4. Tools

MATLAB

In this part of the tutorial a brief introduction to some basic concepts of MATLAB will be given. The students, however, are strongly encouraged to extensively use the MATLAB help files, accessible via the main MATLAB window. Type “doc” on the MATLAB command line to open the help browser. MATLAB blogs also provide useful information about how to program in MATLAB (http://blogs.mathworks.com).

a) Vectors and Arrays

A vector in MATLAB can be easily created by entering each element between brackets and assigning it to a variable, e.g.:

```
a = [1 2 3 4 5 6 9 8 7]
```

Let's say you want to create a vector with elements between 0 and 20 evenly spaced in increments of 2:

```
t = 0:2:20
```

MATLAB will return:

```
t =
     0   2   4   6   8   10   12   14   16   18   20
```

Manipulating vectors is almost as easy as creating them. First, suppose you would like to add 2 to each of the elements in vector 'a'. The equation for that looks like:

```
b = a + 2

b =
     3   4   5   6   7   8   11   10   9
```

Now, suppose you would like to add two vectors. If the two vectors are the same length, it is easy. Simply add the two as shown below:

```
c = a + b

c =
     4   6   8   10   12   14   20   18   16
```

In case the vectors have different lengths, then an error message will be generated.
Entering matrices into MATLAB is the same as entering a vector, except each row of elements is separated by a semicolon (;) or a return:

```
B = [1 2 3 4;5 6 7 8;9 10 11 12]

     B =
          1     2     3     4
          5     6     7     8
          9    10    11    12

B = [ 1  2  3  4
      5  6  7  8
      9 10 11 12]
```

```
B =
     1     2     3     4
     5     6     7     8
     9    10    11    12
```

Matrices in MATLAB can be manipulated in many ways. For one, you can find the transpose of a matrix using the apostrophe key:

```
C = B'

C =
     1     5     9
     2     6    10
     3     7    11
     4     8    12
```

Now, you can multiply the two matrices B and C together. Remember that order matters when multiplying matrices.

```
D = B * C

D =
    30    70   110
    70   174   278
   110   278   446

D  = C * B

D =
   107   122   137   152
   122   140   158   176
   137   158   179   200
   152   176   200   224
```

Another option for matrix manipulation is that you can multiply the corresponding elements of two matrices using the .* operator (the matrices must be the same size to do this).

```
E = [1 2;3 4]
F = [2 3;4 5]
G = E .* F
E =
     1     2
     3     4
F =
     2     3
     4     5
G =
     2     6
    12    20
```

MATLAB also allows multidimensional arrays, that is, arrays with more than two subscripts. For example,

```
R = randn(3,4,5);
```

creates a 3-by-4-by-5 array with a total of 3x4x5 = 60 normally distributed random elements.

b) Cell arrays and structures

Cell arrays in MATLAB are multidimensional arrays whose elements are copies of other arrays. A cell array of empty matrices can be created with the `cell` function. But, more often, cell arrays are created by enclosing a miscellaneous collection of things in curly braces, `{}`. The curly braces are also used with subscripts to access the contents of various cells. For example

```
C = {A sum(A) prod(prod(A))}
```

produces a 1-by-3 cell array. There are two important points to remember. First, to retrieve the contents of one of the cells, use subscripts in curly braces, for example *C{1}* retrieves the first cell of the array. Second, cell arrays contain *copies* of other arrays, not *pointers* to those arrays. If you subsequently change `A`, nothing happens to `C`.

Three-dimensional arrays can be used to store a sequence of matrices of the *same* size. Cell arrays can be used to store sequences of matrices of *different* sizes. For example,

```
M = cell(8,1);
for n = 1:8
   M{n} = magic(n);
end
M
```

produces a sequence of magic squares of different order:

```
M =
    [           1]
    [ 2x2  double]
    [ 3x3  double]
    [ 4x4  double]
    [ 5x5  double]
    [ 6x6  double]
    [ 7x7  double]
    [ 8x8  double]
```

Structures are multidimensional MATLAB arrays with elements accessed by textual *field designators*. For example,

```
S.name = 'Ed Plum';
S.score = 83;
S.grade = 'B+'
```

creates a scalar structure with three fields.

```
S =
     name: 'Ed Plum'
    score: 83
    grade: 'B+'
```

Like everything else in MATLAB, structures are arrays, so you can insert additional elements. In this case, each element of the array is a structure with several fields. The fields can be added one at a time,

```
S(2).name = 'Toni Miller';
```

```
S(2).score = 91;
S(2).grade = 'A-';
```
Or, an entire element can be added with a single statement.

```
S(3) = struct('name','Jerry Garcia',...
              'score',70,'grade','C')
```
Now the structure is large enough that only a summary is printed.

```
S =
1x3 struct array with fields:
    name
    score
    grade
```
There are several ways to reassemble the various fields into other MATLAB arrays. They are all based on the notation of a *comma separated list*. If you type

```
S.score
```

it is the same as typing

```
S(1).score, S(2).score, S(3).score
```

This is a comma separated list. Without any other punctuation, it is not very useful. It assigns the three scores, one at a time, to the default variable `ans` and dutifully prints out the result of each assignment. But when you enclose the expression in square brackets,

```
[S.score]
```

it is the same as

```
[S(1).score, S(2).score, S(3).score]
```

which produces a numeric row vector containing all of the scores.

```
ans =
    83    91    70
```
Similarly, typing

```
S.name
```
just assigns the names, one at time, to `ans`. But enclosing the expression in curly braces,

```
{S.name}
```
creates a 1-by-3 cell array containing the three names.

```
ans =
    'Ed Plum'    'Toni Miller'    'Jerry Garcia'
```
And

```
char(S.name)
```
calls the `char` function with three arguments to create a character array from the `name` fields,

```
ans =
Ed Plum
Toni Miller
Jerry Garcia
```

c) Functions

To make life easier, MATLAB includes many standard functions. Each function is a block of code that accomplishes a specific task. MATLAB contains all of the standard functions such as sin, cos, log, exp, sqrt, as well as many others. Commonly used constants such as pi, and i or j for the square root of -1, are also incorporated into MATLAB.

```
sin(pi/4)

ans =

    0.7071
```

To determine the usage of any function, type `help [function name]` at the MATLAB command window.

MATLAB allows you to write your own functions with the *function* command. The basic syntax of a function is:

```
function [output1,output2] = filename(input1,input2,input3)
```

A function can input or output as many variables as are needed. Below is a simple example of what a function, `add.m`, might look like:

```
function [var3] = add(var1,var2)
%add is a function that adds two numbers
var3 = var1+var2;
```

If you save these three lines in a file called "add.m" in the MATLAB directory, then you can use it by typing at the command line:

```
y = add(3,8)
```

Obviously, most functions will be more complex than the one demonstrated here. This example just shows what the basic form looks like.

d) Loops

If you want to repeat some action in a predetermined way, you can use the *for* or *while loop*. All of the loop structures in MATLAB are started with a keyword such as "for", or "while" and they all end with the word "end".

The *for loop* is written around some set of statements, and you must tell MATLAB where to start and where to end. Basically, you give a vector in the "for" statement, and MATLAB will loop through for each value in the vector: For example, a simple loop will go around four times each time changing a loop variable, *j*:

```
for j=1:4,
     j
 end
```

If you don't like the *for loop*, you can also use a *while loop*. The *while loop* repeats a sequence of commands as long as some condition is met. For example, the code that follows will print the value of the *j* variable until this is equal to 4:

```
j=0
while j<5
      j
      j=j+1;
end
```

You can find more information about *for loops* on http://blogs.mathworks.com/loren/2006/07/19/how-for-works/

e) Reading from files / Writing to files

Before we can read anything from a file, we need to open it via the *fopen* function. We tell MATLAB the name of the file, and it goes off to find it on the disk. If it can't find the file, it returns with an error; even if the file does exist, we might not be allowed to read from it. So, we need to check the value returned by *fopen* to make sure that all went well. A typical call looks like this:

```
fid = fopen(filename, 'r');
if (fid == -1)
  error('cannot open file for reading');
end
```

There are two input arguments to *fopen*: the first is a string with the name of the file to open, and the second is a short string which indicates the operations we wish to undertake. The string 'r' means "we are going to read data which already exists in the file." We assign the result of *fopen* to the variable *fid*. This will be an integer, called the "file descriptor," which we can use later on to tell MATLAB where to look for input.

There are several ways to read data from a file we have just opened. In order to read binary data from the file, we can use the *fread* command as follows:

```
A = fread(fid, count)
```

where *fid* is given by *fopen* and *count* is the number of elements that we want to read. At the end of the *fread*, MATLAB sets the file pointer to the next byte to be read. A subsequent *fread* will begin at the location of the file pointer. For reading multiple elements from the file a loop can be used in combination with *fread*.

If we want to read a whole line from the file we can use the *fgets* command. For multiple lines we can combine this command with a loop, e.g. :

```
while (done_yet == 0)

    line = fgets(fid);
    if (line == -1)
      done_yet = 1;
    end
end
```

Before we can write anything into a file, we need to open it via the *fopen* function. We tell MATLAB the name of the file, and give the second argument 'w', which stands for 'we are about to write data into this file'.

```
fid = fopen(filename, 'w');
if (fid == -1)
  error('cannot open file for writing');
end
```

When we open a file for reading, it's an error if the file doesn't exist. But when we open a file for writing, it's not an error: the file will be created if it doesn't exist. If the

file does exist, all its contents will be destroyed, and replaced with the material we place into it via subsequent calls to *fprintf*. Be sure that you really do want to destroy an existing file before you call *fopen*!
There are several ways to write data to a file we have just opened. In order to write binary data from the file, we can use the *fwrite* command, whose syntax is exactly the same as *fread.* In the same way, for writing multiple elements to a file, *fwrite* can be combined with a loop.
If we want to write data in a formatted way, we can use the *fprintf function, e.g. :*

```
fprintf(fid, '%d %d %d \n', a, b, c);
```

which will write the values of *a,b,c* into the file with handle *fid*, leaving a space between them. The string *%d* specifies the precision in which the values will be written (single), while the string \n denotes the end of the line.
At the very end of the program, after all the data has been read or written, it is good to close a file:

```
fclose(fid);
```

f) Avoiding "Divide by zero" warnings

In order to avoid "Divide by zero" warnings you can use the *eps* function. *eps(X)* is the positive distance from *abs(X)* to the next larger in magnitude floating point number of the same precision as X. For example if you wish to divide A by B, but B can sometimes be zero which will return *Inf* and it may cause errors in your program, then use *eps* as shown:

C = A / B; % If B is 0 then C is *Inf*
C = A / (B + eps); % Even if B is 0 then C will just take a very large value and not *Inf*.

g) Profiler / Debugging

The *profiler* helps you optimize M-files by tracking their execution time. For each function in the M-file, profile records information about execution time, number of calls, parent functions, child functions, code line hit count, and code line execution time. To open the *profiler* graphical user interface select HOME->Run and Time. So if the execution of your code is slow you can use the *profiler* to identify those lines of code that are slow to execute and improve them.

Another useful function that can be used for debugging is the *dbstop* function. It stops the execution of the program when a specific event happens. For example the commands

dbstop if error
dbstop if warning

stop execution when any M-file you subsequently run produces a run-time error/warning, putting MATLAB in debug mode, paused at the line that generated the

error. See the MATLAB help for more details. Alternatively, you can use the graphical user interface to define the events that have to take place in order to stop the program. Just select EDITOR menu ->Breakpoints->ERROR HANDLING Stop onErrors/Warnings.