AMS 595.03 MATLAB Part 3

Table of Contents

Author: Oliver Yang

For further reading, Chs. 3, 4-5 of *Attoway* and Chs. 4, 6-7 of *Lockhart/Tilleson* also cover some of the topics of this lecture.

Scripts

The Editor

If the Editor is not already open, then it can be opened with the `edit` function in the Command Window:

```
edit
```

A file's name, with or without the .m extension, can also be passed as an input to the `edit` function to open that file:

```
edit( 'example')
edit( 'example.m')
```

If a file with that name does not yet exist, then MATLAB will ask you whether to create it.

Another way that scripts can be created is by selecting one or more commands in the Command History, right-clicking them, and selecting Create Script.

Using the Editor allows you to take advantage of MATLAB's keyboard shortcuts, or **key bindings**. They make typing code much faster, since you do not need to take time to move your fingers from the keyboard to the mouse or trackpad and back. This will also reduce the stress on your fingers caused by using a computer.

- CMD + S saves the file.
- CMD + Z undoes the last action.
- CMD + SHIFT + Z redoes the last action.
- CMD + F opens the Find (& Replace) dialog box, which can be found in the Navigate section of the Toolstrip.

These actions may also be found in the Edit section of the Toolstrip:

- CMD + / comments all highlighted lines.

- CMD + T decomments highlighted lines. (Most other code editors use CMD + / to toggle commenting, but MATLAB is different.)
- CMD + SHIFT + W wraps comments in all highlighted lines.
- CMD +] indents and CMD + [deindents all highlighted lines.

The Linter

Notice that there is a green square at the top of the scrollbar on the right. It the MATLAB's **linter**. Most modern code editors (e.g. Atom and Visual Studio Editor) have linters. When you start typing text in the Editor, the square becomes white. It means that the linter is analyzing your code as you are typing it. If you type a statement that will execute without error, but there are things that will cause slow performance, or MATLAB thinks that the code you typed is not what you actually meant, then MATLAB will give an warning. Enter this code into the Editor:

```
a = 1
```

The linter square turns orange. Clicking on it will bring you to the line and character in the file that triggered the warning. You can also see descriptions of all the warnings by hovering your mouse pointer over dashes in the scrollbar. Clicking the Details button gives you a more detailed description and suggested actions. In some cases, you can click Fix to have MATLAB fix the problem for you. The relevant characters will also be underlined in orange and the description of the warning will popup when you hover your mouser pointer over them.

Note that not all warnings indicate mistakes in your code. Your code could be correct and express exactly what you want, and there may still be warnings.

Errors, on the other hand, prevent the code from being executed and definitely indicate mistakes in the code. They are indicated in red by the linter. For example, replace the text in the Editor with this:

```
1a = 1;
```

Compilation

In traditional programming languages like C and Fortran, you must compile all your source files into **object files**, **link** them into an executable or **library**, and only then can you run your program. (Usually this process is **makefiles**. You should have Wenbin teach you to use the *Make* utility, as well as a tool called *Automake*, with C++. The CSE department has an entire graduate course on compilers (CSE 504) and Wenbin's group actually works on compilers.)

You will notice that we did not do any compiling or linking to run our script. This is because MATLAB is an **interpreted** language. What MATLAB is doing is it is actually continuously compiling some C code in the background. So to write efficient MATLAB code, you often have to think about what is happening in the background in C.

MATLAB source code is code written in the MATLAB language. MATLAB source files have the extension .m. A **script** is an M file with a collection of statements. Executing the script simply tells MATLAB to execute the statements in order. A set of statements that you want to execute repeatedly can be conveniently saved in a script. Scripts, therefore offer a way to modularize your code, that is, to separate functionally different parts of your code.

First, let's create two variables in the Workspace by entering the following in the Command Window:

```
a = 1;
b = 2;
```

Then save the following code in our `example` script:

```
a
b = 3;
c = 4;
```

MATLAB identifies scripts by their filenames (without the .m extension). Scripts are executed by clicking Run in the Run section of the Toolscript, or typing the name of the script in the Command Window:

```
example
```

Note that scripts run from the Command Window access variables in the Workspace. Changes to existing Workspace variables persist after the script terminates. Any new variables defined in the script also remain in the Workspace. Any warnings and errors will be displayed in the Command Window and link back to the lines and characters that triggered them.

To begin a new section in a script, have a line starting with `%%`, and then the title following a space:

```
a = 1;
%% Another Section
b = 2;
```

Sections allow different parts of the script to be executed separately. To execute the section in which the cursor is currently located, click Run Section. To move to the next section, click Advance. To do both, click Run and Advance.

Scripts can also be called from other scripts instead of from the Command Window. Let's save the following code in a file named 'increment.m':

```
a = a + 1;
```

Then replace the code in the `example` script with the following:

```
a = 0;
increment;
increment;
```

Now execute `example` again. Note that both scripts use the same workspace.

The Terminal

When running MATLAB on a cluster, for example, it is often helpful to disable the MATLAB GUI (graphical user interface, pronounced "GOO-ee") and run MATLAB in a terminal to execute scripts. To do this on Linux, enter the following command in a terminal:

```
matlab -nojvm -nodisplay -nosplash
```

The `nojvm` option disables the Java Virtual Machine, since the GUI uses Java. The `nodesktop` option does not display the MATLAB desktop. The `nosplash` option does not display the splash screen.

On Mac, the MATLAB binaries folder may not be on the path, so you should enter:

```
/Applications/MATLAB_R2018a.app/bin/matlab -nojvm -nodisplay -nosplash
```

As far as I know, it is not possible to run a terminal session of MATLAB on Windows.

To exit from the terminal session of MATLAB, use the `exit` function:

```
exit
```

The `quit` function does the same thing:

```
quit
```

In MATLAB, you can issue shell commands through a character array input argument to the `system` function. This is equivalent to `os.system` or `subprocess` calls in Python. On Mac or Linux, use the `ls` command to display the contents of the present working directory:

```
system( 'ls')
```

```
Apps   TestMeshes  multiply.m  test
Core   Vis   pkgs   util
IO   bin   projroot.m
ParaCoder  coding_standard.txt startup.m
README   license.txt  startup_numgeom.m
ans =
     0
```

It returns a variable indicating the exit status. 0 indicates that the command executed without errors. In the C ++ unit, you should have Wenbin teach you Linux commands.

On Windows, the command is `dir`:

```
system( 'dir')
```

```
/bin/bash: dir: command not found
ans =
   127
```

The `unix` and `dos` functions also exist for a particular operating system:

```
unix( 'ls')
```

```
Apps   TestMeshes  multiply.m  test
Core   Vis   pkgs   util
IO   bin   projroot.m
ParaCoder  coding_standard.txt startup.m
README   license.txt  startup_numgeom.m
ans =
     0
```

```
dos( 'dir')
```

```
/bin/bash: dir: command not found
ans =
   127
```

An alternative is to use the bang symbol, followed by the command:

```
!ls
```

```
Apps   TestMeshes  multiply.m  test
Core   Vis   pkgs   util
IO   bin   projroot.m
ParaCoder  coding_standard.txt startup.m
README   license.txt  startup_numgeom.m
```

```
!dir
```

```
/bin/bash: dir: command not found
```

Debugging

Programming workflows usually involve the following steps:

1. planning the structure of your code (e.g. deciding which data structures and algorithms to use for runtime and memory-efficiency, how to modularize your code into classes, functions, modules, and packages),
2. actually writing and documenting the code (hopefully in a style that is elegant and easy for others to understand), and
3. testing and debugging it.

All code that you write needs to be tested. If you do not test your code, you cannot be sure that it actually does what you intend it to. Typically, you start by testing on simple cases and then move on to more complicated ones. Ideally, you should all the cases in which your code might be used.

If your code produces errors when you test it, or it produces the wrong results, it means that there are bugs in your code, and you need to debug it.

Debugging usually takes the most amount of time. The largest amount of code I have ever written without a single bug was only about 200 lines. Every time you add code to your program or change some code that you have written (however small), you have potentially introduced some bugs, and you need to retest it.

You should also test every independent piece of your code (e.g. every function) separately. This is called **unit testing**. It allows you to isolate bugs in your code. This is another good reason to modularize your code. What is not advisable to do is write ten functions that call each other and then test the entire program. Then it becomes more difficult to find exactly where your computations went wrong.

Using a debugger allows you to stop the execution of your program at a certain line of code and view all the current values of the current variables. You can advance the program until another line of code, and so on. By testing a simple example that you can check by hand, the debugger will help you determine the point at which your computations go wrong.

To pause execution of the program before a specified line of code, set a breakpoint at that line. This is done by clicking the dash to the right of the line number, putting the cursor on the line and clicking Breakpoints > Set/Clear in the Breakpoints section of the Toolstrip, or typing CMD + \. A red dot should appear instead of the dash.

For example, set breakpoints at each line of code of the following:

```
a = 1;
b = 2;
c = 3;
d = 4;
```

When you click Run, the **control flow**, which is indicated by a green arrow to the right of the red dot, stops before the first line, at the first breakpoint. Run has now been replaced by Continue. When you click Continue, the first line is executed and the control flow stops before the second line, since the next breakpoint there. As you click Continue further, the control flow moves on further breakpoints, until the script terminates.

Alternatively, if you set a breakpoint only at the first line, then you can click Step to follow each further step of the control flow. You can put the cursor on, say, the fourth line and click Run to Cursor to execute statements until the fourth line.

To stop debugging, click Quit Debugging in the Debug section of the Toolstrip. To clear all breakpoints, so that files execute normally, click Breakpoints > Clear All.

In the case of `example` calling `increment` above, if you set a breakpoint at the second line of `example` and step, the debugger does not enter the `increment` script. To enter the `increment` script, you must click Step In. Now, back in `example`, there is a white arrow signaling where the control flow exited the `example` script. Alternatively, if you put the breakpoint in `increment` and run `example`, the debugger pauses within `increment`.

In the C++ unit, Wenbin will teach you to use command-line debugger *GDB* (GNU Debugger) and its GUI counterpart *DDD* (Data Display Debugger) in *GCC* (the GNU Compiler Collection).

How do you know when your code is free of bugs? The answer is that you are never completely sure. Every sufficiently large piece of software, e.g. an operating system, will have bugs hidden somewhere, yet undiscovered. Even compilers will sometimes also have bugs. For example, I once discovered a compiler bug in gcc. The only thing that you can do is keep on testing your code.

Control Structures

For example, sometimes you would like your program to decide what to execute based on a certain condition. For this, you need a **conditional** statement.

The `if` Statement

One type of conditional statement is the **`if` statement**. Let's enter this code into the editor and save it as a file 'conditional.m':

```
if a == int32( 0)
    disp( 'a exists.');
    disp( 'Moreover, it is 0.');
end % if
```

The statement starts with the keyword `if`, following by a logical expression. If the expression (condition) is true, then the collection of statements following the condition is executed. It is good style to start the collection of statements on the next line and indent them. (But note that lack of indentation does not change the meaning of the code! By default, MATLAB inserts 4 spaces for tabs or indentation. Some people like to use 2 spaces. This can be changed by going to Preferences > MATLAB > Editor/Debugger > Tab in the Environment section of the Toolstrip.) The collection of statements is ended by the keyword `end`. It is good style to put the `end` on a new line, with the same indentation as its corresponding `if`. I like to add an inline comment after the end describing the conditional statement, since in very long code, it can be difficult to tell what statement is closes.

The `disp` function is the simply displays the input. It is the simplest way to display variables.

```
a = int32( 0)
conditional
a = int32( 1)
conditional
```

Note that nothing is displayed if `a` is 1, since it does not match any condition.

If the original condition is not satisfied, an `if` statement can execute another collection of statements with the keyword `else`. Let's change the code in conditional.m to the following:

```
a = input( 'Enter a number: ');
if (a < int32( 0)) && (abs( a) == int32( 1))
    disp( 'The number entered is -1.');
else
    disp( 'The number entered is not -1.');
end % if
```

The `input` function allows for simple I/O in the Command Window. Note that multiple relational operators may be used as long as they evaluate to a single expression. The `else` takes the place of the `end`, and there is no condition after it. This is often called an `if-else` statement. If the original condition was false, then the second collection of statements after the `else` is executed:

```
conditional
-1
conditional
1
```

Note that MATLAB displays "Waiting for input" in the status bar after the script is executed.

One of the statements after a condition can itself be an `if-else` statement. That is, an `if-else` statement can be **nested** inside another one:

```
if a < int32( 0)
    disp( 'a exists.');
    disp( 'Morevoer, it is negative.');
    if a <= int32( -2)
        disp( 'Moreover, its magnitude is at least 2.');
    else
        disp( 'Moreover, its magnitude is less than 2.');
    end
else
    disp( 'a exists.');
    disp( 'Morevoer, it is non-negative.');
    if a >= int32( 2)
        disp( 'Moreover, its magnitude is at least 2.');
    else
        disp( 'Moreover, its magnitude is less than 2.');
    end
end % if
```

```
a = int32( -2)
conditional
a = int32( -1)
conditional
a = int32( 0)
conditional
a = int32( 1)
conditional
```

This is an example of doing simple tests of all the cases in which the code may be used.

If the original condition is not satisfied, an `if` statement can also check for a collection of other conditions with the keyword `elseif`:

```
if a < int32( 0)
    disp( 'a exists.');
    disp( 'Moreover, it less than 0.');
elseif a == int32( 0)
    disp( 'a exists.');
    disp( 'Moreover, it is 0.');
elseif a < int32( 2)
    disp( 'a exists.');
    disp( 'Moreover, it is between 0 and 2.');
else
    disp( 'a exists.');
    disp( 'Moreover, it is greater than or equal to 2.');
end % if
```

After the first collection of statements, `elseif` is used instead of `else` to check whether the second condition is true. If not, then the next `elseif` condition is checked. Note from the third condition that later conditions may overlap with earlier conditions. Only after all the elseif conditions, may there be a condition following else. If all the `elseif` conditions are false, then the else condition (if it exists) is checked. Another way to think about it is: MATLAB executes the collection of statements associated with the first true condition:

```
a = int32( -1)
conditional
a = int32( 0)
conditional
```

```
a = int32( 1)
conditional
a = int32( 2)
conditional
```

The `switch` Statement

If all the conditions to be checked are equality conditions, then the other type of conditional statement, a `switch` statement, can be used:

```
switch a
    case int32( 0)
        disp( 'a exists.');
        disp( 'Moreover, it is 0.');
    case int32( 1)
        disp( 'a exists.');
        disp( 'Moreover, it is 1.');
    otherwise
        disp( 'a exists.');
        disp( 'Moreover, it is neither 0 nor 1.');
end % switch
```

The statement starts with the `switch` keyword, followed by the variable to be compared. Instead of `if`, the `switch` statement then uses the keyword `case`, following by the expression to which the variable is to be compared. Again, it is good practice to indent `case` on a new line, but it does not affect the meaning of the code. More `case`s can be used, just like `elseif`s; the meaning is the same. Finally, instead of `else`, the `otherwise` keyword is used.

```
a = int32( 0)
conditional
a = int32( 1)
conditional
a = int32( 2)
conditional
```

Any `switch` statement can be written as an `if` statement, but not vice-versa. For example, the previous `if-elseif` statement could not be written as a `switch` statement.

The `for` Loop

Often, you want your program to execute a set of statements for a number of different cases, where the statements change in a prescribed way for each case. For this, you need a **`for` loop**.

Let's say that I have a column double vector of zeroes of length 5, and I want to set each entry equal to its index. Let's enter this code into the editor and save it as a file 'loop.m':

```
a = coder.nullcopy( zeros( 5, 1));
for i = 1 : size( a, 1)
    a( i) = i;
end
a
```

```
a = 5×1
     1
     2
     3
     4
     5
```

Note that I did not initialize the array since the entries will be assigned later. The loop starts with the `for` keyword, followed by the name of the loop index. It's traditional to use the letters i, j, and k for loop indices. The loop index is assigned a vector of values using the colon notation. (Note that the colon notation is easier to use, but is not necessary. It could easily have been a column or row vector.) After the assignment follows the collection of statements to be executed for each value of the loop index (which usually uses the value of the loop index). Again, it is good style to start the collection of statements on the next line and indent them. As with conditionals, the `for` loop terminates with the `end` keyword.

Note that setting a breakpoint at the line with `for` and stepping allows you to step through every iteration of the loop.

As with conditionals, loops are often nested inside other loops. This code sets each entry of a 3×3 matrix to the product of its row index and column index:

```
a = coder.nullcopy( zeros( 3, 3));
for i = 1 : size( a, 1)
    for j = 1 : size( a, 2)
        a( i, j) = i * j;
    end
end
a
```

```
a = 3×3
     1     2     3
     2     4     6
     3     6     9
```

As mentioned before, the colon notation can be used to create loop indices that are evenly spaced. This code sets each entry of a vector equal to its index only for odd indices:

```
a = zeros( 5, 1);
for i = 1 : 2 : size( a, 1)
    a( i) = i;
end
a
```

```
a = 5×1
     1
     0
     3
     0
     5
```

The `continue` keyword allows you to exit the current iteration of a loop and begin the next iteration. Statements appearing in the loop after `continue` are not executed. This code counts the number of odd integers from 1 to 5:

```
count = int32( 0);
for i = 1 : 5
    if mod( i, 2) == 0
        continue
    end
    count = count + 1;
end
count
```

```
count = int32

   3
```

The variable `count` is called a **counter**. If the current integer is even, then the counter increment statement is skipped. Note that in this case, the value to which `count` is set outside the loop matters -- it must be set to 0.

Note that if `continue` appears in a nested loop, it exits the current iteration of the immediate (inner) loop, and not the outer loop. This code counts the number of odd integers in each row of a 3×3 matrix:

```
A = [1, 2, 3; 4, 5, 6; 7, 8, 9]
```

```
A = 3×3
     1     2     3
     4     5     6
     7     8     9
```

```
count = zeros( 3, 1, 'int32');
for i = 1 : size( A, 1)
    for j = 1 : size( A, 2)
        if mod( A( i, j), 2) == 0
            continue
        end
        count( i) = count( i) + 1;
    end
end
count
```

```
count = 3×1 int32 column vector
   2
   1
   2
```

The `break` keyword allows you to break out of the immediate loop. It can be useful for avoiding unnecessary computation. This code sums the elements of a vector, but breaks when a non-finite number is encountered:

```
a = [1; 2; Inf; 3; NaN]
```

```
a = 5×1
     1
     2
   Inf
     3
```

```
    NaN
```

```
sum = 0;
for i = 1 : length( a)
    if ~isfinite( a( i))
        break
    end
    sum = sum + a( i);
end
sum
```

```
sum =
     3
```

In this case, the subsequent entries of the vector are irrelevant.

With loops, you should take care that new arrays are being created within the loop. Memory allocation incurs significant overhead, particularly if it has to be done for every iteration in a loop. This code will therefore run very slowly:

```
for i = 1 : 1000
    a = zeros( i, 1);
end
```

Any **buffers** that are needed for computation within the loop should be preallocated outside. This will also be true of functions, which will discuss soon.

The `while` Loop

If you want your program to execute a set of statements whenever a certain condition is true, then you need a **`while` loop**. This code finds the smallest natural number whose square is greater than 1000:

```
a = int32( 1);
while a ^ 2 <= 1000
    a = a + 1;
end
a
```

```
a = int32

   32
```

The loop uses the `while` keyword instead of `for`, followed by the logical expression that needs to be true for the set of statements to be executed.

Note that MATLAB does not have **pre** or **post-increment** operators (e.g. `a++`) like in C++.

With `while` loops, you must be careful that the condition actually becomes false after a reasonable number of iterations. If it is never satisfied, it becomes an **infinite loop**:

```
a = int32( 1)
while a > 0
```

```
        a = a + 1;
    end
    a
```

Such infinite loops cause the program to run forever. In such cases, execution can be stopped in the Command Window by pressing CTRL + C.

Functions

Recall that the `increment` script that we wrote above accessed the variable `a` in the Workspace. Therefore, whenever we want to use the script, we always need to have a variable `a` in the Workspace. This will be inconvenient, say, when we want to increment another variable `b`.

There are other things that could be improved about scripts. The `increment` script might do some internal computations with a variable it calls `c`. If `c` already exists in the Workspace, then its value will be inadvertently overwritten, which will cause problems if the previous value is needed for other computations. So we don't want the `increment` script to be able to access any variables in the Workspace other that the ones that it needs. In fact, it would be best if the script could take a variable as an input, regardless of its name, and return its incremented value as an output. To do this, we need to write a function instead of a script.

Let's create a function that performs the same task as the `increment` script. Replace the code in 'increment.m' with the following:

```
function b = increment( a)
    b = a + 1;
end % function increment
```

The definitions of functions begin with the `function` keyword. Then, on the same line, comes the names of any output variables (`b` in our case), the assignment operator, the function name (`increment` in our case), and the names of any input variables enclosed by parentheses (`a` in our case).

MATLAB takes the filename as the function name, but it is good practice to have the same name in the function definition. MATLAB will issue a warning otherwise, but the function will still execute without error when called by the filename. Function names must satisfy the same rules as variable names. You should avoid having user-defined functions with the same names as MATLAB built-in functions. If you overload a built-in function, MATLAB will give you a warning, and will use your overloaded function in the directory in which it is located.

On the following line, usually indented, begins the body of the function, containing the statements to be executed. The end keyword after the body, usually on a separate line, is not necessary in this case, but I like to include it, followed by an inline comment with the function name, to make it clear where exactly the body of the function ends.

Let's replace the code in the `example` script with the following:

```
c = 0;
d = increment( c);
```

When executed, it does exactly what we wanted.

Workspaces, the Function Call Stack, and Stack vs. Heap

First, note that the variable names c and d in example are different than those in increment.
So a and b in increment were **dummy variables** in the sense that they only represented the input and output variables with arbitrary names. Furthermore, after the example script terminates, c and d remain in the Workspace, but there is no a or b.

Now let's set a breakpoint at the second line of example and run the script. At this point there is a variable c in the Workspace. This is the **base workspace** used by scripts. Now step into the function increment. c has disappeared and now there is a variable a. Step once and the function creates the b variable. Now step out of increment. c returns and now a and b have disappeared. Step once more and the script creates the d variable.

All of this means that the increment function uses a different workspace, separate from the base workspace. In the increment workspace, c and d do not exist. In the base workspace, a and b do not exist. If another function, say rank, were to be called within the body of increment, then rank would have yet another workspace:

```
function b = increment( a)
    b = a + 1;
    e = rank( a);
end % function increment
```

Note that in the Debug section of the Toolstrip, there is a **Function Call Stack**. You may remember from introductory computer science that **stacks** are data structures that are "first in, last out" (FILO), while **queues** are "first in, first out" (FIFO). The function at the top of the stack is the one that the control flow is currently in. Once rank terminates, it is **popped** from the function call stack (i.e. control is returned to the function that called rank) and the function at the top of the stack is again increment. a and b are called **stack variables**. Stack variables persist only for as long as their corresponding function remains in the function call stack. After that function is popped, the stack variables are deleted. c and d, on the other hand, are variables on the **heap**. They persist for the duration of the program.

The source code of MATLAB built-in functions is usually closed, i.e. hidden from the MATLAB end-user. rank is one of the few built-in functions whose source is open. Replacing rank with abs, for example, would not allow you to step into abs.

A function can invoke a script instead of another function. For example, replace the put the statement b = a + 1; from increment in a file 'add.m' and replace it with a call to add. The result is the same.

But now, set a breakpoint at the invocation of add in increment. Run example and step into add. Note that the variable a defined in increment persists when stepping into add. Therefore a script shares the same workspace as the function that called it.

A function can also call itself. This is **recursion**:

```
function n_fact = fact( n)
    if n <= int32( 1)
        n_fact = int32( 1);
    else
        n_fact = n * fact( n - 1);
    end
end
```

Let's test the factorial function:

```
fact( int32( 0))
fact( int32( 1))
fact( int32( 2))
fact( int32( 3))
```

Passing Variables into Functions

The variable `c` is initialized at the beginning of the script. To pass `c` into the `increment` function, by default, MATLAB copies the value in `c` into `a`. In this case, a is called a **temporary**. The output value is returned by copying the value of `b` into `d`. This is called **passing by value**.

Passing by value should be avoided in scientific computing, since arrays sizes can be very large. If `a`, `b`, `c`, and `d` were arrays, then every entry of `c` would have to copied into the corresponding entry of `a`, and the same for `b` and `d`. This is unnecessary. To avoid the creation of temporaries, you can **pass by reference**. In this case, MATLAB will pass the location of `c` in memory, so that the function can directly modify its values. There is no separate output variable, since all changes are made directly on the input variable. In `increment` and `example`, if the input and output variables have the same name, then MATLAB will pass by the input variable by reference:

```
function a = increment( a)
    a = a + 1;
end % function increment
```

```
c = 0;
c = increment( c);
```

Now if `example` is run, changes are made directly onto `c`.

(MATLAB, by default, actually does not quite pass by value. What it actually does is called **copy-on-write**. MATLAB only creates a temporary copy of an input variable if and when the variable is modified inside the function. But this is a subtle distinction and can be ignored on a first pass through the topic.)

Creation of stack variables other than temporaries, i.e. declaring new variables in the body of a function, such as `e` above, should also be avoided. Any memory space that is needed for storage in the function should be allocated outside the function and passed in by reference. As an example, see the WORK array passed into the LAPACK routine for SVD (singular value decomposition).

Help

Passing `increment` to the `help` function at the moment only the displays the **signature** of `increment`:

```
help increment
```

Help information for user-defined functions can be added in the commment block immediately following the function signature:

```
function b = increment( a)
% INCREMENT Increments a number by 1.
```

```
%
% b = INCREMENT( a)
%
% Inputs:
% a, numeric
%     the number to be incremented PLUS
%
% Outputs:
% b, numeric
%     the incremented number
%
% See also PLUS.

    b = a + 1;

end % function increment
```

The first line of the help block should have the function name, followed by a one-line description of what it does. This will display when the folder that contains the function file is passed to the `help` function. It is helpful to include examples of calls to the function, a description of all the input and output variables, including their classes, and any related functions. Function names should be in capitals. MATLAB will display the function itself in bold and other functions on a comment line beginning with `See also` as links to their help information. Including help information allows users to understand the **interface** of a function without knowing its **implementation**. A function with a fixed interface could have many different implementations.

Well-written comments in the source code can also double as documentation for your program. In the C++ unit, you should ask Wenbin to teach you to use a tool called *Doxygen*, which will automatically generate documentation files from the comments. This will also allow you to avoid manually changing your documentation every time your code changes.

Input and Output Arguments

Functions can take no input arguments. In such a case, the parentheses are empty:

```
function return_value = return_pi()
    return_value = pi;
end % function return_pi
```

Functions can also have no output arguments. In this case, there can be no assignment in the defintion of the function:

```
function display_pi()
    disp( ['This is the value of pi: ', num2str( pi, 16)]);
end % function display_pi
```

The third input argument in this function is optional:

```
function d = custom_sum( a, b, c)
    if nargin <= 2
        d = a + b;
    else
        d = a + b + c;
    end % if
end % function custom_sum
```

MATLAB automatically stores the number of input arguments to a function call in the variable `nargin`. MATLAB returns an error if there are not enough input arguments:

```
custom_sum( 1)
custom_sum( 1, 1)
custom_sum( 1, 1, 1)
```

Inputs to user-defined functions can also be specified through the Run pull-down in the Run section of the Toolstrip.

Multiple output variables are enclosed by square brackets and separated by commas:

```
function [c, d] = sum_and_product( a, b)
    c = a + b;
    d = a * b;
end % function sum_and_product
```

To return both output variables, execute the following:

```
[c, d] = sum_and_product( 2, 3)
```

To return only the sum (the first variable), you can treat the second output variable as optional:

```
c = sum_and_product( 2, 3)
```

To return only the product (the second variable), you must put a tilde in the place of the first one:

```
[~, d] = sum_and_product( 2, 3)
```