San Francisco State University

Engineering 315

Laboratory #1 - Introduction to Matlab

Outline

1. Purpose

This is a brief tutorial on Matlab to help you get started. You are basically responsible for learning this language on your own, but there's a lot of information available to help you.

- The Matlab Reference Manual that comes with the Student Edition of Matlab is very good.
- Matlab's online help and tutorial functions are always there for you.
- A variety of books are available, most of which don't really add much more than Matlab's reference manual for the basics. However, they are useful as a source of examples and for more advanced material. As far as I'm concerned, the best of the lot is: Mastering MATLAB 6: A Comprehensive Tutorial and Reference by Duane C. Hanselman, Bruce C. Littlefield

This lab gives you an overview of Matlab basics. We then discuss how to represent continuous-time signals, such as those that we work with in ENGR 305, with digital computers, which are fundamentally discrete-time systems. Finally, we give you some exercises to show your understanding of the material.

2. Matlab Basics

What is Matlab?

Matlab is a program for scientific applications. It is many things at once -- a calculator, a programming language, and a sophisticated system for displaying and analyzing data. It is used extensively in industry to prototype sophisticated applications in many fields. All our laboratory exercises in this course will be based on Matlab.

Calculator functions
At its simplest, Matlab can be used as a calculator

```
» 1+1
ans =
    2
```

In this tutorial, Matlab's output is given in blue type, and your input is given in green. The caret character, », is provided by Matlab at the beginning of every line in the command window, and means that Matlab is ready for input.

More complex algebraic calculations can also be done:

```
» (1+2^3)/(6-3 * (4/2-1))
ans =
    3
```

The usual rules of precedence apply in Matlab to complex expressions such as this: exponentiation is done first, multiplication and division are next (and equal) in precedence and are evaluated left to right. Addition and subtraction are lowest. Parenthesis can be used to over-ride these usual rules of precedence.

Variables
In addition to using Matlab as a calculator, you can set and manipulate variables:

```
» x = 3
x =
    3

» y = 2
y =
    2

» z = x + y
z =
    5
```

Note that Matlab echoes your work each time you type return. You can suppress this echoing by terminating each line with a semicolon

```
» x = 3;
» y = 2;
» z = x + y;
» z
z =
    5
```

The last line shows that just typing the variable name, Matlab gives us the value of the requested variable. You can also get Matlab to display a variable's value less verbosely:

```
» disp(z)
    5
```

If you type the name of a variable that you have not previously set, Matlab will complain

```
» q
??? Undefined function or variable 'q'.
```

Matlab keeps all your variables around until you clear them with the `clear` command. You can clear all variables at once or just specific variables.

You can query all your variables in the current Matlab workspace:

```
» who
Your variables are:
ans    x    y    z
```

You can also get a more involved display:

```
» whos
  Name      Size         Bytes  Class
  ans       1x1              8  double array
  x         1x1              8  double array
  y         1x1              8  double array
  z         1x1              8  double array
Grand total is 3 elements using 24 bytes
```

ans is a built-in Matlab variable that gives the result of the last computation. You can use `ans` just like any other variable, and it is replaced with each successive computation

```
» ans
ans =
    3

» ans + 2
ans =
    5
```

There are other reserved Matlab variables, such as `i` and `j` (see below) and `pi`.

```
» pi
ans =
    3.1416
```

Matlab is not a strongly typed language, in the sense that C is. That is, you don't have to declare variables as floating point or integer. By default, all variables and calculations in Matlab are done in double precision and displayed in a format specified by the Matlab format command. By default, the format is short floating point. That's why the value of pi is a bit truncated. That's easy to fix:

```
» format long
» pi
ans =
    3.14159265358979
```

Matlab is inherently capable of working with complex numbers. The reserved variables `i` and `j` are used to denote the imaginary part. All operations (i.e. multiplication, division, addition and subtraction) work transparently with complex numbers

```
» x = 1 + 2 * j
x =
    1.0000+ 2.0000i

» y = 2 - j
y =
    2.0000- 1.0000i

» z = x + y
z =
    3.0000+ 1.0000i
```

All Matlab's arithmetic functions also work with complex numbers:

```
» exp(z)
ans =
  10.8523 +16.9014i
```

You can use `j` (or any other built-in Matlab variable) as a variable in your own program by reassigning its value, but then it will no longer denote the imaginary operator:

```
» j = 2;
» x = 1 + 2 * j
x =
    5
```

Ooops! You can reset a built-in Matlab variable such as `j` to its default value by clearing it:

```
» clear j
» j
ans =
    0 + 1.0000i
```

You can also use `clear` to delete other variables and functions that you have defined:

```
» who
Your variables are:
ans        x          y          z
» clear y z
» who
Your variables are:
ans        x
```

Vectors and matrices

One of Matlab's most powerful features is the ability to express vectors and matrices compactly and efficiently. Vectors (or arrays) in Matlab are simply declared:

```
» x1 = [1 2 3 4 5]
x1 =
     1     2     3     4     5
```

`x1` is a row vector. A space (or coma) separates elements of a row vector.

```
» x2 = [1; 2; 3; 4; 5]
x2 =
    1
    2
    3
    4
    5
```

`x2` is a column vector. A semicolon separates elements of a column vector. The transpose operator converts row vectors to column vectors

```
» x3 = x2'
x3 =
    1     2     3     4     5
```

Be *very* careful when taking the transpose of complex quantities, since the transpose operator performs the complex conjugate:

```
» xc = [1+j 2+j 3+j]
xc =
   1.0000 + 1.0000i   2.0000 + 1.0000i   3.0000 + 1.0000i

» xc'
ans =
    1.0000 - 1.0000i
    2.0000 - 1.0000i
    3.0000 - 1.0000i
```

To get a non-conjugate transpose, use the dot-transpose operator:

```
» xc.'
ans =
    1.0000 + 1.0000i
    2.0000 + 1.0000i
    3.0000 + 1.0000i
```

Matlab gives you an easy, shorthand way to generate sequences like `x1`, above:

```
» x1 = 1:5
x1 =
    1     2     3     4     5
```

You can increment by more than one

```
» y1 = 1:2:9
y1 =
    1     3     5     7     9
```

or even negatively

```
» y2 = 9:-2:1
y2 =
    9     7     5     3     1
```

or negatively

```
» y2 = 9:-2:1
y2 =
    9     7     5     3     1
```

or in non-integer increments

```
» t = 0:0.5:2
t =
    0    0.5000    1.0000    1.5000    2.0000
```

Here's another way to do the preceding

```
» t = linspace(0, 2, 5)
t =
   0    0.5000    1.0000    1.5000    2.0000
```

The `linspace` command creates an array of a requested number of points (in this case, 5) spanning a given range (in this case, betwen 0 and 2 inclusive).

All of Matlab's arithmetic functions work with vectors and matrices. For example, to add two vectors of the same dimension:

```
» q = x1 + x3
q =
    2     4     6     8    10
```

Adding vectors with different dimensions will produce an error (try it).

Vector multiplication is simple. Multiplying a row vector by a column vector produces a scalar.

```
» z = x1 * x2
z =    55
```

Multiplying a column vector by a row vector produces a matrix.

```
» z = x2 * x1
z =
     1     2     3     4     5
     2     4     6     8    10
     3     6     9    12    15
     4     8    12    16    20
     5    10    15    20    25
```

In Matlab, arrays are just matrices that have only a single row or column. You can multiply two equally sized arrays or matrices together element by element by using the `.*` operator:

```
» y1 .* y2
ans =
     9     21     25     21      9
```

You can exponentiate an array by using the `.^` operator:

```
» y1 .^ 2
ans =
     1     9     25     49     81
```

Accessing array values
You can access any value of an array or vector or matrix with subscripts of the general form `z(row, column)`. So, using the examples above,

```
» q(2)
ans =
    4


» z(2, 3)
ans =
    6
```

It is important to note that Matlab considers index 1 to be the first value of the array (unlike the C language, for example). Trying to declare or access array values with non-positive integers produces an error:

```
» q(0)
??? Index into matrix is negative or zero.  See release notes on changes to
logical indices.
```

This is sort of a bummer for many operations in fields like digital signal processing, since we would like to be able to manipulate arrays with zero or negative indices.

In Matlab, you can use sequences as subscripts as well, to select portions of arrays or matrices:

```
» q(1:3)
ans =
    2     4     6

» z(2:4, :)
ans =
     2     4     6     8    10
     3     6     9    12    15
     4     8    12    16    20
```

In the preceding example, the syntax says "give me rows 2 through 4 and all the columns". Using a colon by itself as an index tells Matlab that you want all the columns. You can do very sophisticated things with the array indices, such as replacing or manipulating portions of vectors and matrices. Here we replace rows 2 to 4 in columns 3 and 5 with the scalar value 99:

```
» z(2:4, [3 5]) = 99
z =
     1     2     3     4     5
     2     4    99     8    99
     3     6    99    12    99
     4     8    99    16    99
     5    10    15    20    25
```

Matlab allows you to use reversed indices to index into an array. For example

```
» q(end:-1:1)
ans =
    10     8     6     4     2
```

Note that `end` is a Matlab keyword that, when used as an array argument, denotes the last value in the sequence. Also, `length` or `size` can be used to give the size of an array.

Matlab gives you the ability to produce arrays and matrices with all zeros or all ones using the `zeros` or `ones` commands. For example, here we produce an array of one row and four columns:

```
» zeros(1, 4)
ans =
     0     0     0     0
```

You can concatenate two horizontal arrays, `x` and `y`, into one longer array by using `[x y]`. For example, to pad the beginning or end of a sequence with zeros, concatenate that sequence with a sequence of zeros made with the `zeros` command.

```
» [zeros(1, 2) [1 2 3] zeros(1, 3)]
ans =
     0     0     1     2     3     0     0     0
```

pads the beginning of the array with two zeros and the end with three zeros. Note that `zeros(1, 3)`is not the same as `zeros(3, 1).` Note that `zeros(1, 0)` will add no zeros.

As an example of how you use complicated indexing, we'll create a matrix by replicating a single row several times.

```
» r = 1:5;
» s = r([1 1 1], :)
s =
     1     2     3     4     5
     1     2     3     4     5
     1     2     3     4     5
```

The syntax here is pretty tricky. It says, "make a matrix with row one and all columns repeated three times". You could also do this with the following syntax, which is more convenient for a larger number of rows:

```
» s = r(ones(1, 3), :)
```

Matlab has a number of commands, such as `reshape` and `repmat` to reshape and replicate matrices.

Logical operations
Matlab can handle logical as well as arithmetic operations. Relational operaters like `>`, `>=`, `<`, `<=`, `==` and `~=` allow one to make logical comparisons of variables and constants. For example:

```
» r = 1:5;
» s = (r > 2)
s =
     0     0     1     1     1
```

The output, `s`, is an array of the same dimensions as the input array, `r`. Each element of the output array is computed by comparing the corresponding element of the input array to 2. If the input element is greater than 2, the output element is 1; otherwise, the output element is zero. You can do a lot of fancy things using logical operations, such as using the result of relational operations to index arrays. For example, here's how you would select only those values of r for r > 2:

```
» r(r>2)
ans =
     3     4     5
```

You should look up other logical operators, such as `any` and `all`. The related command, `find` is also particularly valuable for indexing arrays.

Matlab Help
There are a number of ways of getting help on anything in Matlab. From the command line, use the `help` function,

```
help
```

By itself, `help` gives you a list of topics (try it) and lists all the subtopics on which Matlab offers help. You can then get help on any subtopic, for example,

```
help elfun
```

gives all the elementary functions.

Built-in functions
Matlab includes an impressive array of built-in functions. To get help on any given function, type `help` and the name of the function, for example:

```
» help sin
SIN Sine.
SIN(X) is the sine of the elements of X.
```

helpwin is equivalent to `help`, but pops up its own window. You can activate Matlab's extensive HTML-based help system including the complete Matlab

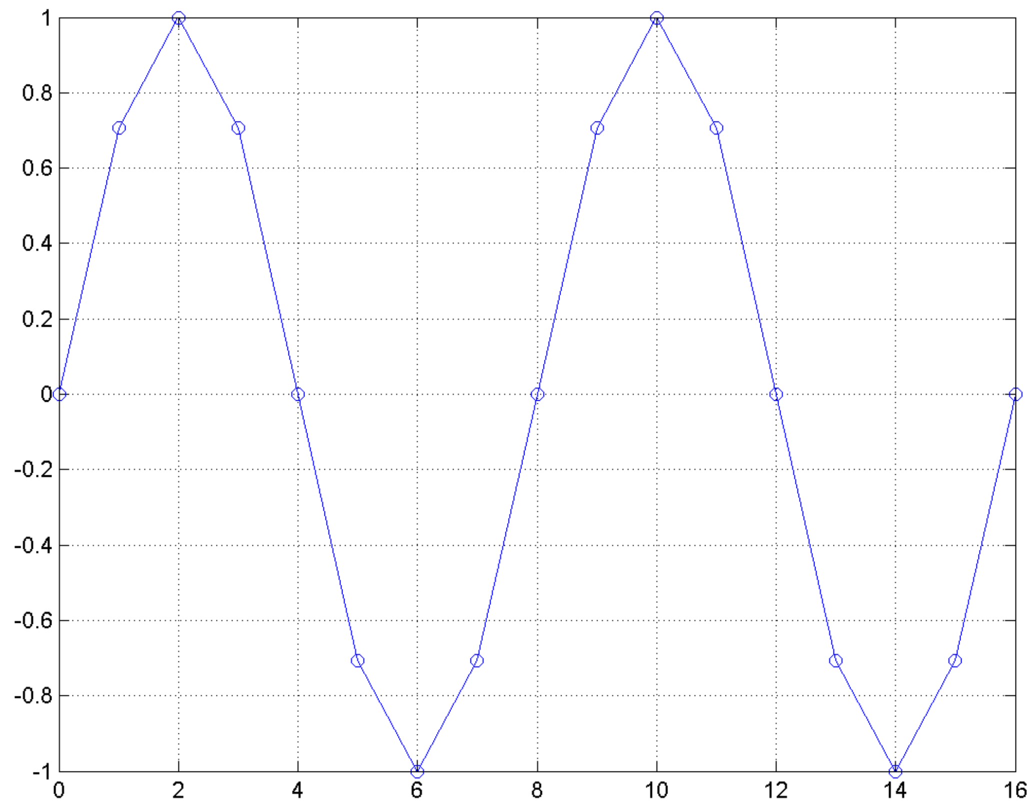
documentation from the 'Help' menu of the command menu or by typing `helpdesk` or `doc` in the command window.
If you don't know what the name of the function is, search in HTML documentation using the 'Search' or 'Index' tabs, or you can type `lookfor` from the command window. For example,

```
» lookfor fourier
contents.m: % Data analysis and Fourier transforms.
FFT Discrete Fourier transform.
FFT2 Two-dimensional discrete Fourier Transform.
FFTN N-dimensional discrete Fourier Transform.
IFFT Inverse discrete Fourier transform.
IFFT2 Two-dimensional inverse discrete Fourier transform.
IFFTN N-dimensional inverse discrete Fourier transform.
XFOURIER Graphics demo of Fourier series expansion.
```

gives you all of Matlab's Fourier transformation functions.

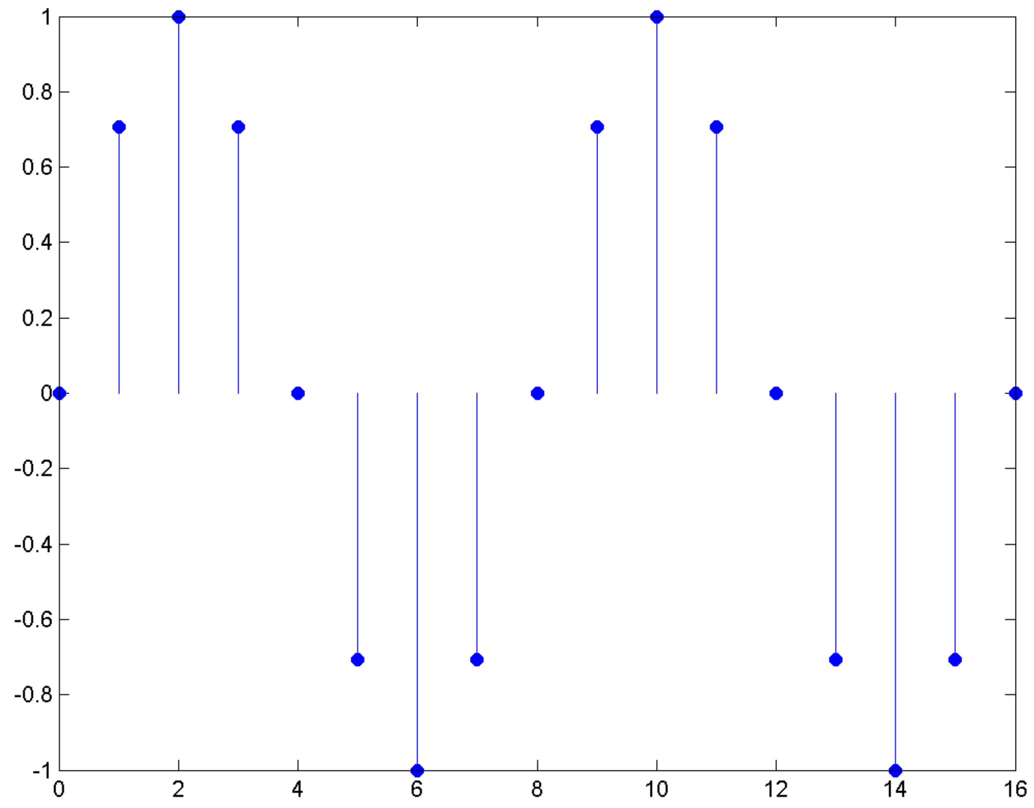
Plotting
Matlab provides many powerful and easy-to-use plotting functions. The most useful for us are `plot` and `stem`. `plot` creates two-dimensional graphs with connected lines and `stem` creates number-line style graphs. For example:

```
» t = 0:31;
» y = sin(2 * pi * t / 8);
» plot(t, y, 'bo-')
```



plots ordinate variable `y` against abscissa variable `i` using a continuous blue line plus a circle for each data point. Extremely sophisticated control of plot attributes, such as colors, line styles, symbols, text and axes, as well as multiple and overlaid plots are possible with Matlab's plotting functions. Type `help plot` for more info.
A `stem` plot of the same data looks like this:

```
» stem(t, y, ‘filled’)
```



Representing continuous-time functions in Matlab
Of primary importance to us is the ability to represent *continuous-time functions*, that is, functions of a continuous-time variable. For example, if t is continuous-time variable, its value is not quantized but continuous, which means that it can take on any value between $-\infty$ and $+\infty$. So, $\sin 2\pi t$ exists for any value of *t.* , such as $t = 1$ or $t = 2$ or any value of t in between, such as $t = 1.2142657\cdots$ Whereas continuous-time functions can be evaluated at any value of time, functions of a discrete-time variable can only be evaluated at integer values of the time variable. For example, the discrete-time function *sin[n]* only has value at integer values of *n*, that is, for $n = 1$, $n = 2$ …The value of $\sin[n]$ for non-integer value of *n*, such as $n = 1.2142657\cdots$, is undefined.

The distinction between continuous-time and discrete-time functions is an important one. In ENGR 305, we deal almost exclusively with continuous-time signals and systems. In ENGR 451, we deal with discrete-time signals and systems, so we need to represent both types of functions. Matlab can only numerically deal with discrete-time functions. In order to represent continuous-time systems, we must resort to quantizing the time axis in very small increments. For example, to quantize the time axis in increments of one msec over the range of 0 to 2 secs, we could create a time variable, *t*, in either of the following two equivalent ways:

```
» t = 0:0.001:2;
```

or

```
» t = linspace(0, 2, 2001);
```

(Incidentally, why 2001 points?). To create an 8-kHz sin wave sampled at 1 msec time intervals, we just do

```
» y = sin(2 * pi * 8000 * t);
```

We obviously have to be careful to sample any given signal 'fast enough', that is, to quantize the time variable sufficiently finely. For example, plot what would happen if we tried to create a 1-kHz sine wave with the same *t* specified above.

Matlab scripts and functions

One of Matlab's great strengths is that it allows you to create your own scripts and functions.
A script is just a sequence of operations that Matlab executes sequentially. Create a script using your favorite word processor, or invoke Matlab's own editor by typing `edit` on the command line. When you finish entering your script, save it as a .m file in your home directory or in a directory that is on Matlab's search path. For example here is my file `myscript.m`:

```
% MYSCRIPT A little script to add two numbers
%
% T. Holton
x = 3;
y = 5;
z = x + y
```

Now, when I type `myscript` following the Matlab prompt, I get the answer of this very complex calculation:

```
» myscript
z =
    8
```

Note that the first few lines of the script start with the `%` character. This denotes a comment. It is good practice to comment your programs, giving the name of the program, the author and perhaps the revision history. If you put comments like this in the first few lines of the program, then your script is automatically accessible from the Matlab's `help` function:

```
» help myscript
MYSCRIPT A little script to add two numbers

T. Holton
```

All the variables set in a script are available in the current Matlab workspace, that is to say, they are effectively available to any other command or script that executes from the command line. This may be an advantage in some situations, but most often it's not. You really like the script to work as an independent unit, perhaps one that takes one or more inputs and produces one or more outputs and doesn't litter your workspace with variables. This is precisely what Matlab's functions are for. Matlab's user-defined functions are like scripts except that all calculations take place in a separate workspace (like those in other languages such as C). All variables declared in the function are local in scope, that is, only visible within the function. We make functions just like scripts:

```
function out = myfunction(in1, in2)
```

```
% MYFUNCTION A little function to add two numbers
%
% T. Holton
temp = 2 * in1;
out = temp + in2; % this is a comment
return
```

The first line of the function specifies the input and output variables. In this case, `out` is the output variable and `in1` and `in2` are the input variables. The last line of the function is the return statement. When I call this function

```
» clear
» y = myfunction(3, 4)
y =
    10
» who
Your variables are:
y
```

Because all lines in `myfunction` are terminated with semi-colons, all printing of calculations from within the function is suppressed. It is important to understand that Matlab did all the calculations required by the function in a separate workspace that is normally not accessible to you. None of the variables that we created in `myfunction` will show up in the calling workspace. If you ask for variables in the function's workspace, you will be out of luck:

```
» in1
??? Undefined function or variable 'in1'.
```

There are ways of sharing variables between workspaces in Matlab if absolutely necessary; specifically, variables can be declared `global` or one can use Matlab's built-in `assignin` function. However, extensive use of global variables is generally considered a hallmark of bad program design. It defeats the purpose of using functions, which is to isolate variables in different workspaces so that they can't interact with variables in the main workspace or with other functions in unforeseen ways.

3. Basic signals and functions

In this course, almost all your work will be directed at creating Matlab functions to perform various tasks. This week's assignment is to help you become familiar with some of these Matlab operations.

You will learn how to create basic signals in Matlab, and then write a series of functions to perform elementary operations on these signals such as flipping, shifting and adding.

Basic signals

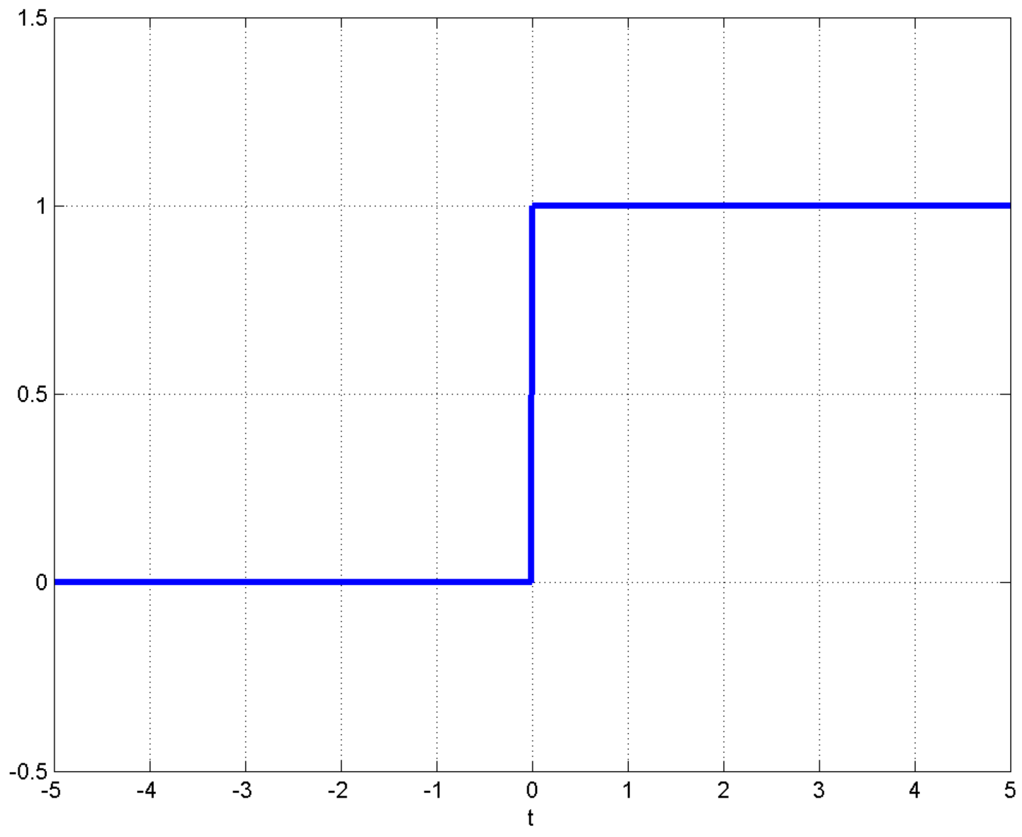
Basic signals are things like impulses, steps and ramps. It is not possible to represent an impulse well in Matlab since it is infinitely narrow and infinitely high. However, we can approximate a step function pretty well. Here is one possible definition, which uses relational operations:

```
function y = u(t)
%   U  Step function
%      y = u(t) gives result 0 for t < 0 and 1 for t >= 0
%
%      T. Holton
y = (t >= 0);
return
```

Now, lets create a "continuous" time variable which spans the range from -5 to 5 secs with a quantization of 1 msec and plot the step. Here's the entire program:

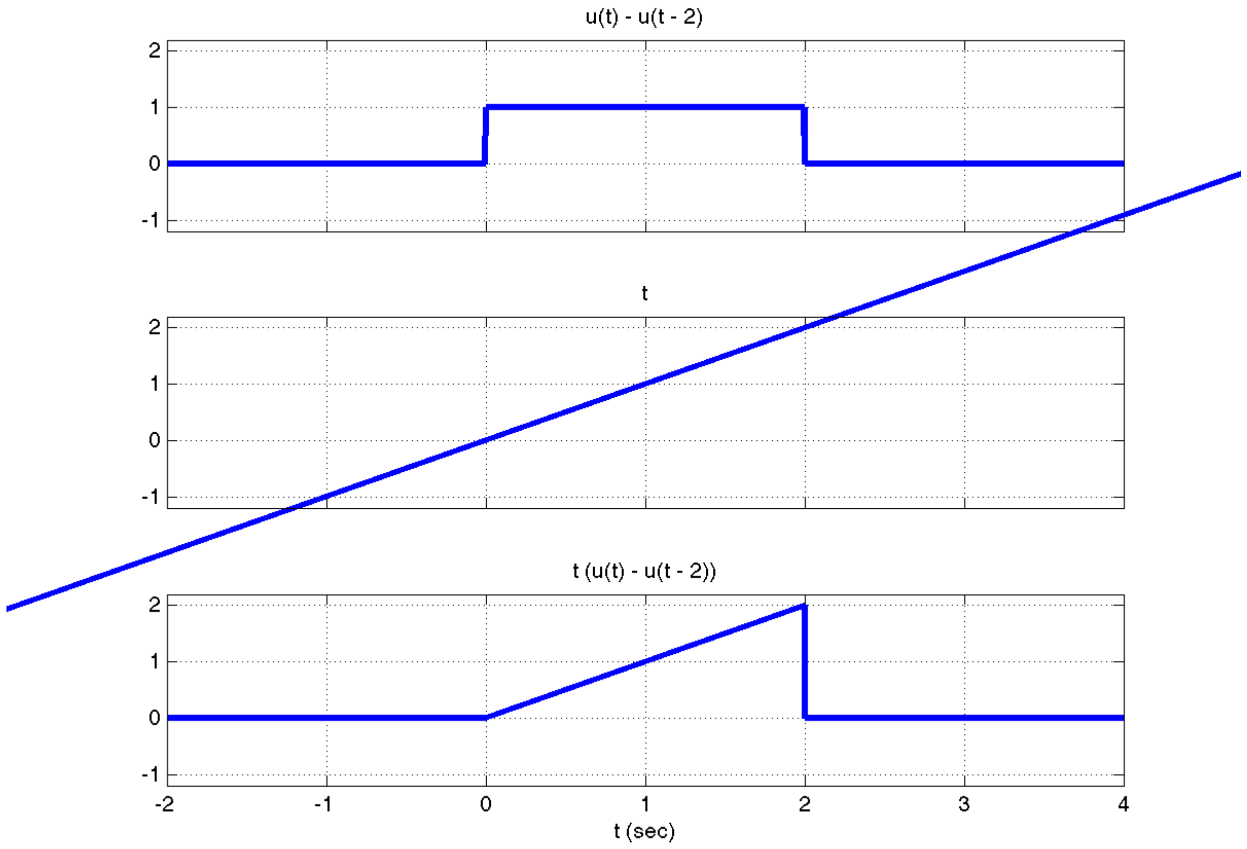
```
t = -5:0.001:5;
figure(clf);
plot(t, u(t), 'LineWidth', 3);
xlabel('t (sec)');
grid on;
ylim([-0.5 1.5]);
```

Here's the result of running it:



Note that I have used the `ylim` command to scale the y-axis to the range -0.5 to 1.5. Other useful related axis scaling commands are `xlim` and `axis`.

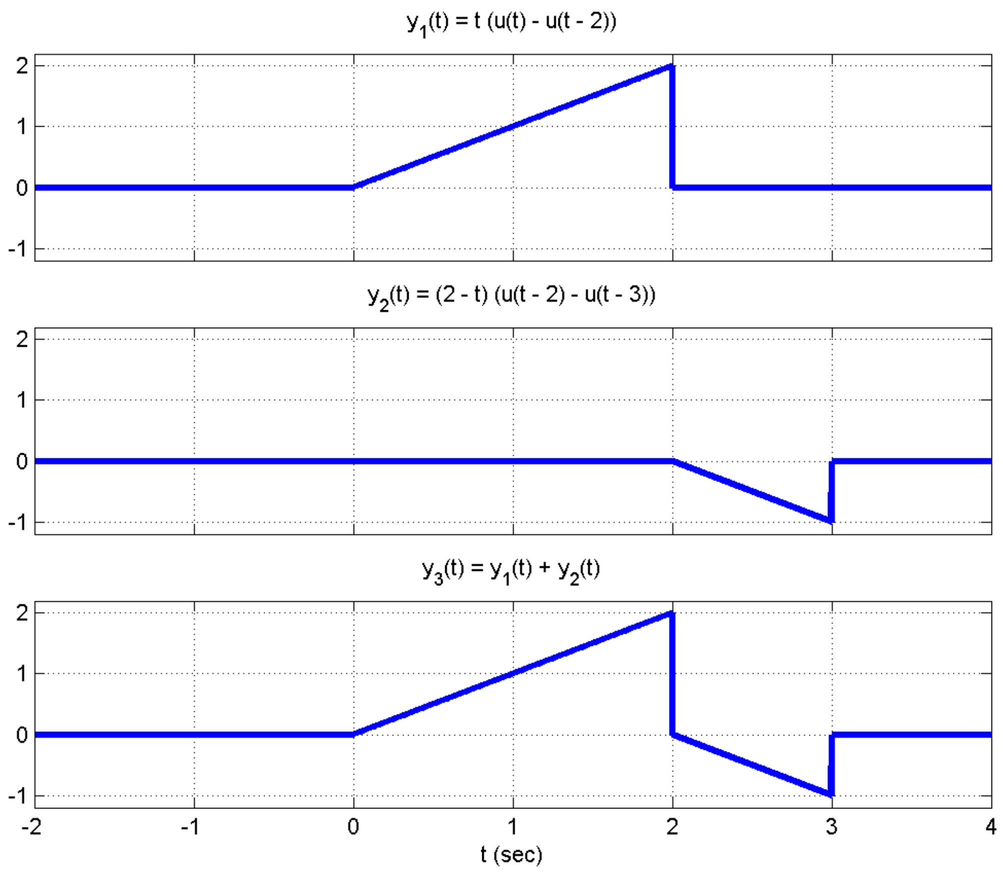
Construction of piecewise-continuous functions
Using the step as a basic building block, one can create and plot a variety of piecewise-continuous functions.



For example, the top panel shows a finite-duration time pulse starting at $t = 0$ and extending to $t = 2$. This pulse is created by taking the difference of a step and a shifted step: $u(t) - u(t-2)$. The pulse has the property that it is one in the interval between $t = 0$ and $t = 2$ and zero outside this interval. Hence, if we multiply an arbitrary time function, $q(t)$, by this pulse, we multiply $q(t)$ function by 1 in the region between $t = 0$ and $t = 2$ (which leaves the function unchanged), and multiply $q(t)$ by zero outside this region (which sets the result to zero). By multiplying an arbitrary function by a pulse, we can effectively select only the portion of the time function we desire. To give a specific example, the second panel shows the line $q(t) = t$. This line has non-zero value over the entire range from $-\infty$ to $+\infty$ (except of course exactly at $t = 0$). The lower panel shows the result of multiplying $q(t)$ by the pulse, yielding a short non-zero line segment described by the equation

$$y(t) = t\big(u(t) - u(t-2)\big) = \begin{cases} t, & 0 < t < 2 \\ 0, & \text{otherwise} \end{cases}$$

We can create functions of high complexity by adding together different segments, each segment created by multiplying a different time function with a time-shifted pulse. For example,



The first panel shows $y_1(t) = t\left(u(t) - u(t-2)\right)$, the line segment we produced in the previous picture. This segment is non-zero only between $t = 0$ and $t = 2$. The second panel shows another line segment, $y_2(t) = (2-t)\left(u(t-2) - u(t-3)\right)$, which is non-zero only between $t = 2$ and $t = 3$ .The lower panel shows the sum of the two line segments, $y_1(t) + y_2(t)$. Because the two segments are disjoint (i.e. non-overlapping) in time, the sum is equivalent to joining together the contiguously non-zero parts of the two segments. We can also simplify the equation of the sum by combining like terms (i.e. all terms which contain $u(t-2)$):

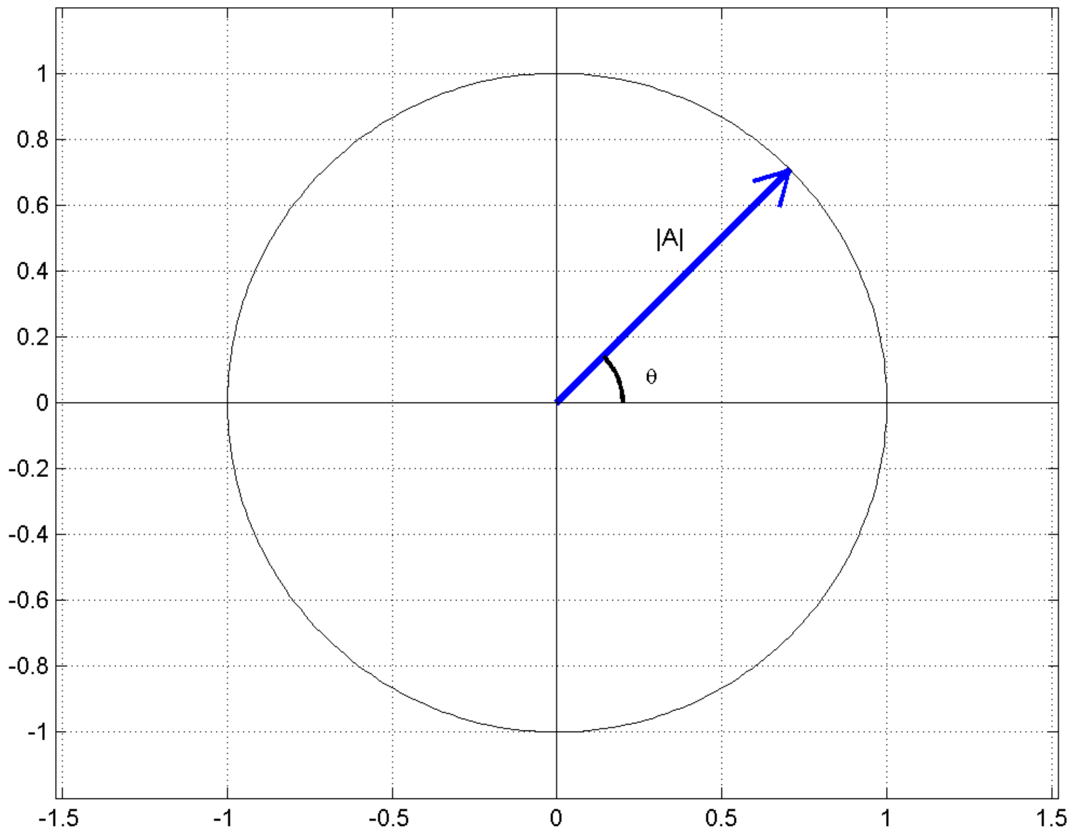
$$\begin{aligned} y(t) &= t\left(u(t) - u(t-2)\right) + (2-t)\left(u(t-2) - u(t-3)\right) \\ &= tu(t) + 2(1-t)u(t-2) - (2-t)u(t-3) \end{aligned}$$

Complex exponentials

Perhaps the important class of functions we will discuss in this course are the complex exponential time functions. These are functions of the form $Ae^{j\omega t}$, where *A* is a constant, ω is a frequency in radians/sec and *t* is the time in seconds. *A* may be complex, so we can write it in polar form: $A = |A| e^{j\measuredangle A}$, where $|A|$ is the magnitude, which is always non-negative (i.e. $|A| \geq 0$), and $\measuredangle A$ is the phase angle, expressed in radians ranging from *0* to2π. Hence,

$$Ae^{j\omega t} = \left(|A| e^{j\measuredangle A}\right) e^{j\omega t} = |A| e^{j\omega t + \measuredangle A} .$$

This last form shows that we can express an arbitrary complex exponential time function as function of three parameters: magnitude, radial frequency and phase. The way to visualize this time function is as a vector in the complex exponential plane:



The length of the vector is the magnitude, $|A|$. When $t=0$, the vector rests at angle $\theta = \measuredangle A$ with respect to the real axis. As t increases, the vector rotates around the center with an angle, $\theta = \omega t + \measuredangle A$, that increases linearly with time. If ω is large it rotates fast; if ω is small it rotates slowly. If ω is positive it rotates counterclockwise, if ω is negative it rotates clockwise.
Recall the De Moive relations:

$$e^{j\omega t} = \text{Re}\left\{e^{j\omega t}\right\} + j\,\text{Im}\left\{e^{j\omega t}\right\} = \cos\omega t + j\sin\omega t$$

and

$$\begin{aligned} e^{-j\omega t} &= \text{Re}\left\{e^{-j\omega t}\right\} + j\,\text{Im}\left\{e^{-j\omega t}\right\} = \cos(-\omega t) + j\sin(-\omega t) \\ &= \cos(\omega t) - j\sin(\omega t) \end{aligned}.$$

Using these relations, you can derive the Euler relations relating *cos* and *sin* to the sum of complex exponentials

$$\cos\omega t = \text{Re}\left\{e^{j\omega t}\right\} = \frac{e^{j\omega t} + e^{-j\omega t}}{2}$$

and

$$\sin \omega t = \operatorname{Im}\left\{e^{j\omega t}\right\} = \frac{e^{j\omega t} - e^{-j\omega t}}{2j}.$$

Thus endeth this little Matlab tutorial. Now on to your assignment.

4. Assignment

In this assignment, you are going to write four Matlab functions, `lab1_1.m`, `lab1_2.m`, `lab1_3.m`, and `lab1_4.m` to plot various things. Then, you will download `lab1.m` from the website, place it in your directory and run it using Matlab's `publish` command (e.g. by typing '`publish lab1`' in the command window). Then you can look for the file '`../html/lab1.html`' which has been placed in your directory. Print it out and hand it in (or upload it to iLearn, if requested by your instructor). Notice that the file already contains printout of your code, so you do not need to hand in a separate printout of your code.

In order for all this to work properly and look nice, you need to do a few things.

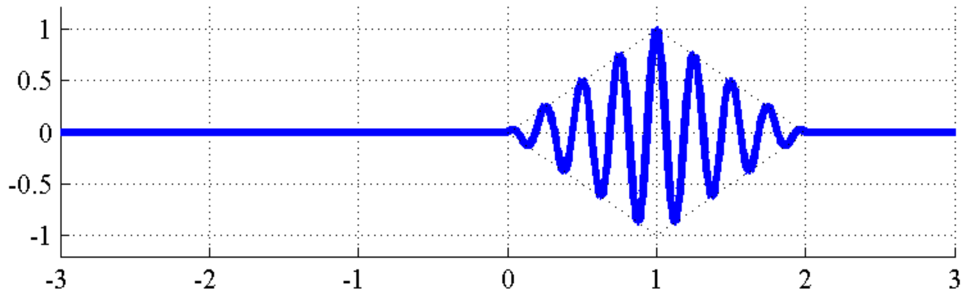
1. Please make sure that your functions produce no typed output. They only should produce the plots that are specified.
2. Each of your functions should have the following as the first line of code (after the function statement and comments);

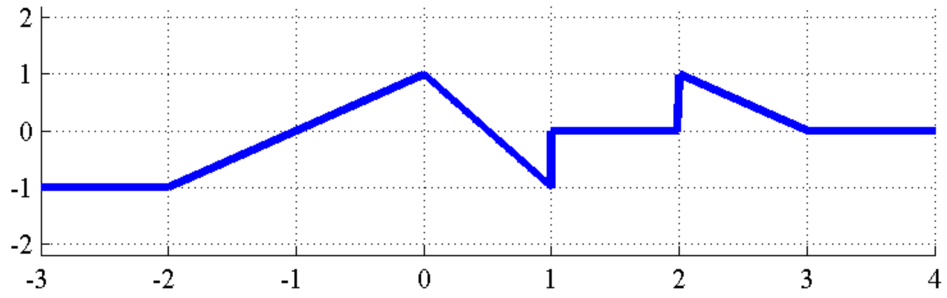
```
set(clf, 'Color', 'w');
```

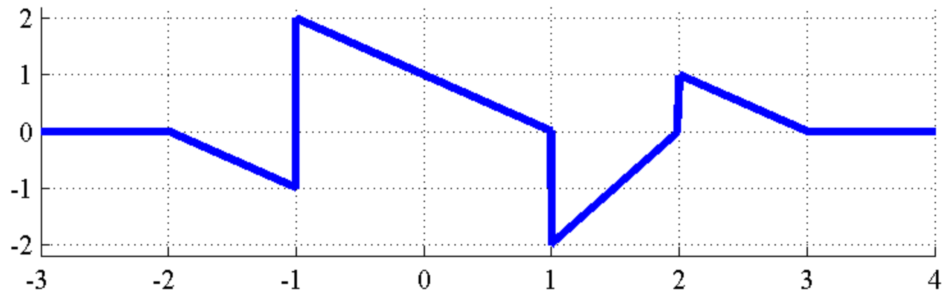
This line will assure that the figure is cleared and that the background color is white.

Here is the assignment:

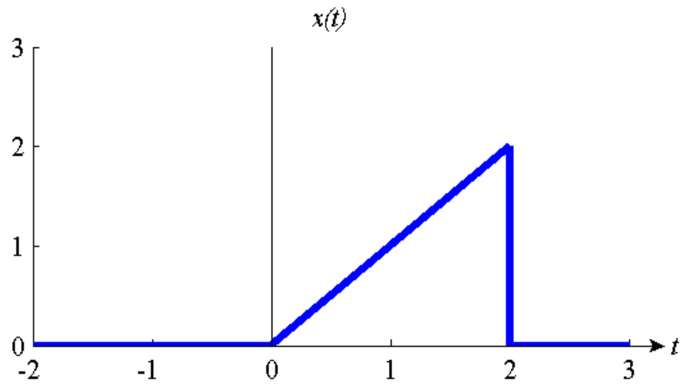
1. For each of the figures below, find the simplified equation (such as the one above) that describes the plotted function. Assume that function is zero outside of the visible limits of the plot. Now to check your answer, use Matlab to create a function called `lab1_1.m` that, when run, produces plots your expressions like the ones below. Your plots should pretty closely match those shown in the figure.







2. Given the plot of $x(t)$ below, create a Matlab function, `lab1_2.m,` that when run produces plots of the following over the interval $-4 < t \le 4$ secs. Plot all these functions in a single figure in a 3x2 matrix of subplots using Matlab's `subplot` command.
    a. $x(t-1)$
    b. $x(3+t)$
    c. $x(t)+1$
    d. $x(2t)$
    e. $x(t/2+1)$
    f. $tx(t)$



A very important class of signals of the form

$$x(t) = e^{-\alpha t}\cos(\omega t + \theta)u(t)$$

3. For this part, let $\theta = 0$, which leaves two parameters: α and ω . Use Matlab to write a function, lab1_3.m, plot $x(t)$ for $0 \le t \le 5$ with different values of ω and α . Make a plot with two horizontal panels using Matlab's subplot command. On the top panel plot $x(t)$ with α held constant and different values of ω .
   a. $\omega = 2\pi \cdot 1$, $\alpha = 1$
   b. $\omega = 2\pi \cdot 2$, $\alpha = 1$
   c. $\omega = 2\pi \cdot 4$, $\alpha = 1$

Plot the three traces in different colors, and use Matlab's `legend` command to annotate your plot:

```
legend({'\omega=2\pi1' '\omega=2\pi2' '\omega=2\pi4'});
```

Look up this command in Matlab's help system and understand the above line.

On the lower panel, plot $x(t)$ with ω held constant and different values of α.
   d. $\omega = 2\pi \cdot 2$, $\alpha = 1$
   e. $\omega = 2\pi \cdot 2$, $\alpha = 2$
   f. $\omega = 2\pi \cdot 2$, $\alpha = 4$

Again, plot the traces in different colors, and use Matlab's `legend` command to annotate your plot.

4. Using the same form of $x(t)$ as in the previous problem, fix $\omega = 2\pi \cdot 1$, $\alpha = 1$ and investigate the effect of different values of θ. Use Matlab to write a function, `lab1_4.m`, that plots $x(t)$ on a single plot for
   a. $\theta = 0$
   b. $\theta = \pi / 4$
   c. $\theta = \pi / 2$
   d. $\theta = 3\pi / 4$
   e. $\theta = \pi$

Again, plot the traces in different colors, and use Matlab's `legend` command to annotate your plot.

General rules for lab assignments

The following comments apply to this and all subsequent lab assignments.

1. Your code should be neat, easy to read and commented where appropriate. If you instructor has to struggleto figure out what you are doing, your grade will suffer.
2. You should be prepared to demonstrate that your code works, but if asked by the instructor.
3. You are encouraged to help each other, but **YOU ARE EACH RESPONSIBLE FOR DESIGNING AND CREATING YOUR OWN WORK**. If you merely copy your neighbor's labwork, you will both get exactly the same grade: **ZERO**.