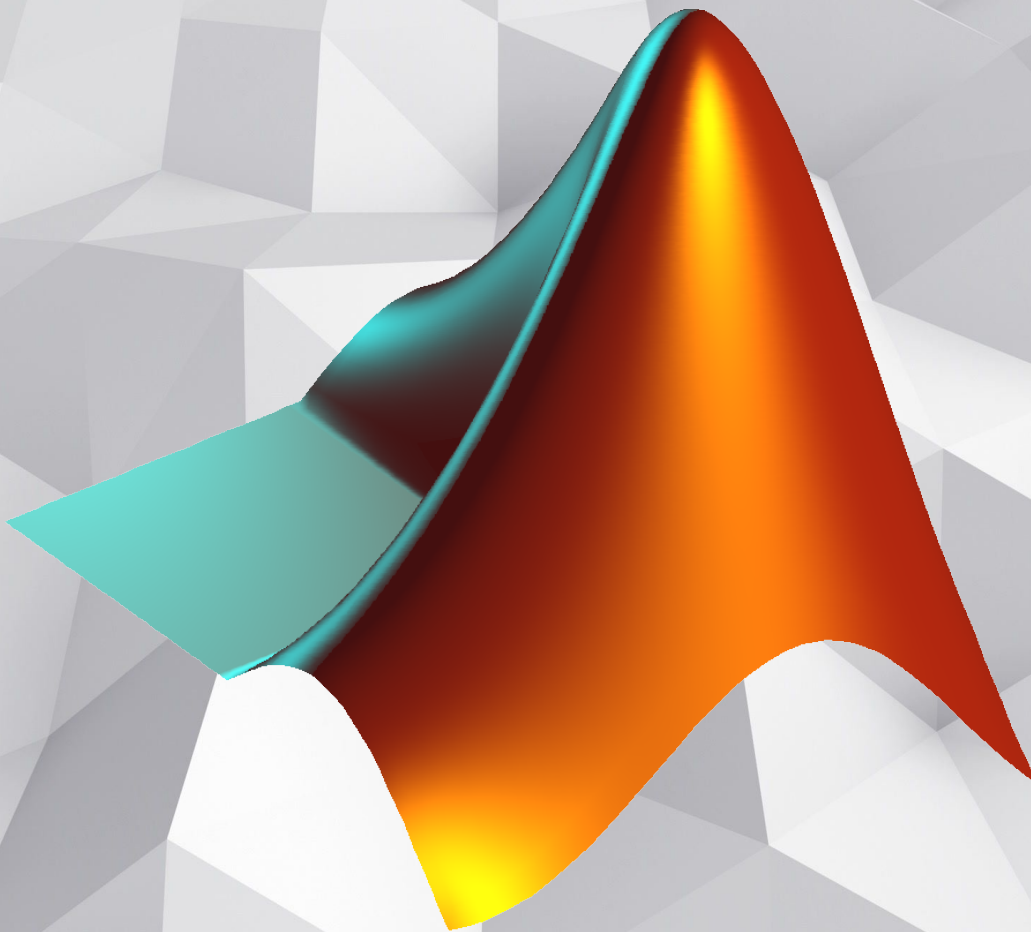


**FIRST EDITION**

**An Introduction to**

# MATLAB



**HARFORD**  
COMMUNITY COLLEGE

**Authors: Amir Ghaemi | Mohammad Ghaemi**



# DEDICATIONS

**To Chris Jones for being a great mathematician, a wonderful teacher, and an amazing mentor. Who presented us with such great opportunity to write this book.**

**To Harford Community College, an amazing two-year college, to which enabled us and gave us a purpose towards writing this book.**

# A WORD FROM THE AUTHORS

*“Hello, My name is Amir Ghaemi, I am currently a Computer Engineering student at the University of Maryland in College Park. I graduated from Harford Community College with an Associates degree in Engineering in 2018. Throughout my career, I have had many interactions with MatLab and I understand the importance of this programming language within any STEM major. Looking back at my two years at HCC, I wish that there was a course offered to teach us the fundamentals of MatLab in order to prepare us for when we go to a four-year University. So, I decided to meet with Prof. Chris Jones, and try to set up a new optional MatLab course at HCC for future students. In this book, I will be passing on my current knowledge as well as the research conducted on MatLab to future students interested in learning the MatLab language.”*

**-Amir Ghaemi**

*“Hi, I am Mohammad Ghaemi, and I am currently a student of the University of Maryland: College Park. I am planning to get a degree in Computer Science: Data Science there. Although I am only in my first year at college, MATLAB was still used frequently throughout my math classes and the professors simply expected students to know it or learn it. And so when Amir introduced me to Professor Chris Jones over the summer and talked to me about this project, I accepted it immediately with the hope that we could help future students learn MATLAB from a single source that provides concise information, accurate examples, and helpful practice problems.”*

**-Mohammad Ghaemi**



# PURPOSE

**Our goal in writing this book is to help students that are new to MATLAB and programming in general, learn the fundamental concepts that come with the subject. This book should help students utilize MATLAB in their mathematical studies and more. MATLAB can and is used beyond simple mathematical computation (across a variety of studies from physics to chemistry) and while chapter one will cover some possible applications, because the purpose of this book is to introduce students to MATLAB, these applications will not be covered beyond what is shown in chapter one. We hope to pass on our knowledge and research of MATLAB to our future students pursuing this useful language.**

# CONTENTS

## **Chapter 1: Introduction to MATLAB**

- 1.1 What is MATLAB?
- 1.2 Windows of MATLAB
- 1.3 Basic MATLAB Commands And Functions
- 1.4 Importing, Saving, Loading, And Executing In MATLAB
- 1.5 Real World Applications of MATLAB
- Student Exercises

## **Chapter 2: Data Types and Basic Evaluations**

- 2.1 Numeric Data Types
- 2.2 Characters and Strings
- 2.3 Conversion and Identification
- 2.4 Basic Numeric Operations
- 2.5 Input and Output
- Student Exercises

## **Chapter 3: Algebraic Computation**

- 3.1 Simplifying
- 3.2 Expanding
- 3.3 Factoring
- 3.4 Solving Equations
- 3.5 Partial Fraction Decomposition
- Student Exercises

## **Chapter 4: Conditions and Loops**

- 4.1 Conditional Statements
- 4.2 Basic Boolean Operations
- 4.3 Switch, Case, Otherwise
- 4.4 For Loops
- 4.5 While Loops
- 4.6 Try and Catch
- 4.7 Useful Commands
- Student Exercises

## **Chapter 5: Arrays**

- 5.1 Categorical Arrays
- 5.2 Cell Arrays
- 5.3 Tables
- 5.4 Structures
- 5.5 Conversions
- Student Exercises

## **Chapter 6: Classes and Functions**

- 6.1 Functions
- 6.2 Classes
- 6.3 Enumeration
- 6.4 Recursion
- Student Exercises

## **Chapter 7: Data and Statistics**

- 7.1 Basic Statistical Functions
- 7.2 Importing Data
- 7.3 Plotting Data
- 7.4 Analyzing Data
- Student Exercises

## **Chapter 8: Coordinate Systems**

- 8.1 Plotting Basics
- 8.2 Rectangular
- 8.3 Parametric
- 8.4 Polar
- 8.5 Cylindrical
- 8.6 Spherical
- Student Exercises

## **Chapter 9: Two-Dimensional Vector Operations**

- 9.1 Vectors and Basic Vector Operations
- 9.2 Plotting
- 9.3 Other Operations
- Student Exercises

## **Chapter 10: Calculus In MATLAB**

- 10.1 Limits
- 10.2 Derivatives
- 10.3 Integrals
- 10.4 Infinite Series
- Student Exercises

# CHAPTER 1

## Introduction to MATLAB:

Section 1: What is MATLAB?

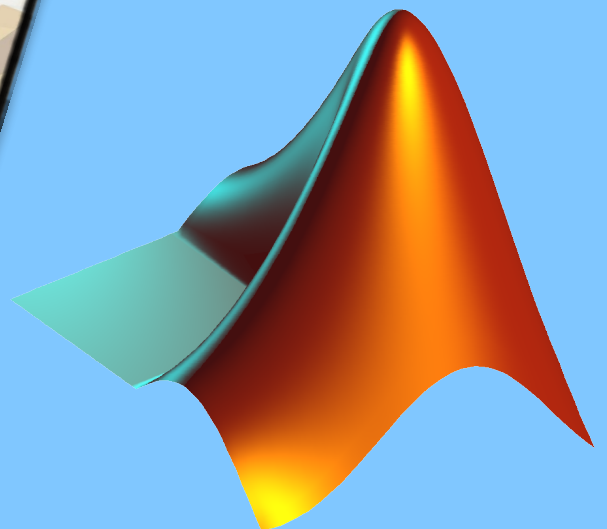
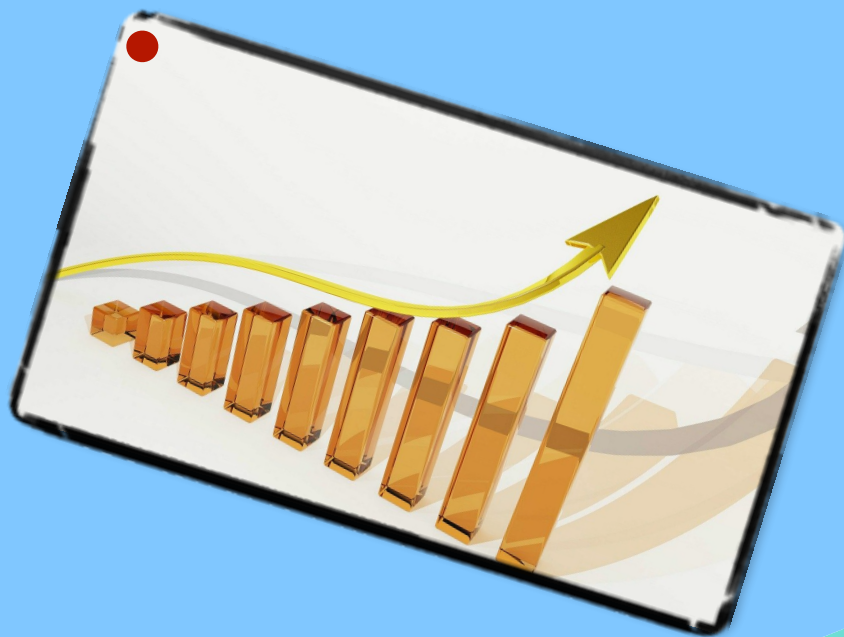
Section 2: Windows of MATLAB

Section 3: Basic MATLAB Commands

Section 4: Importing, Saving, Loading, and Executing In MATLAB

Section 5: Real Worlds Applications of MATLAB

### Student Exercises



## Section 1: What is MATLAB?

**W**hat is MATLAB? By definition MATLAB is a high-performance language for technical computing that was developed by MathWorks. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. MATLAB is derived from two words, Matrix and laboratory. In 1984 when MATLAB was created using C, MATLAB was originally for working with matrices and vectors.



## Section 2: Windows of MATLAB

There are multiple panes or regions in MATLAB. By the default configurations MATLAB shows 3 panes. The below figure shows MATLAB in action, and shows what each pane is:

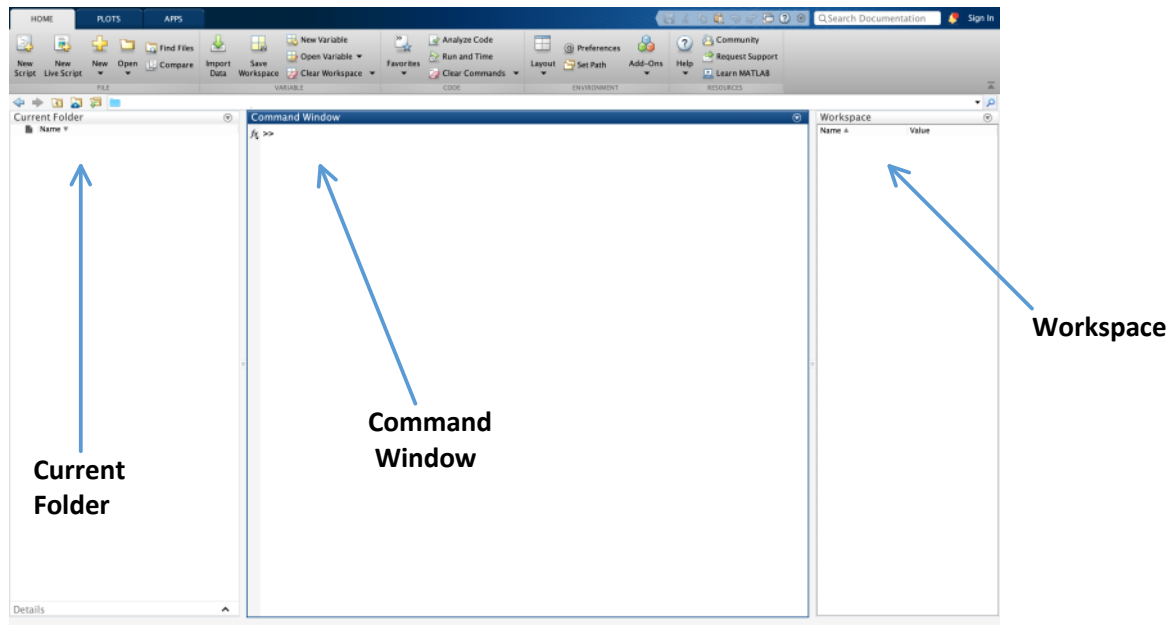


Figure 1.1

### **Current Folder:**

The pane on the left (labelled "Current Folder") allows you to browse through the files on your computer

### **Command Window:**

The pane in the center (labelled "Command Window") is the place where you enter commands that you want MATLAB to perform. Notice the prompt that will end with `>>`. That's the point where you type a command!

### **Workspace:**

The pane on the right (labelled "Workspace") serves several purposes. For beginners, its main functionality is that it displays the values of all of the current variables that you have defined. Note that you can edit the stored values by right-clicking the variable name.

You can add a new script by clicking the "New Script" button on the top-left corner, or use *control+n* (for Windows) or *command+n* (for Mac) as a shortcut.

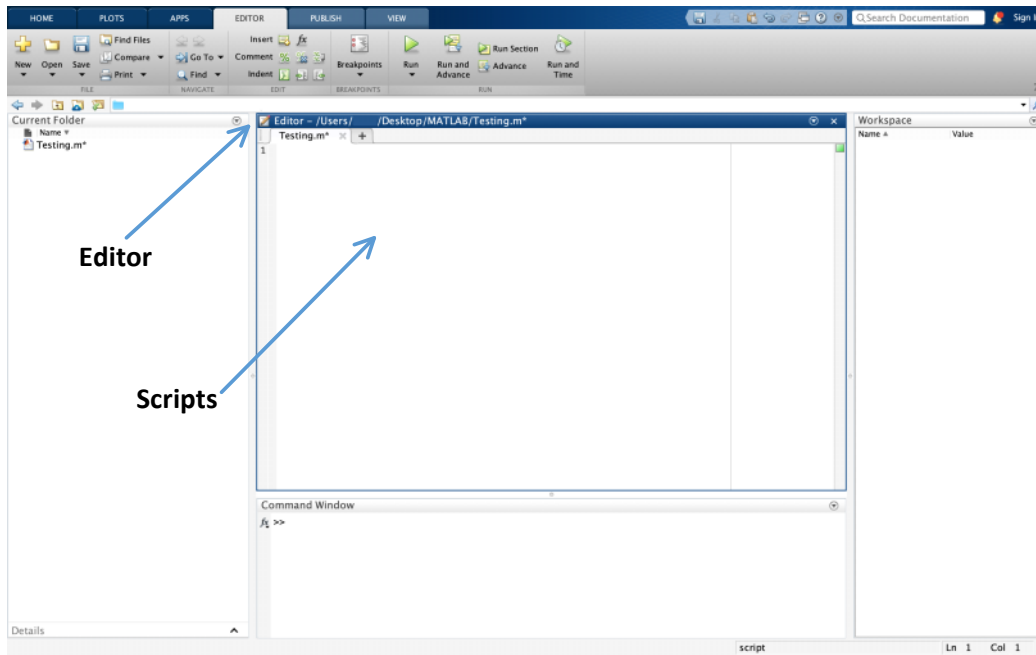


Figure 1.2

**Editor:**

The Editor pane includes all scripts, which are selectable as different tabs.

**Scripts:**

A script is a file that contains multiple sequential lines of MATLAB commands and function calls.

## Section 3: Basic MATLAB Commands and Functions

**M**ATLAB has a set of commands that can be directly typed into the command window. Commands such as quit, help, doc, stop and helpwin.

**quit:** 'quit' terminates the MATLAB program, note that it does not save your progress

**help:** 'help' will bring up a huge list of topics you can get help on. You can just click on the topics' links and it will show details on the selected topic. You can also ask for help on a specific MATLAB command. For example, to get help on the 'quit' command, you can type '*help quit*'.

**doc:** In addition to the help feature, you can access general documentation about the MATLAB program. Note that the amount of information contained here is encyclopedic and may be a bit overwhelming for beginners! But it's useful to have a complete documentation of the program available.

**stop:** The 'stop' function stops the execution of a MATLAB command. In other words, it breaks out of a computation.

**helpwin:** This command displays a more conventional graphical interface for the help command, 'helpwin' brings up a help window that looks something like this:

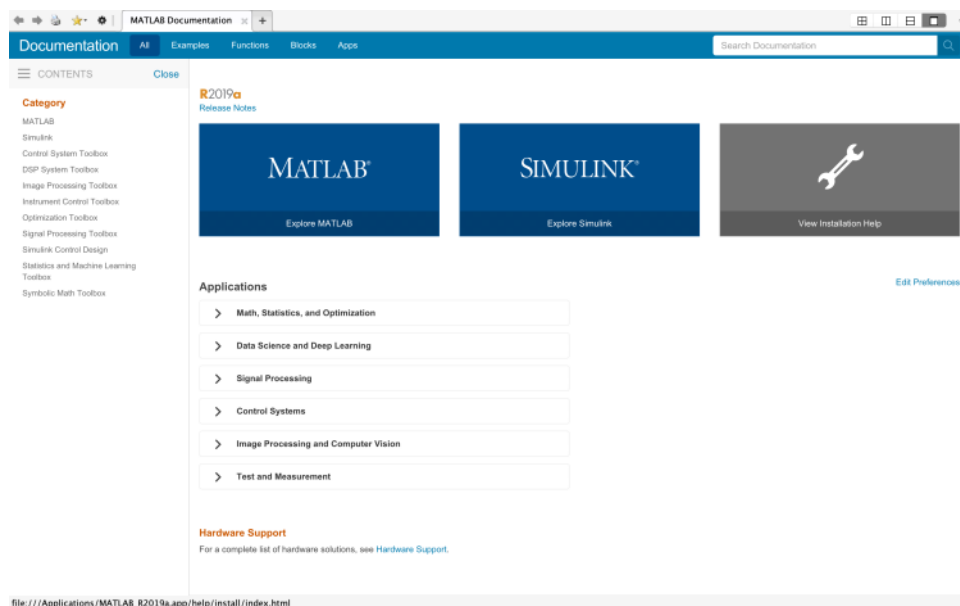


Figure 1.3

**Comments:** Computer programmers use comments throughout their code to explain their logic and their code within the code file. They do this so that in the case if they ever need to go back to their code, or if another developer looks at their code, they can understand it better. So, many computer languages allow the programmer to place comments within their code in various ways. MATLAB uses '%' before a line, to make the line a comment. In other words, placing the '%' character will exclude that line of code from compiling as a part of the program.

## Section 4: Importing, Saving, Loading, and Executing In MATLAB

In MATLAB, importing data is a very useful feature, which will be needed later on. To import a file, navigate to the top of the screen, make sure you are on the '**HOME**' tab, then click import data and select your file directory. Below is a figure displaying 'Import Data':

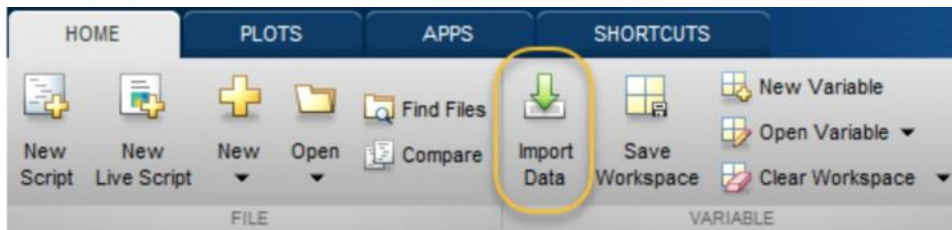


Figure 1.4

There are many supported file formats that can be useful for imports and exports. Some of these standard file formats are as listed below:

**Text Files:** MATLAB is able to read any text file extensions, such as 'TXT' and 'CSV'.

**Spreadsheets:** Microsoft excel spreadsheets, 'XLS', 'XLSM', 'XLSX', and more.

**Images:** 'JPEG', 'TIFF', 'PNG', 'ICO', 'GIF', 'JPG', 'JP2', and many more extensions.

**Scientific Data:** 'CDF', 'HDF', 'FITS', 'NetCDF', and other formats.

**Audio:** MATLAB can both read and write audio files, almost all file extensions are supported, some examples would be: 'AU', 'WAV', 'MP4', and 'M4A'.

**Video:** MATLAB can both play and record video files, almost all file extensions are supported, some examples are as follows: 'MPG', 'WMV', 'MP4', and 'MOV'.

As shown in Section 2, you can make a script where you would write your program in, and it would store all of your code within the script. MATLAB does not save your scripts automatically, so you will need to save them manually. To save a script, navigate to the '**EDITOR**' tab, and click save, or as a shortcut use *control+s* (for Windows) and *command+s* (for Mac). Note that the desired script in the Editor pane needs to be selected, and that the file extension will be '.m'. Below is a figure displaying the save button.



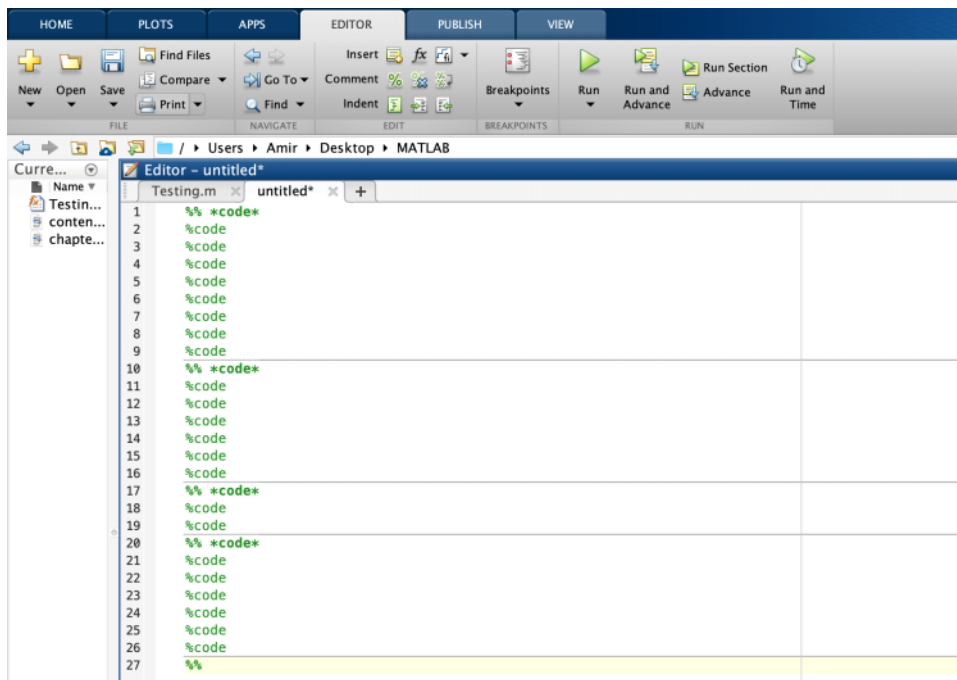


Figure 1.5

### Saving Variables into a 'MAT' file:

To save all workspace variables to a MAT-file, on the 'HOME' tab, in the 'Variable' section, click 'Save Workspace', and select the directory you want to save the file to.

You also can save workspace variables programmatically using the 'save' function. For example, to save all current workspace variables to the file Testing.mat, use the following command:

```
1 save('Testing')
```

And to save only variables A and B to the file Testing.mat, use the following command:

```
1 save('Testing', 'A', 'B')
```

### Loading Variables from a 'MAT' file:

To load saved variables from a 'MAT' file into your workspace, double-click the 'MAT' file in the Current Folder pane.

You also can load saved variables programmatically, use the 'load' function. For example, to load all variables from the file Testing.mat, use the following command:

```
1 load('Testing')
```

And to load only variables A and B from the file Testing.mat, use the following command:

```
1 load('Testing', 'A', 'B')
```

## Running The Program:

To run a program you have written inside a script, navigate to the '**EDITOR**' tab and under the '**Run**' section, click '**Run**'. This will first prompt you to save the script, and after the file is saved, it will compile and run the program.

Another way to quickly run a program without having to save every run is to use the '**Run Section**' button, or use *control+enter* (for Windows) or *command+enter* (for Mac). This will quickly run the section of the program you are trying to run, Below is a figure as an example of how the '**Run Section**' works.

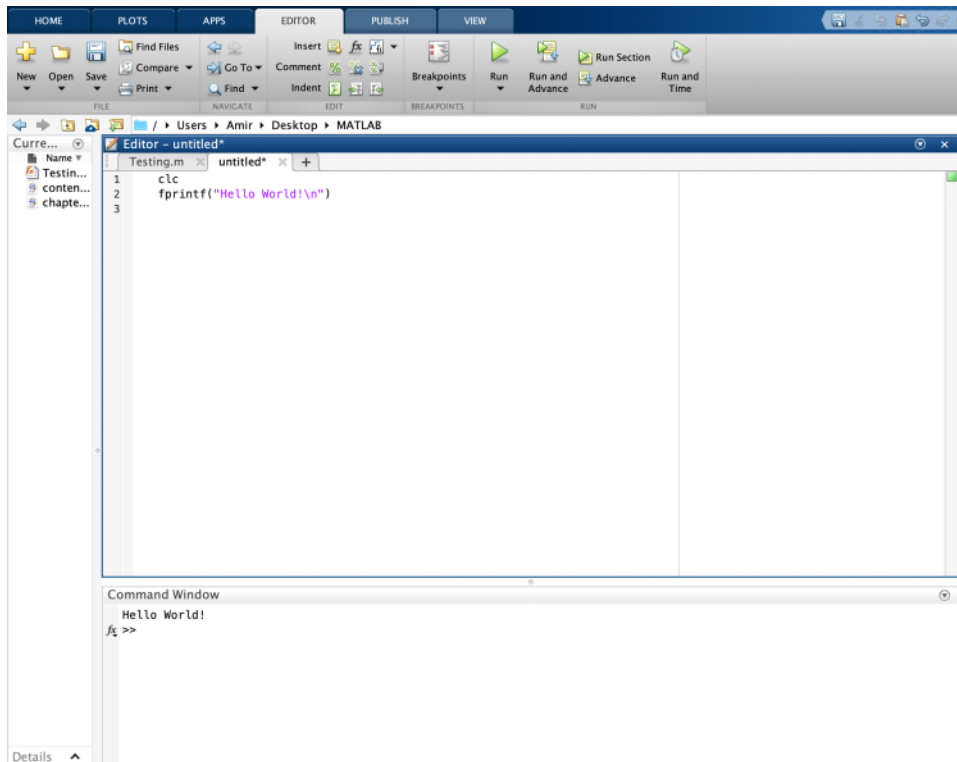


Figure 1.6

Note: The '`clc`' function at line 1 will clear the command window, so every time we compile the program, it will clear everything else that was previously displayed in the command window.

## Section 5: Real World Applications of MATLAB

**M**ATLAB is used in many different fields such as Mathematics, Physics, Engineering, Chemistry, Biology, Business, and Geoscience.

**Note: Do not be intimidated by the examples found in this section. They are meant to give you an idea on the great variety of tools MATLAB holds.**

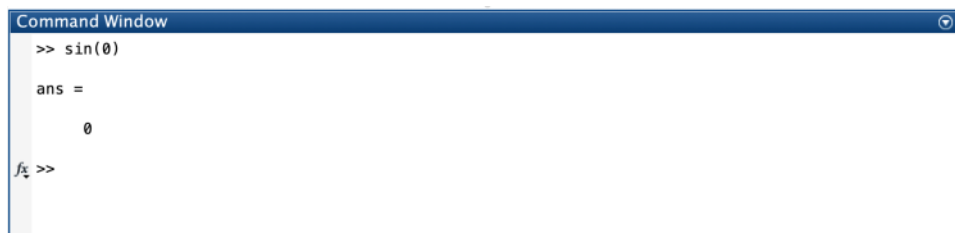
MATLAB is used in:

**Elementary Math:** Trigonometry, exponentials and logarithms, complex values, rounding, remainders, discrete math.

MATLAB, just like any other calculator, can just solve elementary mathematical problems by simply inputting the problem in the expected syntax, and MATLAB will provide an answer.

### Examples:

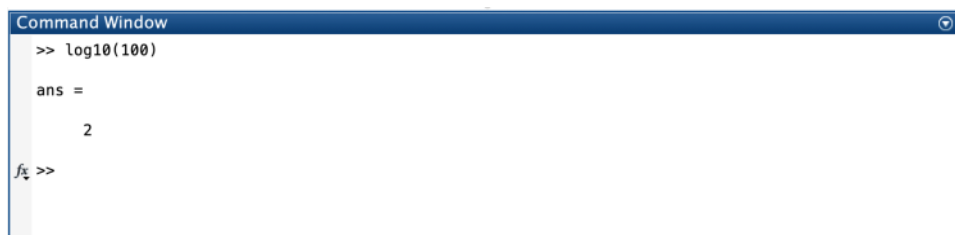
**Example 1.1:** Calculate  $\sin(0)$ , using MATLAB:



```
Command Window
>> sin(0)
ans =
     0
fx >>
```

Figure 1.7

**Example 1.2:** Calculate the  $\log_{10}(100)$ , using MATLAB:



```
Command Window
>> log10(100)
ans =
     2
fx >>
```

Figure 1.8

**Algebraic and Matrix Operations:** Linear equations, eigenvalues, singular values, decomposition, matrix operations, matrix structure.

**Example 1.3:** Solve  $e^x + x = 15$ , using MATLAB:

```

Editor - Untitled*
Untitled* x +
1  syms x
2  f = exp(x)+x-15;
3  z = solve(f,x);
4  fprintf("The solution to exp(x) + x = 15 is: %.7f\n\n",double(z))

Command Window
The solution to exp(x) + x = 15 is: 2.5238211

```

Figure 1.9

**Calculus and Differential Equations:** differentiation, integration, vector analysis, multiple integration.

**Example 1.4:** Solve  $y' - y = x$ , subject to  $y(0) = 1$ , using MATLAB:

```

Editor - Untitled*
Untitled* x +
1  clear all
2  syms y(x)
3  ode = diff(y(x),x) == y + x;
4  cond = y(0) == 1;
5  ySol(x) = dsolve(ode, cond);
6  fprintf("The solution to the DE is: %s\n\n",ySol(x))

Command Window
The solution to the DE is: 2*exp(x) - x - 1

```

Figure 1.10

**Fourier Analysis:** Fourier transforms, convolution, digital filtering.

**Example 1.5:** The `fft` function in MATLAB® uses a fast Fourier transform algorithm to compute the Fourier transform of data. Consider a sinusoidal signal  $x$  that is a function of time  $t$  with frequency components of 15 Hz and 20 Hz. Use a time vector sampled in increments of 150 of a second over a period of 10 seconds. Compute the Fourier transform of the signal, and create the vector  $f$  that corresponds to the signal's sampling in frequency space. When you plot the magnitude of the signal as a function of frequency, the spikes in magnitude correspond to the signal's frequency components of 15 Hz and 20 Hz.

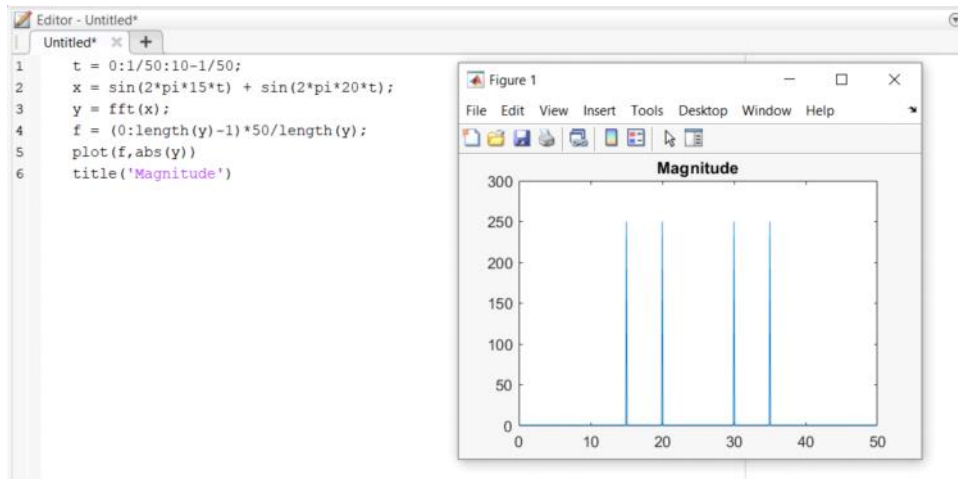


Figure1.11

**Graphs and Computational Geometry:** Directed and undirected graphs, network analysis, Triangulation, bounding regions, Voronoi diagrams, polygons.

**Example 1.6:** This code uses the Voronoi function to plot the Voronoi diagram for 10 randomly generated points.

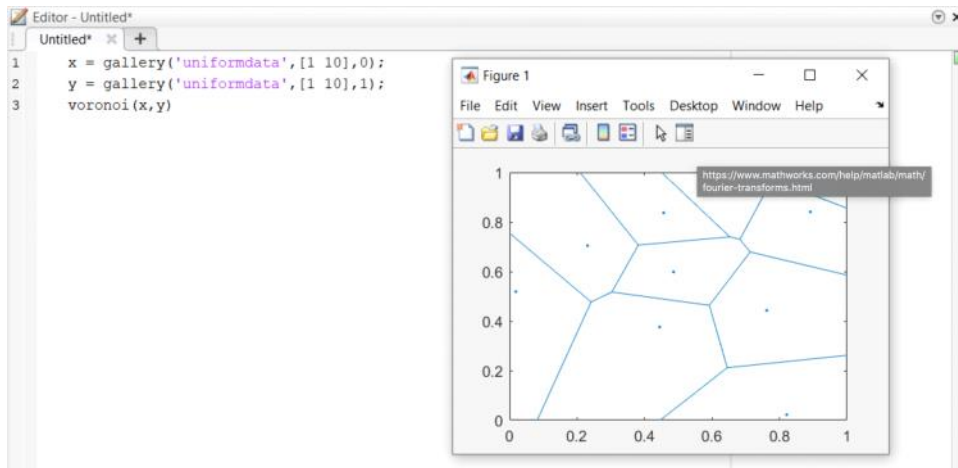


Figure 1.12

**Computational Biology:** Pharmacokinetics, bioinformatics, systems biology, bioimage processing, and biostatistics.

**Example 1.7:** MATLAB is used in Pharmacokinetics to model/simulate the movement of drugs within the human body. It can simulate the time course profile of drug exposure, drug efficacy, as well as enzyme and metabolite levels.

**Simulating the Real World:** Airflow, heat dispersion, sounds waves, friction, fluid dynamics and more, EEG.

**Example 1.8:** Movement of heat across different types of materials based on its intensity and duration allows for the creation of efficient and suitable tools for a variety of functions.

**Example 1.9:** MATLAB can be used to analyze the waves that are generated by the brain, and can produce intelligible data.

**Nature Analysis:** Analyzing seismic activity, climate change, coral reef growth, and weather patterns.

**Example 1.10:** Processing and visualizing weather patterns with MATLAB allows for better prediction of weather and even potential natural disasters when paired MATLAB's machine learning capabilities.

**Cost and Benefit Analysis:** Processing the net cost and benefit of different investments over time.

**Example 1.11:** Create an amortization schedule based on duration, mortgage rate, and payment rate. In addition, MATLAB allows the user to visualize this data by plotting the outstanding balance, cumulative principle, and cumulative interest.

## Student Exercises

1. Where did the name MATLAB come from?
2. What Language was MATLAB originally created in?
3. By the default, how many panes that MATLAB display? What are the panes called?
4. What was MATLAB originally for?
5. What does the 'quit' command do? Does it automatically save your progress?
6. What is the difference between the 'quit' command and the 'stop' command?
7. 'fprintf' is a command that will print out information which we will learn more about in the future chapters, if we wanted to get help from MATLAB on how to use 'fprintf' what command would we need to type?
8. What does the 'helpwin' command do?
9. Which character is used for commenting in MATLAB?
10. Write a line of code to save all the workspace variables into a 'MAT' file named "Practice".

# CHAPTER 2

## Data Types and Basic Evaluations:

Section 1: Numeric Data Types

Section 2: Characters and Strings

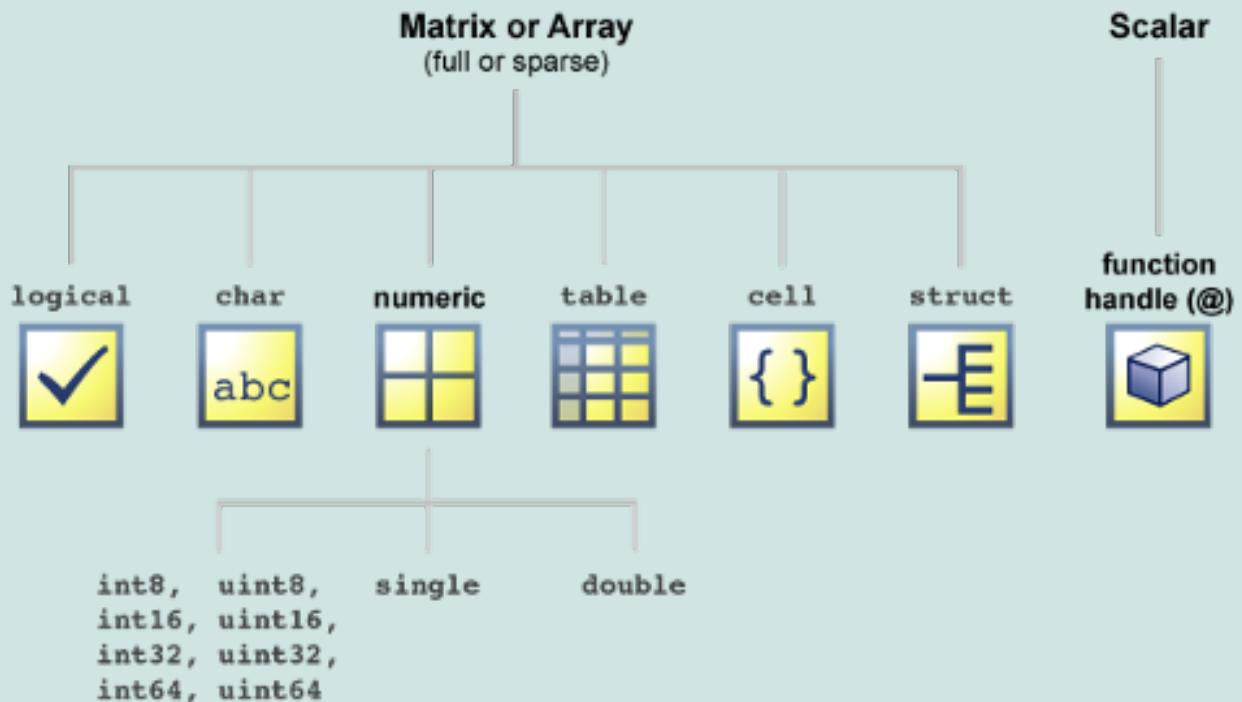
Section 3: Conversion and identification

Section 4: Basic Numeric Operations

Section 5: Input and Output

**Practice Examples**

**Student Exercises**





## Section 1: Numeric Data Types

**N**umeric data types are used to assign decimals and whole numbers to a certain variable. In MATLAB, integers come in four different types that can be **signed** and **unsigned**. Signed integers can hold both positive and negative values but instead hold a smaller range compared to unsigned integers which can only be positive. Table 2.1 displays all eight integer types as well as the range of values they can store.

Integer Type	Range of Values
int8	$-2^7$ to $2^7-1$
int16	$-2^{15}$ to $2^{15}-1$
int32	$-2^{31}$ to $2^{31}-1$
int64	$-2^{63}$ to $2^{63}-1$
uint8	0 to $2^8-1$
uint16	0 to $2^{16}-1$
uint32	0 to $2^{32}-1$
uint64	0 to $2^{64}-1$

Table 2.1

In order to store an integer into a variable, the following syntax can be used:

```
1 myInt = int16(523)
```

**Note** that entering a negative integer into an unsigned type returns 0 and entering a decimal results in a rounded number. Also **note** that you do not always have to define the data type of a variable.

Decimal values can be stored into two different data types (**double** and **single**) with the major difference being that doubles (64-bits) are bigger than singles (32-bits). If a number is stored without specifying the data type, MATLAB assumes the number is a double.

## Section 2: Characters and Strings

**M**uch like numerical data types, there are character and string data types. Character data types store both characters and numbers, but the numbers cannot be used in computations. To create a character variable, use `'char'` before the variable. For example to create a character variable named "x", we do the following:

```
1 char 'x'  
2 %OR  
3 char x
```

**Note** that placing a space between a series of characters will create an array of characters which we will learn about in future chapters.

To assign the value "h" to a character named 'height', we do:

```
1 height = char ('h')  
2 %OR  
3 height = 'h'
```

String data types store a series of words and characters. The main difference between a string and a character is that string data types take up more space in memory compared to characters. Another difference is that, we can add to strings but not to a character. To create a string variable, we use `'string'` before the variable name. For example to create a string variable named "firstName", we do the following:

```
1 string 'firstName'  
2 %OR  
3 string firstName
```

To assign the value "Chris" to a string named 'firstName', we do:

```
1 firstName = string('Chris')  
2 %OR  
3 firstName = "Chris"
```

The Table below shows how much space in memory strings and characters take: Note that spaces count as a character.

Integer Type	Space in Memory
Strings of length 0-10	132 bytes
Strings of length 11-15	142 bytes
Strings of length 16 and higher	142 bytes + 16 bytes for each 8 characters or fewer
Characters	Each character takes 2 bytes

Table 2.2

In MATLAB we can add to strings using the `append` function, we can concatenate two strings and combine them into one. For example, if we wanted to greet a person, we would have two strings. The first string is named `'greeting'` and the second is `'name'`. The value `'greeting'` holds is "Hello" and the value `'name'` holds is "Chris". To concatenate these two strings into a new string named `'str'`, we do the following:

```
1 str = append(greeting, name)
```

**Output:**

```
str = "HelloChris"
```

Appending two strings will not leave a space between them, so we will need to add a space character in between, in coding terms:

```
1 greeting = "Hello";  
2 str = append(greeting, ' ', name)
```

**Output:**

```
str = "Hello Chris"
```

An alternative to appending strings without using the `append` function is to use the `+` operator. For example:

```
1 str = greeting + ' ' + name
```

**Output:**

```
str = "Hello Chris"
```

## Section 3: Conversion and Identification

**M**ATLAB allows us to convert different data types, for example you can turn a number into a string value, and inversely, you can also turn a string into a numerical value. You can use predefined MATLAB functions to do these conversions, below are the conversion functions:

**Converting from a character to a string:** If you have a character named 'char' and you want to convert that to a string named 'str', you can use the '*convertCharsToStrings*' function as seen in the example below:

```
1 str = convertCharsToStrings(char)
```

**Converting from a string to a character:** If you have a string named 'str' and you want to convert that to a character named 'char', you can use the '*convertStringsToChars*' function as seen in the example below:

```
1 char = convertStringsToChars(str)
```

**Converting from a string to a double:** To convert a string to a double, the string's value needs to be numerical. If you have a string named 'PI' which holds the value "3.14" and you want to convert that to a double named 'x', you can use the '*str2double*' function as seen in the example below:

```
1 x = str2double(PI)
```

**Converting from a double to a string:** If you have a double named 'x' which holds the value "3.14" and you want to convert that to a string named 'PI', you can use the '*num2str*' function as seen in the example below:

```
1 PI = num2str(x)
```

## Section 4: Basic Numeric Operations

**M**ATLAB supports all basic mathematical operations such as addition, subtraction, multiplication, division, exponents, and more. The symbol used for each mathematical operation can be found in table 2.2.

Symbol	Function Name	Role
+	plus	Addition
-	minus	Subtraction
*	times	Multiplication
/	rdivide	Division
^	power	Exponent
N/A	sqrt	Square Root
N/A	rem	Remainder
N/A	round	Round
N/A	mod	Modulus

Table 2.3

**Note** that MATLAB follows the order of operations so that the below code gives an output of 192.

```
1 x = 5
2 y = 6
3 1 + x * y ^ 2 + (x + y)
```

MATLAB will give an error if the user tries to add integers of different types or add integers with singles.

```
1 x = int16(5);
2 y = int32(4);
3 x + y
```

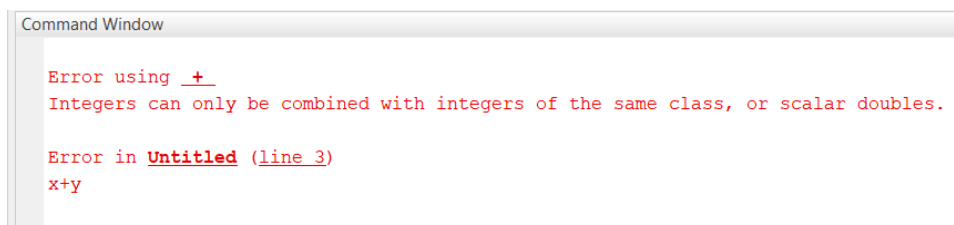


Figure 2.1

Doubles and integers however, can be added with no issues. The result of a double and an integer is another integer of the same type.

```
1 x = double(5);
2 y = int32(4)
3 x + y
```



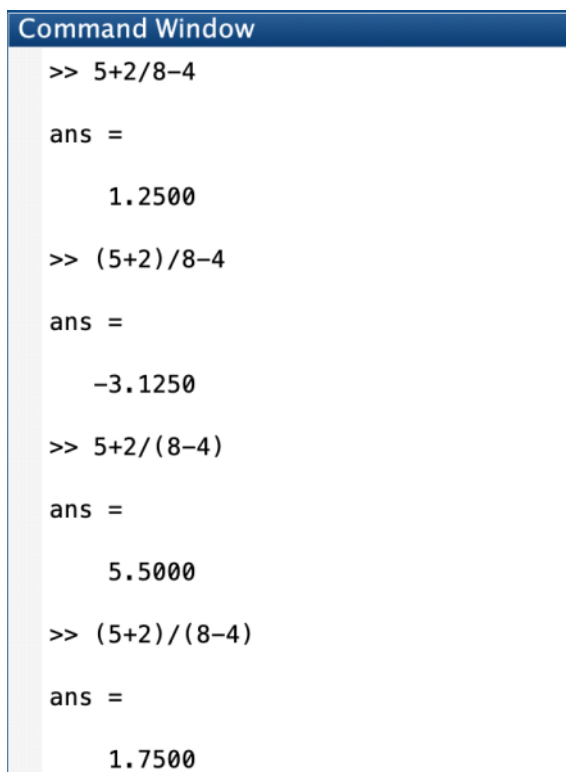
```
Command Window
ans =

    int32

     9
```

Figure 2.2

The correct use of parenthesis is very important in operations. For example to do  $\frac{5+2}{8-4}$ , it is very important to use parenthesis to separate the operations. The incorrect use of parenthesis effects our answers as shown below:



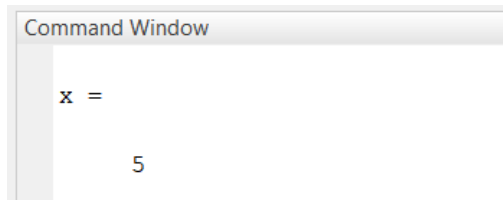
```
Command Window
>> 5+2/8-4
ans =
    1.2500
>> (5+2)/8-4
ans =
   -3.1250
>> 5+2/(8-4)
ans =
    5.5000
>> (5+2)/(8-4)
ans =
    1.7500
```

Figure 2.3

We know the correct answer to this operation is **1.75** but as you can see, without parenthesis we would get various incorrect answers.

## Section 5: Input and Output

There are three different ways to output text in MATLAB. The first and most simple way is to not put semicolons at the end of a statement. Meanwhile, the same code without ";" results in the output seen in figure 2.3.

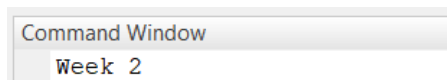


```
Command Window
x =
    5
```

Figure 2.4

The second output method involves using the function **disp()**. Disp can print variables, numbers, and strings. Adding a number and string with "+" causes them to concatenate to each other.

```
1 weekNum = 2;
2 disp("Week " + weekNum);
```

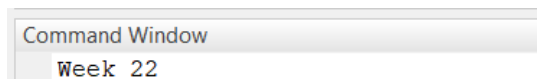


```
Command Window
Week 2
```

Figure 2.5

Adding two numbers in disp can cause them to either concatenate or mathematically add depending on the situation. If text comes before the addition then the numbers will concatenate, otherwise they will add. Thus, in figure 2.5, the numbers join into a 22 rather than add up to a four.

```
1 disp("Week " + 2 + 2);
```



```
Command Window
Week 22
```

Figure 2.6

On the other hand, the numbers add up to a 4 in figure 2.6 since the text appears after the addition.

```
1 disp(2 + 2 + " weeks left!");
```



```
Command Window
4 weeks lefts!
```

Figure 2.7

The final method for outputting text in MATLAB uses the function **fprintf()**. Simply printing string with fprintf shows that the function does not automatically move on to the next line every time it is used.

```
1 fprintf("Hi");
2 fprintf(", how are you?");
```

```
Command Window
Hi, how are you?
```

Figure 2.8

Inserting line breaks requires special formatting, a list of which can be found in table 2.4.

Format Name	Symbol
Single quotation mark	"
Percent character	%%
Backslash	\\
Alarm	\a
Backspace	\b
Form feed	\f
New line	\n
Carriage return	\r
Horizontal tab	\t
Vertical tab	\v

Table 2.4

Variables can still be concatenated using the "+" operator, however, they can also be added within the string using a special format. Placing "%i" inside the text and the variable name on the outside allows for this to happen as seen in figure 2.8.

```

1 age = int16(25);
2 fprintf("Hi\n");
3 fprintf("I am %i years old!", age);

```

```
Command Window
Hi
I am 25 years old!
```

Figure 2.9

Other markers like %i can be found in table 2.5.



Specifier	Description
%c	Single character
%d	Decimal notation (signed)
%e	Exponential notation (using a lowercase e as in 3.1415e+00)
%E	Exponential notation (using an uppercase E as in 3.1415E+00)
%f	Fixed-point notation
%g	The more compact of %e or %f, as defined in [2]. Insignificant zeros do not print.
%G	Same as %g, but using an uppercase E
%i	Decimal notation (signed)
%o	Octal notation (unsigned)
%s	String of characters
%u	Decimal notation (unsigned)

Table 2.5

To ask the user for an input, we use the **input()** command. Input prints the prompt you give it and waits for the user to enter a value. By placing an 's' we tell the computer that the user will enter a string.

```
1 name = input("\nWhat is your name? ", 's');
2 age = input("How old are you?");
3 fprintf("Hi %s,\n", name);
4 fprintf("I am also %i years old!\n", age);
```

```
Command Window
What is your name? Amir
How old are you? 537
Hi Amir,
I am also 537 years old!
```

Figure 2.10

To display a double to a specific number of decimal places using %f, we place the number of decimal places before the 'f'. To display a number in 2 decimal places, we can do %.2f. The example below shows how to display the value of 'pi' in different specific number of decimal places.

```
1 a = pi;
2 fprintf("The value of pi is: %f \nTo 2 decimal places: %.2f \nTo 4 decimal places: %.4f\n", a, a, a);
```

```
Command Window
The value of pi is: 3.141593
To 2 decimal places: 3.14
To 4 decimal places: 3.1416
fx >>
```

Figure 2.11

To round a number to the nearest integer or to a certain amount of decimal places, we can use

the `'round'` function, the syntax is as shown below:

`round(x)`: will round the value of `x` to the nearest integer.

`round(x, n)`: will round the value of `x` to `n` amount of decimal places.

- `n > 0`: round to `n` digits to the right of the decimal point.
- `n = 0`: round to the nearest integer.
- `n < 0`: round to `n` digits to the left of the decimal point.

```
1 round(4.4998, 4)
```

```
2 round(3.5)
```

```
3 round(1.2464, 2)
```

Command Window

```
ans =
```

```
4.4998
```

```
ans =
```

```
4
```

```
ans =
```

```
1.2500
```

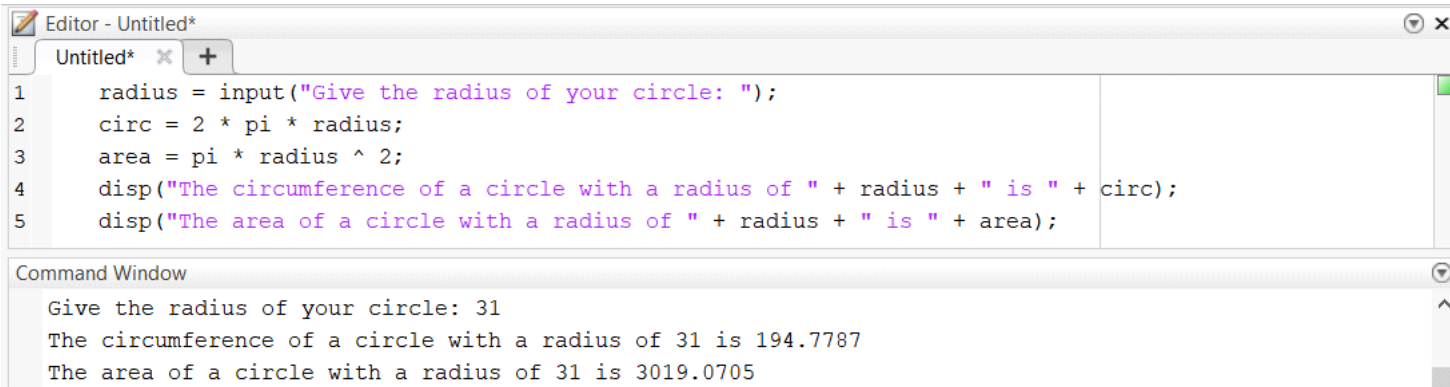
Figure 2.12

## Practice Examples

### Practice Example 2.1

Create a program that asks the user for the radius of a circle and prints out its area and circumference.

### Solution 2.1



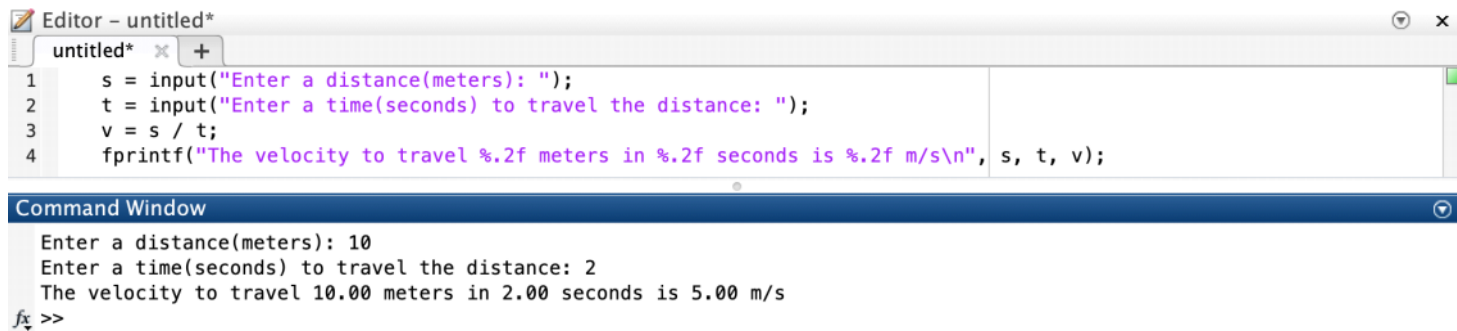
```
Editor - Untitled*
Untitled* x +
1 radius = input("Give the radius of your circle: ");
2 circ = 2 * pi * radius;
3 area = pi * radius ^ 2;
4 disp("The circumference of a circle with a radius of " + radius + " is " + circ);
5 disp("The area of a circle with a radius of " + radius + " is " + area);

Command Window
Give the radius of your circle: 31
The circumference of a circle with a radius of 31 is 194.7787
The area of a circle with a radius of 31 is 3019.0705
```

### Practice Example 2.2

Create a program that asks the user to input a distance (m) and how long it takes to travel that distance (s), and it will calculate and print the velocity (m/s) using the following equation:

$$\bar{v} = \frac{\Delta s}{\Delta t}$$



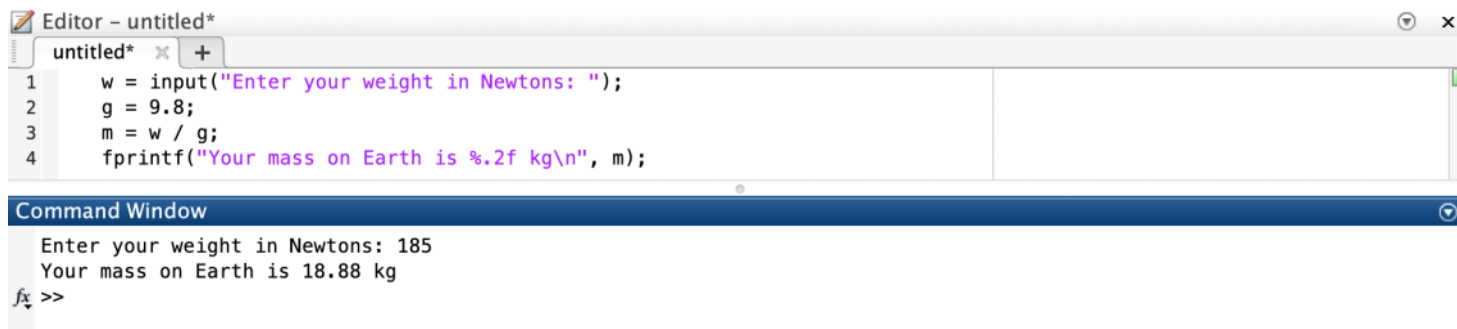
```
Editor - untitled*
untitled* x +
1 s = input("Enter a distance(meters): ");
2 t = input("Enter a time(seconds) to travel the distance: ");
3 v = s / t;
4 fprintf("The velocity to travel %.2f meters in %.2f seconds is %.2f m/s\n", s, t, v);

Command Window
Enter a distance(meters): 10
Enter a time(seconds) to travel the distance: 2
The velocity to travel 10.00 meters in 2.00 seconds is 5.00 m/s
fx >>
```

### Practice Example 2.3

Create a program that asks the user to input their weight (N) and assuming they are on Earth, it will calculate and print their mass (kg) using the following equation:

$$w = mg$$



```
Editor - untitled*
untitled* x +
1 w = input("Enter your weight in Newtons: ");
2 g = 9.8;
3 m = w / g;
4 fprintf("Your mass on Earth is %.2f kg\n", m);

Command Window
Enter your weight in Newtons: 185
Your mass on Earth is 18.88 kg
fx >>
```

## Student Exercises

1. What is the difference between int and double?
2. Create a variable named "price" that is set to be 249.
3. How many bytes would the string "There are 7 days in a week" be?
4. You are given the strings shown below:  
forecast = "The weather is";  
weather = "sunny";  
date = "Sunday, June 16th, 2019";  
Write a line of code that will print out the following:  
"The weather is sunny today, Sunday, June 16th, 2019"
5. The Earth's gravity is 9.8 m/s<sup>2</sup>, the variable named 'gravityValue' holds the value of "9.8". Convert the double to a string and name it "gravityOfEarth."
6. Create a program that takes the length and width of a rectangle as input and outputs its perimeter and area.
7. Create a program that takes in a side and angle of a right triangle and returns the length of its other sides.
8. Create a program that accepts temperature input in degrees Fahrenheit, Celsius, or Kelvin then converts from one to the other.
9. Write a program that will ask the user to input a velocity (m/s) and a time (s), and will calculate and print the accelerations (m/s<sup>2</sup>). Use the following equation:

$$\bar{a} = \frac{\Delta v}{\Delta t}$$

# CHAPTER 3

## Algebraic Operations:

Section 1: Simplifying

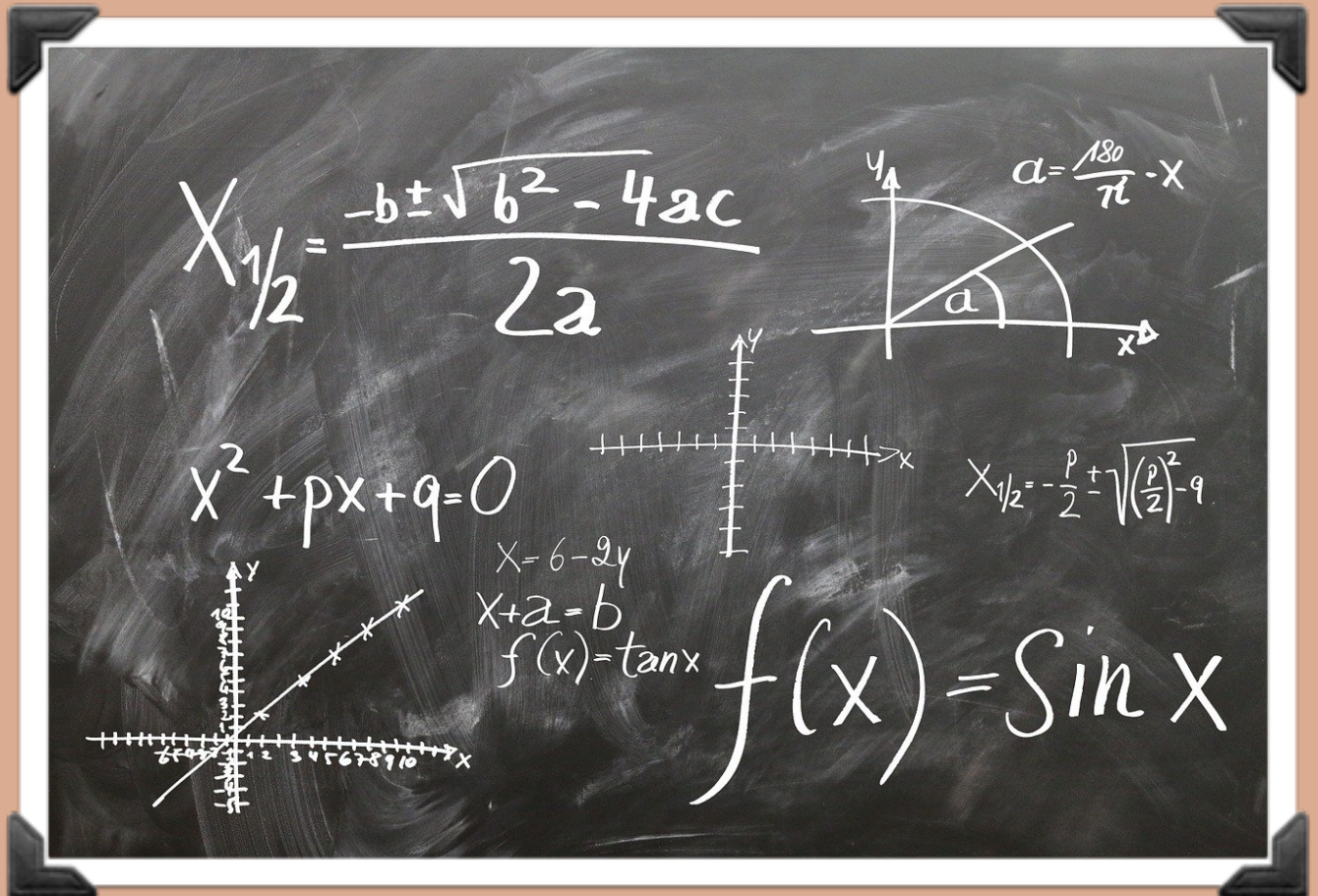
Section 2: Expanding

Section 3: Factoring

Section 4: Solving Equations

Section 5: Partial Fraction Decomposition

**Student Exercises**



## Section 1: Simplifying

**S**implification of a mathematical expression is not a clearly defined subject. There is no universal idea as to which form of an expression is simplest. For example, the following two mathematical expressions present the same in different forms:

$$x^4 - 2x^3 - 13x^2 + 14x + 24$$

$$(x + 1)(x - 2)(x + 3)(x - 4)$$

MATLAB allows us to do many algebraic operations, one of many is simplifying. The prebuilt '*simplify*' function will perform an algebraic simplification on an expression. The correct syntax for this function is as follows:

```
1 simplified = simplify(EXPRESSION)
```

As an example of how to use the '*simplify*' function, simplify the expression:  $\sin(x^2) + \cos(x^2)$

Initially use the '*syms*' command to define the variable **x** :

```
1 syms x
2 simplify(sin(x)^2 + cos(x)^2)
```

As a result, we should get 1.

## Section 2: Expanding

**M**uch like simplification, MATLAB also expands expressions. For example, looking at the same expression from before, the simplified expression,  $(x + 1)(x - 2)(x + 3)(x - 4)$  can be expanded into  $x^4 - 2x^3 - 13x^2 + 14x + 24$ . The prebuilt '*expand*' function will perform an algebraic expansion on an expression. The correct syntax for this function is as follows:

```
1 expanded = expand( EXPRESSION )
```

As an example of how to use the '*expand*' function, expand the expression:  $(x+1)^3$

Initially use the '*syms*' command to define the variable **x** :

```
1 syms
2 expand( (x+1)^3 )
```

As a result, we should get  $x^3 + 3x^2 + 3x + 1$ .

In other words, **expand(S)** multiplies all parentheses in S, and simplifies inputs to functions such as  $\cos(x+y)$  by applying standard identities.

## Section 3: Factoring

**F**actoring in the simplest terms is finding what to multiply together to get an expression. It is much like splitting an expression into a multiplication of simpler expressions. For example, to factor  $2y + 6$ :

Both  $2y$  and  $6$  have a common factor of 2, therefore  $2y + 6$  can be written as  $2(y + 3)$ .

MATLAB's '*factor*' function returns the most reduced simplified values. In other words **factor(x)** returns all irreducible factors of x. If x is an integer, factor returns the prime factorization of x. If x is a symbolic expression, factor returns the subexpressions that are factors of x. The correct syntax to use the '*factor*' function is as follows:

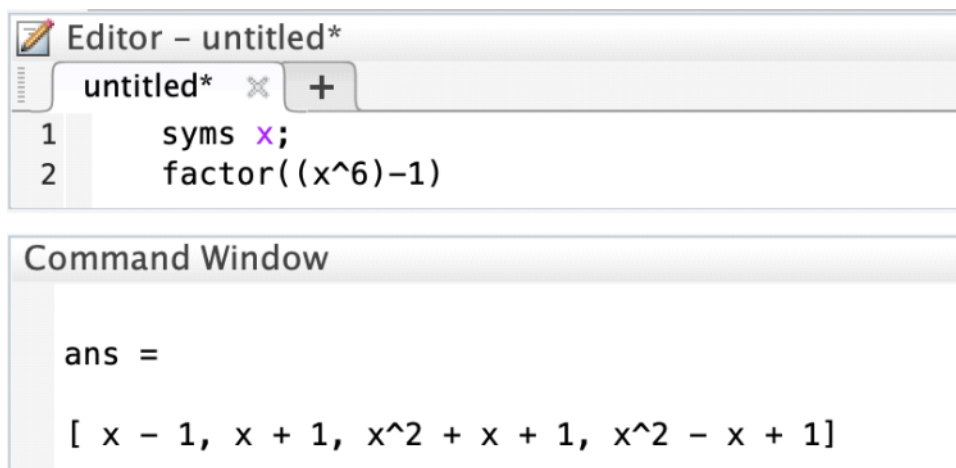
```
1 factor(VALUE/EXPRESSION)
```

For example, to factor the integer value  $6$ , we would do:

```
1 factor(6)
```

As a result we would get 2 and 3, as they are the most reduced values from 6, and the multiplication of 2 and 3 would be 6.

Factoring an expression uses the same syntax, except we have to use the '*syms*' function to define the variables. To factor the expression  $(x^6 - 1)$ , we do the following:



```
Editor - untitled*
untitled* x +
1 syms x;
2 factor((x^6)-1)

Command Window

ans =

[ x - 1, x + 1, x^2 + x + 1, x^2 - x + 1]
```

Figure 3.1

You can also factor based on a specific variable.  $F = \text{factor}(x, \text{vars})$  returns an array of factors  $F$ , where **vars** specifies the variables of interest. All factors not containing a variable in **vars** are separated into the first entry  $F(1)$ . The other entries are irreducible factors of  $x$  that contain one or more variables from **vars**.

For example, to factor out  $x$ , from  $y^2x^2$ , you can do the following:



```
Editor - untitled*
untitled* x +
1     syms x y;
2     factor((y^2) * (x^2), x)

Command Window

ans =

[ y^2, x, x]
```

Figure 3.2

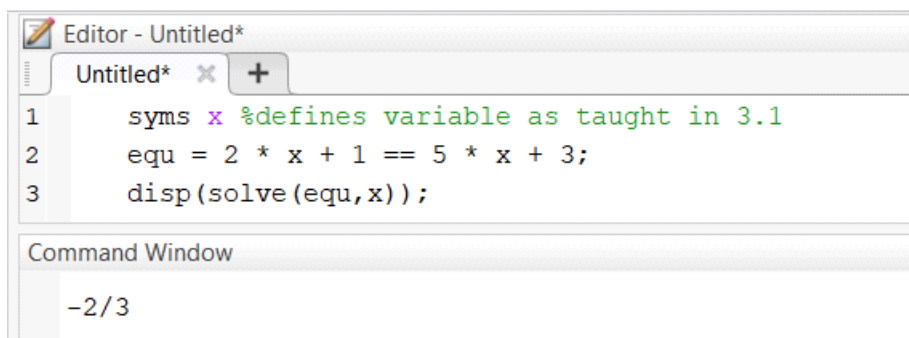
## Section 4: Solving Equations

Solving equations first requires knowing how to express equations in MATLAB. In mathematics, an equation is a statement formed from combining two expressions as a sign of their equality. While in mathematics the symbol for the equal sign is '=', in MATLAB the equal sign is formed with '=='. So to store the equation  $2x + 1 = 5x + 3$  in a variable, we do the following:

```
1 equ = 2 * x + 1 == 5 * x + 3
```

For MATLAB to solve this equation, we can use the 'solve' function which follows the below format

```
1 solve(equation, variable)
```



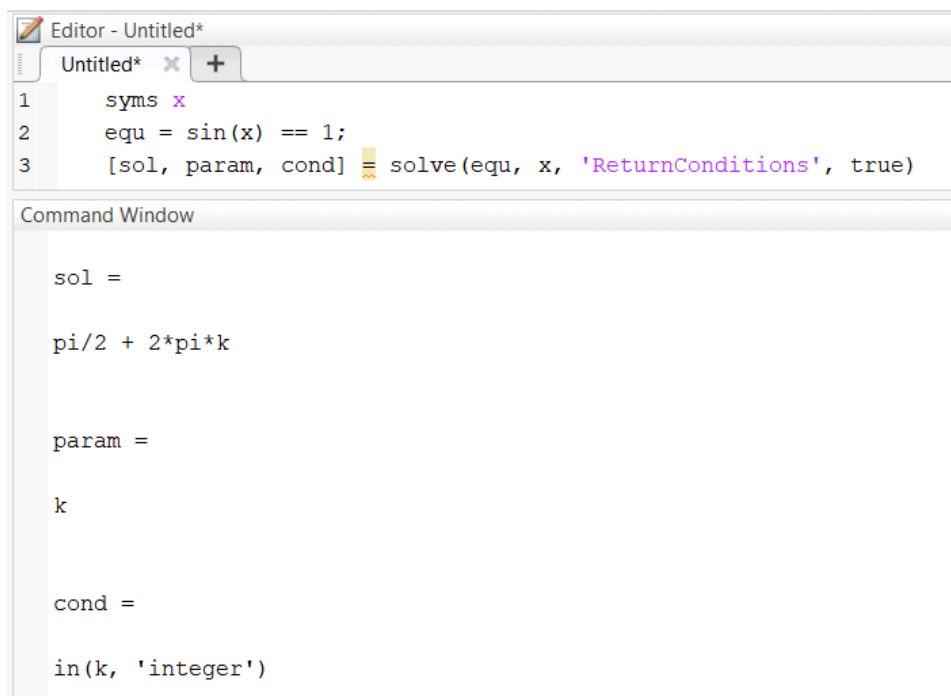
```
Editor - Untitled*
Untitled* x +
1     syms x %defines variable as taught in 3.1
2     equ = 2 * x + 1 == 5 * x + 3;
3     disp(solve(equ,x));

Command Window
-2/3
```

Figure 3.3

An equation like  $\sin(x) = 1$  can have an infinite number of solutions. By adding the parameters 'ReturnConditions' and true, the solve function can be modified to handle this.

```
1 solve(equation, variable, 'ReturnConditions', true)
```



```
Editor - Untitled*
Untitled* x +
1     syms x
2     equ = sin(x) == 1;
3     [sol, param, cond] = solve(equ, x, 'ReturnConditions', true)

Command Window
sol =
pi/2 + 2*pi*k

param =
k

cond =
in(k, 'integer')
```

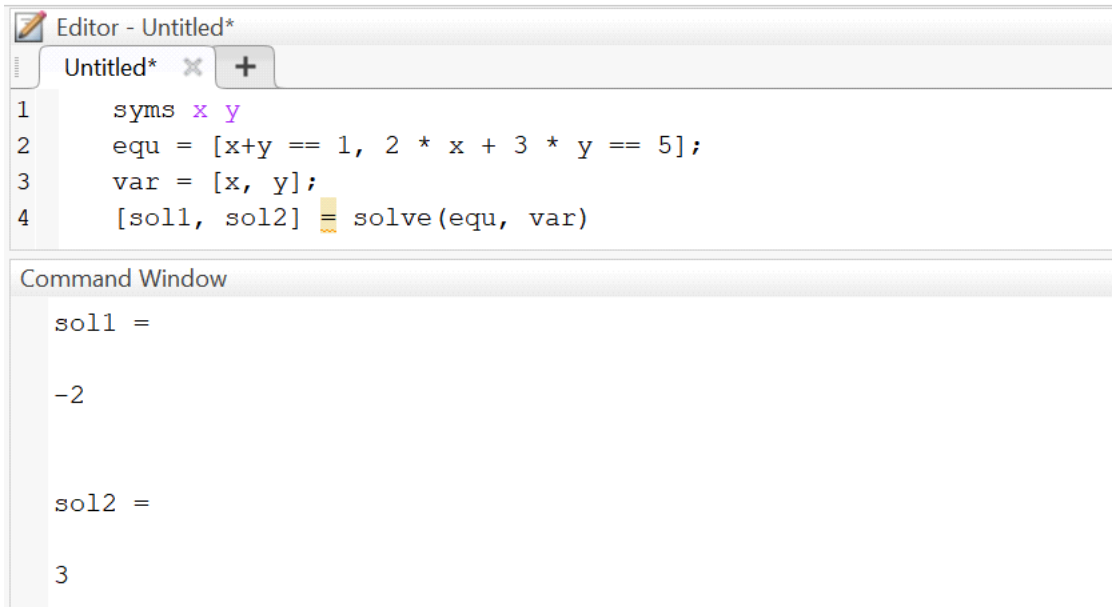
Figure 3.4

The new version of the solve function returns three variables. The first variable is the **solution**. If the solution repeats then MATLAB displays the solution as an expression with a **parameter** and defines the **conditions** for that parameter. In the case of figure 3.4, the solution to  $\sin(x) == 1$  is  $\pi/2 + 2 * \pi * k$  where the parameter **k** must be an integer.

**Note** that if no solution exists then MATLAB will return an empty object and if MATLAB returns an empty object with a warning, then there is a possibility that a solution exists.

To solve multivariate equations, we must first store all the equations in a single array (arrays will be discussed further in a later chapter). For example, the equations,  $x + y = 1$  and  $2x + 3y = 5$  would be stored in the following way:

```
1 equ = [x + y == 1 , 2 * x + 3 * y == 5]
2 var = [x, y]
```



The screenshot shows the MATLAB Editor window with the following code:

```
1 syms x y
2 equ = [x+y == 1, 2 * x + 3 * y == 5];
3 var = [x, y];
4 [sol1, sol2] = solve(equ, var)
```

The Command Window displays the output:

```
sol1 =
-2
sol2 =
3
```

Figure 3.5

## Section 5: Partial Fraction Decomposition

In mathematics, partial fraction decomposition (PFD) occurs when a fraction of polynomials is converted into the sum of two simpler fractions. MATLAB uses the function named **residue** to accomplish this.

Finding the PFD:

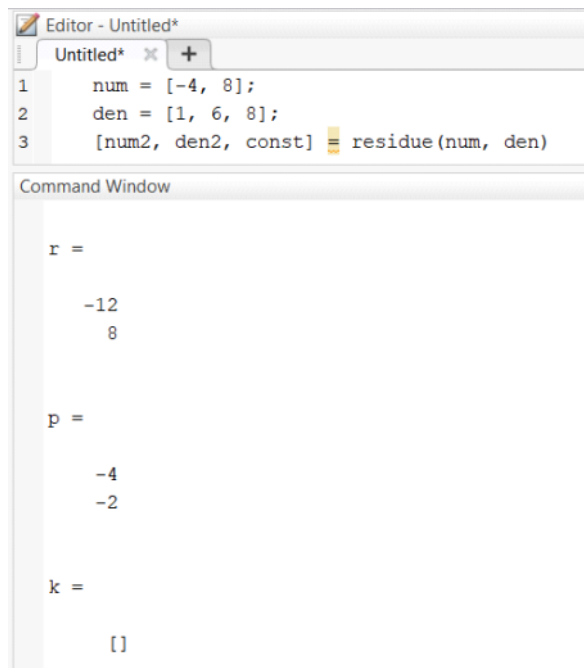
```
1 [numerator2, denominator2, constant] = residue(array of numerator values, array of denominator values)
```

Reverting back:

```
1 [array of numerator values, array of denominator values] = residue(numerator2, denominator2, constant)
```

For example, let's say we have the fraction:  $\frac{-4x + 8}{x^2 + 6x + 8}$

We would put all the constants of the numerator into one array and all the constants of the denominator into another array (while maintaining order) and the full code would be:



```
Editor - Untitled*
Untitled* x +
1 num = [-4, 8];
2 den = [1, 6, 8];
3 [num2, den2, const] = residue(num, den)

Command Window

r =

    -12
     8

p =

    -4
    -2

k =

    []
```

Figure 3.6

The output given by figure 3.6 means  $\frac{-4x + 8}{x^2 + 6x + 8} = \frac{-12}{x + 4} + \frac{8}{x + 2}$

## Student Exercises

1. Simplify the following equation using MATLAB:

$$\frac{\sin(2x)(\cos(x)^2 - \sin(x)^2)(e^{2x} - 2e^x + 1)}{(\cos(2x)^2 - \sin(2x)^2)(e^{2x} - 1)}$$

2. Expand the following equation in MATLAB:

$$e^{(a+b)^2}$$

3. Factor the following equations using MATLAB:

- $9x^2 + 12x + 4$
- $24x^2 - 6xy - 9y^2$
- $-2x^3 - 6x^2 + 56x$
- Factor out **a**, from  $a^3b^8 - 7a^{10}b^4 + 2a^5b^2$

4. Solve the equations below:

- $3x^2 - \frac{1}{2}x - 7 = 3$
- $e^{3x^2} = 3$
- $x \log(x^2 + 12x + 3) = 10$

5. Find all solutions to the equations below (be sure to state the condition if necessary):

- $\cosh(x) + \cos(2x) = 10$
- $$\begin{cases} \tan(3x + 1) - 2y = 10 \\ \tan(7x) + 7y = 23 \end{cases}$$
- $$\begin{cases} 5a + 3b - 10c = 0 \\ 8a + 17b = 12c \\ 18a + 20b = 6c \end{cases}$$

6. Expand the fractions below:

- $\frac{8x - 42}{x^2 + 3x - 18}$
- $\frac{9x + 25}{(x + 3)^2}$
- $\frac{3x^3 + 7x - 4}{(x^2 + 2)^2}$

# CHAPTER 4

## Conditions and Loops:

Section 1: Conditional Statements

Section 2: Basic Boolean Operations

Section 3: Switch, Case, Otherwise

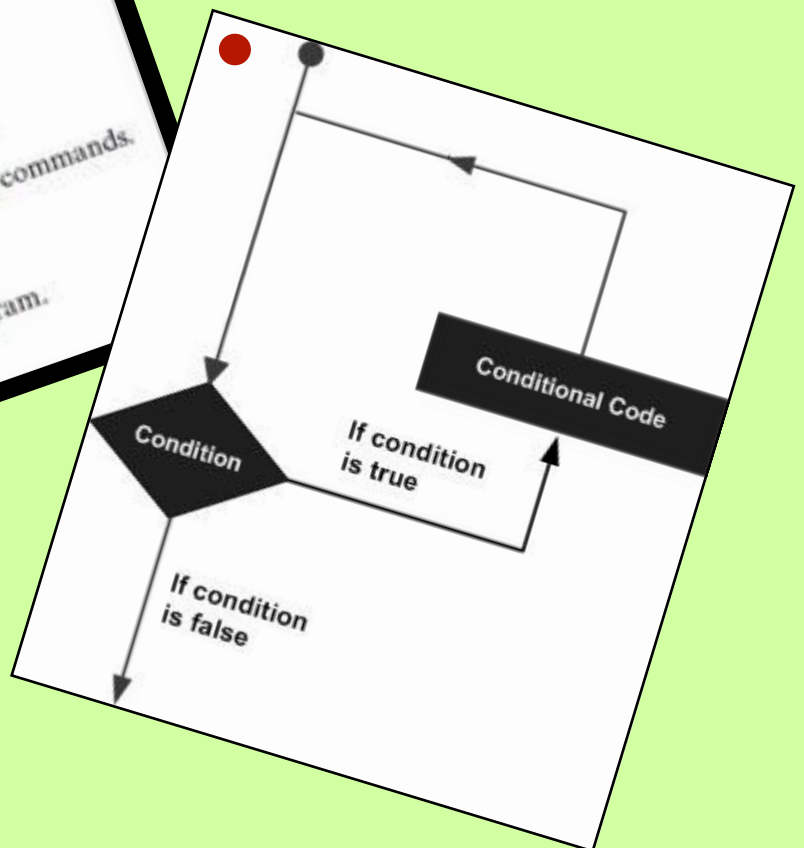
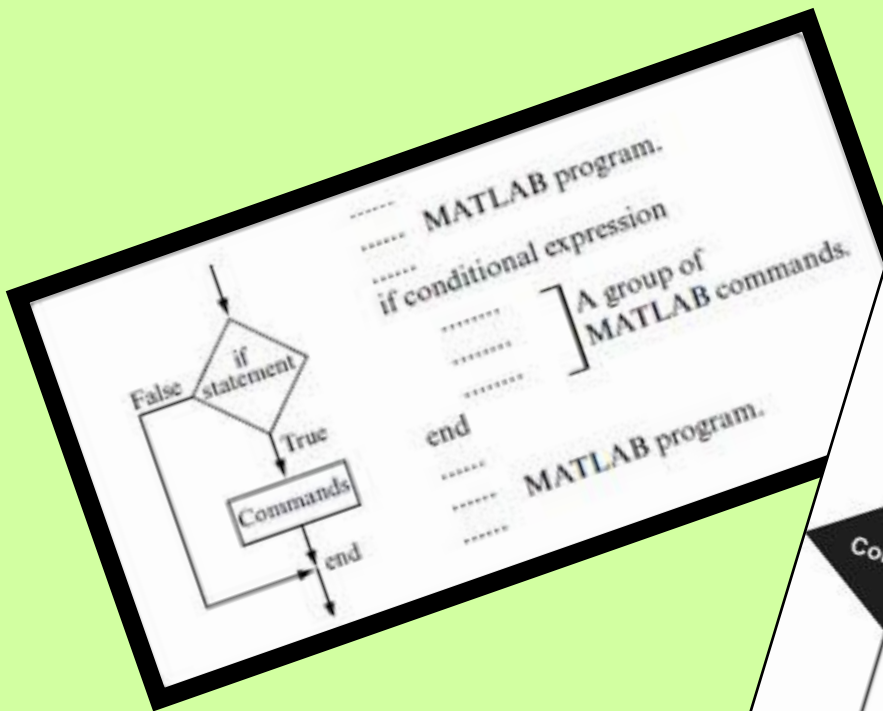
Section 4: For Loops

Section 5: While Loops

Section 6: Try and Catch

Section 7: Useful Commands

### Student Exercises

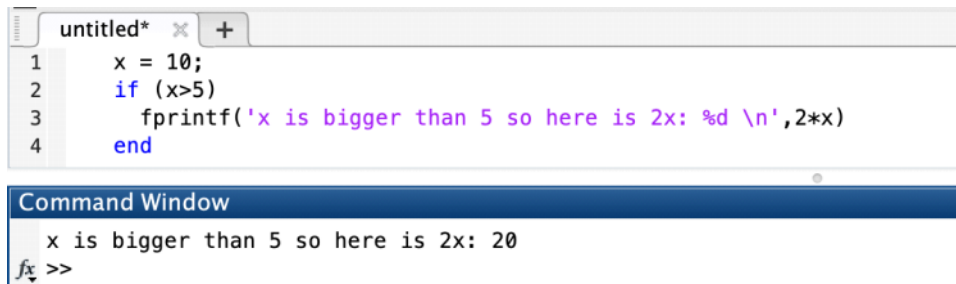


## Section 1: Conditional Statements

Conditional Statements allow the user to execute a block or line of code based on a provided condition. The simplest conditional statement is an if-statement. An if-statement can be used to execute a statement if a certain condition is met. The general simplest syntax is:

```
1 if [condition]
2   [statements]
3 end
```

For example to write a program that if the value of x is greater than 5, the program will output 2x, we do the following: (Set the original value of x to 10)



```
untitled* x +
1   x = 10;
2   if (x>5)
3       fprintf('x is bigger than 5 so here is 2x: %d \n',2*x)
4   end

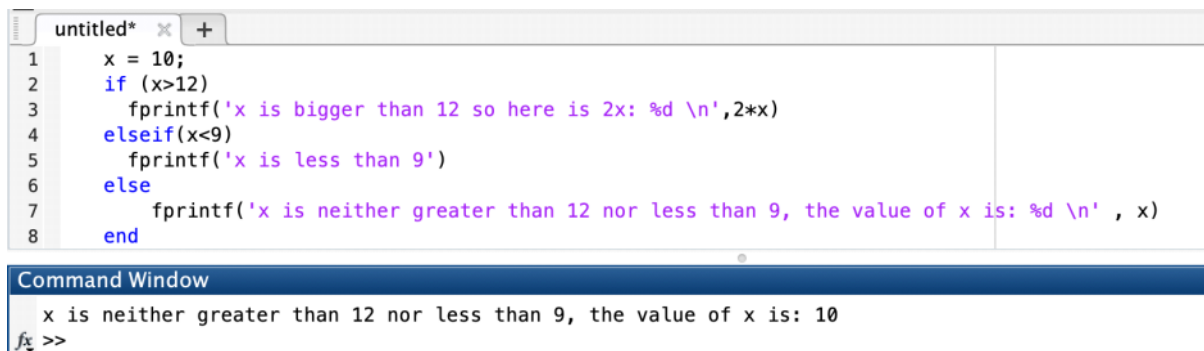
Command Window
x is bigger than 5 so here is 2x: 20
fx >>
```

Figure 4.1

If-statements can include alternate choices, using the optional keywords elseif or else:

elseif	An additional if statement for different conditions
else	if none of the conditions in the if and elseif statements are met, then we use else.

For example:

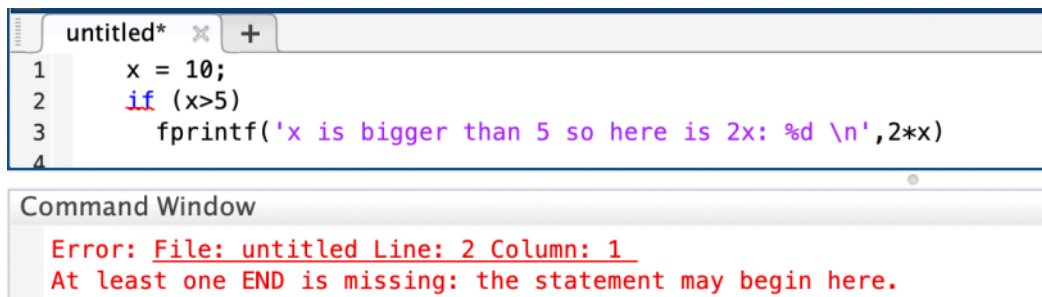


```
untitled* x +
1   x = 10;
2   if (x>12)
3       fprintf('x is bigger than 12 so here is 2x: %d \n',2*x)
4   elseif(x<9)
5       fprintf('x is less than 9')
6   else
7       fprintf('x is neither greater than 12 nor less than 9, the value of x is: %d \n' , x)
8   end

Command Window
x is neither greater than 12 nor less than 9, the value of x is: 10
fx >>
```

Figure 4.2

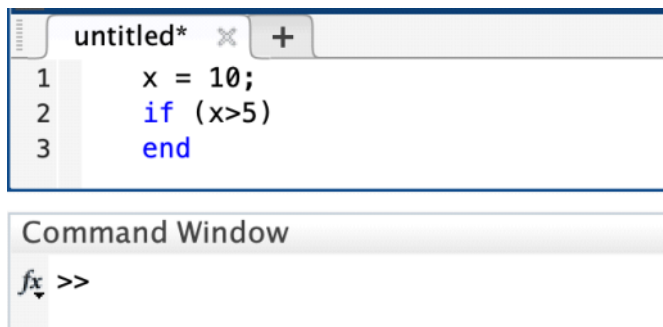
**NOTE** that if-statement will not execute if there is no 'end' at the end of the statement:



```
untitled* x +
1     x = 10;
2     if (x>5)
3         fprintf('x is bigger than 5 so here is 2x: %d \n',2*x)
4
Command Window
Error: File: untitled Line: 2 Column: 1
At least one END is missing: the statement may begin here.
```

Figure 4.3

**NOTE** that MATLAB is situation-aware, which means if you just type an if-statement with no code to execute, MATLAB will not do anything:



```
untitled* x +
1     x = 10;
2     if (x>5)
3     end
Command Window
fx >>
```

Figure 4.4



## Section 2: Basic Boolean Operations

The Logical Boolean Operations represents false and true states using the numbers 0 and 1, respectively. Certain MATLAB functions and operators return logical values to indicate fulfillment of a condition. You can use those logical values to execute conditional code. Below is a list of different basic Boolean operations:

Operation	Description	Syntax
and (short circuiting)	Checks if <b>A AND B</b> are both true	A && B
or (short circuiting)	Checks if either one of <b>A OR B</b> is true	A    B
and	Checks if <b>A AND B</b> are both true	A & B and(A,B)
not	Returns the opposite of <b>A</b>	~A not(A)
or	Checks if either one of <b>A OR B</b> is true	A   B or(A,B)
false	Value of false (Numerically 0)	false
true	Value of true (Numerically 1)	true

Table 4.1

**Short Circuiting** is when second operand is evaluated only when the result is not fully determined by the first operand, in other words:

To check if **A OR B** are both true we do:

**A && B**

Notice that if **A** is true the end result depend on the value of **B**, in this case we scanned for both **A** and **B** so there was no short circuiting. But, if **A** was false, then the operation would return false as one of the two operands is false, in this case we can say there was short circuiting.

And to check if either one of **A OR B** is true we do:

**A || B**

Notice that if **A** is true, the operation will return true as one of the two operands is true and there is no need to scan for the second operand, so, there was short circuiting. On the opposite side, if **A** was false, then the program would continue to scan for the value of **B**, in this case there was no short circuiting.

Write a program that prints out "Access Granted" if both 'username' and 'password' are *true*. In this example preset the values of 'username' and 'password' to *true*:

```
untitled* x +
1 username = true;
2 password = true;
3 if (username && password)
4     fprintf('Access Granted \n')
5 end
```

```
Command Window
Access Granted
fx >> |
```

Figure 4.5

Write a program that will ask the user "What is 2 + 2? " and print out "Correct!", if the answer is "4" or "four" or "Four".

```
untitled* x +
1 answer = input("What is 2 + 2? \n", 's');
2 if(strcmp(answer,"4") || strcmp(answer,"four") || strcmp(answer,"Four"))
3     fprintf('Correct!\n');
4 end
```

```
Command Window
What is 2 + 2?
Four
Correct!
fx >>
```

Figure 4.6

## Section 3: Switch, Case, Otherwise

**S**witch statements are like specialized if-else if statements. They check different values a single variable can hold (called **cases**) and execute the proper commands based on that. If none of the cases are met then the default commands are executed (called **otherwise**). They follow the pattern below:

```
1 switch [variable]
2   case [variable_value]
3     [statements]
4   case [variable_value2]
5     [statements]
6   ...
7   otherwise
8     [statements]
9 end
```

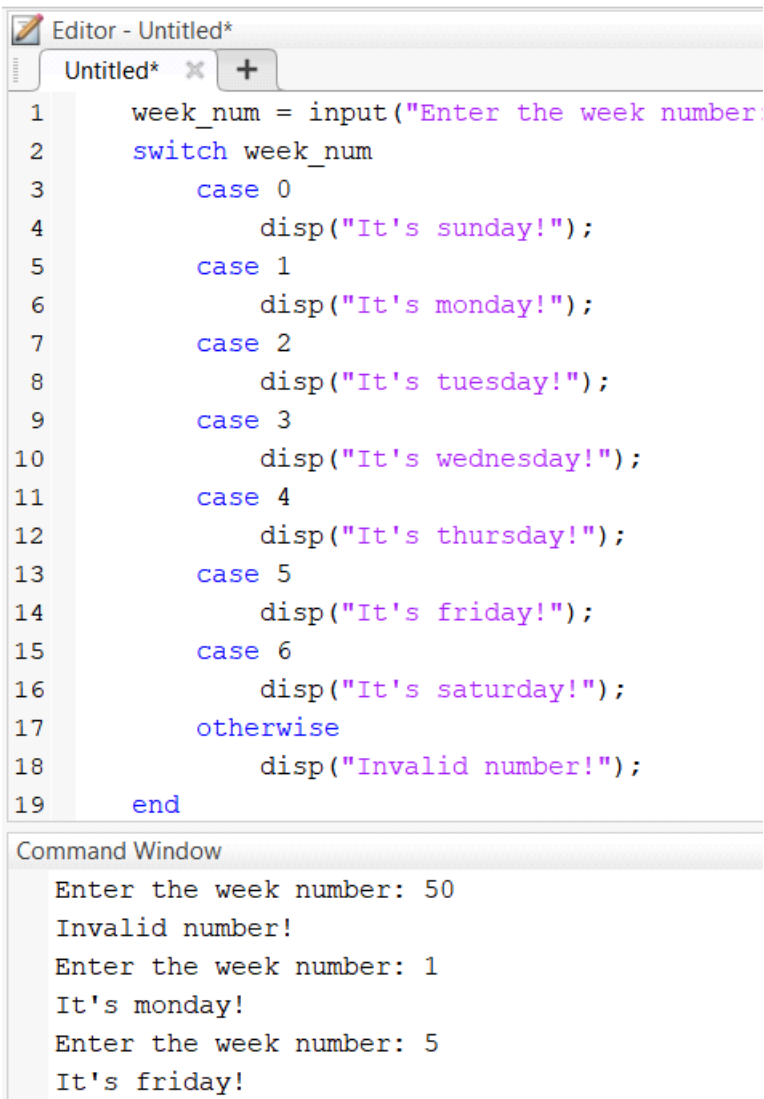
The above code would be the same as the following in if-else if format:

```
1 if variable == variable_value
2   [statements]
3 else if variable == variable_value2
4   [statements]
5 ...
6 else
7   [statements]
8 end
```

**NOTE** that there is no major difference between if-else if and switch statements, thus their use is usually based on the style of the programmer.

**NOTE** that if otherwise is not used and none of the cases are met, then the switch statement is simply skipped over.

Here is an example that tells you the week name based on number:



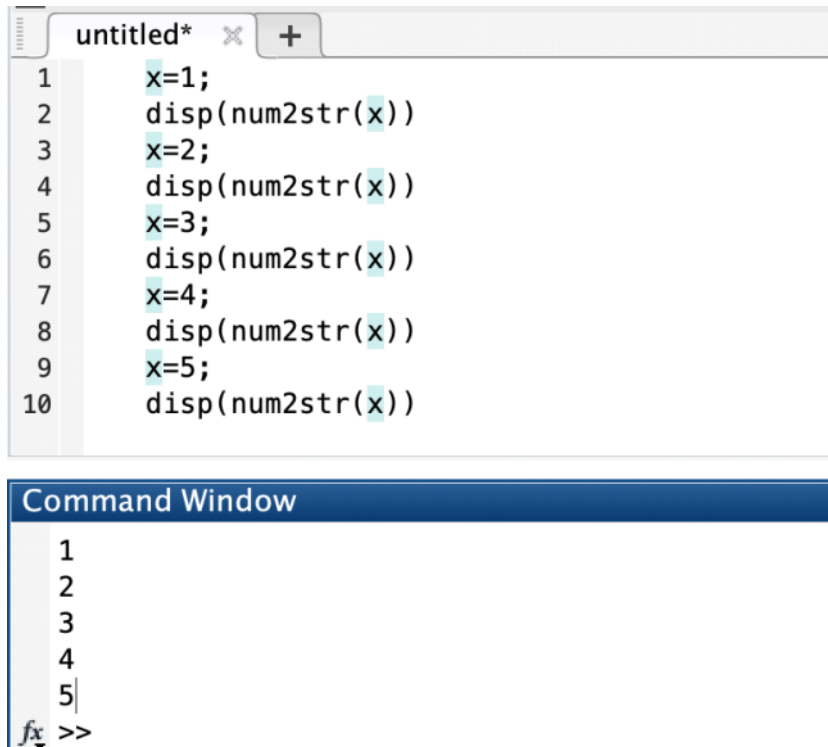
```
Editor - Untitled*
Untitled* x +
1   week_num = input("Enter the week number:");
2   switch week_num
3       case 0
4           disp("It's sunday!");
5       case 1
6           disp("It's monday!");
7       case 2
8           disp("It's tuesday!");
9       case 3
10          disp("It's wednesday!");
11         case 4
12            disp("It's thursday!");
13         case 5
14            disp("It's friday!");
15         case 6
16            disp("It's saturday!");
17         otherwise
18            disp("Invalid number!");
19     end

Command Window
Enter the week number: 50
Invalid number!
Enter the week number: 1
It's monday!
Enter the week number: 5
It's friday!
```

Figure 4.7

## Section 4: For Loops

Suppose we wanted to assign the variable `x` to each of the numbers 1 through 5 and then print each. We could do this:



```
untitled* x +
1     x=1;
2     disp(num2str(x))
3     x=2;
4     disp(num2str(x))
5     x=3;
6     disp(num2str(x))
7     x=4;
8     disp(num2str(x))
9     x=5;
10    disp(num2str(x))

Command Window
1
2
3
4
5|
fx >>
```

Figure 4.8

But as you can see, this can be a very difficult task if we needed to do something more complicated. A for loop allows us to do this in a much cleaner way. The general syntax for a 'for' loop is:

```
1 for [variable] = [values]
2     [statements]
3 end
```

For example, to do the same task asked above with a for loop, we can do the following:

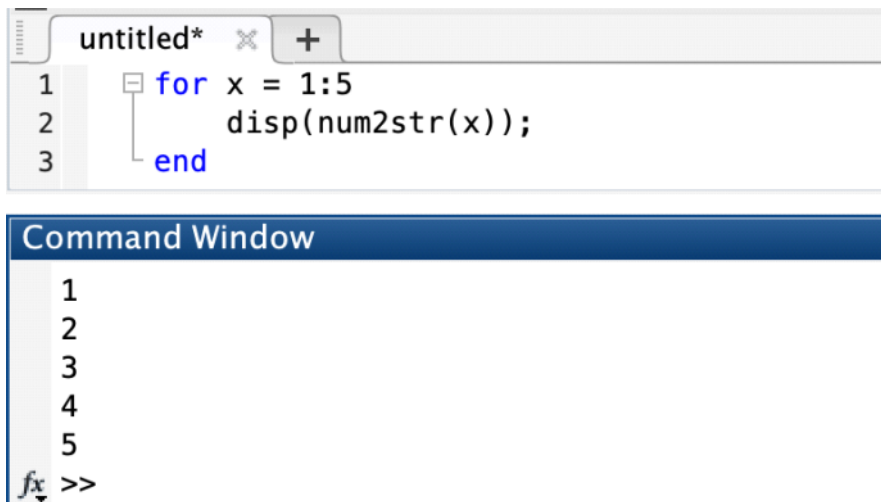


Figure 4.9

**NOTE** that ':' will cause incrementation by 1's from the initial value to the end value.

In order for the for loop to go in different step sizes, the following format should be used:

```
1 x = [start]:[step]:[end]
```

A for loop that counts by 2s from zero to ten can be seen in the below example:

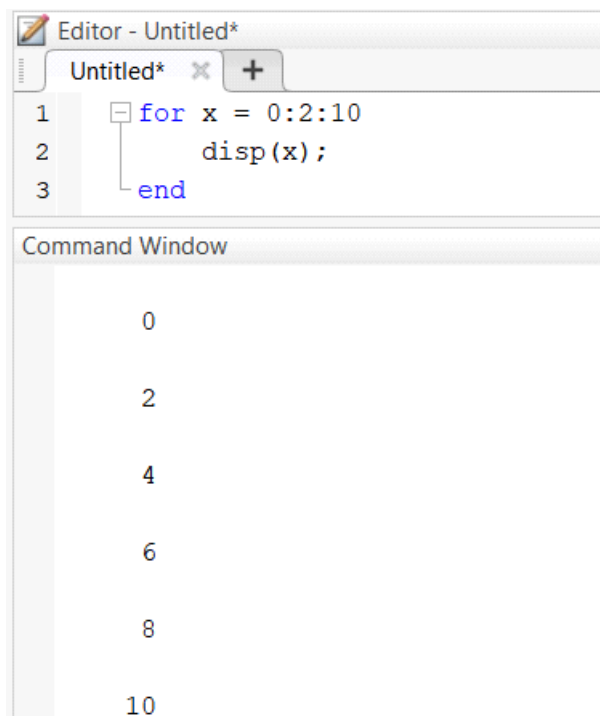


Figure 4.10

**Example 4.4.1:**

Write a program that will ask the user "Enter an integer between 1-100", and will calculate and print out the factorial of the entered integer. Assume the user will input a value between 1-100.

```
untitled* x +
1   x = input('Enter an integer between 1-100: \n');
2   answer = 1;
3   for x = 1:x
4       answer = answer * x;
5   end
6   disp(num2str(answer))

Command Window
Enter an integer between 1-100:
12
479001600
fx >>
```

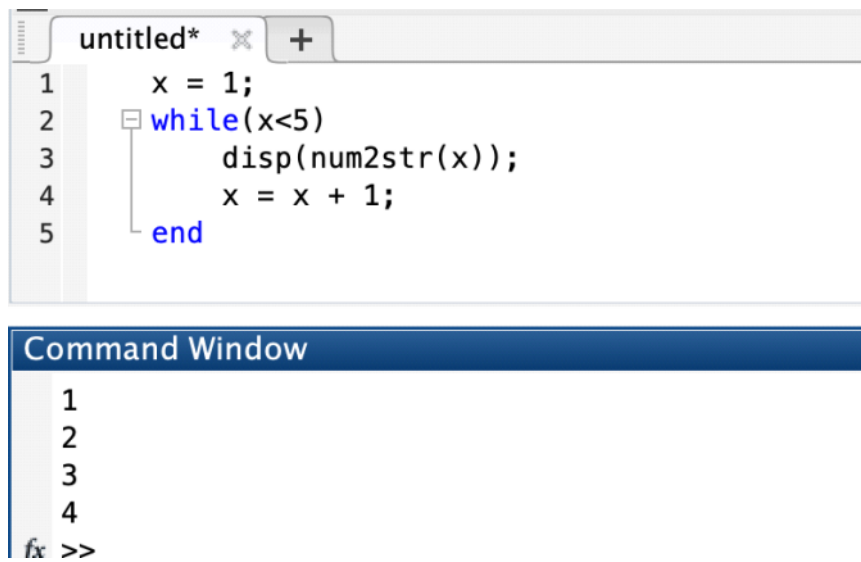
Figure 4.11

## Section 5: While Loops

The trouble with a for loop is that it executes a statement a certain fixed number of times. Often in mathematics we wish to execute a statement an unknown number of times until a criteria is met. In MATLAB a while loop will do exactly that. The syntax for a while loop is:

```
1 While (CONDITIONS)
2     statement
3     .....
4 end
```

What happens is that MATLAB enters the while statement and tests the conditions. If they're true it proceeds through the statements. At the end it goes back to the beginning and tests again. If the statements (any of them) are ever false it ends the loop. Here is a simple example:



The figure shows a MATLAB editor window titled 'untitled\*' with a code editor containing the following code:

```
1 x = 1;
2 while(x<5)
3     disp(num2str(x));
4     x = x + 1;
5 end
```

Below the editor is the Command Window, which displays the output of the code:

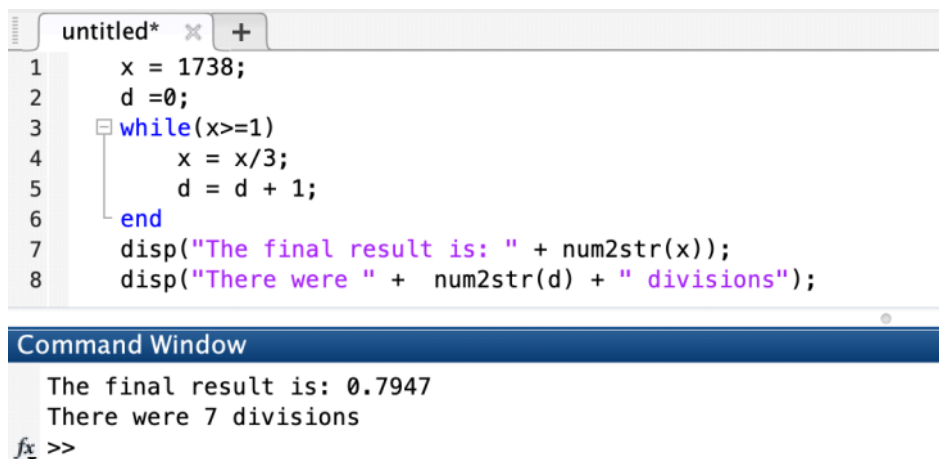
```
1
2
3
4
fx >>
```

Figure 4.12

### Example 4.5.1:

Write a program that will continuously divide an integer by 3, until the result is less than or equal to 1. The initial integer is 1738. The program will display the final result.

**BONUS:** Make the program display how many divisions took place.



The figure shows a MATLAB editor window titled 'untitled\*' with a code editor containing the following code:

```
1 x = 1738;
2 d = 0;
3 while(x>=1)
4     x = x/3;
5     d = d + 1;
6 end
7 disp("The final result is: " + num2str(x));
8 disp("There were " + num2str(d) + " divisions");
```

Below the editor is the Command Window, which displays the output of the code:

```
The final result is: 0.7947
There were 7 divisions
fx >>
```

Figure 4.13

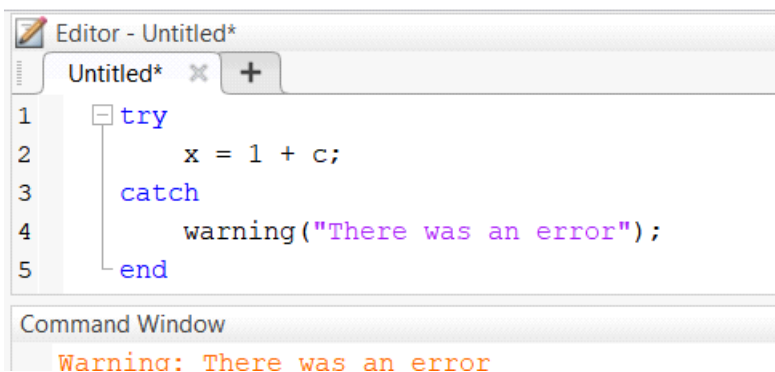


## Section 6: Try and Catch

**T**ry and catch blocks are a method of handling **exceptions** or errors in code. The **try** segment runs the commands in it and any errors produced are caught by the **catch** segment. They follow this format:

```
1 try
2     [statements]
3 catch
4     [statements]
5 end
```

The **warning** function can be utilized in the catch statement to display warning text in an emphasized manner. In the following example, the program attempts to use a variable that is undefined inside a try and catch statement:



The screenshot shows the MATLAB Editor window titled "Editor - Untitled\*" with a single tab "Untitled\*" open. The code in the editor is as follows:

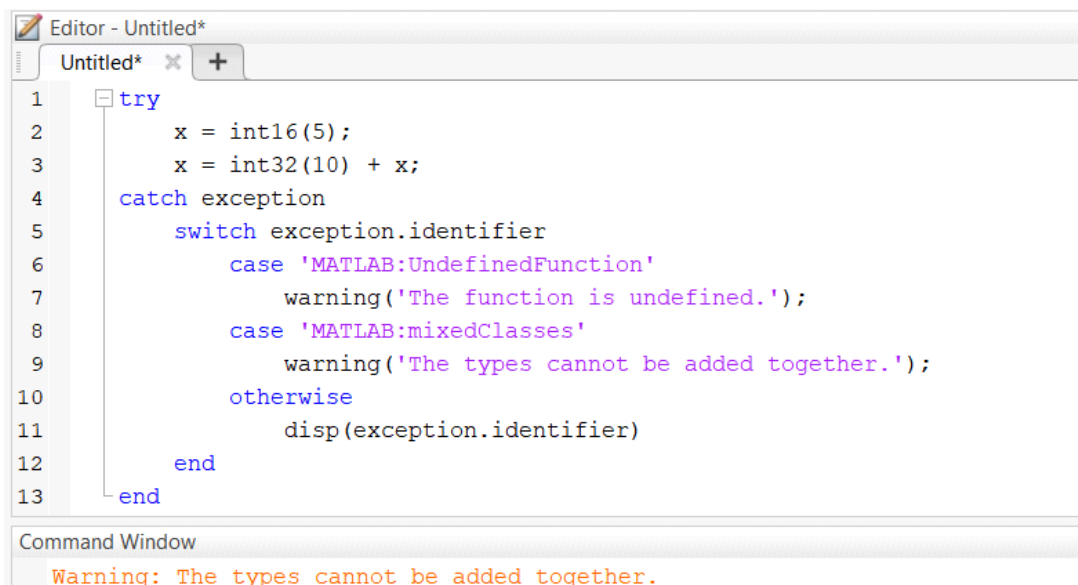
```
1 try
2     x = 1 + c;
3 catch
4     warning("There was an error");
5 end
```

Below the editor is the Command Window, which displays the following warning message:

```
Warning: There was an error
```

Figure 4.14

A more specialized use of try and catch includes specifying the output based on different exceptions. This can be done by giving catch a parameter then checking `parameter_name.identifier` for the name of MATLAB exceptions. The below example gives a specific action based on the MATLAB exception that occurs:



The screenshot shows the MATLAB Editor window titled "Editor - Untitled\*" with a single tab "Untitled\*" open. The code in the editor is as follows:

```
1 try
2     x = int16(5);
3     x = int32(10) + x;
4 catch exception
5     switch exception.identifier
6         case 'MATLAB:UndefinedFunction'
7             warning('The function is undefined.');
```

```
8         case 'MATLAB:mixedClasses'
9             warning('The types cannot be added together.');
```

```
10        otherwise
11            disp(exception.identifier)
12        end
13 end
```

Below the editor is the Command Window, which displays the following warning message:

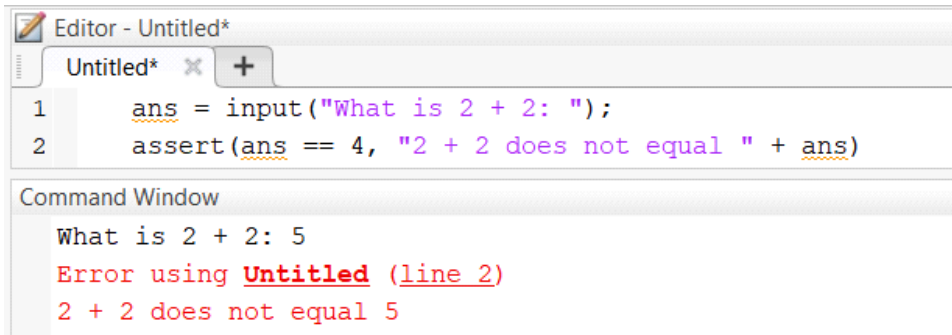
```
Warning: The types cannot be added together.
```

Figure 4.15

Another method for error handling is the **assert** function. Assert accepts a condition and a string to display as an error if the condition fails.

```
1 assert([condition],[message])
```

The following program displays an error because the user fails the assertion test:



```
Editor - Untitled*
Untitled* x +
1 ans = input("What is 2 + 2: ");
2 assert(ans == 4, "2 + 2 does not equal " + ans)

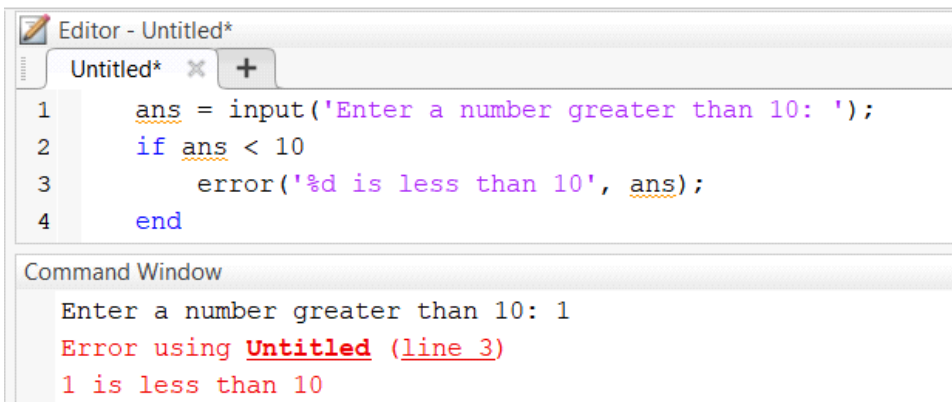
Command Window
What is 2 + 2: 5
Error using Untitled (line 2)
2 + 2 does not equal 5
```

Figure 4.16

Lastly, a custom error message can be created with the **error** command which takes in a message as its parameter.

```
1 error([message])
```

Example 4.17 gives a custom error message when the user gives an invalid input.



```
Editor - Untitled*
Untitled* x +
1 ans = input('Enter a number greater than 10: ');
2 if ans < 10
3     error('%d is less than 10', ans);
4 end

Command Window
Enter a number greater than 10: 1
Error using Untitled (line 3)
1 is less than 10
```

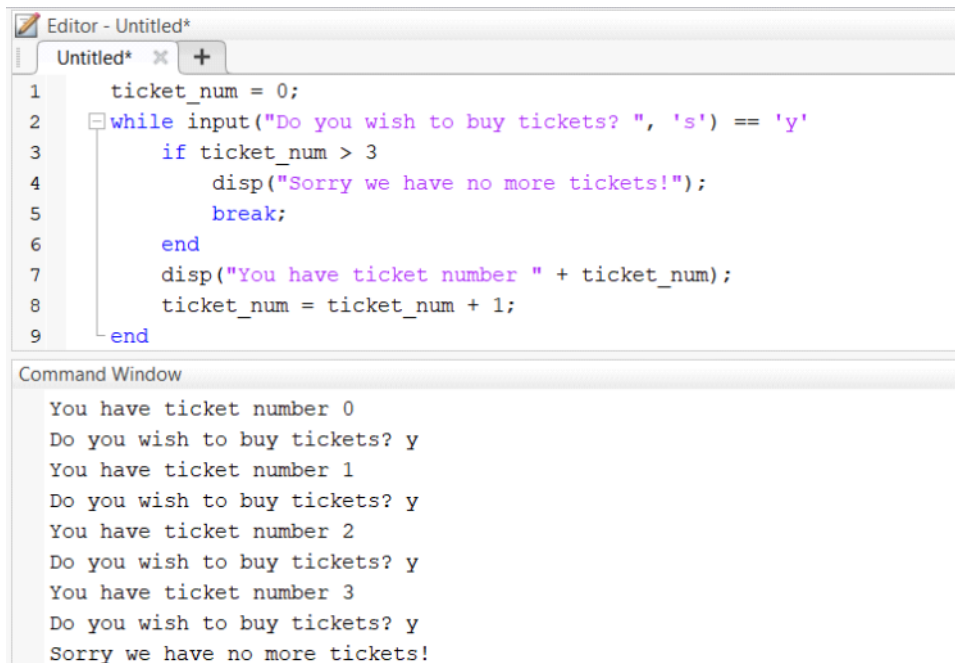
Figure 4.17

**NOTE** that error() follows the fprintf() methodology of displaying variables which was discussed in section 2.5

## Section 7: Useful Commands

Certain commands can be used to control the flow of loops during iterations.

The **break** statement can be used to exit a loop which can be seen in the example below where we set a limit to the number of tickets that can be purchased:

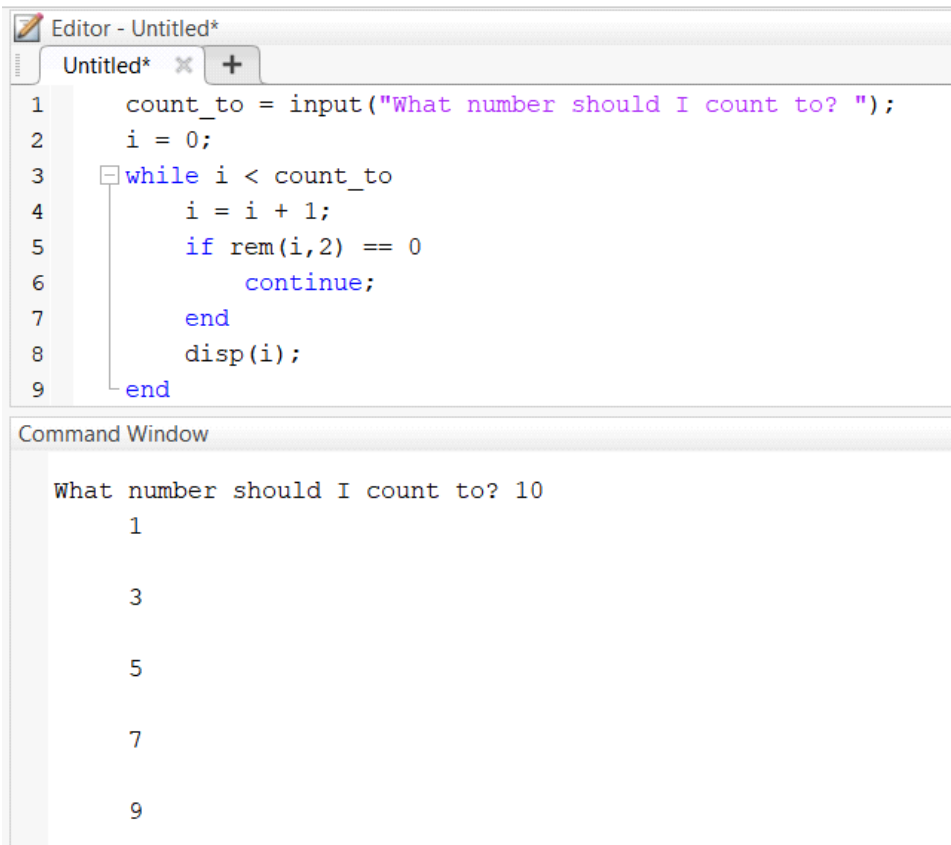


```
Editor - Untitled*
Untitled* x +
1 ticket_num = 0;
2 while input("Do you wish to buy tickets? ", 's') == 'y'
3     if ticket_num > 3
4         disp("Sorry we have no more tickets!");
5         break;
6     end
7     disp("You have ticket number " + ticket_num);
8     ticket_num = ticket_num + 1;
9 end

Command Window
You have ticket number 0
Do you wish to buy tickets? y
You have ticket number 1
Do you wish to buy tickets? y
You have ticket number 2
Do you wish to buy tickets? y
You have ticket number 3
Do you wish to buy tickets? y
Sorry we have no more tickets!
```

Figure 4.18

Another statement is the **continue** statement. This interrupts the loop and forces it to move on to the next iteration. In this example, continue is used to display all odd numbers up to a certain number.



```
Editor - Untitled*
Untitled* x +
1   count_to = input("What number should I count to? ");
2   i = 0;
3   while i < count_to
4       i = i + 1;
5       if rem(i,2) == 0
6           continue;
7       end
8       disp(i);
9   end

Command Window

What number should I count to? 10
1
3
5
7
9
```

Figure 4.19

**Pause** statements interrupt the flow of a program for a designated period of time. Using `pause()` with no parameter results in an interruption until the user presses a key while using `pause(number)` with a number as parameter causes the program to stop for that many seconds.

## Student Exercises

**Be sure to catch and display custom error messages whenever the user submits an invalid input.**

1. Write a program that will ask the user "Enter an integer in inches: ", the program then will convert that integer to centimeters and will ask the user "Calculate and convert that value to centimeters (you may round to the nearest tenth): ". The program then will compare the provided answer with the calculated answer and will print "Correct!" if the answers match, and if not, it will print "Wrong! The correct answer is: [ANSWER]"
2. What is short-circuiting?
3. Write a program that will ask the user "Enter a username: " and will store that under the variable **username**, then the program will ask "Set a password: " and will store that under the variable **password**. The program will then say "Account created, you may login now.". The program will then ask the user "Enter username to login: " and if the input does not equal **username** the program will display "Username not found! Try another user name: " and will ask again till a matching username has been entered" Once a matching username has been entered, the program will ask the user "Enter the password for [USERNAME]: ", if the password does not match the previously set **password**, the program will display "The password does not match, Try another password: ". Once a matching password is entered, the program will print "Successfully Logged In!".
4. Write a complete program that prints the sum of numbers between min (inclusive) and max (inclusive). The program will read the values min and max and display the sum. You can assume min and max are integer values. For example:  
Enter min: 5  
Enter max: 10  
The sum is 45.
5. Write a program that outputs the amount of ways numbers between two points (inclusive) can be used to reach a specific number (repeats are not allowed) by adding them up. For example, if given number 1 as min and 5 as max, and given 8 as the number to reach, the output should be "3" because all the ways to reach 8 with numbers between 3 and 5 are:  
3 + 5  
1 + 3 + 4  
1 + 2 + 5
6. Create a program that will split any given number into each of its digits and display them separately.
7. Write a program that accepts a simple polynomial as input and tells the user what its degree is. For example, an input of ' $3x^2 + 5x + 10$ ' would result in an output of 2.
8. Create a program that takes the derivative of simple 3rd degree polynomials and outputs them.

9. A skydiver with a mass of 80 kg (including parachute) jumps out of an airplane at an altitude of 4000 m, the initial vertical velocity is 0 m/s. He free-falls for 40 seconds before opening a parachute. The parachute has a 10 m diameter when completely opened (you can assume it opens instantaneously). Create a program in MATLAB that calculates and plots the skydiver's vertical speed and altitude as a function of time using the following equations and values. Remember that the skydiver cannot exceed the terminal velocity and the parachute has a failsafe to open at 1000 m.

Hints: There are different terminal velocities with and without the parachute open, assume that the parachute opens instantaneously.

Constant values  $g = 9.81$  [m/s<sup>2</sup>] (Gravitational Acceleration)  $\rho_{\text{air}} = 1.225$  [kg/m<sup>3</sup>] (Density of air)

Before Opening:  $A = 0.5$  [m<sup>2</sup>]  $C_d = 0.7$

After Opening:  $A = \pi/4 * D^2$  [m<sup>2</sup>]  $C_d = 1.4$

Calculation of Terminal Velocity

$$V_t = 0.99 * \sqrt{(2 * \text{mass} * \text{gravity}) / (\rho_{\text{air}} * A * C_d)}$$

Calculation of Downward Acceleration (derived from  $F = m * a$ )

$$a_i = (1/2) * (\rho_{\text{air}} / \text{mass}) * (V_{i-1}^2) * (A) * (C_d) - g$$

Kinematics Equations Setup for a loop

$$V_i = V_{i-1} + a_i * (\Delta t) \quad t \text{ is time}$$

$$Y_i = Y_{i-1} + V_{i-1} * (\Delta t) + (1/2) * a_i * (\Delta t)^2$$

Setup the diameter of the parachute, initial altitude, time of opening the parachute, and the time step as user inputs.

10. Develop a two player rock–paper–scissors game that loops and holds scores until a player exits from the game.



## Section 1: Arrays

**A**rrays are the fundamental representation of information and data in MATLAB®. You can create common arrays and grids, combine existing arrays, manipulate an array's shape and content, and use indexing to access array elements. Below is a table of different commands:

Commands	Description	Syntax
zeros	Create array of all zeros with a size ( <b>n x n</b> ) or ( <b>n x m</b> )	zeros zeros( <b>n</b> ) zeros( <b>n,m</b> )
ones	Create array of all ones with a size ( <b>n x n</b> ) or ( <b>n x m</b> )	ones ones( <b>n</b> ) ones( <b>n,m</b> )
rand	Uniformly distributed random numbers with a size ( <b>n x n</b> ) or ( <b>n x m</b> ). Values are from 1 to <b>max</b>	randi( <b>max, n</b> ) randi( <b>max, n,m</b> )
true	Returns an array of size ( <b>n x n</b> ) or ( <b>n x m</b> ) of logical 1's (true)	true( <b>n</b> ) true( <b>n,m</b> )
false	Returns an array of size ( <b>n x n</b> ) or ( <b>n x m</b> ) of logical 0's (false)	false( <b>n</b> ) false( <b>n,m</b> )
eye	Returns an ( <b>n x n</b> ) or ( <b>n x m</b> ) identity matrix with ones on the main diagonal and zeros elsewhere.	eye( <b>n</b> ) eye( <b>n,m</b> )
diag	Returns a square diagonal matrix with the elements of vector <b>v</b> on the main diagonal.	diag( <b>v</b> )
cat	cat(dim, A, B) concatenates <b>B</b> to the end of <b>A</b> along dimension <b>dim</b> when <b>A</b> and <b>B</b> have compatible sizes	cat( <b>dim, A, B</b> )
horzcat	horzcat(A, B) concatenates <b>B</b> horizontally to the end of <b>A</b> when <b>A</b> and <b>B</b> have compatible sizes	horzcat( <b>A, B</b> )
vertcat	vertcat(A, B) concatenates <b>B</b> vertically to the end of <b>A</b> when <b>A</b> and <b>B</b> have compatible sizes	vertcat( <b>A, B</b> )
repelem	Repeats each element of vector <b>V</b> , <b>n</b> times into a new vector.	repelem( <b>v,n</b> )
repmat	Returns an ( <b>n x n</b> ) or ( <b>n x m</b> ) array containing of <b>A</b> in the row( <b>n</b> ) and column( <b>m</b> ) dimensions	repmat( <b>v ,n</b> ) repmat( <b>v ,n, m</b> )
magic	Creates an array of size ( <b>n x n</b> ) with values 1- <b>n</b> <sup>2</sup>	magic( <b>n</b> )

Table 5.1

There are different ways to creating arrays, look at the examples below:



Command	Output
V = [1 2 3]	1 2 3
V = [1 2 3; 4 5 6]	1 2 3 4 5 6
V = [1:5]	1 2 3 4 5
V = [1:3;4:6]	1 2 3 4 5 6
zeros(2)	0 0 0 0
Ones(2)	1 1 1 1
randi(10,1,2)	3 7
true(2)	1 1 1 1
eye(3)	1 0 0 0 1 0 0 0 1
magic(3)	8 1 6 5 9 2 3 7 4

Table 5.2

**Concatenating** arrays allows you to add two arrays to each other, for example:

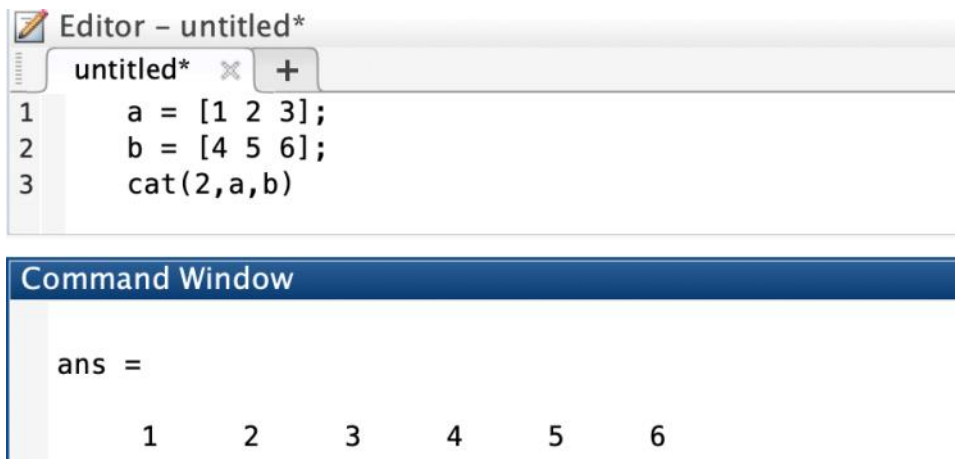


Figure 5.1

**NOTE** that dim '2' concatenates horizontally and '1' is vertically.

**Indexing** has an important role in MATLAB and creating arrays, for example if you wanted to create an array from 1-20, you could do the following:

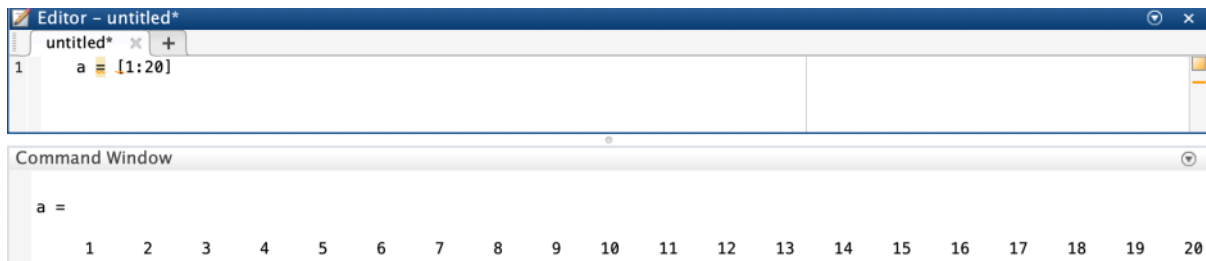


Figure 5.2

And to create an array from 1-50 with an index of 3, we would do the following:

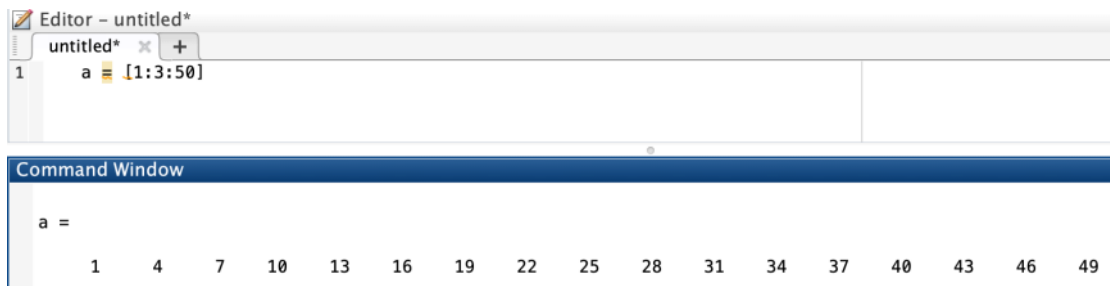


Figure 5.3

**Adding** to an array is very common, and in MATLAB it is very simple to do:

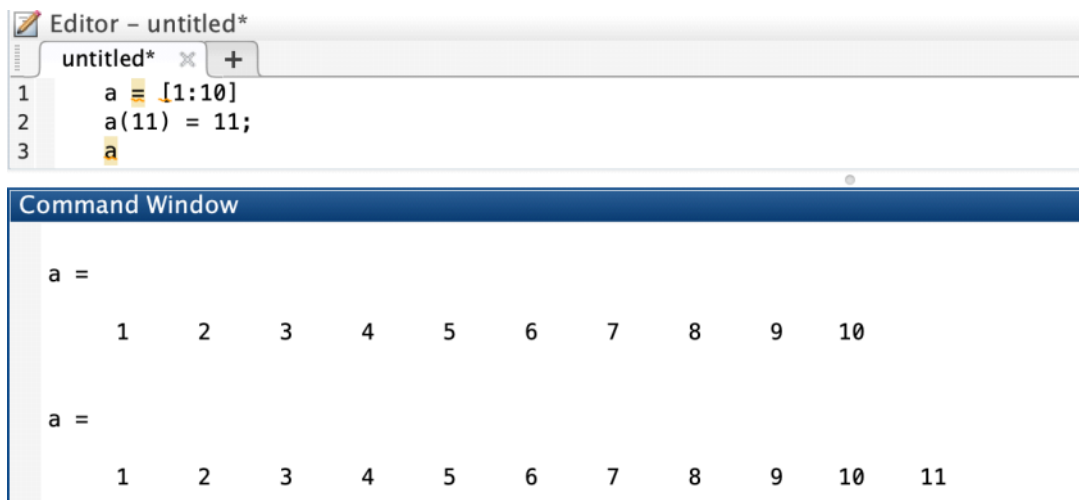


Figure 5.4

**Note** that 'a(11)' is addressing the 11th element in vector `a`, and adding the value '11' to it.

**Replacing** an element in an array is very similar to adding. First, you address the element, and then you replace the value of that element. For example, to replace the value of 5 with a 0, we do the following:

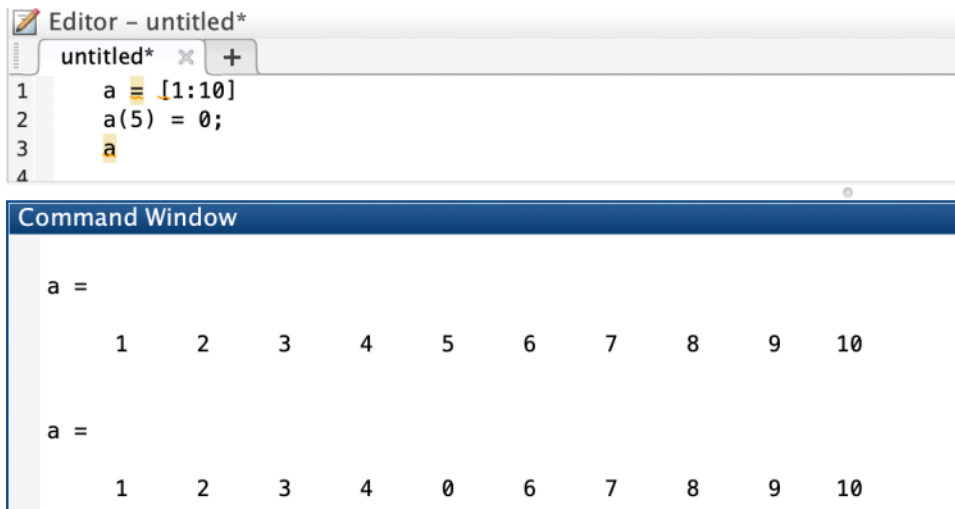


Figure 5.5

**Removing** from an array: The easiest way to remove a row or column of a matrix is setting that row or column equal to a pair of empty square brackets []. For example to remove the 5th element from the above example, we can do this:

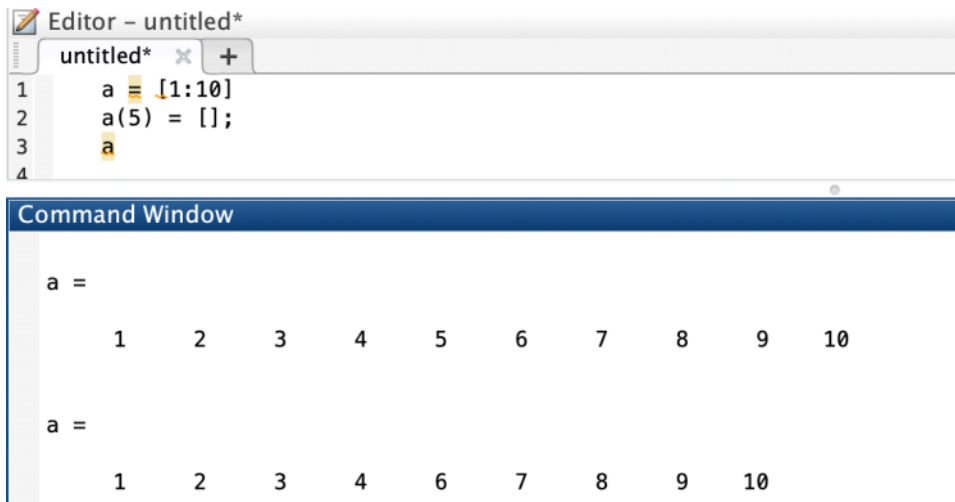


Figure 5.6

As another example, to create a 4x4 array and remove the 4th row and 4th column from it, we do as follows:

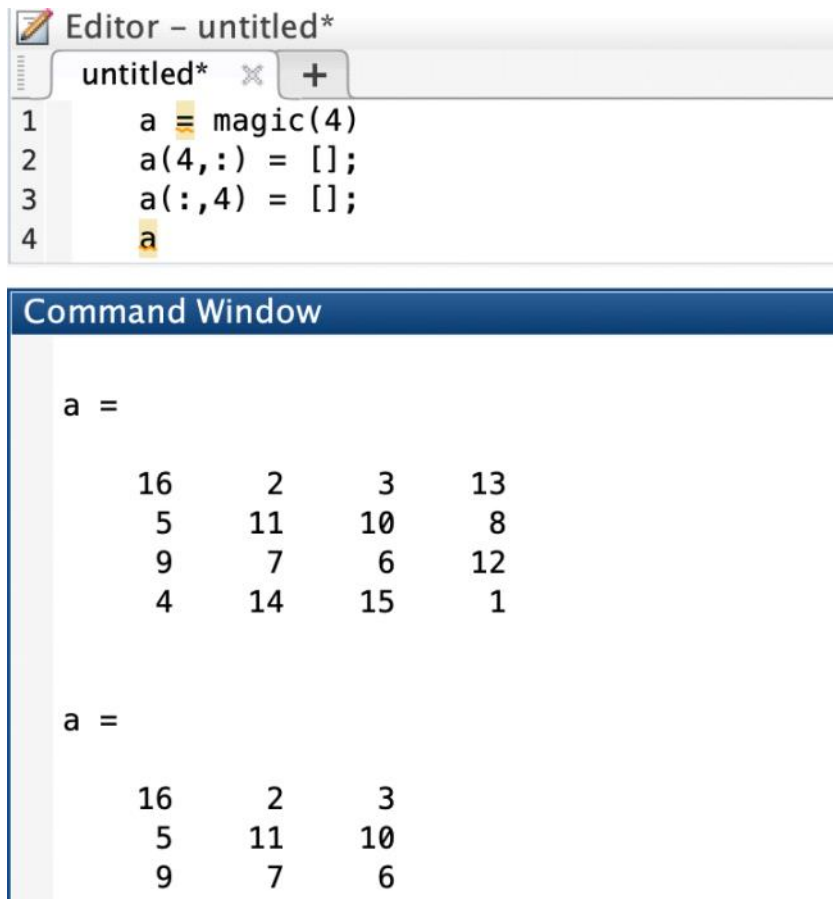


Figure 5.7

In this example, we first removed all elements in the 4th row and then removed all elements in the 4th column.

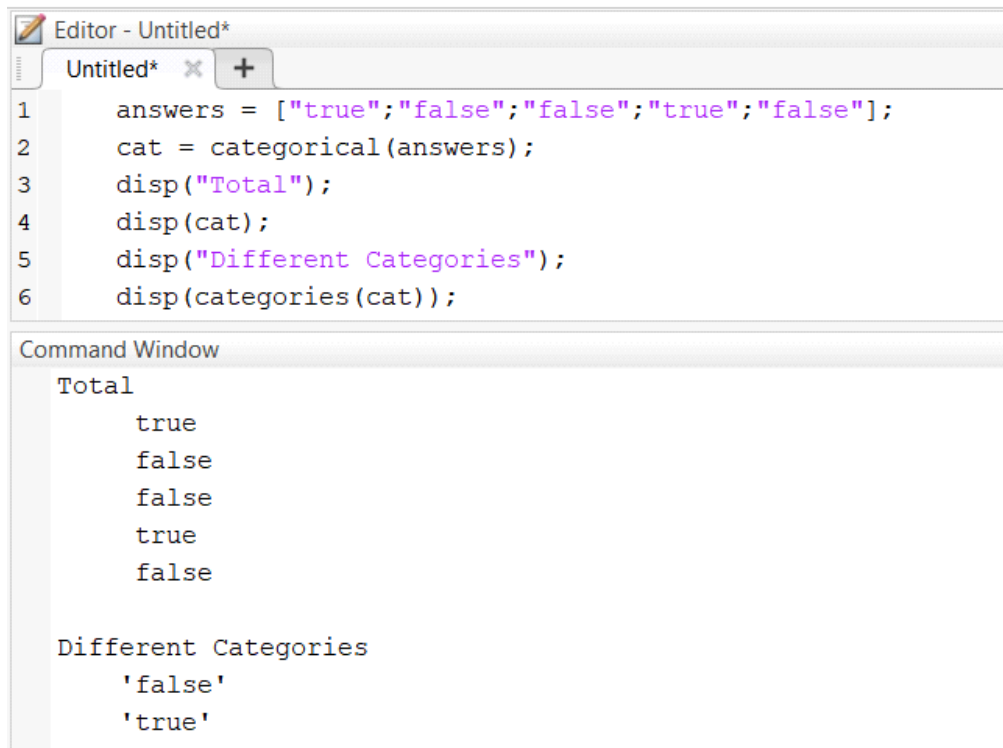
**Note** that ':' refers to all elements in that row or column.

## Section 2: Categorical Arrays

Categorical arrays are a data type that can store a set of categories and are useful when holding nonnumerical data. The simplest way to use them is through the following format:

```
1 cat = categorical([array]);
```

Simply displaying the value of **cat** would display the array it holds, however, utilizing the **categories()** function on it gives us a list of all the categories. Here is an example that records answers to a true/false test:



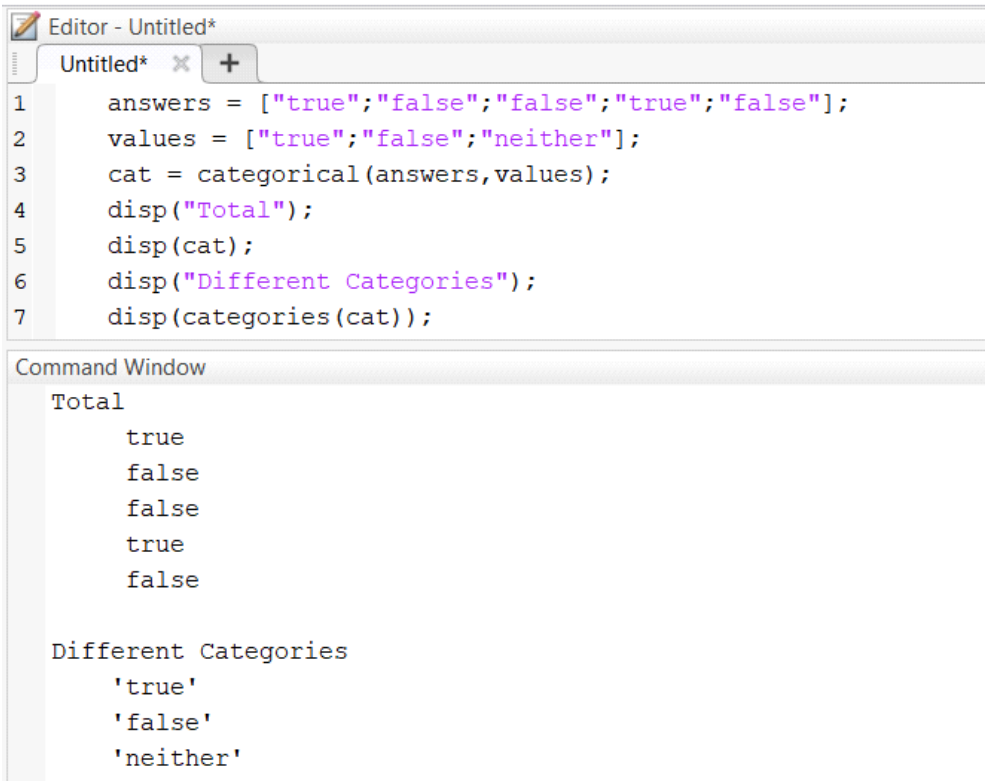
```
Editor - Untitled*
Untitled* x +
1  answers = ["true";"false";"false";"true";"false"];
2  cat = categorical(answers);
3  disp("Total");
4  disp(cat);
5  disp("Different Categories");
6  disp(categories(cat));

Command Window
Total
    true
    false
    false
    true
    false

Different Categories
    'false'
    'true'
```

Figure 5.8

A second parameter can be added which tracks all possible values that can occur in the dataset even if they have not appeared yet. In the example below, "neither" is not an answer however using the **categories()** function still gives us it as an option because we told it, "neither" was a possible category:



```
Editor - Untitled*
Untitled* x +
1  answers = ["true";"false";"false";"true";"false"];
2  values = ["true";"false";"neither"];
3  cat = categorical(answers,values);
4  disp("Total");
5  disp(cat);
6  disp("Different Categories");
7  disp(categories(cat));

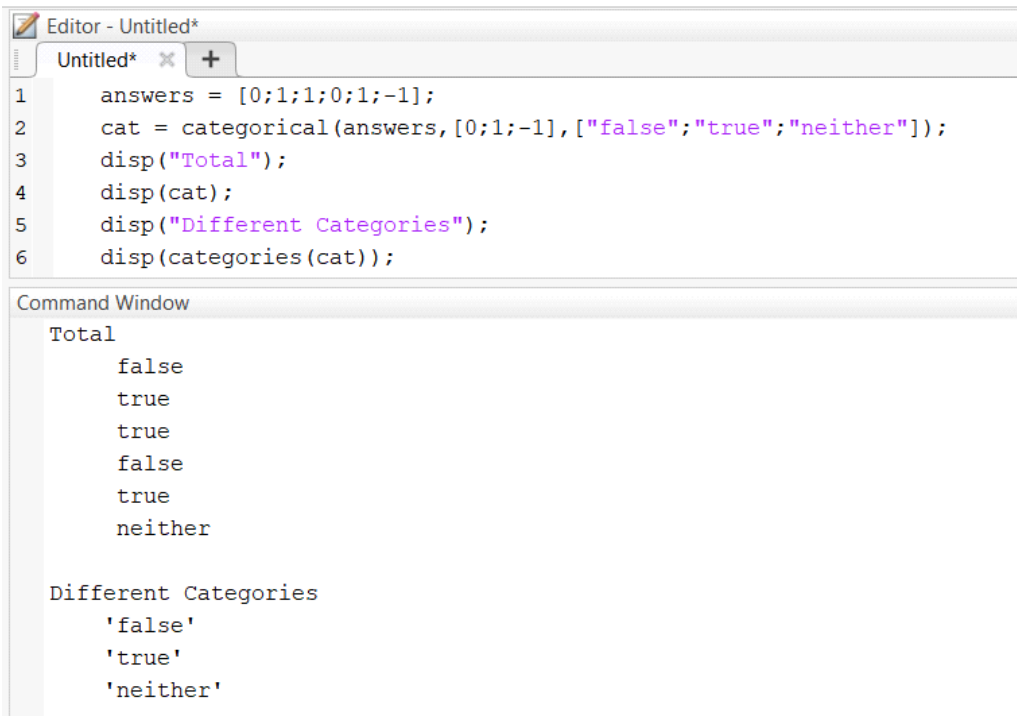
Command Window
Total
    true
    false
    false
    true
    false

Different Categories
    'true'
    'false'
    'neither'
```

Figure 5.9

You can give MATLAB a key to convert from integer data to categories by specifying a second and third parameter that match. In the below figure, 0 is set to false, 1 is set to true, and -1 is set to neither. MATLAB converts the integers in the variable **answers** into the categories specified in the third parameter.

**NOTE** that the second and third parameter must follow the same format and have the same number of elements, otherwise MATLAB will give an error.



```
Editor - Untitled*
Untitled* x +
1  answers = [0;1;1;0;1;-1];
2  cat = categorical(answers,[0;1;-1],["false";"true";"neither"]);
3  disp("Total");
4  disp(cat);
5  disp("Different Categories");
6  disp(categories(cat));

Command Window
Total
    false
    true
    true
    false
    true
    neither

Different Categories
    'false'
    'true'
    'neither'
```

Figure 5.10

Categories can be given ordinal values by adding the fourth and fifth parameter 'Ordinal' and true. In the below example, old is given 0, older is given 1, and oldest is given 2, thus giving the categories the following order old < older < oldest:

```

Editor - Untitled*
Untitled* x +
1 age = [0;1;2;0;1;2];
2 cat = categorical(age, [0;1;2], ["old";"older";"oldest"], 'Ordinal', true);
3 disp("Total");
4 disp(cat);
5 disp("Different Categories");
6 disp(categories(cat));

Command Window
Total
    old
    older
    oldest
    old
    older
    oldest

Different Categories
    'old'
    'older'
    'oldest'

```

Figure 5.11

The below table displays a set of useful commands and functions:

Function	Description	Syntax
addcats	Adds more categories to the array.	addcats(cat1, cat2) addcats(cat1, cat2, 'Before', cat_name) addcats(cat1, cat2, 'After', cat_name)
removecats	Removes categories from the array.	removecats(cat) removecats(cat, cat_old)
mergecats	Merges categories from the array.	mergecats(cat, cat_old) mergecats(cat, cats_old, cat_merge_to)
renamecats	Renames categories from the array.	renamecats(cat, names_new) renamecats(cat, names_old, names_new)
setcats	Sets new categories into the array.	setcats(cat, cat_new)

Table 5.3

**Addcats** adds the specified categories to the category array, it does **NOTE** add to the array's values. The before and after parameters can be used to specify the ordinal value of the new categories (this only works in categorical arrays that have ordinal set to true).

The screenshot shows a MATLAB Editor window with the following code:

```

1 a = categorical({'old','older','oldest'},'Ordinal',true);
2 b = addcats(a,{'youngest','younger','young'},'before','old');
3 disp(b); %values are the same.
4 disp(categories(b));%categories are different, notice the order too.

```

The Command Window displays the output of the code:

```

    old      older      oldest

    'youngest'
    'younger'
    'young'
    'old'
    'older'
    'oldest'

```

Figure 5.12

Without the second parameter **removecats** removes all unused categories. The second parameter specifies what category should be removed. The second version is shown below:

The screenshot shows a MATLAB Editor window with the following code:

```

1 a = categorical({'red','red','blue','purple','blue'});
2 b = removecats(a,{'red'});
3 disp(b); %Red values are replaced with undefined.
4 disp(categories(b));%Red category is gone.

```

The Command Window displays the output of the code:

```

    <undefined>    <undefined>    blue    purple    blue

    'blue'
    'purple'

```

Figure 5.13

**Mergecats** allows for a category to be converted to another category when it only uses two parameters. The second version allows for any number of categories to be converted into a new category. Both of these versions are shown in the figure below:



```

Editor - Untitled*
Untitled* x +
1 a = categorical({'red','yellow','blue','purple','blue'});
2 b = mergecats(a,{'red','blue'}); %converts blue to red
3 disp(b);
4 c = mergecats(b,{'purple','yellow'},'not red'); %converts purple and yellow to not red
5 disp(c);

```

Command Window

```

red    yellow    red    purple    red
red    not red    red    not red    red

```

Figure 5.14

With only two parameters, **renamecats** forces the user to rename all categories. The third parameter allows the user to choose a specific category to rename.

```

Editor - Untitled*
Untitled* x +
1 a = categorical({'red','yellow','blue','purple','blue'});
2 b = renamecats(a,{'red','blue'},{'green','violet'});
3 disp(b);

```

Command Window

```

green    yellow    violet    purple    violet

```

Figure 5.15

**Setcats** creates a new categorical array that contains all categories in the second parameter, but it only keeps values that existed in the original array and are a category in the second parameter. All other values become undefined.

```

Editor - Untitled*
Untitled* x +
1 a = categorical({'apples','dogs','oranges','cats'});
2 b = setcats(a,{'beavers','dogs'});
3 disp(b);
4 disp(categories(b));

```

Command Window

```

<undefined>    dogs    <undefined>    <undefined>

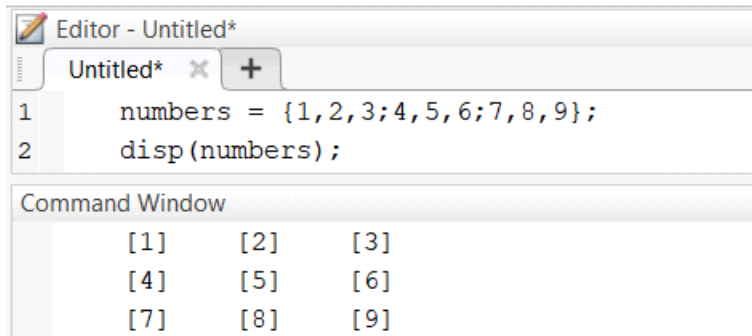
'beavers'
'dogs'

```

Figure 5.16

## Section 3: Cell Arrays

Cell arrays are a datatype where each element is called a **cell** and each cell can contain its own datatype. They are defined with curly brackets {}, commas indicate a change in column while semicolons indicate a change in row. An example can be seen below:



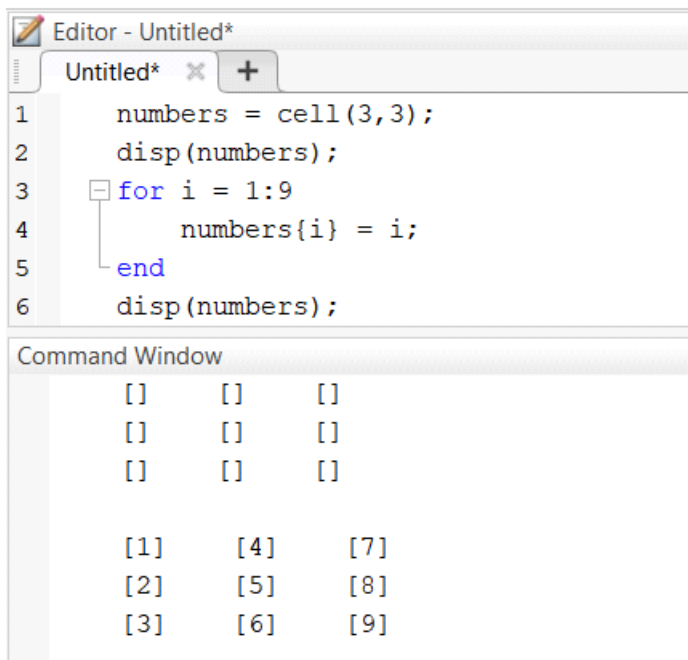
```
Editor - Untitled*
Untitled* x +
1 numbers = {1,2,3;4,5,6;7,8,9};
2 disp(numbers);

Command Window
[1] [2] [3]
[4] [5] [6]
[7] [8] [9]
```

Figure 5.17

**Note** that you can have different datatypes in different cells of the same cell array.

The **cell** function can be used to create an empty cell array of n-dimensions which can then be filled up with a for loop. The below figure shows an example of this with a 3x3 cell array.



```
Editor - Untitled*
Untitled* x +
1 numbers = cell(3,3);
2 disp(numbers);
3 for i = 1:9
4     numbers{i} = i;
5 end
6 disp(numbers);

Command Window
[] [] []
[] [] []
[] [] []

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

Figure 5.18

Accessing cell arrays is divided into two different methods. Use parenthesis () to get a set of cells from the array, and use curly braces {} to obtain the content of cells. Differences between indexing with parenthesis and curly braces can be seen in the example below:

```

Editor - Untitled*
Untitled* x +
1   A = {'red', 'white', 'blue'; 1, 2, 3};
2   disp(A);
3   %Both {} and () follow the [row_number, column_number] format when indexed.
4   disp(A(1,2));
5   disp(A{1,2});
6
7   %() follows indexing similar to arrays but {} cannot do this.
8   disp(A(2,:));

```

Command Window

```

'red'   'white'  'blue'
[ 1]    [ 2]    [ 3]

'red'   'white'  'blue'
[ 1]    [ 2]    [ 3]

'white'

white
[1]    [2]    [3]

```

Figure 5.19

Additionally, it's important to note that attempting to set values to larger indices that do not exist results in MATLAB adjusting the cell array size as shown in the figure below:

```

Editor - Untitled*
Untitled* x +
1   A = {'red', 'white', 'blue'; 1, 2, 3};
2   disp(A);
3   A{4,4} = 'a';
4   disp(A)

```

Command Window

```

'red'   'white'  'blue'
[ 1]    [ 2]    [ 3]

'red'   'white'  'blue'  []
[ 1]    [ 2]    [ 3]    []
[]      []      []      []
[]      []      []      'a'

```

Figure 5.20

## Section 4: Tables

Tables hold categories of data by storing columns of arrays, each of which can store a different datatype. All columns must have the same number of rows. In similar fashion to cell arrays, parenthesis can be used to access subtables of data while curly braces can be used to access specific values in a table. In its most simple form, the function **table** can be used with different variables. The figure below shows a table holding measurements for three different fruits and demonstrates how each variable in the table can be accessed through **dot indexing**.

**Note** that each table column can consist of a different number of columns.

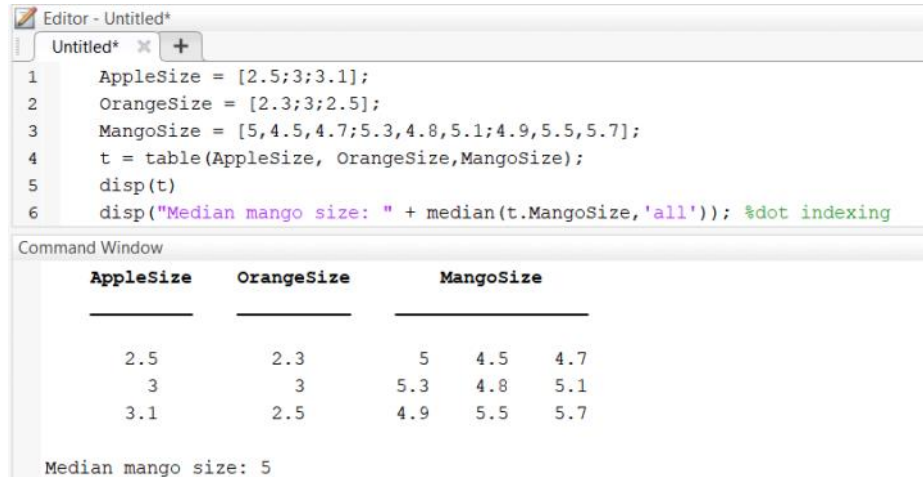


Figure 5.21

An empty table can be made by specifying the size, variable type, and variable name of the table following this format:

```
1 table('Size', [Array_of_Sizes], 'VariableTypes', [Array_of_Variable_Types],
    'VariableNames', [Array_of_Variable_Names], 'RowNames', [Cell_Array_of_Row_Names])
```

Here is an example of this in use:



Figure 5.22

Setting values is similar to cell arrays, in that the size of the table will be expanded if an index that is too large is used. The example below shows this because while the table is at first 3x3, it grows to a 4x4 when the user attempts to add a fourth row and column.

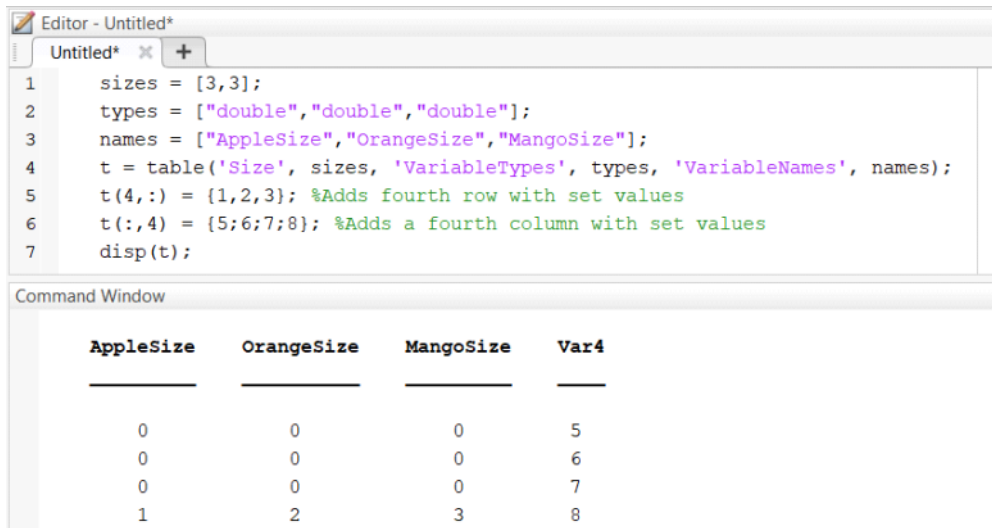


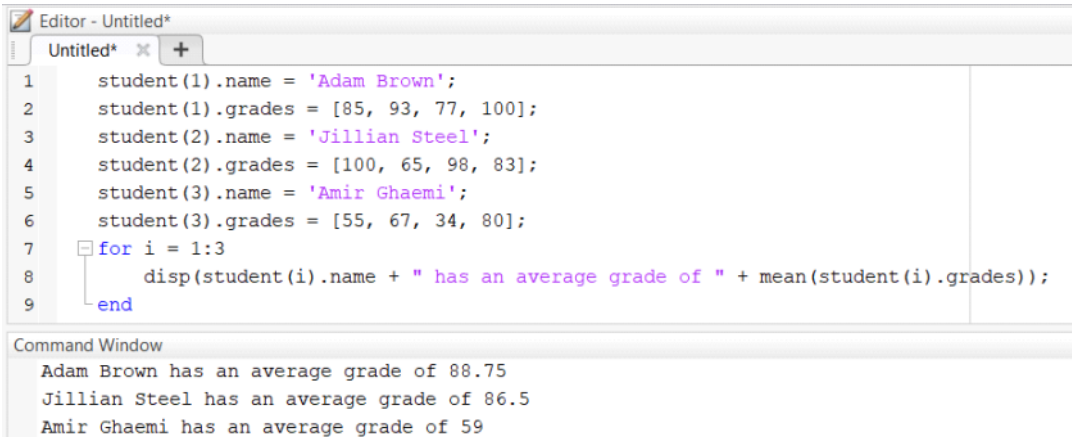
Figure 5.23

## Section 5: Structures

Structures are a type of array that can hold groups of information called **fields**. Each field can hold a different data type and can be accessed through dot indexing which was mentioned briefly in section 5.4. Setting a structure follows this format:

```
1 struct_name(index).field_name = data;
```

Here it is in use:



```
Editor - Untitled*
Untitled* x +
1 student(1).name = 'Adam Brown';
2 student(1).grades = [85, 93, 77, 100];
3 student(2).name = 'Jillian Steel';
4 student(2).grades = [100, 65, 98, 83];
5 student(3).name = 'Amir Ghaemi';
6 student(3).grades = [55, 67, 34, 80];
7 for i = 1:3
8     disp(student(i).name + " has an average grade of " + mean(student(i).grades));
9 end

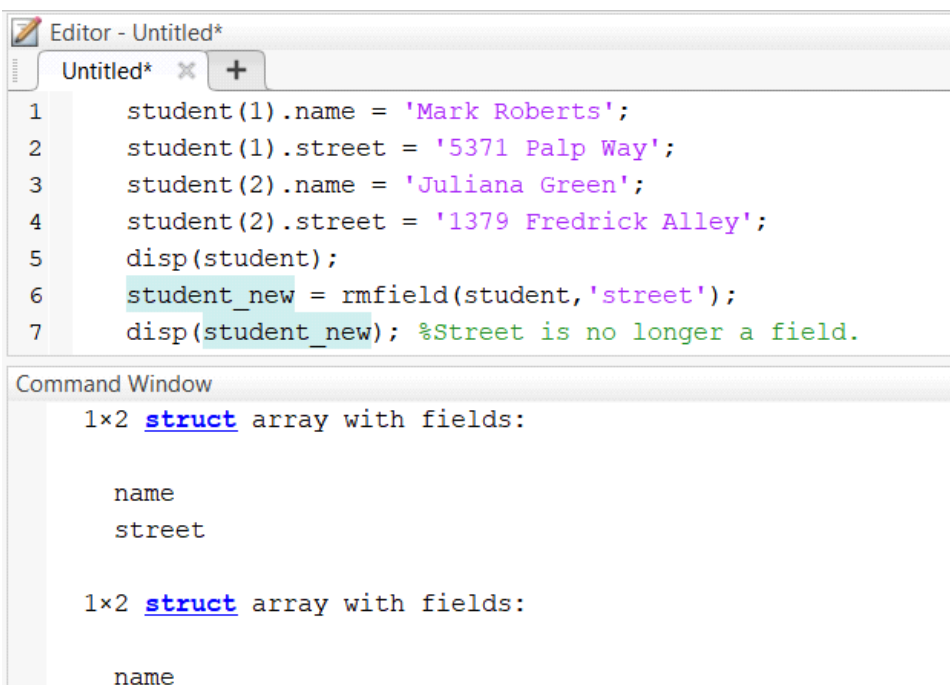
Command Window
Adam Brown has an average grade of 88.75
Jillian Steel has an average grade of 86.5
Amir Ghaemi has an average grade of 59
```

Figure 5.24

The function **rmfield** removes a field from a structure, its format is as follows:

```
1 rmfield(structure_name, field_name)
```

Here is an example of it:



```
Editor - Untitled*
Untitled* x +
1 student(1).name = 'Mark Roberts';
2 student(1).street = '5371 Palp Way';
3 student(2).name = 'Juliana Green';
4 student(2).street = '1379 Fredrick Alley';
5 disp(student);
6 student_new = rmfield(student, 'street');
7 disp(student_new); %Street is no longer a field.

Command Window
1x2 struct array with fields:
    name
    street

1x2 struct array with fields:
    name
```

Figure 5.25

## Section 6: Conversions

**M**ATLAB allows us to swiftly convert from different array types; we can convert between numeric arrays, character arrays, cell arrays, structures, or tables.

Function	Description	Syntax
int2str	Converts integers to characters	int2str(n)
mat2str	Converts matrix to characters	mat2str(n) mat2str(n, precision)
num2str	Converts numbers to characters	num2str(n) num2str(n, precision)
str2double	Converts strings to double precision values	str2double(str)
str2num	Converts character array or a string to numerical array	str2num(chr)
table2array	Converts table to homogenous array	table2array(tbl)
table2cell	Converts table to cell array	table2cell(tbl)
table2struct	Converts table to structure array	table2struct(tbl)
array2table	Converts homogenous array to table	array2table(A)
cell2table	Converts cell array to table	cell2table(C)
struct2table	Converts structure array to table	struct2table(S)
cell2mat	Converts cell array to matrix array	cell2mat(C)
cell2struct	Converts cell array to structure array	cell2struct(C)
mat2cell	Converts matrix array to cell array	mat2cell(A)
num2cell	Converts numeric array to cell array	num2cell(A) num2cell(A, dimension)
struct2cell	Converts structure array to cell array	struct2cell(S)

Table 5.3

Example 5.6.1: Converting integer array to string

```
Editor - untitled*
untitled* x +
1   n = [1:10]
2   str = int2str(n)

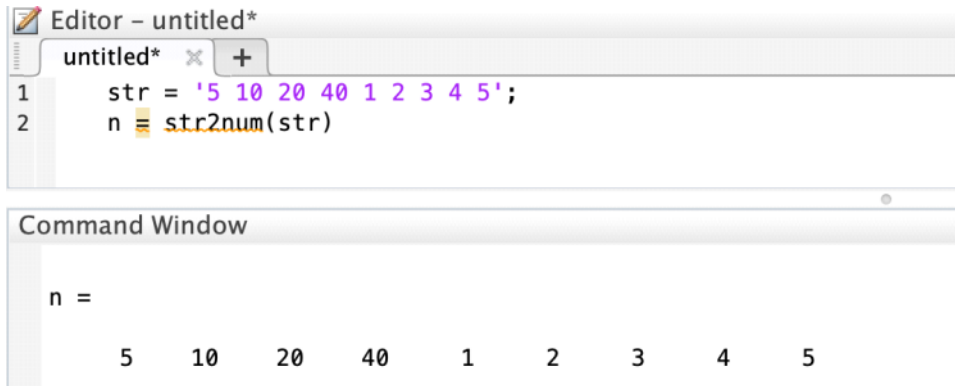
Command Window

n =
     1     2     3     4     5     6     7     8     9    10

str =
'1 2 3 4 5 6 7 8 9 10'
```

Figure 5.26

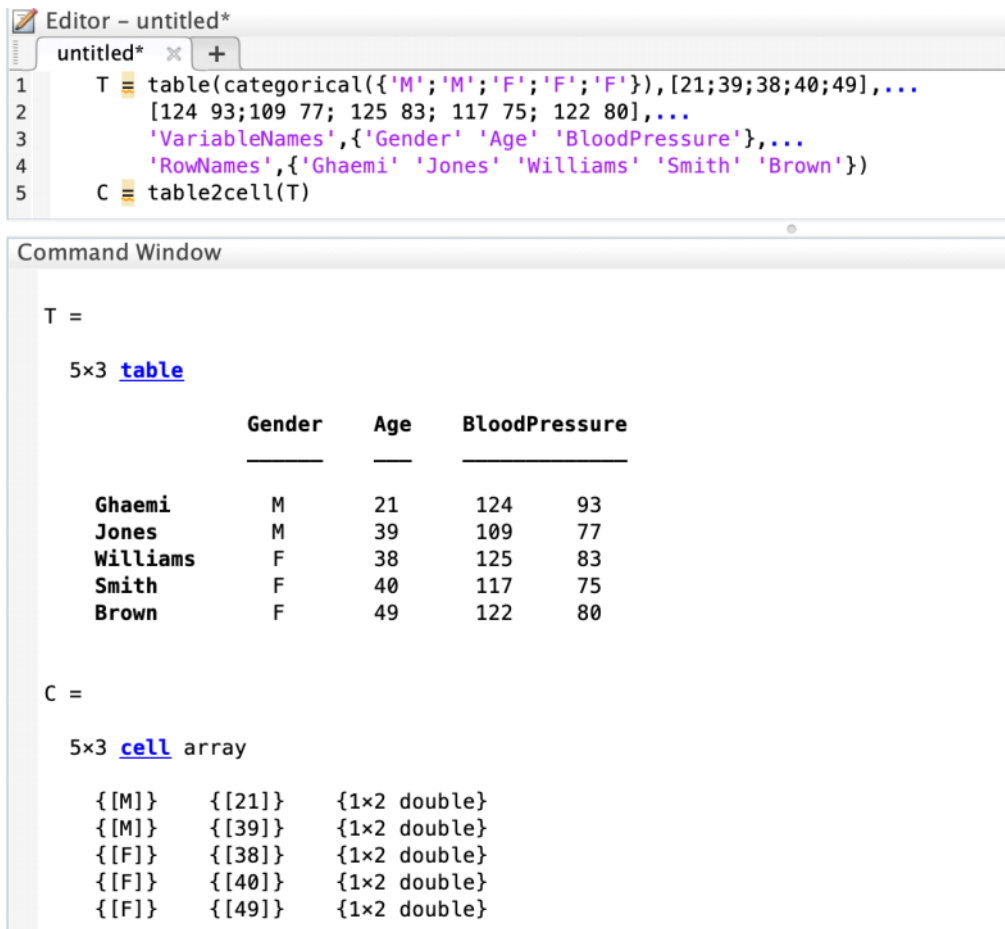
Example 5.6.2: Converting string to numerical array



The image shows a MATLAB Editor window with two lines of code: `str = '5 10 20 40 1 2 3 4 5';` and `n = str2num(str)`. Below the editor is the Command Window, which displays the result of the command: `n =` followed by the numerical array `5 10 20 40 1 2 3 4 5`.

Figure 5.27

Example 5.6.3: Converting a table to a cell array



The image shows a MATLAB Editor window with five lines of code: `T = table(categorical({'M';'M';'F';'F';'F'}),[21;39;38;40;49],... [124 93;109 77; 125 83; 117 75; 122 80],... 'VariableNames',{'Gender' 'Age' 'BloodPressure'},... 'RowNames',{'Ghaemi' 'Jones' 'Williams' 'Smith' 'Brown'})` and `C = table2cell(T)`. Below the editor is the Command Window, which displays the result of the command: `T =` followed by a 5x3 table with columns **Gender**, **Age**, and **BloodPressure**. The rows are **Ghaemi**, **Jones**, **Williams**, **Smith**, and **Brown**. Below the table, it displays `C =` followed by a 5x3 cell array with the following contents: `{[M]}`, `{[21]}`, `{1x2 double}`; `{[M]}`, `{[39]}`, `{1x2 double}`; `{[F]}`, `{[38]}`, `{1x2 double}`; `{[F]}`, `{[40]}`, `{1x2 double}`; `{[F]}`, `{[49]}`, `{1x2 double}`.

Figure 5.28

Example 5.6.4: Converting homogenous array to table



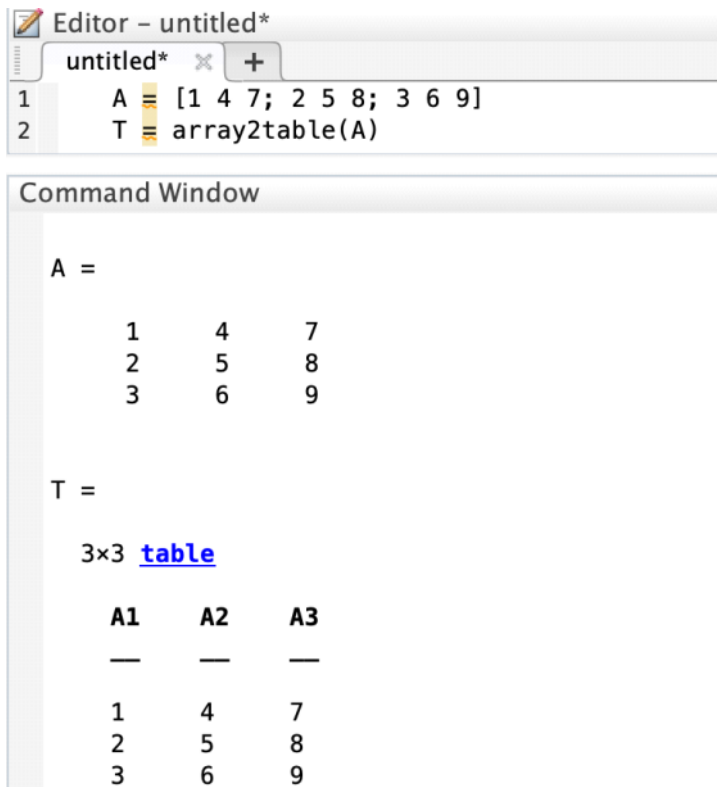


Figure 5.29

Example 5.6.5: Converting structure array to cell array

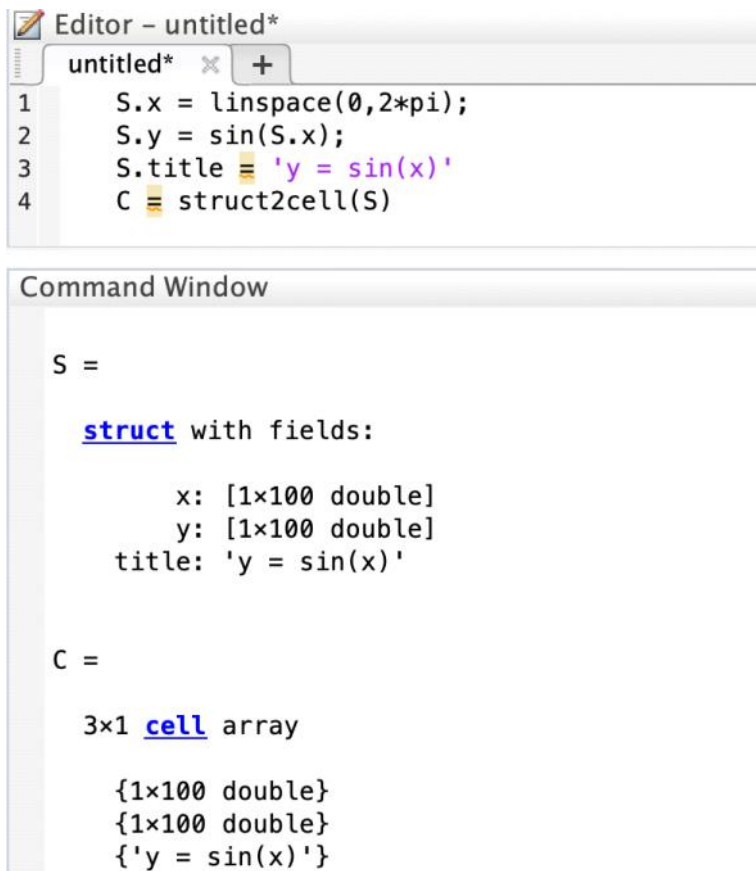


Figure 5.30

## Student Exercises

1. Write a program that finds the mean of an array of numbers entered by the user.
2. Create a program that will loop through an array of integers and add one to each odd element.
3. Create a 10x10 array using **magic(10)** and use a loop to continuously remove from its columns and rows until we have a 5x5 array
4. Write a program that will read an array of any size, removing all values that are less than 60, and replaces all values that are from 60-69 with 70.
5. This program will receive an array of 2-dimensional coordinates and output a line that passes through all points. If a line cannot pass through all points then the program will instead output 'impossible.'
6. Create a program that will store the name and grade of a set of students into a table and displays them at the end.
7. Write a program that accepts an array of numbers and attempts to find between which elements at least one minimum or maximum exists.
8. Create a program that holds a database of different elements, their mass, their charge, and their half-life (if they have one). Use structures for this and repeatedly ask the user if they wish to enter a new entry until the user is done. When done, output a table of all the elements.

# CHAPTER 6

## Classes and Functions:

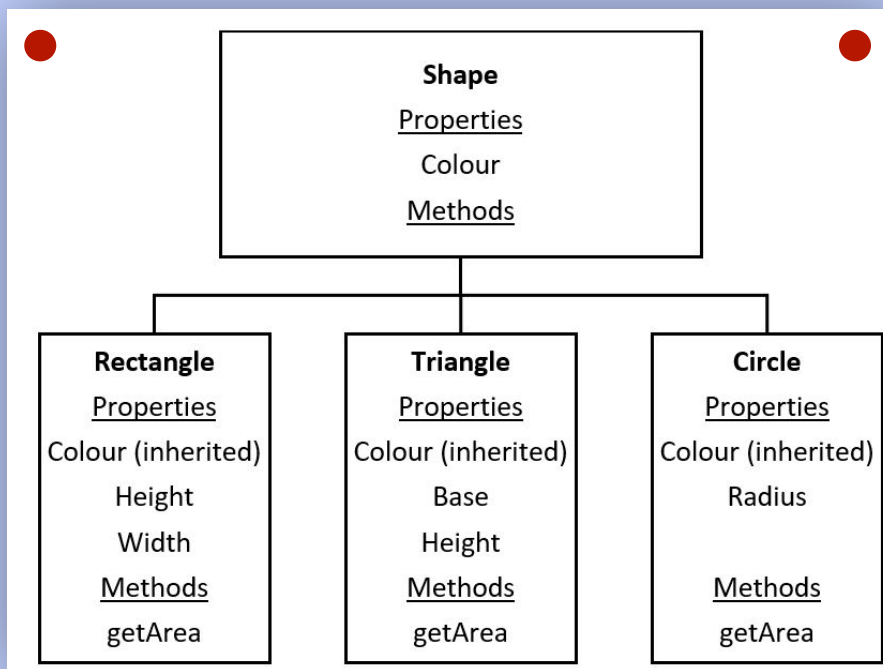
Section 1: Functions

Section 2: Classes

Section 3: Enumeration

Section 4: Recursion

### Student Exercises



## Section 1: Functions

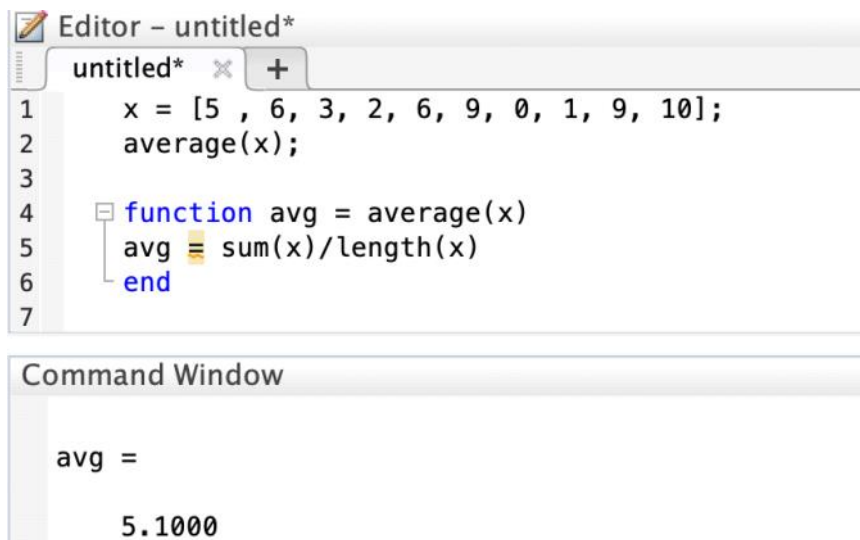
**A** function is a group of statements that together perform a task. In MATLAB, functions are defined in separate files. Functions can have both single and multiple inputs and outputs, meaning, a function will take at least one input value and will output at least one value. The syntax to use functions is:

```
1 function [a, b, c, ...] = myFunction(d, e, f, ...)
2     ...
3 end
```

The code above will declare a function named 'myFunction' and will take 'd, e, f, ...' as inputs and 'a, b, c, ...' as output values.

**Note** that valid function names begin with an alphabetic character, and can contain letters, numbers, and underscores.

For example, to write a function that will take the average of a series of numbers, we can do:



The screenshot shows the MATLAB Editor window titled 'Editor - untitled\*'. The code in the editor is as follows:

```
1 x = [5 , 6, 3, 2, 6, 9, 0, 1, 9, 10];
2 average(x);
3
4 function avg = average(x)
5     avg = sum(x)/length(x)
6 end
7
```

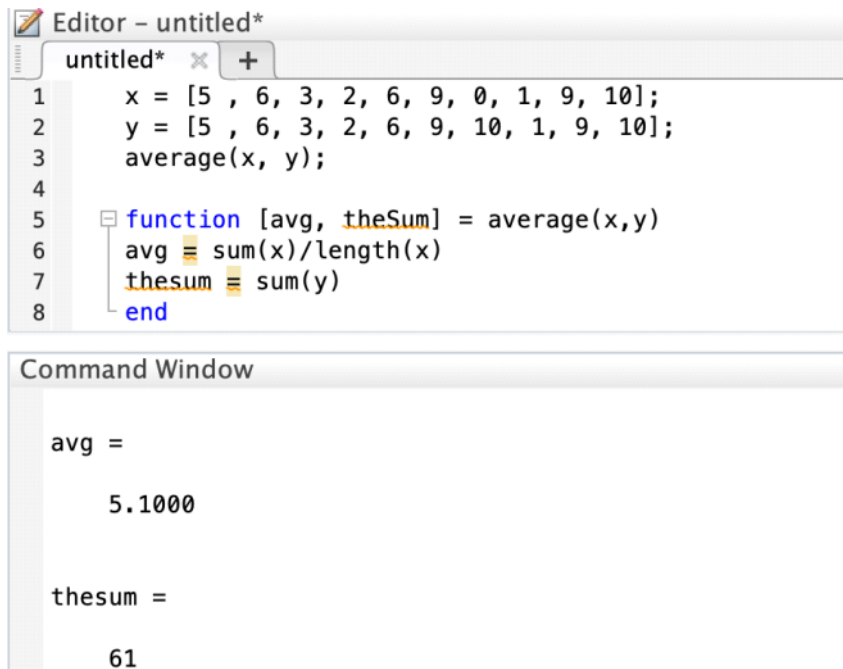
Below the editor is the Command Window, which displays the output of the function call:

```
avg =
    5.1000
```

Figure 6.1

**Note** that `sum(x)`, will sum all of the values of `x`. And `length(x)`, will return how many values are in `x`.

And to write a function that will accept two values and output two values, we do:



The screenshot shows a MATLAB Editor window titled "Editor - untitled\*" with a tab for "untitled\* x +". The code in the editor is as follows:

```
1 x = [5 , 6, 3, 2, 6, 9, 0, 1, 9, 10];
2 y = [5 , 6, 3, 2, 6, 9, 10, 1, 9, 10];
3 average(x, y);
4
5 function [avg, theSum] = average(x,y)
6     avg = sum(x)/length(x)
7     thesum = sum(y)
8     end
```

Below the editor is the Command Window, which displays the results of the function call:

```
avg =
    5.1000

thesum =
    61
```

Figure 6.2

In MATLAB there is a 'return' command, which will exit the function. For example, to write a function that exits if  $n = 2$ , we can do:

```
1 function test = test(x)
2     if (x==2)
3         disp("the number was 2");
4         return
5     end
6     disp("The number was not two");
7 end
```

## Section 2: Classes

**A** MATLAB class definition is a template whose purpose is to provide a description of all the elements that are common to all instances of the class. The basic purpose of a class is to define an object that encapsulates data and the operations performed on that data.

For example, SimpleClass defines a property and two methods that operate on the data in that property:

Value — Property that contains the data stored in an object of the class

Subtract— Method that subtracts 2 from the value

Add— Method that adds 'n' to the value

```
1 classdef SimpleClass
2     properties
3         Value
4     end
5     methods
6         function r = subtract(obj)
7             r = [obj.Value] - 2;
8         end
9         function r = add(obj, n)
10            r = [obj.Value] + n;
11        end
12    end
13 end
```

To create an object of the class using the class name we do:

```
1 a = SimpleClass
```

To assign a value to a property, we can do:

```
1 a.Value = 5
```

To read a value of a property, we do:

```
1 a.Value
```

To call a method on object a that we just created, we do:

```
1 subtract(a)
2 %OR
3 add(a,3) %This will add replace 'n' with 3, which will add 3 to Value
4 %OR
5 a.add(3) %This is called the dot notation
```

Classes can define a special method to create objects, called a constructor. Constructor

methods enable you to pass arguments to the constructor, and to validate and assign property values. Here is a constructor for the SimpleClass class:

```
1 methods
2     function obj = BasicClass(val)
3         if nargin > 0
4             if isnumeric(val)
5                 obj.Value = val;
6             else
7                 error('Value must be numeric')
8             end
9         end
10    end
11 end
```

**Note** The name of the class and the constructor should be the same.

**Note** that adding a constructor will make it easier to create objects, now to create an object, we can alternatively do:

```
1 a = SimpleClass(5) %This will assign Value to be 5
```

Class members are the properties, methods, and events that define the class. Attributes modify the behavior of classes and the members defined in the class-definition block. For example, you can specify that methods are static or that properties are private.

1. Class Attributes
2. Method Attributes
3. Property Attributes
4. Event Attributes

**Class Attributes:** All classes support the attributes listed in the following table. Attributes enable you to modify the behavior of class. Attribute values apply to the class defined within the '*classdef*' block. The syntax is:

```
1 classdef (Attribute1 = value1, Attribute2 = value2, ...) ClassName
2     ...
3 end
```

Attribute Name	Class	Description
Abstract	logical (default = false)	If specified as true, this class is an abstract class (cannot be instantiated).
AllowedSubclasses	meta.class object or cell array of meta.class objects	List classes that can subclass this class. Specify subclasses as meta.class objects in the form: <ul style="list-style-type: none"> <li>• A single meta.class object</li> <li>• A cell array of meta.class objects. An empty cell array, {}, is the same as a Sealed class (no subclasses).</li> </ul>
ConstructOnLoad	logical (default = false)	If true, MATLAB calls the class constructor when loading an object from a MAT-file. Therefore, implement the constructor so it can be called with no arguments without producing an error.
HandleCompatible	logical (default = false) for value classes	If specified as true, this class can be used as a superclass for handle classes. All handle classes are HandleCompatible by definition.
Hidden	logical (default = false)	If true, this class does not appear in the output of the superclasses or help functions.
InferiorClasses	meta.class object or cell array of meta.class objects	Use this attribute to establish a precedence relationship among classes. Specify a cell array of meta.class objects using the ?operator.
Sealed	logical (default = false)	If true, this class cannot have sub classes

Table 6.1

**Method Attributes:** Specifying attributes in the class definition enables you to customize the behavior of methods for specific purposes. Control characteristics like access, visibility, and implementation by setting method attributes. Subclasses do not inherit superclass member attributes. With the syntax of:

```

1 methods(Attribute1 = value1, Attribute2 = value2, ...)
2     ...
3 end

```



Attribute Name	Class	Description
Abstract	logical default = false	<p>If true, the method has no implementation. The method has a syntax line that can include arguments that subclasses use when implementing the method:</p> <ul style="list-style-type: none"> <li>• Subclasses are not required to define the same number of input and output arguments. However, subclasses generally use the same signature when implementing their version of the method.</li> <li>• The method can have comments after the function line.</li> </ul>
Access	enumeration, default = public	<p>Determines what code can call this method:</p> <p>public — Unrestricted access</p> <p>protected — Access from methods in class or subclasses</p> <p>private — Access by class methods only (not from subclasses)</p>
Hidden	Logical default = false	<p>When false, the method name shows in the list of methods displayed using the methods or methodsview commands. If set to true, the method name is not included in these listings and its method does not return true for this method name.</p>
Sealed	logical default = false	<p>If true, the method cannot be redefined in a subclass. Attempting to define a method with the same name in a subclass causes an error.</p>
Static	Logical default = false	<p>Specify as true to define a method that does not depend on an object of the class and does not require an object argument. Use the class name to call the method: <i>classname.methodname</i> or an instance of the class: <i>obj.methodname</i></p>

Table 6.2

**Property Attributes:** Specifying attributes in the class definition enables you to customize the behavior of properties for specific purposes. Control characteristics like access, data storage, and visibility of properties by setting attributes. Subclasses do not inherit superclass member attributes. The syntax is:

```

1 properties(Attribute1 = value1, Attribute2 = value2, ...)
2     ...
3 end

```

Attribute Name	Class	Description
AbortSet	logical default = false	If true, MATLAB does not set the property value if the new value is the same as the current value. MATLAB does not call the property set method, if one exists.
Abstract	logical default = false	<p>If true, the property has no implementation, but a concrete subclass must redefine this property without Abstract being set to true.</p> <ul style="list-style-type: none"> <li>• Abstract properties cannot define set or get access methods.</li> <li>• Abstract properties cannot define initial values.</li> <li>• All subclasses must specify the same values as the superclass for the property SetAccess and GetAccess attributes.</li> <li>• Abstract=true use with the class attribute Sealed=false (the default).</li> </ul>
Access	• enumeration, default = public	<p>Use Access to set both SetAccess and GetAccess to the same value. Query the values of SetAccess and GetAccess directly (not Access).</p> <p>public – unrestricted access</p> <p>protected – access from class or subclasses</p> <p>private – access by class members only (not subclasses)</p>
Constant	logical default = false	<p>Set to true if you want only one value for this property in all instances of the class:</p> <ul style="list-style-type: none"> <li>• Subclasses inherit constant properties, but cannot change them.</li> <li>• Constant properties cannot be Dependent.</li> <li>• SetAccess is ignored.</li> </ul>
Dependent	logical default = false	<p>If false, property value is stored in object. If true, property value is not stored in object. The set and get functions cannot access the property by indexing into the object using the property name.</p> <p>MATLAB does not display in the command window the names and values of Dependentproperties that do not define a get method (scalar object display only).</p>
GetAccess	enumeration default = public	<p>public — unrestricted access</p> <p>protected — access from class or subclasses</p> <p>private — access by class members only (not from</p>

		subclasses)  MATLAB does not display in the command window the names and values of properties having protected or private GetAccess or properties whose Hidden attribute is true.
GetObservable	logical default = false	If true, and it is a handle class property, then you can create listeners for access to this property. The listeners are called whenever property values are queried.
Hidden	logical default = false	Determines if the property can be shown in a property list (e.g., Property Inspector, call to set or get, etc.).  MATLAB does not display in the command window the names and values of properties whose Hidden attribute is true or properties having protected or privateGetAccess.
NonCopyable	logical default = false	Determine if property value can be copied when object is copied.  You can set NonCopyable to true only in handle classes.
SetAccess	enumeration default = public	public — unrestricted access  protected — access from class or subclasses  private — access by class members only (not from subclasses)  immutable — property can be set only in the constructor.
SetObservable	logical default = false	If true, and it is a handle class property, then you can create listeners for access to this property. The listeners are called whenever property values are modified.
Transient	logical default = false	If true, property value is not saved when object is saved to a file.

Table 6.3

**Event Attributes:** The following table lists the attributes you can set for events. To specify a value for an attribute, assign the attribute value on the same line as the event keyword. For example, all the events defined in the following events block have protected ListenAccess and private NotifyAccess.

```

1 events (ListenAccess= protected , NotifyAccess= private)
2     EventName1
3     EventName2
4 end

```

Attribute Name	Class	Description
Hidden	logical default = false	If true, event does not appear in list of events returned by events function (or other event listing functions or viewers).
ListenAccess	Enumeration default = public	Determines where you can create listeners for the event. public — Unrestricted access protected — Access from methods in class or subclasses private — Access by class methods only (not from subclasses)
NotifyAccess	Enumeration default = public	Determines where code can trigger the event public — Any code can trigger event protected — Can trigger event from methods in class or derived classes private — Can trigger event by class methods only (not from derived classes)

Table 6.4

There are two kinds of MATLAB classes—handle classes and value classes:

- Value classes represent independent values. Value objects contain the object data and do not share this data with copies of the object. MATLAB numeric types are value classes. Values objects passed to and modified by functions must return a modified object to the caller.
- Handle classes create objects that reference the object data. Copies of the instance variable refer to the same object. Handle objects passed to and modified by functions affect the object in the caller’s workspace without returning the object.

**Class definitions** are blocks of code that are delineated by the `classdef` keyword at the beginning and the `end` keyword at the end. Files can contain only one class definition. The following diagram shows the syntax of a `classdef` block. Only comments and blank lines can precede the `classdef` keyword.

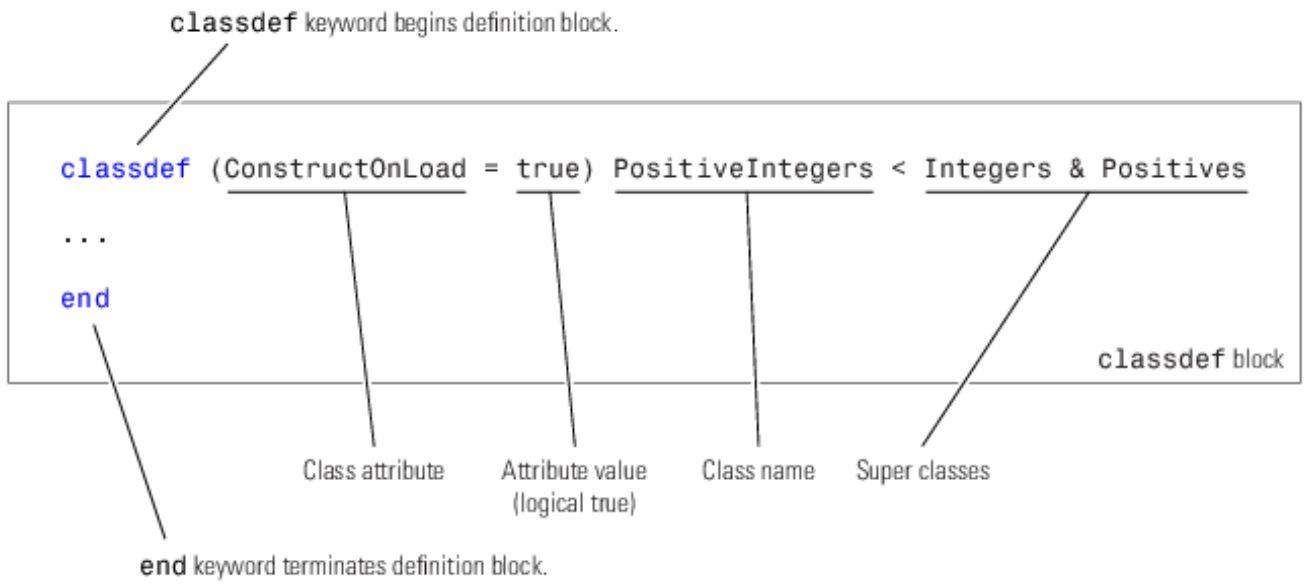
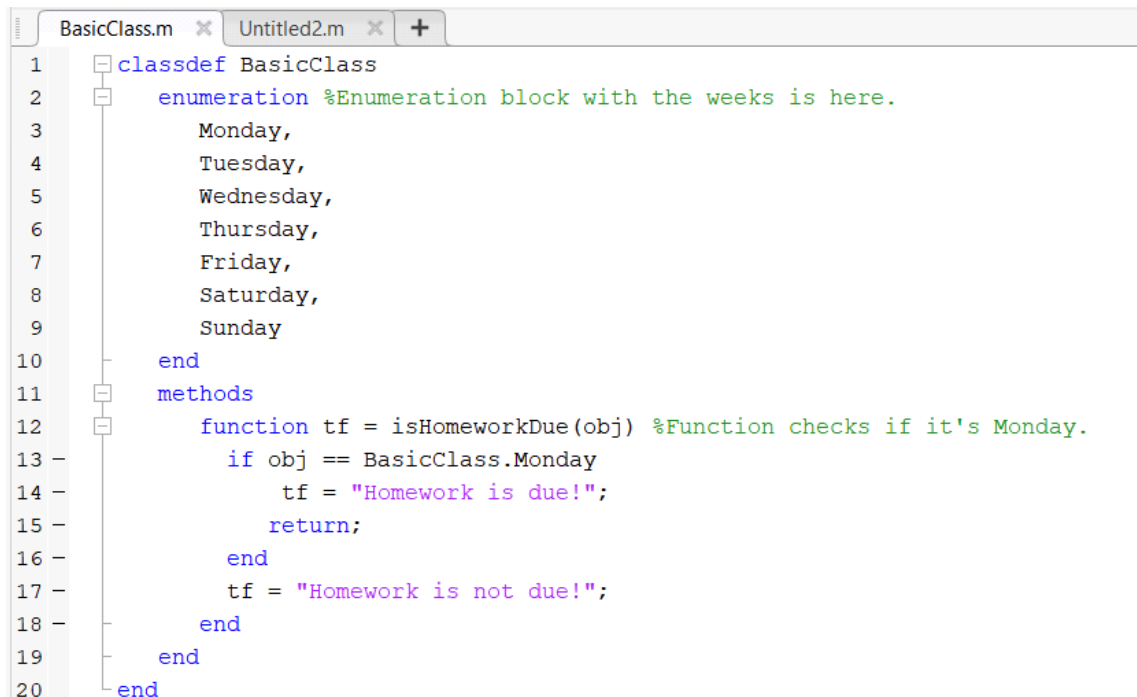


Figure 6.3

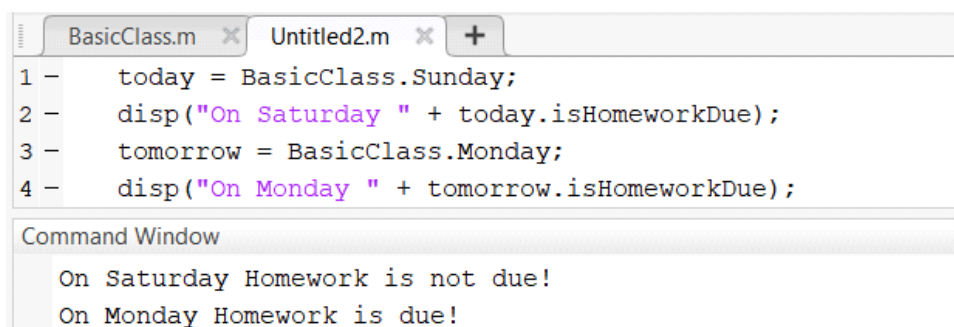
## Section 3: Enumeration

**E**numeration is a data type in MATLAB that allows for a set of values to be represented through categories. An enumeration class is created by adding an **enumeration** block inside a class. This can be useful when we want a limited number of choices rather than the infinite number of choices numbers provide. For example, days of the week can be represented with numbers, however, it is much more readable to represent them through enums as shown below:



```
1  classdef BasicClass
2      enumeration %Enumeration block with the weeks is here.
3          Monday,
4          Tuesday,
5          Wednesday,
6          Thursday,
7          Friday,
8          Saturday,
9          Sunday
10     end
11     methods
12     function tf = isHomeworkDue(obj) %Function checks if it's Monday.
13         if obj == BasicClass.Monday
14             tf = "Homework is due!";
15             return;
16         end
17         tf = "Homework is not due!";
18     end
19     end
20 end
```

Figure 6.4



```
1  today = BasicClass.Sunday;
2  disp("On Saturday " + today.isHomeworkDue);
3  tomorrow = BasicClass.Monday;
4  disp("On Monday " + tomorrow.isHomeworkDue);
```

Command Window

```
On Saturday Homework is not due!
On Monday Homework is due!
```

Figure 6.5

Figure 6.4 creates the enumeration block and defines a function which checks if the current value is Monday. The next figure picks two different days and runs the function on them.

Numeric values can be assigned to enums as well by following the following class format:

```

1 classdef ClassName < int32
2     enumeration
3         item1 (12),
4         item2 (55),
5         item3 (34)
6     end
7 end

```

Here is an example using the days of the week again:

```

BasicClass.m x  Untitled2.m x  +
1  classdef BasicClass < int32
2      enumeration
3          Sunday (1),
4          Monday (2),
5          Tuesday (3),
6          Wednesday (4),
7          Thursday (5),
8          Friday (6),
9          Saturday (7)
10     end
11     methods
12         function tf = getDayNum(obj)
13             tf = char(obj) + " is day " + int32(obj) + " of the week.";
14         end
15     end
16 end

```

Figure 6.6

Here is the class in action:

```

BasicClass.m x  Untitled2.m x  +
1 - today = BasicClass.Wednesday;
2 - disp(today.getDayNum())

```

Command Window

```

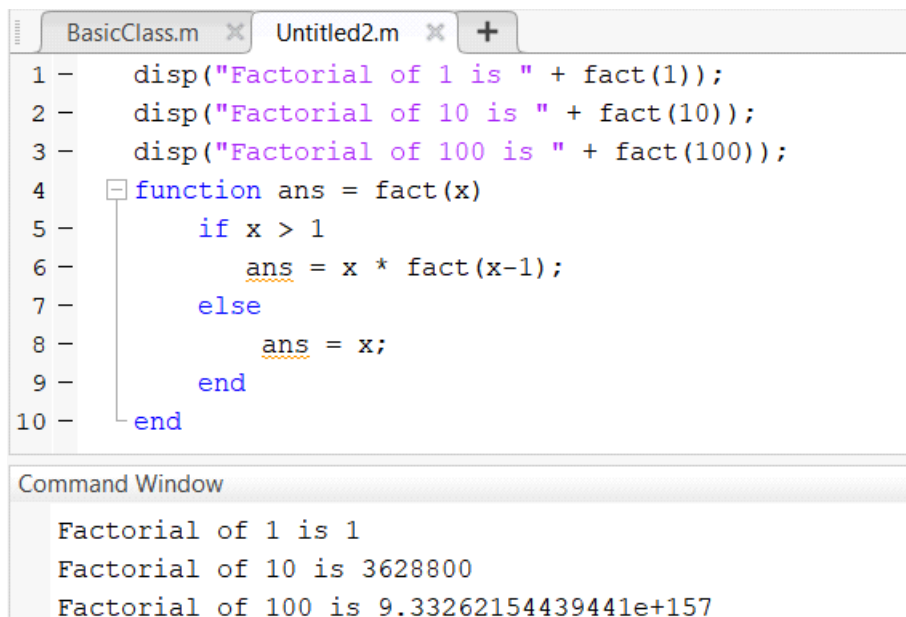
Wednesday is day 4 of the week.

```

Figure 6.7

## Section 4: Recursion

Loops are not the only method of iteration. Recursion is an algorithm that accomplishes a loop by repeating itself until a condition is met. In programming, this involves a function calling itself until it is forced to stop. Perhaps the most common example of recursion is the factorial which can be found below:



```
BasicClass.m x  Untitled2.m x  +
1 -   disp("Factorial of 1 is " + fact(1));
2 -   disp("Factorial of 10 is " + fact(10));
3 -   disp("Factorial of 100 is " + fact(100));
4 -   function ans = fact(x)
5 -       if x > 1
6 -           ans = x * fact(x-1);
7 -       else
8 -           ans = x;
9 -       end
10 -  end

Command Window
Factorial of 1 is 1
Factorial of 10 is 3628800
Factorial of 100 is 9.33262154439441e+157
```

Figure 6.8

The function multiplies its parameter X by the result of calling itself with the parameter X-1 until X reaches one. The table below will present how the factorial of 4 is calculated step by step using this method:

X	Fact(X)
4	4 * Fact(3)
3	3 * Fact(2)
2	2 * Fact(1)
1	1
<b>Result</b>	<b>4 * 3 * 2 * 1</b>

Table 6.5

Another example with recursion could involve using it to reverse a string:



```

BasicClass.m x  Untitled2.m x  +
1 - disp(reverse('MATLAB is fun!'));
2 - function ans = reverse(x)
3 -     if length(x) > 1
4 -         ans = "" + x(length(x)) + reverse(x(1:length(x)-1));
5 -     else
6 -         ans = x;
7 -     end
8 - end

Command Window
!nuf si baltam

```

Figure 6.9

This one works by continuously taking letters from the end of the word until we run out of letters. The table once again shows what is happening step by step.

X	Reverse(X)
MATLAB	B + Reverse(MATLA)
MATLA	A + Reverse(MATL)
MATL	L + Reverse(MAT)
MAT	T + Reverse(MA)
MA	A + Reverse(M)
M	M
<b>Result</b>	B + A + L + T + A + M

Table 6.6

## Student Exercises

1. Write a function that will find and return the minimum and maximum values in an array as well as the min and max's index in the array. Example:  
Enter array: [5, 3, 7, 10]  
The maximum value is 10 at index 4.  
The minimum value is 3 at index 2.
2. Define a class that contains the days of the week, and will be able to output the day after the entered day. For example, if the user inputs Monday, the program will output Tuesday.
3. Using recursion, create a function, where given an array and a number, it checks for the first instance of the number in the array and which index it is located at.
4. Create a function that given a, b, and c, will solve the quadratic formula.
5. Create a function that checks if a word is a palindrome (the same when written backwards) using recursion.
6. Construct a function that accepts an array of  $M1 \times N1$  and an array of  $M2 \times N2$  and outputs their dot product. If the two matrices are not compatible, then the function should return a warning.
7. Construct a class that has a variety of functions based on which variables (time, acceleration, initial velocity, final velocity, and displacement) the user submits to solve for another variable (hint: use the kinematic equations and assume constant acceleration).
8. Given a recursive, exponential formula  $a(n)$ , an initial value  $a(1)$ , and a value for  $n$ , create a function to find  $a(n)$  using recursion.  $A(n)$  will follow the following basic format:  $R * a(n - 1)$
9. Create a class that makes true and false tests and stores them for later use. When a new Test is created, the user can call a method that takes in a question and answer. An array of all questions and answers in that test will be recorded. The user can then run the Test with another function resulting in a question to be displayed followed by the computer waiting for the user to answer. At the end of the test, the user's score is displayed.

# CHAPTER 7

## Data and Statistics:

Section 1: Basic Statistical Function

Section 2: Importing Data

Section 3: Plotting Data

Section 4: Analyzing Data

**Student Exercises**



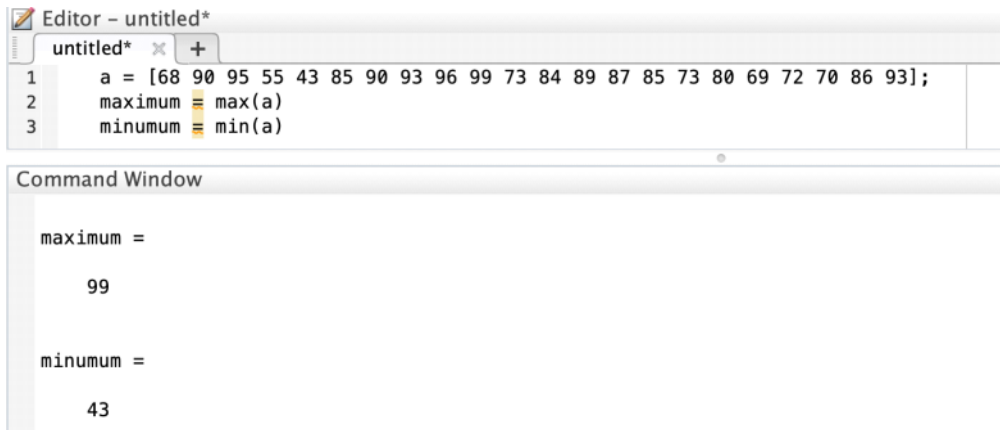
## Section 1: Basic Statistical Functions

MATLAB has predefined statistical functions that each perform a task. Some of the functions are as shown in the table below:

Function	Description
max	Maximum value
mean	Average or mean value
median	Median value
min	Smallest value
mode	Most frequent value
std	Standard deviation
var	Variance, which measures the spread or dispersion of the values

Table 7.1

For example, you are given an array of 22 students' grades called '*a*', and are trying to find the highest and lowest grades. You can use the MATLAB functions 'max' and 'min' respectively as follows:



```
Editor - untitled*
untitled* x +
1 a = [68 90 95 55 43 85 90 93 96 99 73 84 89 87 85 73 80 69 72 70 86 93];
2 maximum = max(a)
3 mininum = min(a)

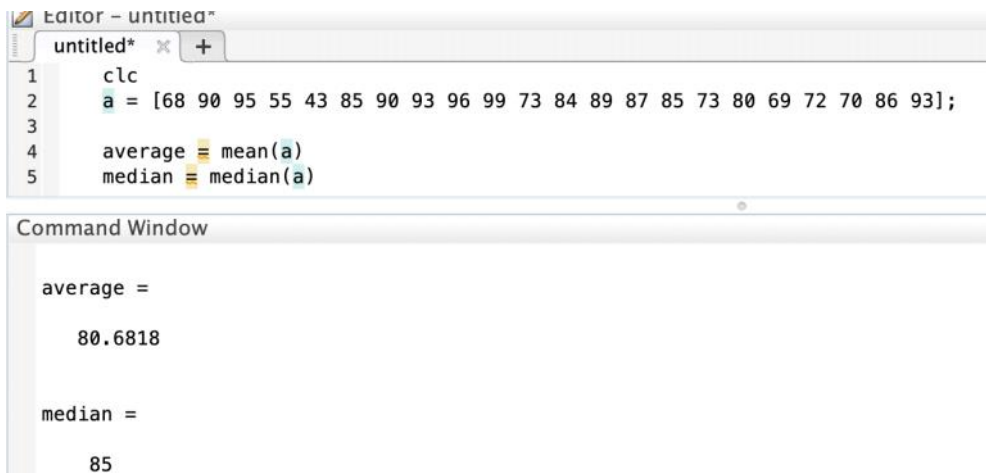
Command Window

maximum =
    99

mininum =
    43
```

Figure 7.1

And to get the average and median grades of the class, we can do:



```
Editor - untitled*
untitled* x +
1 clc
2 a = [68 90 95 55 43 85 90 93 96 99 73 84 89 87 85 73 80 69 72 70 86 93];
3
4 average = mean(a)
5 median = median(a)

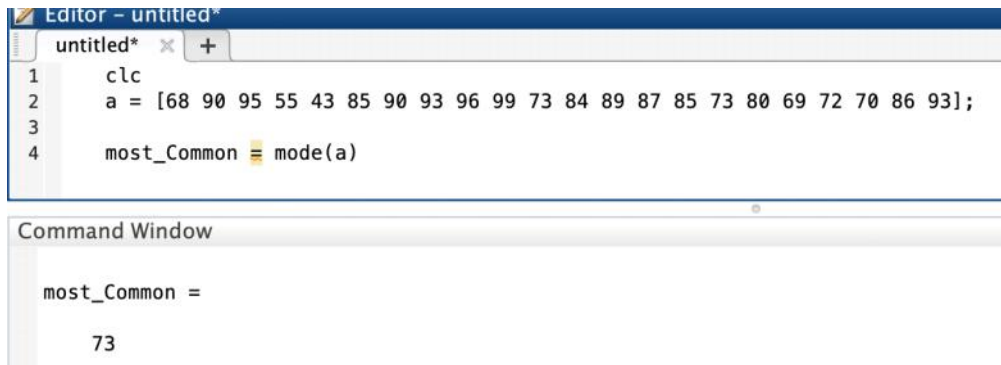
Command Window

average =
    80.6818

median =
    85
```

Figure 7.2

To get the most frequent grade of the class we can use the 'mode' function:

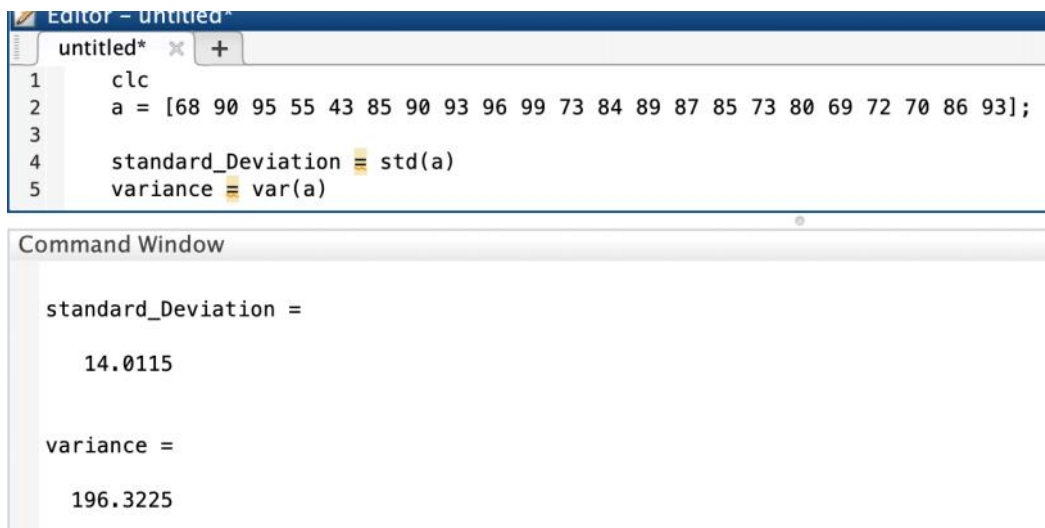


```
Editor - untitled*
untitled* x +
1   clc
2   a = [68 90 95 55 43 85 90 93 96 99 73 84 89 87 85 73 80 69 72 70 86 93];
3
4   most_Common = mode(a)

Command Window
most_Common =
    73
```

Figure 7.3

The functions 'std' and 'var' allow us to get the standard deviation and variance respectively:



```
Editor - untitled*
untitled* x +
1   clc
2   a = [68 90 95 55 43 85 90 93 96 99 73 84 89 87 85 73 80 69 72 70 86 93];
3
4   standard_Deviation = std(a)
5   variance = var(a)

Command Window
standard_Deviation =
    14.0115

variance =
    196.3225
```

Figure 7.4

## Section 2: Importing Data

In MATLAB there are different methods of importing data, you can import data from a disk file or the system clipboard.

To import data from a file, do one of the following:

- On the **Home** tab, in the **Variable** section, select **Import Data**

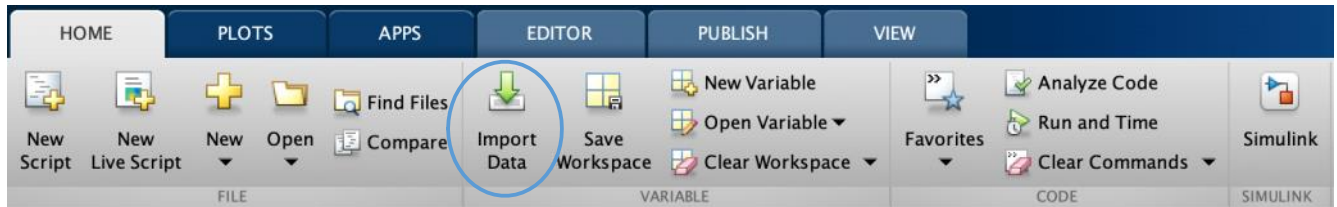


Figure 7.5

- Double-click a file name in the Current Folder browser.

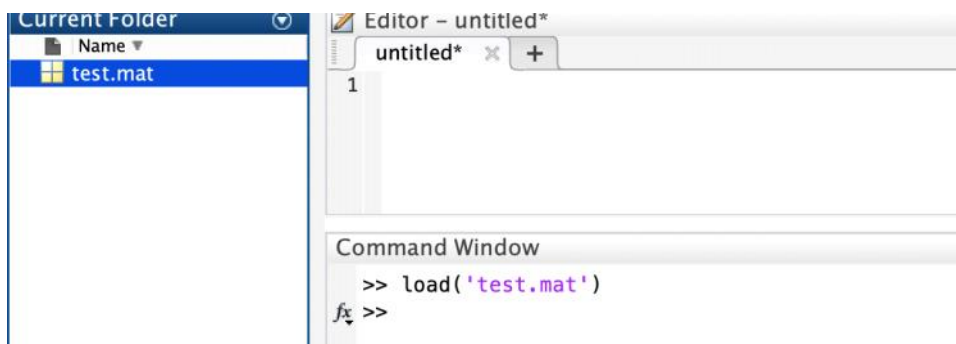


Figure 7.6

- Call **uiimport**: *uiimport(filename)* opens the file specified in filename using either **Import Tool** or **Import Wizard** depending on the file type. For spreadsheet and text files, uiimport opens the file using **Import Tool**. For all other file types, such as image, audio, or MAT-files, uiimport opens the file using **Import Wizard**.

To import data from the clipboard, do one of the following:

- On the Workspace browser title bar, click the drop down, and then select **Paste**.

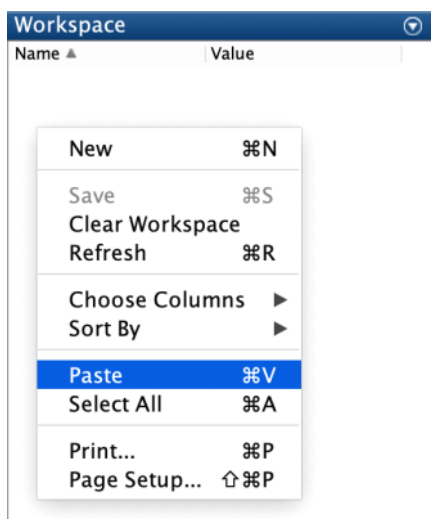


Figure 7.7

## Section 3: Plotting Data

Previously in the chapter we learned about importing data into MATLAB. MATLAB allows us to seamlessly plot 2d graphs. We can use the **plot(x,y)** function to plot a data. For example to plot a data of monthly sales in MATLAB, we can do the following.

```
1 months = [1 2 3 4 5 6 7 8 9 10 11 12] ;  
2 sales = [1.4 1.2 1.5 1.8 1.4 1.6 1.9 1.3 1.2 1.1 1.8 2.3];  
3 plot(months, sales)  
4 xlabel('months')  
5 ylabel('Sales')  
6 title('Monthly Sales')
```

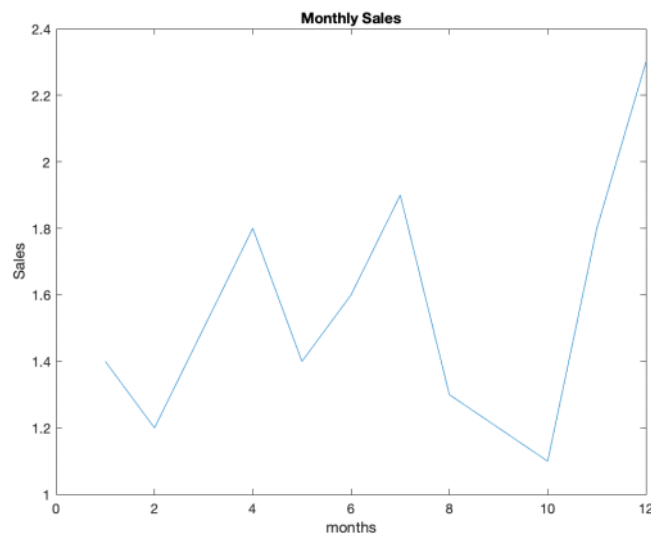


Figure 7.8

**Note** that 'months' and 'sales' are representing **x** and **y** respectively. Also **note** that **xlabel** and **ylabel** are used to label the xy-axis, and that **title** is used to title the graph.

If you are trying to plot multiple sales on one graph to compare and see the differences of the two, you can do the following:

```
1 months = [1 2 3 4 5 6 7 8 9 10 11 12] ;  
2 Chris_sales = [1.4 1.2 1.5 1.8 1.4 1.6 1.9 1.3 1.2 1.1 1.8 2.3];  
3 Gene_sales = [1.9 1.7 2.5 2.8 1.9 1.4 1.3 2.3 2.6 2.7 2.8 3.1];  
4 plot(months, Chris_sales, months, Gene_sales)  
5 xlabel('months')  
6 ylabel('Sales')  
7 title('Monthly Sales')  
8 legend('Chris', 'Gene')
```

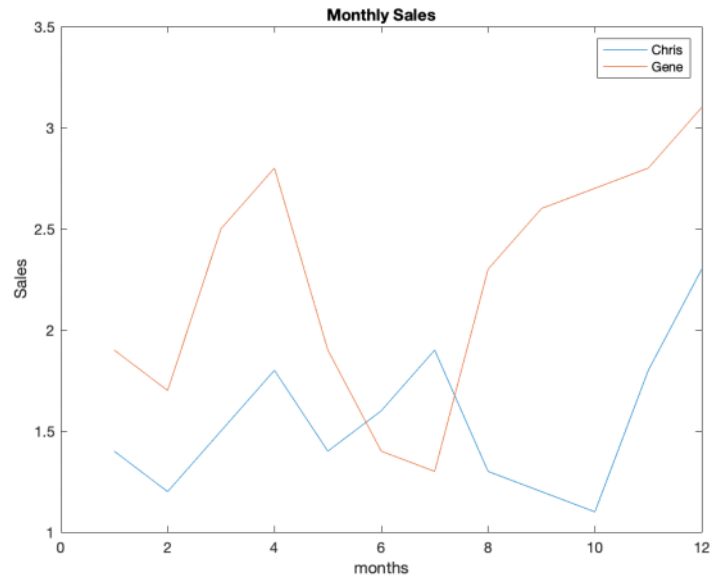


Figure 7.9

**Note** that in `plot` we used the same variable for months, but different variable for sales. Also **note** that **legend** will label the different plots in a box on the upper right corner of the graph.

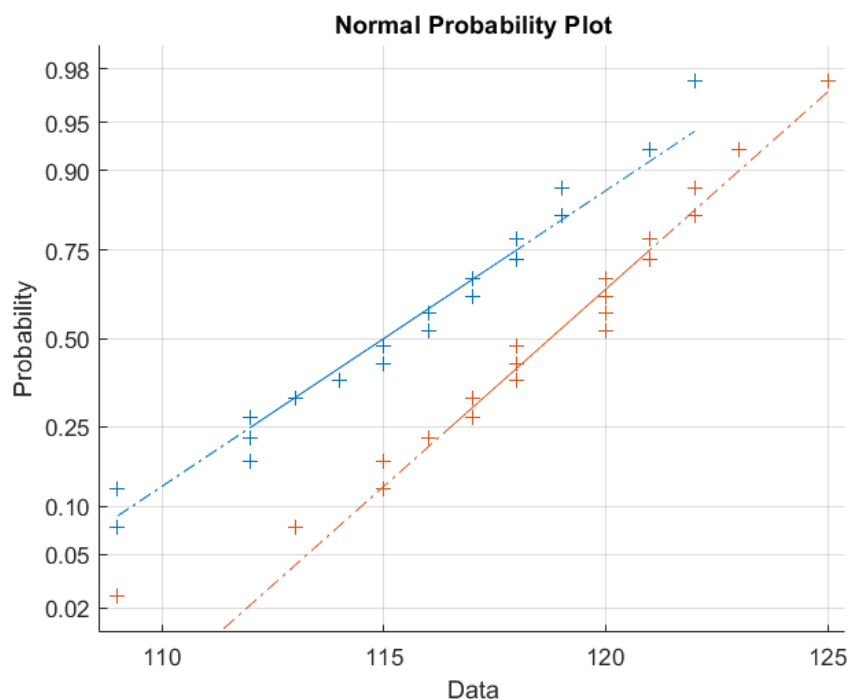


## Section 4: Analyzing Data

In this section, we will analyze a single dataset using multiple tools MATLAB gives us. This will not be a fully comprehensive guide to every or even most statistical tools MATLAB has to offer but it will give a good general overview.

This section will use one of MATLAB's default datasets called **gas** which can be loaded with the command **load gas**. Before we begin testing, we want to make sure the dataset is normally distributed. MATLAB contains multiple tests of normality, in this example, we will use the Anderson-Darling test called **adtest()**. As a general reminder, the null hypothesis in this test is that the sample population is normal while the alternate is that it is not, thus a value of zero would mean the null hypothesis cannot be rejected and that the dataset is probably normal. The function **normplot()** also allows us to look for normality in a more visual way.

```
1 load gas
2 prices = [price1 price2];
3 disp("price1 gives " + adtest(price1));
4 disp("price2 gives " + adtest(price2));
5 normplot(prices)
```



Command Window

```
price1 gives false
price2 gives false
```

Figure 7.9

The data points in the graph appear to be normally distributed and the normality test failed to reject both datasets.

We can test if the mean price (\$1.15) across the first dataset is statistically close to that of the second dataset with a t-test using the function **ttest()**.

```

1 load gas
2 prices = [price1 price2];
3 [h,pvalue] = ttest(price2/100,1.15);
4 disp("Test gives " + h);
5 disp("P value is " + pvalue);

```

```

Command Window
Test gives 1
P value is 0.00049517

```

Figure 7.10

Our results reject the null hypothesis, meaning the second dataset's mean is different from the first's.

Finally, we can visualize all this with a boxplot using the **boxplot()** command.

```

1 load gas
2 prices = [price1 price2];
3 boxplot(prices,1);
4 A = gca;
5 A.XTick = [1 2];
6 A.XTickLabel = {'January', 'February'};
7 xlabel('Month')
8 ylabel('Prices ($0.01)')
9 title('January vs February')

```

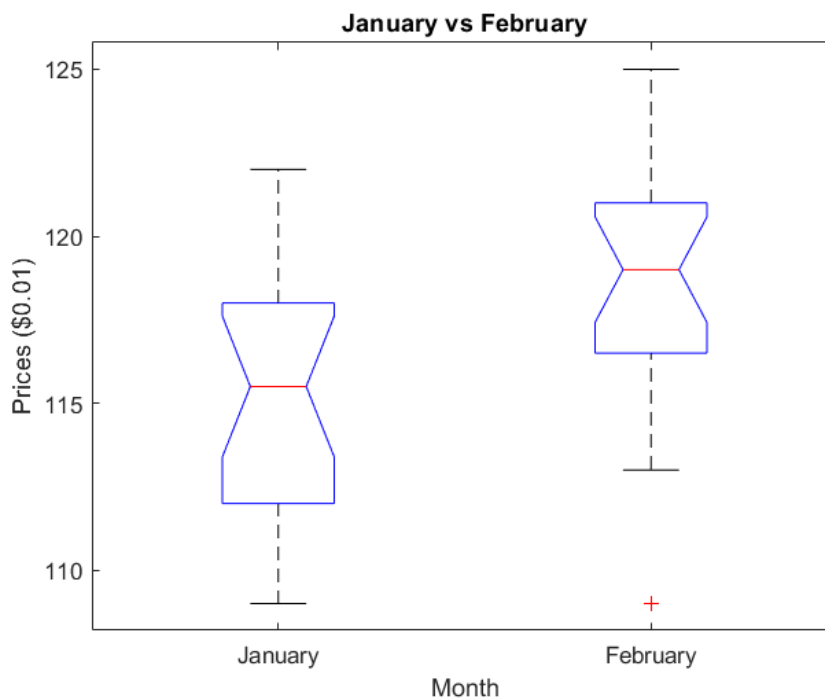


Figure 7.11

The boxplots shows the distribution of the data, the median, as well as any outliers, making it one of the best methods for analyzing data visually.

## Student Exercises

**Be sure to use appropriate titles, labels, and legends in all plots.**

1. Write a program that will calculate the Grade Point Average (GPA) of a student.
2. What does **uiimport** do?
3. (a) Create a scatter plot of two arrays called "**Engr Grades**" and "**Months**" which will plot the different grades during different times of a semester. (b) On the same plot add another array called "**CompSci Grades**". The plot will compare the Engineering students' grades with the Computer Science students' grades.
4. Find the region with the lowest and the region with the highest standard deviation in MATLAB's **flu** sample dataset.
5. Display the distribution of MATLAB's data samples of bacteria counts in different shipments of milk (**hogg**) in graphic form.
6. Organize all exam grades in the MATLAB sample data **examgrades** into a table of A's, B's, Cs, D's, and E's.
7. Create a function that accepts an array of numbers as data and other graphing features such as titles and labels, then plots the graph for the user.
8. Create a function that given an array of numbers, creates a plot with the data points as well as a line of best fit.

# CHAPTER 8

## Coordinate Systems:

Section 1: Plotting Basics

Section 2: Rectangular

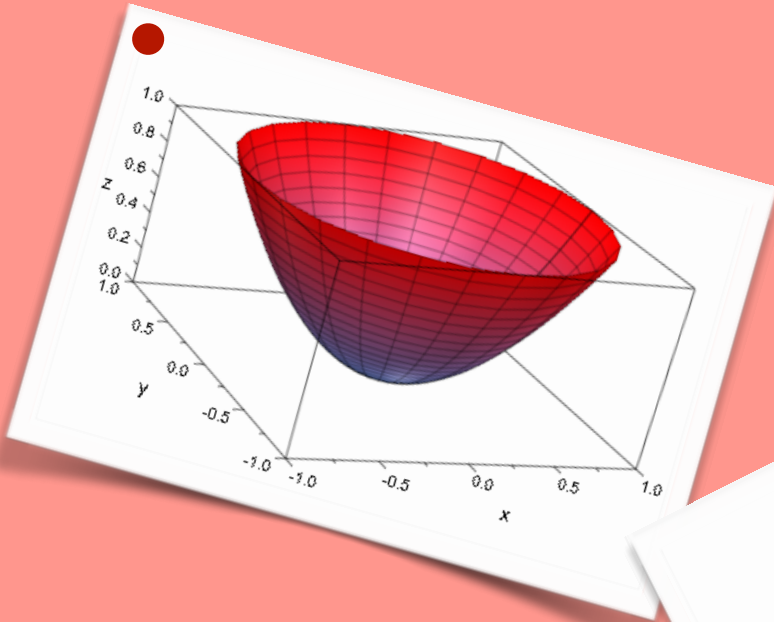
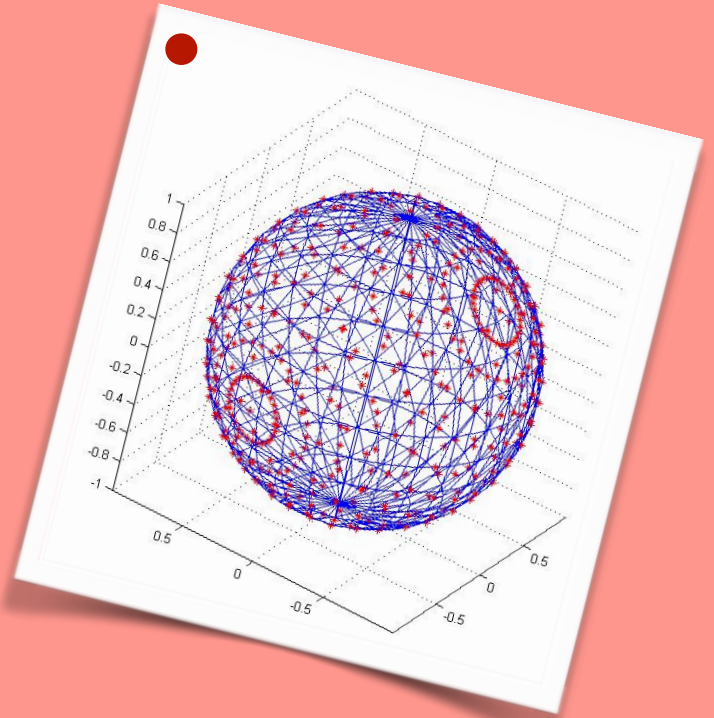
Section 3: Parametric

Section 4: Polar

Section 5: Cylindrical

Section 6: Spherical

**Student Exercises**



## Section 1: Plotting Basics

**P**lotting can be achieved by either giving MATLAB a set of points or an equation. The function **plot()** can display coordinates when given an array of x-values and y-values as well as a third parameter which controls whether the lines are connected not.

```
1 plot(x_array,y_array, '. or -');
```

In this case, a dot in the third parameter causes the coordinates to be separated while a dash connects them. Other line styles include '--' which creates dashed lines, ':' which creates a faded line, and 'o' which creates dots that are empty circles. Below is an example of a single format:

```
1 x = [0 1 2 3];  
2 y = [0 1 5 7];  
3 plot(x,y, 'o');
```

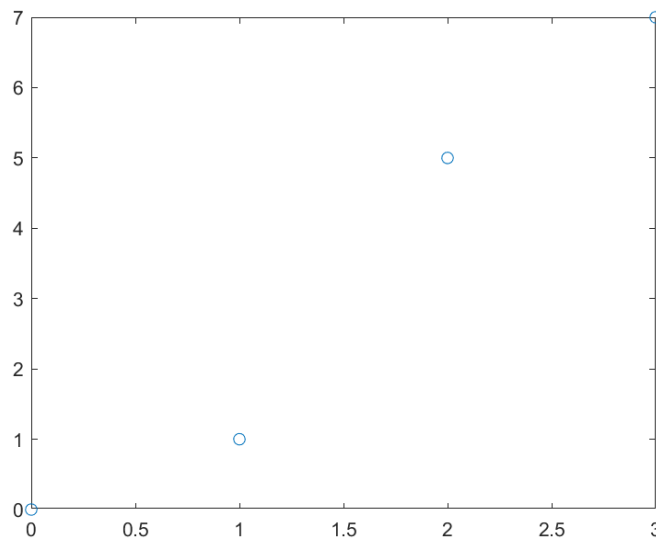
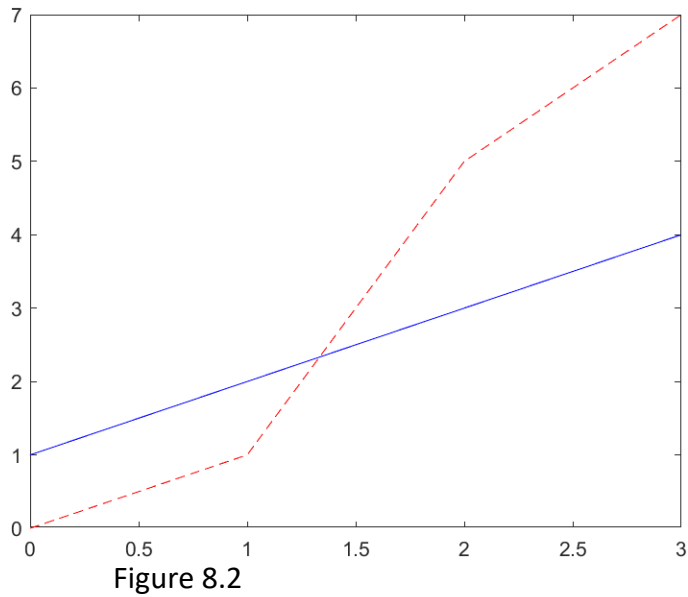


Figure 8.1

Multiple lines can be displayed by simply inserting another set of parameters. Also note worthy is that adding certain letter prefixes like 'r', 'g', and 'b' before the line style changes the line's color.

```
1 x = [0 1 2 3];  
2 y = [0 1 5 7];  
3 y2 = [1 2 3 4];  
4 plot(x,y, 'r--', x, y2, 'b--');
```

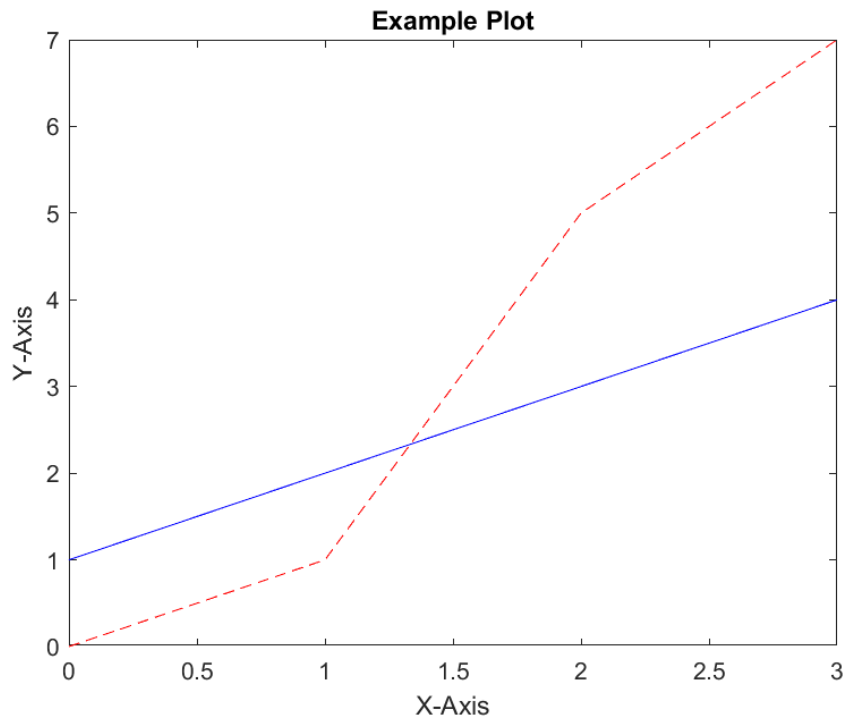


Adding labels is possible through the use of the functions **title()**, **ylabel()**, and **xlabel()**.

```

1 title("Example Plot");
2 ylabel("Y-Axis");
3 xlabel("X-Axis");
4 a(1) %print properties of the first line

```



Line four in the code prints changeable properties of the dashed (first) line, these properties include:

```

1 Line with properties:
2
3     Color: [1 0 0]
4     LineStyle: '--'
5     LineWidth: 0.5000
6     Marker: 'none'
7     MarkerSize: 6
8     MarkerFaceColor: 'none'
9     XData: [0 1 2 3]
10    YData: [0 1 5 7]
11    ZData: [1x0 double]

```

For example, one could change what each data point looks like and the width of each line like this:

```

1 a(1).Marker = '*';
2 a(2).LineWidth = 5;

```

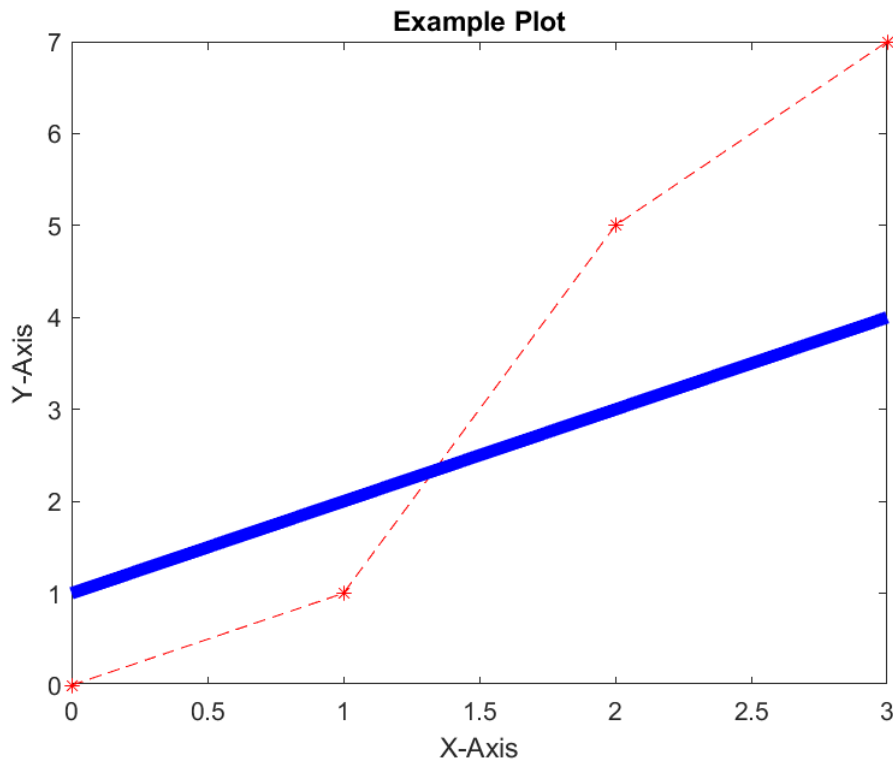


Figure 8.4

Grid can be displayed with **grid on** and the limit for the x and y-axis can be set with the function **axis()**. Axis() takes in an array in the following format: **axis([x-min x-max y-min y-max])**

```

1 grid on
2 axis([-5,5,-5,5]);

```

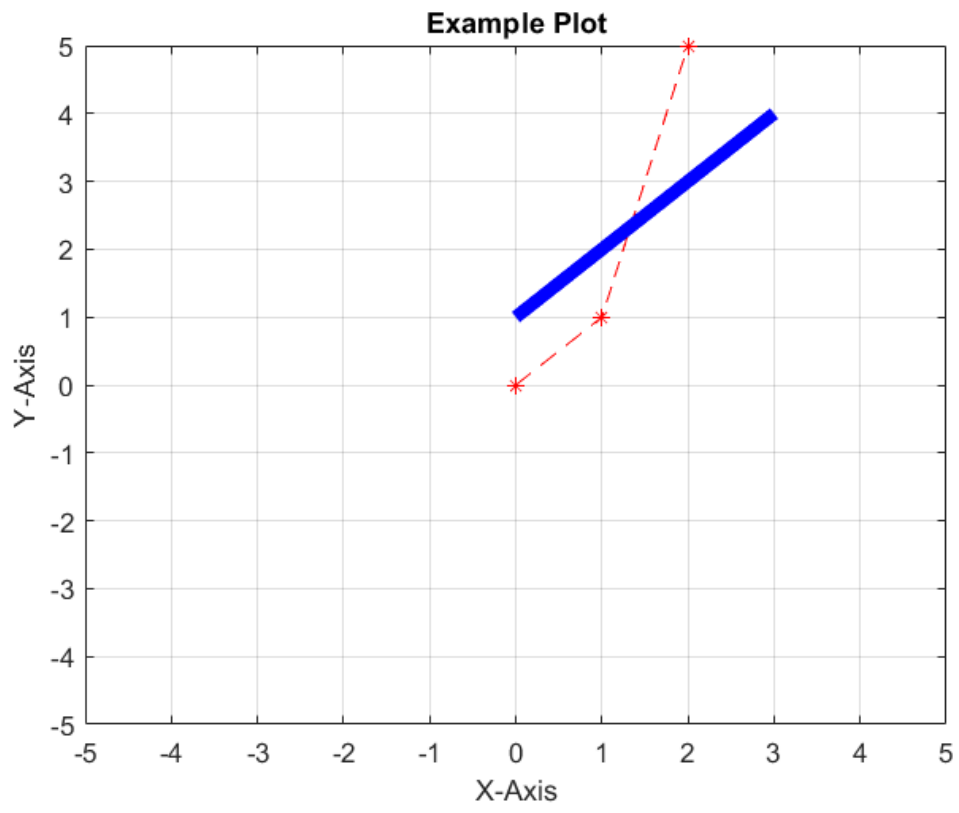


Figure 8.5



## Section 2: Rectangular

**R**ectangular equations can be displayed by giving the **x** parameter of `plot()` a range of x-values and the **y** parameter an equation in terms of x. Before we learn to plot this, we need to learn about the difference between vector and elementwise operations in MATLAB.

Vector operations are used when working with multiple matrices/vectors while elementwise operators deal with scalars. Elementwise operators follow the same syntax as the vector ones, except that they have a dot before them, so the elementwise equivalent for multiplication would be `.*`.

This comes into play when we for example, attempt to square the x parameter in a quadratic equation. The regular exponent sign would cause an error because MATLAB would attempt to multiply the non-square matrix x by itself thus creating a dimension error.

By using `.^` we tell MATLAB to multiply each element of x by itself. The below figure shows MATLAB plotting a quadratic equation.

```
1 x = -5:0.01:5;  
2 y = x.^2 + 5.*x + 1;  
3 a = plot(x,y, 'r--');
```

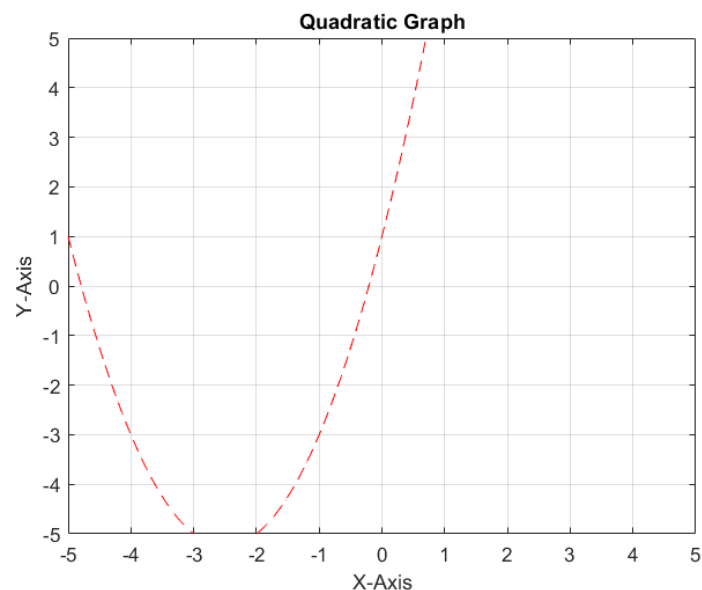


Figure 8.4

MATLAB also supports the plotting of other mathematical functions such as sine and cosine as well.

```
1 x = -10:0.01:10;  
2 y = x.*(cos(2*x) + sin(2*x));  
3 a = plot(x,y, 'r--');
```

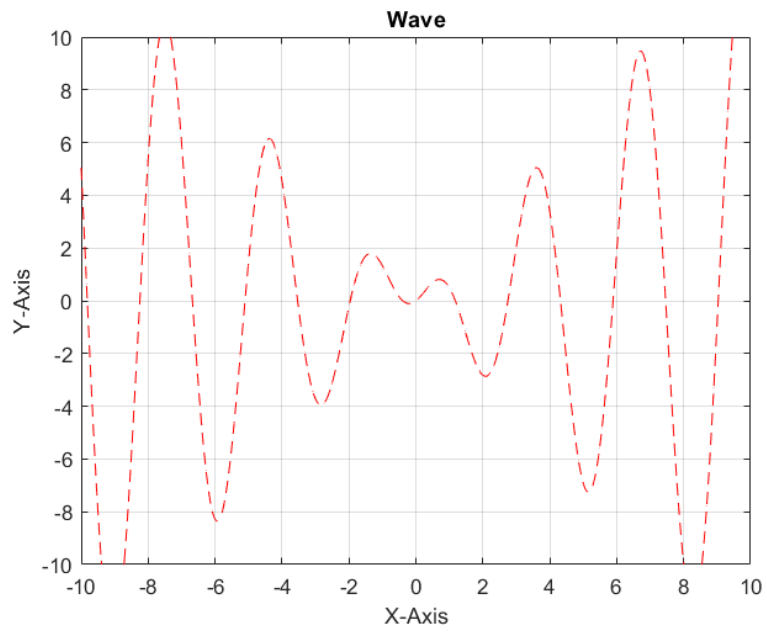


Figure 8.5

## Section 3: Parametric

Parametric equations can be handled similarly to how rectangular ones are except now the independent variable for both the functions  $x$  and  $y$  is a new variable named  $t$ . Figure 8.5 utilizes parametric equations to plot a circle as an example.

```
1 t = -10:0.01:10;  
2 x = 5.*cos(t);  
3 y = 5.*sin(t);  
4 a = plot(x,y, 'r--');
```

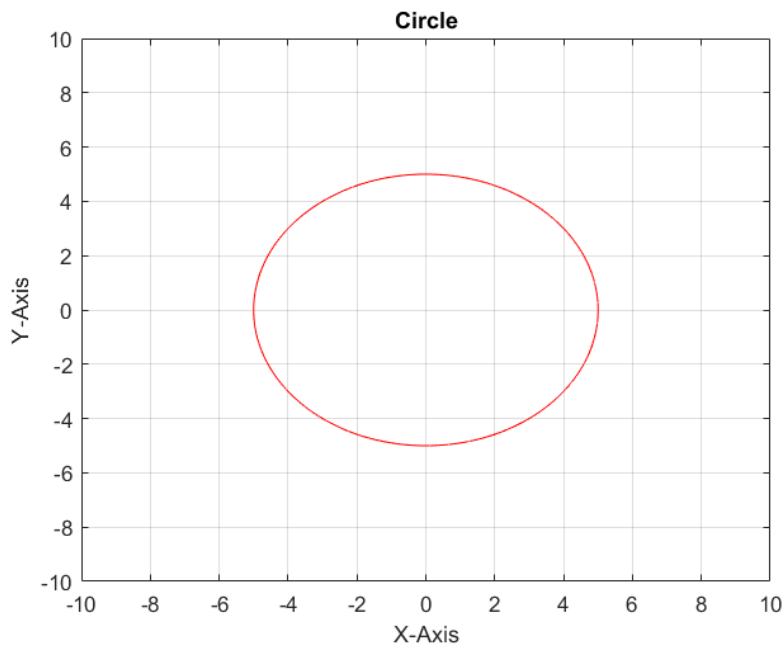


Figure 8.5

Even though the plot is a circle, MATLAB appears to display it as an ellipse. This occurs because the data units of the two axis is not equal and can be fixed by adding the line **axis equal** or **axis square** at the very end of the program.

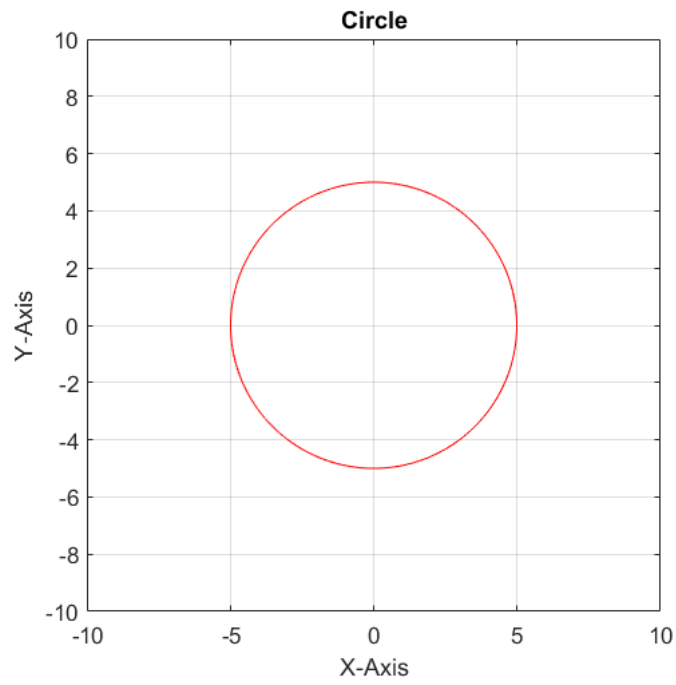


Figure 8.6

## Section 4: Polar

While rectangular and parametric could be plotted using the same function, polar equations will require a new one named **polarplot()**. Polarplot accepts the same parameters the plot() function does, however, it utilizes a polar coordinate system when displaying rather than a rectangular one. This means that the coordinates are no longer x and y-based but instead use angle and radius from the origin. Therefore, if we want to draw a circle, we simply go from zero to 360 degrees with a constant radius:

```
1 theta = 0:0.01:2*pi;  
2 rad = 2;  
3 polarplot(theta,rad)
```

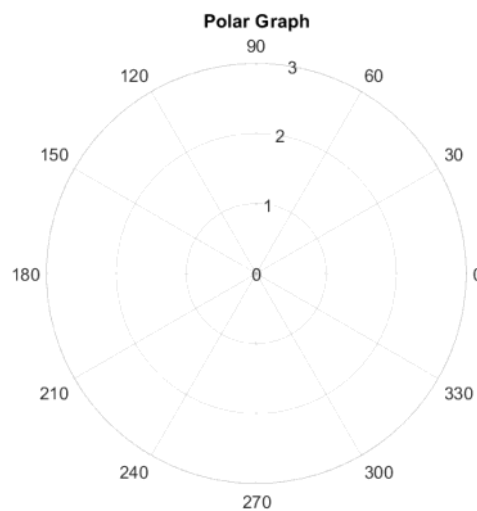


Figure 8.6

**Note** that polarplot() accepts radians and not degrees in theta. However, the plot in figure 8.6 does not display a circle with a radius of 2 because while **theta** is an array, **rad** is only a number so only the very first point (theta = 0 and rad = 2) would get plotted. To turn the radius into an array of the same size as theta we simply multiply the radius by an array of ones with the length of theta.

```
1 theta = 0:0.01:2*pi;  
2 rad = 2 .* ones(length(theta),1);  
3 polarplot(theta,rad)
```

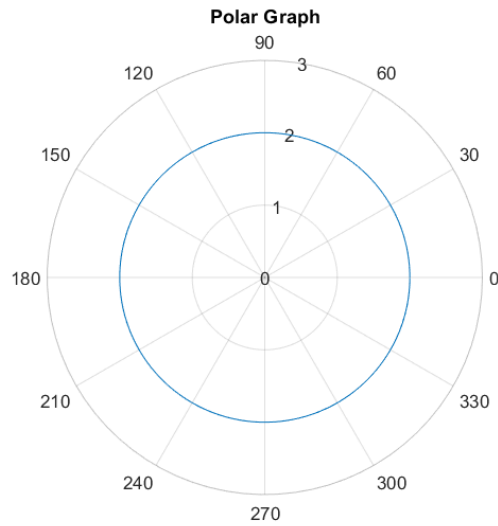


Figure 8.7

The `axis()` function works for polar graphs as well but its parameters now are `axis([theta0 theta1 radius0 radius1])`. For example, the following shows a polar graph with a minimum degree of zero and a maximum of 270 degrees and a minimum radius of -5 to a maximum of 5:

```
1 theta = 0:0.2:500;
2 rad = 4.*cos(60.*theta);
3 polarplot(theta,rad);
4 axis([0,270,-5,5]);
```

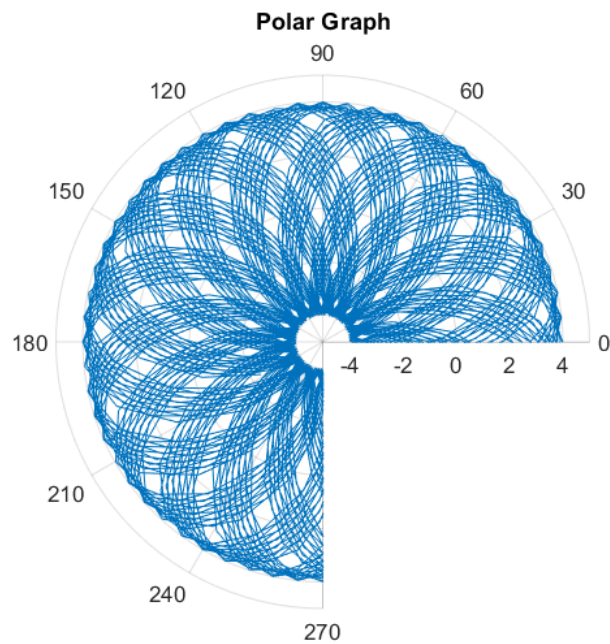


Figure 8.8

**Note** that unlike `polarplot()`, `axis()` uses degrees when referring to theta.

## Section 5: Cylindrical

Cylindrical coordinates are a generalization of two-dimensional polar coordinates to three dimensions by superposing a height ( $z$ ) axis. In a cylindrical coordinate system either  $r$  or  $\rho$  is used to refer to the radial coordinate and either  $\theta$  or  $\varphi$  to refer to the azimuthal coordinates. For instance  $(r, \theta, z)$  or  $(\rho, \varphi, z)$  are used as cylindrical coordinates.

The most simple way to plot a cylinder is typing the command `cylinder`. You can change the radius of the cylinder by using the syntax `cylinder(r)`, where  $r$  represents the radius. For example, to create a cylinder with radius  $2\pi$ , we do:

```
1 cylinder(2*pi)
```

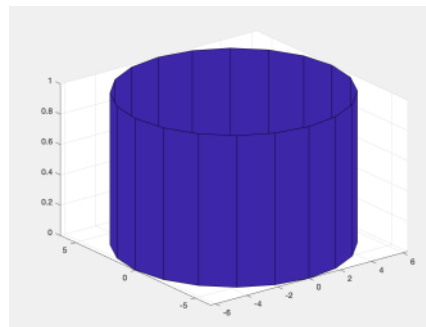


Figure 8.9

To generate other figures such as a cone in MATLAB, you will need to work with cylindrical coordinates, although this is more complicated than just plotting a cylinder.

```
1 [r,theta] = meshgrid(linspace(0,0.7,30),linspace(0,2*pi,30));  
2 z = r,meshgrid(linspace(0.4,0.4,30));  
3 surf(r.*cos(theta),r.*sin(theta),z);
```

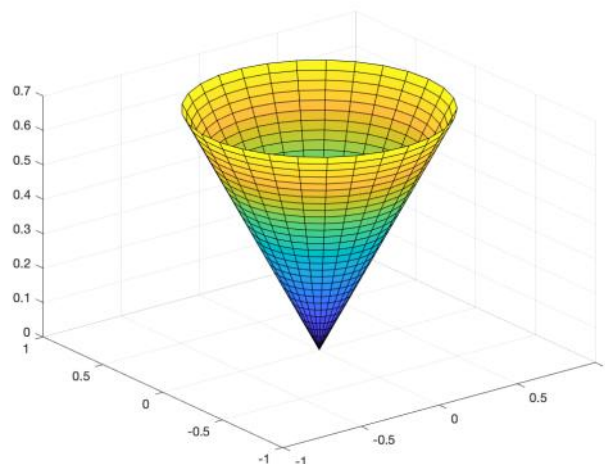


Figure 8.10

First, we define what  $r$  and  $\theta$  are, in this example  $r$  is from 0 to 30 with an index of 0.7, and  $\theta$  is also from 0 to 30 with an index of  $2\pi$ . Next we define a new variable called  $z$ , which

based on radius  $r$  also increases from 0.4 to 30 with intervals of 0.4 (starts from small radius and increases as  $z$  increases. Finally, we can plot using surf, we multiply all values of radius by  $\cos(\theta)$  for  $x$  and by  $\sin(\theta)$  for  $y$ , and used  $z$  for height axis. The syntax to using surf is; **surf(X, Y, Z)**.

In other cases, we sometimes have to convert from cylindrical coordinates to cartesian. For example:

```
1 theta = linspace(0,2*pi); z = linspace(0,10);
2 [TH,Z] = meshgrid(theta,z);
3 R = sin(Z)+1+5*sin(TH); %// For cylinder it would be simply R = ones(size(Z));
4 [x,y,z] = pol2cart(TH,R,Z);
5 mesh(x,y,z); axis equal
```

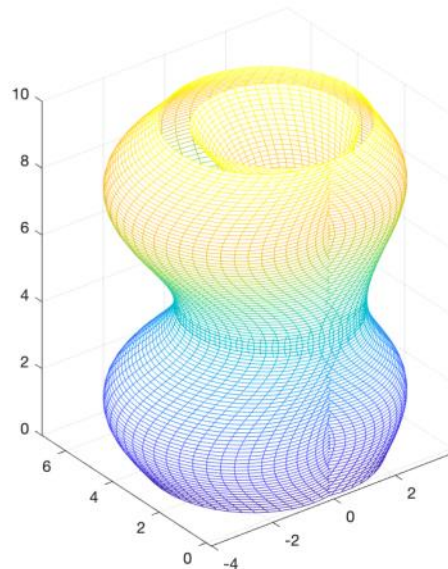


Figure 8.11

1. We created a vector for  $\theta$  and  $z$ .
2. Created a meshgrid from  $\theta$  and  $z$ .
3. Wrote a function  $R(\theta, z)$ .
4. Converted from cylindrical to cartesian.
5. Plotted the result using mesh **Note** that we could also have used surf to plot.



## Section 6: Spherical

**S**pherical coordinates are a system of curvilinear coordinates that are natural for describing positions on a sphere or spheroid. Define theta to be the azimuthal angle in the  $x y$ -plane from the  $x$ -axis with  $(0 \leq \theta < 2\pi)$ . phi to be the polar angle from the positive  $z$ -axis with  $0 \leq \phi < \pi$ , and  $r$  to be the radius from a point to the origin.

To plot a sphere we first need to define  $x$ ,  $y$ , and  $z$  as coordinates of a sphere. For example, to plot a sphere with its origin at  $(0, 1, -3)$ , we can do:

```
1 [x,y,z] = sphere;  
2 surf(x,y+1,z-3)
```

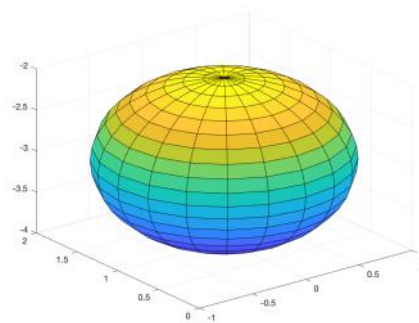


Figure 8.12

To plot more complicated figures using the spherical coordinate system, we approach very similarly to cylindrical coordinate system. Lets look at an example below:

```
1 R=10;  
2 Phi=linspace(-pi/3,pi/3);  
3 Theta=linspace(0,pi);  
4 [Phi,Theta]=meshgrid(Phi,Theta);  
5 [X,Y,Z]=sph2cart(Theta,Phi,R);  
6 surf(X,Y,Z);
```

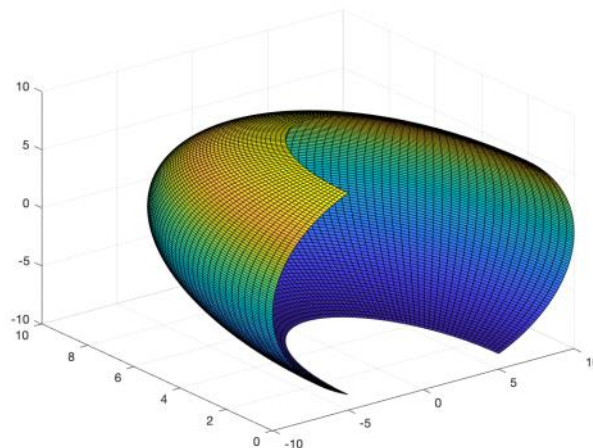
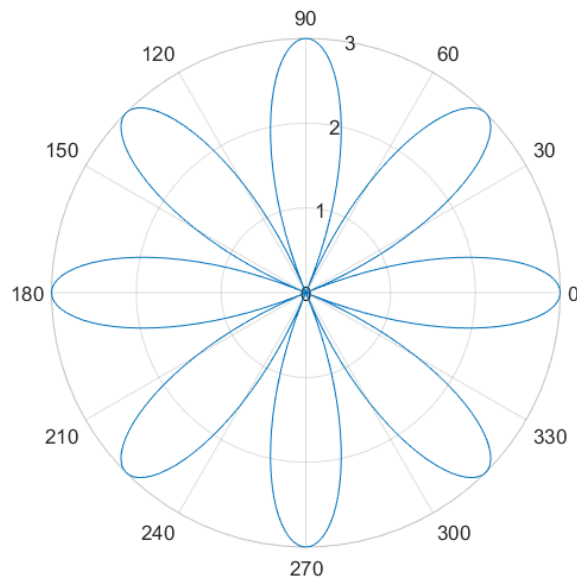


Figure 8.13

1. We set the radius **R** to 10.
2. We defined **Phi** to be from  $-\frac{\pi}{3}$  to  $\frac{\pi}{3}$ .
3. We defined **Theta** to be from 0 to  $\pi$ .
4. Created a meshgrid from **Phi** and **Theta**.
5. Converted from spherical coordinate system to cartesian.
6. Plotted the result using surf.

## Student Exercises

1. Create a function that accepts a rectangular equation and other MATLAB plot properties then plots the given curve.
2. Create a function that accepts a parametric system of equations and other MATLAB plot properties then plots the given curve.
3. Define a function that accepts a polar equation and other MATLAB plot properties then plots the given curve.
4. Write a program that draws the diagram shown below:



5. Evaluate the following integral by first converting to an integral in cylindrical coordinates.

$$\int_0^{\sqrt{5}} \int_{-\sqrt{5-x^2}}^0 \int_{x^2+y^2-11}^{9-3x^2-3y^2} 2x - 3y \, dz \, dy \, dx$$

6. Plot the following:

$$z = \sqrt{x^2 + y^2} \text{ on } \{(x, y): -3 \leq x, y \leq 3\}.$$

7. Write a program that will convert between Cartesian, Cylindrical, and Spherical coordinate systems. Hint: Use different functions for each conversion. You may use the following conversion table.

$$\text{cart} \leftrightarrow \text{cyl} \quad \begin{cases} x = r \cos \theta, \\ y = r \sin \theta, \end{cases} \quad \begin{cases} r = \sqrt{x^2 + y^2}, \\ \theta = \arctan \frac{y}{x}, \end{cases} \quad \begin{cases} \sin \theta = \frac{y}{\sqrt{x^2 + y^2}}, \\ \cos \theta = \frac{x}{\sqrt{x^2 + y^2}}. \end{cases}$$

$$\text{cyl} \leftrightarrow \text{sph} \quad \begin{cases} r = \rho \sin \phi, \\ z = \rho \cos \phi, \end{cases} \quad \begin{cases} \rho = \sqrt{r^2 + z^2}, \\ \phi = \arctan \frac{r}{z}, \end{cases} \quad \begin{cases} \sin \phi = \frac{r}{\sqrt{r^2 + z^2}}, \\ \cos \phi = \frac{z}{\sqrt{r^2 + z^2}}. \end{cases}$$

$$\text{cart} \leftrightarrow \text{sph} \quad \begin{cases} x = \rho \cos \theta \sin \phi, \\ y = \rho \sin \theta \sin \phi, \\ z = \rho \cos \phi, \end{cases} \quad \begin{cases} \rho = \sqrt{x^2 + y^2 + z^2}, \\ \theta = \arctan \frac{y}{x}, \\ \phi = \arctan \frac{\sqrt{x^2 + y^2}}{z} \\ = \arccos \frac{z}{\sqrt{x^2 + y^2 + z^2}}. \end{cases}$$

# CHAPTER 9

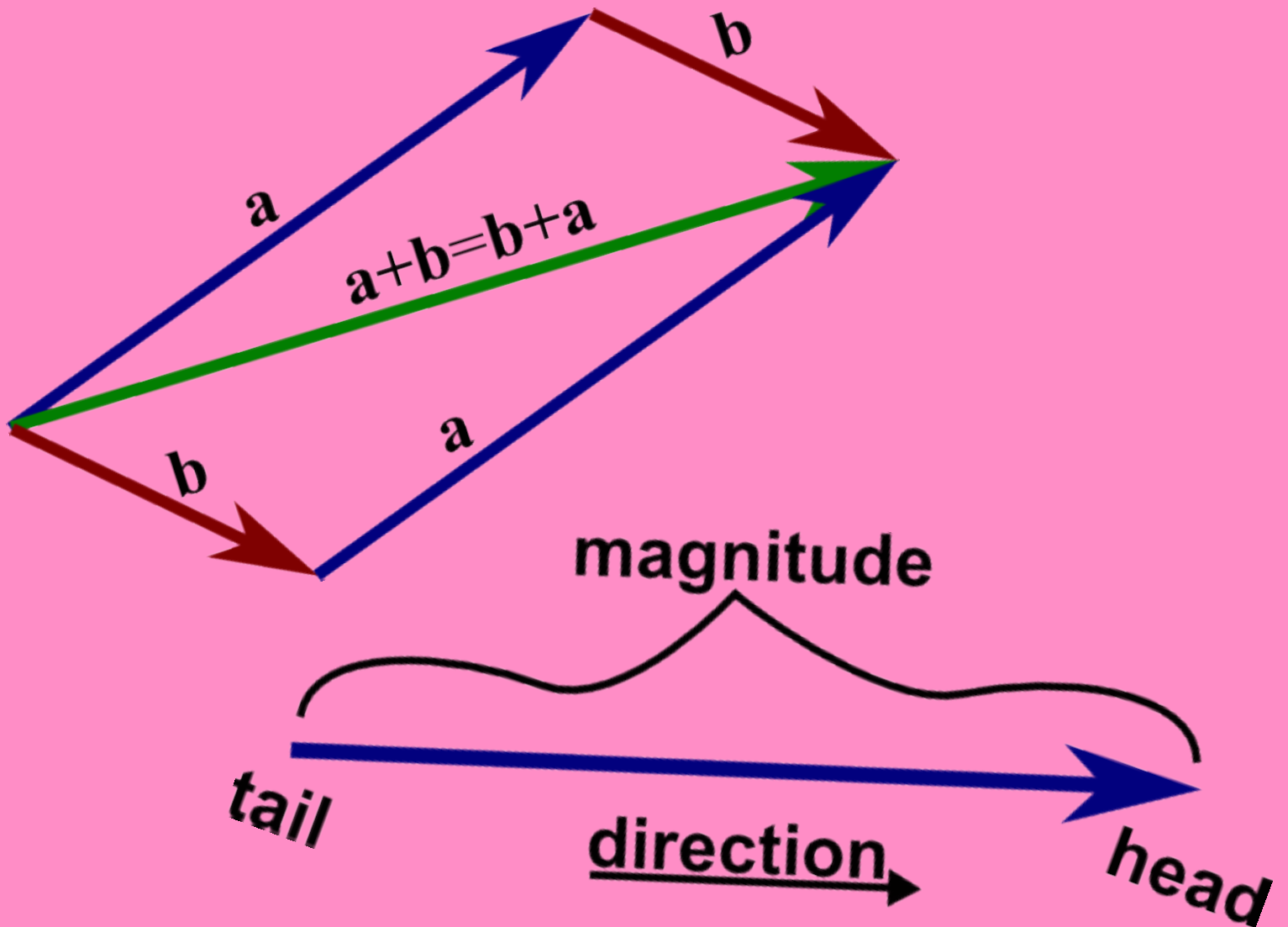
## Two-Dimensional Vector Operations:

Section 1: Vectors and Basic Vector Operations

Section 2: Plotting

Section 3: Other Operations

**Student Exercises**



## Section 1: Vectors and Basic Vector Operations

**M**ATLAB allows us to enter vectors very easily. We can do a row vector whereby columns are separated by spaces or commas and rows are separated by semicolons. Try the following for a vertical vector. The semicolons put the numbers on different rows.

```
1 v=[1;2;3]
```

```
v =  
    1  
    2  
    3
```

And for a horizontal vector, the commas put the numbers on different columns. **Note** that you could also just use spaces instead of commas.

```
1 v=[1,2,3]
```

```
v =  
    1    2    3
```

Similarly we can create matrices easily, as rows are separated by semicolons while columns in the vector can be separated either by commas or spaces. For example the following gives a 2 x 3 matrix:

```
1 A=[1 2 3;4 5 6]
```

```
A =  
    1    2    3  
    4    5    6
```

We can also create vectors with variables in them provided we declare the variables symbolically first. For example:

```
1 syms t;  
2 v=[t;t^2;t-3]
```

```
v =  
    t  
    t^2  
    t - 3
```

MATLAB deals with vectors very intuitively. For example we can add and subtract vectors using + and -

```
1 A=[3;1;0];  
2 B=[-2;1;3];  
3 A+B
```

```
ans =  
1  
2  
3
```

and we can do scalar multiplication simply by multiplying by a constant.

```
1 A-3.1*B
```

```
ans =  
9.2000  
-2.1000  
-9.3000
```

Suppose we wish to multiply each entry in **a** by the corresponding entry in **b**, and **a** and **b** are:

```
a =  
1 2 7  
b =  
4 3 0
```

We can use the special MATLAB notation `.*`, so `a.*b` would find `[1*4 2*3 7*0]`, or we can use `./` and `.^` which will divide or apply powers as follows:

```
1 a=[1 2 7]  
2 b=[4 3 0]  
3 a.*b  
4 a./b  
5 a.^b
```

```
ans =  
4 6 0  
ans =  
0.2500 0.6667 Inf  
ans =  
1 8 1
```

## Section 2: Plotting

**P**lotting is a simple function in MATLAB, in previous chapters we discussed plotting a data on a scatter plot, and plotting graphs in different coordinate systems. In this chapter we will discuss plotting vectors and drawing the contour.

A quiver plot displays velocity vectors as arrows with components  $(u,v)$  at the points  $(x,y)$ . **quiver** $(x,y,u,v)$  plots vectors as arrows at the coordinates specified in each corresponding pair of elements in  $x$  and  $y$ . The matrices  $x$ ,  $y$ ,  $u$ , and  $v$  must all be the same size and contain corresponding position and velocity components. **quiver** $(u,v)$  draws vectors specified by  $u$  and  $v$  at equally spaced points in the  $x$ - $y$  plane. **quiver** $(...,scale)$  automatically scales the arrows to fit within the grid and then stretches them by the factor  $scale$ .  $scale = 2$  doubles their relative length, and  $scale = 0.5$  halves the length. Use  $scale = 0$  to plot the velocity vectors without automatic scaling. **Note** that by default, the arrows are scaled to not overlap, but you can scale them to be longer or shorter if you want.

MATLAB® expands  $x$  and  $y$  if they are not matrices. This expansion is equivalent to calling `meshgrid` to generate matrices from vectors:

```
1 [x,y] = meshgrid(x,y);  
2 quiver(x,y,u,v)
```

For example to create a vector plot with the two components  $u = \cos(x) \cdot y$  and  $v = \sin(x) \cdot y$  we do the following:

```
1 [x,y] = meshgrid(0:0.2:2,0:0.2:2);  
2 u = cos(x).*y;  
3 v = sin(x).*y;  
4  
5 figure  
6 quiver(x,y,u,v)
```

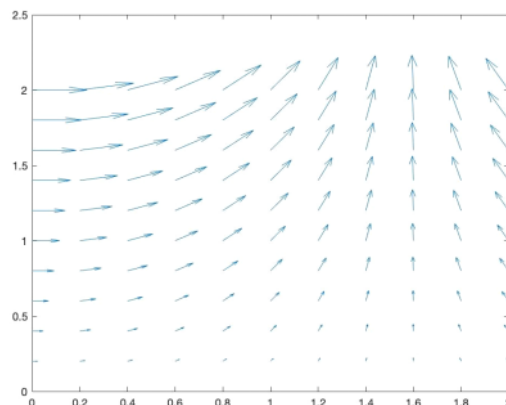


Figure 9.1



## Section 3: Other Operations

**F**or two vectors/matrices to have a valid dot product they must follow the format **MxN dot NxK** where the number of rows in the first matrix matches the number of columns in the second matrix.

Here is an example of a dot product:

$$\begin{pmatrix} 1 & 7 \\ 2 & 4 \end{pmatrix} \begin{pmatrix} 5 & 3 \\ 8 & 9 \end{pmatrix} = \begin{pmatrix} 5 + 56 & 3 + 63 \\ 10 + 32 & 6 + 36 \end{pmatrix} = \begin{pmatrix} 61 & 66 \\ 42 & 42 \end{pmatrix}$$

Taking the dot product in MATLAB is as simple as using the `*` operator with two matrices. Below shows the previous example done in MATLAB.

```
1 mat1 = [1 7; 2 4];
2 mat2 = [5 3; 8 9];
3 disp(mat1*mat2)
```

Unlike the dot product, the cross product requires the two vectors to be three dimensional.

Here is an example of a cross product:

$$(1 \ 2 \ 3) \times (5 \ 6 \ 7) = (2 * 7 - 3 * 6 \ 3 * 5 - 1 * 7 \ 1 * 6 - 2 * 5) = (-4 \ 8 \ -4)$$

The cross product of two vectors can be taken using the command **cross()**, where the parameters are the two vectors. The example above is repeated below in MATLAB:

```
1 mat1 = [1 2 3];
2 mat2 = [5 6 7];
3 disp(cross(mat1,mat2))
```

To find the magnitude of a vector, we can use the **norm()** command. The magnitude is calculated by adding up the square of each element in a vector (regardless of dimension) and taking the square root of the whole sum.

```
1 vec = [4,2;1,3];
2 norm(vec)
```

MATLAB outputs  $\sqrt{(4^2+2^2 + 1^2 + 3^2)} = 5.1167$

The **transpose()** function allows us to transpose vectors and matrices.

```
1 vec = [4,2;1,3]
2 transpose(vec)
```

$$\begin{bmatrix} 4 & 2 \\ 1 & 3 \end{bmatrix} \rightarrow \begin{bmatrix} 4 & 1 \\ 2 & 3 \end{bmatrix}$$

MATLAB also allows users to take the determinant of a matrix with the command **det()**. Remember that the determinant only applies to square matrices.

```
1 vec = [4,2;1,3;]
2 det(vec)
```

The determinant of **vec** gives 10.

Finally, the reduced row echelon form of a matrix can be calculated with the function **rref()**.

```
1 vec = [4,2,1;1,3,1]
2 rref(vec)
```

The result would be the following matrix:

$$\begin{bmatrix} 1 & 0 & 0.1 \\ 0 & 1 & 0.3 \end{bmatrix}$$

## Student Exercises

1. Write down a series of MATLAB commands which will define  $t$  symbolically and then find the product of the matrices  $\begin{bmatrix} 1 & t \\ -t & 2 \end{bmatrix}$  and  $\begin{bmatrix} 1/t^2 & t \\ 2 & -1 \end{bmatrix}$ .
2. Create a function that finds the magnitude of a given vector (do not use **norm()**).
3. Construct a function that accepts a system of equations and uses **rref()** to solve for the variables.
4. Given a matrix, create a function that outputs its identity.
5. Given the three vertices of a triangle, write a function that outputs its area.
6. A basic Hill cipher involves first choosing a 2x2 matrix of random elements from one to 26.

$$A = \begin{bmatrix} 5 & 23 \\ 6 & 15 \end{bmatrix}$$

To encrypt a message, we must separate the text into pairs of letters.

HELLO -> HE LL OO

We then convert the letters into numbers (A=1, B=2, ... Z=26).

HE LL OO -> 8,5 12,12 15,15

Now we take the dot product of **A** and each pair of numbers.

$$\begin{bmatrix} 5 & 23 \\ 6 & 15 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 5 \end{bmatrix}$$

The resulting numbers is the encoded message. Note that numbers larger than 26 are corrected by finding their remainder when divided by 26.

HE -> 8,5 -> 155, 123 => 25, 19

Create a function that accepts a 2x2 key and message and outputs the message encrypted with the hill cipher.

7. Now write a function that accepts a 2x2 key and a hill cipher encrypted message and returns plaintext. Hint: You will need to use MATLAB's **inv(X)** function.

# CHAPTER 10

## Calculus In MATLAB:

Section 1: Limits

Section 2: Derivatives

Section 3: Integrals

Section 4: Infinite Series

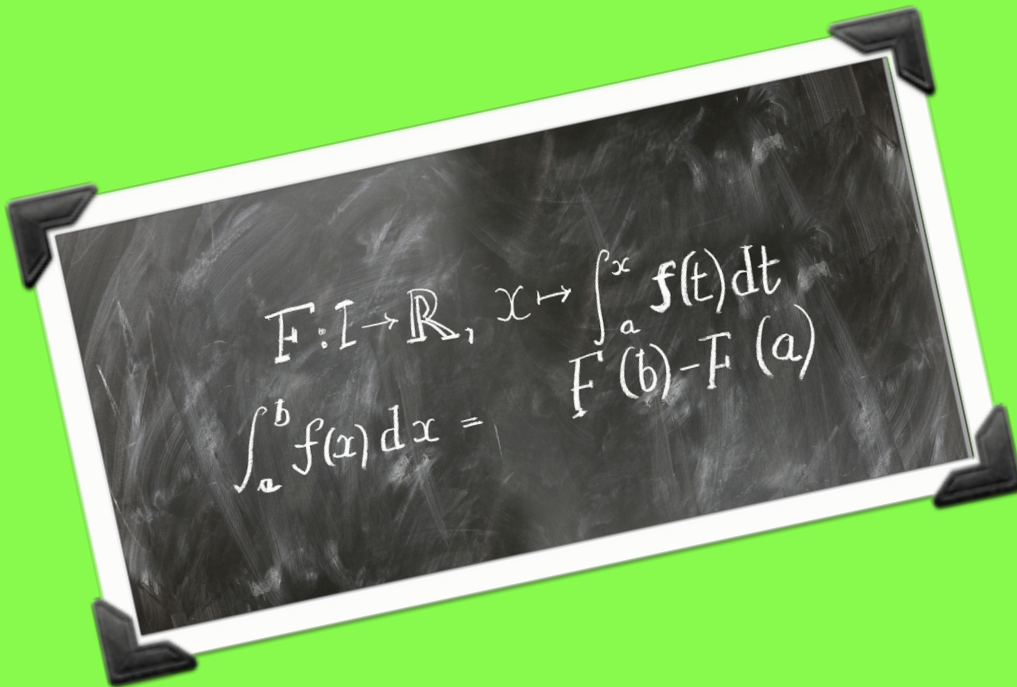
**Student Exercises**

$$\lim_{x \rightarrow a} f(x) = L$$

function

As you approach  $a$  along the x-axis

What is the y-value getting closer to?



## Section 1: Limits

The fundamental idea in calculus is to make calculations on functions as a variable “gets close to” or approaches a certain value. Recall that the definition of the derivative is given by a limit, assuming a limit exists. Fortunately MATLAB can solve limits using a simple command.

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

By the default case, **limit(f)** is the same as **limit(f,x,0)**. Explore the options for the limit command in this table, where f is a function of the symbolic object x. The table below will better explain how to work with limits in MATLAB:

Mathematical Operation	MATLAB Command
$\lim_{x \rightarrow 0} f(x)$	limit(f)
$\lim_{x \rightarrow a} f(x)$	limit(f, x, a) or limit(f, a)
$\lim_{x \rightarrow a^-} f(x)$	limit(f, x, a, 'left')
$\lim_{x \rightarrow a^+} f(x)$	limit(f, x, a, 'right')

Table 10.1

For example, to calculate the limit as x approaches 0 from the left,

$$\lim_{x \rightarrow 0^-} \frac{x}{|x|}$$

We do:

```
1 syms x
2 limit(x/abs(x), x, 0, 'left')
```

And MATLAB will return **-1**

And to calculate the limit as x approaches 0 from the right,

$$\lim_{x \rightarrow 0^+} \frac{x}{|x|} = 1$$

We do:

```
1 syms x
2 limit(x/abs(x), x, 0, 'right')
```

And MATLAB will return **1**

And Finally to calculate the two sided limit as x approaches 0, we can do:

```
1 syms x
2 limit(x/abs(x), x, 0)
```

However, we know that since the limit from the right and the left differ, a two-sided limit does not exist. So, MATLAB will return **NaN**

## Section 2: Derivatives

**M**ATLAB uses a simple command, *diff* to take the derivative of a function, we can also take the second derivative of a function using the same command. Also, MATLAB allows us to take derivatives of functions with several variables. To demonstrate how to take the derivative of an expression in MATLAB let's look at the following example:

```
1 syms x % Create a symbolic expression
2 f = sin(5*x); % Create a function
3 diff(f) % differentiate the function
```

MATLAB will differentiate  $f$  with respect to  $x$  and will return  $5*\cos(5*x)$

To take the second derivative of the function, we can approach two different methods. We can take the derivative twice as follows:

```
1 diff(diff(f))
```

Or we can use a more appropriate method and use the function like below:

```
1 diff(f,2)
```

To differentiate an expression that contains more than one symbolic variable, specify the variable that you want to differentiate with respect to. The *diff* command then calculates the partial derivative of the expression with respect to that variable. For example:

```
1 syms s t
2 f = sin(s*t);
3 diff(f,t)
```

Will differentiate  $f$  with respect to  $t$ , and will return  $s*\cos(s*t)$

To differentiate  $f$  with respect to the variable  $s$ , we do:

```
1 diff(f,s)
```

This returns  $t*\cos(s*t)$

**Note** that if you do not specify a variable to differentiate with respect to, MATLAB chooses a default variable. Basically, the default variable is the letter closest to  $x$  in the alphabet.

You can also take the second derivative of an expression that contains several variables. For example to take the second derivative of the previous function with respect to  $t$  we do:

```
1 diff(f, t, 2)
```

This returns  $-s^2*\sin(s*t)$

To better illustrate how to use the *diff* command, refer to the following table:

Mathematical Operator	MATLAB Command
$\frac{df}{dx}$	diff(f) or diff(f, x)
$\frac{df}{da}$	diff(f, a)
$\frac{d^2f}{db^2}$	diff(f, b, 2)

Table 10.2

## Section 3: Integrals

Integration has been made very simple and straight forward in MATLAB, much like differentiation, there is a command **int** for integration. For example, Suppose we need to integrate the function  $\frac{7}{(1+x^2)}$ . We can do:

```
1 int(7/(1 + x^2))
```

And MATLAB returns **7\*atan(x)**

**Note** that **atan** is the same as arctan which is the same as inverse tangent.

That is how indefinite integration works in MATLAB. And to take a definite integral lets look at another example. For example, to compute the area under the curve  $f(x) = x^3 + \ln x$  over the interval where  $x$  is between 5 and 10. Just add those end-points to the **int** command as two additional parameters as follows:

```
1 int(x^3 + log(x), 5, 10)
```

And MATLAB will return **5\*log(20) + 9355/4**

The table below will better explain and illustrate how to use integration in MATLAB:

Mathematical Operator	MATLAB Command
$\int x^n dx$	int(x^n) int(x^n, x)
$\int_0^{\pi/2} \sin(2x) dx$	int(sin(2*x), 0, pi/2) int(sin(2*x), x, 0, pi/2)
$g = \cos(at + b)$ $\int g(t) dt$	g = cos(a*t + b) int(g) int(g, t)

Table 10.3

MATLAB has a specific function for taking double and triple integrals as well called **integral2()** and **integral3()** respectively.

```
1 i2 = integral2(fun, xmin, xmax, ymin, ymax)
2 i3 = integral3(fun, xmin, xmax, ymin, ymax, zmin, zmax)
```

As you would expect, the new commands are only different in the number of parameters they need to setup bounds for each variable in the integral, and the expression the user wishes to integrate must be given in the form of an **anonymous function** now. Anonymous function follow this format:

```
1 func = @( [variables] ) [expression];
```



Thus, an example of a triple integral in MATLAB would be:

```
1 func = @(x,y,z) x + y + z;  
2 A = integral3(func, 0, 5, 0, 5, 0, 5)
```



Figure 10.1

## Section 4: Infinite Series

MATLAB allows us to create and evaluate infinite series with the `symsum()` function. This function contains two different forms:

```
1 F = symsum(f, k, a, b)
2 F = symsum(f, k)
```

The first version evaluates the sum of expression `f` which uses variable `k` from bounds `a` to `b`. The second version does not require bounds and instead evaluates the indefinite sum. `Inf` can be used to set the bounds to positive or negative infinity. For example, let's say we want to evaluate this infinite series, which we know will converge to -1:

$$\sum_{k=0}^{\infty} \frac{(-1)^k \left(\frac{\pi}{2}\right)^{2k}}{(2k)!} = \cos(\pi) = -1$$

Translated into MATLAB, the series would be:

```
1 syms k
2 x = pi;
3 symsum((-1)^k *
  (x)^(2*k))/factorial(2*k), k, 0, Inf)
```

Evaluating an infinite series that does not converge results in infinity or negative infinity.

```
1 syms n
2 A = symsum(1/n, n, 1, Inf);
3 disp(A)
```



Figure 10.2

Below is an example of an indefinite sum and the result in MATLAB:

```
1 syms n
2 A = symsum(1/(n^2), n);
3 disp(A)
```



Figure 10.3

## Student Exercises

1. Have MATLAB integrate the function  $f(x) = \sqrt{x^2 + 3}$ . Then Have MATLAB compute the definite integral of that same function over the interval  $[2, 4]$ .
2. Use MATLAB to solve the separable differential equation  $\frac{dy}{dx} = y + y\sqrt{x}$ .
3. Write a function that accepts the expression and bounds of an infinite series as input and tells the user if the series converges or diverges. If it converges, it tells the user at what number the series converges.
4. When given the weight of an object and the 1-dimensional distance it has traveled, create a function that can output the amount of work needed.
5. Create a method that outputs a mathematical function when given a point the function passes through  $(x, y)$  and the function's first derivative (following the format  $y' = Ax^n + Bx^{n-1} + \dots + C$ ).
6. Construct a method where given an object's velocity function, the method outputs an equation for the object's acceleration as well as plots both velocity and acceleration on the same graph.



**This book is designed to give students an introduction to MATLAB in the following topics:**

**Data Types and Basic Evaluations**

**Algebraic Computations**

**Conditions and Loops**

**Arrays**

**Classes and Functions**

**Data and Statistics**

**Coordinate Systems**

**Two-Dimensional Vector Operations**

**Calculus In MATLAB**

**For additional resources and documentations visit:**

**[www.mathworks.com/help/index.html](http://www.mathworks.com/help/index.html)**

**Refer to the following to purchase a MATLAB License:**

**[www.mathworks.com/store](http://www.mathworks.com/store)**

**If you already have a MATLAB License, download and activate MATLAB here:**

**[www.mathworks.com/downloads](http://www.mathworks.com/downloads)**