Matlab Summary

Save code in files with the extension ".m".
To run the file filename.m in Matlab, use command "matlab –r filename"
To run the file filename.m in Octave, use command "octave –q filename.m"

Commenting

```
a = 1; % This is how you write a line comment.
% A line comment is ignored from the "%" to the end of the line.
%{
a = a + 1;
disp('These two lines are ignored');
%}
% That was a block comment, created with "%{" and "%}" on otherwise empty lines.
disp(a); % Because of the block comment, this displays "1".
```

Variables, Assignments, and Access

Variables are named locations in the computer memory that store values. The names can be any sequence of letters, numbers, and underscores(_), beginning with a letter. One assigns a variable by placing the variable name to left of an equal (=) and an expression to evaluate to the right of the equal. The value evaluated is then placed in the named location. Example:

```
a = 42; % If this semicolon is omitted, the value is printed.
variable_2 = 12;
a = 0; % The value 42 has been replaced by 0.
a = a + 1; % The right hand side now has value 1, so this increments a.
```

Input/Output and String Concatenation

```
disp('Enter two numbers to add.') % display a string of characters
a = input('a? '); % evaluate a typed expression after prompt and assign to a
bStr = input('b? ', 's'); % read a string into bStr
b = sscanf(bStr, '%d'); % scan the string for a decimal integer and assign it to b
fprintf('a + b = '); % a formatted print of a string (with no formatting)
disp(a + b); % display the sum of a and b
fprintf('%s as %d.\n', 'Easy', 123);
% formatted printing with string (%s) and decimal integer (%d)
name = input('Name? ', 's'); % input string name

% Ways to greet:
disp(['1. Hello, ', name, '!']); % display (horizontal) concatenation of strings
fprintf('2. Hello, %s!\n', name); % formatted print of the same
fprintf(1, '3. Hello, %s!\n', name); % formatted print to standard output
fprintf(2, '4. Hello, %s!\n', name); % formatted print to standard error output
greeting = sprintf('5. Hello, %s!\n', name); % print to string
disp(greeting); % display that greeting string
```

Notation

Expressions in MATLAB can be in infix notation (e.g. `2 + 3`) or prefix/functional notation (e.g. `plus(2, 3)`).

Strings

```
'A string of letters is enclosed by single quotes.'
'It''s easy to make an single quote in a string by doubling it.'
s = '7 * 6' % ans = 7 * 6
```

```
length(s) % ans = 5 (string length)
eval(s) % ans = 42 ('7 * 6' evaluated)
s(3) % ans = * (the 3rd character)
state = 'Ohio';
state(2:3) % ans = hi (substring of characters 2 - 3)
```

Vectors and Plotting

```
one_through_ten = 1:10
jenny = [8, 6, 7, 5, 3, 0, 9]; % commas optional
jenny(1) % ans = 8
jenny(7) % ans = 9
% A string is just a vector.
s = 'hi';
s(1) + 0 % shows that 'h' is represented internally as 104
v = [104, 105];
char(v) % ans = hi
x = 0:.01:2*pi; % a vector of numbers from 0 to 2*pi in steps of .01
y = sin(x); % a vector of the sine function applied to each x value
plot(x, y); % a plot of x versus y
v1 = [1 2]
v2 = [3 4]
v3 = [v1 v2] % appended vectors: 1 2 3 4
empty_vector = []
[v1 empty_vector] % same as v1
% A length N vector is really a 1xN matrix.
% A scalar (single number) = a length 1 vector = a 1x1 matrix.

% Many operators work on both scalars and vectors.
a = [1 2 3];
b = [4 5 6];
a + b
a - b
b' % transpose of b (rows/columns become columns/rows)
a * b' % dot product - sum of element products
dot(a, b) % same
a .* b % element products
a ./ b % element quotients

% Other operators work on vectors to compute elementary statistics
data = [1 1 2 3 5]
fprintf('N = %d\n', length(data))
fprintf('sum = %d\n', sum(data))
fprintf('mean = %d\n', mean(data))
fprintf('standard deviation = %d\n', std(data))
fprintf('maximum = %d\n', max(data))
fprintf('minimum = %d\n', min(data))

find(data == 1) % true at positions 1 and 2, returns [1 2]
data(1) % first element
data(end) % last element
data(end + 1) = 8;  % add an element 8 to the end of data
data
```

Booleans

A Boolean value is a true/false value. In Matlab, like the C programming language, 0 represents false and all other non-zero values are treated as true.

```
b1 = [0 0 1 1];
b2 = [0 1 0 1];
~b1 % ~ means "not"
b1 | b2 % | means "or"
b1 & b2 % & means "and"
xor(b1, b2) % xor is exclusive or: "either this or that but not both"
any(b1) % true if any value is true
all(b1) % true if all values are true
values = [0 1 3 6]
~values % [1 0 0 0]
any(values > 2) % => any([0 0 1 1]) => 1
all(values <= 10) % => all([1 1 1 1]) => 1
any(values == 3) % == equal to (not assignment)
all(values ~= 3) % ~= no equal to
rand(1, 10) > .5
```

Matrices

```
% Matrices may be predefined ...
magicSquare = magic(3)
allOnes = ones(3)
% ... or manually defined:
m = [1, 2, 3; 4, 5, 6; 6, 8, 9]
% Many operators above will work with matrices as well:
m + 1
m .* 2
m .^ 3
sum(m) % sums columns
sum(m') % sums rows
m(1:2, 2:3) % extract the first two rows and the last two columns
m(1:2, :) % extract the first two rows and all columns
```

Mixed Data

```
% Vectors and matrices may only hold one type of information.
% Structures allow us to store mixed data (e.g. numbers and strings)
% together in the same data structure.
people = struct
people.name = 'Fred';
people.yearOfBirth = 1960;
people.sex = 'm';
people

% We can also create an array of structures:
people(2).name = 'Wilma';
people(2).yearOfBirth = 1960;
people(2).sex = 'f';
people % ans = 1x2 struct array with field: name yearOfBirth sex
people(2) % ans = (whole Wilma struct)
people(2).name % ans = Wilma
```

Functions

Functions are units of code that (1) take input parameters, (2) perform computations on those parameters, and (3) return output value(s). Suppose we wish to create a function "linear" that takes in vector x, slope m, and y-intercept b, and returns a vector y such that y = mx+b? In file linear.m, we would code:

```
function y = linear(x, m, b)
    y = m*x + b;
end
```

Note the form:

```
function <return value(s)> = linear(<parameter 1>, …, <parameter n>)
    <computation that assigns the variables in <return value(s)>>
end
```

Similarly, one might create a quadratic function (y = a*x^2 + b*x + c) as follows:

```
function y = quadratic(x, a, b, c)
    y = a*x.^2 + b*x + c
end
```

This must appear in a file quadratic.m. Functions can return more than one value. Consider the quadratic equation where one computed the roots (i.e. the x values for which y is 0) according to (-b +- sqrt(b^2 – 4ac))/(2a). There are two potential roots that we can place in a return vector:

```
function [root1, root2] = quadroots(a, b, c)
    root1 = (-b - sqrt(b * b - 4 * a * c)) / (2 * a);
    root2 = (-b + sqrt(b * b - 4 * a * c)) / (2 * a);
end
```

Scripts

Now suppose that one wishes to store computations in a file with no input parameters or output return values. One can simply also store these in m-file (i.e. a file ending with extension .m). You can make sure you're not using a reserved/predefined function name using the command "help <name>" where <name> is the function name you are considering. Here are the contents of quadplot.m that uses some of the above functions:

```
% Plot a quadratic with its roots
x = -5:.1:5
y = quadratic(x, 2, 3, -5)
plot(x, y)
hold on
plot([-5 5],[0 0])
plot([x1 x1], [-10 60])
plot([x2 x2], [-10 60])
```

Inline Functions

One can make an unnamed "inline" function that can be stored in a variable and applied using feval at a later time:

```
square = inline('x .* x', 'x')
```

```
feval(square, 1:5)
```

Further, one can create an return an unnamed “inline” function from another function. Thus, we can write functions that can create functions. In this example, we create a sinusoidal function with a given amplitude, frequency, and phase shift:

```
function f = makesine(amplitude, frequency, phase)
    f = inline([num2str(amplitude),' * sin(', num2str(frequency), ' * (x + ',
num2str(phase), '))'], 'x');
end
```

(num2str converts numeric values to strings.)
The following script demonstrates the use of the makesine function:

```
t = 1:.01:10;
s1 = makesine(10, 6, .3)
s2 = makesine(5, 7, -.2)
plot(t, feval(s1, t))
hold on % keeps the previous plot in place when plotting new data
plot(t, feval(s2, t))
hold off
pause % causes program to wait for the user to press a key
plot(feval(s1, t), feval(s2, t))
```

Decisions

if - When you want code to execute conditionally for a single case:

```
x = input('x? ');

if x > 0
    fprintf('positive\n');
end
```

if/else - When you want code to execute conditionally for a two cases:

```
if x > 0
    fprintf('positive\n');
else
    fprintf('nonpositive\n');
end
```

if/elseif/else chain - When you want code to execute conditionally for more than two cases:

```
if x > 0
    fprintf('positive\n');
elseif x == 0
    fprintf('zero\n');
else
    fprintf('negative\n')
end
```

nested ifs - When you want cases to have subcases:

```
y = input('y? ');

if x > 0
    if y == 0
        fprintf('positive x axis\n');
    elseif y > 0
        fprintf('quadrant I\n');
    else
        fprintf('quadrant IV\n');
    end
elseif x == 0
    if y == 0
        fprintf('origin\n');
    elseif y > 0
        fprintf('positive y axis\n');
    else
        fprintf('negative y axis\n');
    end
else % Note: x < 0
    if y == 0
        fprintf('negative x axis\n');
    elseif y > 0
        fprintf('quadrant II\n');
    else
        fprintf('quadrant III\n');
    end
end
```

The “switch” statement allows one to choose cases based on whether a value matches a target value or values:

```
month = input('Month? ', 's');
switch lower(month)
    case {'sep', 'september', 'apr', 'april', 'jun', 'june', 'nov', 'november'}
        days = 30;
    case {'feb', 'february'}
        days = 28; % except leap year!
    otherwise
        days = 31;
end
fprintf('%s has %d days.\n', month, days);
```

Iteration (Loops)

The while loop is a good general loop that (1) checks if a condition is true, (2a) executes a block of code and repeats (1) if so, or (2b) continues execution after the loop if not.

```
while strcmp(input('Are we there yet? ', 's'), 'no')
    display('5 minutes pass...');
end
display('Yea! We''re here!');
```

The for loop is a specially suited to iterating through a set of values, performing a given computation for each value. The loop control variable (“time” below) is assigned the first value, the body of the loop executes using the

loop control variable, the second value is assigned to the loop control variable, the body executes again, and this process continues until all values after the "=" have been assigned to the loop control variable with body execution.

```
for time = 5:-1:1
    fprintf('%d...\n', time)
    pause(1)
end
fprintf('Launch!\n')
```

One can break out of a while loop at any time using a `break` statement. If the `break` occurs within nested loops, it only breaks out of the outermost loop. Here, we use `break` to terminate user input as we collect unique nonnegative integers and return them in a vector:

```
function res = uniqueInput()
    res = [];
    while 1
        i = input('Enter a nonnegative integer (-1 to quit): ');
        if (i == -1)
            break;
        end
        if ~ismember(i, res)
            res(end + 1) = i;
        end
    end
end
```

Note that in we did not know how large the return vector would be in advance. If we want to iterate through the elements of a vector/matrix and ensure the return value has the same dimensions, we can initialize the result matrix to be a matrix of zeros of the same size, and use single-indexing of both to iterate through the elements:

```
function res = myAbs(mat)
    res = zeros(size(mat));
    for i = 1:prod(size(mat))
        res(i) = mat(i);
        if res(i) < 0
            res(i) = - res(i);
        end
    end
end
```

Variable Environments and Scope

In its simplest form, an environment is a mapping of variable names to values. Scripts work with a global environment. Functions create and use a local environment create for each function call and destroyed thereafter. These files illustrate the environment behaviors of scripts and functions:

Script envTest.m:

```
a = 1;
```

```
b = 2;
fprintf('initially in envTest: a = %d, b = %d\n', a, b);
% Scripts can use and add to the global environment.
envTestScript; % This script uses a and b and defines c.
fprintf('after envTestScript: a = %d, b = %d, c = %d\n', a, b, c);
% Functions have local private variable environments.
envTestFunction(); % Creates local values for a and b. Cannot use c.
% Functions do not directly use or add to the global environment.
% Thus no changes are evident here:
fprintf('after envTestFunction: a = %d, b = %d, c = %d\n', a, b, c);
```

Script envTestScript.m:

```
a = 3;
c = 4;
fprintf('within envTestScript: a = %d, b = %d, c = %d\n', a, b, c);
```

Function envTestFunction.m:

```
function envTestFunction()
    a = 5;
    b = 6;
    fprintf('inside envTestFunction: a = %d, b = %d\n', a, b);
    % This next line would cause an error if uncommented:
    % fprintf('Global variable c = %d does not exist in the local function
scope.\n', c);
end
```

Printed output from running envTest.m:

```
initially in envTest: a = 1, b = 2
within envTestScript: a = 3, b = 2, c = 4
after envTestScript: a = 3, b = 2, c = 4
inside envTestFunction: a = 5, b = 6
after envTestFunction: a = 3, b = 2, c = 4
```

Debugging: In Matlab and Octave, it is possible to debug by setting “breakpoints” in your program where execution halts, and you can manually step execution, inspecting the changes to the environment (e.g. variable values).

Here is a summary of commands:

- dbstop – to set a break point in line 1 of envTest.m in Matlab, enter “dbstop in envTest at 1”. In Octave, it is ‘dbstop(“envTest”)’. Having done this, now type “envTest;” to reach that breakpoint.
- dbstatus – lists your breakpoint information. Try this.
- dbstep – execute a step of the code in your current function/script. Do this twice. (You can repeat the previous command quickly by pressing the up arrow and enter.)
- who – list the variable values of your current environment. Try this, and then do “dbstep” four more times. Notice that the stepping did not enter into the “envTestScript” line-by-line execution. Essentially, it stepped until the next line (the script call) was completed.

- dbstep in – execute a step of the code, or any code called by the code. Try this three times. Then type "who" to see the local variable environment
- dbstack – show the current call stack for execution. Try this.
- dbup – move to the previous call environment. Try this and type "who" to note the surrounding environment.
- dbdown - undo dbup. Do this.
- dbcont – continue execution until the next breakpoint or end of program. Do this and observe the completion of the program.

There are graphical buttons for aiding in this process, but these text commands will work in both Matlab and Octave.