CHAPTER 2

A BRIEF INTRODUCTION TO MATLAB

The mere mention of MATLAB can make a student reach for a change of major form. I will not cure you of this revulsion here. My goal in this chapter is to help you get comfortable doing the simple tasks we will need to analyze electrical systems.

If you wish to study MATLAB in greater detail I suggest two things. First, find and study a more comprehensive book on the subject. Second, use MATLAB to solve any numbers of problems you'll find in your other classes. It is a powerful tool and can be used for many things beyond the scope of this book.

2.1 Parts of the IDE

The MATLAB integrated development environment (IDE) is comprised of many smaller child-windows. Each of these windows can be arranged, docked/undocked, or closed depending on your preference. One layout of the IDE is shown in Figure 2.1.

The different child-windows of the MATLAB IDE can be accessed from the *Home* tab under the *Layout* button.

2.1.1 Current Folder

The path to the current directory is displayed below the toolstrip or in the current folder window. This directory is the first place MATLAB will call a script file from. It is also the place that files will be saved if no additional path is specified. The files from the current folder are displayed in the *Current Folder* child-window.

2.1.2 Workspace

The *Workspace* window displays the current variables in memory. If the value is simple to display it will show up directly in the *Workspace* window. If the value is larger, such as a two-dimensional matrix of double floating point values, it is better to view in the separate *Variables* window. You can see the type of each variable and double-click a variable name to view its contents in the *Variables* window. I use this window most when something isn't working the way I want it to. I can execute a command or script and see how the variables change.

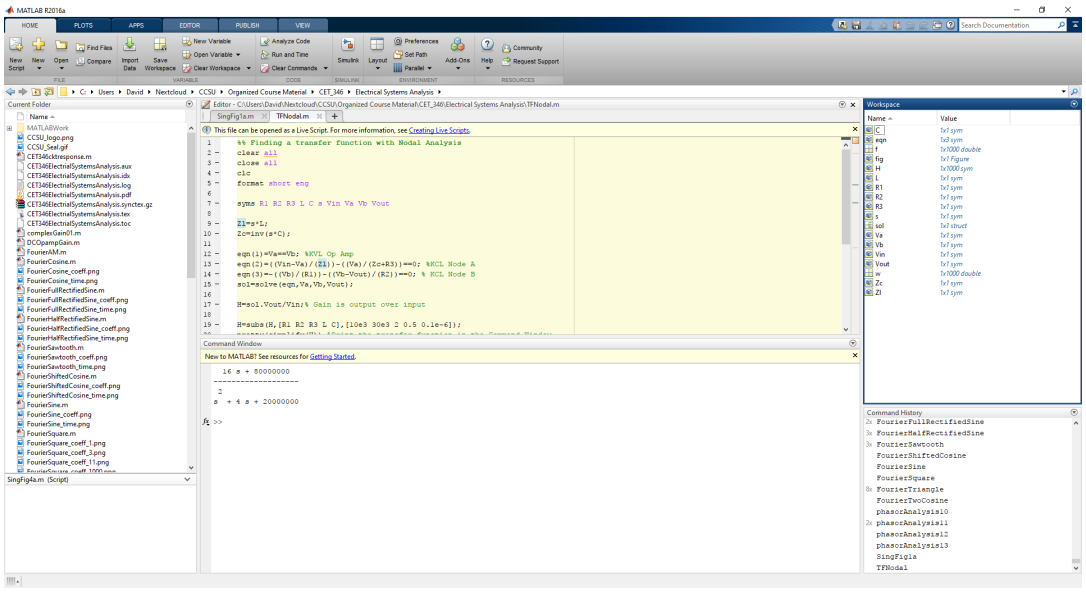


Figure 2.1 MATLAB Integrated Development Environment

2.1.3 Command History

The *Command History* window lists all previous command run. I use this window when I'm trying things I'm not certain of. Once I figure out how to use a command I can select in in this window and cut and paste it into a script.

The shift key allows selection of multiple commands in this window. The selected command can then be executed either from the right-click menu or pressing *F9*.

These are the windows, along with the editor to be discussed later, that you will use the most during this class. This is not a comprehensive list. This is enough knowledge for you to accomplish **most** things in MATLAB.

2.2 Entering an Expression into the Command Window

```
(1+sqrt(5))/2 %mathematical expression
abs(3+4j) %magtitude of a complex value
exp(1) %natural constant, e
exp(2) %e^2
2*pi %Radians in a cycle
```

2.2.1 Matrix operations vs. Element-by-element Operations

2.3 Storing Variables

MATLAB stores values in variables much like many programming languages. This is done with the assignment operator, a single '='.

While variable names can be about anything you dream up, preferably related to their function in your code, there are some rules that must be followed. MATLAB variables are valid if they conform to the following:

1. Variables shall start with a letter
2. Variables shall contain only letters, digits, and/or underscores.

MATLAB variables are case sensitive meaning that *FOO* and *Foo* are distinct variables. Finally, MATLAB variables shall not conflict with keywords such as *if* or *end*.

Try entering the following assignment statements into the MATLAB *Command Window*

```
%Valid variable assignment statements
a1=0
A_variable=(2+4)/7
A_Variable=sqrt(-5+2)

%Invlaid variable assignment statements
1a=0
if=4
Kittens!=3.14159
```

The first three are valid and the last three are invalid. Pay attention to the error messages displayed when you enter the invalid expressions. It is important to read the error messages and begin to learn what they are telling you. I find most students ignore them and get frustrated. This strategy won't lead to better knowledge of MATLAB.

```
a1 =

     0


A_variable =

    0.8571


A_Variable =

   0.0000 + 1.7321i

>> 1a=0
 1a=0
  |
Error: Unexpected MATLAB expression.

>> if=4
 if=4
   |
Error: The expression to the left of the equals
sign is not a valid target for an assignment.

>> Kittens!=3.14159
 Kittens!=3.14159
        |
Error: Unexpected MATLAB operator.
```

2.4 Types of Variables in MATLAB

There are many types of variables and this is by no means an exhaustive list. This list is intended to cover the types we will encounter in this class.

2.4.1 Double

Most real-valued numbers are stored in double precision floating point variables. These variables are 64-bit long and conform the the IEEE 754-2008 Standard. An individual double is displayed in the *Workspace* window as the value is contains. The size and type of the variable are displayed when arrays of values are stored that are too long to display.

Arrays of doubles can be stored using the following notations.

```
%an assignment statement storing a single value.
aDouble=8

%assignment statement to store a row vector, note the spaces between the
%columns
doubleRow=[0 6 0 5 3]

%an equivalent assignement statement to that shown on line 6.  Commas can
%be used between the columns rather than spaces
doubleRow=[0,6,0,5,3]

%assignment statement to store a column vector, note the semi-colons
   between the
%rows
doubleColumn=[1;8;5;6]

%assignment statement to store a matrix, note the commas between the
%columns and semi-colons between the rows
doubleMatrix=[5,-2;-2,16]
```

2.4.2 String

At times we will display and manipulate text data. Text data is stored as a character array in MATLAB (prior to version 2017b). Text can be assigned to a variable by entering it between single quotation marks.

Other operations are possible with strings such and concatenation, capitalization, forcing all characters to lower case, and comparison. Try the following commands in the *Command Window* and pay attention to the results in both the *Command* and *Workspace* windows.

```
%string assignments
a='David'
b='Broderick'

%concatenate strings (link together)
[a,' ',b]

%make all lower case
lower(a)

%make all upper case
upper(b)

%compare strings, returns 0 if not the same, 1 if identical
strcmp(a,b)
strcmp(a,'David')
```

2.4.3 Struct

A *Struct* variable is one that has "sub-variables" called *fields*. To access a field (assign or use the value in an expression) you an type *structName.fieldName*.

```
%A general example of a struct
student.name='A. Student';
student.bannerID=123456789;
student.course='CET346';

```

```
%print 'student' in command window
student

%An example more typical to this class
solution.I1=2;%A
solution.I2=-4;%A
solution.I3=300e-3;%A

%print 'solution' in command window
solution

%Using 'solution' in an expression
Vout=4*(solution.I1-solution.I2)
```

2.4.4 Boolean

We can perform Boolean operations on Boolean variables. It is more common to use boolean expression in flow control statements as discussed a little further down. The Boolean constants are *true* and *false*. Enter the command below and look at the *Workspace*. The different variable type will be evident by the icon next to the variable name.

```
%Boolean values can be true or false
A=false
B=true
```

2.4.5 Symbolic

MATLAB can manipulate and solve symbolic expressions if the Symbolic Toolbox is installed. The variables must be declared as symbolic before use with the *sym* command.

Notice the double-equal sign. MATLAB (and other languages) differentiates between an assignment operation and an equal sign. A single equal sign tell MATLAB to evaluate the expression on the right hand and place it in the variable on the left hand side. The double-equal sign is used in symbolic expressions to indicate the expression on the left of the operator is equivalent to the expression on the right. It is a subtle but important difference.

We will frequently solve symbolic expressions only to use them with component values to find a desired value (Output voltage/current, Mesh current, node voltage, resistance, etc.) The *subs* command will substitute values into a symbolic expression. In the example below I substitute into the expression for *x*. The symbolic variables I'm substituting for are *m*,*x*, and *b*. The values I'm substituting for these variables are *3*,*2*, and *-2* respectively. The order of the variables and the order of the values must match.

MATLAB will do its best to keep the results as precise as possible. Most of the time the output will be kept in fractional form. This form can often be unreadable and overly complicated. One method to reduce the complexity of the expression is with the *vpa* command (Variable Precision Arithmetic). This command will evaluate and approximate any numerical values in the expression. Line 12 in the example below approximates the values with as many decimal places as it can. Line 13 fixes the number of digits in the approximation to 2.

```
%Declare variables to be symbolic
syms y m x b
eqn=y==m*x+b

%A simple application of the solve command
x=solve(eqn,x)

%substituting values into a symbolic expression
x=subs(x,[m x b],[3 2 -2])

%approximating an expression
vpa(x)
```

```
13 vpa(x,2)
```

2.5 Formatted Output

Printing information to the screen can be useful for a number of reasons. These reasons include providing output to the user and troubleshooting a script or function. When we deal with a mix of text and numerical output the *fprintf* command is useful.

When you only provide a string, that's all you get out as seen on line 2 below. Lines 6 and 10 show values being passed to *fprintf*, first a string and then a float. the float token used is '%.1f' indicating that a single digit should be used after the decimal place. This value can be adjusted to meet your needs ie$\sim$ if you would like to print more digits.

```
1  % a simple fprintf command
2  fprintf('Say it with me, I LOVE MATLAB\n');
3
4  % an fprintf command with string parameter
5  stringVar='Broderick';
6  fprintf('Thanks Professor %s\n',stringVar)
7
8  % an fprintf command with a float parameter
9  gpa=3.000;
10 fprintf('I hope my GPA is at least %.1f\n',gpa)
```

Here is a small subset of the format tokens *fprintf* accepts

Format Token	Variable Type
%s	String
%c	Character
%d	Whole Decimal Value
%x	Whole Hexadecimal Value (lower case)
%X	Whole Hexadecimal Value (upper case)
%f	Floating point (notation based on value)
%e	Floating point (scientific notation)
\n	New Line
\t	Horizontal Tab

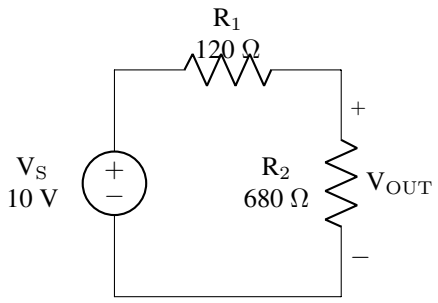
2.6 Creating a Script

There are two types of files that MATLAB can use to run a series of commands. The first we will look at is the script file. A script for MATLAB is often referred to as an m-file and has the file extension *.m*.

Many novice MATLAB users will enter command directly into the command window. I use this approach when experimenting with new commands but if I intend to use the commands more than once I move the commands into a script file. Using script files, and functions described later, improves repeatability. Once you figure out what commands to run, and in what order, the script will allow you to quickly make small changes and rerun your code.

Begin by opening a new file. It should appear in the *Editor* window. When a script runs it leaves any variables that it creates in the workspace. Subsequent calls to the script will then use any variables that are present in the workspace. I find it is good practice to clear all variables present using the *clear all* command. I also make a habit of closing all open plots, clearing the command window output, and setting the output to Engineering Notation.

Here is a quick example of a script that solves for the output of the voltage divider shown here:



Run the script shown below to find the output. Then assume you change R2 to 1 kΩ. Make a change to the script and run it again to find the new output voltage.

```
clear all %clear all variables in workspace
close all %close all open plots
clc %clear the command window
format short eng %output should be in engineering notation

Vs=10;%V
R1=120;%Ohms
R2=680;%Ohms

Vout=Vs*(R2/(R1+R2));
fprintf('The output voltage is %.3f V\n',Vout)
```

This is a small example but the same principal exists when the complexity of the math and circuit analysis grows.

2.7 Flow Control

A script with no flow control will run each line in turn, one right after another. As your scripts become more complex you may find the need to include some form of flow control. Flow control allows you to decide whether a block of code runs or not. Other forms of flow control allow you to run a block of repeatedly for a fixed or variable amount of times.

2.7.1 Comparison Operators

We tell MATLAB to run a block of code, or re-run a block of code, using Boolean values. The most common means of generating the Boolean values necessary for your code is to compare other values using relative operators. Here are the most common relative operators:

Comparison Operator	Description
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
==	equal
~=	not equal

```
%comparisons that result in Boolean values
(4<2) %false
(2<4) % true
(1==2) %false
(1~=2) %true
(pi==pi) %true
```

2.7.2 Boolean Operators

We rarely operate on a Boolean variable in this class but we frequently chain multiple relational operations together to check multiple conditions at once.

Comparison Operator	Description
&	AND
|	OR
~	NOT

```
%Boolean operators linking multiple relational operators
A=7;
B=5;
C=-2;

(A>B)&(B>C) %true
~((A>B)&(B>C)) %false
(A<B)&(B>C) %false
(A<B)|(B>C) %true
```

2.8 If Statements

An "If" statement allows your script to control which block of code is executed based on previous results or user input. For instance, you can check if a user entered a value correctly or if a function call returned a valid result.

If-statements in all languages use Boolean expressions to determine which block of code. Lines 2-4 in the example below form the simplest example of an If-statement. In this code we are checking the power limitation of a component. The block of code between the if-statement and the keyword *end* on line 4 form the block of code that will be executed **IF** the Boolean expression on line 2 is true. The value of *P* has been provided earlier on line 1. If the value of P is greater than the rated power (150 W in this example) MATLAB will print a warning to the command line.

```
P=200;%W, a fictional power dissipation
if (P>150)
    fprintf('Component is over power rating\n')
end
```

2.8.1 Else Statements

The example described above is just the first step. There are times we would like the script to run a different block of code when the condition is false. The example below shows a simple if-else-statement. The condition on line 2 determines which block of code will be run. If the condition is true line 3 will be run. If the condition is false line 5 will be run.

```
P=200;%W, a fictional power dissipation
if (P>250)
    fprintf('Component is over power rating\n')
else
    fprintf('Component won\''t explode\n')
end
```

2.8.2 Elseif Statements

Finally there are cases where multiple conditions should be checked in a specific order. The if-elseif statement serves that purpose.

In the example shown below the condition on line 2 is checked first. If that is true line 3 is run. If it is not true the condition on line 4 is checked. If it is true line 5 is run. If it is not then line 7 is run.

In each case the block of code to be run is bounded by the keywords *if*, *elseif*, *else*, and *end*. It is good practice (but not syntactically required) to indent each block of code to improve readability. This practice was demonstrated in all of the examples in this section.

```
1 P=200;%W, a fictional power dissipation
2 if (P<150)
3     fprintf('A 150 W rated component will work\n')
4 elseif (P<250)
5     fprintf('A 250 W rated component will work\n')
6 else
7     fprintf('Component will explode\n')
8 end
```

2.9 Loops

Loops are the construct that allow us to use the strengths of a computer. We use loops to perform the same block of code repetitively, typically with a small change in a variable value. There are two types of loop we will look at here 1) the indefinite while-loop and 2) the definite for-loop.

2.9.1 While Loops

A while-loop is used to run a block of code an indefinite number of iterations. A Boolean condition is part of the while statement (line 3 of the example below) that determines whether the block of code is run on each iteration. If the condition is false execution continues after the end statement on line 6 of the example below.

```
1 %An example of a while loop
2 x=input('Enter an even number: ');
3 while(mod(x,2)~=0)
4     fprintf('That''s not an even number.\n')
5     x=input('Enter an even number: ');
6 end
7 fprintf('You entered %d. That''s an even number.\n',x)
```

2.9.1.1 Infinite Loops

In the previous example, notice that in each iteration of the loop the variable from the condition on line 3 can change. Without the variable changing to cause the condition to be false the loop will run forever.

In the following example the condition on line 3 does not change. It is always evaluated as *true*. The loop will run forever (aka for an infinite number of iterations) unless the script is stopped. If you run this one, try pressing Ctrl-C to stop it.

```
1 %An example of a while loop
2 x=input('Enter a number to break out of the loop: ');
3 while(true)
4     x=input('Enter a number to break out of the loop: ');
5 end
6 fprintf('You entered %d. That''s an even number.\n',x)
```

2.9.2 For Loops

In contrast to a while-loop a for-loop runs for a predetermined number of iterations. On line 2 of the example below we see the opening line of a for-loop. On this line I've defined the loop variable *index* to iterate over the range defined on the right side of the =. The block of code between the for statement and the end keyword will run once for each value defined by that range.

```
1 %a simple for-loop example
2 for index=1:5
3   fprintf('Iteration Number %d\n',index)
4 end
```

2.9.2.1 *Specifying Ranges*

There are two common forms of ranges used in MATLAB. These ranges are used in many places. For loops are only the first place we will see them. The format of the statement in the previous example looks like:

Range Start : Range End

The step size is implied to be 1 in this case. That is, the first iteration of the block of code will take *index* to be 1. The next iteration will use a value of 2 for *index*. This will continue adding one to *index* between each iteration until the end of the range is reached.

If you run the example above you can see this illustrated. The block of code simply prints out the loop variable. You will see 5 iterations of this code block with an output of 1, 2, 3, 4, and 5.

We will use the alternate form of a range in this class. Frequently we will define the range of a for loop to cover a period of time of a range of frequencies. In these cases we want to use step sizes other than 1. In that case we can specify the range in this format:

Range Start : Step Size : Range End

The example below shows a for loop that iterates the loop variable *t* over a period from 0 to 5 with a step size of 0.1. We will often work with time-variant sinusoidal signals and in this case I'm considering the loop variable to be time. That's why I named it *t* instead of *index*.

The block of code will run 51 times with the range as defined on line 2. During the first iteration *t* will be 0. Again, I'm interpreting this as 0 seconds. In the next iteration *t* will be 0.1 ms. In each subsequent iteration *t* will increase by the 0.1 ms step size until 5 s is reached. The block of code in this example prints out the time and the value of sin(t) at that time.

```
1 %a simple for-loop example
2 for t=0:.1:5
3   fprintf('t= %.1f\t v= %.3f\n',t,sin(t))
4 end
```

2.10 Functions

Functions allow us to reuse code by grouping instructions into a separate file. The simplest form of a MATLAB function is shown here:

```
1 function firstFunction()
2     fprintf('Hello from inside firstFunction.\n')
```

You can call the code contained within this file from another script or function as shown here:

```
1 fprintf('Hello from the script.\n')
2 firstFunction
3 fprintf('Goodbye from the script.\n')
```

It is important to note two things. First, the file containing the function should be named after the function. In this case, the file should be named *firstFunction.m* to match the name on line 1.

Second, this form of a function is the simplest form and does not operate differently than a script with the same name. In order to use the power of functions in MATLAB you should read the next sections.

2.10.1 Putting Values In

```
function secondFunction(input)
    for i=1:input
        fprintf('%d\t',i)
    end
```

2.10.2 Getting Values Out

```
function out=add(a,b)
 out=a+b
```

```
function [minimum,maximum]=minMax(a,b)
    minimum=min(a,b);
    maximum=max(a,b);
```

2.10.3 Variable Scope

```
function scopeExample(input)
    x=10;
    fprintf('%.2f\n',input)
    fprintf('%.2f\n',x)
```

```
x=2;
scopeExample(x)
```

2.11 Matrices

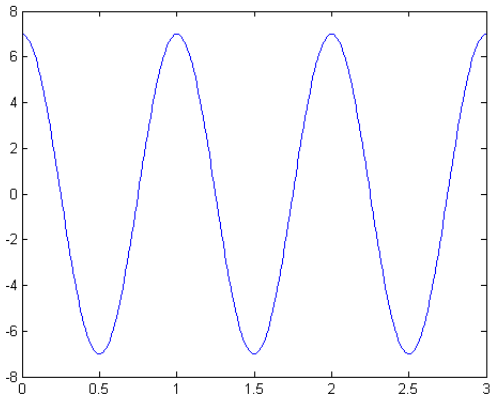
2.11.1 Assigning Values to Locations

2.11.2 Assigning a Range of Values in a Single Statement

2.11.3 Retrieving Length and Size

2.12 Plotting

```
%Example plotting a time-domain function
clear all
close all
clc
format short eng

t=0:.01:3;
v=7*cos(2*pi*t);
plot(t,v)
```

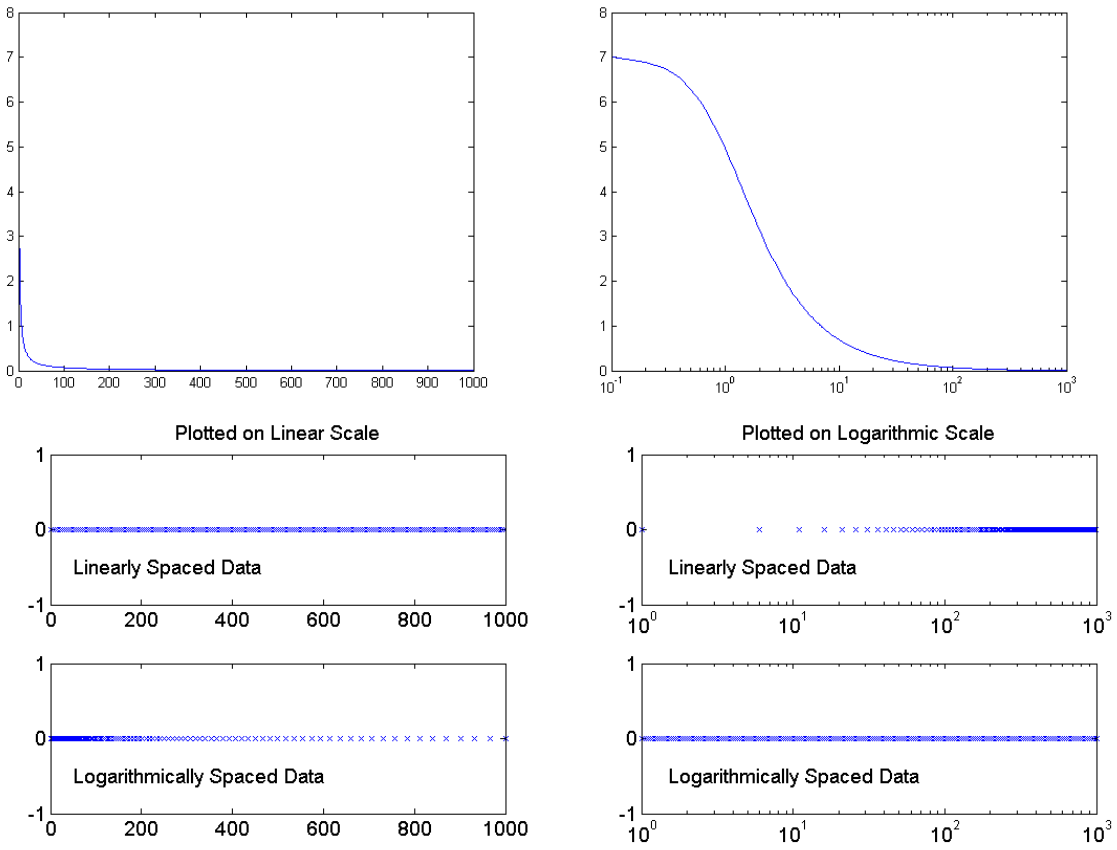


2.12.1 Logarithmic Scales

2.12.1.1 Logarithmically Spaced Variables Logspace

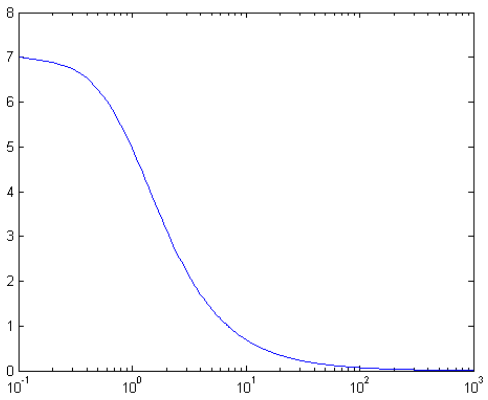
```
clear all
close all
clc
format short eng

C=125e-3;
R=8;

w=.1:.1:1000;
Zr=R;
Zc=1./(j.*w.*C);

V1=12*exp(j*-90*(pi/180));
V2=6*exp(j*65*(pi/180));
Vs=V1+V2;
Vo=Vs.*(Zc./(Zc+Zr));

plot(w,abs(Vo))
```

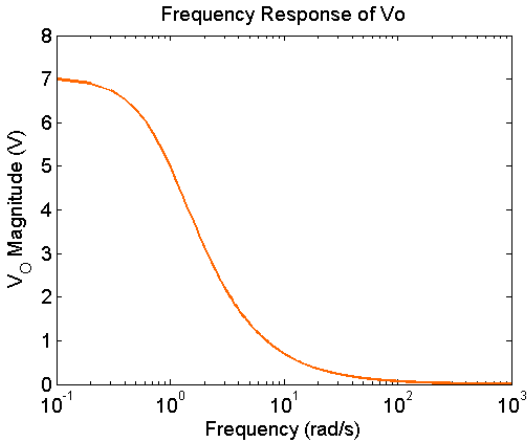
```
clear all
close all
clc
format short eng

C=125e-3;
R=8;

w=.1:.1:1000;
Zr=R;
Zc=1./(j.*w.*C);

V1=12*exp(j*-90*(pi/180));
V2=6*exp(j*65*(pi/180));
Vs=V1+V2;
Vo=Vs.*(Zc./(Zc+Zr));

semilogx(w,abs(Vo))
```



2.12.2 Titles, Labeling Axes, and Legends

```
clear all
close all
clc
format short eng

C=125e-3;
R=8;

w=.1:.1:1000;
Zr=R;
Zc=1./(j.*w.*C);

V1=12*exp(j*-90*(pi/180));
V2=6*exp(j*65*(pi/180));
Vs=V1+V2;
Vo=Vs.*(Zc./(Zc+Zr));

semilogx(w,abs(Vo),'LineWidth',2,'Color',[1 .4 0])
xlabel('Frequency (rad/s)')
ylabel('V_O Magnitude (V)')
title('Frequency Response of Vo')
set(findall(gcf,'-property','FontSize'),'FontSize',14)
```





2.12.3 Making Things Pretty

Font sizes set(findall(gcf,'-property','FontSize'),'FontSize',12)

2.13 Keeping a Clean Workspace

clear all close all clc

2.14 Interpreting Errors

2.15 Debugging

2.15.1 Stepping Through Your Script

2.15.2 Breakpoints

2.15.3 Conditional Breakpoints

2.15.4 Pause

2.15.5 Keyboard