GEORGIA INSTITUTE OF TECHNOLOGY

SCHOOL of ELECTRICAL and COMPUTER ENGINEERING

ECE 2026 Fall 2013

Lab #10: Everyday Sinusoidal Signals & Filtering

Date: 11-14 Nov 2013

Each Lab assignment in ECE2026 consists of three parts: Pre-Lab, In-lab Tasks, and Take-home Questions. It requires you to come into lab prepared. Be sure to read the entire lab carefully before arriving.

Pre-Lab: You should read the Pre-Lab section of the lab and go over all exercises in this section before going to your assigned lab session. Although you do not need to turn in results from the Pre-Lab, doing the exercises therein will help make your in-lab experience more rewarding and go more smoothly with less frustration and panicking.

In-lab Tasks and Verification: There are a number of designated tasks for each student to accomplish during the lab session. Students are encouraged to read, prepare for and even attempt at these tasks beforehand. These tasks must be completed **during your assigned lab time** and the steps marked *Instructor Verification* must also be signed off **during the lab time**. One of the laboratory instructors must verify the appropriate steps by signing on the **Instructor Verification** line. When you have completed a step that requires verification, simply put a plastic cup on top of your PC and demonstrate the step to one of the TAs or the professor. (You can also use the plastic cups to indicate if you have a more general question, i.e. you can use it to get our attention even if you don't have an Instructor Verification ready.)

Take-home Questions: At the end of each lab sheet below all the verification steps, several questions are to be answered by the student, who can choose to complete the answers while in the lab or after leaving the lab.

The lab sheet with all verification signatures and answers to the questions needs to be turned in to the Lab-grading TA at the beginning of the next lab session.

1 Introduction

This lab introduces a practical application where sinusoidal signals are used to transmit information: a touch-tone dialer. The information that a touch-tone dialer is used to transmit includes the digits, from 0 to 9, four letters, "A", "B", "C", and "D", and two special characters, "#" and "*". Bandpass, lowpass or highpass FIR filters can be used to extract the information encoded in the waveforms. The goal of this lab is to implement the appropriate FIR filters in MATLAB so as to enable information-decoding. 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.

1.1 Background: Telephone Touch Tone Dialing

Telephone touch-tone[2] pads generate *dual tone multiple frequency* (DTMF) signals to dial a telephone number. When any key is pressed, the sinusoids of the corresponding row and column frequencies (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 at 770 Hz and 1336 Hz together.

The frequencies in Fig. 1 were chosen (by the design engineers) to avoid harmonics. No frequency is an integer 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

[1]If you do not have the function **`freqz.m`**, there is a substitute called **`freekz.m`** in the *SP-First* toolbox.

[2]Touch Tone is a registered trademark.

[3]More information can be found at: http://www.genave.com/dtmf.htm, or search for "DTMF" on the internet.

exactly which tones are present in the dialed signal in the presence of non-linear line distortions[4].

FREQS	1209 Hz	1336 Hz	1477 Hz	1633 Hz
697 Hz	1	2	3	A
770 Hz	4	5	6	B
852 Hz	7	8	9	C
941 Hz	*	0	#	D

Figure 1: Extended DTMF encoding table for Touch Tone dialing. When any key is pressed the tones of the corresponding column and row are generated and summed. Keys A-D (in the fourth column) are not implemented on commercial and household telephone sets, but might be used in some special signaling applications, e.g., military communications.

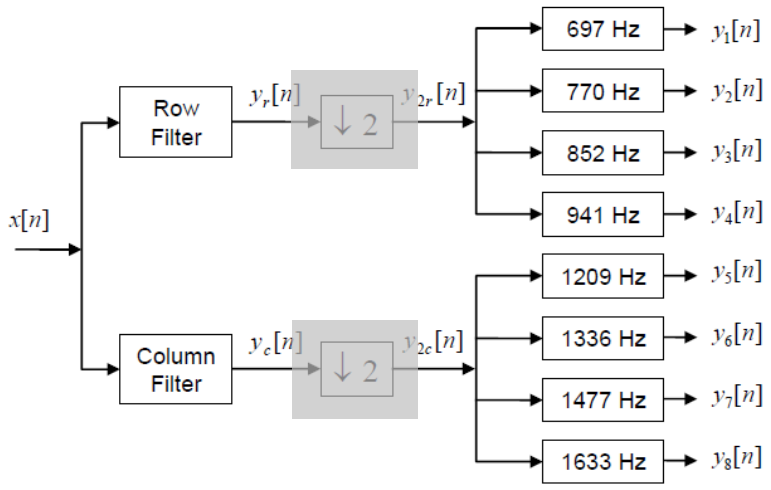

Figure 2: Filter bank consisting of three stages: row/column filters to separate the dual tones, down-samplers (denoted by ↓2, disabled in the current lab), and then frequency estimators, that determine the frequency corresponding to the individual sinusoidal components of the DTMF signal as listed in Fig. 1.

1.2 DTMF Decoding

Several effective algorithms exist in the commercial world for decoding touch-tone signals. Here is a typical approach which involves the following steps:

1) Filter the signal to separate the possible frequency components into row (low) and column (high) components.
2) Determine row (or column) frequencies by using the corresponding set of band-pass filters (or some other frequency estimation algorithms); a narrow band-pass filter can be used to detect the presence of a sinusoidal component by measuring the level of the corresponding output.
3) Determine the short time intervals where *distinct* keys have been pressed. Gaps between separate key presses must be detected.
4) Determine which key was pressed, 0–9, A–D, *, or # by converting frequency pairs back into key names according to Fig. 1.

[4]A paper on a DSP implementation of the DTMF decoder, "A low complexity ITU-compliant dual tone multiple frequency detector", by Dosthali, McCaslin and Evans, in *IEEE Trans. Signal Processing*, March, 2000, contains a short discussion of the DTMF signaling system. You can get this paper on-line from the GT library, and you can also get it at **`http://www.ece.utexas.edu/~bevans/papers/2000/dtmf/index.html`**.

Note: In some algorithms, Step 1 and 2 above may be combined. Such algorithms are designed to simultaneously detect the presence of multiple sinusoids.

2 Pre-Lab

2.1 Bandpass Filter Design

As mentioned in 1.2, appropriate filters have to be designed for the detection of sinusoids. The **`filterdesign`** GUI is specifically used for this purpose. However, you'll learn the details of and how to use **`filterdesign`** in the next lab as it requires some additional knowledge about system representations. For this lab, the two filters that separate row and column frequencies will be provided in the support files, **`row_filter.txt`** and **`col_filter.txt`**, available for download from t-square. You can easily load these filter coefficients to the MATLAB workspace.

2.2 Signal Concatenation

In one of the previous labs, a long music signal was created by joining together many sinusoids. When two signals are played one after the other, the composite signal could be 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 = [ xx, xxnew ];
```

where **`xxnew`** is the sub-signal being appended. The length of the new signal is equal to the sum of the lengths of the two signals **`xx`** and **`xxnew`**. A third signal could be added later on by concatenating it to **`xx`**. In this lab, this type of signal concatenation is adequate as no overlapping of successive signal segments is needed (different from the case of repeating speech epochs at a certain frame rate in Lab 7).

Comment on Efficiency

In MATLAB the concatenation method, **`xx = [ xx, xxnew ];`** would append the signal vector **`xxnew`** to the existing signal **`xx`**. However, this becomes an *inefficient* procedure if the signal length gets to be very large. The reason is that MATLAB must re-allocate the memory space for the vector **`xx`** every time a new sub-signal is appended via concatenation. If the length of **`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 final signal length is known ahead of time.

2.2 Dual Tone Signals

For the DTMF synthesis each key-press generates a signal that is the sum of two sinusoids. For example, when the key 7 is pressed, the two frequencies are 852 Hz and 1209 Hz, so the generated signal is the sum of two sinusoids which could be created in MATLAB via

```
fsamp = 4000;
tt = 0:1/fsamp:0.5;
xx = cos(2*pi*852*tt+rand(1)) + cos(2*pi*1209*tt+rand(1)); %-Use random phases
soundsc(xx,fsamp); %-Listen to the DTMF signal
plotspec(xx,fsamp); %-you can also use "spectrogram" to view its spectrogram
```

In the above sample code, a sampling rate of 4000 samples/s is used. Since the highest frequency among all row and column frequencies is 1633Hz, this sampling rate is sufficient. In the subsequent exercises, you can stay with this sampling rate. When playing back with **`soundsc`**, make sure to specify the right sampling rate.

2.3 Encoding from a Table

Challenge yourself: Explain how the following program uses frequency information stored in a table to generate a long signal via concatenation. Determine the size of the table and all of its entries, and then state the playing order of the frequencies. Determine the total length of the signal played by the **soundsc** function. How many samples and how many seconds?

```
ftable = [1;3;2;4;5]*[140,150]
fs = 4000;
xx = [ ];
keys = rem(2:2:20,10) + 1;
for ii = 1:length(keys)
      kk = keys(ii);
      xx = [xx,zeros(1,400)];
      krow = ceil(kk/2);
      kcol = rem(kk-1,2) +1;
      xx = [xx, cos(2*pi*ftable(krow,kcol)*(0:1000)/fs) ];
end
soundsc(xx,fs);
spectrogram(xx,256,128,256,fs,'yaxis');
```

2.4 The `find` Command

MATLAB **find** command is very useful in locating elements in an array that satisfy a specified condition. the code:

```
ind = find(X);
```

locates all nonzero elements of array **X**, and returns the linear indices of those elements in vector **ind** and

```
[row,col] = find(X, ...);
```

returns the row and column indices of the nonzero entries in the matrix **X**. Use **help** to learn more about this command.

Of specific interest to us is to find the location of an entry of **X** that has a certain value. For example,

```
PadKeys = ['1','2','3','A';
           '4','5','6','B';
           '7','8','9','C'];
[jrow,jcol] = find('A'==PadKeys)
jrow =
           1
jcol =
           4
```

The output is the row (**jrow**) and column (**jcol**) position of the element in **PadKeys** that equals the character **'A'**.

2.5 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 magnitude response of the 7-point averager, created from

```
HH = freqz((1/7)*ones(1,7),1,ww); plot(ww,abs(HH))
```

Make sure to properly define **ww** so that the horizontal frequency axis extends from $-\pi$ to $+\pi$.

(b) Use the **stem** function to place vertical markers at the zeros of the frequency response.

```
        hold on, stem(2*pi/7*[-3,-2,-1,1,2,3],0.3*ones(1,6),'r.'), hold off
```

2.6 Plotting Multiple Signals – Information Only

The MATLAB function **`strips`** is a good way to plot several signals at once, e.g., the outputs from the row and column filters. Observe the plot(s) made by **`strips(cos(2*pi*linspace(0,1,201)'*(4:10)));`** In the *SP-First* toolbox, the function **`striplot`** can be used to plot multiple signals contained in the columns of a matrix via: **`striplot(xmat,fs,size(xmat,1))`**.

3 Lab Exercises

3.1 DTMF Synthesis: Touch-Tone Dial Function

Download from t-square an incomplete code file, **`DTMFdial.m`**, which implements a Touch-Tone dialer based on the frequency table defined in Fig. 1. The code is incomplete, but the beginning of the code reads like:

```
function xx = DTMFdial(keyNames,fs)
%DTMFDIAL Create a signal vector of tones that will dial
% a DTMF (Touch Tone) telephone system
%
% usage: xx = DTMFdial(keyNames,fs)
% keyNames = vector of CHARACTERS containing valid key names
% fs = sampling frequency
% xx = signal vector that is the concatenation of DTMF tones.
TT.keys = ['1','2','3','A';
           '4','5','6','B';
           '7','8','9','C';
           '*','0','#','D'];
TT.colTones = [1209,1336,1477,1633];
TT.rowTones = [697,770,852,941];
    .....
    .....
....
```

You are asked to complete this function by filling in proper code, marked by **`???`**, to generate the dual-tone sinusoids. The vector of characters needed for **`keyNames`** is a string, e.g., **`'666788799004'`**.

In this exercise, you must complete the dialing code so that it implements the following:

1) The input to the function is a vector of characters, each one being equal to one of the key names on the telephone. The *n*-th character is **`keyNames(n)`**. The MATLAB structure called **`TT`** contains the key names in the field **`TT.keys`** which is a 4 × 4 matrix that corresponds exactly to the keyboard layout in Fig. 1. To convert any key name to its corresponding row-column indices, consider the following example:

   ```
   [jrow,jcol] = find('3'==TT.keys)
   ```

2) The output should be a vector of samples (based on an appropriately chosen sampling frequency f_s) containing the DTMF sinusoids—each key being the sum of two sinusoids. Remember that each DTMF signal is the sum of a pair of (equal amplitude) sinusoidal signals. The duration of each tone pair should be exactly 200ms and a gap of silence, exactly 80ms long, should separate the DTMF tone pairs. These times are declared as fixed code in **`DTMFdial`**.

3) The frequency information is given as two vectors (**`TT.colTones`** and **`TT.rowTones`**): one contains the column frequencies, the other has the row frequencies. You can translate a key such as the "6" key into the correct location in these 4 × 4 matrices by using MATLAB's **`find`** function. (See 1 above.) For example, the "6" key is in row 2 and column 3, and with the row-column positions, we would generate sinusoids with frequencies equal to **`TT.colTones(3)`** and **`TT.rowTones(2)`**. Also, consult the MATLAB code in **`DTMFdial.m`**.

4) You should implement error checking so that an illegitimate key name is rejected.

Your function should create the appropriate tone sequence to dial an arbitrary phone number. In fact, when played through a speaker into a telephone handset, the output of your function will be able to dial the phone. You could use **`spectrogram`** or **`plotspec`** to check your work[5]. Generate a DTMF signal for the key sequence "**`33312219966`**" and play it to the instructor. Also show a spectrogram of the generated signal to the instructor. Keep the output for use in 3.3.

Instructor Verification (separate page)

3.2 Bandpass Filters

Two filters have been designed for this exercise. These are two bandpass filters (BPFs) designed to separate the row tones (697, 770, 852, 941) and column tones (1209, 1336, 1477, 1633). These are FIR filters. The coefficients are stored in two **`txt`** files, **`row_filter.txt`** and **`col_filter.txt`**, which can be read or loaded to the MATLAB workspace as two arrays. Let's call these arrays **`row_filter`** and **`col_filter`**, respectively.

Use **`freqz`** to obtain and plot the frequency response of these two filters. Complete the following code and plot the frequency response in two separate graphs, one for the magnitude response and the other the phase response, for the two filters, respectively. Each pair of the magnitude response and the phase response must be plotted in one figure; the upper panel for the magnitude response and the lower for the phase. Use **`figure`** to retain plots, which upon completion must be shown to the instructor for verification. Point out the locations of the row tone frequencies and the column tone frequencies.

```
ww = -pi:pi/100:pi;
HH = freqz(??,??,ww);
plot(??,??)
```

Instructor Verification (separate page)

3.3 Separating Row Tone from Column Tone

Now filter the signal **`xpx`** (corresponding to `'33312219966'`) with the two BPFs. In order to verify that the filter worked properly, plot spectrogram(s) of the input and output signals. If the column frequencies have been removed from the output signal through use of the **`row_filter`**, there should be only one row frequency present at a time. You can use either **`firfilt()`** or **`conv()`** commands to implement the filtering operation. For example,

```
yy(:,1) = conv(row_filter, xpx);           %retain row tones
spectrogram(yy(:,1), ?, ?, ..., 'yaxis')  %what do you see?
yy(:,2) = conv(col_filter, xpx);           %retain column tones
spectrogram(yy(:,2), ?, ?, ..., 'yaxis')  %what do you see?
```

Show the two spectrograms for the row-column separated signals, respectively, to the instructor.

Instructor Verification (separate page)

3.4 Estimation of Sinusoidal Frequency

This lab has been simplified to suit the time limit of a lab session. We are only going to decode one key at a

[5] In MATLAB the demo called **`phone`** also shows the waveforms and spectra generated in a Touch-Tone system.

time. Therefore, in this section, use the previously written function **`DTMFdial`** to generate a DTMF signal for a single key for each test or demo.

The row-column separation filter is employed to split the generated DTMF signal into two separate signals to ensure that each component signal contains only one sinusoid. This will make frequency estimation straightforward. A MATLAB function called **`onefreq.p`** is provided to you for doing frequency estimation. The file you'll download from t-square is a p-file which is a protected m-file; the calling sequence is the same though as for an m-file. The synopsis for **`onefreq.p`** can be found in Fig. 3.

```
Function omegahat = onefreq( xn )
%ONEFREQ determine the freq when x[n] is one sinusoid
%
% usage: omegahat = onefreq( xn )
%
% xn = input signal (must contain at least 5 points)
% omegahat = frequency (between 0 and pi) when x[n] is A*cos(omegahat*n+phi)
```

Figure 3: Synopsis for the **`onefreq.p`** function. The input **`xn`** should be a vector of data points from a portion of the single frequency sinusoid. The output is given as $\hat{\omega}$; further calculations will be needed to relate the $\hat{\omega}$ value to one of the DTMF frequencies.

The frequency estimator in **`onefreq`** requires a minimum of five data points to produce a valid answer, but can take an **`xn`** vector of any length. If there is no noise or interference in the signal, short segments would be fine, but for noisy signals, longer data vectors produce better results. Since each signal array we generate contains only one pair of frequencies, which have also been separated by the row-column band-pass filters, it is alright to use the entire signal array (but omit the silence part) as input to **`onefreq.`** Perform the following steps:

(a) Create a single DTMF tone for one key. Filter the signal through the row filter and the column filter, respectively. Put the filtered results in **`yy(:,1)`** and **`yy(:,2)`**, respectively; the row-filtered in the first column and the column-filtered in the second.

(b) Then call **`onefreq`** using the filtered signal as input to produce a frequency estimate. It may be better to use only the signal portion (where does it begin?), but it is fine in the current context not to particularly worry about it. Verify and explain via hand calculations that you obtained the correct frequency from **`onefreq`**.

(c) Now write a function **`DTMF1key.m`** to decode a single key. A sample of the code can be downloaded from t-square. You need to fill in those **`???`** lines to complete the code. Here is a skeleton of the code:

```
function keyid = DTMF1key(yy,fs)
%DTMF1key
% yy = filtered signal that is guaranteed to contain only one sinusoid
%            put the row-filtered signal yyrow in the first column,
%            and the column-filtered signal yycol in the 2nd column
% TT = structure containing freqs and key names
% omegahats = estimated frequencies
% fs = sampling frequency
% keyid = detected key
%
TT.keys  =  ['1','2','3','A';
             '4','5','6','B';
             '7','8','9','C';
             '*','0','#','D'];
TT.colTones = [1209,1336,1477,1633];
TT.rowTones = [697,770,852,941];
```

```
…
%
% retrieve the character at the right location in TT.keys
%
keyid = TT.keys(jrow,jcol);
end
```

Figure 4: Skeleton of the **`DTMF1key.m`** function. Complete this function with additional lines of code.

Note: the frequency estimate produced by **`onefreq`** will not necessarily be an exact match to one of the eight DTMF frequencies. You should allow some tolerance in the frequency match during decoding; e.g., to declare a match when the frequency estimate is within 2% of a target. A sample code component may contain:

```
abs(onefreq_est-DTMF_freq)<0.02*DTMF_freq
```

Complete the code and test a couple of keys with the following sample wrapper code:

```
xpx = DTMFdial('A',fs);
yy1(:,1) = conv(row_filter,xpx);
yy1(:,2) = conv(col_filter,xpx);
keyed = DTMF1key(yy1,fs)
```

Instructor Verification (separate page)

Lab #10
ECE-2026 Fall-2013
Lab Sheet

For each verification, be prepared to explain your answer and respond to other related questions that the lab TA's or professors might ask. Turn this page in at the beginning of your next lab period.

Name: ________________________________ Date of Lab: ______________________

Part 3.1: Complete the dialing function **`DTMFdial.m`**. Listen to a phone number. Show a spectrogram.

Verified: ___________________________ Date/Time:______________________

Part 3.2: Display the BPF frequency response, as instructed, for the two filters.

(a) Verify that all the column frequencies would be passed by the **`col_filter`** BPF, and all the row frequencies rejected when f_s = 4000 Hz.
(b) Repeat (a) for all the row frequencies.

Verified: ___________________________ Date/Time:______________________

Part 3.3: Make a Spectrogram of the BPF output signal (at f_s= 4000 Hz) to verify that only the column frequency components survive the filtering process.

Verified: ___________________________ Date/Time:______________________

Part 3.4: Show and test the completed code; verify that the decoded results are correct.

Verified: ___________________________ Date/Time:______________________

Question: The frequency estimator **`onefreq`** will not be able to produce an accurate result when the signal contains substantial noise or more than one sinusoid. In this lab, we used two filters to ensure that in each output there is only one frequency to estimate. If we do not use these filters and want to resort to a single frequency estimator to estimate all likely frequencies in the signal, how would you go about designing such a frequency estimator? Conceive a solution and discuss why you think it will work. (This is posed as an engineering question that may have many plausible answers. Just present your thought and try to make sense.)