MATLAB Primer

Department of Mathematics and Engineering
Queen's University

August 21, 2021

Contents

1 Introduction

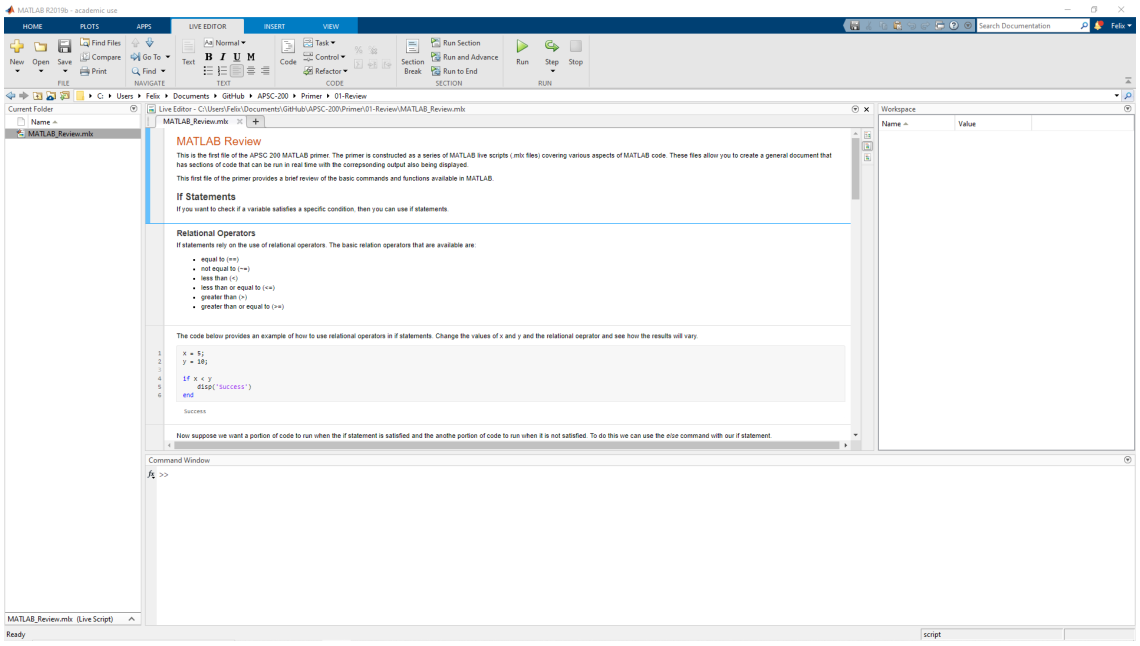
The purpose of this primer is to provide a brief overview of MATLAB along with descriptions and examples of MATLAB commands that you may find yourself using while completing the MATLAB coding portion of the APSC 200 P2 project. This is not a comprehensive summary, but an introduction to commonly used MATLAB syntax. For more detailed descriptions of any command you wish to use in MATLAB, you can find documentation by searching Google or by using MATLAB's built-in Search Documentation bar.

2 Introduction to MATLAB

2.1 MATLAB Layout

The MATLAB application has four key layout components: The Command Window, the Workspace, the Editor (or Live Editor), and the Current Folder. If any of these components do not appear in your MATLAB application, you can go to the Home tab, and in the Environment section, click layout. Here you can toggle the layout components on or off.

In Figure 1 below you can see that the Live Editor takes up the majority of the space in the center, the Workspace is on the right, the Command Window is on the bottom, and the Current Folder is on the left. However, the MATLAB layout can be customized and does not require these placements. To customize the interface to match your personal preferences, you may drag and drop components to different sections of the interface.



MATLAB Review

This is the first file of the APSC 200 MATLAB primer. The primer is constructed as a series of MATLAB live scripts (.mlx files) covering various aspects of MATLAB code. These files allow you to create a general document that has sections of code that can be run in real time with the correpsonding output also being displayed.

This first file of the primer provides a brief review of the basic commands and functions available in MATLAB.

If Statements

If you want to check if a variable satisfies a specific condition, then you can use if statements.

Relational Operators

If statements rely on the use of relational operators. The basic relation operators that are available are:

- equal to (==)
- not equal to (~=)
- less than (<)
- less than or equal to (<=)
- greater than (>)
- greater than or equal to (>=)

The code below provides an example of how to use relational operators in if statements. Change the values of x and y and the relational oeprator and see how the results will vary.

```
x = 5;
y = 10;

if x < y
    disp('Success')
end
```

```
Success
```

Now suppose we want a portion of code to run when the if statement is satisfied and the anothe portion of code to run when it is not satisfied. To do this we can use the *else* command with our if statement.

```
>>
```

Figure 1: MATLAB Layout

2.1.1 Command Window

MATLAB's Command Window enables you to write single-line input statements and view corresponding output statements. Input statements written in the Command Window will result in an output statement being displayed below them; however, output statements can be suppressed through the use of a semi-colon at the end of a command.

Example 1:

input:

```
>> 1 + 1 % this command will appear as an output
```

output:

$$\text{ans} = 2$$

Example 2:

input:

```
>> 2 + 2; % this command will not appear as an output
```

output:

There are a number of functions you can use to print variables to the Command Window. Removing the semi-colon at the end of specific line of code that you want displayed is considered to be poor practice. Instead you can use the `disp()` function within MATLAB to print your variable.

Example:

input:

```
>> x = 10;
>> disp(x)
```

output:

```
10
```

If you want to create a string that contains a variable, you can use the `sprintf()` function. This function allows you to specify formatting such as how many decimal places of the variable should be shown in the string. The string can then be displayed using the `disp()` function.

Example 1:

%d is used within a string to represent an integer (x) which is entered as the second argument in the `sprintf()` function after the string.

input:

```
>> output1 = sprintf('There are %d agents.', x);
>> disp(output1);
```

output:

```
There are 10 agents.
```

Example 2:

%f is used within a string to represent a floating point number (y) which is entered as the second argument in the `sprintf()` function after the string. The .2 specifies that the floating point number should be rounded to two decimal places.

input:

```
>> y = 12.1250;
>> output2 = sprintf('The value of y is: %.2f', y);
>> disp(output2);
```

output:

```
The value of y is: 12.13
```

Additionally, MATLAB has the `fprintf()` function which combines `disp()` and `sprintf()`. `fprintf()` takes a string a variables as inputs similar to `sprintf()`, and displays the string as output. It is common practice to end the string input with the new line character `\n` to ensure a new line is printed after the string.

Example 3:

%f is used within a string to represent a floating point number (z) which is entered as the second argument in the `sprintf()` function after the string. The .0 specifies that the floating point number should be rounded to the nearest integer.

input:

```
>> z = 9.5;
>> fprintf('The value of z is: %.0f\n', z);
```

output:

The value of z is: 10

As the Command Window processes single-line input statements one at a time, it is not possible to edit statements that have already been ran; however, you can navigate through and run previously ran statements using the up and down arrow keys. Additionally, you can navigate through and run a series of previously ran statements by holding down the shift key while using the up and down arrow keys.

Furthermore, you can clear all previously inputted statements with the following command:

input:

```
>> clc % this command will clear the Command Window
```

2.1.2 Workspace

MATLAB's Workspace is used to store and display variables. Variables can be created via declaration statements, or they can be imported from data files (.mat) via `load` statements. Additionally, variables stored in the Workspace can be exported to data files via `save` statements.

Example 1:

input:

```
>> x = 5; % x will be stored in the workspace with a value of 5
```

output:

Example 2:

input:

```
>> load data.mat; % variables in data.mat will be loaded into the Workspace
```

output:

Example 3:

input:

```
>> save mydata.mat; % variables in the Workspace will be saved in mydata.mat
```

output:

Furthermore, you can clear the Workspace with the following command:

input:

```
>> clear % this command will clear the Workspace
```

output:

2.1.3 Editor (or Live Editor)

MATLAB's Editor and Live Editor components are used for writing scripts (.m files) and live scripts (.mlx files), respectively. The Editor enables you to write multi-line input statements to be processed sequentially. When running scripts, outputs will appear in the Command Window. When running live scripts, outputs will appear beside or beneath your code within the Live Editor.

2.1.4 Current Folder

MATLAB's Current Folder section enables you to view which directory you are working out of. Typically, you will want to write scripts that are able to call functions and load data from other files. To do so, you should understand how to effectively manage files and folders in MATLAB. A common issue that arises for new programmers is that a file needed in a script (e.g. a .csv file) is not located in the current directory where the script itself is located.

Useful functions:

input:

```
>> currentFolder = pwd % used to print the working directory
```

output:

```
currentFolder = 'C:\Users\Felix\Documents\GitHub\APSC-200\Primer\01-Review'
```

input:

```
>> cd 'C:\Users\Felix\Documents\GitHub\APSC-200\Primer\02-ArraysMatrices'
% used to change the directory to a different folder
```

output:

input:

```
>> currentFolderContents = ls
% used to list folder contents of the current directory
```

output:

```
currentFolderContents =
    '.  '
    '.. '
    'CellArrays.mlx '
    'Intro.mlx '
    'LogicalIndexing.mlx'
```

Note the first two items of the array, '.' and '..'. These entries correspond to the current folder and the parent folder, respectively. Hence, if I wanted to change to the parent directory, I could enter:

input:

```
>> currentFolderContents = cd ..
% used to change the directory to the parent folder
```

output:

2.1.5 Search Documentation

Lastly, one other key component of the MATLAB interface that you will likely use often is the Search Documentation bar that can be found in the upper right corner of the interface. The Search Documentation bar will allow you to find syntax, descriptions, and examples of all MATLAB functions and it should be your first point of contact when seeking to understand and utilize new functions.

3 Basic Concepts

3.1 Variables

Variables can be created in either the Command Window or the Editor and will be stored in the Workspace. Variables can be assigned constant values or they can be assigned equations where their value depends on other variables.

Example 1:

input:

```
>> a = 1; % a will be stored in the Workspace with a value of 1
>> b = a*2; % b will be stored in the Workspace with a value of 2
>> fprintf('a: %d, b: %d\n', a, b);
```

output:

```
a: 1, b: 2
```

Variables are mutable meaning their values can be modified. However, modifying a variable's value will not automatically update the value of other variables.

Example 2:

input:

```
>> a = 10; % the value of a in the Workspace will now be 10
>> fprintf('a: %d, b: %d\n', a, b);
```

output:

```
a: 10, b: 2
```

Example 3:

input:

```
>> b = a*2; % the value of b in the workspace will now be 20
>> fprintf('a: %d, b: %d\n', a, b);
```

output:

```
a: 10, b: 20
```

3.2 Built-in Functions and Constants

MATLAB has built-in constants and functions available for you to use. Below are examples of two you may find yourself using; however, there are many more. The Search Documentation bar, or Google, can be used to find available constants and functions.

Example 1:

input:

```
>> x = pi % x will be stored in the Workspace with an approximated value of pi
```

output:

x = 3.1416

Although only four decimal places are shown for pi, it is represented internally with greater precision (but still not exact).

Example 2:

input:

```
>> y = sin(x) % y will be stored in the Workspace with a value of sin(pi)
```

output:

y = 1.2246e-16

As you can see, $sin(\pi)$ is not zero as you would expect. This is because the constant pi holds an approximated value. This can be resolved through alternate commands; however, I will leave that to you to research on your own.

3.3 Relational and Logical Operators

Relational and logical operators evaluate whether a condition is true or false. They are used in controlling the flow of your program with loops and if statements. True statements hold a logical value of 1 and false statements hold a logical value of 0.

Table 1: Various relational and logical operators and their MATLAB syntax.

Name	**MATLAB Syntax**	**Description**
Equal To	==	Returns 1 if left value is equal to right value. Returns 0 otherwise.
Not Equal To	~=	Returns 1 if left value is not equal to right value. Returns 0 otherwise.
Less Than	<	Returns 1 if left value is less than right value. Returns 0 otherwise.
Greater Than	>	Returns 1 if left value is greater than right value. Returns 0 otherwise.
Less Than or Equal To	<=	Returns 1 if left value is less than or equal to right value. Returns 0 otherwise.
Greater Than or Equal	>=	Returns 1 if left value is greater than or equal to right value. Returns 0 otherwise.
Or	||	Used to compare logical values. Returns 1 if left value and/or right value is 1. Returns 0 otherwise.
And	&&	Used to compare logical values. Returns 1 if left value and right value are 1. Returns 0 otherwise.
Not	~	Used to negate logical values. Returns 1 if value is 0. Returns 0 if value is 1.

Example 1:

input:

```
>> 1 + 1 == 2 % creates variable ans with a logical value of 1
```

output:

$$\text{ans} = 1$$

Example 2:

input:

```
>> 1 + 1 ~= 2 % replaces value of ans with a logical value of 0
```

output:

$$\text{ans} = 0$$

Example 3:

input:

```
>> 1 + 1 < 5 % replaces value of ans with a logical value of 1
```

output:

$$\text{ans} = 1$$

Example 4:

input:

```
>> 1 + 1 > 5 % replaces value of ans with a logical value of 0
```

output:

$$\text{ans} = 0$$

Example 5:

input:

```
>> 2 + 3 <= 5 % replaces value of ans with a logical value of 1
```

output:

$$\text{ans} = 1$$

Example 6:

input:

```
>> 2 + 3 >= 5 % ans maintains a logical value of 1
```

output:

$$\text{ans} = 1$$

Example 7:

input:

```
>> 1 || 0 % ans maintains a logical value of 1
```

output:

$$\text{ans} = 1$$

Example 8:

input:

```
>> (1 == 2) && (10 < 10 + 1) % replaces value of ans with a logical value of 0
```

output:

```
ans = 0
```

Example 8:

input:

```
>> ~(3 + 3 >= 3 * 3) % replaces value of ans with a logical value of 1
```

output:

```
ans = 1
```

3.4 If Statements

If statements are used to run commands based on the logical value of specified expressions. An If statement structure may include only an `if` statement; an `if` statement and an `else` statement; an `if` statement and one or more `else if` statements; or an `if` statement, one or more `else if` statements, and an `else` statement. An If statement structure should end with a single `end` statement.

Table 2: If statements and their MATLAB syntax

Name	MATLAB Syntax	Description

if	if *expression*
	  statement
	end
		Runs a section of code only when the specified expression has a logical value of 1.

else if	elseif *expression*
	  statement
	end
		Runs a section of code only when the specified expression has a logical value of 1 and the specified expressions of any if or elseif statements above have a logical value of 0.

else	else
	  statement
	end
		Runs a section of code only when the specified expressions of any if or elseif statements above have a logical value of 0.

Example 1:

input:

```
x = 1;
y = 1;
if x == y
    disp('x and y are equal')
end
```

output:

```
x and y are equal
```

Example 2:

input:

```
x = 1;
y = x + 1;
if x == y
    disp('x and y are equal')
else
    disp('x and y are not equal')
end
```

output:

```
x and y are not equal
```

Example 3:

input:

```
x = 1;
y = x - 1;
if x == y
    disp('x and y are equal')
elseif x < y
    disp('x is less than y')
elseif x > y
    disp('x is greater than y')
end
```

output:

```
x is greater than y
```

Example 4:

input:

```
x = 1;
y = -1;
if x == y
    disp('x is equal to y')
elseif x < 0 && y < 0
    disp('x and y are both less than 0')
elseif x >= 0 && y >= 0
    disp('x and y are both greater than or equal to 0')
else
    disp('x and y have opposite signs')
end
```

output:

```
x and y have opposite signs
```

3.5 Loop statements

There are two main types of loops; `for` loops and `while` loops. `for` loops will run a section of code a specified number of times. `while` loops will run a section of code until a specified condition is satisfied.

Although it is useful to know about loop statements, the biggest mistake all new MATLAB programmers (regardless of skill) will make is the unnecessary use of loop statements. MATLAB is not optimized for traditional `for` or `while` loop iteration. Instead, MATLAB relies on a code base of predefined functions that allow you to operate on entire arrays of data at once.

Table 3: Loop structures and their MATLAB syntax

Name	**MATLAB Syntax**	**Description**

for	for *index* = *values*
	  statement
	end
		Repeats a statement a specified number of times.

while	while *expression*
	  statement
	end
		Repeats a section of code as long as *expression* has a logical value of 1.

Example 1:

input:

```
for i = 1:3 % loop through 1-3 with a default step of 1
    disp(i)
end
```

output:

```
1
2
3
```

Example 2:

input:

```
for i = 1:2:5 % loop through 1-5 with a step of 2
    disp(i)
end
```

output:

```
1
3
5
```

Example 3:

input:

```
counter = 0;
while counter < 10 % if counter is less than 10, the code below will run
```

```
    counter = counter*2 + 2;
    disp(counter)
end
```

output:

```
2
6
14
```

3.6 Timing

Occasionally you may wish to incorporate timing commands into your MATLAB code. Below are some available functions that may be of use.

Table 4: Various Timing commands within MATLAB

Name	**MATLAB Syntax**	**Description**
tic	tic	Starts computer stopwatch
toc	toc	Reads elapsed time on stopwatch from previous tic
clock	clock	Six valued vector [Year Month Day Hour Minute Seconds]
Elapsed CPU Time	cputime	cputime returns the total CPU time used by MATLAB since it was started. The returned CPU time is expressed in seconds.

Example 1:

input:

```
tic;
startCPUtime = cputime;
X = zeros(1,1000); % Intializes an array of 1000 zeros
for i = 1:1000
    X(1,i) = i^2;
end
totalCPUtime = cputime - startCPUtime
toc;
```

output:

```
totalCPUtime = 0.0469
Elapsed time is 0.028750 seconds.
```

Notice how totalCPUtime is greater than elapsed time? This is because when multiple threads are used on a multi-processor system or a multi-core system, more than one CPU may be used to complete a task. In this case, the CPU time may be more than the elapsed time.

Example 2:

input:

```
clock
```

output:

$$\text{ans} = [2021,\ 6,\ 25,\ 12,\ 19,\ 24.8120]$$

4 Functions

To organize your project and limit the amount of code you are required to write, you can create functions to perform specific actions which you may wish to use multiple times throughout your project.

4.1 Function Definition

Functions can be defined in one of two ways: in a script or in a function file. When creating a script that contains commands and function definitions, functions must be at the end of the file. Additionally, the script file cannot have the same name as a function in the file. When creating a function file, the file must only contain function definitions. Additionally, the name of the file must match the name of the first function in the file. Furthermore, valid function names begin with an alphabetic character, and can contain letters, numbers, or underscores.

4.1.1 Syntax

Proper function definitions follow the format below which declares a function named myfun that accepts inputs x1,...,xM and returns outputs y1,...,yN.

```
function [y1,...,yN] = myfun(x1,...,xM)
% calculation
end
```

Example 1:

input:

```
z = add(9,10);
fprintf('z: %d', z);

function z = add(x,y)
z = x + y;
end
```

output:

```
z: 19
```

Example 2:

input:

```
[y,z] = addSubtract(9,10);
fprintf('y: %d, z: %d', y, z);

function [y,z] = addSubtract(w,x)
y = w + x;
z = w - x;
end
```

output:

```
y: 19, z: -1
```

4.2 Function Handles

A function handle is a MATLAB data type that stores an association to a function. Function handles are useful if you wish to pass functions as variables to other functions (ex: integration).

Example 1:

input:

```
f = @myfunction;
f_5 = f(5);
fprintf('f_5: %.2f', f_5);

function y = myfunction(x)
y = (x^2)/10 - 10;
end
```

output:

```
f_5: -7.50
```

Example 2:

input:

```
f = @myfunction;
f_int_0_1 = integral(f,0,1);
fprintf('f_int_0_1: %.1f', f_int_0_1);

function y = myfunction(x)
y = (x^2)/10 - 10;
end
```

output:

```
f_int_0_1: -9.97
```

4.2.1 Anonymous Functions

Additionally, anonymous functions can be created to simply define a function handle that can accept multiple inputs and return one output in a single executable statement. Proper anonymous function definition follows the format below which declares an anonymous function handle named myfun that accepts inputs x1,...,xM and returns a their sum.

```
myfun = @(x1,...,xM) x1 + ... + xM
```

Example 1:

input:

```
sqr = @(n) n.^2;
x = sqr(3)
```

output:

```
x = 9
```

5 Arrays and Matrices

An array is a collection of elements of the same data type whose contents can be accessed via indexing. Arrays are fundamental data structures in all coding languages and thus it is important to quickly become familiar with them. 1D arrays are referred to as vectors and can be formatted as row vectors (horizontal) or column vectors (vertical). 2D arrays are referred to as matrices.

In mathematics the concept of a vector or a matrix is readily captured through the array. MATLAB is a programming language geared towards numeric computing, and as such offers significant functionality in terms of array operations. MATLAB itself is an abbreviation of the words 'Matrix Laboratory'.

5.1 Array Declaration

To create an array you can follow the convention outlined in the examples below. Row elements (typically of type `int` or `double`) are separated by either a comma or a space and new rows are defined using a semicolon.

Example 1:

input:

```
row = [1 2 3 4]
% could equally write: row = [1,2,3,4]
```

output:

$$\text{row} = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

Example 2:

input:

```
col = [1;2;3;4]
```

output:

$$\text{col} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Example 3:

input:

```
matrix = [1, 2, 3, 4;
          5, 6, 7, 8;
          9, 10, 11, 12;
          13, 14, 15, 16]
% in one line: matrix = [1,2,3,4; 5,6,7,8; 9,10,11,12; 13,14,15,16]
```

output:

$$\text{matrix} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

MATLAB offers useful functions to create commonly needed matrices. A list of these functions relevant to the project scope can be found below.

Table 5: Built-in matrix template functions with links to documentation

MATLAB Syntax	**Description**
linspace(x1,x2,n)	Returns an array of n evenly spaced points between $x1$ and $x2$
zeros(sz)	Returns a matrix of size sz with zeros in every location
ones(sz)	Returns a matrix of size sz with ones in every location
eye(n)	Returns the $n \times n$ identity matrix
diag(v)	Returns a diagonal matrix from a vector v or returns the diagonal entries of a matrix as a vector

Example 4:

input:

```
x = linspace(0,2*pi,100); % 100 evenly spaced points between 0 and 2pi
y = sin(x);

plot(x,y);
% linspace is extremely useful in evaluating functions
```

output:

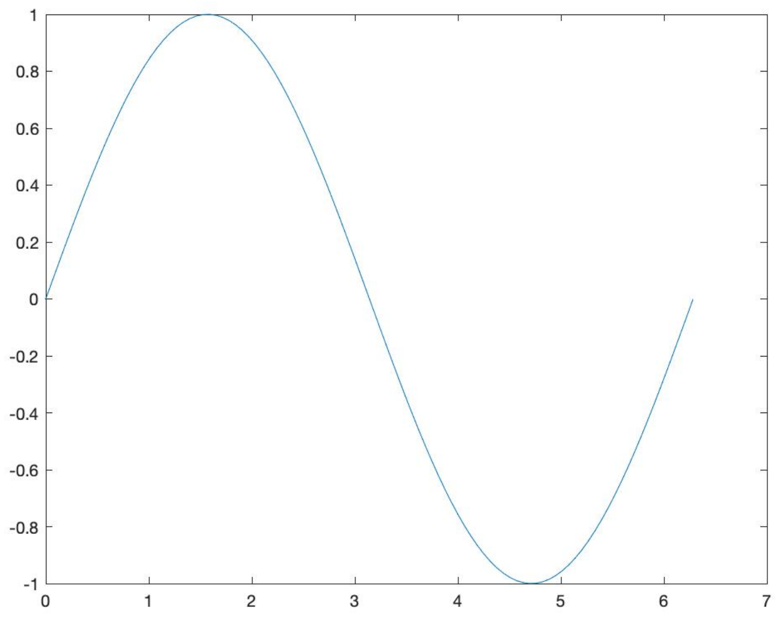


Figure 2: Using linspace to efficiently plot sin(x) on [0,2π]

Example 5:

input:

```
% generating the 3x3 identity matrix using different methods

% manually
I = [1,0,0; 0,1,0; 0,0,1];

% eye
I = eye(3);

% diag
v = ones(1,3); % 1x3 row vector [1 1 1]
I = diag(v);
```

Oftentimes matrix functions can be called with different parameters than the default examples above. Refer to the resources linked in the table above for further information.

5.2 Array Indexing

Indexing in MATLAB is 1-based (first element indexed by a 1 instead of 0) and elements are accessed using parentheses. For multi-dimensional arrays, either 2 indices (i, j) specifying the (row, col) location, or a single index counting column-wise can be used.

Example 1:

input:

```
A = [1,2;
     3,4];

v = [3, 2, 1];

result = A(1,1) + v(3) % result = 1 + 1
```

output:

$$\text{result} = 2$$

Additionally, you can use the keyword *end* to specify the index of the last element in a row or column. *end* can be used arithmetically in indexing (e.g. "*end*-1" is the second last index).

Example 2:

input:

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

result = A(end, 1) + A(2, end-1) % result = 7 + 5
```

output:

$$\text{result} = 12$$

5.3 Array Operations

Matrix addition and subtraction is performed element-wise as you'd expect and can be executed simply by using the + and - operators. Array multiplication or power operations; however, can be performed element-wise **or** with matrix multiplication. To perform element-wise operations, you are required to use the '.' prefix. Consider the following example:

Example 1:

input:

```
A = [1,0;
     0,2];

B = [0,5;
     3,0];

C = A + B; % addition is done element-wise

% 2x2 identity matrix
I = eye(2);

result1 = I*C % matrix multiplication
result2 = I.*C % element-wise multiplication
result3 = 3.*(C.^2)
```

output:

$$\text{result1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 5 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 5 \\ 3 & 2 \end{bmatrix}$$

$$\text{result2} = \begin{bmatrix} 1 * 1 & 0 * 5 \\ 0 * 3 & 1 * 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$$

$$\text{result3} = \begin{bmatrix} 3 * (1)^2 & 3 * (5)^2 \\ 3 * (3)^2 & 3 * (2)^2 \end{bmatrix} = \begin{bmatrix} 3 & 75 \\ 27 & 12 \end{bmatrix}$$

5.3.1 Built-in Array Functions

It is very common for there to already exist a function within MATLAB that can perform a standard matrix operation. In the table below several such functions are provided with links to the official documentation.

Table 6: General Matrix/Array Commands

MATLAB Syntax	Description
size(arr)	Returns a row vector containing the dimensions of an array, *arr*
sum(arr)	Returns the sum of array elements along a specified axis. For matrices the default function call returns a row vector with the sum over columns.
max(arr)	Returns the max values in array *arr* or matrix along a specific axis (i.e. row or column)
min(arr)	Returns the min values in array *arr* or matrix along a specific axis (i.e. row or column).
sort(arr)	Returns a sorted structure given an array, *arr*, in ascending order by default. This direction can be modified by changing function parameters.
arrayfun(f, arr)	Applies function, f, to each element in the array *arr*. f must be a function handle

Example 1:

input:

```matlab
A = [5,6;
      3,8]

v = [1,2,3,4,5];

disp(size(A))    % dim(A) = 2x2
disp(size(v))    % dim(v) = 1x5
```

output:

$$\text{size(A)} = [2 \quad 2], \quad \text{size(v)} = [1 \quad 5]$$

Example 2:

input:

```matlab
A = [5,6;
      3,8]

max_cols = max(A,[],1) % returns the max value of each column
max_rows = max(A,[],2) % returns the max value of each row


sum_cols = sum(A,1) % returns the sum of each column
sum_rows = sum(A,2) % returns the sum of each row
```

output:

$$\text{max\_cols} = \begin{bmatrix}5 & 8\end{bmatrix}, \quad \text{max\_rows} = \begin{bmatrix}6\\8\end{bmatrix}, \quad \text{sum\_cols} = \begin{bmatrix}8 & 14\end{bmatrix}, \quad \text{sum\_rows} = \begin{bmatrix}11\\11\end{bmatrix}$$

Example 3:

input:

```
% let d be the distances from some agents to a reference point
% sort according to closest then farthest from the reference point

d = [1, 3.3, 0.5, 0.1, 1.2, 2, 6];  % unordered collection

near = sort(d, 'ascend')
far = sort(d, 'descend')
```

output:

$$\text{near} = \begin{bmatrix}0.1 & 0.5 & 1 & 1.2 & 2 & 3.3 & 6\end{bmatrix}, \quad \text{far} = \begin{bmatrix}6 & 3.3 & 2 & 1.2 & 1 & 0.5 & 0.1\end{bmatrix}$$

In addition to general utility functions for arrays, MATLAB also offers convenient functions for common linear algebra operations.

Table 7: Useful Functions for Linear Algebra

MATLAB Syntax	**Description**
det(A)	Returns the determinant of a square matrix A
eig(A)	Returns a column vector containing the eigenvalues of a square matrix A
inv(A)	Returns the inverse of a square matrix A
transpose(A) or A'	Returns the transpose of a matrix A
cross(u,v)	Returns the cross products given two arrays u and v
dot(u,v)	Returns the dot product given two arrays u and v

Example 4:

input:

```
% create some basis/column vectors
i = [2;0;0];
j = [0;0.5;0];
k = cross(i,j);

A = [i j k] % 3x3 matrix with column vectors i,j,k

% find determinant of A
det_A = det(A)

% find the max and min eigenvalues for A
eig_vals = eig(A); % {2,0.5,1}
max_eig = max(eig_vals) % 2
min_eig = min(eig_vals) % 0.5
```

output:

$$\text{det\_A} = 1, \quad \text{max\_eig} = 2, \quad \text{min\_eig} = 0.5$$

Example 5:

input:

```matlab
% some methods for the dot product of u and v
u = [1,2,3];
v = [4,5,6];

% manual
result = u*v'; % equivalent to u*transpose(v)

result = sum(u.*v);

% built-in
result = dot(u,v)
```

output:

$$\text{result} = 32$$

Whenever possible, use a pre-written function provided by MATLAB instead of writing them on your own.

5.4 Size & Dimension Manipulation

We often need to manipulate the dimension of an array to store new information (an increase in dimensionality) or to make our data structure compatible with a pre-existing function. The term used to describe the "splicing" of two separate arrays is called **concatenation**. We can also change the shape of an array while still keeping all the information through **reshaping**.

5.4.1 Concatenation

Suppose we have the two matrices:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \text{ and } B = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

And we want to attach B onto A. We can only do this in one way is by adding B to the bottom of A as rows 4 and 5. This would look like:

$$\text{concatenation(A,B)} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

We can not add B to A as columns since the number of rows of A and B differ. This will also prevent B from being added to A in the third dimension.

In MATLAB we can make use of the following two functions for concatenation:

Table 8: MATLAB Functions for Array Concatenation

MATLAB Syntax	**Description**
horzcat(A,B)	Concatenates matrices A and B along the horizontal axis
vertcat(A,B)	Concatenates matrices A and B along the vertical axis

Example 1:

input:

```
% size(A) = (4,2)
A = [1,2;
      3,4;
      5,6;
      7,8];

% size(B) = (2,2)
B = [9,10;
      11,12];

vertcat(A,B)
```

output:

$$\text{ans} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}$$

Example 2:

input:

```
A = [1  2  3  4  5];      % size(A) = (1,5)
B = [6;7;8;9;10];     % size(B) = (5,1)

horzcat(A, transpose(B))
```

output:

$$\text{ans} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{bmatrix}$$

5.4.2 Reshaping

Reshaping involves manipulating the dimensions of a data structure for the purpose of representing information in a different manner. This is often done so that we can pass our array to a function that requires input arguments of a specific shape. The importance of this idea extends beyond simply MATLAB to the many other coding languages as well.

Table 9: MATLAB Functions for Array Manipulation

MATLAB Syntax	**Description**
reshape(A, sz)	Reshapes array A into an array of size sz

Example 1:

input:

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

A_reshaped = reshape(A, [2,3])
% we've reshaped a 3x2 array to a 2x3 one
```

output:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}, \quad \text{A\_reshaped} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Example 2:

input:

```
% many machine learning models take in nx1 feature vectors as inputs
% we can 'flatten' high dimensional information into a 1D vector with reshape

agent_coords = [0,0,0; 1,1,1; 5,2,1]

feature_vector = reshape(agent_coords, [9,1])
% note: 3x3 array has 9 elements...
```

output:

$$\text{agent\_coords} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 5 & 2 & 1 \end{bmatrix}, \quad \text{feature\_vector} = \begin{bmatrix} 0 \\ 1 \\ 5 \\ 0 \\ 1 \\ 2 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

5.5 Array Logic

Since arrays efficiently store information via elements, it is often useful to know which elements satisfy certain conditions. Similar to scalar-valued expressions, we can construct logical statement using arrays in a very intuitive manner.

When we evaluate a logical expression using arrays, the output is also an array with the same dimension as the original with 1's assigned to elements which satisfy the prescribed relation, and 0's for those that don't. Observe the examples below to see how powerful this can be.

Example 1:

input:

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

a>=4
```

output:

$$\text{ans} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Example 2:

input:

```
% suppose D contains the relative distances from some agents to one another
% where D(i,j) = D(j,i) = distance(agent_i, agent_j)
D = [0,1,5;
     1,0,2;
     5,2,0];

% find which agents are less than 2 units apart
D_bool = D<2

% recover indices of elements satisfying condition above
find(D_bool)
```

output:

$$\text{D\_bool} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \text{ans} = \begin{bmatrix} 1 \\ 2 \\ 4 \\ 5 \\ 9 \end{bmatrix}$$

Recall that 2D arrays can be indexed by a single integer which counts column-wise in a manner referred to as linear indexing.

Example 3:

input:

```
%random pairwise distances between agents
distances = [1,2,3;
             4,5,6;
             7,8,9];

%agent radii of communication
rc = [2;
      6;
      7];

distances<=rc
```

output:

$$\text{ans} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

In the example above we're essentially asking the 3 following questions:

- which pairwise distances in row vector [1,2,3] are less than or equal to a communication radius of 2?
- which pairwise distances in row vector [4,5,6] are less than or equal to a communication radius of 6?

- which pairwise distances in row vector [7,8,9] are less than or equal to a communication radius of 7?

We can also make use of an idea known as **logical indexing** where we pass the output of a logical expression as an index to our data structure to access the elements which satisfy our predefined conditions.

Example 4:

input:

```
%double the odd entries in matrix A
A = [1,2;3,4];

idx = (mod(A,2)==1)     % array with 1's where A is odd

A(idx) = 2*A(idx)    % access odd elements and double
```

output:

$$\text{idx} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}, \quad \text{A} = \begin{bmatrix} 2 & 2 \\ 6 & 4 \end{bmatrix}$$

5.6 Cell Arrays

Cell arrays are a similar to traditional arrays but differ in that they can contain multiple different data types and objects. In this sense the cell array is a more abstracted version of an array. Cell arrays are useful for grouping related information which is not restricted solely to numeric types.

5.6.1 Declaration

The example below runs through the basics of cell array declaration and element accessing/modification. Note that we use curly braces when we define our cell array.

Example 1:

input:

```
% consider a structure representing student grades and overall averages
grades = {'jack', [50, 83, 90, 92], 79;
          'felix', [85, 90, 94, 73], 86;
          'nate', [84, 89, 93, 72], 85};

% elements accessed using curly braces
felix_average = grades{2,3}
jack_grades = grades{1,2}
```

output:

$$\text{felix\_average} = 86, \quad \text{jack\_grades} = \begin{bmatrix} 50 & 83 & 90 & 92 \end{bmatrix}$$

In the example above we created an array-like structure which contains elements of type `string`, `int[]`, and `int`. It helps to think of the cell array as a container where elements are anything you want them to be (even other cells).

5.6.2 cellfun

`cellfun` is a utility function which allows one to apply a function to each cell in a cell array. The output is structure with the same dimensions of the cell array, where each entry is the function applied to the corresponding entry in the cell array.

Table 10: MATLAB Functions for Array Manipulation

MATLAB Syntax	**Description**
cellfun(func, C)	Applies function *func* to every cell in cell array C. Here, f must be a function handle

Example 1:

input:

```
C = {1, [1,2,3], [1,2;3,8]};     %stores scalar, array, and matrix
f = @(x) max(x,[],'all');        %max across all dimensions

cellfun(f,C)
```

output:

$$ans = \begin{bmatrix} 1 & 3 & 8 \end{bmatrix}$$

Example 2:

input:

```
C = {1, [1,2,3], [1,2;3,8]};     %stores scalar, array, and matrix
f = @(x) sum(x,2);               %sum over rows

cellfun(f,C, 'UniformOutput', false)     %outputs don't have to be scalar
```

output:

$$ans = \{[1]\} \quad \{[6]\} \quad \{\begin{bmatrix} 3 \\ 11 \end{bmatrix}\}$$

6 Distance and Norm Functions

Throughout your APSC 200 project you will find yourself looking to measure distances between sets of points. Perhaps you will want to find the distances between agents in order to determine if they are within range for communication, or perhaps you will want to find the distances between multiple agents and a point in space in order to determine which agent is closest to the point. Additionally, different scenarios may require different distance measurements. Perhaps you will want to calculate the Euclidean distance between two flight enabled agents hovering in space, or perhaps you will want to calculate the cityblock distance of two grounded agents travelling through city streets.

6.1 Pairwise Distance

Table 11: Pairwise Distance Functions

MATLAB Syntax	**Description**
pdist(X)	Computes the pairwise distance between observation pairs in array X
pdist2(X,Y)	Returns the pairwise distances between observations in sets X and Y

6.1.1 pdist

`pdist` is used to calculate the pairwise distance between pairs of observations. `pdist` takes as input an $n \times m$ array where n represents the number of observations ($n >= 2$) and m represents the dimension of the observations. Additionally, `pdist` may take as input a string distance metric to indicate the distance metric which is to be used (Euclidean, cityblock, etc.). The default distance metric is Euclidean. All distance metrics can be found within MATLAB's documentation.

Example 1:

input:

```
% consider the matrix X which has two observations: (1,1) and (2,2)
X = [1, 1;
     2, 2];

% determine the euclidean distance between observations in X
D = pdist(X)
```

output:

D = 1.4142

Example 2:

input:

```
% consider the matrix X which has three observations
% each row is the position of an agent in 3-D
X = [1, 7, 2;
     3, 9, 8;
     5, 5, 6];

% determine the cityblock distances between observations in X
D = pdist(X, 'cityblock')
```

output:

$$D = [10, 10, 8]$$

In this case, the pairwise distances returned correspond to the agent combinations (1,2), (1,3), and (2,3). Alternatively, you can easily locate the distance between observations by using the `squareform` function. The `squareform` function returns an $n \times n$ array where cell (i, j) represents the distance between observations i and j. One should note that cell (i, j) equals cell (j, i).

Example:

input:

```
% alternate format of distance array
Z = squareform(D)
```

output:

$$Z = \begin{bmatrix} 0 & 10 & 10 \\ 10 & 0 & 8 \\ 10 & 8 & 0 \end{bmatrix}$$

6.1.2 pdist2

`pdist2` is used to calculate the pairwise distance between two sets of observations. `pdist2` takes as input an $n \times k$ array, X, and an $m \times k$ array, Y, where n and m represent the number of observations in the two sets of observations, respectively, and k represents the dimension of the two sets of observations. Additionally, `pdist2` may take as input a string distance metric to indicate the distance metric which is to be used (Euclidean, cityblock, etc.). The default distance metric is Euclidean. All distance metrics can be found within MATLAB's documentation. `pdist2` returns an $n \times m$ array where the cell (i,j) corresponds to the pairwise distance between observation i in X and observation j in Y.

Example 1:

input:

```
% consider two sets of observations X & Y (sets of agent positions in 3-D)
X = [1,2,3;
     1,3,5;
     1,4,7];

Y = [1,5,3;
      2,9,2;
      2,3,1];

% determine the euclidean distances between observations in X and  Y
D = pdist2(X,Y)
```

output:

$$D = \begin{bmatrix} 3 & 7.1414 & 2.4495 \\ 2.8284 & 6.7823 & 4.1231 \\ 4.1231 & 7.1414 & 6.1644 \end{bmatrix}$$

6.2 rangesearch

`rangesearch` is used to determine neighbors between two sets of observations with a specified range. `rangesearch` takes as input an $n \times k$ array, X, and an $m \times k$ array, Y, where n and m represent the number of observations in the two sets of observations, respectively, and k represents the dimension of the two sets of observations. Additionally, `rangesearch` takes as input a numeric value, $r \in \mathbb{R}_+$, that represents the specified range to determine neighboring points. `rangesearch` returns an $m \times 1$ cell array, Idx, where $j \in Idx(i) \Leftrightarrow d_j \leq r$ where $d_j = ||X(j) - Y(i)||$. Additionally, `rangesearch` can optionally return an $m \times 1$ cell array, D where $d_j \in D(i) \Leftrightarrow d_j \leq r$ where $d_j = ||X(j) - Y(i)||$.

Table 12: Nearest-Neighbour Function

MATLAB Syntax	**Description**
rangesearch(X,Y,r)	Find all neighbors in X within specified distance r using input data Y.

Example 1:

input:

```
% consider two sets of observations X & Y
X = [1,2,3;
      1,3,5;
      1,4,7];

Y = [1,5,3;
       2,9,2;
       2,3,1];

% determine neighbors within a range of 5
[Idx,D] = rangesearch(X,Y,5)
```

output:

$$\text{Idx} = \begin{bmatrix} [2,1,3] \\ [] \\ [1,2] \end{bmatrix} \quad \text{D} = \begin{bmatrix} [2.8284, 3, 4.1231] \\ [] \\ [2.4495, 4.1231] \end{bmatrix}$$

This shows that all observations in X are neighbors with observation 1 in Y, no observations in X are neighbors with observation 2 in Y, and observations 1 and 2 in X are neighbors with observation 3 in Y. Corresponding distances are shown in D.

6.3 vecnorm

`vecnorm` is used to determine the vector-wise norm of an array. `vecnorm` takes as input an array, A. Additionally, `vecnorm` may take as optional inputs two positive integers, p and d, which represent the norm type and the dimension to operate along. p holds a default value of 2 and d holds a default value of 1 `vecnorm` returns the p-norm of A.

Table 13: Vector Norm Function

MATLAB Syntax	Description
vecnorm(A,p,d)	Computes the p-norm across a vector over a specified axis (e.g. rows or columns)

Example 1:

input:

```
% consider the matrix A
A = [1,2;
     2,2;
     4,1];

% return the vector-wise norms of A
N = vecnorm(A)
```

output:

$$N = \begin{bmatrix} 4.5826 \\ 3.0000 \end{bmatrix}$$

Notice how there are only two values in N? This is because by default, `vecnorm` returns the vector-wise norms of the columns of A. To obtain the vector-wise norms of the rows of A, you must specify that you wish to operate in dimension 2.

Example 2:

input:

```
% consider A to be 3 (x,y) pairs
A = [1,2;
     2,2;
     4,1];

% return the vector-wise norms of the rows of A
% the first 2 specifies the p-norm, the second 2 specifies the dimension
N = vecnorm(A,2,2)
```

output:

$$N = \begin{bmatrix} 2.2361 \\ 2.8284 \\ 4.1231 \end{bmatrix}$$

7 Symbolic Math

The Symbolic Math Toolbox in MATLAB is a powerful tool for working with mathematical expressions analytically. Working with symbolic functions allows you to perform a number of operations such as transforms, differentiation, and integrals without ever needing to evaluate the function numerically. It essentially emulates the pen-and-paper formulation of the mathematics.

7.1 Symbolic Variables

7.1.1 Declaration

Symbolic variables can be declared using either the `sym` or `syms` functions. There are minor differences in the syntax between the two, and their definitions with links to the official documentation can be found below. Generally you will wish to use `syms` when creating symbolic variables.

Table 14: Functions for Symbolic Variable Declaration

MATLAB Syntax	**Description**
sym	Creates symbolic variables and numbers
syms	Shortcut for `sym`

Example 1:

input:

```
% can create symbols for exact numbers or variables
x = sym('x');
y = sym(1/3);

% the following output is exactly 4/3 instead of 1.3333
y+1
```

output:

$$\text{ans} = 4/3$$

Example 2:

input:

```
% syms allows you to declare multiple symbols at once
syms x y z w;
```

output:

7.1.2 Symbolic to Numeric Conversion

Although symbolic variables can preserve the precision of exact numbers, there are often cases where we wish to recover the numeric approximation for use in computation. This is accomplished mainly through the use of the `double` function. The example below illustrates this symbolic-double conversion.

Table 15: Functions for Symbolic to Numeric Conversion

MATLAB Syntax	Description
double(s)	Converts symbolic variable s to a numeric type with `double` precision

Example 1:

input:

```
exact = sym(1/3);
numeric = double(exact);

disp(exact)     % display the symbolic variable
disp(numeric)   % display numeric conversion
```

output:

exact = 1/3,   numeric = 0.3333

7.2 Symbolic Functions

7.2.1 Declaration

Symbolic functions are built off of symbolic variables and are advantageous from an analytical standpoint. Symbolic functions can be declared similarly to symbolic variables **or** through the use of the `symfun` command.

Table 16: Functions for Symbolic Function Declaration

MATLAB Syntax	Description
symfun	Creates symbolic functions

There are many ways to declare a symbolic function. The examples here provided run through several alternative methods.

Example 1:

input:

```
% define f(x) = 5*x^2 + (1/2)*sin(x)^2 symbolically
syms f(x);

f(x) = 5*x^2 + (1/2)*sin(x)^2;

formula(f)  % returns the expression associated with f
```

output:

ans = sin(x)^2/2 + 5*x^2

Example 2:

input:

```
% define f(x,y) = x^2 + y^2 symbolically
syms x y;

f(x,y) = x^2 + y^2;

formula(f)
```

output:

ans = x^2 + y^2

Example 3:

input:

```
% define f(x,y) = (1/3)*e^(x^2+y^2) symbolically
x = sym('x');
y = sym('y');

f(x,y) = (1/3)*exp(x^2+y^2);

formula(f)
```

output:

ans = (1/3)*exp(x^2+y^2)

Example 4:

input:

```
% define f(x,y) = x + y symbolically
syms x y;

f = symfun(x+y,[x y]);

formula(f)
```

output:

ans = x + y

7.2.2 Evaluating Symbolic Functions

Symbolic functions can be differentiated or integrated without the need of numerical methods. Additionally you can easily evaluate symbolic functions at specific points or over a domain with a numeric return type. Some relevant functions for working with symbolic functions are included in the table below.

Table 17: Functions for Evaluating Symbolic Functions

MATLAB Syntax	**Description**
diff(f, σ)	Returns the derivative of a symbolic function, f, w.r.t. parameter σ as another symbolic function
int(f, σ)	Returns the definite or indefinite integral of a symbolic function, f, w.r.t. parameter σ
subs(f, old, new)	Substitutes parameter *old* with parameter *new* in a symbolic expression f

Example 1:

input:

```
syms f(x);

f(x) = x^2;

% evaluate at a point;
f(2)

% evaluate over a domain
domain = linspace(-10,10);
range = f(domain);
plot(domain, range)
```

output:

```
ans = 4
```

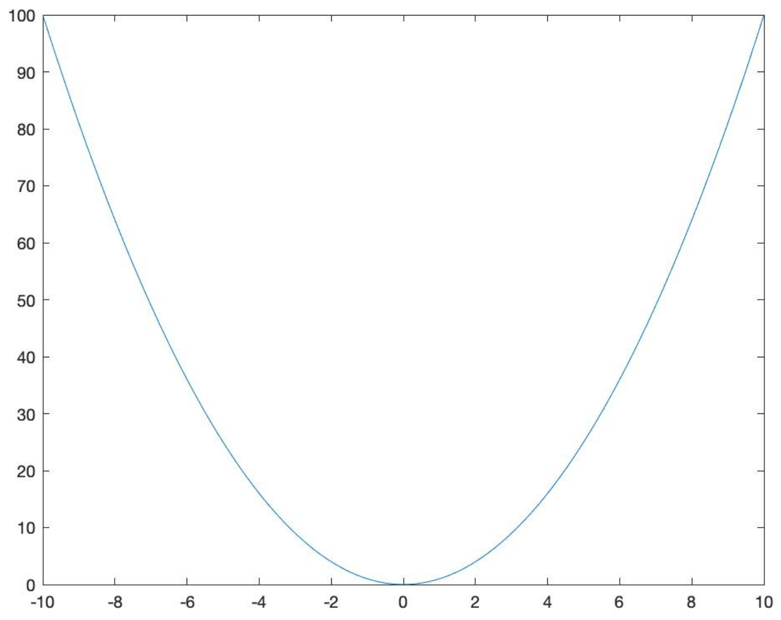


Figure 3: Evaluating symbolic function over a domain

Example 2:

input:

```
syms f(x);

f(x) = x^2;

% can use subs to change the variable x->s
syms s;
```

```
subs(f, x, s);
formula(f)

% can use subs to evaluate at the point s=2
subs(f, s, 2)
```

output:

ans = sˆ2,   ans(x) = 4

Example 3:

input:

```
syms f(x);

f(x) = sin(xˆ2);
df = diff(f,x)  % differentiate f(x) w.r.t x

formula(df)
```

output:

ans = 2*x*cos(xˆ2)

Example 4:

input:

```
syms f(x);

f(x) = 1/x;

% indefinite integral
indef = int(f,x);    % symbolic
formula(indef)

% definite integral from 1 to 10
def = int(f,x,1,10)
double(def)          % numeric
```

output:

indef = log(x),   ans = 2.3026

8 Plotting

It is important to understand and implement plotting features in MATLAB in order to better visualize problems and communicate results obtained from your models. The plotting capabilities of MATLAB are in fact so useful that they were emulated in the popular Python package `matplotlib` which is the primary plotting library for most Python developers. The following section outlines plotting in 2-D and 3-D.

8.1 2-D Line Plots

Most times you will be working with the 2-D line plot. This type of plot is best suited for time-series and sampled data, as well as for visualizing scalar functions defined on real number line.

8.1.1 Basics

In order to produce 2-D plots in MATLAB, we make use of the `plot` function. This function generates highly customizable figure and axis objects with which our data can be visualized.

Table 18: 2D Plot Function

MATLAB Syntax	**Description**
plot(X, Y)	Generates a 2-D line plot for vector Y versus X, where X and Y are of equal length.

Example 1:

input:

```
x = linspace(-pi/2, pi/2, 100);

y1 = cos(x)  % generated from x => (x,y1) equal length
plot(x,y1);

hold on;    % hold on command allows multiple plots on same figure

y2 = 1 - (1/2).*x.^2 + (1/4).*x.^4;
plot(x,y2)

hold off;   % release the hold on command
```

output:

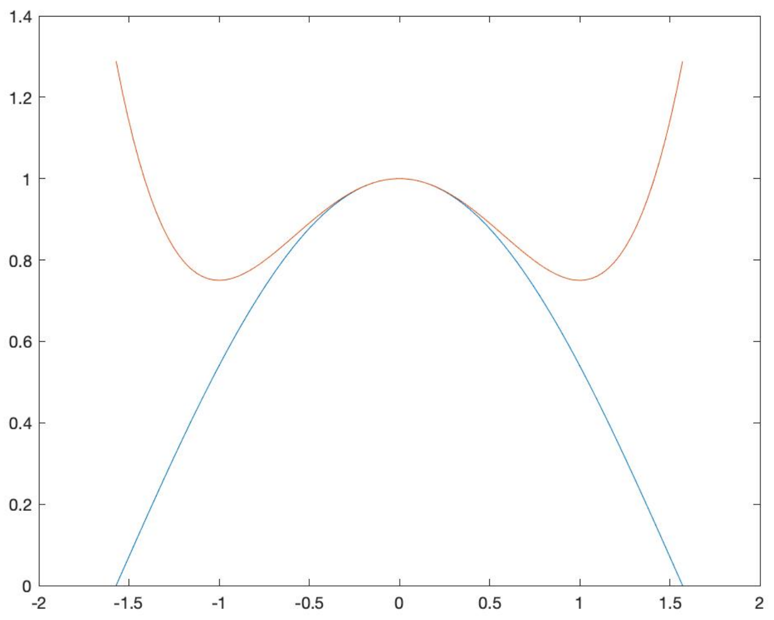


Figure 4: Plotting cos(x) versus its fourth order Taylor approximation on [-$\frac{\pi}{2}$,$\frac{\pi}{2}$]

Oftentimes we will wish to label the axes and plot itself. This can be done through the addition of a few short commands.

Example 2:

input:

```
x = linspace(0, 10, 100);

y1 = x.^2;
y2 = x;

plot(x,y1);
hold on;
plot(x,y2);

% plot a point
plot([7], [30], 'ko');

% labelling
title('this is the title');
xlabel('this is our x-axis label');
ylabel('this is our y-axis label');


hold off;   % release the hold on command
```

output:

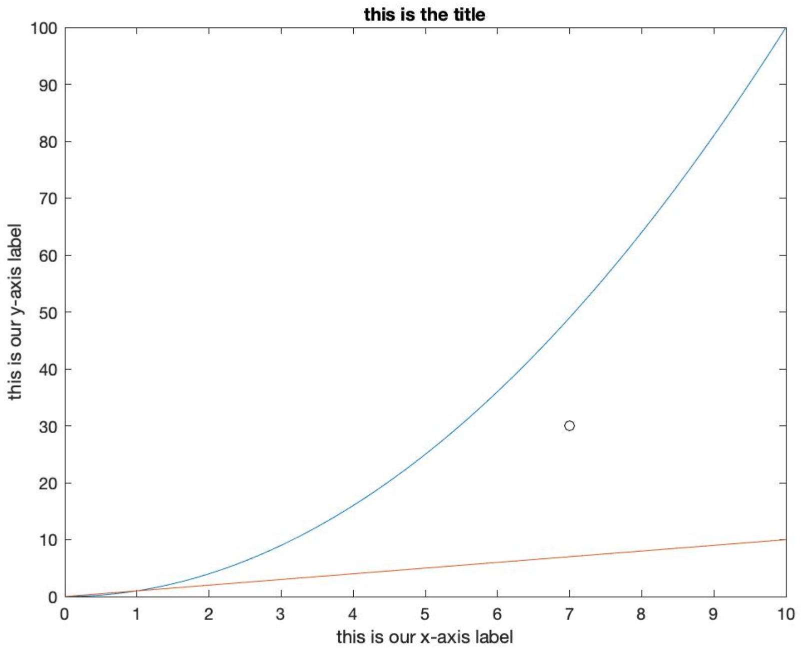


Figure 5: Plot with axis labels and title

8.1.2 Customizing Plot Features

Outside of the addition of labels and a title, we also have significant control over the line properties and figure elements. Attributes such as line colour, line thickness, line style, and marker size are easily manipulated. Features such as legends and axis grids can also be implemented in a few lines of code.

In the following examples we will show some different methods for generating more customized plots.

Example 1:

input:

```
x = linspace(-5, 5, 100);
y1 = x.^2;
y2 = x.^3;

% customize lines using name-value pair arguments
plot(x,y1, 'LineStyle', '--', 'color', 'b', 'LineWidth',0.5);
hold on
plot(x,y2, 'color', 'r', 'LineWidth',2.5);

% label
xlabel('x');
ylabel('f(x)');

% we can change default axis limits
xlim([-5 5]);
ylim([-5, 25]);

% add legend
```

```
legend('x^2', 'x^3');
```

output:

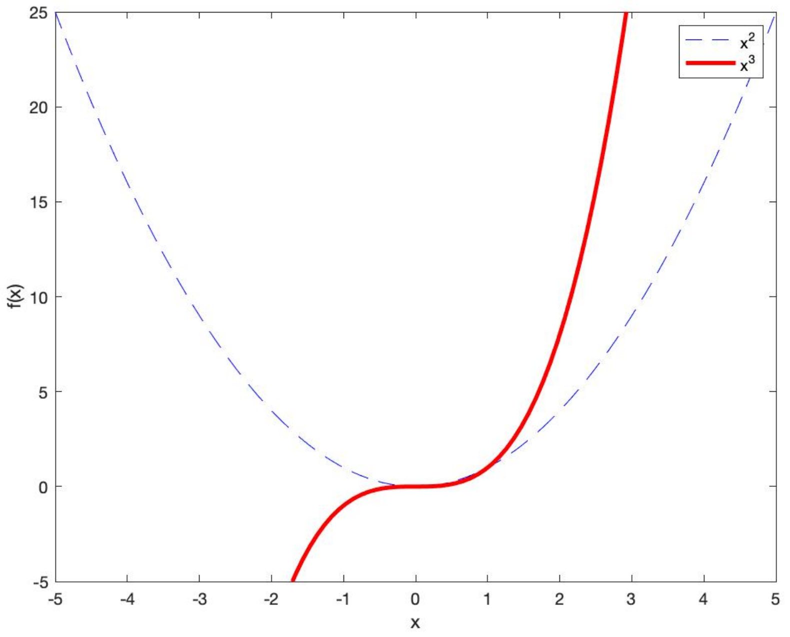


Figure 6: Plotting with custom line properties

Here we've modified line properties through the inclusion of "Name-Value Pair Arguments" in the `plot` function call. For more information on the arguments available read the official documentation.

Lines objects can also be assigned to a variable and modified after the plot command.

Example 2:

input:

```
x = linspace(0, 20, 100);
y1 = log(x);
y2 = exp(-x);

% create line objects
line1 = plot(x,y1);
hold on;
line2 = plot(x,y2);

% customize line object attributes
line1.LineWidth = 2;
line1.Color = "r";
line1.DisplayName = "log(x)";

line2.LineStyle = "--";
line2.Marker = "o";
line2.Color = "m";
```

```
line2.DisplayName = "e\^{}(-x)";     % LaTeX convention for e^(-x)

% labels
xlabel('x');
ylabel('f(x)');

% already named lines so we can call legend() as such
legend();
```

output:

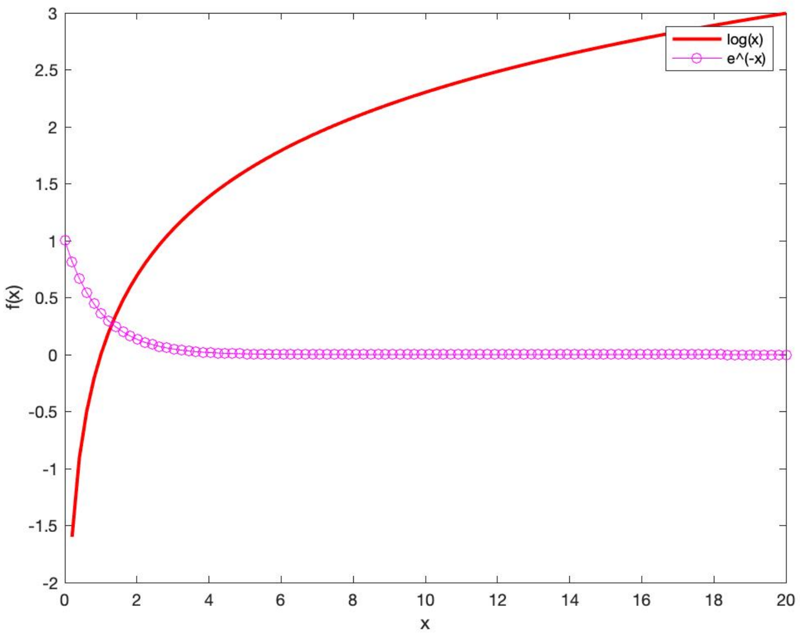


Figure 7: Plotting with custom line properties

8.1.3 Subplots

It can often be useful to visualize multiple plots in the same figure. Implementing subplots in MATLAB can be accomplished through the use of the `subplot()` or `tiledlayout()` functions.

Table 19: Subplot Function

MATLAB Syntax	Description
subplot(m,n,p)	Divides the current figure into an $m \times n$ grid and plots in position p
tiledlayout(m,n)	Creates a tiled layout grid of size $m \times n$

Review the following examples to better understand how each function works and to pick up on differences between the two functions.

Example 1:

input:

```
% plot using subplot
x = linspace(-pi, pi);

subplot(2,1,1);      % 2 rows, 1 col, plot in position 1
y1 = cos(x);
plot(x,y1);

subplot(2,1,2);      % 2 rows, 1 col, plot in position 2
y2 = sin(x);
plot(x,y2);
```

output:

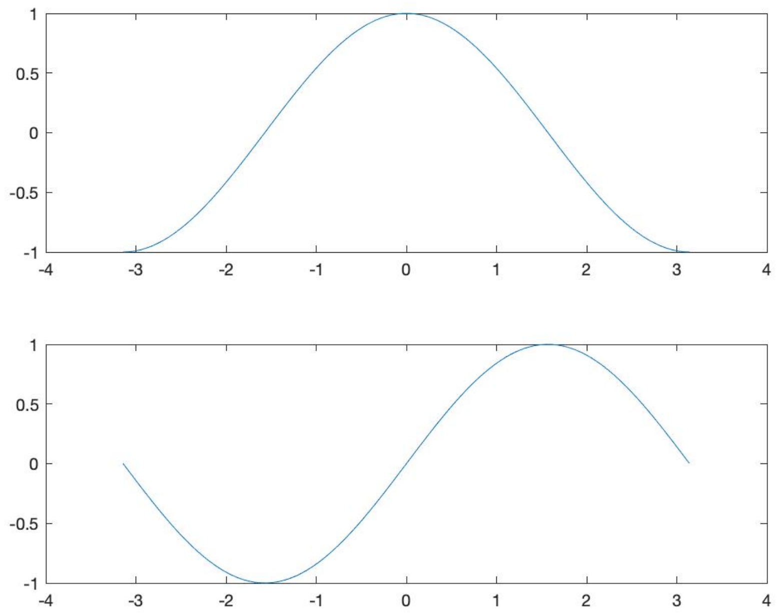


Figure 8: Plotting cos(x) and sin(x) in subplot template

Example 2:

input:

```
% plot using tiledlayout
x = linspace(-pi, pi);

tiledlayout(1,2);    % 1 row, 2 col

nexttile; % increments position in tiled layout
y1 = cos(x);
plot(x,y1);

```

```
nexttile;
y2 = sin(x);
plot(x,y2);
```

output:

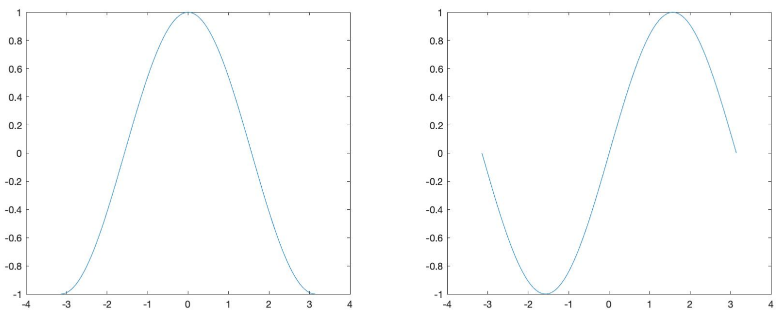


Figure 9: Plotting cos(x) and sin(x) in tiledlayout

Example 3:

input:

```
x = linspace(-1, 1);

tiledlayout(2,2);

nexttile;
y1 = x;
plot(x,y1, 'color', 'r', 'LineStyle', ':', 'LineWidth', 2);
title("first subplot");

nexttile;
y2 = x.^2;
plot(x,y2, 'color', 'b');
legend('line');
title("second subplot");

nexttile;
y3 = x.^3;
plot(x,y3, 'LineStyle', '-.', 'LineWidth', 1.5);
title("third subplot");

nexttile;
y4 = x.^4;
plot(x,y4);
grid on;
title("fourth subplot");
```

output:

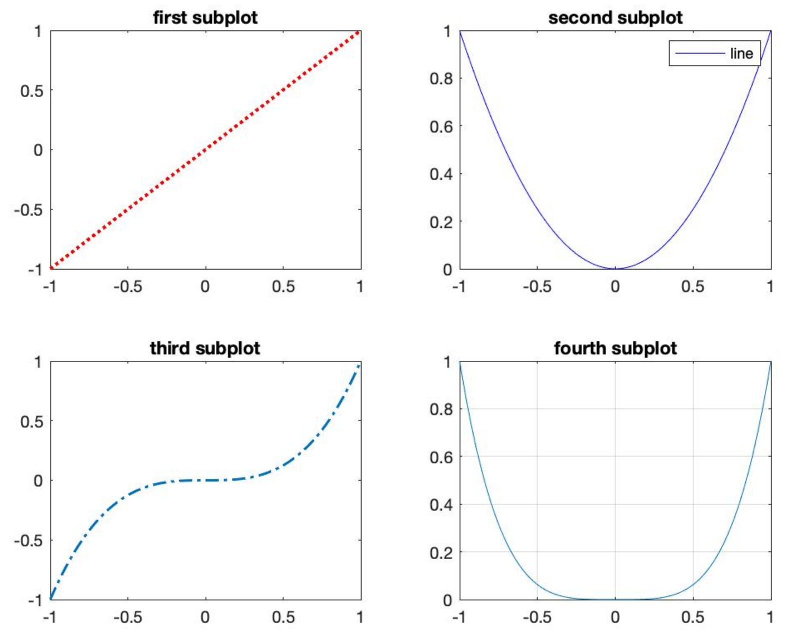


Figure 10: Another subplot example, 2x2 grid

The above example can be recreated using `subplot()` as follows:

Example 4:

input:

```
x = linspace(-1, 1);

subplot(2,2,1);     % 2 rows, 2 col, plot in position 1
y1 = x;
plot(x,y1, 'color', 'r', 'LineStyle', ':', 'LineWidth', 2);
title("first subplot");

subplot(2,2,2);     % plot in position 2
y2 = x.^2;
plot(x,y2, 'color', 'b');
legend('line');
title("second subplot");

subplot(2,2,3);     % plot in position 3
y3 = x.^3;
plot(x,y3, 'LineStyle', '-.', 'LineWidth', 1.5);
title("third subplot");

subplot(2,2,4);     % plot in position 4
y4 = x.^4;
plot(x,y4);
grid on;
title("fourth subplot");
```

Though both plotting functions accomplish the task of generating subplots, it is recommended to use `tiledlayout()` since it produces more readable and straightforward code.

8.1.4 Graph Objects

Graph objects represent undirected graphs, which is a set of nodes with edges connecting the nodes together. The `graph` function takes in the square and symmetric adjacency matrix `A` and returns the graph object `G`. Additionally, graph objects have two properties, `Edges` and `Nodes`. The `Edges` property holds data regarding the edges of the graph such as the end nodes connected by edges and the weight of the edges. The `Nodes` property holds data regarding the nodes of the graph such as the names of the nodes. Furthermore, the graph can be plotted using the `plot` function.

Example 1:

input:

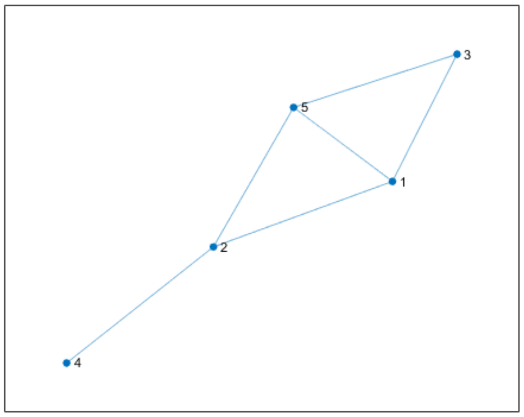
```
% suppose there are 5 agents in R^2, each with a communication radius of 5
agentPos = [3 4;
            6 7;
            1 5;
            9 3;
            2 4;];
r = 5;

% create the distance matrix D
D = pdist(agentPos);
D = squareform(D);

% create the adjacency matrix A
A = D<=r;
A = A - eye(size(A,1));

% crete and plot the graph G
G = graph(A);
plot(G)
```

output:



Example 2:

input:

```
% add property 'Names' to the graph nodes
names = {'Agent 1';
         'Agent 2';
         'Agent 3';
         'Agent 4';
         'Agent 5'};

G.Nodes.Names = names;
nodes = G.Nodes
```

output:

nodes = 5x1 table

	Names
1	'Agent 1'
2	'Agent 2'
3	'Agent 3'
4	'Agent 4'
5	'Agent 5'

Example 3:

input:

```
edges = G.Edges
```

output:

edges = 6x2 table

	EndNodes		**Weight**
1	1	2	1
2	1	3	1
3	1	5	1
4	2	4	1
5	2	5	1
6	3	5	1

8.2 Plotting in 3-D

Generating 3-D plots in MATLAB is syntactically similar to how we produced 2-D plots in the section above. The only difference is that we have a bit more information to keep track of.

8.2.1 Meshgrid

`meshgrid` is essentially the 3-D analogue of the `linspace` function in 2-D. This function allows us to easily generate a grid upon which we can evaluate scalar functions. This will make more sense as we observe the examples below.

Table 20: Mesh Generation

MATLAB Syntax	**Description**
meshgrid(X, Y)	Generates a 2-D grid from vectors X and Y

Example 1:

input:

```
x = 1:3;     % vector of unit-spaced points from 1 to 3
y = 1:3;

[X, Y] = meshgrid(x,y); % returns matrices X and Y

disp(X)
disp(Y)
```

output:

$$\mathrm{X} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}, \quad \mathrm{Y} = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}$$

By superimposing the returned matrices from the `meshgrid` call, we can better understand what we are looking at. Indeed, X and Y are nothing other than matrices storing the x and y coordinates of a grid generated by the input arguments.

$$\begin{bmatrix} (1,1) & (2,1) & (3,1) \\ (2,1) & (2,2) & (3,2) \\ (3,1) & (3,2) & (3,3) \end{bmatrix}$$

In other words, `meshgrid` allows us to re-create a subset of the Cartesian plane numerically.

Example 2:

input:

```
x = linspace(-10,10);
y = linspace(-10,10);

[X, Y] = meshgrid(x,y);

% define a surface over the grid
Z = X.^2 + Y.^2;

% plot
surf(X,Y,Z)
```

output:

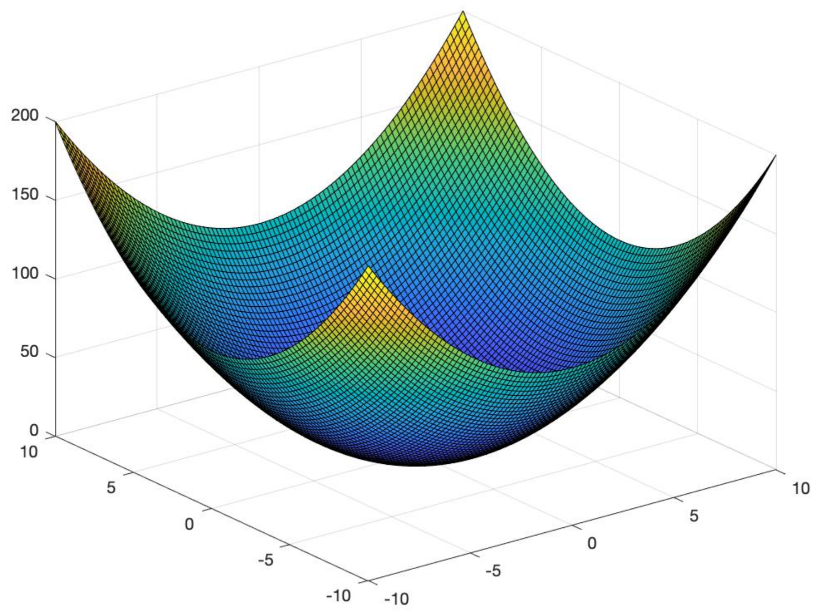


Figure 11: Plotting $z^2 - x^2 - y^2 = 0$

8.2.2 Plotting

In order to generate 3-D plots in MATLAB we make use of the functions listed in the table below.

Table 21: 3-D Plotting Functions

MATLAB Syntax	**Description**
surf(X, Y, Z)	Produces the surface Z over meshgrid (X, Y)
plot3(X, Y, Z)	Produces a scatter plot of points in matrix Z over meshgrid (X, Y)

9 App Designer

MATLAB's built-in App Designer allows users to easily create functional and interactive Graphical User Interfaces (GUIs). The App Designer can be accessed with the following command:

input:

```
>> appdesigner
```

output:

9.1 App Designer Layout

The App Designer has two separate views that you can toggle between while creating your GUI: the Design View and the Code View.

9.1.1 Design View

As you can see in Figure 12, the Design View features three separate sections, the Component Library, the Canvas, and the Component Browser. Available components are shown in the Component Library and can be drag and dropped to the Canvas to add them to the GUI. Once a component is added to the GUI it will appear in the Component Browser with information regarding its appearance and functionality.

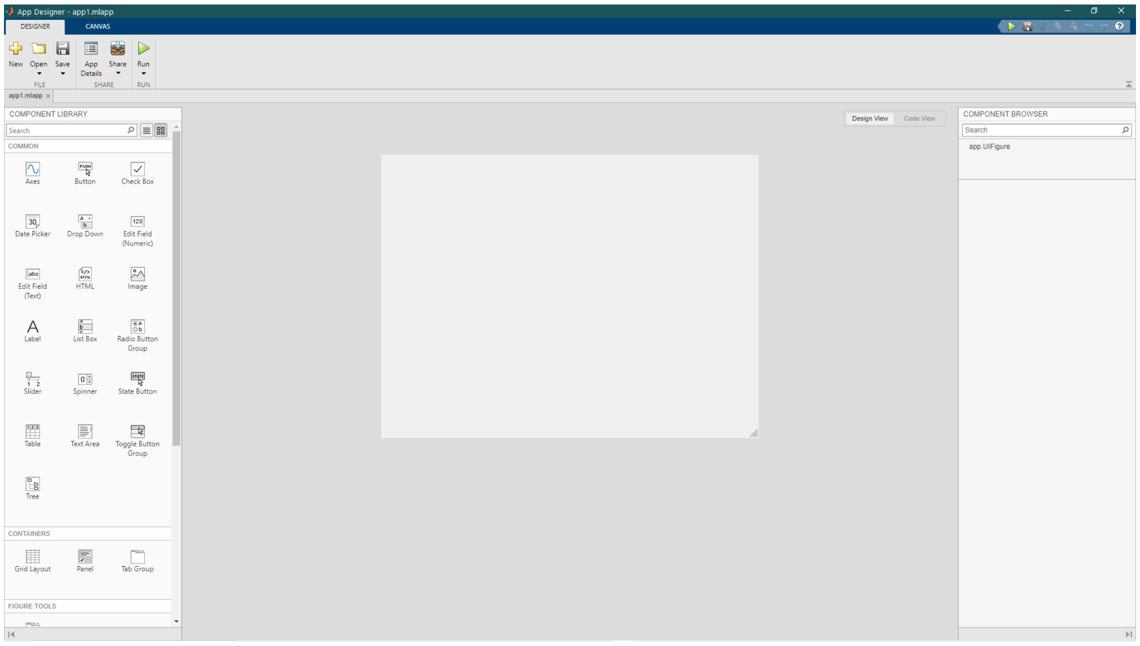
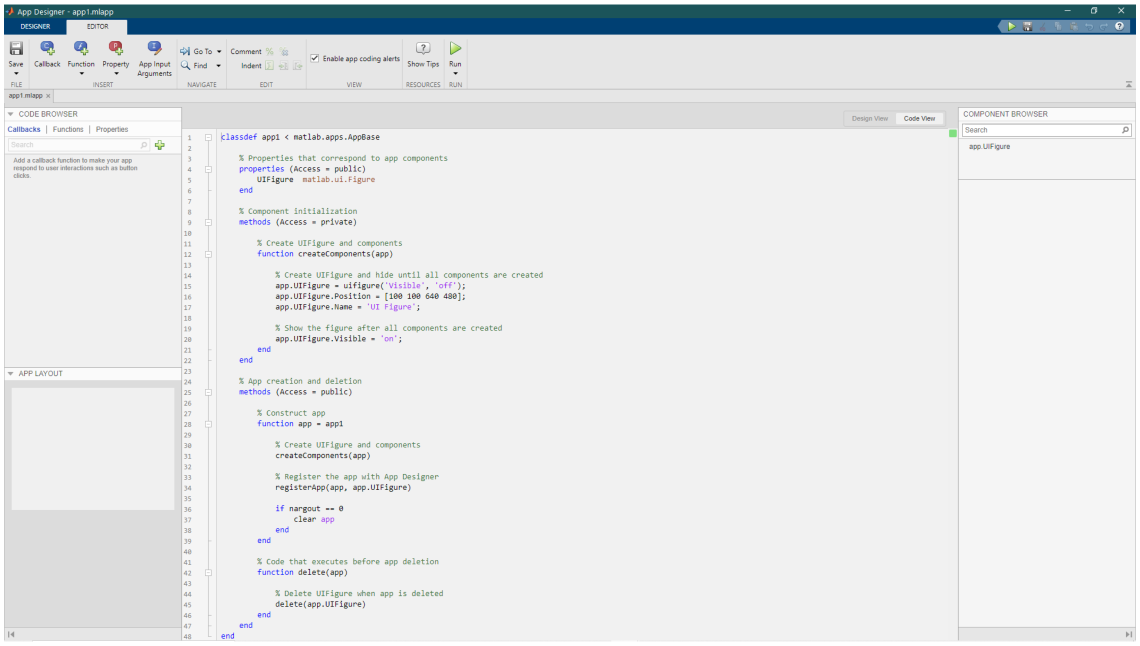


Figure 12: Design View Layout

9.1.2 Code View

As you can see in Figure 13, The Code View features two new sections along side the Component Browser. These new sections are the Code Browser and the Editor. The Editor provides access to the code which corresponds to the components that have been added to the GUI. Pre-set code that is integral to the GUI is not editable and is shown with a grey background. New sections of code that correspond to new Callbacks, Functions, and Properties are editable and are shown with a white background. These sections of code are the key to creating a functional and interactive GUI.



Add a callback function to make your app respond to user interactions such as button clicks.

```
classdef app1 < matlab.apps.AppBase

    % Properties that correspond to app components
    properties (Access = public)
        UIFigure  matlab.ui.Figure
    end

    % Component initialization
    methods (Access = private)

        % Create UIFigure and components
        function createComponents(app)

            % Create UIFigure and hide until all components are created
            app.UIFigure = uifigure('Visible', 'off');
            app.UIFigure.Position = [100 100 640 480];
            app.UIFigure.Name = 'UI Figure';

            % Show the figure after all components are created
            app.UIFigure.Visible = 'on';
        end
    end

    % App creation and deletion
    methods (Access = public)

        % Construct app
        function app = app1

            % Create UIFigure and components
            createComponents(app)

            % Register the app with App Designer
            registerApp(app, app.UIFigure)

            if nargout == 0
                clear app
            end
        end

        % Code that executes before app deletion
        function delete(app)

            % Delete UIFigure when app is deleted
            delete(app.UIFigure)
        end
    end
end
```

Figure 13: Code View Layout

9.1.3 Callbacks

Callbacks are used to add functionality and interactivity to components of the GUI in order to enable your app to respond to user interactions such as button clicks. Callbacks can be added through the Callbacks section of the Code Browser, the Callbacks section of the Component Browser, or by right-clicking eligible components on the Canvas.

9.1.4 Functions

Functions can be defined within your app so that you can call them in different places and keep your code organized. Functions can be private or public. Private functions can only be called inside the app, which makes them useful for single-window apps. Public functions can be called either inside or outside the app, which makes them useful for multi-window apps. For this project, as you will only be dealing with a single-window app, you will only be needing private functions.

9.1.5 Properties

Properties are used within your app to store and share data between callbacks and functions. Like functions, properties can be either private or public. Private properties store data to be shared inside the app, which makes them useful for single-window apps. Public properties store data to be shared inside or outside the app, which makes them useful for multi-window apps. For this project, as you will only be dealing with a single-window app, you will only be needing private properties.

10 Resources

10.1 Every Function

Topic	MATLAB Syntax	Description
Arrays	linspace(x1,x2,n)	Returns an array of n evenly spaced points between $x1$ and $x2$
	zeros(sz)	Returns a matrix of size sz with zeros in every location
	ones(sz)	Returns a matrix of size sz with ones in every location
	eye(n)	Returns the $n \times n$ identity matrix
	diag(v)	Returns a diagonal matrix from a vector v or returns the diagonal entries of a matrix as a vector
	size(arr)	Returns a row vector containing the dimensions of an array, arr
	sum(arr)	Returns the sum of array elements along a specified axis. For matrices the default function call returns a row vector with the sum over columns.
	max(arr)	Returns the max values in array arr or matrix along a specific axis (i.e. row or column)
	min(arr)	Returns the min values in array arr or matrix along a specific axis (i.e. row or column).
	sort(arr)	Returns a sorted structure given an array, arr, in ascending order by default. This direction can be modified by changing function parameters.
	arrayfun(f, arr)	Applies function, f, to each element in the array arr. f must be a function handle
	horzcat(A,B)	Concatenates matrices A and B along the horizontal axis
	vertcat(A,B)	Concatenates matrices A and B along the vertical axis
	reshape(A, sz)	Reshapes array A into an array of size sz
Lin Alg	det(A)	Returns the determinant of a square matrix A
	eig(A)	Returns a column vector containing the eigenvalues of a square matrix A
	inv(A)	Returns the inverse of a square matrix A
	transpose(A) or A'	Returns the transpose of a matrix A
	cross(u,v)	Returns the cross products given two arrays u and v
	dot(u,v)	Returns the dot product given two arrays u and v
Distance	pdist(X)	Returns the pairwise distance between observation pairs in array X
	pdist2(X,Y)	Returns the pairwise distance between observations in sets X and Y
	rangesearch(X,Y,r)	Find all neighbors in X within specified distance r using input data Y.
	vecnorm(A,p,d)	Computes the p-norm of a vector over a specified axis (e.g. rows or columns)
Symbolic	sym	Creates symbolic variables and numbers
	syms	Shortcut for `sym`
	double(s)	Converts symbolic variable s to a numeric type with `double` precision
	symfun	Creates symbolic functions
	diff(f,σ)	Returns the derivative of a symbolic function, f, w.r.t. parameter σ
	int(f, σ)	Returns the definite or indefinite integral of a symbolic function, f, w.r.t. parameter σ
	subs(f, old, new)	Substitutes parameter *old* with parameter *new* in a symbolic expression f

Plotting	plot(X, Y)	Generates a 2-D line plot for vector Y versus X, where X and Y are of equal length.
	subplot(m,n,p)	Divides the current figure into $m \times n$ subplots and plots in position p
	tiledlayout(m,n)	Creates a tiled layout grid of size $m \times n$ for easy subplots
	meshgrid(X, Y)	Generates a 2-D grid from vectors X and Y
	surf(X, Y, Z)	Produces the surface Z over meshgrid (X, Y)
	plot3(X, Y, Z)	Produces a scatter plot of points in matrix Z over meshgrid (X, Y)