OVERVIEW: Amplitude Modulation

The world is full of functions like the two shown here:

![Graph of y=y(t)](image1)

![Graph of y=y(t)](image2)

Figure 1: Two ideal analog signals

Both have the mathematical form

\[ y(t) = A(t) \sin(\omega [t - t_0]), \]

where \( \omega > 0 \) is a constant angular frequency, \( t_0 \) is a constant reference time, and \( A(t) \) is a time-varying amplitude factor. The next sketch shows the graph of \( A \), drawn in red, together with the graphs shown above. Since the values of \( \sin \theta \) always lie between \(-1\) and \(+1\), inclusive, the values of \( y(t) \) are always somewhere between \(-A(t)\) and \(+A(t)\).

![Graph of y=y(t)](image3)

![Graph of y=y(t)](image4)

Figure 2: Modulating functions shown in red

As evidence for the importance of functions like these, note that the first graph shown above appears prominently in the Dynamics section of the MECH 221 formula sheet. But these functions are not used only to solve exam problems: they are also involved in radio communications, where “AM” stands for Amplitude Modulation. Informally, to send the information in function \( A(t) \) (perhaps an audio waveform) by radio, we multiply \( A(t) \) by a high-frequency “carrier” function \( \sin(\omega [t - t_0]) \) whose frequency is high enough that it will fly off a transmitting antenna. A radio receiver catches the function \( y(t) \) and does some processing to extract the “modulation” function \( A(t) \), which it then sends to the speakers. See Wikipedia (“Amplitude Modulation”) for much more on this topic, including some nice illustrations.

The goal of this lab is to reconstruct a plot like the ones in Figure 2, using nothing more than a few measured data points \((t_k, y_k)\) taken from the graph of \( y(t) \). This will require (i) estimating the angular frequency \( \omega \), and (ii) producing a function \( \tilde{A}(t) \) that provides a good approximation for the true function \( A(t) \).
**MODULATION AND DEMODULATION**

**Vectorized Commands.** Matlab’s built-in commands are “vectorized”. This means that if your Matlab workspace contains a vector (or even a matrix) object named \( T \), and a scalar named \( \omega \), the expression \( \omega \cdot T \) will correctly generate a vector (or matrix) object in which every element of the original \( T \) has been multiplied by \( \omega \). Then, the command

\[
S = \sin(\omega \cdot T);
\]

will create a new vector (or matrix) object named \( S \), with exactly the same shape as \( T \), but with each component in \( S \) being the sine of the corresponding component of \( \omega \cdot T \).

The `plot` command works with vectors of \( x \)- and \( y \)-coordinates, joining them with line segments by default. To see vectorization at work, enter these commands at the Matlab prompt:

```matlab
T = linspace(0,4*pi,721); % Build a row vector with 721 elements
omega = 8; % Pick some angular frequency (a scalar)
S = sin(omega*T); % Fill a row vector with 721 sine values
plot(T,S); % Plot the 720 segments joining those (t,s) pairs
```

**Modulation.** To vectorize your own calculations, the dot prefix is key. For the mathematical function \( A(t) = t(4\pi - t) \), and the 721-element Matlab vector \( T \) produced in the previous block of commands, saying

\[
A = T .* (4\pi - T); \quad \text{or equivalently,} \quad A = 4\pi \cdot T - T.^2;
\]

\[
Y = A \cdot S;
\]

\[
\text{plot}(T,Y);
\]

triggers an element-by-element calculation that fills a new Matlab vector \( A \) with the numbers \( A(t) \) that correspond to the entries in vector \( T \). Next an element-by-element multiplication populates a vector \( Y \) with 721 matching values for \( y(t) = A(t) \sin(\omega t) \). Finally, the `plot` command joins the 721 resulting \((t, y(t))\) pairs with green segments.

**Demodulation.** The dot prefix works on division, too. Consider the mathematical relationship

\[
y(t) = A(t) \sin(\omega t) \quad \iff \quad A(t) = \frac{y(t)}{\sin(\omega t)}.
\]

If we have Matlab vectors \( T \) and \( Y \) of matching shapes, perhaps containing measured data values, and some scalar \( \omega \), the dot in the command

```
A = Y ./ sin(omega*T);
```

creates a vector \( A \) of the same shape as \( T \) and \( Y \), with entries calculated in perfect agreement with the ideal mathematical equation above. In this situation, we can produce a sketch like Figure 2 above by following up with the commands

```
plot(T,Y); % Graph the basic function y(t), in blue (the default)
hold on; % Hold that curve, we’re about to add more
plot(T,A,'r'); % Superimpose the graph of A(t), in red
```

(Danger: What if the denominator vector \( \sin(\omega \cdot T) \) has one or more zero components? Response: This can certainly happen, and it really pollutes \( A \). To avoid it, the sample code provided with this lab simply discards all data points \((t_k, y(t_k))\) with \( y(t_k) \approx 0 \) before calculating \( A \).)
SAMPLING AND INTERPOLATION

Sampling. Imagine some function $y(t)$ out in the real analog world, where $t$ can have any real value. In taking experimental measurements, we can only record values of $y$ at certain instants. Even if our observations are perfect, we acquire only a few data points $(t_k, y(t_k))$ from the true graph of $y(t)$. It’s natural to store these in Matlab vectors named $T$ and $Y$. Both vectors will have the same number of components, which is the number of measurements we have recorded. To graph these measurements as green circles, we then say

$$\text{plot}(T,Y,'go'); \quad \% 'g' \text{ is for green, 'o' is for circle}$$

Figure 3, below, shows 22 measured values from a certain function; the function itself is graphed for reference.

Interpolation. After making some measurements, the data points $(t_k, y(t_k))$ are all we have. For any time $t$ when we did not record a value, the true value of $y(t)$ is simply unknown. Some kind of sensible approximation will be the best we can do. One approach is called interpolation. It is nicely illustrated by the default operation of Matlab’s \texttt{plot} command. With vectors $T$ and $Y$ as above, saying \texttt{plot}(T,Y) automatically draws line segments joining the data points we actually know. The resulting piecewise-linear graph implicitly defines a new function, $\tilde{y}(t)$, by “linear interpolation”. See Fig. 4, below, for the linear interpolation built from the 22 data points illustrated in Fig. 3. The resulting piecewise-linear function $\tilde{y}(t)$ is a reasonable approximation to the true function $y(t)$. More data values, especially in regions where the true function has high curvature, would obviously improve its accuracy.
Of course there are other ways to define a new function \( \tilde{y}(t) \) by “joining the dots” of measured data, and Matlab’s built-in function \texttt{interp1} offers several ways to do so. With the measured data values in vectors \( T \) and \( Y \), and one or more times in a vector \( T_i \), the command

\[
Y_i = \text{interp1}(T,Y,T_i,'linear');
\]

will produce a corresponding vector of values \( \tilde{y}_i = \tilde{y}(t_i) \) using linear interpolation. Alternatives to the \texttt{‘linear’} method include \texttt{‘spline’} and \texttt{‘cubic’}. See Matlab’s online help about \texttt{interp1} to explore this topic further.

**MEASURING ANGULAR FREQUENCY**

**General Theory.** Splitting a given function \( y(t) = A(t) \sin(\omega[t - t_0]) \) into its component factors requires two steps:

1. finding \( \omega \) and \( t_0 \), and
2. calculating \( A(t) = \frac{y(t)}{\sin(\omega[t - t_0])} \).

Once \( \omega \) and \( t_0 \) are known, Step 2 is simple demodulation, as discussed above. To save time in this lab, we will simply

always assume, for this lab only: \( t_0 = 0 \) in all cases.

All that remains is half of Step 1: finding \( \omega \) from \( y(t) \). For this, we look at the times when \( y(t) = 0 \). Most of these will be times when \( \sin(\omega t) = 0 \), and we know the spacing between adjacent zeros of \( \sin \theta \) is \( \Delta \theta = \pi \).

![Zeros of \( \sin(\theta) \) are separated by \( \Delta \theta = \pi \)](image)

**Figure 5: Zeros of \( \sin(\theta) \) are separated by \( \Delta \theta = \pi \)**

Matching \( \theta = \omega t \) gives \( \Delta \theta = \omega \Delta t \), so the separation between zeros of \( y(t) \) should typically be

\[
\Delta t = \frac{\pi}{\omega}.
\]

Ideally, we would simply measure \( \Delta t \) and then calculate

\[
\omega = \frac{\Delta \theta}{\Delta t} = \frac{\pi}{\Delta t}. \quad (\heartsuit)
\]

In practice, however, measuring \( \Delta t \) is not so simple. We have nothing to use except for some measured points \( (t_k, y(t_k)) \), so we have no chance to find the zeros of \( y(t) \) exactly. The best we can hope for is to find the zeros approximately, collect all possible gaps between the approximate zeros,
calculate the average of all these values, and use the result to define $\Delta t$. Figure 6, below, illustrates the plan: we will get approximate zeros using linear interpolation between measured data points.

![Figure 6: Finding approximate zeros from linear interpolation](image)

We will use the average gap between red dots as our approximation for $\Delta t$. In a nutshell, most of the work in today’s lab boils down to this: \textit{find the red dots in the situation of Fig. 6, given the coordinates of the green dots.}

**ACTION ITEMS:** Roadmap for this lab

During this lab, you will build a Matlab script (a \textit{script}, not a \textit{function}) that implements the following five steps.

1. Load and plot the given $(t, y)$ data points, in green.
2. Find (approximately) all zero-crossings in the given data. Plot them all; print decimal values for some.
3. Calculate the average gap between zero-crossings, $\Delta t$, and then let $\omega = \pi / \Delta t$. Print $\omega$.
4. Construct sampled values of the function $A$ using the command

   $$A = Y ./ \sin(\omega \times T);$$

   Plot them in red, on top of the data shown in green in item 1.
5. Use interpolation to define new functions $\tilde{A}(t)$ and $\tilde{y}(t)$ that approximate the unknown $A(t)$ and $y(t)$ from which the given data are drawn. Plot these, together with the experimental data, in a new figure.

You will hand in a printed copy of the script itself, and the output it generates when applied to 3 sets of test data.

To get started, download a prototype for the desired script from Connect. It is named \texttt{demodulate.m}. The given script is packed with useful things to learn and imitate. Its overall structure and plotting commands are usable as given. Unfortunately, other parts of the script are ugly, inefficient, and/or just plain wrong. So, after you acquire a copy of the raw script, work through the detailed comments that follow to improve it and complete the five steps summarized above.
**STEP ONE: Load and plot the given data**

Copy the Matlab script `demodulate.m` and the following six files from Connect into your work space:

`test1.csv`, `test2.csv`, `test3.csv`, `submit1.csv`, `submit2.csv`, `submit3.csv`

Edit the script to insert your name and UBC ID. Notice how the script learns where to find the data values. Once you have the script working properly, the only change required to analyze a new data set should be editing the filename.

Run the script. You should get some printed information, and a usable figure showing the raw data as green circles. Check that the output is correct. This step is complete.

**STEP TWO: Find all zero-crossings in the given data.**

Remove a strategically-positioned `return` statement in the script to activate some prototype code for Step 2.

The code is supposed to populate a vector named `rootlist` with all the $t$-values where the linear interpolator $\tilde{y}(t)$ for the given data points has a zero. Right now the instructions provided here get this job wrong in a variety of ways. Replace this travesty with something that works.

This part of the given script includes some plotting commands that may help you visualize what is actually happening and compare what should be happening. The plotting commands are trustworthy; don’t change them much.

Run the script over and over again, using each of the test data files, until it behaves as expected.

**STEP THREE: Estimate the zero-spacing $\Delta t$ and angular frequency $\omega = \pi / \Delta t$.**

Remove a strategically-positioned `return` statement in the script to activate some prototype code for Step 3.

The script defines a variable named `DeltaT` corresponding to $\Delta t$ in the notes above. We want $\Delta t$ to be the average spacing between the roots found in Step 2, but the script does something worse. Fix it. Look around for other silly mistakes. Finally, test your repair job on all of the test data files provided.

The script does some data-pruning at this stage, to avoid division by 0 later. Leave this hack in place.

**STEP FOUR: Demodulate.**

Remove a strategically-positioned `return` statement in the script to activate some prototype code for Step 4. Enter the one-line command that produces amplitude values in $A$ from the (sanitized) data in $T$, $Y$, and the computed value of $\omega$.

Trust the plotting commands provided, but look at the figures you get to check your work.

**STEP FIVE: Reconstruct $y(t)$ by interpolating the amplitude.**

Remove a strategically-positioned `return` statement in the script to activate the prototype code for Step 5. This code is correct as provided.

**DELIVERABLES: What to hand in**

When you are satisfied that your script is working, print a copy to for your hand-in package. Then run the script 3 times, once for each data files with `submit` in its name. For each run, hand in

- The printed text generated by the script,
- The plot from Step 2, showing the approximate roots,
- The plot from Step 5, showing the interpolated amplitude function $\tilde{A}(t)$ and the corresponding reconstruction of $y(t)$.