MATLAB and Complex Numbers

The following provides an introduction to some of the ways *MATLAB* can be used to work with complex numbers. After covering some basic concepts, we will work through an example using a circuit assumed to be in the sinusoidal steady state. We will also illustrate the use of some of the plotting capabilities of *MATLAB*.

MATLAB can be used as a calculator that is as comfortable with complex numbers as it is with real numbers. After starting *MATLAB*, in the "Command Window" you should see the prompt >>. You type your input at this prompt and then hit return/enter. For example, the following shows the result of multiplying 6 and 7, the result of taking the square root of 42, and the result of raising 2 to the 20th power ("^" is the exponentiation operator).

```
1  >> 6 * 7
2  ans =
3      42
4  >> sqrt(42)
5  ans =
6      6.4807
7  >> 2^20
8  ans =
9       1048576
```

Note that the numbers shown to the left are line numbers that are provided in these notes merely for the sake of reference. You will not see these in the Command Window.

When you enter a command at the >> prompt, the result is shown to you and the results is stored in the variable `ans`. You can store values or results to names of your own choosing. Variable names must start with a letter which can be followed by any number of letters, digits, or underscore characters. To suppress results from being shown, end the statement with a semicolon (`;`). (Suppression of output will become important latter when you are dealing with large collections of data.) To see the value of a variable, enter the variable name and hit return. The following illustrates these behaviors.

```
1  >> 6 * 7;
2  >> ans
3  ans =
4      42
5  >> a1 = sqrt(42)
6  a1 =
7      6.4807
8  >> two_to_the_20 = 2^20
9  two_to_the_20 =
10      1048576
```

Notice that line 1 doesn't generate any output, but behind the scenes the value of `ans` has been set to 42. This fact is illustrated in lines 2 through 4.

File: `matlab-n-complex-numbers.tex`

Again, *MATLAB* is perfectly comfortable with complex numbers. For example, if we want the square root of -42, we could simply enter either of the following commands.

```
1 >> sqrt(-42)
2 ans =
3    0.0000 + 6.4807i
4 >> sqrt(-6 * 7)
5 ans =
6    0.0000 + 6.4807i
```

Line 4 illustrates the fact that we can have an expression as the argument of a function.

The results shown in lines 3 and 6 indicate a real part of zero and an imaginary part of (approximately) 6.4807. Typically we (in engineering) would write $\sqrt{-42} = j6.4807$ where $j = \sqrt{-1}$. In physics and math, i is often used to represent $\sqrt{-1}$ and that is the notation *MATLAB* uses when it shows results. Also, importantly, in *MATLAB* the i or j comes at the *end* of the number, *not* the beginning. This is a consequence of variable names starting with a letter. So, if you write `i1`, and you haven't previously defined a variable named `i1`, you will get an error message. Nevertheless, in *MATLAB* you can simply write `i` or `j` to indicate $\sqrt{-1}$. These are built-in *MATLAB* variables[1]. However, as you may recall from previous programming courses, `i` is frequently used as a user-defined variable name (especially in the context of indexing). *MATLAB* will allow you to assign values to built-in variable names. It does this "silently" (no warning or error messages are generated). This can lead to errors that can be avoided if you consistently write either `1i` or `1.0j` to represent $\sqrt{-1}$ (you can interchange `i` and `j` in these two numeric representations, i.e., `1j` or `1.0i` also represent $\sqrt{-1}$). More generally, you can enter complex values simply by putting either an `i` or a `j` at the end of the imaginary part. (Despite the fact that *MATLAB* uses `i` to indicate the imaginary part when displaying values, it accepts either `i` or `j` for input.) If we are entering a number that is purely imaginary, there is no need to explicitly indicate the real part is zero. The following illustrates these behaviors.

```
1  >> 1i * 1.0j
2  ans =
3      -1
4  >> 6.4807i^2
5  ans =
6    -41.9995
7  >> i * j
8  ans =
9      -1
10 >> i = 3;
11 >> j = 2;
12 >> i * j
13 ans =
14      6
15 >> i1
16 Undefined function or variable 'i1'.
```

[1]Technically they are built-in functions, but that distinction isn't really important to us.

```
>> 1i
ans =
   0.0000 + 1.0000i
```

Note that in line 1 we uses both `i` and `j` to indicate the imaginary part of the number (and these numbers are purely imaginary). As you can see, that mixture is perfectly fine. In line 4 we square $j6.4807$ which was what we were previously told was the square root of -42. As shown in line 6, the result is not what we expected. The problem is that, by default, *MATLAB* does not display all its digits of precision (and, even if it did, because the square root of 42 is irrational, we would never get the true exact result). We'll return to this point in a moment. In lines 7 through 9 we see that indeed *MATLAB* says that `i` times `j` is -1. However, in lines 10 and 11 we overwrite the built-in values of `i` times `j`. *MATLAB* is happy to let us do this! Now, as we see in lines 12 to 14, `i` times `j` is 6. Lines 15 and 16 shows that, because we have not previously defined a variable names `i1`, we get an error if we try to use that variable name. However, in lines 17 through 19 we again see that `1i` is the same as $\sqrt{-1}$. This will always be true (there is simply no way to change `1i` to be anything other than $\sqrt{-1}$).

If we continue with the code we've written so far, we now have `i` and `j` set to 2 and 3. We can clear the value of a variable, thus returning it to its built-in value—assuming it has one or making the variable name undefined if it doesn't have a built-in value—by using the `clear` command. We can `clear` selected variables by specifying them individually or we can clear all variables by simply entering the command `clear` by itself. Before demonstrating this, note that you can put multiple statements on a single line by separating them with semicolons.

```
>> i = 3; j = 2; abc = 123;
>> i * j * abc
ans =
   738
>> clear i
>> i * j * abc
ans =
   0.0000e+00 + 2.4600e+02i
>> clear
>> i * j * abc
Undefined function or variable 'abc'.
>> abc = 123;
>> i * j * abc
ans =
  -123
```

Lines 2 through 4 shows that the product of the current values of `i`, `j`, and `abc` is 738. In line 5, the current value of `i` is cleared, returning it to its built-in value corresponding to $\sqrt{-1}$. This is verified in lines 6 through 8 where we now see this product is equal to $j246$. Note that *MATLAB* uses the standard floating-point notation for the exponential representation used by many programming languages and calculators where "`e+nn`" is equivalent to 10^{nn}. Thus, `2.4600e+02i` is equal to $j2.46 \times 10^2 = j246$. ("`e-nn`" is equivalent to 10^{-nn}.) In line 9 we use `clear` to clear all the variables. Thus, when we attempt to multiply `i`, `j`, and `abc` we get an error because `abc` is now

undefined. In line 12 we reset the value of `abc` to 123. Now when we multiply `i`, `j`, and `abc` we get -123 because both `i` and `j` are set equal to $\sqrt{-1}$.

Let's return to the fact that we didn't get -42 in the previous listing when we thought we were squaring its square root, i.e., when we squared `6.4807i` we didn't get -42. By using the "`format`" command we can tell *MATLAB* we want to see more (or fewer) digits in the echoed output. This command does not affect the accuracy of the underlying operations—this merely affects what is shown.

Before turning to more code, note that a comment in *MATLAB* is indicated with the percent sign (`%`). The percent sign and everything that follows it on a line is ignored. You can start a line with an expression to be executed and also include a comment on the remainder of the line. So, let's try again to obtain -42 via its square root.

```
1  >> format long  % Display more digits of precision.
2  >> sqrt(-42)
3  ans =
4    0.000000000000000 + 6.480740698407860i
5  >> % See what the square of this more accurate value is.
6  >> 6.480740698407860i ^ 2
7  ans =
8     -42
```

There! That's more like it. We get the expected result of -42 when we use the additional digits of accuracy. As something of an aside, in line 6 space was added between the exponentiation operator and its operands. This was done merely to demonstrate that *MATLAB* doesn't care about whitespace in such expressions.

Now, let's set the display (i.e., `format`) of numbers back to the default value of `short` and also illustrate that behind the scenes *MATLAB* maintains all the digits of accuracy it has available.

```
1  >>  % Return to default display format which is short.
2  >> format short
3  >> life = sqrt(-42)
4  life =
5     0.0000 + 6.4807i
6  >> life * life
7  ans =
8     -42
9  >> format long
10 >> life
11 life =
12   0.000000000000000 + 6.480740698407860i
```

From what we see in lines 4 and 5, it looks like the variable `life` has reduced precision. However, when we square it in line 6, we get the correct value of -42. Furthermore, when we change the `format` back to `long`, in lines 11 and 12 we see that `life` does indeed contain all the digits of precision *MATLAB* can provide.

Given the discussion above, something to keep in mind is that if you use the default `format` and you copy and paste values, you may be using reduced precision and, in some cases, this may lead to results that are significantly off. You can reduce the risk of accidentally introducing additional errors associated with finite precision if you store results to variables and then use those variables directly in subsequent calculations.

So far we have only considered purely real or purely imaginary numbers. To get a general complex number, as implied by the results you have seen above, you add (with the plus sign) the real and imaginary parts. The following shows how we can multiply $3 + j4$ times $5 + j12$. Note that, because multiplication has higher precedence than addition, *the parentheses that appear in the following are necessary*.

```
>> (3 + 4j) * (5 + 12j) % Yields desired result.
ans =
 -33.0000 +56.0000i
>> % The following does not yield the desired result owing
>> % to the multiplication occurring before the addition.
>> 3 + 4j * 5 + 12j
ans =
   3.0000 +32.0000i
```

MATLAB displays complex numbers in rectangular format. To obtain the magnitude and phase necessary for polar representation, we use the `abs()` and `angle()` functions, respectively. This is illustrated in the following:

```
>> s = (3 + 4j) * (5 + 12j);
>> abs(s)
ans =
    65
>> angle(s)
ans =
    2.1033
```

MATLAB typically expresses angles in radians and assumes angles are input as radians. Thus, the angle we see in line 7 is in radians. To convert between radians and degrees, we can use `rad2deg()` (to convert radians to degrees) or `deg2rad()` (to convert degrees to radians). This seems the appropriate point to mention that *MATLAB* provides the built-in variable `pi` corresponding to π (3.1415...). *MATLAB* also provides the sine and cosine functions via `sin()` and `cos()`. These functions assume arguments in radians.

```
> s = (3 + 4j) * (5 + 12j);
>> theta = angle(s)
theta =
    2.1033
>> rad2deg(theta)
ans =
  120.5102
```

```
>> cos(theta)
ans =
   -0.5077
>> sin(theta)
ans =
    0.8615
>> pi
ans =
    3.1416
>> cos(pi/2)
ans =
   6.1232e-17
>> sin(pi/4)
ans =
    0.7071
```

In line 2 we set `theta` equal to the angle of `s`. In lines 5 through 7, we obtain the angle in degrees. Lines 8 through 13 show the cosine and sine of this angle (again, the argument is assumed to be in radians). Lines 14 through 16 show the built-in variable `pi` (displayed using the default `short` format). As shown in lines 17 through 19, taking the cosine of `pi/2` yields a very small, but non-zero, number (the correct value of $\cos(\pi/2)$ is zero). This error is a consequence of finite precision and there is nothing we can do about this. Lines 20 through 22 provide the sine of $\pi/4 = 45°$.

MATLAB does provide versions of the sine and cosine functions that assume the arguments are in degrees. These functions are `sind()` and `cosd()`. You are, of course, welcome to use these functions, but please be careful when switching back and forth between radians and degrees as this is a common source of errors.

Before turning to the circuit example, you should note that you can get help within *MATLAB* by typing `help` followed by the thing on which you want help. In addition to demonstrating the use of `cosd()` and `sind()`, the following shows what `help` returns for a couple of the *MATLAB* features we have been discussing. (For brevity, some of the output has been omitted and the font size has been reduced to permit mirroring what you would see in the Command Window.)

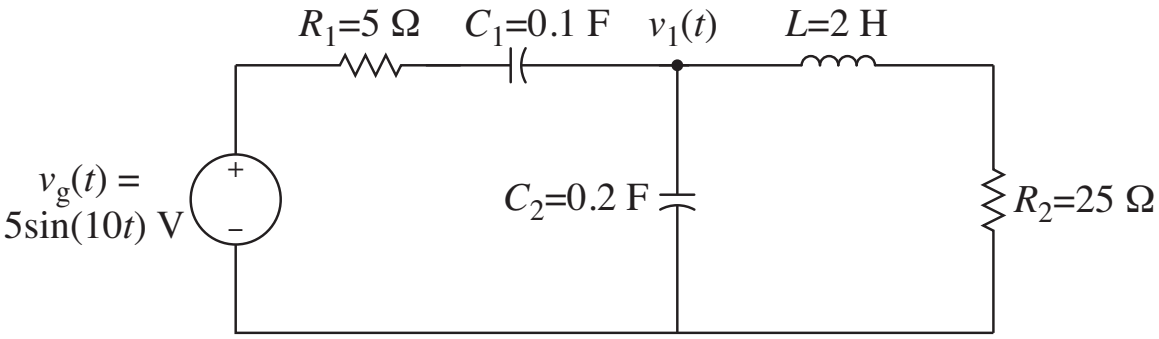
```
>> % Unlike with cos(pi/2), here we do get the exact result!
>> cosd(90)
ans =
     0
>> sind(45)
ans =
    0.7071
>> help i
 i  Imaginary unit.
    As the basic imaginary unit SQRT(-1), i and j are used to enter
    complex numbers.  For example, the expressions 3+2i, 3+2*i, 3+2j,
    3+2*j and 3+2*sqrt(-1) all have the same value.

    Since both i and j are functions, they can be overridden and used
```

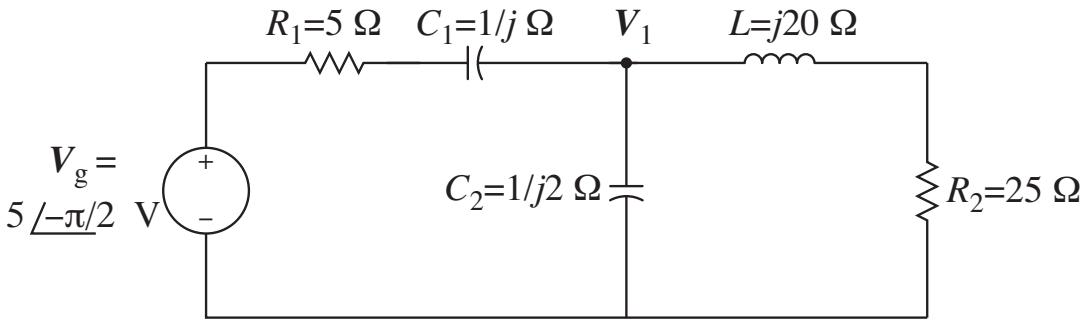
```
    as a variable.  This permits you to use i or j as an index in FOR
    loops, etc.
>> help cosd
 cosd   Cosine of argument in degrees.
    cosd(X) is the cosine of the elements of X, expressed in degrees.
    For odd integers n, cosd(n*90) is exactly zero, whereas cos(n*pi/2)
    reflects the accuracy of the floating point value for pi.
```

Sinusoidal Steady State Analysis

Consider the following circuits which is assumed to be in the sinusoidal steady state.



Our goal is to obtain the voltage $v_1(t)$. To accomplish this, we first redraw the circuit in the frequency domain, showing the impedance of each element. As a reminder, the impedance of a resistor is its resistance, the impedance of an inductor is $j\omega L$, and the impedance of a capacitor is $1/(j\omega C)$. In this particular case, we see from the source that ω is 10 rad/s. Also, for convenient conversion to the frequency domain, we can rewrite the source as $v_g = 5\cos(10t - \pi/2)$. The circuit now becomes:



To find the phasor $\mathbf{V}_1$, we can use voltage division. Let's say the series combination of R_1 and C_1 is the impedance Z_1 while the inductor, R_2, and C_2 form the impedance Z_2. The voltage $\mathbf{V}_1$ is then given simply by

$$\mathbf{V}_1 = \frac{Z_2}{Z_1 + Z_2}\mathbf{V}_g. \tag{1}$$

Noting that $\mathbf{V}_g = 5\angle -\pi/2 = -j5$, let's turn to *MATLAB* to do the dirty work. The following illustrates one approach to obtaining a solution.

```
>> % z1 is the series impedance of R1 and C1.
>> z1 = 5 + 1/1j;
>> % Let zc be the impedance of C2 and zlr be the impedance of
>> % of the series combination of the inductor and R2.
>> zc = 1/2j;
>> zlr = 25 + 20j;
>> % Calculate z2 via the reciprocal of the reciprocals.
>> z2 = 1 / (1/zc + 1/zlr);
>> % Define the source voltage.
>> vg = -5j;
>> % Now obtain the desired voltage V1.
>> v1 = z1 / (z1 + z2) * vg
v1 =
  -0.4607 - 0.1447i
>> % Obtain the magnitude and phase.
>> abs(v1)
ans =
    0.4829
>> angle(v1)
ans =
   -2.8373
```

We're done! We now know the phasor voltage $\mathbf{V}_1$ is $0.4829\underline{/-2.8373}$ V. Converting this to the time domain, we obtain $v_1(t) = 0.4829\cos(10t - 2.8373)$ V.

Note: Everything above was presented as having been entered into the Command Window. You will likely find that, if you ever need to change anything (such as may happen when you discover you've entered a typo), your life will be much easier if you enter the results in the "Live Editor." When you do that, nothing happens until you click the "Run" button. Once you do that, *MATLAB* executes the commands you wrote "all at once." The nice thing is if you need to change anything, you simple make the desired changes and click "Run" again.

Plotting the Results

Let's assume we also want to plot the results of our calculation. First, let's determine the period of the source function. The frequency is $\omega = 2\pi f = 10$ rad/s. Thus, the frequency f is $10/(2\pi) = 1.5915$ cycles/second (i.e., in one second the source passes through 1.5915 cycles). The reciprocal of the frequency is the period, which we typical write as T. Hence, we have $T = 1/f = 0.6283$ s. Let's plot the source over three periods going from $-T$ to $2T$.

In *MATLAB* we can use the `plot()` command to create an x-y plot where we have to provide two "vectors" (arrays of numbers) corresponding to x values and y values. Here we want the x values to be evenly spaced time samples between $-T$ and $2T$. We can use the `linspace()` command to provide these. This take three arguments: the starting point, the ending point, and the total number of samples. The following demonstrates the use of `linspace()` to generate eight values between -2 and 4.

```
>> linspace(-2, 4, 8) % Start = -2; end = 4; number of points = 8.
ans =
   -2.0000   -1.1429   -0.2857   0.5714   1.4286   2.2857   3.1429   4.0000
```

With that background, and keeping in mind we can use a semicolon to suppress output, let's generate 200 time samples between $-T$ and $2T$. Note that *MATLAB* is case sensitive so that t and T are different variables.[2]

```
>> T = 0.6283;   % Period.
>> t = linspace(-T, 2*T, 200);   % An array of 200 time samples.
```

We can now use the `sin()` function to obtain the voltages $v_g(t)$ that correspond to each of these time samples. *MATLAB* will distribute the sine function over each of the time samples and put the result in a new array that we can store to a variable. Once we have that, we can simply call `plot()` with the "x" and "y" arrays. The following will generate the desired plot.

```
>> vg_time = 5 * sin(10 * t);
>> plot(t, vg_time)
```

We're calling the voltages here `vg_time` because these are the time-domain voltages rather than the phasor value `vg` used above. Rather than showing you the result of these commands, let's first tweak things to make the plot a bit prettier. We can add labels to the x and y axes with the `xlabel()` and `ylabel()` commands. The arguments are the strings we want to have associated with the x and y axes. Strings in *MATLAB* are enclosed in single quotes.[3]

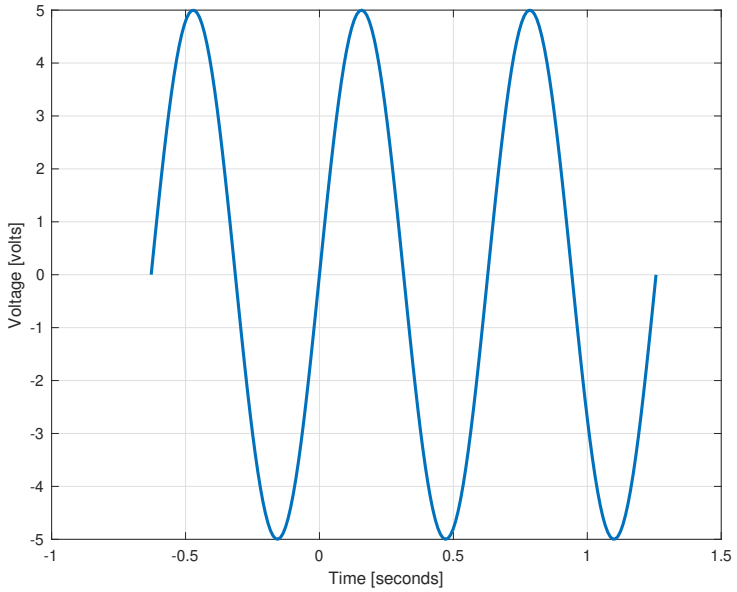
We can have the plot display a grid by issuing the command "`grid on`". Finally, we can modify the "properties" of the plot by providing additional arguments to the `plot()` command where we provide the property name (as a string) and its corresponding value. We can make the plot line thicker via the `LineWidth` property. With this background, let's revisit the previous two commands and issue these commands instead:

```
>> vg_time = 5 * sin(10 * t);
>> plot(t, vg_time, 'LineWidth', 2)
>> xlabel('Time [seconds]')
>> ylabel('Voltage [volts]')
>> grid on
```

This produces the figure shown below.

[2]Best practices would dictate that we use longer and more descriptive variable names, i.e., rather than `T` and `t`, we should use names such as `period` and `t_samples`. But we'll momentarily ignore best practices.

[3]In more recent versions of *MATLAB* you can also enclose strings in double quotes. Technically, single quotes create an $N \times 1$ character array where N is the number of characters in the string while double quotes create a 1×1 string array, but none of that matters to us and I'll stick with single quotes.



Of course, plotting the source function isn't very interesting because we knew that all along. So, let's add the voltage $v_1(t)$ to the plot. If we want to plot multiple lines on the same plot, we issue the command "`hold on`" (we can turn off this feature with "`hold off`").

You will notice that our plot window extended beyond the times for which we had results. We can manually control the extend of the plot using the `axis()` function that has an argument that is an array that specifies four values: the minimum and maximum values in x and the minimum and maximum values in y. The syntax is a little mysterious for now, but because the argument is an array, you have to enclose the values in square brackets ("`[···]`").

Don't forget that you can use `help` to obtain a load of information about *MATLAB*'s various functions. Looking at the `help` for `plot()`, you will see that we can also change a plot's color by providing yet another argument. The color (and line style) is specified rather cryptically via a short string, but `help` provides all the information you need to decipher things. For now, we'll simply note that `'r'` produces a red plot.

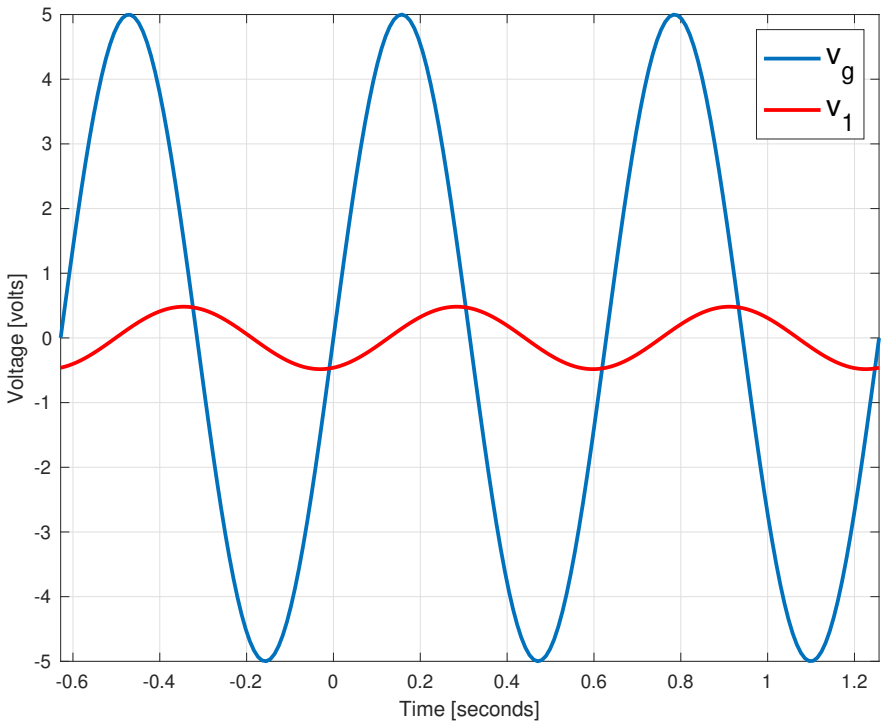
We can add a legend to the plot with the command `legend()` that takes a string argument describing each plot. We can control the font size for the legend with the `'FontSize'` property (I find the default font size too be too small).

The following puts all this information together to generated a plot of both the input voltage and the voltage v_1. We assume that the phase `v1` has been calculated as shown previously and do not repeat those commands here.

```
>> % Plot the source function over three periods.
>> T = 0.6283;
>> t = linspace(-T, 2*T, 200);
>> vg_time = 5 * sin(10 * t);
>> plot(t, vg_time, 'LineWidth', 2)
>> xlabel('Time [seconds]')
>> ylabel('Voltage [volts]')
>> grid on
```

```
>> axis([-T, 2*T, -5, 5])
>> % Obtain the magnitude and phase of v1.
>> mag = abs(v1);
>> phase = angle(v1);
>> % Generate the time-domain values of v1 and plot
>> % the results on the same plot as vg.
>> v1_time = mag * cos(10 * t + phase);
>> hold on
>> plot(t, v1_time, 'r', 'LineWidth', 2)
>> % Add a legend with an enlarged font.
>> legend('v_g', 'v_1', 'FontSize', 16)
```

These commands produce the following plot where you see the voltage v_1 is significant smaller than v_g and, as we know from the previous calculations, significantly out of phase.



Again, you will find life much more pleasant when doing things like this (i.e., plot generation and the like) if you issue the commands in the "Live Editor." This will allow you to easily make changes that might otherwise be very cumbersome to realize in the "Command Window."