ELE 201, Spring 2014
Laboratory No. 1
Matlab and Signal Analysis

1 General Pointers

Throughout this and subsequent lab handouts, you'll see boxes in the margins that mean the following things:

$\boxed{\text{Q0}}$	A simple, single-right-answer question for you to answer in your write-up
$\boxed{\text{D0}}$	A more open-ended discussion question or series of questions for you to address in your write-up
$\boxed{\text{M0}}$	Matlab signals or functions you need to create and show to your TA during the lab

Your lab grade is affected by:

- The presentation quality of your write-up (to a small degree)
- Answers to the $\boxed{\text{Q}}$ and $\boxed{\text{D}}$ questions in the write-up
- Additional comments and observations you make in the write-up
- The signals and graphs you create and show to your TA during the lab

Each lab assignment is intended to take two weeks. The handouts are divided into two parts, presented in separate sections, for your convenience. However, they are to be handed in together.

2 Introduction to Matlab

If you are unfamiliar with Matlab, these websites might be helpful:

- **http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-094-introduction-to-matlab-january-iap-2010/lecture-notes/**
- **http://web.cecs.pdx.edu/ gerry/MATLAB/intro/intro.html**

Google is also a fast way to find an answer.

2.1 Working directory

Some Matlab files for these labs are provided by us and downloadable from the webpage. You will need to put them in Matlab's working directory or path. The working directory can be changed by clicking the '`...`' button at the top of the Matlab command window.

2.2 Matlab programming

Matlab is an industry standard software package for the analysis and visualization of data. Much of its power lies in its highly optimized operations on vectors and matrices. In many cases, you should be able to eliminate the `for` loops you would need to write in C code with Matlab's simple and fast vectorized syntax. Here's a brief introduction:

- **Help** – Type `help command_name` at the Matlab prompt to get help on a specific command. For a nicer interface to the help utility, click the question mark button at the top of the Matlab command window to get a separate help window. If you don't know the specific name of the command you're looking for, try using the `lookfor` command.

  You can also use the `help` command on functions and scripts that you write yourself. The `help` command will always output the comments under the function declaration or those at the top of the script, so in order to have `help` work, you must always comment your functions and scripts!

- **Variable representation** – All Matlab variables are vectors or matrices (arrays) with double precision complex entries. You can create variables using the = sign, e.g. `v = [1 2 3]`.

- **Operations on signals** – The operators + and − operate how you would expect in Matlab. However, be sure not to confuse the matrix multiplication operator `*` with the element-by-element multiplication operator `.*`. The same applies to the matrix division and exponentiation operators / and ˆ and their element-by-element counterparts `./` and `.^`. That is, if you have two vectors `x = [x(1), ... , x(5)]` and `y = [y(1), ... , y(5)]`, the command `x.*y` produces the vector `[x(1)*y(1), ... , x(5)*y(5)]` and the command `x*y` produces an error since these vectors can't be matrix-multiplied. Note that no `for` loops are required for these element-by-element operations. The transpose operator `'` swaps the dimensions of a matrix, so that a row vector becomes a column vector and vice versa. All the normal elementary functions like `cos` and `log` are in Matlab; see `help elfun` for a list.

- **Other commands** – For vectors, `max(X)` is the largest element in X. For matrices, `max(X)` is a vector containing the maximum element from each column. To get the maximum element of the entire matrix X, try `max(X(:))`. Functions like `min`, `mean`, `median`, and `abs` can be used in a similar manner.

- **Indexing** – To create a vector that ranges from `a` to `b` in steps of 1, use `v = a:b`. To specify a different step size `step`, use `v = a:step:b`. You can also use the colon operator and the `end` keyword to get at the entries of vectors and matrices easily. For example, to get every 5th element of the vector `a`, use `a(1:5:end)`. See `help colon` and `help end`.

- **Elementary matrices** – To help you create signals, you may find the following commands useful:
  - `zeros(m,n)` creates an `m` by `n` matrix of all zeros.
  - `ones(m,n)` creates an `m` by `n` matrix of all ones.
  - `eye(n)` creates an `n` by `n` identity matrix (ones down the diagonal, zeros elsewhere.)
  - `rand(m,n)` creates an `m` by `n` matrix of independent *uniformly distributed* random variables between 0 and 1.
  - `randn(m,n)` creates an `m` by `n` matrix of independent *normally distributed* random variables with mean 0 and variance 1.

  Also, the syntax `c = [a,b]` makes a new matrix `c` which is `a` and `b` put together side-by-side (provided the vertical dimensions match). Using `d = [a;b]` makes a new matrix `d` by putting `a` and `b` together top-to-bottom.

- **Program flow** – The Matlab programming language contains standard programming constructions such as

```
if (condition)          if (condition)          if (condition1)
    commands                commands1               commands1
end                     else                    elseif (condition2)
                            commands2               commands2
                        end                             ⋮
for (variable = expr)                           elseif (conditionN)
    commands            while (condition)           commandsN
end                         commands            end
                        end
```

- **Display** – Use a semicolon `;` at the end of a command to suppress the display of results on the screen. **Ctrl-C** stops the execution of any command. You can use the command `whos` to see the variables in your workspace and their sizes. You can hit the up arrow key to get to previous commands, or type the first few letters of a previous command and hit the up arrow to find the last matching command. For example, typing `p` and pressing the up arrow brings back the last command that started with the letter `p`.

- **Viewing 1-D signals** – Use the command `plot(x)` to plot a vector `x`. To plot a vector `y` against a vector `x` (both must be the same size), use `plot(x,y)`. You can specify the color of the plot, the type of markers, and the line style; see `help plot` for all the details. To plot several signals in the same window, type `hold on` to prevent new plots from replacing the current plot. Use `figure` before any visualizing commands such as `plot` and `imagesc` to bring up additional figures.

- **Viewing 2-D signals** – Use the command `imagesc(M)` to view the matrix M as an image. The (1,1) entry of the matrix will be at the upper left hand corner. You may need to use the command `colormap gray` to make the image more understandable; large values are plotted as white, and small values are plotted as black.

- **Listing to sound signals** – Use the command `sound(x,Fs)` to send the signal `x` with sample frequency `Fs` to the computer sound system. A similar command, `soundsc(x,Fs)`, will first rescale the signal to give maximum volume without clipping. You can also write and read `.wav` files using `wavread` and `wavwrite`.

- **Saving and loading** – Matlab data can be stored into binary `.mat` files on your disk. To save your whole workspace in a file called `filename.mat` use `save filename`.
  To save one particular variable called `variable_name` into `filename.mat` type
  `save filename variable_name`.
  Saving will overwrite whatever filename you specify. To append instead of overwrite, use
  `save filename variable_name -append`.
  Note that the saved files are in a special format, and are unreadable by other applications. You can use `save ... -ascii"` to save your workspace as a text file. See `help save` for more details.
  Load a `.mat` file by typing `load filename`. Again, you can load only specific variables using `load filename variable_name`. See `help load` for more details.

- **Writing Matlab programs** – Procedures that you call repeatedly can be stored either as functions or as macros. You create both of these by writing a `.m` file in a text editor and storing it in your working directory. Macros operate on existing variables in your workspace (i.e., change them, modify them, create new ones, etc.). They do not have to be passed any inputs. You run them by typing `macro_name` while running Matlab. Functions operate only on variables passed to them and they do not change the existing workspace. The headers of their `.m` files have to include a statement like:
  `function [out1, ... outN] = function_name(input1, input2,.., inputN)`
  The final value of variables `out1`, ... , `outN` will be automatically returned, once the function execution is finished. User created functions are called in the same manner as built-in functions.

2.3 Matlab examples

We will be working with samples of continuous-time signals. Let $x(t), t \in [0, T]$, be a continuous-time signal defined over the time interval $[0, T]$. Fix an integer $N > 1$ and set $\Delta = T/N$. Then provided N is sufficiently large the signal x can be adequately represented by the set of N uniformly spaced samples $s(k) = x(t_k)$ where $t_k = k\Delta$, $k = 0, 1, \ldots, N-1$.

The sampling frequency in the above representation is $F_s = 1/\Delta = N/T$. Roughly, Nyquist sampling theorem says that if the sample frequency is more than twice the highest frequency present in the signal, then the samples accurately represent the original signal. (We will do more on this later.)

A sampled signal can be stored as the elements of a $1 \times N$ matrix. For example, the signal $x(t) = \sin(2t), t \in [0, 10]$, can be represented in Matlab as a $1 \times N$ matrix as follows:

```
Ts = 0.1;
N = 100;
t = [0:N-1]*Ts;
x = sin(2*t);
```

Here, `Ts` is the inter-sample time Δ (So $1/Ts$ is the sample frequency); `N` is the total number of samples; `t` is a $1 \times N$ matrix indexed from 1 to `N` containing the sample times 0, Ts, 2Ts, ... (N-1)Ts; `x` is a $1 \times N$ matrix containing the samples of the signal $\sin(2t)$ at the sample times.

2.3.1 Plotting a signal

You can plot the signal with the commands:

```
plot(t,x);
grid
xlabel('time - secs')
ylabel('signal x')
title('Plot of x vs t')
```

You can also use the plot command to plot several signals on the same graph and you can control the colors of the lines as well as their type. For example,

```
y = 2*x;
plot(t,x,'-',t,y,'--');
grid
xlabel('time - secs')
ylabel('signal x')
title('Plot of x and y vs t')
```

will plot the signals x and y on the same graph. The first signal will be plotted with a solid line and the second signal will be plotted in a dashed line. Use the `help` facility to learn all the color and line type controls.

Sometimes you may want to plot more than one signal on the same page but on different graphs. In this case you can use the `subplot` command to set up an array of plots. For example,

```
subplot(2,1,1), plot(t,x,'g');
grid
xlabel('time - secs')
ylabel('signal x')
title('Plots of x and y')

subplot(2,1,2), plot(t,y,'r');
grid
xlabel('time - secs')
ylabel('signal x')
```

will plot the signals x and y on a 2 by 1 grid of separate plots. The first signal will be plotted in a green line and the second in a red line.

You can also plot each group of signals on a completely new page using the `figure` command. For example,

```
figure (1)
plot(t,x);
grid
xlabel('time - secs')
ylabel('signal x')
figure(2)
plot(t,y);
grid
xlabel('time - secs')
ylabel('signal y')
```

will plot the signal x on the first figure window and the signal y on the second figure window.

2.3.2 Writing an m-file

Usually the best way to use Matlab is to first write a Matlab program, called an m-file. You can create scripts and functions this way. Scripts allow you to make changes, fix bugs, etc., in an efficient manner, without having to retype a bunch of commands.

To write an m-file you use any text editor to create a file containing Matlab commands and store the file as ASCII with a `.m` extension. Matlab also has a built in editor for this purpose. The m-files need to be stored in the working directory or path. The m-file is executed by calling the name of the file (without the `.m` extension).

A script is the simplest program you can write in Matlab. It is an automated sequence of Matlab commands. When it is executed, it is as if you typed all of those commands in order. A script might look like this:

```
clear           % clear memory
clf reset       % clear all figures

T = 10;         % signal duration
Fs = 8000;      % sampling frequency
f = 440;        % frequency of sound

t = 0:1/Fs:T;   % time axis
n = T*Fs;       % length of vector

x = 4 * exp(-2*t).*sin(2*pi*f*t);
plot(t(1:n),x(1:n),'r')
grid
xlabel('time-secs')
ylabel('signal value - volts')
title('A Plot of a Simple Signal')

sound(x,Fs)     % play the signal
```

Notice that if Matlab encounters a `%` symbol on any input line it ignores the rest of that line. So the `%` symbol can be used to add comments to your programs.

To run a script file, just type the name of the file into the command line. Values such as `T`, `Fs`, `t`, and `x` will become workspace variables in your environment after running the script.

2.3.3 Functions

A function can have input arguments and output arguments. Matlab functions are very similar to methods in Java and functions in C. The m-file must start with the word `function` and state the inputs and outputs.

A simple function program might look like this:

```
function x = myFunction(Fs, f)

% function x = myFunction(Fs, f)
% Fs is the sampling frequency
% f is the frequency of sound
% x is the sound signal

T = 10;          % signal duration
t = 0:1/Fs:T;    % time axis
n = T*Fs;        % length of vector

x = 4 * exp(-2*t).*sin(2*pi*f*t);
plot(t(1:n),x(1:n),'r')
grid
xlabel('time-secs')
```

```
ylabel('signal value - volts')
title('A Plot of a Simple Signal')

sound(x,Fs)      % play the signal
```

To run this function with `Fs = 8000` and `f = 440`, type the following into the command line:

```
>> output = myFunction(8000, 440);
```

If instead you want to write `myFunction` so that it returns more than one variable, enclose all your output variables with square brackets:

```
function [x, t] = myFunction(Fs, f)
```

To run this in your command line, you can type:

```
>> [output1, output2] = myFunction(8000, 440);
```

If the function has no outputs, the function definition can be written like either or the following:

```
function myFunction(Fs, f)
```

or

```
function [] = myFunction(Fs, f)
```

A few things to keep in mind about functions:

- You must use the keyword `function` in your function declaration.
- The name of the m-file must be the same as the name of the function. (The reason is because when Matlab calls a function, it uses the name of the file rather than the name of the function itself. To avoid confusion, always name your file the same as the function name.)
- Variables defined inside a function are local to the function and will not appear as workspace variables.

3 Lab Procedure—Part 1

This section of the handout needs to be completed and handed in. This is part 1 of a two part lab to be completed in two weeks.

The goals of this lab are the following:

- Familiarize yourself with the Matlab environment
- Learn how to write Matlab programs
- Create and manipulate one- and two-dimensional signals
- Get a feel for the effects of frequency content, sampling rate, and noise on the sound of a signal

3.1 Vector operations vs. looping

First of all, it's important to recognize the crucial difference between using *for loops* and using vector operations. This is key to efficiently using Matlab, since it uses code that is highly optimized for vector operations.

Suppose you have two vectors, `a` and `b`, each with 10,000 elements, that you would like to add. There are two ways in which you can add them.

The first (and better) method is to simply add the vectors:

```
result = a + b;
```

The second (and inferior) method is to use *for loops* to add each element of the vectors:

```
for i = 1:length(a)
    result(i) = a(i) + b(i);
end
```

To illustrate why the first method is better than the second, write a Matlab script called `addCompare` which displays how long the first and second methods take. Initialize vectors `a` and `b` to have length 10,000. These vectors can have any values you choose. To properly compare the timing, you should `clear result` before each experiment because the memory allocation is one of the bottlenecks of the second method.

For timing, use the `tic` and `toc` functions. If you don't know how to use them, ask the mighty `help` function.

After running your script, you should find that the second method is much slower. One of the reasons the second method is slower is because the variable `result` changes size at each iteration of the loop. This means that Matlab has to fetch more memory and copy the values for each iteration, which is very slow. To avoid this, you can initialize the size of `result`. Add the following line before the *for loop*:

```
result = zeros(1, 10000);
```

This line sets the variable `result` to 10,000 zeros. (The `zeros` function takes dimension of a matrix as input, which is why we need to have the 1 there).

Try your script again with this line. You should find that the total time for the second method has decreased significantly. However, even this improved version of the second method is still slower than the first method. How fast are these three methods relative to each other? $\boxed{\text{Q1}}$

Many other operations are like addition and can be sped up using the vector operation instead of the *for loop.* If you want to multiply each element of a vector with its corresponding element in another vector, then you can use

```
result = a .* b;
```

instead of

```
result = zeros(1, length(a));
for i = 1:length(a)
    result(i) = a(i) * b(i);
end
```

The moral of the story is: ***For loops* should be avoided if possible!** They are slow and inefficient. This important lesson will be useful in future labs.

3.2 Simple 1-D signals

To get some practice in creating signals, generate and plot the following variables. All of the variables should be row vectors.

1. `twos`, a vector 50 elements long where every element is 2
2. `ramp1`, a vector that goes from 5 to 55 in steps of 1
3. `ramp2`, a vector that goes from 23 to 18 in steps of -.1
4. `rampprod`, the element-by-element product of `ramp1` and `ramp2` $\boxed{\text{M1}}$
5. `oto4pi`, a vector that goes from 0 to 4π in steps of .01. (Use the built-in variable `pi`)
6. `cos1`, the function $f(t) = a\cos(2\pi\nu t + b)$, where a is 3, ν is .7, b is .3, and t ranges from 0 to 4π. (Use the variable `oto4pi` above) $\boxed{\text{M2}}$

3.3 Pure Tones

The musical notes of the western music scale are grouped into *octaves* with each octave containing 12 notes. The octave containing middle C covers the frequency range from 220Hz to 440Hz. Within each octave the notes are logarithmically spaced so that jumping one octave doubles the frequency. Thus the frequency of each note is $2^{1/12}$ times the frequency of the note below it. The frequencies of the notes in the middle C octave are shown in Table 1.

A	220Hz
As	$2^{1/12}$*A $\approx$ 233Hz
B	$2^{1/12}$*As $\approx$ 247Hz
C	$2^{1/12}$*B $\approx$ 262Hz
Cs	$2^{1/12}$*C $\approx$ 277Hz
D	$2^{1/12}$*Cs $\approx$ 294Hz
Ds	$2^{1/12}$*D $\approx$ 311Hz
E	$2^{1/12}$*Ds $\approx$ 330Hz
F	$2^{1/12}$*E $\approx$ 349Hz
Fs	$2^{1/12}$*F $\approx$ 370Hz
G	$2^{1/12}$*Fs $\approx$ 392Hz
Gs	$2^{1/12}$*G $\approx$ 415Hz

Figure 1: Table 1.

A pure tone of frequency f Hz is the sinusoidal signal $x(t) = A\sin(2\pi f t + \phi)$. Here A is the amplitude of the tone, f is the frequency in Hz, and ϕ is the phase.

1. Write an m-file called `tone.m` to generate a pure tone, play the signal through the sound system using the `sound` command, and plot the signal. Start your program by clearing the workspace and then assign values to the following variables:

   `Fs` - sampling frequency in Hz
   `f` - tone frequency Hz
   `T` - duration of signal
   `t` - vector of sample times
   `Amp` - amplitude of tone
   `ph` - phase of tone
   `x` - the vector of signal values
   `N` - no. of samples to be plotted

   Make sure your plot displays a grid, has the axes correctly labeled, and is given a title.

2. Use `tone.m` to generate a pure tone of 3 seconds duration at middle C. Use a sampling frequency 8 KHz. Let the amplitude be one and the phase be zero, and plot $N = 300$ samples.

3. What do you get if you try to plot to whole signal vs time. Explain. Q2

4. How does changing the value of `Amp` effect the sound of the signal? Try values greater than and less one. D1

5. How does changing the value of `ph` effect the sound of the signal? Q3

6. What happens if you decease the sampling frequency to 1 KHz? Does it drastically alter the quality of the results? How low can you bring the sampling frequency before the sound quality changes dramatically? D2

7. Demonstrate your program to the TA. M3

3.4 Chords

Create the C Major chord by adding three sinusoids together of the notes C, E, and G, with the same amplitudes. Listen to the result using `soundsc` (to avoid clipping). Also, plot this signal. M4

3.5 Signals as sounds

We have created a command `splay` (for **s**ignal **play**er) that is a graphical interface to the sound function, shown in Figure 2, with a few built-in sounds. Make sure the files are in your working directory or path.

The number in the box labeled *Sampling frequency* controls how many samples are played per second. Changing this value while keeping the signal the same is like shifting the frequencies of the signal. Decreasing the sampling frequency makes the sound slower and more low-pitched. For example, you can make Aretha (Respect) sound like a man by changing the sampling frequency to around 34,000. Experiment with the sampling frequency on the other sounds to see its effect. (Don't use values that are too low, or the sounds will take forever to play!)

Which sound and sampling frequency created your favorite effect? Q4

You can import a workspace variable into the signal player by typing its name in the *Workspace variable* box and hitting return.

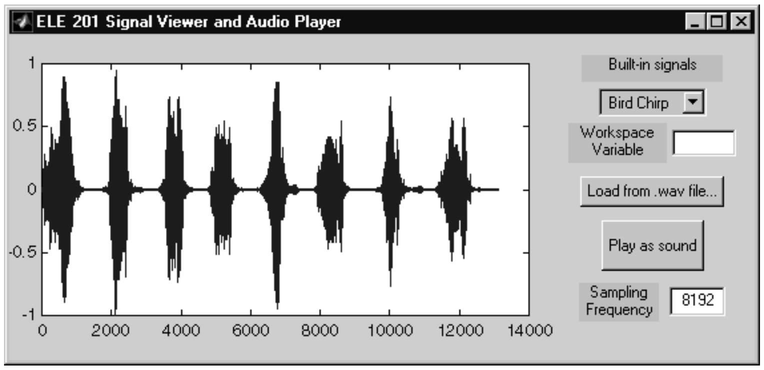

Figure 2: ELE 201 Signal Viewer and Audio Player

3.6 Familiar sounds

Now use your signal-creating savvy to create the following signals, each defined as a mathematical function. Create the signals using a sampling frequency of 10,000, and play them as sounds. You should use amplitude 1/2 and phase shift 0 for the sinusoids.

1. Create a variable d that is the sum of two sinusoids with frequencies 350 Hz and 440 Hz, evaluated over the interval $[0, 4]$. $\boxed{\text{M5}}$
2. Create a variable $b1$ that is the sum of two sinusoids with frequencies 480 Hz and 620 Hz, evaluated over the interval $[0, .5]$. Next, create a vector $z1$ of 5,000 zeros. Finally, create a variable b which is composed of alternating copies of $b1$ and $z1$, four of each. $\boxed{\text{M6}}$
3. Create a variable $r1$ that is the sum of two sinusoids with frequencies 440 Hz and 480 Hz, evaluated over the interval $[0, 2]$. Next, create a vector $z2$ of 40,000 zeros. Finally, create a variable r which is composed of alternating copies of $r1$ and $z2$, three of each. $\boxed{\text{M7}}$

If you created these vectors correctly, `d`, `b`, and `r` should sound familiar. What are these sounds? $\boxed{\text{Q5}}$

3.7 Simple 2-D signals

Create the following variables as matrices of size 64×64, and look at them using the `imagesc` command. Use no `for` loops! You may find the vector outer product to be useful. For example, if `v = [1 2 3 4 5]`, what is `v'*v`?

1. `twosm`, a matrix where every element is 2
2. `rampm1`, a matrix which is constant along the rows and a ramp along each column
3. `rampm2`, a matrix which looks like a ramp in both rows and columns $\boxed{\text{M8}}$
4. `sinm1`, a matrix which looks like a sinusoid with the same frequency in both directions $\boxed{\text{M9}}$
5. `sinm2`, a matrix which looks like a sinusoid with different frequencies in each direction $\boxed{\text{M10}}$

If you find the resulting images hard to visualize, try using the `surf` command instead. By typing `rotate3d` after the figure appears, you can grab the plot and move it around to see it from different angles.

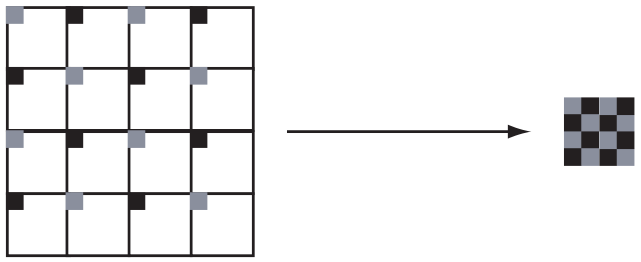

Figure 3: Subsampling the cropped image

3.8 Image manipulation

1. Load the file called `artdeco.mat`, which should add a matrix named `im` into your workspace.
2. View the matrix as an image with `imagesc` and describe what you see. (You may need to additionally type `colormap gray` to make the image grayscale.)
3. Crop the image from rows 128 to 287 and columns 58 to 217 to make a 160×160 matrix called `cropped`
4. Make a smaller 40×40 image by dividing the matrix `cropped` into 4×4 blocks and taking the (1,1) entry of each block. (See Figure 4). Remember: no `for` loops! $\boxed{\text{M11}}$
5. If you did these steps correctly, you should see a familiar object. What is it? $\boxed{\text{Q6}}$

4 Lab Extras

If you have time, here are some extra activities to try. This section is not to be turned in for credit but is highly recommended.

4.1 Harmonics

When a particular note is played on a musical instrument, not only is the fundamental frequency (tone) generated but also higher harmonics of the fundamental, i.e. pure tones at the frequencies $2f, 3f, 4f, \ldots$. These harmonics give the instruments richer and unique sounds.

1. Copy your m-file `tone.m` into a new file `harmon.m` and modify the new file to generate a signal of 1.5 seconds duration that is the sum of the tone middle C and its harmonics up to and including the M'th harmonic. Allow for the possibility that the harmonics have different amplitudes and phases. You can do this by using one vector to represent the amplitudes and another to represent the phases of the component signals.

   Use a sampling frequency of 16 KHz. Under this sampling frequency, what is the largest value of M we can accurately simulate? Select a value for M accordingly.

   Have your program do the following:

   - Plot the pure tone and the sum of the tone and its harmonics on the same graph vs time (again just the first N samples).
   - Play both signals using the sound command.

- Play a signal that consists of the pure tone for 1.5 secs concatenated with the tone plus harmonics for 1.5 secs, concatenated with just the harmonics for 1.5 secs. This will allow you to hear the effects of the harmonics very clearly. Can you tell the pitch with just the harmonics? Try dropping the first few harmonics as well, can you still tell the pitch? Can you suggest why this may be so?

You can experiment with different amplitudes and phases to see the effect on the composite signal appearance and sound.

4.2 Phase error

Let's try an experiment. Create a program `randph.m` that generates and plays two signals. First it generates a signal at the frequency of middle C with ten harmonics. Then it generates a second signal using the same harmonic amplitudes but the phases are random numbers in the range $[-\pi, \pi]$. You can do this with the command `rand`. Have the program plot both signals on the same graph so that you can see the effect of the random phase. The program then randomly selects one of the signals and plays it. It then waits 4 seconds, again randomly selects a signal and plays it. It then asks you to say if the two signals it played were the same or different.

Try it out - give honest answers on 20 successive attempts. In how many trials where you correct? Is the outcome of your experiment statistically significant? To be regarded as statistically significant the standard benchmark is that the probability of obtaining the same result by random guessing should be less that 0.05. Can your draw any conclusion about the tolerance of the human ear to differences in the phase of harmonic components? How could you strengthen the experiment to make a more stringent test?

[hint: binomial distribution. Try `help binomial`]

4.3 Adding noise

Now, we'll add some uniform noise to the signals we created. It's common to make the assumption that noise is centered at 0, and has some maximum amplitude a, but the output from the `rand` command is in the range $[0, 1]$ and is centered at 0.5. Write a function `unoise` which takes as input the number of samples `n` and the noise amplitude `a` and returns a vector of `n` zero-mean uniform noise samples with maximum amplitude a. Add noise of varying amplitudes to the signal r you created in the last section (you can use `length(r)` to find out how long to make the noise signal), and play the noisy signals. What is the result of varying a? When does the noise become too obtrusive? At what point does it not make a difference in sound quality? That is, when does (signal + noise) sound like the original signal?

5 Lab Procedure—Part 2

The goals of this lab are the following:

- Develop intuition about the frequency content of a signal
- Explore the Fourier transform
- Learn how to use the `fft` and `ifft` commands
- Become familiar with the spectrogram

5.1 Signal Viewer and Audio Player

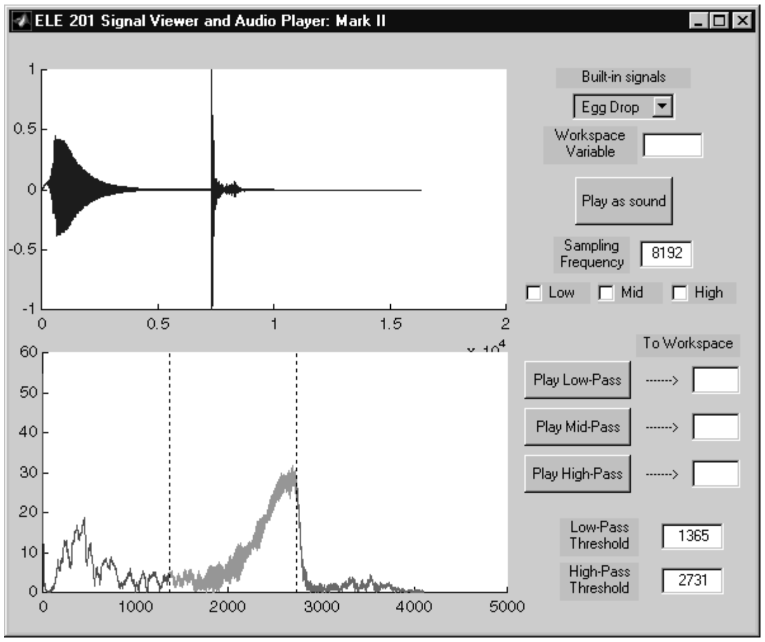

Figure 4: `splay`'s bigger brother

The ELE 201 Signal Viewer and Audio Player has been beefed up as `splay2`.

The upper half is the same as the old `splay`. However, the lower half shows the magnitude of the Discrete Fourier Transform (DFT) of the signal you choose. The vertical dotted lines indicate thresholds in the frequency domain which split the frequency axis into three parts.

By checking the boxes labeled "Low", "Mid", and "High" underneath the *Sampling Frequency* box, you can see the time-domain signals that result from taking the inverse DFT of each segment of the frequency axis. By clicking the buttons labeled "Play Low-Pass", "Play Mid-Pass", and "Play High-Pass", you can hear these signals played as sounds. You can change the frequency-domain thresholds by typing into the appropriate boxes. You can also zoom in and out of the DFT plot by left- and right-clicking, respectively.

5.1.1 DFTs of simple signals

The built-in signals in `splay2` include the dial tone, busy signal, and phone ring sounds from Lab 1, as well as the bird chirp, dropping egg, and train whistle. Load each of these signals and take a look at the frequency-domain representation. For the telephone sounds, adjust the thresholds to isolate the pure tones of the signal, and play them for your TA. The train whistle sound has three major peaks and a couple minor ones. Adjust the thresholds to isolate them and play the corresponding time signals separately. $\boxed{\text{M12}}$

Compare the DFTs of the dial tone and the busy signal. Superficially, the two DFTs look the same: two sharp peaks. But, zoom in on the DFT of each signal. How are these DFTs different? What do you think accounts for the difference? (Hint: How are a busy signal and a dial tone different, besides the change in pitch?) Try thresholds of 479 and 481 for the busy signal and describe what the resulting signals look and sound like. Can you explain what's happening? To view the isolated signal, enter a variable name in the box next to the "Play Mid-Pass" button to create a variable (say, `bt`) in your Matlab workspace. You can take a look at this signal by itself by entering `bt` into the "Workspace Variable" box. $\boxed{\text{D3}}$

5.1.2 Removing Unwanted Tones

You have been provided with a sound file, `buzzjc.wav`, which is an audio clip of someone saying something, but it has been corrupted by adding a buzzer signal consisting of a few tones. First, load the signal into Matlab using `wavread`. Then, load this into `splay2` and figure out the tone frequencies (at least approximately).

Next remove these tones. You can do this using `splay2` by isolating the buzz sounds, exporting them to the workspace using the interface buttons, and then subtracting them from the original signal. After processing, you should be able to understand what the person is saying. Play the denoised sound for the TA. $\boxed{\text{M13}}$

5.1.3 DFTs of more complex signals

Let's take a look at the DFT of a music signal: Load the built-in signal "I Wish 1". Set the low threshold at 500 and the high threshold at 2,000, and play the three components. What can you say about what instruments/sounds fall into each range? $\boxed{\text{D4}}$

Now the fun begins. Load the built-in signal "I Wish 2" and describe what you hear. Take a look at the frequency domain and describe what you see. By moving the high-pass threshold, can you isolate the noise? When you've done it, enter a variable name in the box next to the "Play High-Pass" button to create a variable (say, `fred`) in your Matlab workspace. You can take a look at this signal by itself by entering `fred` into the "Workspace Variable" box. You should see that it only has a high-frequency component.

This sound has a secret message hidden in it. To find the message, use the m-file we provided to shift the frequency information to a lower frequency:

```
>> fredlow = shiftdown(fred,11025,highthresh);
```

where `highthresh` is the high-pass threshold you used to obtain `fred` in the first place, and 11,025 is the sample frequency for the "I Wish 1" sound. Now you can hear the message by listening to the resulting signal. Play the sound to the TA, Also, note the high-frequency threshold you used. $\boxed{\text{M14}}$ $\boxed{\text{Q7}}$

5.2 Using the Matlab `fft` command

Make an m-file to do the following. Generate the signal

$$\cos(2\pi f_1 t) + 2\sin(2\pi f_2 t). \tag{1}$$

To begin with, let f_1 be 400Hz and f_2 be 440Hz. Use a sample rate of $F_s = 8$kHz and $N = 1000$ samples.

Next, compute the DFT using the FFT command. The frequencies associated with the DFT vector are $0, 1/N, 2/N, \ldots$. However, since this discrete-time signal represents samples of a continuous-time signal, we should interpret these frequencies scaled by the sampling frequency. That is, the frequencies in hertz are $0, F_s/N, 2F_s/N, \ldots, (N-1)F_s/N$. Furthermore, in discrete-time, frequencies that differ by one (or by F_s after relabeling) are equivalent due to aliasing. The convention for audio devices is to interpret each frequency component as close to zero in frequency as possible. For this reason we will use `fftshift` on the

DFT output, to swap the first and second half, and label the frequencies from $-F_s/2$ to $F_s/2 - F_s/N$ with increments of F_s/N.

Remember that the values of the DFT are complex even though the signal is real. Plot the magnitude (use `abs`) and phase (use `angle`) of the DFT. Instead of using the `plot` command, use `stem`. Center the frequencies at zero using `fftshift`, as mentioned, and use a frequency vector for your plot that accurately represents the frequencies in terms of hertz. Is this what you expected? (Hint: The code to produce the centered DFT is `fftshift(fft(signal))`.) $\boxed{\text{Q8}}$

Now adjust f_1 to the frequencies 401Hz, 402Hz, and so on to 410Hz. What is happening? What is special about 400Hz and 440Hz? Similarly, try changing the number of points N slightly while keeping f_1 at 400Hz. Show a few examples to the TA. $\boxed{\text{D5}}$ $\boxed{\text{M15}}$

To more fully understand what is happening above, it might be helpful to concatenate the signal upon itself, as in `[x,x]`, and plot it. Recall that the DFT uses sinusoids to match the periodic extension of a signal. This gives you a glimpse of two periods of the signal, and hopefully you will notice an anomaly, as you noticed in the DFT. (Hint: Zoom in to the time where the two copies meet.) If you also look at the DFT of this concatenated signal, it should look very much like the original with zeros inserted between frequencies. See if you and your lab partners can figure out why this is.

5.3 DFTs of Images

Before you start this next part, close all the Matlab figures on the screen by typing `close all`.

The DFT can be extended to two dimensions to operate on images. You can't just use the `fft` command on images; you need to use the 2-D command `fft2`.

It's harder to visualize the image DFTs in two dimensions than it is in one dimension. Try using the `fftim` command that we provided to visualize the 2-D frequency domain. (To see what `fftim` is doing, type `open fftim`. Notice that what is displayed is the *log* of the magnitude.) Red corresponds to a high value of the DFT, and dark blue corresponds to a low or almost zero value. The middle of the DFT image corresponds to low frequencies (slowly varying components), and the corners correspond to high frequencies (quickly varying components). This is because we used `fftshift`. You can zoom in and out of the DFT image by left-clicking and right-clicking (or shift-clicking), respectively.

The DFTs of synthetic images like the ones you created in Lab 1 have very sharp peaks, and lots of very low values. For example, consider the double sinusoid image given by

```
t = [0:63]/64;
im = sin(2*pi*2*t)'*sin(2*pi*3*t);
```

First take a look at the image with `imagesc`, then take the 2-D DFT using `fftim`. Describe what you see and why. (Hint: Zoom in and note the locations of the peaks.) Also, show this to the TA. $\boxed{\text{D6}}$ $\boxed{\text{M16}}$

The DFT of a real image can be much different. For example, take a look at the image of the Chrysler building from Lab 1's `artdeco.mat`, and describe what you see in the time and frequency domains. $\boxed{\text{D7}}$

5.4 DTMF

On a touch tone phone, there is a frequency associated with each column of the keypad and there is a frequency associated with each row of the keypad. When a button is pressed, the signal transmitted over the telephone line is the sum of two sinusoids, one with the column frequency and one with the row frequency. This system is called a Dual Tone Multi-Frequency (DTMF) system. The frequencies for the columns are 1,209, 1,306, and 1,477 Hz from left to right, with 1,633 Hz associated with a fourth column that is usually not there. The row frequencies are 697, 770, 852, and 941 Hz, from top to bottom. The standard requires that each 'tone' must be at least 40ms long and that the space between tones must be 40ms. The encoding is shown in Figure 5.

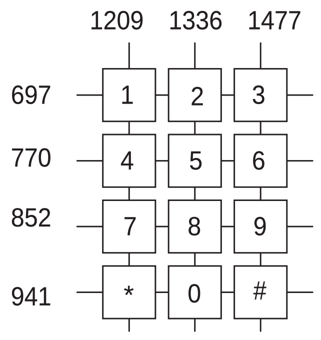

Figure 5: Telephone keypad and the row and column frequencies.

In this lab you will decode a DTMF signal. To find which buttons were pressed and the order in which they were pressed, the spectrogram will be a useful representation. (In practice we would not need to compute the whole spectrogram, only specific values relevant to the DTMF system).

5.4.1 Telephone dialing signal

You have been provided with a .wav file, tel.wav, with a telephone number encoded using the DTMF scheme. Your objective is to decode the phone number.

Plot the magnitude and phase of the DFT of the signal versus frequency in Hz. Include vertical dotted lines at each of the DTMF column and row frequencies on the plot. How much information can you determine from the plots? It must be possible to decode the number from the signal and hence from its DFT (why?). Can you visually decode the number dialed from this plot of the DFT? D8

5.4.2 Spectrogram of telephone dialing signal

The spectrogram is a visualization of the short-time Fourier transform, which is created by dividing a signal into multiple shorter segments, sometimes with overlap, and computing the DFT of each segment. The spectrogram is a visualization of this transform. The DFT of each segment is inserted as a row of a matrix, and the log-magnitude of this matrix is plotted as an image. This allows for visualization of both time and frequency in one plot.

Matlab has a built-in command `spectrogram`. You can simply pass it the name of your signal as a single argument, and it will produce a visualization. But it won't have the axes scaled properly unless you also pass it the sample rate, which is the fifth optional function input (to omit inputs to a function, use `[]`). Try something like the following, with `tel` as the signal and `fs` as the sample rate:

```
>> spectrogram(tel,[],[],[],fs);
```

Notice that the time axis is too course. To change this, you can specify the segment length as the second input. Try a length of 500. This should give better time resolution. Show this to the TA. M17

Can you use the spectrogram to decode the number dialed? Include it in the lab report Q9

6 Lab Extras

If you have time, try loading in some music or speech and looking at the spectrogram. Experiment with different window (segment) lengths and overlap. You can use sound clips we've already been using in the lab or new ones. This section is not to be turned in for credit but is highly recommended.