**MatLab Training Course Notes  
Session 5**

**Introduction to programming with MatLab**

Instructor:  
Bill Miller  
Dynamic Neuroimaging Laboratory  
415.502.3726  
<http://dnl.ucsf.edu/>

**Topics for Session 4**

1. [Concepts: programs and algorithms](http://dnl.ucsf.edu/matlab/session5.html#Concepts)
2. [Loops: for, while](http://dnl.ucsf.edu/matlab/session5.html#Loops)
3. [Branching: if, switch, break](http://dnl.ucsf.edu/matlab/session5.html#Branching)
4. [Creating permanent programs in MatLab](http://dnl.ucsf.edu/matlab/session5.html#Creating)

Note: This session concludes the instruction of the basics of MatLab. A few follow-on sessions will explore advanced topics.

**1. Concepts: programs and algorithms**

It is useful to define some basic terms related to creating programs. A *program* is a series of instructions for the computer's processor for how to manipulate data. They must be translated from words to machine language (1's and 0's). The data itself (on the disk) is also stored as 1's and 0's. So somehow the instructions and data interact in binary language (i.e., a Turing machine). MatLab insulates you from the translation of programs to binary ('compiling'), which makes it easier to deal with than other programming languages. An *algorithm* refers to a technique or specific series of steps, expressed in words or math, to compute something.

**Example: An algorithm to remove the global mean from a series of data points**

Step 1. Compute the mean:

Step 1a. calculate the sum of all data values

Step 1b. divide sum by the number of points

Step 2. Remove the mean:

Step 2a. subtract the mean from each data point

This algorithm could be *implemented* as a program (in perhaps more than one way) in any programming language. In MatLab, the above algorithm can in fact be implemented in several ways, for example in the conventional way by using ["loops"](http://dnl.ucsf.edu/matlab/session5.html#Loops) (discussed below):

a=[2 3 5 8 10];

b=0;

for z=1:5

b=b+a(z);

end

b=b/5;

for z=1:5

a(z)=a(z)/b;

end

Alternatively, since MatLab allows compact handling of vectors, as we've seen earlier, we could write shorter programs which implements this algorithm without loops, using MatLab built-in functions sum, length, mean or detrend:

a=[2 3 5 8 10];

b=sum(a)/length(a);

a=a-b;

or

a=a-mean(a);

or

a=detrend(a,0);

**2. Loops: for, while**

Looping mechanisms are provided in every programming language to allow you to perform the same operation repetitively. MatLab provides two typical mechanisms for looping, as follows.

**- for loops:**

Syntax:

for *var=start:finish*

*(statements)*

end

The basic features of a for loop are:

* a counting variable in the for statement
* a counting range in the for statement
* an end statement defining the end of the loop
* this type of loop is always executed in a program

When a for loop is first reached in a program, the counter variable ("var" above) is set to the *start* value. Then, the statements inside the loop are executed. When the end is reached, execution jumps back up to the for line again, at which time the counter variable is set to the next value in the *start:finish* range. The looping continues this way for all values of the range (including the *finish* value), at which point execution drops out of the loop and on to subsequent statements in the program (if any).

**Example:** Fill a vector with 5 even numbers (0...8)

Algorithm:

1. start with an empty vector a

2. loop a counter from 1 to 5

3. calculate the value of an even number based on the counter:

even=2\*counter-2;

4. put the even value into the position in "a" with index corresponding to the current value of counter

Code:

a=[];

for z=1:5

even=2\*z-2;

a(z)=even

end

**- while loops:**

Syntax:

while *[something is true]*

*(statements)*

end

The basic features of a while loop are:

* a logical test in the while statement
* no counters
* an end statement defining the end of the loop
* this type of loop is only executed if the logical test is "true" when the program initially reaches the while statement.

while loops are a nice way of avoiding counters if you don't need to use them, and for executing a loop only if necessary. Once the loop is entered, repetitive execution of the statements inside the loop continues until the logical test at the beginning of the loop is no longer true.

**Example:**

a=10;

while a>2

a=a/2

end

In the above example, when the while loop is first encountered, a=10, so the logical test (i.e., "a" is greater than 2) evaluates to "true" and the loop is entered. During each repetition of the loop, the value of "a" is halved. A some point "a" will be less than 2, and the logical test at the top of the loop will evaluate to "false", whereby execution moves out of the loop an on to any subsequent statements in the program. What will be the final value of "a"? (To find out, copy the above code, paste it into the MatLab command window, and press return).

**3. Branching: if, switch, break**

Branching allows you to conditionally move to another place in the program or start/stop doing something. (The term "branching" refers to the many paths a program can take depending on user choices and variable values). MatLab provides the two typical mechanisms for branching found in most programming languages, as follows.

**- if clause:**

Syntax:

if *[something is true]*

*(statements)*

elseif *[something else is true]*

*(statements)*

else

*(statements)*

end

This is by far the most common way to do branching in programs. When the if statement is reached in a program, the code between if and end is executed only if the logical test evaluates to "true". The elseif and else are optional. If you provide else, the statements between it and end will be executed if the initial logical test on the if line evaluates to "false".

**Example:**

a=[0:2:10];

for z=1:length(a)

if a(z) >= 5

disp('big')

else

disp('little')

end

end

In the above example, we use an if clause "nested" inside a for loop. At the beginning of the program, "a" is defined as a vector of even numbers. The program then loops through each element of "a" and tests if that element has a value greater or equal to 5: if so, it prints "big" in the command window; if not, it prints "little" in the command window.

Note that it is very important to keep track of the end statements in your programs. In the above example, we needed two, one for the for loop and one for the inner if clause.

**- switch clause:**

Syntax:

switch *var*

case *value(s)*

*(statements)*

case *value(s)*

*(statements)*

:

:

otherwise

*(statements)*

end

switch is a useful, compact alternative to if when you want to take a few or many different alternative paths based on particular values of a single variable. The variable *var* above can have a single value or be a string; no numeric vectors or matrices are allowed. There can be one to many case statements, one for each value of *var* that you want to deal with; otherwise is optional and serves the same role as else in the if syntax. It is possible to consider more than one value in a case statement by using curly brackets ( {..,..} ).

**Example:**

switch a

case 0

disp('a=0')

case {1,2,3,4}

disp('a is less than 5')

case 10

disp('a is 10')

otherwise

disp('a is out of range')

end

In the above example, we have "trapped" the values 0, (1-4) and 10 of variable "a", and for each of these the switch clause will produce a different output in the command window; any other values will produce a "default" message. To try this code, first create "a" with some value, copy the above code and paste it into the command window, and press return.

**- break:**

break is a very useful statement that allows you to exit a for or while loop prematurely, and it is most commonly used within an if clause:

**Example:**

a=10;

while a>2

a=a/2;

if a<5

break;

end

end

The above example is a modification of the earlier example for the while loop. Here, when the value of "a" falls below 5, the break statement is reached, and program execution jumps out of the while loop and continues on to any subsequent statements.

**4. Creating permanent programs in MatLab**

* M-files
* Matlab built-in editor
* Program structure: Function definition, comment lines, and body
* Scripts vs. functions
* Built-in functions

**- M-files**

Usually if you are doing repetitive analysis, you don't want to retype the same lines each time. Instead, you typically store the program lines in a file which can then be executed at any time. In Matlab, stored programs contain exactly the same kinds of statements as you would type on the command line. They are stored in text files - so you can use any text editor to create them. There are just two special requirements on the naming of MatLab program files:

1. The filename can't contain a period ('.'), hyphen or spaces
2. The filename must end with a ".m" extension (they are thus referred to as "m-files)"

Also, m-files must be located somewhere on your MatLab path (or in your current working directory) in order for MatLab to find and be able to execute them.

**- The MatLab built-in editor, medit.exe**

MatLab provides a nice editor for creating programs which can be started at the MatLab command line (by typing "edit"). It uses different colors for different types of statements for easier reading, and automatically indents the internal statements in loops and in branching clauses. (There are also debugger features we won't discuss).

**- Program structure: Function definition, comment lines, and body**

MatLab programs are really "functions", since you use them on the command line just as if they were built-in functions provided by MatLab. In fact, the top line of an m-file will usually have a function definition statement:

function [*output args*]=*function\_name*(*input\_args*)

The function name should match the name of the file itself (without the ".m" extension).

Following the function definition can be any number of comment lines for help text. These must begin with the percent character (%). These are not executed by MatLab, but are very useful in providing for providing some help text explaining how your function works and its syntax.

After the initial help comments comes the "body" of your program, i.e., the executable lines of MatLab code. You can insert comment lines anywhere in your code (even on the same line after an executable statement), as long the comment starts with "%".

**Example: HelloWorld.m**

(copy the code below and paste it into an empty document window

of the MatLab editor or any text file; save the file as

"HelloWorld.m" in a directory somewhere on your MatLab path)

function b=HelloWorld(a)

%HelloWorld displays nice messages

%

% Format: b=HelloWorld(a)

% - a is a number

% if a>0 a happy message is displayed

% if a<0 a sad message is displayed

% - b contains the displayed message

%

%Created March 28, 2001 by Bill Miller

%Modified: never

if a>0

b='Hello World!';

else

b='Goodbye cruel world...';

end

disp(b);

%end of program

In the above example, the function definition indicates that HelloWorld takes one input argument (variable "a"), and returns one output argument (variable "b"). "a" and "b" are used internally inside the function, but when you call this function, the arguments you type at the command line don't have to be "a" and "b", as we'll see below.

If you've saved HelloWorld.m somewhere on your MatLab path, try typing the following on the MatLab command line:

type helloworld %lists the contents of HelloWorld.m in the command window

help helloworld %lists the initial comment lines

helloworld(1); %returns no output argument

c=helloworld(-1) %returns the output argument into a variable c

Next, try

edit helloworld %launch the editor with HelloWorld.m as the document

Note that you don't have to include the extension of the m-file when using edit.

We see that the MatLab editor has colored the various lines of code according to their type. Here's a good place to suggest a good structure for your help comments:

* The first line should be the name of the function followed by a brief description of what it does
* There should be a "format" description, which shows a copy of the function definition line, followed by a description of each of the arguments and what actions the input arguments cause.
* The last lines should show who created the function, when it was created, and when it was modified (and how).

--> Note that in addition to the function for which the m-file is named, the file can contain additional "embedded" or "private" functions (each with its own function definition line) which are only recognized and used by the first (top-most) function.

**Another example: creating a graphic figure with an m-file:** One of the convienient uses of m-files is for design and regeneration of graphic figures (e.g., as part of an analysis or for publication). Once you've designed the figure, and recorded it as a MatLab program in an m-file, you essentially have a permanent record of how you created the figure, and an easy mechanism for modifying/updating it.

The following example program takes a matlab demo image, along with text strings, and creates a simple figure. Follow the instructions in the help section of the program, and play around with the default settings. (Click on the link to download the program, or copy the program into a new file called makemyfigure.m).

[**MakeMyFigure.m**](http://dnl.ucsf.edu/matlab/makemyfigure.m)

function makemyfigure(imagefile,titletext,xtext,ytext)

%MakeMyFigure illustrates how to create a nice figure

%which plots your data using a function

% Format: makemyfigure(imagefile,titletext,xtext,ytext)

% -imagefile is a .mat file containing an image (as X)

% (try with spine.mat or topo.mat)

% -titletext is a string for the plot title

% -xtext,ytext are strings for the x- and y- axes

% Created March 28, 2001 by Bill

%default values for subplot

rows=3;

cols=3;

index=5;

switch nargin

case 3

ytext='y-axis';

case 2

xtext='x-axis';

ytext='y-axis';

case 1

titletext='title';

xtext='x-axis';

ytext='y-axis';

case 0

imagefile='spine.mat';

titletext='title';

xtext='x-axis';

ytext='y-axis';

end

load(imagefile)

if exist('X')

data=X;

else

whos

q=input('Which variable is the image?: ','s');

if ~isempty(q) & exist(q)

data=eval(q);

else

return;

end

end

figure

subplot(rows,cols,index)

imagesc(data)

title(titletext)

xlabel(xtext)

ylabel(ytext)

**- Functions vs. scripts: the workspaces**

You can create an m-file without the "function" line. In that case, all variables created by the program are located in the "base" workspace - available on the command line. Otherwise, with functions, all variables exist only in the workspace of the function, and disappear when the function completes (except for the output arguments - which are created in the base workspace - i.e., the link between the function and the base workspace.

--> Note that global variables can be declared in a function and can continue to exist after the function stops (and you can access them from the command line using the "global" statement).

**- Built-in functions**

MatLab provides many useful basic functions as part of the basic package. Some are "built-in", i.e., actually part of the executable application, so you aren't able to look at the actual function code; but others are written as m-files, so you can in fact look at their code (with type or edit followed by the function name). All, however, provide help information (i.e. responses to help followed by the function name). Take particular look at the following groups of functions (e.g., type help elmat to get the list of elementary matrix functions, then help with any of the functions in that group):

matlab\general - General purpose commands.

matlab\ops - Operators and special characters.

matlab\lang - Programming language constructs.

matlab\elmat - Elementary matrices and matrix manipulation.

matlab\elfun - Elementary math functions.

matlab\matfun - Matrix functions - numerical linear algebra.

matlab\datafun - Data analysis and Fourier transforms.

matlab\graph2d - Two dimensional graphs.

matlab\graph3d - Three dimensional graphs.

matlab\graphics - Handle Graphics.

matlab\strfun - Character strings.

matlab\iofun - File input/output.