**Objective:** The objective of this lab is to learn to how AM and FM signals can be synthesized. The resulting signal can be analyzed to show its time-frequency behavior by using the *spectrogram.* There are several specific steps that will be considered in this lab:

1. Synthesizing a single short-duration sinusoid with a MATLAB M-file, and adding it to an existing long signal vector.

2. Concatenating many short-duration sinusoids with different frequencies and durations.

3. *Spectrogram:* Analyzing the long (concatenated) signal to display its time-frequency spectral content.

We have spent a lot of time learning about the properties of sinusoidal waveforms of the form:

$$x(t) = A\cos(2\pi f_0 t + \varphi) = \Re\left\{\left(Ae^{j\varphi}\right)e^{j2\pi f_0 t}\right\} \tag{1}$$

Now, we will extend our treatment of sinusoidal waveforms to more complicated signals composed of sums of sinusoidal signals, or sinusoids with changing frequency, i.e., frequency-modulated sinusoids. The objective of this lab is to learn how short-duration sinusoids can be concatenated to make longer signals that "play" musical notes and dial telephone numbers. The resulting signal can be analyzed to show its time-frequency behavior by using the spectrogram.

### Summation of Sinusoidal Signals

If we add several sinusoids, each with a different frequency ($f_k$), we cannot use the phasor addition theorem, but we can still express the result as a summation of terms with complex amplitudes via:

$$x(t) = \sum_{k=1}^{N} A_k \cos(2\pi f_k t + \varphi_k) = \Re\left\{ \sum_{k=1}^{N} \left(A_k e^{j\varphi_k}\right) e^{j2\pi f_k t} \right\} \tag{2}$$

Therefore, it will be necessary to establish the connection between musical notes, their frequencies, and sinusoids. A secondary objective of the lab is to learn the relationship between the synthesized signal, its spectrogram and the musical notes. There are several specific steps that will be considered in this lab:

1. Synthesizing a single short-duration sinusoid with a MATLAB M-file, and adding it to an existing long signal vector.

2. Spectrogram: Analyzing the long (concatenated) signal to display its time-frequency spectral content.

### Beat Signals: Summing Two Sinusoids with a small Frequency Difference

In the section on beat notes in Chapter 3 of the text, we discussed signals formed as the product of two sinusoidal signals of slightly different frequencies; i.e.,

$$x(t) = Bcos(2\pi f_\Delta t + \phi_\Delta)cos(2\pi f_c t + \phi_c) \tag{3}$$

where $f_c$ is the (high) center frequency, and $f_\Delta$ is the (low) frequency that modulates the envelope of the signal. An equivalent representation for the beat signal is obtained by rewriting the product as a sum:

$$x(t) = A_1 cos(2\pi f_1 t + \varphi_1) + A_2 cos(2\pi f_2 t + \varphi_2) \tag{4}$$

It is relatively easy to derive the relationship between the frequencies $\{f_1, f_2\}$ and $\{f_c, f_\Delta\}$.

### Amplitude Modulation

By multiplying a time-varying signal $y(t)$ with a sinusoid we get an amplitude modulated (AM) signal:

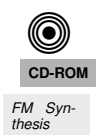
$$x(t) = y(t)cos(2\pi f_c t) \tag{5}$$

where $f_c$ is called a carrier frequency. The beat is a special type of AM signal in which we pick a sinusoid with a low signal frequency for $y(t)$ and one high carrier frequency resulting in a signal $x(t)$ with an addition of two sinusoids with individual frequencies very close together, see Chapter 3.

### Frequency Modulated Signals

In this lab, we will examine signals whose frequency varies as a function of time. Recall that in a constant-frequency sinusoid (2) the argument of the cosine is $(2\pi f_0 t + \varphi)$ which is also the exponent of the complex exponential. We will refer to the argument of the cosine as the ***angle function***. In (2), the *angle function* changes *linearly* versus time, and its time derivative, $2\pi f_0$, equals the constant frequency of the cosine.

A generalization is available if we adopt the following notation for the class of signals with time-varying angle functions (and constant amplitude):




$$x(t) = A\cos(\psi(t)) = \Re\{Ae^{j\psi(t)}\} \tag{6}$$

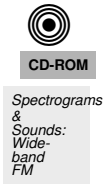
where $\psi(t)$ is the angle function. The time derivative of the angle function $\psi(t)$ in (6) gives a frequency that we call the *instantaneous radian frequency:*

$$\omega_i(t) = \frac{d}{dt}\psi(t) \qquad \text{(rad/sec)}$$

If we prefer units in hertz, then we divide by $2\pi$ to define the *instantaneous cyclic frequency:*

$$f_i(t) = \frac{1}{2\pi}\,\frac{d}{dt}\psi(t) \qquad \text{(Hz)} \tag{7}$$

### Chirp, or Linearly Swept Frequency




A linear-FM *chirp* signal is a sinusoid whose frequency changes linearly from a starting frequency value to an ending frequency. The formula for such a signal can be defined by creating a complex exponential signal with a quadratic angle function. Thus, we define $\psi(t)$ in (6) as the following quadratic function:

$$\psi(t) = 2\pi\mu t^2 + 2\pi f_0 t + \varphi$$

Using (7), we obtain an instantaneous *cyclic* frequency that changes *linearly* versus time.

$$f_i(t) = 2\mu t + f_0 \quad \text{(Hz)} \tag{8}$$

The slope of $f_i(t)$ is equal to $2\mu$ and its intercept is $f_0$. For example, if the signal starts at time $t = t_1$ s with an initial frequency of $f_1$ Hz, and ends at time $t = t_2$ s with a final frequency of $f_2$ Hz, then the slope of the line in (8) will be

$$\text{SLOPE} = 2\mu = \frac{f_2 - f_1}{t_2 - t_1} \tag{9}$$

Note that if the signal starts at time $t = 0$ s, then the intercept $f_0 = f_1$ is equal to the starting frequency.

The frequency variation produced by the time-varying angle function is called *frequency modulation,* or simply FM. Finally, since the linear variation of the frequency can produce an audible sound similar to a bird chirp, linear-FM signals are also called *chirps.*

### MATLAB Synthesis of Chirp Signals

The following MATLAB code will synthesize a linear-FM chirp:

```
fsamp = 8000;   %-Number of time samples per second
dt = 1/fsamp;
dur = 1.1;
tt = 0 : dt : dur;
f1 = 400;
psi = 2*pi*(100 + f1*tt + 500*tt.*tt);
xx = real( 7.7*exp(j*psi) );
soundsc( xx, fsamp );
```

(a) Determine the total duration of the synthesized signal in seconds, and also the length of the `tt` vector.

(b) In MATLAB signals can only be synthesized by evaluating the signal's defining formula at discrete instants of time. These are called *sample values* of the signal, or simply *samples.* For the chirp,

$$x(t_n) = A\cos(2\pi\mu t_n^2 + 2\pi f_0\, t_n + \varphi)$$

where $t_n$ is the $n^{\text{th}}$ time sample. In the MATLAB code above, identify the values of $A$, $\mu$, $f_0$, and $\varphi$.

(c) Determine the range of frequencies (in hertz) that will be synthesized by the MATLAB script above, i.e., determine the minimum and maximum frequencies (in Hz) that will be heard. This will require that you relate the parameters $\mu$, $f_0$, and $\varphi$ to the minimum and maximum frequencies. Make a sketch by hand of the instantaneous (cyclic) frequency $f_i(t)$ versus time.

(d) Use `soundsc()` to listen to the signal in order to determine whether the signal's frequency content is increasing or decreasing. Notice that `soundsc()` needs to know two things: the vector containing the signal samples, and the rate at which the signal samples are to be played out. This rate should be the same as the rate at which the signal values were created, i.e., `fsamp` in the code above.
• More information is available from `help sound` and `help soundsc` in MATLAB.

### Concatenating Signals via Addition

When we want to play several pieces of signals in succession, e.g., generating a C-Major scale with music notes, we must form a MATLAB vector to hold the signals. There are two strategies:

1. *Concatenation by appending:* if we want to play three signal vectors (`sa`, `sb` and `sc`) successively, then we can form a new longer signal vector as `ss = [ sa, sb, sc];` (assuming row vectors).

2. *Concatenation via addition into a long vector:* This technique relies on MATLAB's colon notation. If we pre-allocate a very long vector that will hold the entire concatenated signal, then we can add short signals to get concatenation. For example, if the three signal vectors (`sa`, `sb` and `sc`) have lengths (`La`, `Lb` and `Lc`), respectively, and `ss` is a very long vector initialized to zeros, then the following MATLAB code will produce the concatenation of the three signals.

```
ss(1:La) = ss(1:La) + sa;
ss(La+1:La+Lb) = ss(La+1:La+Lb) + sb;
Lab = La+Lb;
ss(Lab+1:Lab+Lc) = ss(Lab+1:Lab+Lc) + sc;
```

The drawback of the *Concatenation by appending* strategy, we studied in the previous lab, is that it cannot accommodate the common situation where we want to add together multiple signals with different durations that might overlap in time. On the other hand, the *Concatenation via addition into a long vector* method will deal naturally with overlapping signals. Furthermore, *concatenation via addition into a long vector* is more computationally efficient since the *concatenation by appending* method requires MATLAB to allocate memory and copy data to the new larger memory every time a concatenation is performed whereas the memory allocation is performed once by MATLAB with the *concatenation via addition* method.

### Synthesizing and Concatenating Sinusoids

Whenever we take samples of a continuous-time formula, e.g., $x(t)$ at $t = t_n$, we are, in effect, implementing the ideal C-to-D converter. We do this in MATLAB by first making a vector of times, and then evaluating the formula for the continuous-time signal at the sample times, i.e., $x[n] = x(nT_s)$ if $t_n = nT_s$. This assumes perfect knowledge of a formula for the input signal, but we have already been doing this in previous labs.

(a) To begin, create a vector `xx` of samples of the sum of two sinusoidal signals: the first with $A_1 = 100$, $\omega_1 = 2\pi(800)$, and $\varphi_1 = 0.6\pi$; the second with a different frequency: $A_2 = 120$, $\omega_2 = 2\pi(2000)$, and $\varphi_2 = -0.1\pi$. Use a sampling rate of 8000 samples/sec, and compute a total number of samples equivalent to a time duration of 1.2 secs. You may find it helpful to recall that the MATLAB statement

`tt=(0:0.01:3);` which creates a vector of numbers from 0 through 3 with increments of 0.01. Therefore, it is necessary to determine the time increment needed to obtain 8000 samples in one second.

```
function xs = shortSinus(amp, freq, pha, fs, dur)
% amp  = amplitude
% freq = frequency in cycle per second
% pha  = phase, time offset for the first peak
% fs   = number of sample values per second
% dur  = duration in sec
%
tt = 0 : 1/fs : dur; % time indices for all the values
xs = amp * cos( freq*2*pi*tt + pha );
end
```

This function `shortSinus` can be called once the argument values are specified. For example, the following MATLAB code will add two sinusoids—once you fill in the missing code at two places where you see `???`.

```
amps = [100, 120]
freqs = [800,2000]
phases = [0.6*pi, -0.1*pi]
fs = 8000;
tStart = [0.1, 0.1];
durs = [0.4,0.4];
maxTime = max(tStart+durs) + 0.1;  %-- Add time to show ...
    signal ending
durLengthEstimate = ceil(maxTime*fs);
tt = (0:durLengthEstimate)*(1/fs);  %-- be conservative ...
    (add one)
xx = 0*tt;   %--make a vector of zeros to hold the total ...
    signal
for kk = 1:length(amps)
  nStart = round(????)+1;  %-- add one to avoid zero index
  xNew = shortSinus(amps(kk), freqs(kk), phases(kk), fs, ...
      durs(kk));
  Lnew = length(xNew);
  nStop = ????;  %<======== Add code
  xx(nStart:nStop) = xx(nStart:nStop) + xNew;
end
plotspec(xx,fs,256); grid on
```

The starting index is an integer computed from the starting time (in secs) via the sampling rate ($f_s$). The relationship is $t = n/f_s$. In addition, if we define a subvector via the colon notation, e.g., `xx(n1:n2)`, then the length of that subvector is `n2-n1+1`. The fact that you must add one to get the length is confusing, but think of `xx(7:7)` which gives the single element `xx(7)`; its length is one.

Use `soundsc()` to play the resulting vector through the D-to-A converter of the your computer, assuming that the hardware can support the $f_s = 8000$ Hz rate. Listen to the output. Also, the *spectrogram* is an effective tool to verify the spectral content of a changing signal with multiple frequency components (see next Section and Fig. 1).

(b) ***Concatenate*** the two short sinusoidal signals defined in the previous part, and put a short duration of 0.1 seconds of silence in between, i.e., the second sinusoid will start after the first one ends. Modify the MATLAB code above to accomplish this task.

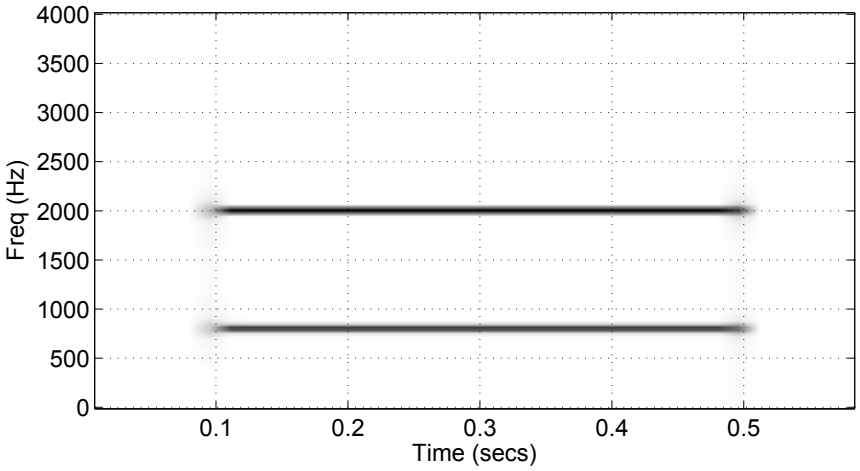



Figure 1: Spectrogram of two sinusoids from Section 2.8(a).

(c) To verify that the concatenation operation was done correctly in part b, make the following plot:

```
tt = (1/fs)*(0:length(xx)-1);   plot( tt, xx );
```

This will plot a huge number of points, but it will show the "envelope" of the signal and verify that the amplitude changes from 100 to zero and then to 120 at the correct times. Notice that the time vector `tt` was created to have exactly the same length as the signal vector `xx`.

### Debugging Skills

Testing and debugging code is a big part of any programming job. Almost any modern programming environment provides a *symbolic debugger* so that break-points can be set and variables examined in the middle of program execution. Nonetheless, many programmers still insist on using the old-fashioned method of inserting print statements in the middle of their code (or the MATLAB equivalent, leaving off a few semicolons). This is akin to riding a tricycle to commute around Atlanta.

There are two ways to use debugging tools in MATLAB: via buttons in the edit window or via the command line. For help on the edit-window debugging features, access the menu `Help->Using the M-File Editor` which will pop up a browser window at the help page for editing and debugging. For a summary of the command-line debugging tools, try `help debug`. Here is part of what you'll see:

```
dbstop     - Set breakpoint.
dbclear    - Remove breakpoint.
dbcont     - Resume execution.
dbdown     - Change local workspace context.
dbmex      - Enable MEX-file debugging.
dbstack    - List who called whom.
dbstatus   - List all breakpoints.
dbstep     - Execute one or more lines.
dbtype     - List M-file with line numbers.
dbup       - Change local workspace context.
dbquit     - Quit debug mode.

When a breakpoint is hit, MATLAB goes into debug mode, the debugger
window becomes active, and the prompt changes to a K>.  Any MATLAB
command is allowed at the prompt.
To resume M-file function execution, use DBCONT or DBSTEP.
To exit from the debugger use DBQUIT.
```

One of the most useful modes of the debugger causes the program to jump into "debug mode" whenever an error occurs. This mode can be invoked by typing:

```
dbstop if error
```

With this mode active, you can snoop around inside a function and examine local variables that probably caused the error. You can also choose this option from the debugging menu in the MATLAB editor. It's sort of like an automatic call to 911 when you've gotten into an accident. Try `help dbstop` for more information. Here is a link to a video about MATLAB's debugger:

`http://www.youtube.com/watch?v=Z4vFymKhNno`

## Spectrograms

It is often useful to think of a signal in terms of its spectrum as discussed in Chapter 3. A signal's spectrum is a representation of the frequencies present in the signal. For a constant frequency sinusoid, the spectrum consists of two spikes, one at $\omega = 2\pi f_0$, the other at $\omega = -2\pi f_0$. For a more complicated signal the spectrum may be very interesting, e.g., the case of FM, where the spectrum components are time-varying. One way to represent the time-varying spectrum of a signal is the *spectrogram* (see Chapter 3 in the text). A spectrogram is produced by estimating the frequency content in short sections of the signal. The magnitude of the spectrum over individual sections is plotted as intensity or color over a two-dimensional domain of frequency and time.

When unsure about a command, use `help`.

There are some additional important things to know about spectrograms following last week's preliminary exercises:

1. In MATLAB the function `spectrogram` will compute the spectrogram. Type `help spectrogram` to learn more about this function and its arguments. The `spectrogram` function used to be called `specgram`, and had slightly different defaults—the argument list had a different order, and the output format always defaulted to frequency on the vertical axis and time on the horizontal axis.

2. A common call to the MATLAB function is `spectrogram(xx,1024,[],[],fs,'yaxis')`. The second argument[1] is the *section length* (or window length) which could be varied to get different

[1] If the second argument of `spectrogram` is made equal to the "empty matrix" then the default value used, which is the maximum of 256 and the signal length divided by 8.

looking spectrograms. The spectrogram is able to "see" very closely spaced separate spectrum lines with a longer (window) section length,[2] e.g., 1024 or 2048.

3. **(Negative) Frequency Range:** Normally the spectrogram image contains only positive frequencies. However, you can produce a spectrogram image containing negative frequencies ***if you use the function*** `plotspec` ***and if you make the input signal complex.*** Even if your signal is real, you can add a very tiny imaginary part, e.g., `xx = xx + j*1e-14`, to make it seem to be complex-valued.
   **Warning:** This trick works nicely with the *SP-First* function called `plotspec`. However, when used with `spectrogram` the result does not have the negative frequency region in the proper location.

4. **Section Length of Spectral Analysis Windows:** A spectrogram is formed by taking successive short sections of a signal and performing an FFT analysis of each of those sections to get the spectrum. Since this is done repeatedly, the result is the spectrum versus time, where time is the location of the short sections. For a specific example, assume that the section length is 100, and the signal is a MATLAB vector `xx`. Then the first short section will be `xx(1:100)`. The sections are usually overlapped and the default in `plotspec` is 50% overlap, so the second short section is `xx(51:150)`, the third `xx(101:200)`, and so on.

   The spectrogram image is, in effect, the spectrum versus time, so we need a reference time for each short section. In `plotspec` this reference time is the "midpoint" of the section. For the length-100 section, the reference index is 50, which is then converted to a time (in secs) by using the sampling rate $(f_s)$. When the spectrogram is displayed as an image, these reference times are used along the horizontal axis.

In order to see a typical spectrogram, run the following code:

```
fs=8000; xx = cos(2000*pi*(0:1/fs:0.5));
plotspec(xx,fs,1024); colorbar
```

Notice that the spectrogram image contains one horizontal line at the correct frequency of the sinusoid. To obtain a spectrogram with negative frequencies, try the following

```
xx = cos(2000*pi*(0:1/fs:0.5));
plotspec(xx+j*1e-9,fs,1024); colorbar
```

[2] Usually the window (section) length is chosen to be a power of two, because a special algorithm called the FFT is used in the computation. The fastest FFT programs are those where the FFT length is a power of 2.

### Adding Short Sinusoid to a Long Signal Vector

For music synthesis, we need a way to add a short-duration sinusoid to an existing long vector. For example, suppose we have a MATLAB vector `xLong` which contains 100 elements, and we generate a sinusoid `xSinus` that contains 23 elements. If we want to add `xSinus` to `xLong` we cannot do `xSinus+xLong` because the dimensions do not match and MATLAB will report an error. Instead, we must find a 23-element subvector of `xLong`, and add `xSinus` to that subvector, e.g., `xSinus+xLong(31:53)`.

In music we must add the sinusoid at a location corresponding to a specified starting time, so we must know how to calculate the starting index of the subvector from a time (in secs). This time-to-index correspondence could be calculated, or it could be contained in a time vector that would accompany the signal vector. In order to calculate the index, we use the sampling relationship ($t_n = n/f_s$) and solve for the index $n$. Since the right hand side might not be an integer, in MATLAB we must round to the nearest integer (use $f_s = 4000$ Hz in the following).

$$n_{\text{START}} = (f_s)(t_{\text{START}})$$

Generate two sinusoids both with zero phase and amplitude one. The first sinusoid should start at $t = 0.6$ s, have a frequency of 1200 Hz and a duration of 0.5 s; the second, a frequency of 750 Hz and a duration of 1.6 seconds starting at $t = 0.2$ s. Use a modified form of the MATLAB code from the pre-Lab. For verification, make a spectrogram (section length = 256) of the sum signal so that you can point out features that correspond to the parameters of the sinusoids. Also, be prepared to explain the modifications you made to the MATLAB code.

### Spectrogram: Section Length and Negative Frequency

In this part, you must display the spectrogram of the signal synthesized in the previous section with different parameters. Remember that the spectrogram displays an image that shows the *frequency* content of the synthesized *time* signal. Its horizontal axis is time and its vertical axis is frequency. Use the function `specgram(xx,512,fs)`. The second argument[3] is the *window length* which could be varied to get different

[3] If the second argument is made equal to the "empty matrix" then its default value of 256 is used.

looking spectrograms. The spectrogram is able to "see" the separate spectrum lines with a longer window length, e.g., 1024 or 2048.[4]

In order to see a typical spectrogram, run the following code:

```
fs=8000; tt=0:1/fs:0.5;
xx = cos(4000*pi*tt); spectrogram(xx,1024,[],[],fs,'yaxis'); colorbar
```

or, if you are using `plotspec(xx,fs)`:

```
tt=0:1/fs:0.5; yy=xx+cos(1600*pi*tt);
plotspec(yy,fs,1024); colorbar
```

Notice that the spectrogram image now contains two horizontal line at the correct frequencies of the sinusoids similar to what was displayed in Fig. 1.
To obtain a spectrogram with negative frequencies, try the following

```
plotspec(yy+j*1e-9,fs,1024); colorbar
```

To obtain a spectrogram with a different section length, try the following

```
plotspec(yy+j*1e-9,fs,128); colorbar
```

For this verification, show the spectrogram features with negative frequencies and different section lengths to your lab instructors. You can also use the spectrogram of the DTMF digit **8** obtained in the previous lab.

### MATLAB Structure for Beat Signals

A beat signal is defined by five parameters $\{\mathrm{B}, f_c, f_\Delta, \varphi_c, \varphi_\Delta\}$ as shown in Section 2.2 so we can represent it with a MATLAB structure that has seven fields (by including start and end times),as shown in the following template:

```
sigBeat.Amp = 10; %-- B in Equation (3)
sigBeat.fc = 480; %-- center frequency in Eq. (3)
sigBeat.phic = 0; %-- phase of 2nd sinusoid in Eq. (3)
sigBeat.fDelt = 20; %-- modulating frequency in Eq. (3)
sigBeat.phiDelt = -2*pi/3; %-- phase of 1st sinusoid Eq.~(\ref{Labeq:beatSigSum})
sigBeat.t1 = 1.1; %-- starting time
sigBeat.t2 = 5.2; %-- ending time
%
%----- extra fields for the parameters in Equation (4)
%
sigBeat.f1 %-- frequencies in Equation (4)
sigBeat.f2 %--
sigBeat.X1 %-- complex amps for sinusoids in Equation (4)
sigBeat.X2 %-- derived from A's and phi's
%
sigBeat.values %-- vector of signal values
sigBeat.times %-- vector of corresponding times
```

[4]Usually the window length is chosen to be a power of two, because a special algorithm called the FFT is used in the computation. The fastest FFT programs are those where the signal length is a power of 2.

(a) Write a MATLAB function that will add fields to a `sigBeatIn` structure. Follow the template below:

```
function sigBeatSum = sum2BeatStruct( sigBeatIn )
%
%--- Assume the five basic fields are present, plus the starting and ending times
%--- Add the four fields for the parameters in Equation (4)
%
% sigBeatSum.f1, sigBeatSum.f2, sigBeatSum.X1, sigBeatSum.X2
```

(b) Produce a beat signal with two frequency components: one at 720 Hz and the other at 750 Hz. Use a longer duration than the default to hear the beat frequency sound. Use the feature discussed in Section 2.7 to generate a spectrogram plot of the beat signal here. Demonstrate the plot and sound to your lab instructor or TA.

### Beat Note Spectrograms

Beat notes have a simple time-frequency characteristic in a spectrogram. Even though a beat note signal may be viewed as a single frequency signal whose amplitude envelope varies with time, the spectrum or spectrogram requires an additive combination which turns out to be the sum of two sinusoids with different constant frequencies.

(a) Use the MATLAB function(s) written in Section 3.4 to create a beat signal defined via: $b(t) = 50cos(2\pi(30)t + \pi/4)cos(2\pi(800)t)$, starting at t = 0 with a duration of 4.04 s. Use a sampling rate of $f_s$ = 8000 samples/sec to produce the signal in MATLABUse testingBeat as the name of the MATLAB structure for the signal. Plot a very short time section to show the amplitude modulation.

(b) Derive (mathematically) the spectrum of the signal defined in part (a). Make a sketch (by hand) of the spectrum with the correct frequencies and complex amplitudes.

(c) Plot the two-sided spectrogram of using a (window) section length of 512 using the commands3:

```
plotspec(testingBeat.values+j*1e-12,fs,512); grid on, shg
```

Comment on what you see. Can you see two spectral lines, i.e., horizontal lines at the correct frequencies in the spectrum found in the previous part? If necessary, use the zoom tool (in the MATLAB figure window).

## Function for a Chirp

Use the code provided in the Pre-Lab section as a starting point in order to write a MATLAB function that will synthesize a "chirp" signal according to the template shown below. This will require that you determine the chirp parameters $\mu$, $f_0$, and $\varphi$ from the parameters passed in the LFM signal structure. Complete the function by filling in code where

```
function   sigOut = makeLFMvals( sigLFM, dt )
% MAKELFMVALS      generate a linear-FM chirp signal
%
% usage:   sigOut = makeLFMvals( sigLFM, dt )
%  sigLFM.f1 = starting frequency (in Hz) at t = sigLFM.t1
%  sigLFM.t1 = starting time (in secs)
%  sigLFM.t2 = ending time
%  sigLFM.slope = slope of the linear-FM (in Hz per sec)
%  sigLFM.complexAmp = defines the amplitude and phase of the FM signal
%  dt = time increment for the time vector, typically 1/fs (sampling ...
     frequency)
%
%  sigOut.values = (vector of) samples of the chirp signal
%  sigOut.times  = vector of time instants from t=t1 to t=t2
%
if( nargin < 2 )   %-- Allow optional input argument for dt
   dt = 1/8000;  %-- 8000 samples/sec
end
%--------NOTE: use the slope to determine mu needed in psi(t)
%--------  use f1, t1 and the slope to determine f0 needed in psi(t)
tt = ???
mu = ???
f0 = ???
psi = 2*pi*( f0*tt + mu*tt.*tt);
xx = real( ??? * exp(j*psi) );
sigOut.times = ???
sigOut.values = ???
```

*Testing:* Plot the result from the following call to test your function.

```
myLFMsig.f1 = 200;
myLFMsig.t1 = 0;  myLFMsig.t2 = 1.5;
myLFMsig.slope = 1800;
myLFMsig.complexAmp = 10*exp(j*0.3*pi);
dt = 1/8000;   % 8000 samples per sec is the sample rate
outLFMsig = makeLFMvals(myLFMsig,dt);
%- Plot the values in outLFMsig
%- Make a spectrogram for outLFMsig to see the linear frequency change
```

The test case above generates a chirp sound whose frequency starts low and chirps up. From the duration (in secs.) and the sampling rate of $f_s = 8000$ samples/s, the size of the output signal vector can be determined. Use MATLAB's `size` command to check that `outLFMsig.values` has the expected size.

Listen to the chirp using the `soundsc` function, and make a two-sided spectrogram of the signal, i.e., including the negative frequencies. You can use MATLAB's cell-mode feature to show the spectrogram within a web page. Zoom in on the beginning and end of the plot to verify that the frequencies have the values expected, and that the starting frequency is lower at the beginning than at the end of the chirp.