ECE 3300
MATLAB Assignment 1

This assignment (1) gives an overview of MATLAB's basic commands, (2) demonstrates how to represent and use simple discrete-time signals in MATLAB, (3) introduces sampling as a way to represent continuous-time signals, (4) shows how to perform signal transformations and combinations, (5) demonstrates how to produce a well-formatted signal plot, (6) illustrates how to determine difference/derivative signals and summation/integral signals.

Getting Started

Go to the **Editor** tab, choose **New** and then **Script**. This starts a new place to work. Then go to **Editor**, **Save** to save your work as **filename.m**, where **filename** is whatever filename you choose. During this process you can choose a directory to save it in. To run your code at any time, use **Editor**, **Run**.

The first line of your file is required to be your name in a comment statement, something like `% Bellwhether Fumblemuffin` if you are unfortunate enough to have that name. Comments always begin with a percent sign (`%`).

The second line of your file is required to be a commented line containing your student ID number, like this: `% C12345678` except that you replace the number with your actual ID. Failure to do this will result in a score of zero on the entire assignment.

The third line of your file is required to be the name of the MATLAB assignment, also as a comment; in this case, you should have `% MATLAB 1`.

You are strongly encouraged to have your fourth line be `clear; clc; close all;` because these commands clear all variables, clean your screen, and close any figures, respectively. The reason this line is recommended is that when you rerun your code while making changes, you can then be sure that there are no strange results arising from your previous activities.

The semicolons in this command suppress printing of output. You are strongly encouraged to end *every* line (except for comments) with a semicolon. (When you want to print output you should use special commands discussed later.)

The fifth line of your file should be the commented statement `% 1`, meaning that the code that follows is to be an answer to the first part of the assignment. When you get to code that works on the second part of the assignment, add the line `% 2` and so on. These lines are required.

Getting Help

MATLAB has extensive help files online. Perhaps the easiest way to get help on a particular command is to enter `matlab command` in a search engine, where `command` is the name of the command you want to learn more about. Web pages that begin with `www.mathworks.com` are official MATLAB pages.

Getting Finished

When you are satisfied that your code has accomplished all that is required for the assignment, go to the **Publish** tab. Under **Edit Publishing Options**, set the output format to **pdf**. Choose your output folder, and then click **Publish**. The result will include your code and outputs. You must submit your work using **Publish** in this way or you will receive a zero on the assignment.

Note that extra long lines will scroll off the page using **Publish**. To avoid this, break a long line with the command `...` (three periods) and continue it on the next line.

Variables and Constants

The command `a=1` sets the variable `a` equal to one. The command `b=3-a` sets the variable `b` equal to `3-a`, which equals 2.

Predefined constants in MATLAB include `pi` (approximately 3.1416) and `e` (approximately 2.7183). Also predefined are `i` and `j` (both $\sqrt{-1}$).

Lists

The command `b=[1 2 3 4]` produces a list of the numbers 1 through 4 with the value stored in the variable `b`. (The command `b=[1,2,3,4]` does the same thing.) Lists can be made of lists; for example, `c=[b 5]` (or `c=[b,5]`) is the same as `c=[1 2 3 4 5]`, and `d=[b b]` is the same as `d=[1 2 3 4 1 2 3 4]`.

The ith element in a list can be accessed by the use of parentheses, and the final element in a list can be referred to via the command `end`. For the example of `d` above, `d(7)` equals 3 and `d(end)` equals 4, and the command `d(7)=8` modifies `d` to produce `d=[1 2 3 4 1 2 8 4]`.

Multiple elements in a list can be obtained or changed by putting lists inside the parentheses. For example, `d(6,8)` equals `[2 4]`, and setting `d(6,8)=[0 0]` modifes `d` (the version of `d` that was already modified) to produce `d=[1 2 3 4 1 0 8 0]`.

Ranges

The colon (":") allows one to easily make a list of equally spaced values. Thus an alternate way to obtain `b=[1 2 3 4]` is to use `b=1:4`. To make the separation between adjacent numbers in the list a value other than one, we use the format `n:m:k`, where `n` is the starting number, `m` is the separation between adjacent numbers, and `k` is the ending number. For example, `1:2:7` is the same as `[1 3 5 7]`.

Ranges can be used absolutely anywhere lists are used. For example, if `d=[1 2 3 4 1 2 3 4]`, `d(1:2:7)` is equal to `[1 3 1 3]`. Note that separation between adjacent numbers `m` can be negative; `5:-1:1` is equal to `[5 4 3 2 1]`. It follows that `d(end:-1:1)` reverses the order of the elements in list `d`.

Arithmetic Operators

In MATLAB, `+` is used to add two numbers, `-` is used to subtract one number from another, `*` is used to multiply two numbers, and `/` is used to divide one number by another. Each of these operators can be used between lists and single values. For example, if `a=[1 2 3 4]`, `a+2` equals `[3 4 5 6]`, `a-2` equals `[-1 0 1 2]`, `a*2` equals `[2 4 6 8]`, and `a/2` equals `[0.5 1 1.5 2]`.

Putting a dot (period) in front of these commands allows them to perform operations element by element across two lists. For example, if `a=[1 2 3 4]` and `b=[4 3 2 1]`, `a.+b` equals `[5 5 5 5]`, `a.-b` equals `[-3 -1 1 3]`, `a.*b` equals `[4 6 6 4]`, and `a./b` equals `[0.25 0.667 1.5 4]`.

To raise a number to a power, `^` should be used. The dot form should be used between two lists as well as between a list and number because MATLAB reserves the power operator in a special way for square matrices. Thus, if `a=[1 2 3 4]` and `b=[4 3 2 1]`, `a.^2` equals `[2 4 8 16]` and and `a.^b` equals `[1 8 9 4]`. Whenever two lists are used with any of these commands, the lists must be the same length.

Built-in Functions

`exp(x)`, `log(x)`, `log10(x)`, and `sqrt(x)` compute the exponential e^x, the natural logarithm $\ln(x)$, the base-10 logarithm $\log_{10}(x)$, and the square root $\sqrt{x}$, respectively. The trigonometric functions `sin(x)`, `cos(x)`, and `tan(x)` do exactly what one expects (in radians), as do their inverse functions `asin(x)`, `acos(x)`, and `atan(x)`. Each of these commands can be used on lists; they apply the operation to every element in the list. There is also a four-quadrant version of the inverse tangent, `atan2(y,x)`, that computes the inverse tangent of $\frac{y}{x}$ but has inputs y and x entered separately so that it can determine the correct quadrant.

Functions Designed Specifically for Lists

`length(c)` produces the length of a list `c`. `max(c)`, `min(c)`, `prod(c)`, and `sum(c)` produce the maxinum value, the minimum value, the product of the values, and the sum of the values of list `c`, respectively. `sort(c)` sorts the values in increasing order. For example, if `d=[1 2 3 4 1 2 3 4]`, `length(d)` equals 8, `max(d)` equals 4, `min(d)` equals 1, `prod(d)` equals 576, `sum(d)` equals 20, and `sort(d)` produces the list `[1 1 2 2 3 3 4 4]`. `find(c)` produces a list of the indices of the nonzero values of `c`. For example, if `d=[1 0 0 4 -3 0 0 1]`, `find(d)` gives `[1 4 5 8]`.

Relational Operators

In MATLAB, two numbers can be compared via an equality or inequality. If the statement is true, the output is a one; if the statement is false, the output is a zero. The relational operators are `<` (less than), `>` (greater than), `<=` (less than or equal to), `>=` (greater than or equal to), `==` (equal to), and `~=` (not equal to). For example, `2<3` equals one, `2>=3` equals zero, and `2~=3` equals one.

These commands can also be used between a list and a number as well as between two equal-length lists. For example, if `a=[1 2 3 4]` and `b=[4 3 2 1]`, then `a<2` equals `[1 0 0 0]` and `a<b` equals `[1 1 0 0]`.

Commands such as these can be modified and combined via "and" (`&`), "or" (`|`), and "not" (`~`) commands. For example, if `a=[1 2 3 4]`, then `(a<=3)&(a>=2)=[0 1 1 0]`, because 2 and 3 are the values of `a` that satisfy `a<=3` AND `a>=2`. Also, `(a>3)|(a<2)=[1 0 0 1]` because 1 and 4 are the values of `a` that satisfy `a>3` OR `a<2`. And `~(a==3)=[1 1 0 1]` because 1, 2, and 4 are the values of `a` such that it is NOT true that `a==3`.

Subsets of Lists

Suppose `a=[5 4 3 2 1]` and we wish to produce the subset of the list in which `a` is strictly greater than 3. Note that `a>3` produces [1 1 0 0 0]. It follows that `find(a>3)` produces the indices where `(a>3)` is nonzero; that is, `find(a>3)=[1 2]`. Because `a(1,2)=[5 4]`, it follows that `a(find(a>3))=[5 4]`.

However, there is an even simpler way to find this subset: it turns out that getting rid of the `find` command and simply writing `a(a>3)` produces `[5 4]` as well. It helps to think of this command as saying "the values of `a` such that `a` is greater than three." The reason this works has to do with MATLAB's definition of logical variables which we will not discuss in this course. However, this simpler approach should *always* be used rather than the approach using `find` because it is easier to read and runs more quickly.

Creating a Discrete-Time Signal

A discrete-time signal can be implemented in MATLAB as two lists: (1) a list of time values, and (2) a list of the values of the signal *at* those time values. These lists must be the same length and must be coordinated; that is, the ith signal value must go with the ith time value. Ordinarily, the list of time values are ordered from smallest to highest value.

Because lists in MATLAB must be of finite length, we can only fully represent time-*limited* signals in MATLAB. However, we can also represent portions of left-sided, right-sided, and two-sided signals. For example, to represent $x[n] = 2(n-1)^2$ from $n = -10$ to $n = 10$, we first use `n=-10:10` to create the list of time values and then use `x=2*(n-1).^2` to create the corresponding list of signal values. The results of these commands are that `n` equals `[-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10]` and `x` equals `[242 200 162 128 98 72 50 32 18 8 2 0 2 8 18 32 50 72 98 128 162]`.

Signal Values

Suppose we have made the lists `n` and `x` and we wish to determine $x[7]$. Note that $x[7]$ is the value of `x` when `n` *equals* seven; it is *not* the seventh item in the list `x`. How is this done? The answer is to use `x(n==7)`. This command works because the time values and signal values are "paired" together; the position in the list where $n = 7$ is the position we want in the list `x`. For the example above, `x(n==7)` equals 72 because $x[7] = 2(7-1)^2 = 72$.

This approach can be generalized to find quantities such as all signal values between two points in time. For example, to obtain $x[3]$ through $x[7]$, one can use `x((n>=3)&(n<=7))`; for our example, this is equal to the list `[8 18 32 50 72]`.

Time Values

Now suppose we wish to determine all time values n where $x[n] = 8$. How do we do this? Here the answer is to flip the roles of `x` and `n` around and use `n(x==8)`. Again this works because the two lists are "paired" together. For our example, `n(x==8)` gives the list `[-1 3]`. This is correct because $x[-1]$ and $x[3]$ both equal eight.

This approach can be generalized. For example, `n(x<=8)` finds all time values n for which $x[n] \leq 8$; for our signal, the result is the list `[-1 0 1 2 3]`. This answer is correct because $x[-1] = 8$, $x[0] = 2$, $x[1] = 0$, $x[2] = 2$, and $x[3] = 8$; for all other values of n, $x[n] > 8$. (To find the *minimum* time n for which $x[n] \leq 8$, use `min(n(x<=8))`.)

Printing Results

The `fprintf` command should be used to print results. The simplest form of the command is `fprintf('Stuff you want to say\n')` This prints `Stuff you want to say`. The `\n` is a newline command; it causes the *next* use of `fprintf` to start on a new line. (The newline is optional, but one normally uses it.) To add an additional new line so that there is a skipped (blank) line before any more text appears, use `fprintf('Stuff you want to say\n\n')`

Suppose `a=3.78513827`. To print the value of `a` with the format `Radius = 3.785 meters`, use `fprintf('Radius = %0.3f meters\n',a)`. The `%` indicates that a variable (in our case, the variable `a`) is to be substituted in, and the `0.3` specifies formatting - the number to the left of the decimal point indicates the minimum amount of space to allow for the number (in this case, zero characters), and the number to the right of the decimal point gives the number of digits that appear to the right of the decimal point. The `f` indicates that the number is formatted for printing as a floating point (real) number. If `a=1.68` and `b=1.72`, then to print `1.68 parents and 1.72 children`, the command to use is `fprintf('%0.3f parents and %0.3f children\n',a,b)`.

If a is a list, fprintf will repeat for each element in the list. For example, if a=[1.234 5.678], fprintf('Radius = %0.3f meters\n',a) will produce:

```
    Radius = 1.234 meters
    Radius = 5.678 meters
```

A "Pretend" Assignment

So that all expectations are absolutely clear, the following is a "pretend" assignment followed by code that would result in a score of 100%. You are strongly encouraged to try running this code yourself. However, do *not* turn it in.

1. Represent the signal $x[n] = \sin(\frac{\pi}{8}n)$ from $n = -16$ to $n = 16$. Do not print out the values.
2. Let x_min denote the minimum value of this signal over this range of time values. Determine and print out this value, including the name of the variable.
3. Let n_min be the time value(s) where this minimum occurs. Find and print out this value/these values, including the name of the variable. (The minimum might occur at more than one location in time.)
4. Let x_sum be the sum of all the values of the signal over this range of time. Find and print out this value, including the name of the variable.

Here is well-written code and output after using **Publish**. Note that there are only 18 lines of code for the entire assignment; 7 of these are comments, and 6 more are for printing outputs or clearing the system, leaving only 5 lines that really "do" something.

```
% Bellwether Fumblemuffin
% C24681357
% MATLAB Pretend Assignment 1A
clear; clc; close all;
% 1
n = -16:16;
x = sin(pi/8.*n);
% 2
x_min = min(x);
fprintf('x_min = %0.3f\n\n',x_min);
% 3
n_min = n(x==x_min);
fprintf('n_min = \n');
fprintf('    %0.3f\n',n_min);
fprintf('\n');
% 4
x_sum = sum(x);
fprintf('x_sum = %0.3f\n',x_sum);

        x_min = -1.000

        n_min =
            -4.000
            12.000

        x_sum = 0.000
```

Hopefully these answers make sense to you: $\sin(-4\pi/8) = \sin(12\pi/8) = -1$ so the minimum value is -1 and occurs at $n = -4$ and $n = 12$. Also, the sum of all the sine values is zero.

PROBLEM STATEMENT: PART ONE

1. Consider the signal $x[n] = \dfrac{10(n^4 + 20n^3 - 1000)}{(n^2 + 100)^2}$. Represent, but do not print out, this signal using a list of time values and a list of signal values from $n = -100$ to $n = 100$. Call the list of time values `n` and the list of signal values `x`.

   (a) Let `xmax` denote the maximum value of this signal and let `nmax` be the time value(s) where this maximum occurs. Let `xmin` denote the minimum value of this signal and let `nmin` be the time value(s) where this minimum occurs. Use MATLAB to determine and print out these values, including the names of the variables.

   (b) Let `nzero` denote the time value(s) where $x[n]$ is closest to five. Use MATLAB to determine and print out these time values, including the name of the variable. *Hint:* $x[n]$ is closest to five at the time value(s) where $(x[n] - 5)^2$ is minimum.

   Remember to use **Publish**.

Creating a Continuous-Time Signal

A continuous-signal *cannot* be directly implemented in MATLAB using lists because the lists would need to be infinite length even when representing a finite-time duration piece of the signal. Instead, continuous-time signals must be *sampled.* The smaller the time between adjacent samples, the better the approximation (and also, the longer the lists, so there is a tradeoff between accuracy and complexity). We define the *sampling time* t_s as the time between adjacent samples, and we define the *sampling rate* to be the inverse of the sampling time; that is, $f_s = 1/t_s$.

As with discrete-time signals, we need a list of times and a corresponding list of signal values. For example, to represent the signal $x(t) = t\cos(2t)$ from $t = -5$ to $t = 5$ (in seconds) with a sampling rate of $f_s = 100$ samples per second, we first compute t_s via $t_s = 1/f_s = 0.01$. We create the list of time values using `t=-5:0.01:5`, and we then create the corresponding list of signal values using `x=t.*cos(2*t)`. The resulting lists are too large to display here (they each contain 1001 values).

Once $x(t)$ (in its sampled form) is implemented in MATLAB via `t` and `x`, how do we determine (for example) $x(3.72)$? We use `x(t==3.72)`. This will only work if 3.72 is actually one of the sampled values of `t`. If it is *unknown* whether 3.72 is one of the sampled values, or if it is known that 3.72 is *not* one of the sampled values, to find the *closest* value use `x(abs(t-3.72)==min(abs(t-3.72)))`.

What about finding all the (sampled) points in time where (for example) $x(t) \geq 2$? For this we use `t(x>=2)`.

Creating a Piecewise Signal

Consider the signal

$$x(t) = \begin{cases} -1 & \text{if } -1 < t \leq 0 \\ t^2 & \text{if } 0 < t \leq 1 \\ 1 & \text{if } 1 < t < \frac{3}{2} \\ 0 & \text{otherwise} \end{cases}$$

To represent this signal in MATLAB from $t = -5$ to $t = 5$ with a sampling rate of 50 samples per second, we first use `t=-5:0.02:5` to set the time values. As for the signal values, we can use

```
   x=-((t>-1)&(t<=0)) +(t.^2).*((t>0)&(t<=1)) +((t>1)&(t<3/2))
```

This works because each of the three expressions involving ampersands are only equal to one for a specific range of t and are zero otherwise. For example, `(t>0)&(t<=1)` is equal to one when $0 < t \leq 1$ and zero otherwise. Because the three expressions are non-overlapping, at most one of these expressions is one and the others are zero at any particular value of t. If t is outside the range of all of the expressions, none will be equal one, and the answer will be zero. This approach can be used to represent any piecewise signal.

Signal Transformations

In the table below, we show how to implement the various signal transformations discussed in the course. The original signal is assumed to be represented by the lists `t_1` and `x_1`, and the transformed signal is to be represented by `t_2` and `x_2`.

Transformation	Equation	Restrictions	`t_2`	`x_2`
Amplitude Scaling	$x_2(t) = ax_1(t)$	any a	`t_1`	`a*x_1`
Time Shifting	$x_2(t) = x_1(t-b)$	any b	`t_1+b`	`x_1`
Time Scaling	$x_2(t) = x_1(at)$	$a > 0$	`t_1/a`	`x_1`
Time Reverse	$x_2(t) = x_1(at)$	$a < 0$	`t_1(end:-1:1)/a`	`x_1(end:-1:1)`
Absolute Value	$x_2(t) = \|x_1(t)\|$	none	`t_1`	`abs(x_1)`
Squaring	$x_2(t) = x_1^2(t)$	none	`t_1`	`x_1.^2`

For time shifting, note that $x_2(t_1+b) = x_1(t_1)$ so `t_2=t_1+b`. For time scaling, $x_2(t_1/a) = x_1(t_1)$ so `t_2=t_1/a`. In the case of time reversals, `end:-1:1` is needed to reverse the order of the elements in the lists so that time values are again ascending.

Signal Combinations

Suppose $x_1(t)$ and $x_2(t)$ are implemented over a common list of time values `t` and the signal values are represented via `x1` and `x2`. Then the signal sum $y(t) = x_1(t) + x_2(t)$ is represented by `t` (the same `t`) and `y=x1.+x2`. Similarly, the signal product $z(t) = x_1(t)x_2(t)$ is represented by `t` and `z=x1.*x2`. Furthermore, assuming $x(t)$ is represented by `t` and `x` and assuming that `t` ranges from $-a$ to a for some positive value of a, $y(t) = \text{Ev}\{x(t)\} = \frac{1}{2}x(t) + \frac{1}{2}x(-t)$ is represented by `t` and `y=0.5*(x.+x(end:-1:1))` and $z(t) = \text{Od}\{x(t)\} = \frac{1}{2}x(t) - \frac{1}{2}x(-t)$ is represented by `t` and `z=0.5*(x.-x(end:-1:1))`. Note that the code for the even and odd parts will *not* work if t does not have a range from $-a$ to a!

Plotting Continuous-Time Signals

Plotting involves multiple commands. To receive full credit on plots, we have very high expectations on their quality and readability! To get full credit, use the the approach below, which assumes three plots are put on the same graph. (Adjust for different numbers of plots.) We assume that three signals $x_1(t)$, $x_2(t)$, and $x_3(t)$ are already set up via `t_1` and `x_1`, `t_2` and `x_2`, and `t_3` and `x_3`, respectively.

There are thirteen commands needed to do everything required. The order is important! Unless you really know what you are doing and have a good reason, we do not recommend changing the order.

1. Use `figure` to set up a figure window. The command to do this is `figure();`
2. Plot the horizontal axis using `plot`. To plot a thin solid line in black from $(-10, 0)$ to $(10, 0)$, use `plot([-10,10],[0,0],'LineStyle','-','Color',[0,0,0],'LineWidth',1);` (Choose whatever endpoints are appropriate for your own situation. It just needs to be long enough. We will "crop" the plot later.)
3. Turn on `hold` so more lines/curves can be added to the figure. The command to do this is `hold on;`
4. Plot the vertical axis using `plot`. To plot a thin solid line in black from $(0, -10)$ to $(0, 10)$, use `plot([0,0],[-10,10],'LineStyle','-','Color',[0,0,0],'LineWidth',1);` (Again, just make it long enough.)
5. Plot the first signal using the form `p1=plot`. (You are naming this curve `p1` because we need it named in order to make a plot legend later.) To plot a thick solid curve in blue, use the following command:

`p1 = plot(t_1,x_1,'LineStyle','-','Color',[0,0,0.8],'LineWidth',2);`
(The first item inside `plot` should always be the list of time values, and the second item should always be the list of signal values. The `LineWidth` of 2 makes it a thicker line; a value of 1 is thinner. The `Color` can be specified many different ways; the approach used here is a form of RGB (red green blue) mixing where each number must be between 0 and 1. Here we are using 0% red, 0% green, and 80% blue; we never use 100% because we find it hard on the eyes. `LineStyle` specifies whether the line is solid, dashed, etc. The `'-'` choice is a solid line.)

6. Plot the second signal using the form `p2=plot`. (You are naming this curve `p2`.) To plot a thick dashed curve in green, use the following command:
   `p2 = plot(t_2,x_2,'LineStyle','--','Color',[0,0.8,0],'LineWidth',2);`
   (We recommend all signal curves have `LineWidth` 2 and the axes have `LineWidth` 1. We strongly recommend using BOTH a different `Color` and a different `LineStyle` for every signal so that it is easy to tell them apart on the legend regardless of whether a black-and-white or color device/printout is used. You will lose points if it is hard to tell your signals apart. The `LineStyle` of `'--'` is a dashed line. The `Color` is 0% red, 80% green, and 0% blue.)

7. Plot the third signal using the form `p3=plot`. (You are naming this curve `p3`.) To plot a thick dash-dot curve in magenta, use the following command:
   `p3 = plot(t_3,x_3,'LineStyle','-.','Color',[0.8,0,0.8],'LineWidth',2);`
   (The `'-.'` `LineStyle` alternates dashes and dots; another choice is `':'` which gives a dotted line. The `Color` is 80% red, 0% green, and 80% blue, and red and blue make purple, or magenta.)

8. Turn off `hold` because we are done adding plots. The command to do this is `hold off;`

9. Limit the range of plotting using `axis`. If you wanted the horizontal axis to go from -5 to 5 and the vertical to go from -3 to 3, use `axis([-5,5,-3,3]);`
   (You typically adjust this after looking at your plots. For signals that are finite duration, you MUST have the horizontal axis go *beyond* the turn-on and turn-off times in each direction, or you will lose points. Your time variables for each plot MUST cover the entire range of your plot – that is, your signal cannot "disappear" partway through the plot – or, again, you will lose points. You want to use `axis` to crop the picture enough so that you don't have a tiny graph on giant axes, but at the same time, you also want to leave enough room for the legend so that it does not obliterate your signals; this too would cost you points. Because of this, you should re-adjust `axis` after you add your legend in Step 13.)

10. Add a title to the plot using `title`. Here is the code:
    `title('Plot 1B.1: Amplitude Scaling');`
    (To not lose points, the title MUST reference the MATLAB Assignment and the part of the assignment that the plot refers to. It also needs to describe what is being presented. In this current assignment, the first part explores amplitude scaling, so this is a good title.)

11. Add a horizontal axis label using `xlabel`. If the horizontal axis is a time axis, use `xlabel('t');`
    (If the horizontal axis is ω, which will occur later in the course, use `xlabel('\omega');`)

12. Add a vertical axis label using `ylabel`. If the vertical axis is a signal axis, use the command `ylabel('x(t)');`
    (If it is $X(j\omega)$, which will occur later in the course, use `ylabel('X(j\omega)');`)

13. Add a legend using `legend` and the names you gave for your curves, `p1`, `p2`, and `p3`. If `x_1` is $x(t)$, `x_2` is $2x(2t)$, and `x_3` is $x(3t-1)$, use
    ```
    legend([p1,p2,p3],'x(t)','2x(2t)','x(3t-1)',...
       'Location','northeast');
    ```
    (The three dots are not necessary for the code to run, but recall that in general they are used for splitting up lines that are too long to be seen when using **Publish**. The `'northeast'` means the upper right corner of the plot. Other choices include `'southwest'`, etc. Make sure your descriptions reflect what the signals really are, *not* your own internal variables which only have meaning to you. You will lose points if you make this mistake. To add

a subscript in a title, label, or legend, use the underscore character; for example, $x_1(t)$ is `'x_1(t)'`. To add a superscript, use a caret; for example, $x^2(t)$ is `'x^2(t)'`.)

It should be clear how to modify this set of commands to print 2 plots, 4 plots, etc. If you are only printing one plot, you should *not* have a `legend`.

If you are using multiple figures in an assignment, you must go through all thirteen steps for *each* figure.

Plotting Discrete-Time Signals

For plotting discrete-time signals, we use "ball-and-stick" plots. MATLAB can produce these with the command `stem`. The command is similar to `plot`. To enable `stem` to produce "balls" and "sticks", we use it with `'Marker'` set to `'.'`. Below is the code to plot 3 signals, complete with labels and everything else needed for full credit:

1. `figure();`
2. `plot([0,0],[-10,10],'LineStyle','-','Color',[0,0,0],'LineWidth',1);`
   (This makes a vertical axis; `stem` makes its own horizontal axis.)
3. `hold on;`
4. `p1 = stem(n_1,x_1,'Marker','.','Color',[0,0,0.8],'LineWidth',2);`
5. `p2 = stem(n_2,x_2,'Marker','.','Color',[0,0.8,0],'LineWidth',2);`
6. `p3 = stem(n_3,x_3,'Marker','.','Color',[0.8,0,0.8],'LineWidth',2);`
7. `hold off;`
8. `axis([-10,10,-5,5]);`
   (This sets the range of plotting. Adjust to your situation.)
9. `title('Plot 4A.1: Add Nice Description');`
10. `xlabel('n');`
11. `ylabel('x[n]');`

12.
    ```
    legend([p1,p2,p3],'Signal 1','Signal 2','Signal 3',...
       'Location','northeast');
    ```

If you are only plotting one signal, a legend should not be used. To create a plot with the "sticks" but without the "balls", set `'Marker'` to `'none'`.

PROBLEM STATEMENT: PART TWO

2. Consider the continuous-time signal

$$x(t) = \begin{cases} 4 & \text{if } -5 \le t < -3 \\ \sqrt{25-t^2} & \text{if } -3 \le t < 4 \\ 3 & \text{if } 4 \le t < 8 \\ 0 & \text{otherwise} \end{cases}$$

   In everything below use a sampling time of 0.01. Vertically, the plot should take up most of the figure and none of the plot should be cut off. Each part below should be a different graph.

   (a) Plot $x(t)$ and $-2x(t+3)$ on the same graph from $t = -10$ to $t = 10$.

   (b) Plot $x(t)$ and $\frac{1}{2}x(\frac{3t}{4})$ on the same graph from $t = -15$ to $t = 15$.

   (c) Plot $\text{Ev}\{x(t)\}$ and $\text{Od}\{x(t)\}$ on separate graphs from $t = -10$ to $t = 10$.

   Remember to use **Publish**.

Differences

If `x` is a list, `diff(x)` is a list of the successive differences between adjacent values of `x`; that is, `diff(x)` is the vector `[x(2)-x(1),x(3)-x(2), ..., x(m)-x(m-1)]`, where `m` is the length of the list `x`. Note that the length of `diff(x)` is `m-1`; that is, the length of `diff(x)` is one less than the length of `x`. For example, if `x=[1 2 4 5 5]`, then `diff(x)=[1 2 1 0]` because $2-1=1$, $4-2=2$, $5-4=1$, and $5-5=0$.

Discrete-Time Difference Signals

Suppose we have a signal $x[n]$ from $n=a$ to $n=c$ represented via `n_1=a:c` and `x`. To represent $y[n]=x[n]-x[n-1]$ via `n_2` and `y`, it is clear that `y=diff(x)`, but what about `n_2`? We note that the equation $y[a]=x[a]-x[a-1]$ is not allowed because $x[a-1]$ is not defined; the smallest time value is a. This means that the first valid equation is $y[a+1]=x[a+1]-x[a]$. Thus the time values for `y` need to start at $a+1$; that is, `n_2` needs to consist of all but the first (smallest) value of `n_1`. We can remove the first value via the code `n_2=n_1(2:end)`. Thus a difference signal is implemented fully via `n_2=n_1(2:end)` and `y=diff(x)`.

Continuous-Time Derivative Signals

We first consider how to obtain an approximation for the derivative signal $y(t)=\frac{d}{dt}x(t)$. We note that the definition of the derivative is

$$y(t)=\tfrac{d}{dt}x(t)=\lim_{\epsilon\to 0}\frac{x(t)-x(t-\epsilon)}{\epsilon}$$

It follows that if b is the sampling time and b is small (much less than one), then

$$y(t)=\tfrac{d}{dt}x(t)\approx\tfrac{1}{b}(x(t)-x(t-b))$$

Suppose $x(t)$ is implemented over $a\le t\le c$ via `t_1=a:b:c` and `x`. (Note that `b` is the sampling time, so `1/b` is the sampling rate.) Then $x(t)-x(t-b)$ is the difference of adjacent terms in `x`. It follows that $y(t)=\frac{d}{dt}x(t)$ can be implemented as `t_2` and `y` via `t_2=t_1(2:end)` and `y=diff(x)/b`. If `b` is sufficiently small, the result should be a good approximation to the derivative.

Impulse Signals

A discrete-time impulse signal $\delta[n]$ can be easily implemented over `n=a:c` (assuming $a\le 0$ and $c\ge 0$ via `delta=(n==0)` because `n==0` is equal to 1 if $n=0$ and 0 otherwise. Also note that $\delta[n]=u[n]-u[n-1]$, so if we define `u=(n>=0)`, we can also obtain `delta` via `delta=diff(n>=0)`.

But what about the continuous-time impulse $\delta(t)$? Well, because $\delta(t)=\frac{d}{dt}u(t)$, it follows that using a sample time of b gives the approximation $\delta(t)\approx\frac{1}{b}(u(t)-u(t-b))$ We can implement $u(t)$ in MATLAB via `u=(t>=0)`, so we can implement $\delta(t)$ (approximately) via `delta=diff(u)/b` or equivalently `delta=diff(t>=0)/b`. But from the above discussion about $\delta[n]$, we know that `diff(t>=0)` equals `(t==0)`. It follows that, if the sample time is b, we can implement $\delta(t)$ (approximately) via `delta=(t==0)/b`. Note that if `b` is small (much less than one), `delta` will be large (much greater than one) at $t=0$. Also, `delta` will be zero at all other values of t. For example, if `b=0.01` then `delta=(t==0)/b` will equal 100 at $t=0$ and zero at $t=\pm 0.01$, $t=\pm 0.02$, etc. This is a reasonable approximation for $\delta(t)$. It improves as `b` is made even smaller.

The more general impulse signals $\alpha\delta[n-\beta]$ and $\alpha\delta(t-\beta)$ can be obtained via the MATLAB signal transformations described in the previous assignment.

Cumulative Summations

If `x` is a list of length `m`, `cumsum(x)` produces the list `[x(1), x(1)+(2), x(1)+x(2)+x(3), ..., x(1)+x(2)+···+x(m)]`. Note that the length of `cumsum(x)` is `m`, the same as the length of `x`. For example, if `x=[1 2 1 0]`, then `cumsum(x)=[1 3 4 4]`, because $1 = 1$, $1 + 2 = 3$, $1 + 2 + 1 = 4$, and $1 + 2 + 1 + 0 = 4$.

Discrete-Time Summation Signals

Because the `cumsum` function is built in to MATLAB, determining cumulative summation signals is extremely simple. If $x[n]$ is represented over $a \le n \le c$ via `n` and `x`, then, assuming $x[n] = 0$ for $n < a$, $y[n] = \sum_{i=-\infty}^{n} x[i] = \sum_{i=a}^{n} x[i]$ is represented via `n` (the same `n` as that for `x`) and `y=cumsum(x)`.

Continuous-Time Integral Signals

We now consider the integral signal $y(t) = \int_{-\infty}^{t} x(\tau)d\tau$. We assume that $x(t) = 0$ for $t < a$. We can approximate the integral as the sum of the areas of a bunch of rectangles; the ith such rectangle has width `b` and height $x(a + (i-1)b)$. Because the area of a rectangle is the product of its width and height, it follows that, if $t = a + kb$,

$$y(t) = \int_{-\infty}^{t} x(\tau)d\tau = \int_{a}^{t} x(\tau)d\tau = \int_{a}^{a+kb} x(\tau)d\tau \approx \sum_{i=1}^{k} x(a + (i-1)b)b$$

This equation can be implemented via `cumsum` and a multiplication by `b`. It follows that if $x(t)$ is represented over $a \le t \le c$ via `t` and `x`, then $y(t) = \int_{-\infty}^{t} x(\tau)d\tau$ can be approximately represented via `t` (the same `t` as `x`) and `y=cumsum(x)*b`.

The following table summarizes the results assuming the starting signal is implemented via `n=a:c` (if discrete time) or `t=a:b:c` (if continuous time) and `x`.

Operation	Equation	n_2 or t_2	x_2
Difference signal	$x[n] - x[n-1]$	`n(2:end)`	`diff(x)`
Derivative signal	$\frac{d}{dt}x(t)$	`t(2:end)`	`diff(x)/b`
Summation signal	$\sum_{i=-\infty}^{n} x[i]$	`n`	`cumsum(x)`
Integral signal	$\int_{-\infty}^{t} x(\tau)d\tau$	`t`	`cumsum(x)*b`

PROBLEM STATEMENT: PART THREE

3. Let $x(t) = \begin{cases} 2t & \text{if } 0 \le t < 3 \\ 18 - 4t & \text{if } 3 \le t < 6 \\ t - 12 & \text{if } 6 \le t < 12 \\ 0 & \text{otherwise} \end{cases}$. Continue to use the sampling time of 0.01.

   (a) Plot, on a single graph, $x(t)$ and the derivative signal $y(t) = \frac{d}{dt}x(t)$ over the range $t = -2$ to $t = 14$.

   (b) Plot, on a single graph, $x(t)$ and the integral signal $z(t) = \int_{-\infty}^{t} x(\tau)d\tau$ over the range $t = -2$ to $t = 14$.

   Remember to use **Publish**.