McGill University
School of Computer Science
Sable Research Group

McTutorial: A MATLAB Tutorial

Lei Lopez

Last updated: August 2014

Contents

Introduction

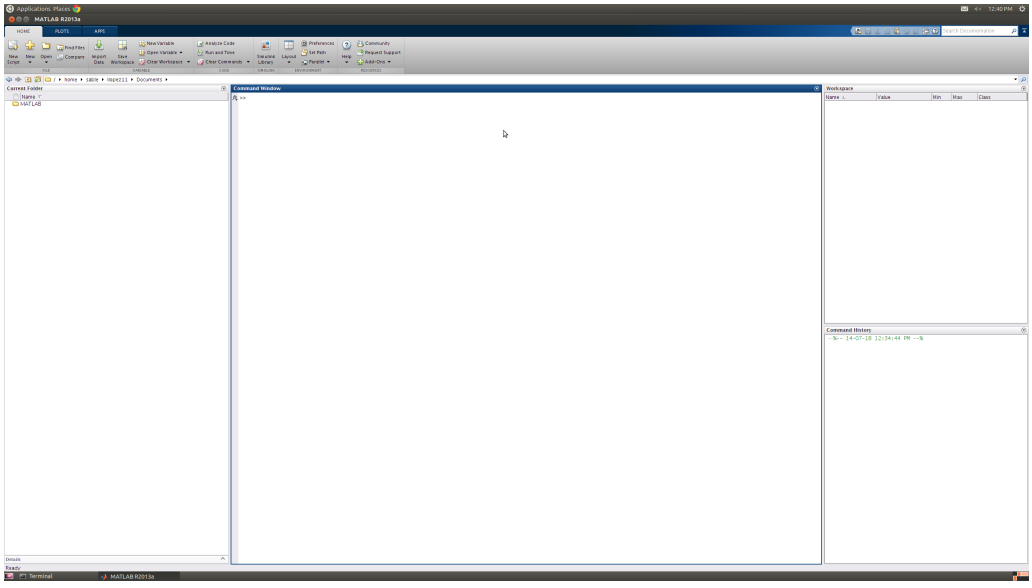
This tutorial is aimed at beginners and novices to MATLAB® who also want to be good programmers. To accomplish that, this tutorial explains many of the computer science concepts behind programming in MATLAB. It is assumed that you know basic linear algebra. While you read through this tutorial, there will be many examples. These are designed for you to emulate and play around with in your own MATLAB environment. The examples here were run in MATLAB R2013a.

Chapter 1

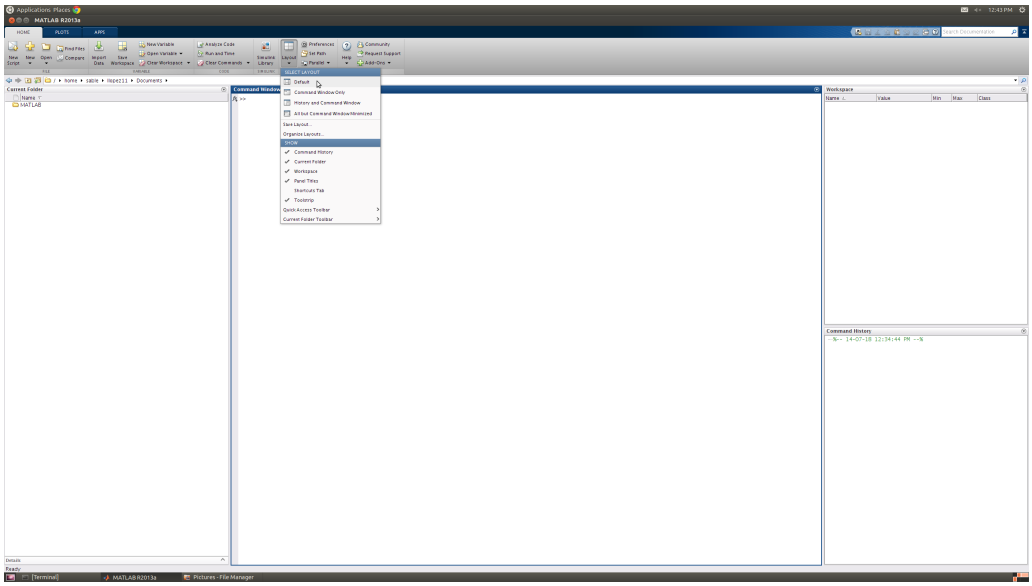
MATLAB BASICS

1.1 MATLAB User Interface

Before we get started, there are a few things you should familiarize yourself with in the MATLAB interface. This section assumes that you have opened MATLAB, and it is in the default layout, as shown below.



To make sure you are in the default layout, click the **Layout** button and select **Default**.



1.1.1 Command Window

When you open MATLAB, the section in the center is called the **Command Window**. At the top there is a symbol, >>, called the prompt. Click anywhere on the **Command Window** to make it active. The prompt should now be followed by a blinking cursor.

This window is where you will be entering commands into MATLAB. A ***command*** is an instruction that triggers a computation. For example, try typing `2+2` after the prompt, then pressing **<Enter>**. Pressing enter executes the computation.

```
>> 2 + 2
ans =
     4
```

When you press enter, MATLAB reads the input `2 + 2`. Then it evaluates the expression, and finds that the ***value*** is 4. MATLAB automatically stores this value in a ***variable*** called `ans`. We'll talk about variables in more depth later.

You'll get more practice using the **Command Window** in the the Basic Calculations section.

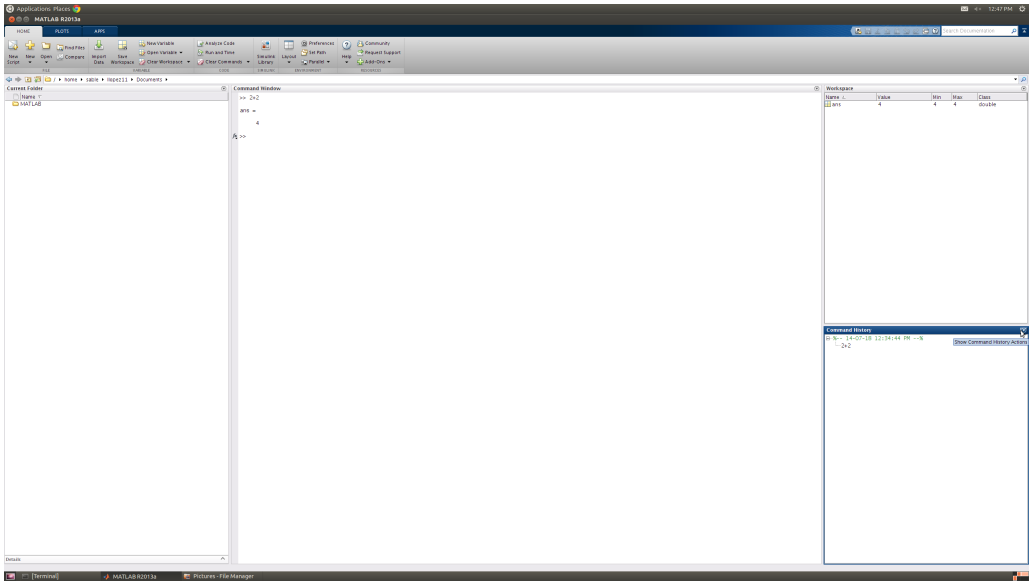
1.1.2 History

In the bottom left, there is a window called the **Command History**. This is where you can check what commands you have entered. If you have been following along, you should be able to see the command `2+2`. If you press the up arrow key (↑), you can scroll through your previous commands beside your prompt (>>). This is called smart recall.

In this case, you can execute the command `2+2` again by pressing the up arrow and pressing <Enter> to execute. If you check your **Command History** now, you will see that MATLAB does not store duplicate commands.

However, when you are scrolling through your previous commands you can use the left and right arrow keys to edit or add to parts of your original command. Once you press enter, the edited command will be saved in the **Command History**.

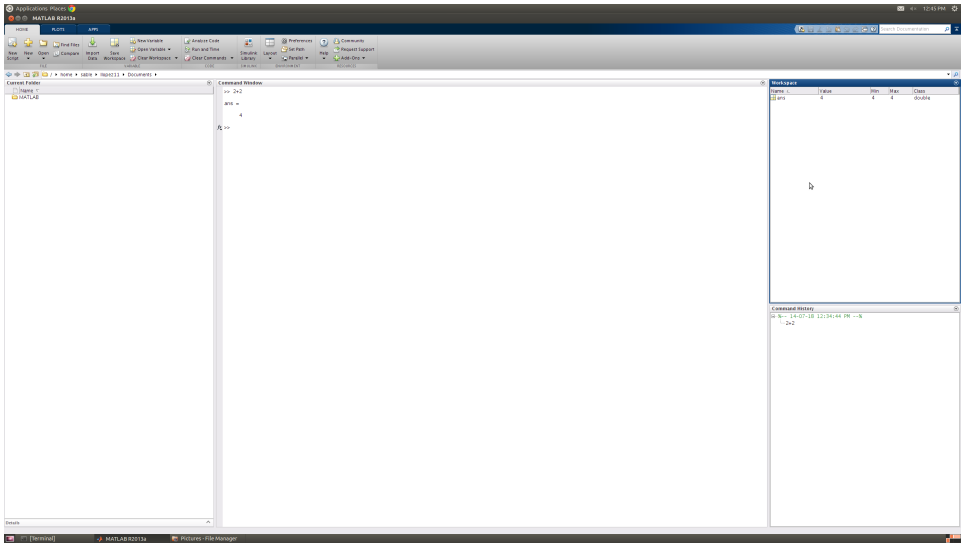
You can clear your history at any time by clicking the small arrow in the top right corner of the **Command History** frame, then selecting **Clear Command History**.



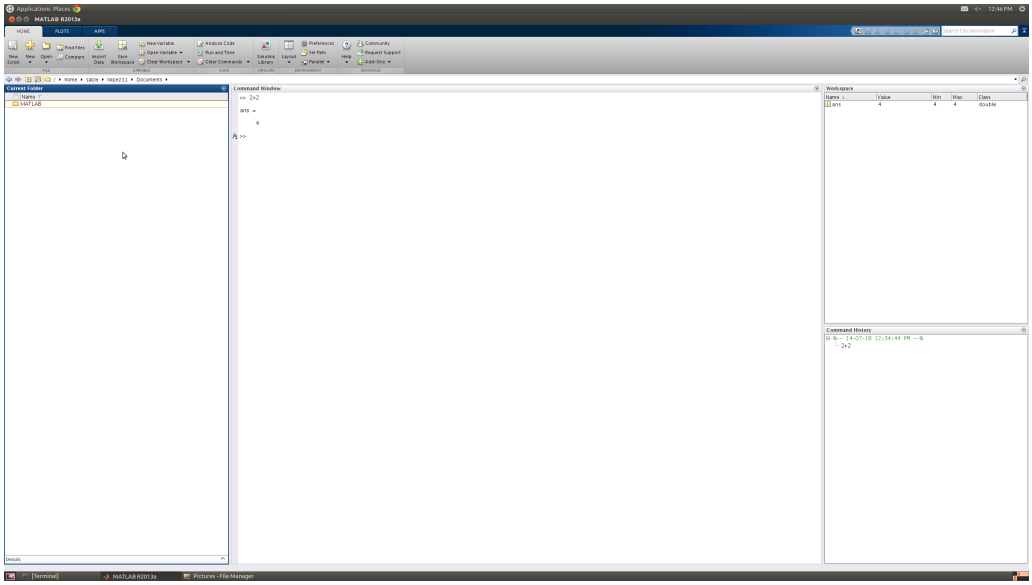
1.1.3 Workspace

To the right of the **Command Window**, there is a section called the **Workspace**. This is where MATLAB diplays the variables you have created. If you have been

following along, you should see one variable listed here (`ans`), along with other details about it. It's important to note that MATLAB stores variables somewhere in memory, and the **Workspace** just displays the variable names and what is stored in them.



1.1.4 Current Folder



At some point while you are using MATLAB, you will definitely need to use external files. This is where the **Current Folder** window comes in handy. To the left of the **Command Window**, it displays what folder you are working in. MATLAB has access to the files that are displayed in this window. We will learn how to to access files in a different folder later on.

1.2 Basic Calculations

In the last section, we practiced using the **Command Window** by executing the command `2+2`. Another name for a command in MATLAB is a statement.

Remember `ans`? It is actually the default variable of MATLAB, because if a variable is not specified, the result of the computation is stored in `ans`.

You can use your own variables to store values in. For example, let's store a value in the variable `x`.

```
>> x = 2 * 3
x =
     6
```

The equals sign (`=`) in MATLAB means ***assignment***. Specifically, an assignment is an operation that stores a value into a variable. Thus, the statement `x = 2 * 3` is called an assignment statement. The value is always obtained from the right side of the `=` sign, by evaluating whatever expression is there. The variable is on left side.

PICTURE HERE

When reading MATLAB code, it is important to read the right of the equals sign (`=`) first, since it gets evaluated, and then put into the variable.

In many other programming languages, you have to declare a variable before you use it. In MATLAB, this is not required, as you can see above.

You can also use variables you've created in computations. For example:

```
>> x - 8
ans =
     -2
>> y = ans / 2
y =
     -1
>> x = x^2 / 9
x =
```

```
     4
```

Notice that once you assign a new value to the variable `x`, this new value overwrites the previous assignment. You may have noticed this happening to the `ans` variable. Each new computation without a user-defined variable overwrites the old result.

If you ever want to delete a variable you have created, you can use the MATLAB function `clear`, followed by the variable name.

```
>> clear y
```

You can also delete all the variables you have created like this:

```
>> clear all
```

Note `clear` should only be used within the **Command Window**. When we start to code in other files, `clear` can interfere with other variables. You'll see why when we get to functions.

What is a function anyway? A function is a type of statement that takes input and does something, and can then return a result. In this case, the input is a variable name. You can find the names of all the variables you've created in the **Workspace**.

`clear` assumes that you did not create a variable named `all`, otherwise only the user-defined variable will be deleted. This is one reason why it is important to choose your variable names carefully.

More examples:

```
>> x = 3 + 1 / 2
x =
     3.5000
>> y = 10 + 4^2 * 2
y =
     42
>> z = (10+4^2)*2
z =
     52
```

From these examples you can see that MATLAB does not simply compute input from left to right. MATLAB actually follows the mathematical rules of precedence:

1. Parentheses

2. Exponents
3. Multiplication and division
4. Addition and subtraction

Within each level, MATLAB applies the operators from left to right, like so:

```
>> 6 / 2 * (1+2)
ans =
     9
```

Note that if multiplication was applied first, the answer would be 1.

```
>> 6 / (2 * (1+2))
ans =
     9
```

So far, we have only entered one command into MATLAB at a time. MATLAB allows you to input multiple commands in the same line if each command is separated by a comma.

```
>> x = 2, y = 3, z = x^2 + y
x =
     2
y =
     3
z =
     7
```

This can be useful in certain cases, but it's not recommended because it can be difficult to read.

We have been re-using the variables `x`,`y`, and `z`. Notice that most recent value is the one that is used each time.

Sometimes you may not want MATLAB to display the result. To do so, you can add a semicolon to the end of the command. This will tell MATLAB not to show you the result. This is also another way to input many commands into one line.

```
>> a = 3; b = 4; c = a/b
c =
     0.7500
```

Adding a final semi-colon will suppress the result of the third command as well.

```
>> a = 8; b = 4; c = a*b;
>>|
```

1.3 Numbers

If you are familiar with other programming languages, you'll know that different languages store numbers in different ways. In MATLAB, a number is automatically stored as a ***double***. Double is short for double-precision floating point, which is a way of storing numbers on the computer with 64 bits.

So although most of the numbers we have been using are displayed like integers, they are stored on the computer as doubles. You may have noticed that MATLAB automatically displays numbers that cannot be expressed as integers with 4 decimal points. This is MATLAB's default display. For example, MATLAB has a built-in representation of the number π.

```
>> pi
ans =
    3.1416
```

However, that is not the value of π that MATLAB stores. Since computers have a finite amount of memory space, they can only store a limited number of digits for each number. You can use the command `format long` to see how many digits MATLAB actually saves.

```
>> format long
>> pi
ans =
     3.141592653589793
```

To put it back in default mode type in `format` or more verbosely, `format short`.

Since there is a limit to how many digits of a number your computer can store, we must be careful when doing calculations. For example, MATLAB has a built-in function for calculating $sin(x)$. What happens when we try to calculate `sin(pi)` in MATLAB?

```
>> sin(pi)
ans =
     1.2246e-16
```

The answer is a tiny number, but not quite zero. It will also differ from computer to computer, based on the computer's architecture. It's important to keep this in

mind when programming in MATLAB, or any programming language, because it will affect calculations.

In addition to `pi` and `sin`, MATLAB includes many useful built-in math functions. Here are some examples (in the default format mode).

You can nest operations.

```
>> sqrt(9*4)
ans =
     12
```

It's important to remember how nested operations work: from inside to out, and from left to right. For example, in the above example, `9*4` is evaluated first, to a value of `36`. Then the sqrt function is used on the value of 36. Hence, evaluation occurs from inside to out.

e^x, where $x = 1$.

```
>> exp(1)
ans =
     2.7183
```

$ln(x)$ note that MATLAB uses log instead of ln.

```
>> log(2)
ans =
     0.6931
```

Other built-in logarithmic bases (note that you can nest commands.)

```
>> log2(log2(16))
ans =
     2
>> log10(10^5)
ans =
     5
```

Other logarithmic bases are not built into MATLAB, but you can use the trick that $log_a(x) = ln(x)/ln(a)$.

```
>> x = 64; a = 2;
>> log(x)/log(a)
ans =
    6
```

Other trigonometric functions (note that MATLAB expects radian values for these functions)

```
>> a = cos(pi)
a =
     -1
>> b = acos(a)
b =
     3.1416
>> c = tan(b/(sqrt(16))
c =
     1.0000
>> asin(c), atan(c)
ans =
     1.5708
ans =
     0.7854
```

MATLAB has a few other important values to deal with special cases.

The imaginary number i is represented by both `i` and `j`.

```
>> i
ans =
     0.0000 + 1.0000i
>> j
ans =
     0.0000 + 1.0000i
>> 4 + 2i
ans =
   4.0000 + 2.0000i
```

This is especially important to remember when you are naming your own variables. If you declare the variable `i` and assign it a new value, your value will have precedence over the built-in value of `i`. This means you won't be able to use MATLAB's value of `i`.

The `which` command allows you to figure out which value of `i` is being used.

```
>> i = 0
i =
     0
>> which i
i is a variable.
>> i
i =
```

```
     0
>> clear i
>> which i
built-in (/usr/local/pkgs/MATLAB/R2013a/toolbox/matlab/
elmat/i)
>> i
ans =
     0.0000 + 1.0000i
```

Note that the second `which i` may return a different location for the built-in, depending on your computer's file system. The important thing to note is that `i` is a built-in.

Another special value is `NaN`, which stands for not a number. MATLAB uses this value when the operation results in something that cannot be represented mathematically. MATLAB also has a special value for infinity, `Inf`. For example:

```
>> 0/0
ans =
     NaN
>> 1/0
ans =
     Inf
>> 1/Inf
ans =
     0
```

Be careful with these values, as using them may not give expected results if you use them in other expressions.

If you have been following along, you probably have many old commands filling up your Command Window. If you no longer need to view these commands, you can clear them from your screen with the command `clc`. However, this will not clear your Command History. This means you can still use smart recall with the up and down arrows to scroll through old commands.

1.4 Vectors

Up until now we have been storing single values in our variables. In MATLAB, these are called ***scalars***. A new way to store values is in a ***vector***, or as MATLAB refers to them, a one-dimensional ***array***. In MATLAB, everything is stored as an array. In fact those single-valued elements we've been using are

actually 1x1 arrays.

You can check this using the `size` command.

```
>> x = pi
x =
  3.1416
>> size(x)
ans =
     1   1
```

The `size` command gives you the number of elements in each row and each column, in that order, so in this case, the size is 1x1.

Here's how to create a row vector:

```
>> x = [1 2 3 4]
x =
     1    2    3    4
>> size(x)
ans =
     1   4
```

You can choose to use spaces or commas to separate elements in your array.

MATLAB provides many built-in operations and functions for dealing with vectors.

For example you can add or subtract two vectors of the same size:

```
>> A = [1 3 5 7]; B = [6 3 0 2];
>> C = A+B
C =
     7   6   5   9
>> D = C - [2.2 5.9 1.5 4.4]
D =
     4.8000     0.1000     3.5000     4.6000
```

What about multiplication?

```
>> A = [1 1 2 3]; B = [1 3 6 9];
>> 2*A
ans =
     2   2   4   6
>> C = A*B
Error using  *
Inner matrix dimensions must agree.
```

In the last case, MATLAB is trying to do mathematical matrix multiplication, which is not defined for two row vectors. In order to multiply the two rows element-by-element, we must use the `.*` ***array operator***.

However, the case before (`2*A`) works because scalar multiplication is defined on row vectors.

```
>> C = A.*B
C =
     1   3   12   27
```

More examples of array operations:

```
>> D = C.^2
D =
     1   9   144   729
>> D/3
ans =
     0.3333   3.0000  243.0000
>> E = [1/3 2 12 1/3];
>> F = D./E
F =
     1.0000   1.5000   4.0000   81.0000
```

You can also generate column vectors. There are a few different ways.

```
>> E = [1;2;3;4]
E =
     1
     2
     3
     4
>> size(E)
ans =
     4 1
```

The semicolons can be replaced with a ***carriage return***(pressing **<Enter>**). MATLAB will not execute the statement until there is a closing bracket.

The following is a transpose (') operation.

```
>> F = [2 4 6 8]'
F =
     2
     4
     6
```

```
     8
```

Now that we know how to create column vectors, we can try some operations with the following vectors.

```
>> a = [3 6]; b = [4 1]';
```

Remember that matrix multiplication is equivalent to calculating the dot product of two vectors or matrices.

```
>> c = a*b
c =
     18
```

Matrix division is defined by the following equations: $A/B = A * (1/B) = A * B^{-}1$.

```
>> d = a / [3 2]
d =
     1.614
```

An array operation, for comparison.

```
>> e = a ./ [3 2]
e =
     1  3
>> e^2
Error using  ^
Inputs must be a scalar and a square matrix.
To compute elementwise POWER, use the array operator
power, ^
```

As the error says, you cannot use ^ on non-square matrices, because is it not defined. In general, a period or dot (.) before an operator indicates that the operation is an elementwise operation.

```
>> e.^2
ans =
     1  9
```

If you want to ***access*** or manipulate only part of a vector, you can index into it. For example, say we want to display only the third element in a vector.

```
>> a = [1 2 8 4];
>> a(3)
```

```
ans =
     8
```

The number in the round brackets tells MATLAB that you want to access the third element of the array `a`.

Now if we want to change the third element, we re-assign its value like we would a scalar.

```
>> a(3) = 3
a =
     1   2   3   4
```

Say we want to add the value 5 to the end of the vector. We can use the same syntax to "access" and assign the fifth index of the array.

```
>> a(5) = 5
a =
     1   2   3   4   5
```

We say "access" because MATLAB actually recognizes that 5 is above the largest index in the vector `a`. Behind the scenes in memory, it resizes the vector to accommodate 5 elements, and then it assigns the fifth element the value of 5. This is a feature in MATLAB that many other languages do not support.

Now let's assign π to the 10th index of the vector.

```
>> a(10) = pi
a =
     1.0000   2.0000   3.0000   4.0000   5.0000   0   0   0   0
3.1416
```

Again, MATLAB sees that 10 is larger than the size of the vector, and resizes it appropriately. Then, MATLAB will automatically fill the non-specified elements of the vector with zeroes. Also, it will change the display of the whole vector to match π. Remember that you can change the format displayed with the `format` command.

MATLAB offers many more functions for dealing with vectors. Here are a few more.

To add up all the elements of a vector:

```
>> x = [1   1   2   3   5   8   13]
>> sum(x)
ans =
```

```
     33
```

Finding maximum and minimum values:

```
>> max(x)
ans =
     13
>> min(x)
ans =
     1
```

Length of a vector:

```
>> length(x)
ans =
     7
```

1.5 Matrices

As you may have guessed, MATLAB can also handle matrices. MATLAB actually stands for MATrix LABoratory. In MATLAB, they are called two-dimensional arrays. Many of the operations used with vectors can be applied to matrices. Let's start with matrix generation. Similar to vectors, matrices can be created in many different ways.

```
>> a = [1 2 3; 4 5 6; 7 8 9]
a =
     1   2   3
     4   5   6
     7   8   9
>> b =
     [1 1
      2 2]
b =
     1   1
     2   2
>> c = [1.1 2.3 5.8;
        1.3 6.9 1.5]
c =
    1.1000   2.3000   5.8000
    1.3000   6.9000   1.5000
```

To access one element in a matrix, we can use the same syntax as a vector. However, since it is a matrix, we need to specify both a row and a column.

```
>> c(1,2)
ans =
    2.3000
```

Notice that you must specify the row, then the column.

What about scalar multiplication?

```
>> a = [1 2 3; 4 5 6; 7 8 9]
a =
    1  2  3
    4  5  6
    7  8  9
>>2a
 2a
  |
Error: Unexpected MATLAB expression.
```

MATLAB cannot infer that you want to multiply `a` by 2. You must explicitly tell MATLAB by using the multiplication symbol (`*`).

```
>>2*a
ans =
     2     4     6
     8    10    12
    14    16    18
```

One of the built-in functions for matrices in MATLAB is `magic(n)`. This function produces a matrix that has n rows and columns with element values from 1 to n. The rows, columns, and diagonals all sum up to the same amount. We can use this to practice some basic matrix operations.

```
>> t = magic(3)
t =
     8     1     6
     3     5     7
     4     9     2
>> u = [2 3 1; 3 6 9; 4 0 8];
```

Scalar operations and addition and subtraction.

```
>> u - 2*t
ans =
```

```
   -14      1    -11
    -3     -4     -5
    -4    -18      4
>> u/10 + t
ans =
    8.2000    1.3000    6.1000
    3.3000    5.6000    7.9000
    4.4000    9.0000    2.8000
>> v = [4.5 2.7; 6.2 7.1];
>> w = [2 5; 8 3];
```

Multiplication (which one is the same as finding the dot product?).

```
>> v*w
ans =
   30.6000   30.6000
   69.2000   52.3000
>> v.*w
ans =
    9.0000   13.5000
   49.6000   21.3000
```

Remember, the first case is a matrix multiplication operation, and the second is an element-by-element multiplication.

Division

```
>> v / w
ans =
    0.2382    0.5029
    1.1235    0.4941
>> v ./ w
ans =
    2.2500    0.5400
    0.7750    2.3667
```

Power

```
>> v^2
ans =
   36.9900   31.3200
   71.9200   67.1500
>> v.^2
ans =
   20.2500    7.2900
   38.4400   50.4100
```

Transpose

```
>> v'
ans =
    4.5000    6.2000
    2.7000    7.1000
>> v.'
ans =
    4.5000    6.2000
    2.7000    7.1000
```

Notice that the results for these two are the same. However, the transpose operator (') in the first example acts differently when used on a matrix with complex numbers. In that case, it also takes the complex conjugate of the numbers. Here's an example:

```
>> x = [2 + 7i 8 + 3i; 5 9]

x =

   2.0000 + 7.0000i   8.0000 + 3.0000i
   5.0000 + 0.0000i   9.0000 + 0.0000i

>> x'

ans =

   2.0000 - 7.0000i   5.0000 + 0.0000i
   8.0000 - 3.0000i   9.0000 + 0.0000i

>> x.'

ans =

   2.0000 + 7.0000i   5.0000 + 0.0000i
   8.0000 + 3.0000i   9.0000 + 0.0000i
```

We can also find the dimensions of a matrix by using the size function.

```
>> y = magic(7);
>> size(y)
ans =
     7 7
>> w = [1 2 3; 4 5 6]
>> size(w)
ans =
```

```
     2 3
```

Notice that the answer is a vector that gives the number of rows, then the number of columns.

The results of functions can be used as values in other expressions. The `length` function returns the number of rows in a matrix.

```
>> length(y) + length(v)
ans =
     9
```

For matrices, the `max` function will return a vector that corresponds to the largest values in each column of the matrix. `min` follows the same pattern, but for the minimum values.

```
>> z = [99  2  1  2;
        98  3 96  2;
        30  5  0 12]
>> max(z)
ans =
     99  5  96  12
>> min(z)
ans =
     30  2  0  2
```

How would you calculate the largest value in the whole array? The smallest value?

```
>> max(max(z))
ans =
     99
>> min(min(z))
ans =
     0
```

What about the largest value in the diagonal? There is a function to help you with that, `diag`, that returns a vector of the values in the main diagonal of the matrix.

```
>> diag(magic(3))
ans =
    8
    3
    2
```

```
>> max(ans)
ans =
    8
```

Along with `magic`, MATLAB includes other commonly used matrices. Like `magic(n)`, these matrices takes the desired number of rows as input.

Identity matrix.

```
>> eye(4)
ans =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1

>> zeros(2)
ans =
     0     0
     0     0
>> ones(3)
ans =
     1     1     1
     1     1     1
     1     1     1
```

These last two are often used to initialize matrices. This is a way to tell the computer how much space in memory you will need for a matrix you will be using, without declaring all the values of the elements right away. Doing this can help speed up your code, especially if you're working with large data sets.

1.6 More Matrices

MATLAB has a shorthand notation for dealing with matrices called colon notation. It allows you to specify a range of values that you want to populate an array with. For example, to create a vector with the elements from 1 to 10, use the following command:

```
>> a = [1:10]
a =
     1     2     3     4     5     6     7     8     9
10
```

From negative to positive.

```
>> b = [-4:4]
ans =
    -4    -3    -2    -1     0     1     2     3     4
```

You can omit the square brackets to the same effect.

```
>> c = 1:5
ans =
     1     2     3     4     5
```

What if we only want odd numbers? MATLAB has a similar notation that allows you to specify step size.

```
>> a = [1:2:10]
d =
     1   3   5   7   9
```

You can also use a mix of ranges and values.

```
>> e = [1:3, 102, 7:2:13]
e =
     1     2     3   102     7     9    11    13
```

Decreasing ranges can also be specified.

```
>> f = [5:-1:0]
f =
     5   4   3   2   1   0
```

Be careful to add in the lower bound when using a decreasing data set. If you leave it out, you'll create an empty matrix.

```
>> f = [5:-1]
f =
   Empty matrix: 1-by-0
```

What happens if your input is not an integer?

```
>> g = 2.2:5
g =
    2.2000    3.2000    4.2000
```

MATLAB will do the same thing, except that it will stop once the next step is greater than the given upper bound.

This notation can also be used to create matrices. Just separate rows with a semicolon, as was done before.

```
>> e = [2:2:10; 1:2:9]
e =
     2     4     6     8    10
     1     3     5     7     9
```

There is a similar function to colon notation, called `linspace`. It takes two or three inputs. If it only receives two inputs, it will automatically generate 100 equally spaced elements, between the two input amounts.

```
>> linspace(1,10)
ans =
Columns 1 through 6
1.0000    1.0909    1.1818    1.2727    1.3636    1.4545
Columns 7 through 12
1.5455    1.6364    1.7273    1.8182    1.9091    2.0000
Columns 13 through 18
2.0909    2.1818    2.2727    2.3636    2.4545    2.5455
Columns 19 through 24
2.6364    2.7273    2.8182    2.9091    3.0000    3.0909
Columns 25 through 30
3.1818    3.2727    3.3636    3.4545    3.5455    3.6364
Columns 31 through 36
3.7273    3.8182    3.9091    4.0000    4.0909    4.1818
Columns 37 through 42
4.2727    4.3636    4.4545    4.5455    4.6364    4.7273
Columns 43 through 48
4.8182    4.9091    5.0000    5.0909    5.1818    5.2727
Columns 49 through 54
5.3636    5.4545    5.5455    5.6364    5.7273    5.8182
Columns 55 through 60
5.9091    6.0000    6.0909    6.1818    6.2727    6.3636
Columns 61 through 66
6.4545    6.5455    6.6364    6.7273    6.8182    6.9091
Columns 67 through 72
7.0000    7.0909    7.1818    7.2727    7.3636    7.4545
Columns 73 through 78
7.5455    7.6364    7.7273    7.8182    7.9091    8.0000
Columns 79 through 84
8.0909    8.1818    8.2727    8.3636    8.4545    8.5455
Columns 85 through 90
```

```
8.6364    8.7273    8.8182    8.9091    9.0000    9.0909
Columns 91 through 96
9.1818    9.2727    9.3636    9.4545    9.5455    9.6364
Columns 97 through 100
9.7273    9.8182    9.9091   10.0000
```

Notice how MATLAB displays matrices with more columns than can fit within the Command Window width.

In `linspace(x,y,z)`, like in the previous case, it will generate equally spaced elements between the first two input values (x and y). However, it will only generate z elements.

```
>> linspace(1,5,10)
ans =
Columns 1 through 6
1.0000    1.4444    1.8889    2.3333    2.7778    3.2222
Columns 7 through 10
3.6667    4.1111    4.5556    5.0000
```

The `linspace` function is most often used for plotting (and we'll get to that later on).

Another use for the colon operator is for *accessing* ranges of matrices. Previously, we were only able to access one element at a time. With the colon operator, we can access a whole row, column, or submatrix.

Sample array

```
>> a = [1:2:10; 10:-2:1; 1:5]
a =
     1     3     5     7     9
    10     8     6     4     2
     1     2     3     4     5
```

Accessing one column.

```
>> a(:,1)
ans =
     1
    10
     1
>> a(:,5)
ans =

     9
```

```
     2
     5
```

Accessing one row.

```
>> a(2,:)
ans =
    10     8     6     4     2
>> a(5,:)
Index exceeds matrix dimensions.
```

Note that the row number must exist.

To access a sub-matrix, the colon operator can indicate the range of rows and columns we would like to access. In the case below, we get rows 2 to 3 in columns 3 to 4.

```
>> submatrix = a(2:3,3:4)
submatrix =
     6     4
     3     4
```

To convert your matrix into one column you can use the following syntax:

```
>> submatrix(:)
ans =
     6
     3
     4
     4
```

1.7 Scripts

Until this point, we have been typing our commands into MATLAB line-by-line, or in small batches with the comma or semi-colon character. In this section, we introduce a new way to execute a set of commands in MATLAB: the script.

A script is a file with a set of instructions for MATLAB to execute. The instructions are executed by typing the name of the file in the MATLAB **Command Window**. Script files always have a .m file extension, and as such, are called M-file scripts.

To create a script, we'll use the MATLAB editor. You can launch it by typing `edit` in the **Command Window**, or clicking the **New Script** in the top left corner of the **HOME** tab in the default setting. To practice, type the following:

```
%Calculate length of a hypotenuse of a right-angle
%triangle with side lengths 5 and 12

disp('Calculating the hypotenuse length of a right-
angle triangle with side lengths of 5 and 12:')
a = 5;                    %define a
b = 12;                   %define b
hyp = sqrt(a^2 + b^2)     %calculate hypotenuse
```

The first two lines are comments. They begin with a `%` sign, which means MATLAB ignores everything in the line after that symbol. Comments are used to help the programmers keep track what's going on in their set of instructions.

The `disp` is a simple command that just prints the characters between the single quotes. Those characters together are called a string.

Now save your file by clicking on the **Save** button (or pressing **<Ctrl-X>** immediately followed by **<Ctrl-S>**). A prompt will pop up to ask you where you want to save your file. It's important to keep your files organized, so each assignment or project should have its own file. For now, you can create a new folder using the **Create New Folder** button in the prompt called MATLAB-Practice.

Then rename your file hypotenuse.m. Click the **Save** button.

Now you're ready to run your script. Make sure your file is listed in the **Current Folder** window. In the **Command Window** type as follows:

```
>> hypotenuse
Calculating the hypotenuse length of a right-angle
triangle with side lengths of 5 and 12:
hyp = 13
```

What if we want to include the result inside the string? The command `fprintf` can help us. Open your hypotenuse.m file again. At the end of the file, type the following in:

```
fprintf('The hypotenuse length of a right-angle
triangle with  side lengths of 5 and 12 is \%d.', hyp)
```

Like in the `disp` command, you must specify the string you want to print out. However, to print a variable value within the statement you cannot just type

the variable name inside the single quotes, since MATLAB will not recognize it as a variable name.

Instead you use a format string. That is what the `%d` is. The `%` tells MATLAB that you want to display a variable here. The `d` specifies that you want to display the varible as a decimal. Finally, you must tell MATLAB which variable you want to print. In this case it is `hyp`. Don't forget the comma between your variable and single quote!

Now if you execute hypotenuse again, here's the new result:

```
>> hypotenuse
Calculating the hypotenuse length of a right-angle
triangle  with side lengths of 5 and 12:
hyp = 13
The hypotenuse length of a right-angle triangle with
side lengths of 5 and 12 is 13.
```

In MATLAB, there is also a command that can help you if you forget how a command works. Its keyword is `help`. For example, say we've forgotten what `clc` does.

```
>> help clc
 clc    Clear command window.
    clc clears the command window and homes the cursor.

    See also home.

    Reference page in Help browser
       doc clc
```

You can also click `doc clc` to read the full documentation of the command.

When you use help on a script, it displays the commented lines at the top of the script.

```
>> help hypotenuse
 Calculate length of a hypotenuse of a right-angle
  triangle with side lengths 5 and 12
```

1.8 Functions

In the last lesson, we wrote a script that calculated the length of the hypotenuse of a right-angle triangle with other side lengths of 5 and 12. However, if we

wanted to calculate it for a different-sized triangle, we would have to assign the new values to the variables `a` and `b`. Each time we would want to calculate a different hypotenuse length, we would have to declare the new values first.

Compare this with the built-in function `sin`.You do not have to redefine values before you use it. You just give it an input, and it calculates the appropriate value for that input.

So, if we are to improve the script, we can transform it into a user-defined function. Writing them is very similar to writing a script, but since you can specify your desired inputs, there is a special way to define a function.

```
function [output1,output2,...]  = functionName(input1,input2,...)
```

You must type this at the beginning of your M-file.

In our case, what is our output? Well, we want to return the value of the hypotenuse, so let's call it `hyp`. What are our inputs? Well, we take in two values to calculate the hypotenuse, so let's call them a and b.

Here's the function version of hypotenuse now:

```
function [hyp] = fhypotenuse(a,b)
% Calculate length of a hypotenuse of a right-angle
%  triangle.
% Requires 2 side length inputs.

hyp = sqrt(a^2 + b^2);  \%calculate hypotenuse
fprintf('The hypotenuse length of a right-angle triangle
with side lengths of \%d and \%d is: \%f',a,b,hyp)
```

Type the above into a new script file, then save it (set the path if need be).

To use your function you must provide it the correct inputs, surrounded with round brackets, like we do with `sin`.

```
>> fhypotenuse
Error using fhypotenuse (line 5)
Not enough input arguments.

>> fhypotenuse(3,4)
The hypotenuse length of a right-angle triangle with
side lengths of 3 and 4: 5
ans =
     5
```

With any function you can also place the output in a variable. We've done so

with `sqrt` above, and below we can do it with `fhypotenuse`.

```
>> h = fhypotenuse(3,4)
The hypotenuse length of a right-angle triangle with
side lengths of 3 and 4: 5
h =
     5
```

Chapter 2

COMPUTER SCIENCE CONCEPTS

2.1 Variables

In the first chapter, we learned that variables are a place where a value can be stored. Here is a more in-depth look at variables and values in MATLAB. Variables are symbolic, which means they are the name for a place in memory that holds a certain value. In MATLAB they are created when they are assigned a value (remember that = sign?). What exactly is a value? In memory, a value is just data, stored as 0s and 1s. Semantically, a value is an expression that cannot be simplified further. Values can be explicit, like in the following:

```
>> a = 2
a =
     2
```

Here, the value is explicitly 2. Otherwise, a value is derived from an expression, such as in the following cases:

```
>> b = 2 * 3
b =
     6
>> c = a * 3
c =
     6
```

Notice that b and c are derived from different expressions, but end up having the same value.

As soon as you hit **<Enter>** in MATLAB, and many other programming languages, the expression will be computed down to its value. Then it will assign that value to a variable. This is called eager evaluation. With MATLAB, if no variable is specified, it will assign it to the variable `ans`.

In other programming languages, a variable has a type. A type describes what kind of data a variable is holding. Different types can take different amounts of memory space. Each programming language has a different set of fundamental types. In MATLAB, there are three basic types of variables that we will be using: numerical, logical, and character. MATLAB has more fundamental types, but they are used by more experienced programmers.

There are many kinds of numerical types in MATLAB, but we will mostly be using the type double. All of the previous examples have values of type double.

You can check what type a variable is using the command `whos`. In MATLAB, the type of a variable is called its class. If you typed in the commands earlier in this less, try typing `whos`. Under the heading Class, you can see the value of a, b, and c are double.

```
>> whos
  Name       Size              Bytes  Class     Attributes

  a          1x1                   8  double
  b          1x1                   8  double
  c          1x1                   8  double
```

Notice that you can also check the array size of a variable and its size in memory. Its size in memory is in the column headed Bytes. A byte is a basic metric of computer memory. One byte is equivalent to 8 bits. One bit can hold a value of 0 or 1, on or off. So 8 bytes is the equivalent of 8 * 8 = 64 bits.

We will talk more about the types other than double later, but for now, double is all you need.

In addition to checking the size and type of a variable, you can also view and edit it in a different screen. Click on the Open Variable menu, then click on the variable a. It opens a new window called Variables. Here you can view the value(s) of your variable, add elements, remove elements, and edit elements. This is especially helpful when you are working with large arrays. When you are done, just hit the x in the title bar and you will return to your command window. Check your Workspace to see the changes.

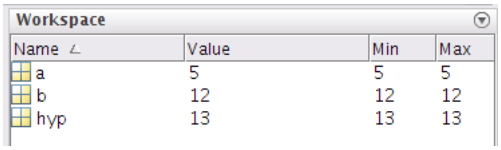
Remember the script and the function we wrote in the first chapter?

```
%Calculate length of a hypotenuse of a right-angle
%triangle with side lengths 5 and 12

disp('Calculating the hypotenuse length of a right-
angle triangle with side lengths of 5 and 12:')
a = 5;                    %define a
b = 12;                   %define b
hyp = sqrt(a^2 + b^2);    %calculate hypotenuse

fprintf('The hypotenuse length of a right-angle
triangle with  side lengths of 5 and 12 is \%d.', hyp)
```

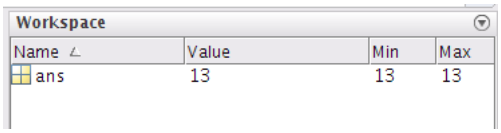
Clear your workspace and run the file. What happens to the Workspace?



The variables a, b, and hyp are added to the Workspace.

Clear your workspace again. What happens if you run the function fhypotenuse(5, 12)?

Only the ans variable is added to the Workspace. Even though both files have the variables a,b, and hyp, only the script adds the three.



This is because MATLAB opens a new Workspace when a function is called. Each variable used in the function Workspace is called a local variable. Once the function completes, the variable dies. That is why MATLAB does not add the variables a,b, and hyp to the main Workspace.

This is not the case with scripts. To MATLAB, a script is just a different place to read a batch of commands from, so each script just shares the main Workspace.

The opposite of a local variable is called a global variable. These variables can be seen, accessed and used by all functions that declare it. Here is the syntax for declaring and assigning a value to a global variable:

```
>> global x
>> x = 3;
```

Each function that wants access to the global x should have the the declaration `global x`. Although they can be useful, global variables should be avoided, because it is very easy to accidentally change its value. This may not be apparent yet, since we have only been working with toy programs. However, when a program is thousands of lines long, you can see that it would be hard to spot an accidental assignment.

2.2 Logical Variables

Another kind of variable in MATLAB is a logical variable. This type is used to represent the two logical variables, true and false.

```
>> x = true
x =
   1
>> y = false
y =
   0
```

As you can see, true is associated with the value 1, whereas false has a value of 0. If you look in the Workspace, you can see that the MATLAB Class of x and y are of type logical.

Logical variables are more often created using relational and logical operators. Relational operators compare variables to produce a logical variable. This comparison is called a boolean expression. MATLAB has 6 relational operators:
-less than ($<$) and greater than ($>$)

-less than or equal to ($<=$) and greater than or equal to ($>=$)

-equal to ($==$) and not equal to ($\sim =$)

Here they are in action:

```
>> 10 > 5
ans =
     1
>> -5 > -5
ans =
     0
>> 100 >= 100
```

```
ans =
       1
```

You can also use logical operators on arrays:

```
>> w = [10 3 -4 7]; x=5; x<=w
ans =
     1      0      0      1
>> y = [12 4 -4 12]; w==y
ans =
     0      0      1      0
>> z = 12; z~=y
ans =
     0      1      1      0
```

Note that when comparing arrays, the operation is done element-by-element.

In addition to the six relational operators, there are six logical operators:

-And (`&`) and Short-circuit And (`&&`)

-Or (|) and Short-circuit Or (||)

-Not ($\sim$)

The logical operators are based on the operators in Boolean logic, which is a branch of math that manipulates values of true and false. A command that uses a logical operator is also called a boolean expression.

2.2.1 And

And evaluates two boolean expressions, and if both are true, a value of true is returned. Otherwise, the expression evaluates to false. A popular way to summarize this is to use a truth table, as shown below. Let a and b be boolean expressions. Remember that a value of 1 means the expression evaluates to True and a value of 0 means the expression evaluates to False.

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

2.2.2 Or

Or evaluates two boolean expressions, and if either one is true, a value of true is returned. Otherwise, the expression evaluates to false.

a	b	$a \mid b$
0	0	0
0	1	1
1	0	1
1	1	1

2.2.3 Not

Not evaluates one boolean expression, and if it is true, a value of false is returned. Otherwise, the expression returns a true value.

a	$\sim a$
0	1
1	0

You can try typing the truth table inputs into MATLAB to verify these tables.

2.2.4 Examples

```
>> true & false
ans =
     0
>> ~0
ans =
     1
>> (1&0) | 0
ans =
     0
>> x = 5; y = 2; z = -3;
>> (x>y) & (z<=y)
ans =
     1
>> ~(x>y) & (z<=y)
ans =
     0
```

2.2.5 Short-circuit Operators

Many programming languages have a And and Or Short-circuit operators. These operators only check the second statement if the first statement passes the condition. The standard logical operators (also known as bitwise operators) always check both conditions. This can speed execution times. In addition, short-ciruit operators only work on scalars, whereas the bitwise operators can work on scalars or arrays.

```
>> a = [1 0 1; 1 1 1; 0 0 0]

a =

     1     0     1
     1     1     1
     0     0     0

>> b = [1.5 6 2.2; 0 0 7.1; 1 5.3 9.6]

b =

    1.5000    6.0000    2.2000
         0         0    7.1000
    1.0000    5.3000    9.6000

>> a & b

ans =

     1     0     1
     0     0     1
     0     0     0
>> a&&b
Operands to the || and && operators must be
convertible to logical scalar values.

>> true && false
ans =
    0
>> 5 && 200
ans =
    1
```

The answer to the bitwise operation (`a&b`) gives a value of 1 if both arrays have a non-zero value in that element, and 0 otherwise.

Note that the short-circuit operator did not work on the array.

2.3 Conditionals

Up until now we've only been able to write instructions that can only execute one-by-one. We know that each instruction will execute. But what if we only want something to happen under certain circumstances? For example, say we want a function that takes in a number and prints a string that tells us if that number is even or odd. How should we begin?

Well we can start with the usual function setup, so open up a new script. First let's give our function a meaningful name: `numEvenOrOdd`. We also know that we need one number as input, so we'll specify that within the round brackets. Since our output will be a print statement, we do not need an output variable (we'll see why later). Finally, we can give our function a descriptive comment, so anyone using our function can see what it does.

```
function [] = numEvenOrOdd(num)
%This function takes one number as input and
prints its parity.
```

But how can we check if a number is even or odd? The most common way to do this is to see if it is divisible by 2. To do this, we can use the `mod` function.

Here are some examples of how it is used:

```
>> mod(6,2)
ans =
     0
>> mod(10,3)
ans =
     1
>> mod(3,9)
ans =
     3
>> mod(4,0)
ans =
     4
>> mod(0,4)
ans =
     0
```

As you can see, the first input to the `mod` function is the number being divided (the dividend), and the second number is the divisor. `ans` contains the remain-

der. Now let's put it into our function. Remember that we are checking if our input number is divisible by 2.

```
function [] = numEvenOrOdd(num)
%This function takes one number as input and
prints its parity.

result = mod(num,2)
```

Okay, now what happens if we save this function and try to run it?

```
>> numEvenOrOdd(10)
result =
     0
>> numEvenOrOdd(5)
result =
     1
```

As you can see, the proper boolean values are printed out. Yay! However, we want a function that prints out 'Number is even.' if the result of the mod function is 0, and 'Number is false' if the result of the mod function is 1.

To accomplish this we need a new construct: the if-statement.

An if-statement evaluates a boolean expression, and executes a certain block of text only if that boolean expression is true. Here is the general structure:

```
if <boolean expression>
  if boolean is true, execute statements here
end

continue executing here
```

Note that if the boolean expression evaluates to false, none of the code in the middle is excuted. We now have some control over what gets executed!

Let's apply this to our function. Our boolean expression is result of the mod function.

```
function [] = numEvenOrOdd(num)
%This function takes one number as input and
prints its parity.

result = mod(num,2);

if(result==0)
```

```
        disp('Number is even.')
```

Notice the semicolon at the end of the `result = mod(num,2);` line. Save and try runnning the script again.

```
>> numEvenOrOdd(10)
Number is even.
>> numEvenOrOdd(5)
>>numEvenOrOdd(1282)
Number is even.
```

Notice that nothing is printed when the number is odd. This is because we haven't handled that case yet! We need a new construct for that, the else block. The else block contains code that should be run if the boolean expression in the if statement evaluated to false. Here is the general structure:

```
if <boolean expression>
  if boolean is true, execute statements here
else
  boolean is false, execute these statements instead
end

continue executing here
```

So now let's add that to our script.

```
function [] = numEvenOrOdd(num)
%This function takes one number as input and
prints its parity.

result = mod(num,2);

if(result==0)
        disp('Number is even.')
else
        disp('Number is odd.')
end
```

Here's the result after we run the script again.

```
>> numEvenOrOdd(10)
Number is even.
>> numEvenOrOdd(5)
Number is odd.
```

Let's pause here to consider if there are any cases that we have missed. Since a number can only be even or odd, it seems like we are okay. However, what if we input a non-integer number into our function?

```
>> numEvenOrOdd(1.5)
Number is odd.
```

Uh-oh! How did this happen? Well, we only checked if the remainder was 0 (i.e., the number was even). Otherwise, since integers can only be even or odd, that the number was odd. This could work in other languages such as Java or C, where variables must be declared first, but in MATLAB, it causes problems. To fix this, we need the `elseif` construct.

`elseif` is used when you want different code to be executed for different cases. Here's the general structure:

```
if <boolean expression>
  if boolean is true, execute statements here
elseif <boolean expression>
  first boolean is false, this boolean true
  execute these statements
else
  boolean is false, execute these statements instead
end

continue executing here
```

You can have as many elseif blocks as you want, however, only one of all the blocks (including the if-block and the else-block) will be executed. Let's add this to our script.

```
function [] = numEvenOrOdd(num)
%This function takes one number as input and
prints its parity.

result = mod(num,2);

if(result==0)
        disp('Number is even.')
elseif(result==1)
        disp('Number is odd.')
else
        disp('Number is not an integer.')
end
```

And here's the result:

```
>> numEvenOrOdd(10)
Number is even.
>> numEvenOrOdd(5)
Number is odd.
>> numEvenOrOdd(1.5)
Number is not an integer.
```

2.4 For Loop

One of the most powerful things a computer can do is execute the same command over and over again. To harness this power, we could just copy-paste the same code over and over again, right? NO! Imagine having to copy-paste a command 1000 times. What about 1 million times?

This would be incredibly inefficient and you would probably miss one. That's why programming languages have structures to specifically deal with this case. They're called loops. Let's start with the for loop.

Here is the general structure of a for loop:

```
for loopVariable = initialValue:stepSize:endValue
        execute statements here
        ...
end
```

The loopVariable is used to keep track of which iteration of the loop we're in. Its value is a vector that is created based on the initial value, stepSize, and endValue. The initialValue is the first element in the loopVariable vector. Then, it is incremented by stepSize, and this value becomes the second element in loopVariable. This continues until the value incremented is larger or equal to endValue. If the loop will not converge, the vector will not be created.

Here's an example:

```
>> for i = 1:2:4          %loop declaration
    i                     %loop body
end                       %loop end
```

Walkthrough: When MATLAB first reaches the loop, it creates the loopVariable vector. Then it checks if the loop will converge. Since 1+2+2=5, which is greater that 4, this loop is valid. Hence, MATLAB will execute the loop body, based on each element in the loopVariable vector. This an example loop, so it just

displays the value of i. Since i is 1, it will display i = 1. Then it reaches the end statement.

Here, MATLAB will move on to the next element in the loopVariable vector, 3. The body of the loop prints 3. Since there are only two elements in the loop variable, the for loop is now complete. Here is the whole thing:

```
>> for i = 1:2:4
    i
end
i =
    1
i =
    3
```

You can also omit the stepSize, and MATLAB will assume a default value of 1.

```
>> for i = 1:4
    i
end
i =
    1
i =
    2
i =
    3
i =
    4
```

There is one detail about the MATLAB for loop that makes it different from for loops in other programming languages. In other programming languages, you can change the value of the loopVariable within the loopVariable. In MATLAB, you can use and manipulate the loopVariable but this does not affect its value as a loop counter, because the loop vector is created before any of those manipulations happen. This means that the number of times a loop is executed is based entirely on the loop declaration.

For example:

```
>> for i = 1:3
    i=i+2
end
i =
     3
i =
     4
```

```
i =
     5
```

Let's do something more complicated, like changing every other element in an array to have a value of 2.

```
>> a = [1  4  5  4  5  8  0  7  1  4  6];
>> for i = 1:2:11
    a(i) = 2;
end
>> a
a =
    2  4  2  4  2  8  2  7  2  4  2
```

However, this is not the most efficient way to do things. Remember subscripting? We can use that here to accomplish the same thing.

```
>> a = [1  4  5  4  5  8  0  7  1  4  6];
>> a(1:2:11) = 2
a =
    2  4  2  4  2  8  2  7  2  4  2
```

In fact, if it's possible you should try to use subscripting rather than a for loop. There are fewer checks that MATLAB has to do, so it is a much faster operation, especially when using larger matrices.

Let's see an example where a for loop makes more sense. For example, let's say we want to calculate multiple results for different user input.

You can open a new script to type in the following for loop example.

```
% Asks for & calculates the square of 5 input values

for i = 1:5
    x = input('Give me a number to square: ');
    x^2
end
```

The `input` function displays a prompt to the command window and waits for an input from the user. The input is stored in x, and can be used like any other variable.

Here is one result, using the values 3, 10, 5, 6, and 9 as inputs.

```
>> square5
Give me a number to square: 3
```

```
ans =
    9
Give me a number to square: 10
ans =
    100
Give me a number to square: 5
ans =
   25
Give me a number to square: 6
ans =
   36
Give me a number to square: 9
ans =
   81
```

This would have been impossible to do using vectors.

2.5 While Loop

In the last lesson, we learned about the for loop, which allows us to loop for a pre-determined amount of steps.

But what if we did not know in advance how many steps we needed?

This would require a while loop. The while loop allows you to repeat a section of code *while* a boolean expression is true.

This is useful for situations when a piece of code needs to execute a different amount of times based on different inputs.

Here is the general structure of a while loop:

```
while(<boolean expression>)
        if boolean is true, execute loop body
end
```

When MATLAB first encounters the loop, it evaluates the boolean expression. If it is true, then it executes the loop body. When the end keyword is reached, MATLAB returns to the boolean expression and re-evaluates it and the process begins again.

But how does the loop end? We must make sure that something in the loop body changes the value of the boolean expression. More typically, a while loop looks like this:

```
<boolean expression initialization>
while(<boolean expression>)
        if boolean is true, execute loop body
        update boolean expression
end
```

For example:

```
>> n = 5
while n>0
    n
    n=n+1;
end
```

Of course, this could easily be done with a for loop. Try to do one now!
So when is a good time to use a while loop? Often it is used to repeat an action until a user wants to quit. In this case, we cannot know in advance how many actions the user wants to complete.

For example, say we are keeping score for a basketball game. You can add 1,2, or 3 points to the score by typing the respective numbers. When the game is done, the user can press 0 to end the game.

The script for this situation is as follows:

```
score = 0;

while(score==0 || (n<4 && n>0))
    disp('Press zero to end game.')
    n = input('Enter point value (1,2, or 3): \\n');
    if(n==0)
        break;
        %This keyword tells MATLAB to break out of the current loop.
    elseif(n==1)
        score =score+1;     %Add 1 to the score.
    elseif(n==2)
        score = score+2;    %Add 2 to the score.
    elseif(n==3)
        score = score+3;    %Add 3 to the score.
    else
        disp('Invalid point value');
        %If n not 1,2,3, display error message.
        n=1;
        %Reset n to 1 to continue loop.
    end
```

```
    fprintf('Current score: %d\n', score)
    %Print current score.
end

fprintf('Final score: %d\n', score)
%Print final score
```

One important command when practicing while loops is <**Ctrl+C**>. This command allows you to quit out of the loop. Try it with the following infinite loop.

```
>> x=1;
while(x>0)
    x = 5
end
```

Chapter 3

APPENDIX

3.1 Command List

The commands below are given in order of appearance.

clear x, removes given x from the current Workspace. If x is all, then all variables

format, changes number display to a given type (ex: long, short, bank., etc.)

sin(x), **cos(x)**, and other trigonometric functions, calculate the appropriate value for the function used and the value given.

sqrt(x), calculates the square root of the given value x.

exp(x), calculates the mathematical value e to the xth power.

log(x), **log2(x)**, **log10(x)**, calculate the log (base-e, base-2, and base-10, respectively) of a given x.

which x, identifies if x is a variable or built-in function.

clc, clears the Command Window of previously entered commands.

size(x), gives a vector describing the rows and columns of matrix x.

magic(n), returns a magic square (rows and columns all add up to the same amount) with n rows and columns.

max(x), returns the largest value in the matrix x.

min(x), returns the smalled value in the matrix x.

length(x), returns the length of a given vector x.

diag, returns the main diagonal of a given matrix x.

zeros(n), **ones(n)**, returns a matrix of size n initialized with zeroes or ones.

linspace(x,y,z), returns a vector of z equally spaced elements between x and y.

disp('s'), displays the string s.

fprintf('s',x), prints s to the screen, replacing format strings with x.

help, displays documentation for a given function.

whos, displays information about the current Workspace.

mod(x,y), returns the modulus after dividing x by y.

Ctrl+c, terminates the command execution, useful for ending long-running loops.

Glossary

access An array is accessed when the values inside it are retrieved. 16, 53

and A logical operator that results in true only if both inputs are true, and is false otherwise. 53

array Organized collection of elements. 13, 53

array operator Or dot operator, specifies that the operation is element-by-element. 15, 53

assignment An operation that binds a value to a variable. 7, 53

boolean A value that is either True (1) or False (0). 53

carriage return Corresponds to pressing the ¡Enter¿ key. 15, 53

colon notation A shorthand form of accessing/retrieving sections of matrices. 53

command A statement that is executed and produces a result. 4, 53

double A way to store a number on a computer using 64 bits. 10, 53

function A MATLAB structure that can take an input, does something, and can return an output. It has its own scope.. 53

index To index into an array is to access a certain location in an array. 53

not A logical operator that negates its input. 53

or A logical operator that results in true if at least on of its inputs are true, and false if both are false. 53

scalars In MATLAB, matrices with one element. 13, 53

script A collection of commands housed in a different file. 53

short-circuit operator A logical operator that does not check the second input if the first one decides it already. 53

submatrix A rectangular subsection of an array. 53

type Also called a Class in MATLAB, the kind of value a variable can hold. 53

value The numerical value an expression simplifies down to. 4, 53

variable A place in memory that can store a value. 4, 53

vector One-dimensional array of elements. 13, 53