GEORGIA INSTITUTE OF TECHNOLOGY

SCHOOL of ELECTRICAL and COMPUTER ENGINEERING

EE 2200 Fall 1998
Lab #7: Everyday Sinusoidal Signals

Date: week of 9 Nov 1998

Lab Quiz: the next one will be the week of 17–19 Nov.

This is *the official* Lab #7 description; it is based on Lab A of Lab C.7 in Appendix C of the text, but the warm-up has been changed quite a bit.

The Warm-up section of each lab must be completed in Lab and the steps marked *Instructor Verification* must also be signed off **during the lab time**.

The lab report for this lab will be informal: discuss your results from section 4. Staple the **Instructor Verification** sheet to the end of your lab report.

The report will due during the week of **17-Nov** at the start of your lab.

---

1 Introduction

The goal of this lab is to learn how to implement FIR filters in MATLAB, and then study the response of FIR filters to inputs such as complex exponentials. In addition, we will use FIR filters to study properties such as linearity and time-invariance.

In the experiments of this lab, you will use `firfilt()`, or `conv()`, to implement filters and `freqz()` to obtain the filter's frequency response.[1] As a result, you should learn how to characterize a filter by knowing how it reacts to different frequency components in the input.

This lab introduces a practical application where sinusoidal signals are used to transmit information: a touch-tone dialer. Bandpass FIR filters can be used to extract the information encoded in the waveforms.

2 Background

2.1 Telephone Touch Tone[2] Dialing

Telephone touch pads generate *dual tone multi frequency* (DTMF) signals to dial a telephone. When any key is pressed, the tones of the corresponding column and row (in Fig. 1) are generated and summed, hence dual tone. As an example, pressing the **5** key generates a signal containing the sum of the two tones 770 Hz and 1336 Hz together.

The frequencies in Fig. 1 were chosen to avoid harmonics. No frequency is a multiple of another, the difference between any two frequencies does not equal any of the frequencies, and the sum of any two frequencies does not equal any of the frequencies.[3] This makes it easier to detect exactly which tones are present in the dial signal in the presence of line distortions.

[1]If you are working at home and do not have the function `freqz.m`, there is a substitute available called `freekz.m`. You can get it from the EE-2200 WebCT page.

[2]Touch Tone is a registered trademark

[3]More information can be found at: `http://arrow.cso.uiuc.edu/telecom/dtmf/dtmf.html`

FREQS	1209 Hz	1336 Hz	1477 Hz
697 Hz	**1**	**2**	**3**
770 Hz	**4**	**5**	**6**
852 Hz	**7**	**8**	**9**
941 Hz	*****	**0**	**#**

Figure 1: DTMF encoding table for Touch Tone dialing. When any key is pressed the tones of the corresponding column and row are generated.

2.2 DTMF Decoding

There are several steps to decoding a DTMF signal:

1. Divide the signal into shorter time segments representing individual key presses.
2. Determine which two frequency components are present in each time segment.
3. Determine which key was pressed, **0**–**9**, *****, or **#**.

It is possible to decode DTMF signals using a simple FIR filter bank. The filter bank in Fig. 2 consists of filters which each pass only one of the DTMF frequencies and whose inputs are the same DTMF signal.

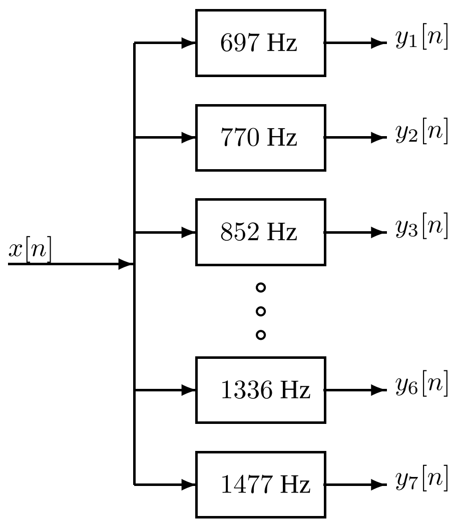

Figure 2: Filter bank consisting of bandpass filters which pass frequencies corresponding to the seven DTMF component frequencies listed in Fig. 1.

When the input to the filter bank is a DTMF signal the outputs of two filters should be larger than the rest. The two corresponding frequencies must be detected in order to determine the DTMF code. A good measure of the output levels is the average power at the filter outputs. This is calculated by squaring the filter outputs and averaging over a short time interval. More discussion of the detection problem can be found in Section 4.

2.3 Notch Filters for Rejection

Notch filters are filters that completely eliminate some frequency other than $\hat{\omega} = 0$ or $\hat{\omega} = \pi$. It is possible to make a notch filter with as few as three coefficients. If $\hat{\omega}_{\text{not}}$ is the desired notch frequency, then the following length-3 FIR filter

$$y[n] = x[n] - 2\cos(\hat{\omega}_{\text{not}})x[n-1] + x[n-2] \tag{1}$$

will have a zero at $\hat{\omega} = \hat{\omega}_{\text{notch}}$. For example, a filter designed to completely eliminate signals of the form $Ae^{j0.5\pi n}$ would have coefficients

$$b_0 = 1, \quad b_1 = 2\cos(0.5\pi) = 0, \quad b_2 = 1.$$

If the specifications are given in terms of continuous-time frequency, e.g., eliminate the spectral component at f_{not}, then we must convert to discrete-time frequency by using the frequency scaling due to sampling:

$$\hat{\omega}_{\text{not}} = 2\pi \frac{f_{\text{not}}}{f_s}$$

where f_s is the sampling frequency.

3 Warm-up: DTMF Synthesis

3.1 Signal Concatenation

In a previous lab, a very long music signal was created by joining together many sinusoids. When two signals are played one after the other, the composite signal is created by the operation of *concatenation.* In MATLAB, this can be done by making each signal a row vector, and then using the matrix building notation as follows:

```
xx = [ xx1, xx2 ];
```

where `xx1` and `xx2` are the two sub-signals. The length of the new signal is equal sum of the lengths of the sub-signals `xx1` and `xx2`. A third signal could be added later on by doing another concatenation to `xx`.

Create a signal that is made up of 5 individual sub-signals:

1. $x_1(t) = \cos(5\pi t)$ sampled at 100 Hz with length equal to *exactly* 0.3 secs.
2. $x_2(t) = 0$ sampled at 100 Hz with length 0.1 secs.
3. $x_3(t) = \cos(8\pi t)$ sampled at 100 Hz with length 0.2 secs.
4. $x_4(t) = 0$ sampled at 100 Hz with length 0.1 secs.
5. $x_5(t) = \cos(9\pi t)$ sampled at 100 Hz with length 0.3 secs.

Write the code in a loop so that one signal at a time is appended via concatenation. Make sure that your final signal has a length that is *exactly one second.* How many samples should be in the final signal vector?

Instructor Verification (separate page)

3.1.1 Comment on Efficiency

In MATLAB the concatenation method will be an *inefficient* procedure if the signal length gets to be very large. The reason is that MATLAB must re-allocate the memory space for `xx` every time a new sub-signal is appended via concatenation. If the length `xx` were being extended from 400,000 to 401,000, then a clean section of memory consisting of 401,000 elements would have to be allocated followed by a copy of the existing 400,000 signal elements and finally the append would be done. This is clearly inefficient, but would not be noticed for short signals.

An alternative is to pre-allocate storage for the complete signal vector, but this can only be done if the length is known ahead of time.

3.2 Overlay Plotting

Sometimes it is convenient to overlay information onto an existing MATLAB plot. The MATLAB command `hold on` will inhibit the figure erase that is usually done just before a new plot. Demonstrate that you can do an overlay by following these instructions:

(a) Plot the signal created in Section 3.1. Make sure that the horizontal axis has units of time in seconds.

(b) Place vertical markers at the starting times of the sub-signals, i.e., at the times $\{ 0.3, 0.4, 0.6, 0.7 \}$

```
hold on, stem([0.3, 0.4, 0.6, 0.7],1.33*ones(1,4),'.'), hold off
```

3.3 The MATLAB FIND Function

Often the signal processing functions are performed in order to extract information that can be used to make a decision. The decision process inevitably requires logical tests, which might be done with `if, then` constructs in MATLAB. However, MATLAB permits vectorization of such tests, and the `find` function is one way to do lots of tests at once. Run the following example to see how `find` works:

```
xx = 1.4:0.33:5, jkl = find(round(xx)==3), xx(jkl)
```

The argument of the `find` function can be any logical expression. See `help` on `relop` for information. Now, suppose that you have two lists of numbers:

```
xx = 1.4:0.33:5; yy = 4.7:-0.33:1.4;
```

Use the `find` command to determine the index where `xx` and `yy` are equal and the matching value.

Instructor Verification (separate page)

3.4 DTMF Dial Function

Write a function, `dtmfdial`, to implement a DTMF dialer defined in Fig. 1. A skeleton of `dtmfdial.m` including the help comments is given in Fig. 3. In this warm-up, you must complete the dialing code so that it implements the following:

1. The input to the function is a vector of numbers that ranges between 1 and 12, with 1–10 corresponding to the digits (10 corresponds to **0**), 11 is the * key, and 12 is the # key. Pay attention to the ordering of frequencies, because the **0** key is located in the middle of the bottom row.

2. The output should be a vector containing the DTMF tones, sampled at 8 kHz. The duration of each tone should be about 0.5 sec., and a silence, about 0.1 sec. long, should separate the DTMF tones. Remember that each DTMF signal is the sum of a pair of sinusoidal signals.

Your function should create the appropriate tone sequence to dial an arbitrary phone number. When played through a telephone handset, the output of your function will be able to dial the phone. You may use `specgram` to check your work.[4]

Instructor Verification (separate page)

[4]In MATLAB the demo called `phone` also shows the waveforms and spectra generated in a DTMF system.

```
function tones = dtmfdial(nums)
%DTMFDIAL  Create a vector of tones which will dial
%            a DTMF (Touch Tone) telephone system.
%
% usage:  tones = dtmfdial(nums)
%      nums  = vector of numbers ranging from 1 to 12
%      tones = vector containing the corresponding tones.
%
if (nargin < 1)
    error('DTMFDIAL requires one input');
end
fs = 8000;   %-- This MUST be 8000, so dtmfdeco( ) will work.
tone_pairs = ...
[ 697  697  697  770  770  770  852  852  852  941  941  941;
  1209 1336 1477 1209 1336 1477 1209 1336 1477 1336 1209 1477 ];
   .
   .
```

Figure 3: Skeleton of `dtmfdial.m`. A DTMF phone dialer.

4 Lab: DTMF Decoding

A DTMF decoding system needs two pieces: a bandpass filter (BPF) to isolate individual frequency components, and a detector to determine whether or not a given component is present. The detector must "score" each possibility and determine which two frequencies are most likely to be contained in the DTMF tone. In a practical system where noise and interference are also present, this scoring process is a crucial part of the system design, but we will only work with noise-free signals to understand the basic functionality in the decoding system.

4.1 Simple Bandpass Filter Design

The FIR filters that will be used in the filter bank (Fig. 2) are a simple type constructed with sinusoidal impulse responses. In the section on useful filters in Chapter 7, a *simple* bandpass filter design method is presented in which the impulse response of the FIR filter is simply a finite-length cosine of the form:

$$h[n] = \frac{2}{L}\cos\left(\frac{2\pi f_b n}{f_s}\right), \qquad 0 \le n < L$$

where L is the filter length, and f_s is the sample frequency. The parameter f_b defines the frequency location of the passband, e.g., we pick $f_b = 697$ if we want to isolate the 697 Hz component. The bandwidth of the bandpass filter is controlled by L; the larger the value of L, the narrower the bandwidth.

(a) Generate a bandpass filter, `h770`, for the 770 Hz component with $L = 32$ and $f_s = 8000$. Plot the filter coefficients in the first panel of a two-panel subplot using the `stem()` function.

(b) Generate a bandpass filter, `h1336`, for the 1336 Hz component with $L = 99$ and $f_s = 8000$. Plot the filter coefficients in the second panel of a two-panel subplot using the `stem()` function.

(c) Use the following commands to plot the frequency response (magnitude) of `h770`

```
fs = 8000;
ww = 0:(pi/256):pi;   %<-- only need positive freqs
ff = ww/(2*pi)*fs;
H = freqz(h770,1,ww);
plot(ff,abs(H)); grid on;
```

(d) Indicate the locations each of the 7 DTMF frequencies (697, 770, 852, 941, 1209, 1336, and 1477 Hz) on the plot from part (c). Hint: use the `hold` and `stem()` commands.

(e) The *passband* of the BPF filter is defined by the region of the frequency response where $|\mathcal{H}(\hat{\omega})|$ is close to one. Typically, the passband width is defined as the length of the frequency region where $|\mathcal{H}(\hat{\omega})|$ is greater than $1/\sqrt{2} = 0.707$.

The *stopband* of the BPF filter is defined by the region of the frequency response where $|\mathcal{H}(\hat{\omega})|$ is close to zero. In this case, it is reasonable to define the stopband as the region where $|\mathcal{H}(\hat{\omega})|$ is less than 0.2.

Use the `zoom on` command to show the frequency response over the frequency domain where the DTMF frequencies lie. Comment on the selectivity of the bandpass filter `h770`, i.e., use the frequency response to explain how the filter passes one component while rejecting the others. Is the filter's passband narrow enough so that only one frequency component lies in the passband and the others are in the stopband?

(f) Plot the magnitude response of the `h1336` filter and compare its passband to that of the `h770` filter. Is the passband of the `h770` filter narrow enough? Explain how the width of the passband is related to filter length L.

4.2 A Scoring Function

The final objective is decoding—a process that requires a binary decision on the presence or absence of the individual tones. In order to make the signal detection an automated process, we need a *score* function that rates the different possibilities.

(a) Complete the `dtmfscor` function based on the skeleton given in Fig. 4. The input signal `xx` to the `dtmfscor` function must be a short segment from the DTMF signal. Assume that the task of breaking up the signal so that each short segment corresponds to one key will be done by another function prior to calling `dtmfscor`.

The implementation of the FIR bandpass filter is done with the `conv` function, but we could also use `firfilt`. The running time of the convolution function is proportional to the filter length L. Therefore, the filter length L must satisfy two competing constraints: L should be large so that the bandwidth of the BPF is narrow enough to isolate individual frequencies, but making it too large will cause the program to run slowly. Try to make your system work reliably with the smallest possible value for L.

(b) Explain the last line in `dtmfscor.m` by giving the mathematical expression being calculated:

```
ss = (mean(conv(xx,hh).^2) > mean(xx.^2)/5);
```

4.3 DTMF Decode Function

The DTMF decode function, `dtmfdeco` will use `dtmfscor` to determine which key was pressed based on an input DTMF signal. The skeleton of this function in Fig. 5 includes the help comments and the table of tone pairs from Fig. 1. You must add the logic to decide which key is present.

Assume that the input signal `xx` to the `dtmfscor` function is actually a short segment from the DTMF signal. The task of breaking up the signal so that each segment corresponds to one key has already been done by the function that calls `dtmfdeco`.

There are several ways to write the `dtmfdeco` function, but you should avoid excessive use of "`if`" statements to test all 12 cases. Hint: use MATLAB's vector logicals (see `help relop` and `help find`) to implement the tests in a few statements.

```
function  ss = dtmfscor(xx, freq, L, fs)
%DTMFSCOR
%          ss = dtmfscor(xx, freq, L, [fs])
%    returns 1 (TRUE) if freq is present in xx
%            0 (FALSE) if freq is not present in xx.
%     xx = input DTMF signal
%   freq = test frequency
%      L = length of FIR bandpass filter
%     fs = sampling freq (DEFAULT is 8000)
%
%   The signal detection is done by filtering xx with a length-L
%   BPF, hh, squaring the output, and comparing with an arbitrary
%   setpoint based on the average power of xx.
%
if (nargin < 4), fs = 8000; end;

hh =           %<======== define the bandpass filter coeffs here
ss = (mean(conv(xx,hh).^2) > mean(xx.^2)/5);
```

Figure 4: Skeleton of the `dtmfscor.m` function.

```
function key = dtmfdeco(xx,fs)
   %DTMFDECO    key = dtmfdeco(xx,[fs])
   %    returns the key number corresponding to the DTMF waveform, xx.
   %     fs = sampling freq (DEFAULT = 8000 Hz if not specified.
   %
   if (nargin < 3), fs = 8000; end;
   tone_pairs = ...
   [ 697  697  697  770  770  770  852  852  852  941  941  941;
     1209 1336 1477 1209 1336 1477 1209 1336 1477 1336 1209 1477 ];
        .
        .
```

Figure 5: Skeleton of `dtmfdeco.m`.

4.4 Telephone Numbers

CD-ROM
dtmfmain.m

There is a function `dtmfmain` supplied with the *DSP First* CD-ROM that will run the entire DTMF system consisting of the three M-files you have written: `dtmfdial.m`, `dtmfscor.m`, and `dtmfdeco.m`. The function `dtmfmain` works as follows: first, it takes the signal produced by `dtmfdial` and breaks it down into individual segments; for each segment, it calls the user-written `dtmfdeco` function to determine the key for that segment; finally, its output is the list of decoded keys. An example of using `dtmfmain` is shown here:

```
>> dtmfmain( dtmfdial([1:12]) )
ans =
    1    2    3    4    5    6    7    8    9   10   11   12
```

For this function to work correctly, all three M-files must be on the MATLAB path. It is also essential to have short pauses in between the tone pairs so that `dtmfmain` can parse out the individual signal segments.

If you are presenting this project in a lab report, the `dtmfmain` function can be used to demonstrate a working version of your programs by running it several times with a variety of phone numbers.

Lab #7
EE-2200
Fall-1998
INSTRUCTOR VERIFICATION PAGE

Staple this page to the end of your Lab Report.

Name: ______________________________ Date of Lab: __________

Part 3.1: Making long signals by concatenating segments:

Verified:____________________ Date/Time:__________

Part 3.3: Using the MATLAB `find` function to do a logical match:

Verified:____________________ Date/Time:__________

Part 3.4: Complete `dtmfdial.m`:

Verified:____________________ Date/Time:__________