GEORGIA INSTITUTE OF TECHNOLOGY
SCHOOL of ELECTRICAL and COMPUTER ENGINEERING

ECE 2025 Spring 2006
Lab #10: Everyday Sinusoidal Signals

Date: 5–11 April 06

******* Lab #10 will be graded out of 150 points. *******

You should read the Pre-Lab section of the lab and do all the exercises in the Pre-Lab section before your assigned lab time. You **MUST** complete the online Pre-Post-Lab exercise on Web-CT at the beginning of your scheduled lab session. You can use MATLAB and also consult your lab report or any notes you might have, but you cannot discuss the exercises with any other students. You will have approximately 20 minutes at the beginning of your lab session to complete the online Pre-Post-Lab exercise. The Pre-Post-Lab exercise for this lab includes some questions about concepts from the previous Lab report as well as questions on the Pre-Lab section of this lab.

The Warm-up section of each lab must be completed **during your assigned Lab time** and the steps marked *Instructor Verification* must also be signed off **during the lab time**. After completing the warm-up section, turn in the verification sheet to your TA.

Forgeries and plagiarism are a violation of the honor code and will be referred to the Dean of Students for disciplinary action. You are allowed to discuss lab exercises with other students and you are allowed to consult old lab reports, but you cannot give or receive written material or electronic files. Your submitted work should be original and it should be your own work.

FORMAL Lab Report: You must write a formal lab report that describes your system for DTMF decoding (Section 4). The report is due during the week of 12–18 April.

---

1 Introduction

This lab introduces a practical application where sinusoidal signals are used to transmit information: a touch-tone dialer and decoder. Bandpass FIR or IIR filters can be used to extract the information encoded in the waveforms. The goal of this lab is to design and implement bandpass IIR filters in MATLAB, and do the decoding automatically. In the experiments of this lab, you will use `filter()` 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. 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 exactly which tones are present in the dialed signal in the presence of non-linear line distortions. [4]

---

[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.

[4] A recent paper on a DSP implementation of the DTMF decoder, "A low complexity ITU-compliant dual tone multiple

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.

1.2 DTMF Decoding

There are several steps to decoding a DTMF signal:

1. Filter the signal to extract the possible frequency components. Bandpass filters can be used to isolate the sinusoidal components.
2. Determine the short time intervals where *distinct* keys have been pressed. Gaps between separate key presses must be detected, and then a starting and stopping time can be found for each time interval.
3. Determine which two frequency components are present in each time interval by measuring the size of the output signal from all of the bandpass filters during that time.
4. Determine which key was pressed, **0–9**, **A–D**, *, or **#** by converting frequency pairs back into key names according to Fig. 1.

It is possible to decode DTMF signals using a simple filter bank, as shown in Fig. 2, consisting of eight bandpass filters—each one passing only one of the eight possible DTMF frequencies. The input signal for all the filters is the same $x[n]$ signal which contains the DTMF signals.

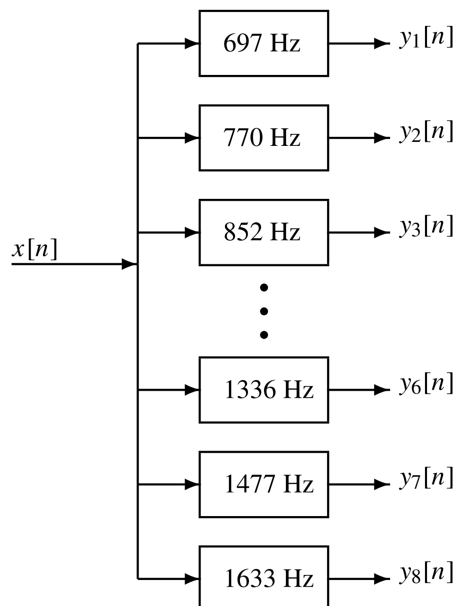

Figure 2: Filter bank consisting of bandpass filters (BPFs) that pass frequencies corresponding to the eight DTMF component frequencies listed in Fig. 1. The number in each box is the *center frequency* of the passband of the BPF.

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.utexas.edu/academic/otl/SpecSheets/DTMFdetection.html`.

Here is how the system should work: When the input to the filter bank is a DTMF signal, the outputs from two of the bandpass filters (BPFs) should be much larger than the rest of the BPF outputs. If we detect (or measure) which two outputs are the large ones, then we know the two corresponding frequencies. These frequencies are then used as row and column pointers to determine the key from the DTMF encoding table. A good measure of the output levels is the *average energy* at the filter outputs, because when the BPF is designed correctly and is working properly it should pass only one sinusoidal signal and the average energy would be high when the sinusoid is passed by the filter. More discussion of the detection problem can be found in Section 4.

2 Pre-Lab

2.1 Bandpass Filter Design

You will need a bandpass filter design function for this lab. In a previous lab, you experimented with second-order IIR filters whose frequency response looks like Fig. 3.

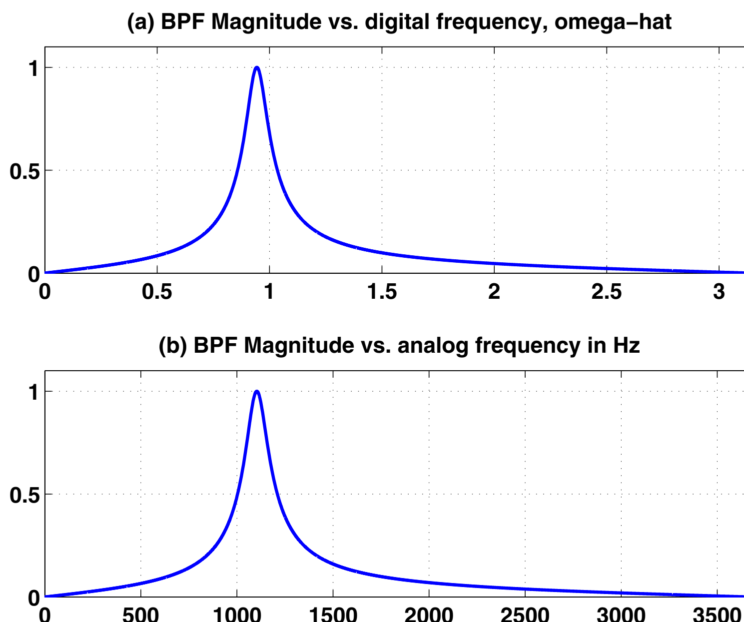

Figure 3: Frequency response of a second-order IIR bandpass filter (BPF) created in MATLAB with numerator `bb = 0.01*[1,0,-1];` and denominator `aa = poly(0.95*exp(j*0.3*pi*[1,-1]));`. (a) $\left|H(e^{j\hat{\omega}})\right|$ plotted versus $\hat{\omega}$, and (b) the same frequency response versus analog frequency f (Hz), assuming that $f_s = 7350$ Hz.

(a) Write a few lines of MATLAB code to make the plot in Fig. 3(a).

(b) Modify the code in part (a) to change the frequency axis to get the plot in Fig. 3(b). Recall that the relationship between analog frequency and digital frequency is $\hat{\omega} = 2\pi(f/f_s)$.

2.2 Signal Concatenation

In a previous lab, a long music signal was created by joining together many individual notes. 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.

2.2.1 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 in MATLAB 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 (or can be estimated) ahead of time.

2.2.2 Encoding from a Table

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;2;3;4;5]*[80,110]
fs = 4000;
xx = [ ];
disp('--- Here we go through the Loop ---')
keys = rem(2:11,10) + 1;
for ii = 1:length(keys)
   kk = keys(ii);
   xx = [xx,zeros(1,1102)];
   krow = ceil(kk/2);
   kcol = rem(kk-1,2) +1;
   xx = [xx, cos(2*pi*ftable(krow,kcol)*(0:3307)/fs) ];
end
soundsc(xx,fs);
```

2.3 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 )
```

Make sure 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
```

2.4 Plotting Multiple Signals

The MATLAB function `strips` is a good way to plot several signals at once, e.g., the eight outputs from the BPFs. 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 Warm-up: DTMF Synthesis

3.1 Touch-Tone Dial Function

Write a function, `dtmfdial.m`, to implement a Touch-Tone dialer based on the frequency table defined in Fig. 1. A skeleton of `dtmfdial.m` is given in Fig. 4.

```
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.
%
dtmf.keys = ['1','2','3','A';
             '4','5','6','B';
             '7','8','9','C';
             '*','0','#','D'];
dtmf.colTones = ones(4,1)*[1209,1336,1477,1633];
dtmf.rowTones = [697;770;852;941]*ones(1,4);
```

Figure 4: Skeleton of `dtmfdial.m`, a DTMF phone dialer. Complete this function by adding more code.

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 characters, each one being equal to one of the key names on the telephone. The MATLAB structure called `dtmf` contains the key names in the field `dtmf.keys` which is a 4×4 matrix that corresponds exactly to the keyboard layout in Fig. 1.

2. The output should be a vector of samples (at $f_s = 7350$ Hz) containing the dual-tone 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 0.25 sec., and a gap of silence, exactly 0.1 sec. long, should separate the DTMF tone pairs.

3. The frequency information is given as two 4×4 matrices (`dtmf.colTones` and `dtmf.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. For example, the **6** key is in row 2 and column 3, so we would generate sinusoids with frequencies equal to `dtmf.colTones(2,3)` and `dtmf.rowTones(2,3)`.

   To convert any key name to its corresponding row-column indices, consider the following example:

   ```
   [ii,jj] = find('3'==dtmf.keys)
   ```

   Also, consult the MATLAB code in Section 2.2 above and modify it for the 4×4 tables 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 sound output of your function will be able to dial the phone. You could use `specgram` or `plotspec` to check your work.[5]

Instructor Verification (separate page)

3.2 Bandpass Filter Design

In a previous lab, you experimented with IIR filters where pole pairs were used to make bandpass filters according to the formula:

$$H(z) = G\frac{(1 - z^{-1})(1 + z^{-1})}{(1 - p_1 z^{-1})(1 - p_1^* z^{-1})} \tag{1}$$

where p_1 and p_1^* are the (complex-conjugate) poles. The parameter G can be adjusted to make the maximum magnitude of $H(e^{j\hat{\omega}})$ equal to one. The design parameters are the radius (r) and angle (θ) of the poles $p_1 = re^{j\theta}$. The angle (θ) of the pole controls the location of the passband; the radius (r) controls the width of the passband and the stopband locations.

Filter Specifications: Generate a bandpass filter with a passband centered at 1000 Hz when the sampling rate is $f_s = 7350$ Hz. Furthermore, make the two stopbands of the filter be $0 \le f \le 800$ Hz and $1200 \le f \le \frac{1}{2} f_s$ Hz.

(a) In order to carry this out, it is necessary to convert the passband center ($\hat{\omega}_{pc}$), the lower stopband $[0, \hat{\omega}_{s_1}]$ and the upper stopband $[\hat{\omega}_{s_2}, \pi]$ into values along the $\hat{\omega}$-axis. Determine those three values.

(b) Determine the angle of the poles (θ) from the desired location of the passband center ($\hat{\omega}_{pc}$).

(c) Make the pole radius $r = |p_1|$ equal to 0.95, and design a BPF with the correct passband center (but don't worry about the stopbands yet). Plot the frequency response (magnitude) of the resulting BPF versus $\hat{\omega}$, and verify that it has the correct passband location.
Reminder: The *passband* of the BPF filter is defined by the region of the frequency response where $|H(e^{j\hat{\omega}})|$ is close to its maximum value of one. In this case, the passband width is defined as the length of the frequency region where $|H(e^{j\hat{\omega}})|$ is greater than 0.707 and less than or equal to 1.
Note: you could use MATLAB's `find` function to locate those frequencies where the magnitude of $H(e^{j\hat{\omega}})$ satisfies $\big||H(e^{j\hat{\omega}})| - 1\big| \le 0.293$.

(d) From the plot of the frequency response (magnitude) of the BPF in the previous part, measure the actual stopband locations, i.e., determine the stopband cutoff frequencies in $\hat{\omega}$.
Note: The *stopband* of the BPF filter is the region of the frequency response where $|H(e^{j\hat{\omega}})|$ is close to zero. In this case, we will define the stopband as the region where $|H(e^{j\hat{\omega}})|$ is less than 0.1.

Instructor Verification (separate page)

(e) Since the sampling rate is $f_s = 7350$ Hz, the frequency response of the digital BPF can be plotted versus analog frequency. Make this plot and show that the passband is at the correct frequency location (in Hz) for this bandpass filter. In addition, determine the actual stopband edges in Hz.

Instructor Verification (separate page)

(f) The filter with $r = 0.95$ does not meet the desired specifications. Therefore, increase (or decrease) the pole radius to create an IIR filter that will exactly meet the specifications given above.

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

4 Lab: Touch-Tone Decoding

A Touch-Tone decoding system processes a signal that is a sequence of sounds, each one being the sum of two sinusoidal components chosen from the fixed set of possible DTMF frequencies. The Touch-Tone decoding system needs two modules: a set of bandpass filters (BPFs) to isolate individual frequency components, and a detector to determine whether or not a given component is present. The number of BPFs is equal to the number of possible DTMF frequencies, in this case eight. The detector must compare all the BPF outputs and determine which two frequencies are the most likely ones to be contained in the Touch-Tone signal. In a practical system where noise and interference are also present, the hardest part of this detection process is finding the temporal locations of the tones. The detection strategy to be implemented in this lab relies on the fact that there will be silence between any two consecutive DTMF signals, and the duration of the silence will be at least 10 millisec. Therefore, the detector can check the filter outputs every 10 millisecs to make a decision. Note, however, that the raw filter outputs are oscillating and must be averaged prior to saying whether or not the sinusoid is present, so the average energy over the past 10 millisecs is calculated and compared to a threshold. If the individual DTMF signals last longer than 10 millisecs, then the same detection will be made repeatedly, so the final decoding logic must remove these repetitions by using the detected silence regions as boundaries between different DTMF signals. For testing, we will initially work with noise-free signals to understand the basic functionality of the decoding system, and then test on some signals that contain noise.

To make the whole system work, you will have to write four M-files: `DTMF_all`, `DTMF_BPF` and `DTMF_energy` and `DTMF_decode`. The main M-file, named `DTMF_all.m`, will call `DTMF_BPF.m`, `filter.m`, `DTMF_energy`, and `DTMF_decode.m`. The following sections discuss how to create, use or complete these functions.

4.1 The Overall Touch-Tone System: `DTMF_all.m`

The Touch-Tone system function, `DTMF_all` runs the entire processing chain. It does the filtering and then calls `DTMF_energy` and `DTMF_decode` to determine the sequence of keys that were pressed. The skeleton of this function in Fig. 5 includes the help comments.

The function `DTMF_all` works as follows: first, it designs the eight bandpass filters that are needed, then it processes the input signal through the eight BPFs. The implementation of the IIR bandpass filters is done with the `filter` function in MATLAB. After bandpass filtering, the average energy is calculated for the signals in each of the eight channels. The M-file `DTMF_energy` makes this calculation every 0.01 seconds by taking the average of the energy for the last 0.01 secs. Finally, `DTMF_all` calls the user-written `DTMF_decode` function to determine the list of decoded keys.

4.1.1 Filter Transient

The IIR BPFs are designed and implemented to filter out individual sinusoidal components. However, the design is based on the frequency response which is a *steady-state* notion. The filter response, however, exhibits a response that can be broken into two types: a *transient response* and a *steady-state response.* When the IIR filter processes a finite-length input signal, the transient occurs at the beginning and the end. For example, suppose that the input $x[n] = \cos(\pi n/2)$ is 100 samples long. If processed by the following IIR filter:

$$H(z) = 0.095\frac{1 - z^{-2}}{1 + 0.81z^{-2}}$$

the output will be infinitely long (strictly speaking), but the output decays after $n = 100$; in fact, $|y[n]| < 0.01$ for $n \geq 145$. Likewise, at the beginning of the output signal, the values increase over the region $0 \leq n < 45$ until $y[n]$ reaches a *steady-state* where it behaves like a cosine, i.e., $y[n] \approx \cos(\pi n/2)$. The regions $[0, 45]$ and $[100, \infty)$ are called the transient regions of the response. When you plot the output signals from the eight channels you should be able to clearly see these transient shapes.

```
function keys = DTMF_all(xx,fs)
%DTMF_ALL   keys = DTMF_all(xx,fs)
%    returns the list of key names found in xx.
%   keys = array of characters, i.e., the decoded key names
%     xx = Touch-Tone waveform
%     fs = sampling frequency
%
dtmf.keys =  ['1','2','3','A';
              '4','5','6','B';
              '7','8','9','C';
              '*','0','#','D'];
dtmf.colTones = ones(4,1)*[1209,1336,1477,1633];
dtmf.rowTones = [697;770;852;941]*ones(1,4);
center_freqs = .... %<==========================FILL IN THE CODE HERE
radPole = .... %<===========================FILL IN THE CODE HERE
for kk=1:8
  [bb(:,kk),aa(:,kk)] = DTMF_BPF( center_freqs(kk), radPole, fs );
end
%  bb = 3x8 MATRIX of numerator coeffs for all the filters; one per column
%  aa = 3x8 MATRIX of denominator coeffs for all the filters; one per column
xx_filtered = ....   %<======== Use loop to filter the signal thru the BPFs
xx_energy = DTMF_energy(xx_filtered,???);  %<== calculate average energy
keys = DTMF_decode(xx_energy,???);    %<== Do the detection
```

Figure 5: Skeleton of `DTMF_all.m`. Complete the `for` loop in this function with more code.

4.2 Narrow Bandpass Filter Design: `DTMF_BPF.m`

The IIR filters used in the filter bank (Fig. 2) will be the second-order filters used in the Warm-up.

```
function  [bb,aa] = DTMF_BPF(fcent, r, fs)
%DTMF_BPF
%    [bb,aa] = DTMF_BPF(fcent, r, fs)
%      returns two vectors containing the numerator and denominator
%      coefficients of H(z) with the numerator scaled so that the
%      maximum magnitude of the frequency response is one.
%  fcent = center frequency (scalar)
%      r = radius of the two complex-conjugate poles
%     fs = sampling frequency (in hertz)
%
%   Each BPF must be scaled so that its frequency response has a
%    peak of 1.0, and stopbands less than 0.1.
```

Figure 6: Skeleton of the `DTMF_BPF.m` function. Complete this function with additional lines of code.

(a) Complete the M-file `DTMF_BPF.m` described in Fig. 6. This function will have to be called eight times to produce all eight bandpass filters needed for the DTMF filter bank system. The filter specs are given in the next part. It might be convenient to store the filters in the columns of two matrices `aa` and `bb` whose sizes are 3×8.

(b) *Filter Specifications:* For each of the eight BPFs, choose the radius (r) so that only one frequency lies within the passband of the BPF and all other DTMF frequencies lie in the stopband. The bandpass filters should have a maximum frequency response value of one in the passband, and be less than 0.1 in the stopband. You should run some experiments to learn how to choose the radius (r).

(c) As a test, generate all eight bandpass filters with the same radius of $r = 0.988$ and $f_s = 7350$. Plot the magnitude of the frequency responses all together on one plot (the range $0 \leq \hat{\omega} \leq \pi$ is sufficient

because $|H(e^{j\hat{\omega}})|$ is symmetric). Use a very dense grid for $\hat{\omega} \in [0, \pi]$, with at least 5000 points along the frequency axis. Indicate the locations of each of the eight DTMF frequencies (697, 770, 852, 941, 1209, 1336, 1477 and 1633 Hz) on this plot to illustrate whether or not the passbands are narrow enough to separate the DTMF frequency components, i.e., convert f to $\hat{\omega}$.
Hint: use the `hold` command and markers to denote the DTMF frequencies, as shown in the pre-lab.

(d) Now try to determine empirically the smallest value of (r) so that all the frequency responses will satisfy the specifications on passband width and stopband rejection given above. Since these specifications are very stringent, the pole radius will have to be quite close to one. One of the BPFs will determine the value of r by just meeting the specs, and then the other BPFs will exceed the specs (on the stopbands).

Use the `zoom on` command to view the frequency response over the frequency domain where the DTMF frequencies lie. Comment on the selectivity of the bandpass filters, i.e., use the frequency response to explain how the filter passes one component while rejecting the others. Explain how each filter's passband is narrow enough to pass only one frequency component while rejecting others that are in the stopband. Determine which filter has its poles closest to the unit circle.

4.3 Average Energy: `DTMF_energy.m`

The second module involves an average energy computation done every 10 millisecs. The average energy is computed over an interval L that corresponds to 10 ms.

$$E_i^{\text{avg}}[n] = \frac{1}{L}\sum_{k=0}^{L-1} y_i^2[n-k]$$

The computation appears to use a running average filter, but only needs to be done for $n = L, 2L, 3L, \ldots$.

(a) Complete the `DTMF_energy` function based on the skeleton given in Fig. 7. The input signal matrix `yy` to the `DTMF_energy` function should contain the output of the 8 BPFs, one signal in each column.

(b) Determine the (integer) value of L that corresponds to 10 ms at $f_s = 7350$. From this, verify that the column length of the output matrix `yy_energy` is $1/L$ times that of the input matrix `yy`.

(c) Make a strip plot of the average energy outputs.

Comment: When debugging your program it might be useful to plot the output from each BPF to see that the output is either large (when the filter is matched to one of the component frequencies), or relatively small when the filter passband and input signal frequency are mismatched.
Hint: use the `strips` function in MATLAB to plot all the BPF outputs on the same graph.

```
function  yy_energy = DTMF_energy(yy, fs)
%DTMF_ENERGY
%  usage:    yy_energy = DTMF_energy(yy, fs)
%  returns average energy in each bandpass filtered channel
%     yy_energy = matrix with average energy from the 8 bandpass filters
%     yy = matrix with all filtered outputs from the 8 bandpass filters
%                one signal per column
%     fs = sampling rate
%
%   The average energy is computed every 10 millisecs by averaging over
%   the past 10 millisecs
```

Figure 7: Skeleton of the `DTMF_energy.m` function. Complete this function with additional lines of code.

4.4 Identify the Keys: **DTMF_decode.m**

The third module is decoding—a process that requires a decision on the presence or absence of the individual tones. In order to make the signal detection an automated process, we need a *detection* function that picks the most likely possibilities. The decode function will need two steps: first, find the two channels with the largest average energies and get the corresponding key; second, eliminate the redundant keys that will occur because the DTMF signals are longer than 10 ms.

(a) Complete the `DTMF_decode` function based on the skeleton given in Fig. 8. The input signal matrix `ee` to the `DTMF_decode` function should contain the average energy from the 8 BPFs, one per column.

```
function  detected_keys = DTMF_decode(ee, fs)
%DTMF_DECODE
%  usage:     detected_keys = DTMF_decode(ee, fs)
%  returns list of keys based on the max amplitude of the filtered output
%     ee = matrix with average energy from the 8 bandpass filters
%     fs = sampling rate
%
%   The signal detection is done by finding the two largest outputs
%   within each segment
```

Figure 8: Skeleton of the `DTMF_decode.m` function. Complete this function with additional lines of code.

(b) Use the following rule for initial detection: within one 10 ms time segment, find the maximum average energy $\max_i\{|E_i^{\text{avg}}[n]|\}$ in two subgroups of the eight BPF outputs. The four lower frequency channels give the row index and four higher ones the column index.

(c) In regions of silence, all 8 channel outputs will be small, so the max function would give a bogus result. Therefore, to distinguish silence from signal, a threshold is needed. One strategy is to compare the maximum values found to the average signal level in the other 6 channels that contain no signal. The maximum values should be at least 5 to 10 times larger, but you will have to experiment with this threshold value. Examine the strip plot of the average energies to gain some insight on setting the threshold.
Note: in a noise-free case, or low noise, most threshold value will work, but in a high noise case, the choice of the threshold would be hard to optimize.

(d) Whenever silence is detected, designate the detected key as `Z`.

(e) After running the detection algorithm over the average energies, you will obtain a string that consists of the detected keys ***every 10 ms.*** For example, you might have the string:

```
111ZZ66Z999ZZAAAAZZZ77777ZZ****ZZ444
```

The key names are repeated because the actual DTMF signals last much longer than 10ms. So the final step in the decoder is to remove these redundancies. The repeated characters need to be reduced to single characters, and then the `'Z'` keys need to be removed because they denote the silence between real keys. For the example above, these two steps would give:

```
1Z6Z9ZAZ7Z*Z4   ⟶   169A7*4
```

The MATLAB function `diff` can be used to find redundant entries in a (string) vector. For example, the locations of all repeats can be extracted with:

```
indicesOfRepeats = find( diff(abs(stringOfKeys))==0 ).
```

4.5 Testing Telephone Numbers

The functions `dtmfdial.m` and `DTMF_all.m` can be used to test the entire Touch-Tone system as shown in Fig. 9. You could use random digits (e.g., `char('0'+9.99*rand(1,9))`), or standard phone numbers as inputs into `dtmfdial`. For the `DTMF_all` function to work correctly, all the M-files must be on

```
>> fs = 7350;  %<--use this sampling rate in all functions
>> tk = ['*','#','0','1','2','3','4','5','6','7','8','9','A','B','C','D'];
>> xx = dtmfdial( tk, fs );
>> soundsc(xx, fs)
>> DTMF_all(xx, fs)
ans =
    *#0123456789ABCD
```

Figure 9: Testing the complete Touch-Tone decoding system.

the MATLAB path. It is also essential to have short intervals of silence lasting 0.1 secs., or more, in between the tone pairs so separate DTMF tone-pairs can be located reliably.

If you are presenting this project in a lab report, demonstrate a working version of your programs by running it on the following "phone number."

`5048941AAA6332BCC***7D#`

In addition, make a spectrogram of the signal from `dtmfdial` to illustrate the presence of the dual tones.

4.6 Testing with Low-Noise

A test signal is available for testing under noisy conditions: A Touch-Tone signal plus wideband additive noise can be found in the variable `sigTT1` in the MAT file `lab10s06data.mat`. Process this signal to determine the phone number dialed. A similar test will be run when your code is evaluated in the next lab. *Note:* it would be interesting to listen to the signal and hear the noise that is interfering with the DTMF signal.

4.7 Demo

When you submit your lab report, you must demonstrate your work to your TA in the lab. Have your code and files ready for the demo. You should call `DTMF_all` for a signal `xx` provided by your TA. The output should be the decoded telephone number. The evaluation criteria are shown at the end of the verification sheet.

Lab #10
ECE-2025
Spring-2006
INSTRUCTOR VERIFICATION PAGE

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 end of your lab period.

Name: ____ Date of Lab: __________

Part 3.1: Complete the dialing function `dtmfdial.m`. Listen to a phone number.

Verified:____________________ Date/Time:__________

Part 3.2(d): Measure the stopband locations of the IIR bandpass filter.

Verified:____________________ Date/Time:__________

Part 3.2(e): Make a plot versus analog frequency. Determine the analog frequency components passed and stopped by the BPF when $f_s = 7350$ Hz. Give the range of stopband frequencies in Hz.

Verified:____________________ Date/Time:__________

Touch-Tone Decoding Evaluation

Does the designed Touch-Tone decoder get the correct telephone numbers for the following cases?

Value for r ____________

Case 1 (No noise): All Numbers _____ Most _____ None _____

Case 2 (Low noise): All Numbers _____ Most _____ None _____