try opening “accrt.txt” in Notepad to see for yourself).

## ***dlmwrite()***

dlmwrite() can write your data using any delimited (hence, dlm write) while formatting your data as you wish. Although a little more complex than save(), dlmwrite() can be still one row command while producing files with number formats that you want:

```
dlmwrite('accuracyandrtdata.txt',accrt,'delimiter','\t','precision',3);
```

The ‘delimiter’ set to ‘\t’ produces tab delimited text. While there exists quite complex formatting syntax that can produce very specific formatting, setting the attribute ‘precision’ to a single number (e.g. 3 or 4, as in example) will format any number to the given number of significant digits, which is usually what you want.

## ***xlswrite()***

xlswrite() can write directly into new or existing Excel sheets. You can use it to write numerical data, just as dlmwrite:

```
accrt = [.75 1.20; .88 1.310; .59 .89; .72 1.52];  
xlswrite('dataonly.xls',accrt);
```

Or you can have your data including header row in a cell array, saving column names with your data:

```
data=num2cell(accrt);  
colnames = {'Accuracy', 'Rt'};  
datawithcolnames = [colnames; data];  
xlswrite('datawithcolnames.xls',datawithcolnames);
```

## ***Array Editor***

Another method of data import and export is to use Array Editor and clipboard capabilities. Array Editor is a spreadsheet-like editor which allows you to enter and edit variable values just as in Excel. If you double-click on a vector or matrix variable name in your workspace, Array Editor opens. You can then directly copy and paste variable values between Matlab and other application.

## Writing programs

Matlab code is nothing else than text file with a sequence of Matlab commands, usually saved with extension '.m' (dot m). There are two main types of code that you will find yourself creating: scripts and functions.

Scripts are sequences of commands that are executed as if they were typed into the command line. Any variable that you create or modify in your script will be present in the workspace after you execute it. You have to be thus very careful how you name any of your help or temporary variables, as you may accidentally overwrite important data. To create your first code, you can open any text editor, type "disp('I can write Matlab scripts!')" and save it as "luckyme.m" into your current Matlab directory. To run the script simply type "luckyme" into the command line and enjoy the fruit of your work. "luckyme" became a Matlab command of its own.

Functions are codes that start with a word "function" and may have several inputs and several outputs. Let say you want to create a function "sumsq" that takes a vector as an input and produces a sum of squares of its values (pretend that you do not know that such a function already exists). Your code would look something like this:

```
function out = sumsq(vector)
    squareall = vector.^2;
    sumall = sum(squareall);
    out = sumall;
```

This function requires one input and produces one output. To call the function, you do not need to know how the variables are named inside the function, it will take any variable name as input and assign the output to whatever you specified as the output variable name:

```
a = [1 2 3];
mysumsq = sumsq(a)
mysumsq =
    14
```

Variables in the workspace that were not explicitly inputed into the function are invisible to the function, variables created inside a function are invisible to the main Matlab environment and are lost once the function is executed.

### ***Working directory and Matlab path***

In order to be able execute (run) any program (code), Matlab must know where to find it. When you type a word in the command line, Matlab first looks if you have a variable of that name in your workspace. It is not a good idea to name your variable using a name of an existing command, but if you do it, Matlab will always assume that you are referring to the variable, not the command. Second, Matlab searches its current directory. Current directory is listed at the top of the Matlab command window or can be obtained by typing "cd". If neither workspace variable nor command (script) in the current directory exist,

Matlab starts searching Matlab path. Matlab path is an ordered list of directories that Matlab considers when looking for Matlab scripts and that you can view and modify using the menu option File/Set path. When you install Matlab, Matlab path will automatically contain all Matlab directories with all the built-in Matlab commands.

## ***Editors***

One of the nice things about MATLAB is that it has a fully integrated text editor specialized for writing programs in the MATLAB language. Personally, I see no reason to consistently use a third party editor. If, however, you find yourself away from a computer that has MATLAB but wish to write a draft of a program, you can do so in any editor or word processor and then paste it into a MATLAB m-file later.

## ***Commenting***

Anything following % (the percent sign) until the end of line is considered a comment and is ignored by MATLAB. *Be prolific in commenting your code.* It will save you and anyone reading your code much cursing and gnashing of teeth.

## ***Cell code structure***

In new versions of Matlab, you can use double percent sign (%%) to separate your code into meaningful chunks (“cells”) that allow for fast overview and debugging of the code. Instead of running the whole code to see if it is running properly, you can evaluate (run) smaller chunks of code using Ctrl+Enter. Cells also help you keep good programming practice and understand the structure of your, or someone else’s, code.

## ***KISS code***

Programmers pride themselves on writing clever, dense, obfuscated code. Do not do this. If a problem naturally breaks down into 5 steps, do it in 5 lines of code, rather than in 1 very cunningly written one. Again, this will save us from blasphemy and dental wear.

## **Help!**

Matlab comes with excellent help, both command line help and html searchable help.

*help(‘function’)* and *help function* (where “function” is the name of a function, like “sample”) are the most direct help, but you need to know the exact name of the function for which you need help.

*lookfor name* provides a search for both exact function name and any function which contains “name” in the first help line (definition line).

I strongly suggest you go the MATLAB website, then click “Documentation -> Other -> Contributed” (there is a direct link from the course web-site) and then download and print out the two “MATLAB reference cards”. Laminate them, carry them with you at all times, and try to memorize a function or two while you brush your teeth at night. Seriously, when I’m teaching this class, I have them in my backpack 24/7 (see “geek” above).

When you write your own code (script or function), do not forget to generate your own little help. Everything what is continuously commented at the beginning of your code will be displayed when you type

```
help codename
```

For instance, when you create a function “robustmean”, your code may look like this:

```
function out = robustmean(x)
% function out = robustmean(x)
%
% computes mean of all elements of x except elements
% that appear to be outliers,
% being more than 3 standard deviations from the mean
outlier = abs(x - mean(x)) < 3*std(x);
out = mean(x(~outlier));
```

Now, when anyone types *help robustmean*, the short usage information and definition will be displayed. Hurray!