Appendix: Tutorial Introduction to MATLAB

This appendix provides a brief introduction to MATLAB for the complete novice. By following the steps in this tutorial you will learn all that you need to know to start at the beginning of the book using the Resampling Stats functions. You do not need to know anything at all about MATLAB to follow this tutorial. (For a more extensive introduction to the various capabilities of MATLAB that go beyond Resampling Stats, see Reference [**?**] or one of the on-line tutorials available at `http://www.mathworks.com`.)

Following this tutorial will be most effective if you do so with a computer. In the following, the $\gg$ sign indicates something that you should type into the machine. After you press ENTER, the computer will print a response. We haven't shown that response here in order to encourage you to type in the commands yourself. Typing in the examples not only will help you to remember the various commands in MATLAB, but also serves as a self-checking exercise: before typing a command write down what you think the answer will be, and compare this to the answer that the computer gives. Remember that the computer is by definition correct: if the computer's answer differs from your own, it is either because you have not completely understood the command or because you typed in the command incorrectly.

Step 1: Starting MATLAB

Exactly how you do this depends on how your computer is set up. On Windows machines, MATLAB will generally be found in the Start/Programs menu. On UNIX computers, you may simply need to type `matlab` at

the shell prompt. In either case, after starting MATLAB you should be looking at the "MATLAB Command Window" which will prompt you with ≫. If MATLAB hasn't been installed on your computer, follow the instructions that come with MATLAB to carry out the installation.

Step 2: Create a variable and assign it a value

MATLAB deals primarily with numbers and collections of numbers called "vectors" and "matrices." If you have written a computer program before, you are familiar with the concepts of variables and values. A *variable* is a container that holds a *value*. In MATLAB, the value is typically a number or a vector or matrix of numbers. Each variable has a name, such as `var1` or `diastolic` or `data`. There are four things you do with variables: create them, assign them a value, changed the assigned value, and use the value that has already been assigned.

The following command creates a variable named `a` and assigns it the value 7. To execute the command, type the command at the prompt (≫) and press Enter. (You do not have to type the prompt: the computer produces this for you.)

```
≫   a = 7
   ans:      a = 7
```

Variables can be named just about anything you like, with the following restrictions: there should be no spaces in the name; the name should start with a letter; only letters and numbers and the underscore (_). Thus, `adog` and `a_dog` are valid names, but `a dog` is not, nor is `a-dog`.

Some names are reserved by MATLAB for "keywords," and you may not use them for variable names. Examples of keywords are `for`, `while`, `end`, `function`, and `return`. These are part of the language itself. If you use a key word as a variable name, you will get an error message from MATLAB.

Other names have already been given a meaning as functions in MATLAB such as `sum`, `count`, `min` and so on. It is best to avoid using such names since doing so will mask their original meaning. With experience, you will learn what names are legitimate and what aren't. For beginners, you may want to stick to simple names like `a`, `x`, `y`, `z`, `y2`, etc.

You can see a list of the variables you have defined by giving the command

```
≫   who
```

Step 3: Using a variable that you have already defined

To see what is the value of a variable, just type its name at the command prompt, e.g.

≫ `a`

To use the value of a variable in a calculation, just type the name of the variable where ever you would like the value to appear. For instance,

≫ `a+a`

adds `a` to itself.

Arithmetic operations follow the conventional notation. Try the following

```
≫  a = 2
≫  b = 3

≫  c = (a+b)^2
```

Note that `^2` means exponentiation to the power 2.

```
≫  d = (a-b)/(a+b)
```

After each assignment, MATLAB prints out the value of the variable. This can sometimes be irritating. You can use a semi-colon at the end of a line in order to suppress this printing. Try:

```
≫  e = 2*a + 7*b;
```

You should take note of the fact that the `=` symbol does not really mean "equals." Instead it means that the variable named on the left side of `=` should be assigned the value specified on the right side of `=`. For instance, consider the command

≫ `a = a+1`

This often perplexes first-time programmers. Obviously there is no number that is equal to itself plus 1, and so at first glance the command is imposing an impossible condition. But actually the command means something very straightforward: take the current value of `a` and add 1 to it. This new value should be assigned to `a`. The overall effect is to increase the value of `a` by 1.

You can make a *copy* of the value of a variable by creating a new variable and assigning it the value you want to copy. For example

≫ `b = a;`

The two variables `a` and `b` have independent identities. Changing the value of `a` (after the copy to `b` is performed) will not change the value of `b`. Try the sequence of commands:

```
≫  a = 10
≫  b = a
≫  a = 7
≫  b
```

Step 4: Use a Function

A *function* (sometimes called a *procedure* or *routine*) performs an operation. MATLAB has many built-in functions that perform a wide range of operations. The way to tell MATLAB to carry out a particular operation is to give the name of that operation as a command. For example,

```
≫  who
```

tells MATLAB to print a list of the variables you have created to the screen.

```
≫  sin(a)
```

tells MATLAB to take the sine of the value of `a`. The `sin` function takes one argument, the number that you want to take the sine of. Functions can take zero, one, or more arguments. For example

```
≫  rand(3,2)
```

produces a matrix of random numbers arranged in 3 rows and 2 columns.

Whenever a function takes arguments, those arguments are enclosed in a pair of parentheses. If there more than one argument, they are separated by commas, and **the order of the arguments is usually extremely important.**

You can find out what a function does, and what its arguments mean by giving the command

```
≫  help NameOfFunction
```

Try

```
≫  help resamp
```

In most cases, the result of using a function is a value. For example, the value of `sqrt(25)` is 5. You can use this value in the same way you would use any other value. For example:

```
≫  a = cos(a)
```

or

```
≫  a = 3.5*cos(a*b) + sin( sqrt(a) )
```

In a very few cases in MATLAB, you give a command without using parentheses around the arguments. We have seen an example of this with `help`, for instance

≫ `help help`

We will encounter another instance of this unusual situation in Resampling Stats with the function `tally`.

Step 5: Make a Vector

MATLAB deals with collections of numbers called vectors and matrices. A vector is simply a list of numbers. A matrix is a rectangular array of numbers. There are many ways to make a vector; one of the most useful is to make a sequence using the colon (`:`) sequencing operator. Try the following commands:

≫ `a = 1:10;`

≫ `b = 5:10;`

The second of these commands translates into English as "b is assigned to be the numbers 5 to 10." You will mostly use the sequencing operator in this way, to make integers (whole numbers). However, the sequencing operator can also handle nonintegers:

≫ `c=5.5:10.5`

or even arithmetic operations

≫ `c=(sqrt(7)):(3+2)`

In both of these cases the size of the step in the sequence is 1. You can set the step size yourself if you want it to be different from 1:

≫ `c=9:0.5:11.2`

which translates as "assign c to be the sequence from 9 to 11.2, stepping by 0.5. Note that 11.2 is not in the sequence since you can't get to 11.2 by taking steps of size 0.5 starting from 9.

Another way to make a vector, which we use extensively in Resampling Stats, is to use the collection brackets, `[ ]` . For instance:

≫ `c = [ 4 7 3 2 1 ];`

You can use variables in the collection brackets. Try

≫ `c = [ c 10*c ]`

≫ `[c 1:10]`

Collection brackets can also be used to make matrices. You don't need to know anything about matrices in order to use Resampling Stats, but you will need to use them from time to time to specify "sampling urns."

≫ `mat = [1 2; 3 4; 5 6; 7 8]`

Note that the rows of the matrix are separated by the semi-colon (`;`). All rows of a matrix must be the same length. The printed form of the matrix reveals the rectangular structure clearly.

```
≫  mat
  mat   1 2
        3 4
        5 6
        7 8
```

In using Resampling Stats, you will encounter other ways to make vectors and matrices, such as reading them from files, generating random numbers, and so on. These will be explained as you need them.

Step 6: Arithmetic with vectors

Arithmetic with vectors in MATLAB is very easy. Try the following:

```
≫  c = 1:5
≫  c+10
≫  c*5
≫  c/2
≫  c-1.1
≫  sqrt(c)
```

Note that the assignment operator was used only in the first command; the other commands use the value of `c` but they do not change it. If you want to change it, you need to use the assignment operator, e.g.,

```
≫  c = 5+c
```

The above examples involve operations on a vector and a single number. In some cases you may want to have operations on two vectors. You will not need to do this much in Resampling Stats, but you should know that it is possible.

```
≫  c=1:5

≫  c + c

≫  c - c
```

These statements add the corresponding elements of the two vectors to one another. For this to work, the vectors need to be the same length. You will get an error message if you try

```
≫  b = 1:10
≫  c+b    generates an error
```

At this point, you are probably thinking that multiplication and division work the same way. But if you try `c*c` you will get an error message. The reason is that MATLAB uses the rules of *matrix* multiplication and these rules don't work with `c*c`. In order to tell MATLAB that you

want to multiply two vectors together in an element-by-element fashion, you need to say `c.*c` Similarly, division is `c./c`

Step 7: Other common operations on vectors

Some operations on vectors give back a single number, or perform some other operation such as plotting the vector. Try

```
≫  d = 1:10
≫  mean(d)   average of the numbers in d
≫  std(d)   standard deviation of the number in d
≫  median(d)
≫  plot(d)
≫  sum(d)
≫  plot(d, sin(d))
```

Step 8: Boolean Questions

A Boolean question is one whose answer is either "true" or "false". For example

```
≫  7 > 2
```

is true. In MATLAB, false is represented by 0 and true [1] by 1. You will be using several logical operations that apply to numbers and that will be familiar. `> < >= <= == ~=` Try the following. First, we'll create short vector to use in the examples:

```
≫  c=0:5 - 2
```

and now ask boolean questions about it.

```
≫  c > 0

≫  c >= 0
≫  c <= 1
≫  c == 1
≫  c ~= 1   not equals
```

These statements can be translated into English as questions: e.g. "is the value greater than 0" and so on. For vectors, the question is asked for each element in the vector independently. Note that to ask "is the value equal to 1" one uses the double equal sign `==`. This is to distinguish the two different operations of assigment (represented by a single `=`) and questioning about equality (represented by `==`). You, as every other computer programmer ever born, will make mistakes here: MATLAB will sometimes detect this mistake and will print an error message,

[1] Actually, true is represented by any non-zero number.

but this error message may be obscure, such as "Missing operator" or "A closing right parenthesis is missing." When you get such an error message for a line with an equal sign, check carefully whether you want there to be a single `=` for assignment or a double equal `==` for asking whether two values are numerically equal.

Matlab also has logical operators for "and", "or" and "not" which can be written &, `|`, and `~` respectively. For instance, to ask whether elements of `c` are between -1 and 1, we can write

```
≫    c >= -1 & c <= 1
```

Finally, the functions `any` and `all` can be used to ask whether any or all of the elements in a vector satisfy a relationship. For example

```
≫   any( c > 1 )
```

asks whether any of the elements in `c` are greater than 1.

```
≫   all( c > 1 )
```

asks whether all of the elements are greater than 1.

Step 9: Loops and Repeating

An important feature of many computer programs is a loop, wherein the same instructions are repeated many times. The most common sort of loop in Resampling Stats is the `for` loop. Type the following lines at the MATLAB command prompt:

```
for k=1:10      Press enter
   k + 100       '"     "   again
end             and again
```

This will print out the numbers 101 through 110. (If you had typed a semi-colon `;` at the end of the line `k + 100`, printing would have been suppressed.)

There are 4 parts to a `for` loop:

1. The key word `for` indicating the start of the loop.
2. A statement of the form *`variable = a list of numbers`*. In the above example, this is the statement `k=1:10`. We'll call `k` the "counting variable." (Remember, `1:10` is the list of numbers `1 2 3 4 5 6 7 8 9 10`.)
3. A set of MATLAB statements, one per line. This is the body of the loop.
4. The keyword `end` to signify where the body of the loop ends.

A loop works this way: The counting variable (`k` in the above example) is set to the value of the first element in the list. Then, all of the statements inside the body are executed in order. Then, the counting variable is set to be the next element of the list, the statements in the body are evaluated again, and so on until all of the elements in the list have been used.

Step 10: Conditional Expressions

A conditional expression is an expression that is evaluated only if a given condition is true. The `if` keyword can be used to indicate a conditional expression, for example

```
if score > 20
  wins = wins + 1;
end
```

In the above, the condition is that the value of `score` be greater than 20. If this is true, then the expression `wins = wins + 1` will be evaluated; otherwise no expression is evaluated.

There are more complicated forms of conditional expressions that involve the keywords `else` and `elseif`. Try the following lines:

```
score = 10;
if score == 0       % Note the double == signs
  val = 0;
elseif score < 10
  val = 1;
else
  val = 2;
end
```

What is `val` after these expressions have been evaluated.

Conditional expressions are often contained in loops. For example

```
scores = [12 8 9 7 16];
for k=scores
  if k < 10
     val = 0;
  else
     val = 1;
  end
  val     % print val.  Don't use a semi-colon!
end
```

There are two things to note about this example.

- The keyword `end` appears twice. The first, inner `end` signifies the end of the `if` statment. The second, outer `end` marks the end of the loop.
- The `if` statement involves the counting variable `k`. Remember, in the `for` loop, `k` will be set successively to each of the values in the vector `scores`.

Step 11: Saving Your Work and Quitting

Let's finish up this session using MATLAB. Before finishing, though, let's save the variables we've created. Giving the command

≫ `who`

prints a list of the variables we've created. We will save these variables for later use. The simplest way to do this is to use the FILE/SAVE WORKSPACE AS command from the menu bar. When you do this, a file-saving dialog box will be displayed. Give some unique name (for example, `mywork` or `jan10`) to the file to be saved, and save it in an appropriate directory. (The file will have a `.mat` file-name extension.)

If you don't like using the graphical user interface, you could also use `save myfile *` on the command-line. `save` also allows you to save just selected variables, for example `save myfile a b scores`.

Now quite MATLAB using the

≫ `quit`

command or the FILE/EXIT command from the menu bar.

Your system may print out a message like `32987 flops` indicating the number of arithmetic operations carried out by MATLAB during the session. In your future sessions using Resampling Stats you may conceivably exceed the number of arithmetic operations carried out by all humans prior to, say, the year 1500. Welcome to the world of computer-intensive statistics!

Step 12: Starting a New Session

Welcome back. Startup MATLAB again, and check to make sure that you have access to the Resampling Stats software by typing the command

≫ `help resamp`

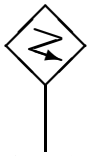
If you get the response `resamp.m not found`, then things are not set up right and you should follow the instructions for setting your path in Sec. **??**.

Now, check on your old variables.

≫ `who`

What? They have disappeared! Of course. MATLAB is oriented around the idea of sessions. Each time you start MATLAB, you start a new session. To recall the variables you saved from an old session, you can give the FILE/LOAD WORKSPACE command from the menu bar, and then load the workspace file you saved in Step **??**. (Or, you could use the `cd` and `load` commands on the MATLAB command line.)

MATLAB will not automatically save the variables from your session when you quit.



CAUTION!

Step 13: M-file Scripts

You have already seen how to save variables so that you can use them in another MATLAB session. You can also save sequences of commands so that you can re-use the commands. To do this, invoke the "m-file" editor by using the FILE/NEW/M-FILE menu-bar command or by giving the

≫ `edit`

command on the command line.

Inside the editing window type the following lines exactly as they appear below:

```
% My first m-file
% Add up the numbers 1 to 10
total = 0;
for k=1:10
  total = total + k;
end
```

Use the FILE/SAVE command in the editing window to save this file with name `mysum.m`. (The editor will automatically add the file-name extension `.m`) Save the file in the directory `c:/resamp/myfiles` or whereever you installed the Resampling Stats software. (See Section **??**.) If you prefer, use another directory, but make sure that it is in the MATLAB path as described in Section **??**.

You have just created a new MATLAB command `mysum`. Try it out:

≫ `mysum`

The consequence of giving this command is that all of the instructions in `mysum.m` are executed.[2] (The `%` sign indicates that the rest of the line is a comment, not a command to be executed.) You can check out the result by asking for the value of `total`

≫ `total`

The value should be 55. Note that before you issued the `mysamp` command there was no variable called `total`; it was created by the instructions in `mysamp.m`.

`mysamp.m` is an example of a script. Creating such scripts saves the tedium of re-typing often-used commands, but it also has a much more important role: it allows you to avoid the bugs that can be introduced by incorrect typing, and allows you to refine, correct, and test your commands.

Unlike variables, m-files do not need to be loaded into each session. Instead, whenever a command is given, MATLAB looks in the current directory for a file with that name (ending in `.m`). If the file is not found in that directory, then MATLAB looks through a set of other directories, the "path," stopping in the first directory in which the file is found. (See Sec. **??**.)

Sometimes it may happen that you create an m-file command that has the same name as an existing MATLAB function. When this occurs, only one of the two synonomous commands will be available to MATLAB. You can avoid this situation by checking before saving a new m-file, using the command `which`. For example, if you are thinking of creating a new command called `opensesame`, try

≫ `which opensesame`

If the response is "opensesame not found," then it is safe to use the name.

Step 14: M-file Functions

You've decided to generalize your `mysum` command to add up the integers in any specified range from `low` to `high`. A sensible way to do this is to use a command of the form `mysum(low,high)`. Such a command is called a "function." `low` and `high` are the arguments to the function.

To create this function, edit your file `mysum.m` so that it looks like this (erasing most of the old contents):

[2] If MATLAB could not find `mysum.m` in your path you will get an error message like `??? Undefined function or variable 'mysum'`. If you get this message, go back and make sure that the file `mysum.m` was saved in an appropriate directory on the MATLAB path. See Section **??**.

```
function res = mysum(low,high)
% mysum(low, high)
% adds all the numbers between low and high
total = 0;
for k=low:high
  total = total + k;
end
res = total;
```

Make sure to save the edited version of `mysum.m`, overwriting the old version.

Note the following differences between the original `mysum.m` and the new version:

- The first word in the file is the keyword `function`. The rest of the first line indicates the `mysum` returns a value called `res`, and that it takes two arguments, `low` and `high`. When the function is evaluated with two arguments, for example, `mysum(1,100)` then `low` will automatically be assigned the value of the first argument (in this case, 1), and `high` will be assigned the value of the second argument (10).

- The next lines, starting with comment signs (`%`) are more than just comments. Whenever someone types
  ≫ `help mysum`
  these lines will be displayed.

- The variable `res`, identified on the first line as being the value returned by `mysum`, is assigned the desired value. In this example, this is done on the last line.

Try out your new function:
≫ `mysum(1,100)`
The returned value should be 5050.

There are two very important things to note about how the function `mysum` works, and how it differs from the script you created in Step **??** .

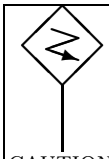
1. No variable called `res` is created in the session. `res` in `mysum.m` is just a temporary handle to give a name for the result.
   ≫ `res`
   ans: ??? No such function or variable 'res'

2. The variable `total` inside `mysum.m` is similarly just a temporary handle. It does not affect the variables in the session:
   ≫ `total` ⇒ *`ans: 55`*
   `total` did not get changed by the `mysum` command.

The general rule for functions is this: nothing that happens inside a function changes the value of any variable in the session. The only way to change a variable in a session is to use the assignment operator *outside* of a function. For example,
≫ `total = mysum(1,100)`
changes the value of total.

This general rule means that you are perfectly free to use any variable names inside a function, without having to worry that you might accidentally change a variable in your session. This is an extremely valuable feature of functions.[3]

Using functions for often-repeated calculations allows you to save typing, as was the case with scripts. Importantly, the invisibility of a function's internal variables to the session means that you can modify a function to make it more efficient or to fix a bug, and that any improvements are automatically picked up by those commands that use the function.



CAUTION!

Remember to use the FILE/SAVE menu command to save any changes you make to a function file. Unless you do this, the changes you make will have no effect.

Step 15: Vectors and Matrices

As already pointed out, a vector is a single row or column of numbers, a matrix is a rectangular array of numbers. Some examples:

```
vec1 = [1 2.2 4 5];
vec2 = [10;
         9;
         8;
         7;
```

[3] For experts: Every rule has an exceptions, and it actually is possible to write functions in a special way so that they change variables outside the function — but this needs to be arranged explicitly and is rarely needed. See TALLY for an example.

```
         6]
mat  = [1 9.3;
        0 6.8;
        1 4.2;
        1 3.9]
```

Sometimes one needs to access a single element of a vector, or perhaps several items. This can be done in either of two ways, best illustrated by example.

1. The index of a vector element tells the position of the element in the vector. The index of the first element is 1, the second is 2, and so on. The last element's index is the same as the `length` of the vector. To access one or more elements, put the desired index or list of indices in parentheses after the vector name:
   ≫ `vec1(1)` ⇒ *ans: 1*
   ≫ `vec1([2 3]` ⇒ *ans: 2.2 4*
   ≫ `vec2([1 3 5])` ⇒ *ans: 10*
   8
   6

2. A Boolean vector is a vector created by using a logical operation with a yes-or-no answer, represented by 1 (for yes) and 0 (for no). (See Step **??**.) You can use a Boolean vector to access members of a vector; the Boolean vector should be the same length as the vector being accessed. Whereever there is a 1 in the Boolean vector, the corresponding element will be accessed.
   ≫ `vec1 > 2`
   ans: *0 1 1 1* a Boolean vector
   ≫ `vec1( vec1 > 2 )`
   ans: *2.2 4 5*
   ≫ `vec2( vec2 = 8 )`
   ans: *10 9 7 6*

Indexing also works for assigning values to elements of a vector. For example.
≫ `vec1(1) = 7`
≫ `vec1` ⇒ *ans: 7 2.2 4 5*
This change to `vec1` is permanent, until we change it again.

Indexing of matrices works in much the same way as vectors, but you need to specify both a row and a column index, separated by a comma.
≫ `mat(3,2)` ⇒ *ans: 4.2*

A colon (":") will take all of the elements

```
≫  mat(3,:)  ⇒  ans:   1 4.2
≫  mat(:,1)  ⇒  ans:   1 0 1 1
≫  mat( mat(:,1)==1, :  )  ⇒  ans:   1.0 9.3
                                     1.0 4.2
                                     1.0 3.9
```

Finally, sometimes you want to transpose a row vector into a column vector, or vice versa. The `transpose` operator does this:

```
≫   a = transpose(vec1)
   ans:     7 2.2 4 5
≫   b = transpose(vec2)
   ans:     10 9 8 7 6
≫   c = transpose(mat)
   ans:     1 0 1 1
            9.3 6.8 4.2 3.9
```