Signal Computing:

Digital Signals in the Software Domain

Laboratory Manual: Matlab Edition

Spring 2020

Michael Stiber
Bilin Zhang Stiber
University of Washington Bothell
18115 Campus Way NE
Bothell, Washington 98011

Eric C. Larson
Southern Methodist University
Lyle School of Engineering
3145 Dyer Street
Dallas, TX 75205

Contents

0 The Matlab Lab, or How to Not Teach a Programming Language

Labs in this class will make liberal use of the Matlab numerical programming environment. Because this class assumes that you are an experienced computer science student, you are expected to be able to learn how to use computer tools, and how to program in new programming languages, pretty much on your own. So, the first thing you should do is check out the Matlab documentation built into the Matlab help system, or online at http://www.mathworks.com/help/matlab/index.html. Of course, you can always search online for Matlab tutorials and the like. We'll include a very brief overview of Matlab below, and then more detailed information about the code developed specifically for this class that you will be using.

0.1 Matlab in a (Very Small) Nutshell

The Matlab GUI environment is very similar to IDEs that you are already familiar with. As you might expect, there are some idiosyncrasies here and there, but nothing terribly unexpected. The major areas of difference are tools and panes that have to do with viewing variables (something in other IDEs that you'd only see when debugging a program), the command pane, and figure windows that open to display graphs. The first two of these differences have to do with the fact that Matlab is an interpreted language/programming environment. You primary area of interaction with the Matlab interpreter will be through the command window, with variable information panes providing views of variables and their contents related to your interaction. It's interesting to note that, if you want, you can run Matlab without the GUI, providing just a command line interface.

Interpreted languages have their advantages and disadvantages. One advantage is that anything you can use as a line of code in a program you can use immediately as a command on the command line. This lets you test code interactively and then copy it into the script or function you're writing. Like any IDE, Matlab includes an editor with syntax highlighting and debugger integration.

The disadvantage is the interpreted programs are slower. In Matlab, we get around this by using built-in functions that operate on entire vectors or arrays as single data objects. The core loops of the built-in functions are compiled for speed. If you make good use of those functions, Matlab code can often be as fast as completely compiled code.

Here are some things to try:

1. Immediate calculations and variables:

```
radius = 5    % Comments start with "%"
circumference = 2 * pi * radius
area = pi * radius^2
```

2. Complex numbers:

```
sqrt(−1)
x = 7 + 14j
conj(x)       % complex conjugate
abs(x)        % magnitude (e.g., for polar representation)
angle(x)      % and angle for polar rep
real(x)
imag(x)
```

3. Complex exponentials:

```
exp(j ∗ pi)
exp(j ∗ pi/2)
exp(j ∗ pi/4)
```

4. Vectors:

```
v1 = [0 1 2 3]    % Four elements
v2 = [0 : 2 : 10] % like a loop (start value : increment : end value)
v3 = pi ∗ [−0.5 −0.25 0 0.25 0.5] % All operations are vectorized
exp(j ∗ v3)
mistake = v1 ∗ v1     % a mistake; vector mult doesn't work this way
dotproduct = v1 ∗ v1' % transposing will work, if you want to do this
arrayprod = v1 .∗ v1  % element−by−element ops include: .+, .−, .∗, ./
```

5. Simple plots (note that ";" suppresses outputting results to the command window — useful if that would generate massive amounts of text, or just if you want things neat):

```
t = [0 : 0.01 : 2∗pi];
x = sin(t);
plot(t, x);       % default plots points connected by lines in blue
plot(t, x, 'r');  % change the line color
plot(t, x, 'r.'); % change the plot style; zoom in to see individual points
xlabel('t, ms');  % X axis label (all of your plots should have this)
ylabel('Mag');    % Y axis label (all of your plots should have this)
title('Triangle');% Graph title (all of your plots should have this)
```

If you take a sequence of commands and save them in a file with an extension of `.m`, the result is a *script*. Assuming that the script is saved in a directory in the MATLAB search path, you can then execute the script by just typing its name (without the `.m`), just as if it were a command. If you want your code to take parameters, return a return value, and have local variables, start your code with a line like:

```
function retval = funcname(parm1, parm2)
```

Your code is now a function. MATLAB functions can take variable numbers of arguments and even return variable numbers of return values, but that's getting beyond what we need right now.

Step 1.1 It's easy to create, concatenate, extract, and modify vectors or parts of vectors. Execute the following lines of Matlab code and explain what each echoes out:

```
a = ones(1,3)
b = zeros(1,5)
x = [b, a, [1:2:12]]
x(7:end)
length(x)
x(1:2:12)
```

Also, explain the difference between the square bracket notation `[1:2:12]` and the parenthetical notation `(1:2:12)`.

Step 1.2 Consider the result of the following assignment:

```
x(7:11) = pi*(1:5)
```

Write a *single* statement that will replace the odd-indexed elements of `x` with the constant -10 (i.e., `x(1)`, `x(3)`, etc).

Step 1.3 One of the side benefits of learning Matlab is that it trains you to think in terms of parallel operations — an increasingly important skill in a profession becoming dominated by multi-core, GPU, and distributed computing. That doesn't mean you can't write loops in Matlab; it's just that your code will be more concise and efficient if you can avoid that. The efficiency arises from the fact that the vectorized Matlab commands are mostly compiled; while the loops you write are interpreted. Consider the following loop:

```
for k=0:7,
   x(k+1) = cos(k*pi/4);
end
x
```

Why is `x` indexed by `k+1` rather than `k`? What happens to the length of `x` for each iteration of the loop? Rewrite this computation without using the loop (as in list item 5). Besides the increase in efficiency from avoiding an interpreted loop, what other major efficiency results from this change?

Step 1.4 Consider the following code that plots a sinusoid:

```
t = [0 : 0.01 : 1]; % time in seconds
f = 5;              % freq in Hertz
x = sin(2*pi*f*t);
plot(t, x);
xlabel('Time (sec)');
```

Use the MATLAB editor to create a script file called `firstsin.m`, verify that you've saved it in a directory in the MATLAB path (or add that directory to the path), and test its execution by typing `firstsin` at the MATLAB command prompt. Note that you can also do:

```
type firstsin   % prints out contents of the script
which firstsin  % shows directory (useful when your code shadows built—ins)
```

If you included documentation for this script (comments at the beginning), the command `help firstsin` would also produce useful output.

Add three lines of code to your script, so that it will plot a cosine using the same axes as the sine (i.e., "on top of the sine"). Use the `hold` function to add a plot of

```
0.75*cos(2*pi*f*t)
```

to the plot. So, your final graph will have two functions plotted. Save the plot using the MATLAB `print` command as a PNG file named `step14.png` by typing:

```
print —dpng step14
```

You should include all plots and code snippets in your lab report, following the instructions in the report rubric.

Step 1.5 You can also use Matlab to generate sounds. A pure tone is merely a sinusoid, which you already know how to generate. Let's generate one with a frequency of 3 kHz and a duration of 1 second:

```
T = 1.0;
f = 3000;
fs = 8000;
t = [0 : (1/fs) : T];
x = sin(2*pi*f*t);
soundsc(x, fs)
```

The vector of numbers `x` are converted into a sound waveform at a certain rate, `fs`, called the *sampling rate* (we will learn a lot more about this in this class). In this case, the sampling rate was set to 8000 samples/second. What is the length of the vector `x`?

Step 1.6 Write a new function that performs the same task as the following function without using any loops. Use the idea in step 1.3 and also consult the section on the `find` function, relational operators, and vector logicals in the MATLAB documentation.

```
function B = denegify(A)
% DENEGIFY Replace negative elements of matrix with zeros
% Usage:
%    B = denegify(A)
%
[W,H] = size(A);
for i=1:W
   for j=1:H
      if A(i,j) < 0
         B(i,j) = 0;
      else
         B(i,j) = A(i,j);
      end
   end
end
```

0.2 Trigonometric Functions and Complex Mathematics in Matlab

Step 2.1 In this step, you are asked to complete a Matlab function to synthesize a waveform in the form of:

$$x(t) = \sum_{k=1}^{N} a_k \cos(2\pi f t + \phi_k)$$

This is a sum of cosines, all at the same frequency but with different phases and amplitudes. Use the following function prototype to start you off:

```
function x = sumcos(f, phi, a, fs, dur)
% SUMCOS Synthesize a sum of cosine waves
% Usage:
%    x = sumcos(f, phi, a, fs, dur)
%        Returns sum of cosines at a single frequency f, sampling
%        rate fs, and duration dur, each with a phase phi and
%        amplitude a.
%    f = frequency (scalar)
%    phi = vector of phases
%    a = vector of amplitudes
%    fs = the sampling rate in Hz (scalar)
%    dur = total time duration of signal (scalar)
```

Include your code in your writeup. Additionally, include a plot of `x = sumcos(20, [0 pi/4 pi/2 3*pi/2], [1 2 3 4], 200, 0.25);` versus time.

Hint: the MATLAB `length` function is useful in determining the number of elements in a vector; the `size` function returns both dimensions of a vector or an array.

Step 2.2 Now, let's see how complex exponentials can simplify things. Re-implement your `sumcos` function using complex exponentials. Take advantage of the fact that multiplying a complex sinusoid $e^{j2\pi ft}$ by the complex amplitude $a_i e^{j\phi}$ will shift its phase and change its amplitude. Thus, you should be able to create a *single* complex sinusoid at the given frequency `f` and then multiply it by different `a * exp(j * phi)` to get multiple phase shifted cosines. Remember that we want a real value to plot; the cosine is the real part of a complex sinusoid. Include your code in your writeup and provide a plot that demonstrates that this function produces the same result as the original implementation.

Step 2.3 Generate four sinusoids with the following amplitudes and phases:

$$x_1(t) = 6\cos(2\pi(10)t - 0.5\pi) \tag{1}$$
$$x_2(t) = 3\cos(2\pi(10)t + 0.25\pi) \tag{2}$$
$$x_3(t) = 2\cos(2\pi(10)t - 0.3\pi) \tag{3}$$
$$x_4(t) = 8\cos(2\pi(10)t + 0.9\pi) \tag{4}$$

a. Make a single plot of all four signals together over a range of t that will generate approximately 3 cycles. Make sure the plot includes negative time so that the phase at $t = 0$ can be measured. In order to get a smooth plot make sure that your have at least 20 samples per period of the wave. Include your plot in your writeup.

b. Verify that the phase of all four signals is correct at $t = 0$, and also verify that each one has the correct maximum amplitude. Use `subplot(3,2,i)` to make a six-panel subplot that puts all of these plots in the same figure, with space for two additional plots at the bottom. Use the `xlabel`, `ylabel`, and `title` functions so that the reader can figure out what the plots mean; reinforce this with your report's figure caption. (You should include the final figure, with all subplots, that results from finishing all of the parts of this step.)

c. Create the sum sinusoid, $x_5(t) = x_1(t) + x_2(t) + x_3(t) + x_4(t)$. Plot $x_5(t)$ over the same range of time as used in the last plot. Include this as the lower left panel in the plot by using `subplot(3,2,5)`.

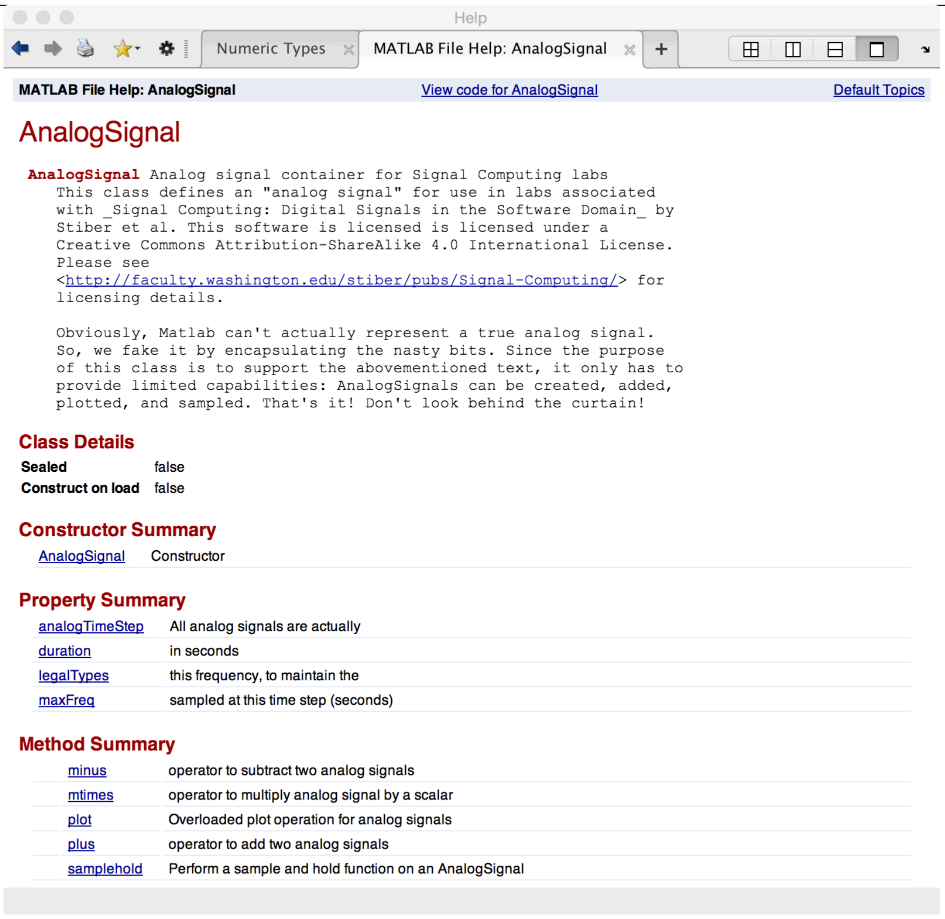
d. Now do some complex arithmetic; create the complex amplitudes corresponding to the sinusoids $x_i(t)$: $z_i = A_i e^{j\phi_i}$, $i = 1, 2, 3, 4, 5$. Include a table in your report of the z_i in polar and rectangular form, showing A_i, ϕ_i, $\text{Re}\{z_i\}$, and $\text{Im}\{z_i\}$.

0.3 Representing Analog, Discrete, and Digital Signals

In our class, we will need to manipulate analog signals (real-valued signals that are functions of continuous time), discrete signals (real-valued signals that are functions of discrete time), and digital signals (discrete-valued signals that are functions of discrete time). No need to worry about the details or these; that will become clear later. The trickiest part of this is representing anything other than digital signals on a digital computer, because you can't. So, we'll need to employ two key elements of software design: information hiding and make-believe.

Information hiding is used in our implementation of analog signals. We make use of the object-oriented programming aspects of Matlab to create an `AnalogSignal` class. If you look up the documentation for `AnalogSignal` (using `doc AnalogSignal`), you'll see something like figure 0.1. The key operations on these analog signals are:

- Creating an analog signal (ex: `a = AnalogSignal('sawtooth', 2.0, 1.0, 10.0)`).


Help
Numeric Types
MATLAB File Help: AnalogSignal
MATLAB File Help: AnalogSignal
View code for AnalogSignal
Default Topics

AnalogSignal

```
AnalogSignal Analog signal container for Signal Computing labs
   This class defines an "analog signal" for use in labs associated
   with _Signal Computing: Digital Signals in the Software Domain_ by
   Stiber et al. This software is licensed is licensed under a
   Creative Commons Attribution-ShareAlike 4.0 International License.
   Please see
   <http://faculty.washington.edu/stiber/pubs/Signal-Computing/> for
   licensing details.

   Obviously, Matlab can't actually represent a true analog signal.
   So, we fake it by encapsulating the nasty bits. Since the purpose
   of this class is to support the abovementioned text, it only has to
   provide limited capabilities: AnalogSignals can be created, added,
   plotted, and sampled. That's it! Don't look behind the curtain!
```

Class Details

Sealed	false
Construct on load	false

Constructor Summary

AnalogSignal	Constructor

Property Summary

analogTimeStep	All analog signals are actually
duration	in seconds
legalTypes	this frequency, to maintain the
maxFreq	sampled at this time step (seconds)

Method Summary

minus	operator to subtract two analog signals
mtimes	operator to multiply analog signal by a scalar
plot	Overloaded plot operation for analog signals
plus	operator to add two analog signals
samplehold	Perform a sample and hold function on an AnalogSignal

Figure 0.1: Example help screen for the `AnalogSignal` class. This example may be out-of-date; use the Matlab `doc AnalogSignal` command to get current documentation.

- Scaling an analog signal (ex: `b = a * 5`)
- Adding two analog signals (ex: `c = a + b`)
- Subtracting two analog signals (ex: `d = c - a`)
- Plotting an analog signal (ex: `plot(d)`)
- Sampling an analog signal (ex: `x = d.samplehold(0.1)`)

You will get a lot more experience working with analog signals shortly in this class, so for the time being just play with this a bit.

1 Let's Get Physical

In this lab, you will use Matlab to explore the effects of summing sinusoids. You will then investigate how a physical signal can be considered to be composed of a sum of sinusoids — its *Fourier series*. You will be using the Matlab `AnalogSignal` class and functions at http://faculty.washington.edu/stiber/pubs/Signal-Computing/.

1.1 Beating

In this section, you will use the Matlab `AnalogSignal` class (described previously in lab 1 and its figure 0.1) to simulate an analog signal generator. The constructor takes the following arguments:

```
% AnalogSignal(type, amplitude, frequency, dur)
%    Where:
%    type = 'sine', 'cosine', 'square', 'sawtooth', or 'triangle'
%    amplitude = signal amplitude
%    frequency = signal frequency
%    dur = signal duration
```

Remember that, though an `AnalogSignal` is simulating an analog signal, in Matlab all functions are sampled at discrete points in time. The crufty details of this are hidden away in the class's implementation. Remember also that you can always plot an `AnalogSignal` to see what you have; include such figures in your report

Step 1.1 Verify that you can get a triangle wave. What is the code to generate and plot a triangle wave ranging from -1 to 1 Volts with a frequency of 10Hz and a duration of 1 second?

Step 1.2 Next, verify that you can generate a sine wave of 1 second duration at a frequency of 440Hz, ranging from -1 to +1. What is the Matlab code to do this? Use the `AnalogSignal` `soundsc` method to play this as a sound. This pitch corresponds to "standard A" on a musical scale — A above middle C. Use the Matlab `set(gca, 'XLim', [`*`Xmin, Xmax`*`])` function call to set the X axis limits so that the waveform is apparent (i.e., you're not just plotting a solid blob).

Step 1.3 Generate three sine waves of identical range and duration, but with frequencies of 442, 444, and 448 Hz. Next, generate the three sums of 440Hz and each of these new signals separately (so, 440Hz + 442Hz, 440Hz + 444Hz, and 440Hz + 448Hz) to generate beating akin to tuning an instrument against the 440Hz standard. Play each sum signal; can you hear the beating? Plot each sum signal for its full 1s duration. The beating "envelope" should be obvious. What is the beat frequency in each case? How does the beat frequency and amplitude relate to the textbook discussion of beating?

1.2 Fourier series representation of a physical signal

Step 2.1 Recall that any periodic signal can be represented as a sum of harmonic sinusoids. The amplitudes of these harmonics is collectively known as the Fourier Series. It may at first seem like sums of sinusoids would be poor approximations of real periodic signals, but this is not the case. We can illustrate this using a triangle wave. The formula for synthesis of a triangle wave with frequency ω_0 is a sum of harmonically related sine waves (its Fourier series):

$$x(t) = \sum_{k=0}^{\infty} \left(\underbrace{\frac{8}{\pi^2} \frac{(-1)^k}{(2k+1)^2}}_{\text{amplitude}} \underbrace{\sin((2k+1)\omega_0 t)}_{(2k+1)^{th} \text{ harmonic}} \right)$$

In this case, in the analog domain, we are dealing with frequencies in Hz, and so $\omega_0 = 2\pi f_0$. Notice that the Fourier Series of the triangle wave only uses odd harmonics (i.e., the only non-zero frequencies are $(2k+1)\omega_0 = \omega_0, 3\omega_0, 5\omega_0 \cdots$). Also notice that the resulting wave will have zero mean because there is no "DC" term (i.e., $2k+1 \neq 0$ for any integer k).

Write a Matlab script that approximates a triangle wave by summing together the first 7 harmonics of its Fourier series; plot the resultant signal (i.e., use f_0, $2f_0$, $3f_0$, $\cdots 7f_0$, where f_0 =10 Hz). How does this signal compare to the triangle wave computed directly in the previous step?

Step 2.2 Another way to view a signal is in the *frequency domain.* For a signal expressed in terms of its Fourier series, the frequency representation is merely the coefficients of the harmonics. Write a Matlab function or script to compute and plot the spectrum of a triangle wave. You may find the MATLAB function `stem` useful. Note that you are *not* being asked to plot the triangle wave as a function of time; you should plot the amplitudes of the component sinusoids as a function of those sinusoids' frequencies (like the vertical lines in textbook figure 1.12). Use your code to plot the spectrum of the triangle wave from the previous step (first 7 harmonics).

2 Hello, Digital!

In this lab, you will investigate how capturing an analog signal for computer use — sampling and quantization — modifies the signal, seeing how the choices you make in the parameters for sampling and quantization affect the quality of the digitized, computer signal.

2.1 Sampling

The first step in digitization is *sample and hold*, in which the continuous analog signal is converted to a discrete-time analog signal (an analog signal that only changes its value at particular points in time). You will use the `samplehold` method to do this:

```
% samplehold Perform a sample and hold function on an AnalogSignal
% Usage:
%  x = obj.samplehold(h)
% where  obj = AnalogSignal
%          h = hold time in sec (sampling interval)
%          x = resultant sampled AnalogSignal
```

Step 1.1 Create an analog sine waveform ranging from -5 to 5V with a frequency of 200Hz and a duration of 2 seconds. Produce a plot with X-axis limits set to make the waveform visible (i.e., don't just make a 2s plot that tries (and fails) to show 400 cycles of the sinusoid.

Step 1.2 Use the `samplehold` method to produce sampled versions of this signal at 300Hz, 500Hz, 1000Hz, and 2000Hz. Use the Matlab `subplot` command, and the `discreteplot` function provided with this class's Matlab code, to plot the original and all four sampled signals together. One of the things that you will note from these plots is that the X axes of these plots do not have units of continuous time; their units are *not* seconds. Instead, the X axis values are sample count, starting with sample 1. You will want these plots to show their signals over the same time duration, and so you'll need to choose that duration, then figure out, for each sampled signal, how many samples correspond to that duration, and then set the X axis limits for each figure so that the plots show equal durations.

Clearly, the results are not the same, and none look identical to the original sine wave. What are the two essential pieces of information about a sine wave that need to be preserved when sampling it? Does it appear that all sampled versions are equally useful in achieving this? Why or why not (in other words, your answer to this question should not be just "yes" or "no")?

Step 1.3 Let's look at aliasing in a little more detail and with a lot more numerical precision. You'll recall from the text that, once we sample a signal, we have limited the range of frequencies that we can represent in our discrete signal to the range $0 \leq \hat{\omega} \leq \pi$, corresponding to a range of apparent frequencies in the physical world of $0 \leq \omega' \leq \omega_s/2$ (or $0 \leq f' \leq f_s/2$). Any frequency in the original signal above $f_s/2$ will be *aliased* into the range of possible apparent frequencies. To keep things simple, we'll stick with a sinusoid; this time, make it 10Hz with an amplitude of -1 to +1 and a duration of 1s. This will make it easy to count cycles when plotted. Set up a figure that can hold three plots and plot this analog signal in the top plot.

Step 1.4 Sample this signal at 25Hz and use the `discreteplot` function to plot the sampled signal in the middle. Sample it at 15Hz and similarly plot that sampled signal at the bottom.

Step 1.5 Before examining the plots in detail, answer the following questions: For each of the two sampling frequencies, what is the range of apparent frequencies that can be represented? For each, will a sinusoid with $f = 10$Hz be aliased? If so, what will be the digital frequency and the apparent frequency of a 10Hz sinusoid?

Step 1.6 Examine the plots and count the number of up-and-down cycles in each. Don't worry that each cycle doesn't look the same, or that every other cycle seems different; just count each. You should see 10 cycles in the top graph; how many do you see in the middle and bottom? How does this compare to the theory you discussed in the previous step? If there is any discrepancy, explain it.

2.2 Analog to Digital Conversion

The last step of digitization is called "analog to digital conversion," or *quantization.* In this step, the sampled analog signal is converted to a discrete signal, with values represented by b bit integers. We will use the `quant` function provided with this class's Matlab code to perform this conversion:

```
% QUANT Quantize a sampled (discrete) signal using a prescribed
%       number of bits per point.
% Usage:
%   y = quant(x,nb,out)
% where y = digital signal quantized to 2^(nb) bits resolution
%       x = vertical points of sampled signal
%       nb = number of bits to use per point
%      out = 'raw' means output binary values: 0,...,2^(nb-1)
%      otherwise, set ouput value range = input value range
```

Step 2.1 Write a Matlab function that compares two signals by computing the *signal to noise ratio* (SNR) that results from changing one into the other (by quantization). Your function should do this by first computing *root mean squared* (RMS) error between the two. In this case, you will be comparing a sampled signal with a quantized version of it. To do this, you should subtract the quantized signal from the sampled signal to produce a vector of differences (errors), then square each difference value using the MATLAB `.^` operator (to get squared errors), then take the mean of these squared values using the MATLAB `mean` function (mean squared error — a scalar value), and finally take the square root of that scalar result (root mean squared error). We can compute the RMS signal in a similar way, by squaring the signal values, getting the mean of those squared values, and then taking the square root. The final result should be a single value — the signal-to-noise ratio (SNR) in *decibels*. What is in the numerator and what is in the denominator of this ratio? Note that this is different than what we did in the textbook, because we are now doing the computation for a *specific* signal, not just figuring SNR for a possible *range* of signal values.

Step 2.2 Use your code to compute the SNR for a quantized sinusoid. Generate an analog signal with with a range of 0 to 5, frequency of 10Hz, and duration 2sec. Sample it at 25Hz. Use 2, 4, 8, 12, and 16 bits quantization, and plot SNR on the Y-axis versus number of quantization bits on the X-axis.

Step 2.3 Repeat Step 2.2 using a square waveform with the same parameters.

Step 2.4 Repeat Step 2.2 using a triangle waveform with the same parameters.

Step 2.5 As you double the number of bits used in quantization, how does the SNR change? How does this compare to what your learned from the textbook? Refer to specific features of your plots from Steps 2.2–2.4 to justify your answer.

3 Feed It Forward

This lab covers the basic concepts of filtering and feedforward filters. You may have also heard of feedforward filters referred to as finite impulse response (FIR) filters. In this lab, we will cover the basic idea of a filter, its mathematical representation (such as the defining equation, frequency response, and transfer function), the relationship among filter coefficients, zero placement, and filter type (low pass, high pass, band reject), and some basic properties of filters.

3.1 Overview of Filtering and Matlab

A *digital filter* is a signal processing operation that can be described equivalently by its *defining equation*, *transfer function*, or *frequency response*. Each representation completely defines the filter. It is advantageous to use each of the different representations depending on whether you are *implementing*, *analyzing*, or *designing* an FIR filter:

$$y[n] = \underbrace{\sum_{k=0}^{M} b_k x[n-k]}_{\text{defining equation}}$$

$$\begin{aligned} Y &= H(z)X \\ H(z) &= \underbrace{\sum_{k=0}^{M} b_k z^{-k}}_{\text{transfer function}} \end{aligned}$$

$$\begin{aligned} Y &= \mathcal{H}(\hat{\omega})X \\ H(e^{j\hat{\omega}}) = \mathcal{H}(\hat{\omega}) &= \underbrace{\sum_{k=0}^{M} b_k e^{-j\hat{\omega}k}}_{\text{frequency response}} \end{aligned}$$

In these equations, $x[n]$ and $y[n]$ are the n^{th} samples from the input and output, respectively, while X and Y represent the entire input and output signal (all of the samples in the signal). A k-sample time delay of a signal is produced by multiplication by the delay operator, $z^{-1} = e^{-j\hat{\omega}k}$. In all three cases (but most simply for the transfer function), we can obtain insight into the filter's operation from the *coefficients*, b_k. We can do this by factoring the transfer function polynomial: its roots are the *zeros* of the filter and they can be real or complex. The placement of the zeros in the complex plane (most usefully expressed in polar coordinates) will tell us which frequencies are suppressed and to what extent those frequencies are suppressed (we can calculate each using the angle and magnitude of the zero, respectively).

Matlab has a modest set of functions related to filtering (a much more substantial set of tools comes along with the Matlab Signal Processing Toolbox, but we will confine ourselves here to using core Matlab). The `filter` function applies a general digital filter to a signal. For this lab, we will stick using this filter as follows:

```
a = [1];              % This will be relevant later for feedback filters
b = [b0 b1 b2];       % Coefficients (remember, Matlab indices start at 1)
y = filter(b, a, x);  % x is input signal; y is output
```

This allows us to define the b_k coefficients for a filter and provide them to the filter function as a vector so that it can filter the input signal.

We'd also like to specify a feedforward filter by indicating its zero locations, which we can do by using the `poly` function to compute the coefficients from a set of roots. So, for example, if we want a feedforward filter with zeros at $0.9+0j$ and $0.75e^{\pm j\pi/4}$, we can compute the b_k values as:

```
r = [ (0.9 + 0.0*j) (0.75*exp(j*pi/4)) (0.75*exp(-j*pi/4))];
b = poly(r);
a = [1];
y = filter(b, a, x);
```

(Where the definition of `r` is written a bit more verbosely than it has to be.) Note that, from the documentation for `poly`, the vector of coefficients it produces are ordered from highest to lowest powers; this corresponds to the same order as the coefficients in the `b` vector, after dividing by z^{-M}, the highest delay term.

You can visualize the zero locations with the following code:

```
plot(complex(r), 'o')
rectangle('Position', [-1 -1 2 2], 'Curvature', [1 1])
line([-1 1], [0 0], 'Color', [0 0 0])
line([0 0], [-1 1], 'Color', [0 0 0])
axis equal
```

Here, we use the unfortunately named `rectangle` function to draw the unit circle and the `line` function to draw the real and imaginary axes. Note also that we have to make sure that the value of r that we pass to the `plot()` function is complex (since we want to plot it on the complex plane) using the `complex()` function, because the roots of a polynomial can be real.

Finally, you can compute and plot the magnitude of the filter's frequency response pretty directly in Matlab:

```
omegahat = [0: 0.01: pi];   % define the frequency axis
z = exp(j*omegahat);        % define the complex frequency axis
H = polyval(b, z);          % evaluate the transfer function polynomial
plot(omegahat, 20*log10(abs(H)/max(abs(H))));
xlabel('$\hat{\omega}$, radians','Interpreter', 'latex');
ylabel('$|\mathcal{H}(\hat{\omega})|$, dB','Interpreter', 'latex')
```

You'll note that the code above plots the ratio of the magnitude of the frequency response to the maximum value of that magnitude. This is done because it is convention to ignore whether a filter amplifies a signal overall; what we are concerned with are the relative amounts that different frequencies are passed or blocked.

Remember, we can define a specific filter using either of the methods above. Sometimes it is easier to understand the filter using zeros and sometimes it is easier to use the coefficients directly. Either method can be used to represent the same filter, and we can go back and forth with the `poly` and `roots` functions (and back and forth between polar and rectangular representations of complex numbers with the `abs`, `angle`, and `exp` functions).

3.1.1 From Filter Coefficients to Transfer Function and Frequency Response

Given the coefficients of an FIR filter we can solve for the zero locations and the frequency response. For example the two-point averaging system is given by:

$$y[n] = \frac{1}{2}x[n] + \frac{1}{2}x[n-1] \tag{5}$$

we can find the transfer function by rewriting the filter using the delay operator, z:

$$\begin{aligned} Y &= \frac{1}{2}X + \frac{1}{2}z^{-1}X & (6)\\ H(z) &= \frac{Y}{X} = \frac{1}{2}(1+z^{-1}) & (7)\end{aligned}$$

If we're interested in the *zero location* we can then multiply $H(z)$ by z/z to obtain:

$$H(z) = \frac{\frac{1}{2}(z+1)}{z} \tag{8}$$

The root of the numerator, $z = -1$ is the location of the only zero (the root(s) of the denominator for an FIR filter are always at $z = 0$, and do not affect the frequency response). We can also derive the frequency response from this by remembering that $H(e^{j\hat{\omega}}) = \mathcal{H}(\hat{\omega})$ or that $z = e^{j\hat{\omega}}$. From the zero location, $z = -1$, we can immediately tell that the frequency response is zero at $e^{j\hat{\omega}} = -1$ or $\hat{\omega} = \pi$. With a zero at an angle of $\hat{\omega} = \pi$, this is a low-pass filter. As a warm-up, use Matlab and the coefficients of equation 5 to verify the expected zero placement and frequency response.

3.1.2 From Zero Placement to Filter Coefficients

When we are given the zero placement, we can very easily determine the filter coefficients because those zeros are the roots of a factored polynomial. For example, given two complex conjugate zeros, z_1 and z_2 (i.e., the real parts are equal, $\mathrm{Re}[z_1] = \mathrm{Re}[z_2] = \mathrm{Re}[z_{1,2}]$, and the imaginary parts are negatives of one another $\mathrm{Im}[z_1] = -\mathrm{Im}[z_2]$ or, equivalently in polar coordinates, $z_1 = re^{j\hat{\omega}_0}$ and $z_2 = re^{-j\hat{\omega}_0}$), the transfer function is:

$$\begin{aligned}
H(z) &= (z - z_1)(z - z_2)/z^2 && (9)\\
&= (z^2 - (z_1 + z_2)z + z_1 z_2)/z^2 && (10)\\
&= 1 - 2\,\mathrm{Re}[z_{1,2}]z^{-1} + r^2 z^{-2} && (11)\\
&= 1 - 2\,\mathrm{Re}[r(\cos(\omega_0) \pm j\sin(\omega_0))]z^{-1} + r^2 z^{-2} && (12)\\
&= \underbrace{1}_{b_0} - \underbrace{2r\cos(\omega_0)}_{b_1} z^{-1} + \underbrace{r^2}_{b_2} z^{-2} && (13)
\end{aligned}$$

At this point, we can rewrite the transfer function as the filter's generating equation, using the delay operator z^{-k}, $y[n] = x[n] - 2r\cos(\omega_0)x[n-1] + r^2 x[n-2]$. This allows us to read off the filter coefficients: $b_0 = 1$, $b_1 = -2r\cos(\omega_0)$, and $b_2 = r^2$.

3.2 Frequency Response and Pole-Zero Plots

Step 1.1 Consider a filter that computes a running average of three points of our input signal (a *three-point averager*):

$$y[n] = \frac{1}{3}\sum_{k=0}^{2} x[n-k] = \frac{1}{3}x[n] + \frac{1}{3}x[n-1] + \frac{1}{3}x[n-2] \qquad (14)$$

a. Draw a block diagram for this filter.

b. How many zeros will this filter have?

c. Find and sketch the zero locations using pencil and paper, then use Matlab to verify this.

d. From the plot of zero locations, sketch the magnitude of the frequency response as a function of $\hat{\omega}$ by hand and verify this using Matlab. How does the minimum of the magnitude of the frequency response relate to the polar representation of the zero locations? What kind of filter would you say this is?

Step 1.2 A *first-difference* filter is an approximation to a discrete derivative operation. Its defining equation is:

$$y[n] = x[n] - x[n-1] \qquad (15)$$

a. Draw a block diagram for this filter.

b. Derive the transfer function, $H(z)$, for this filter. From this, determine the expression for the frequency response, $\mathcal{H}(\hat{\omega}) = H(e^{j\hat{\omega}})$.

c. From the transfer function, determine the filter's zero locations and sketch them. Check your results with Matlab.

d. From the zero plot, sketch the magnitude of the filter's frequency response as a function of $\hat{\omega}$. Use Matlab to check your results. What kind of filter would you say this is?

e. Use Matlab to compute this filter's response to the following input. Generate an analog signal that is a sinusoid with amplitude of 1, frequency of 2, and duration of 1. Sample it at 32 samples/second and quantize it using 16 bits. What is the digital frequency, $\hat{\omega}$, of this $f = 2$Hz sinusoid?

f. Produce a figure with two plots: the top should be the original digital signal, X, and the bottom should be the filtered signal, Y.

g. Examine the plots of X and Y. Note that Y appears to be a scaled and shifted sinusoid of the same frequency as X. The exception is the first point, $y[0]$. Explain why $y[0]$ is different (if you are unsure, consider the defining equation and the input values to it for $n = 0$).

h. Estimate the frequency, amplitude, and phase of Y directly from its plot (ignoring $y[0]$).

i. To compare these measurements to theory, use your expression for the filter's frequency response to calculate the amplitude and phase at the digital frequency $\hat{\omega}$ you determined above. How do these compare to what you determined from the Matlab plots?

Step 1.3 Just as we can compute a discrete first derivative with a first-difference filter, we can compute a discrete second derivative with a *second difference filter*.

a. Use your expression for the transfer function of the first difference filter and your knowledge that the combined transfer function of two filters cascaded, or connected in series, is the product of their individual transfer functions to determine the transfer function for a second-difference filter.

b. Draw a block diagram for this filter.

c. Determine the filter's zero locations and sketch them. Check your results using Matlab.

d. From the zero plot, sketch the magnitude of the filter's frequency response as a function of $\hat{\omega}$. Use Matlab to check your results. What kind of filter would you say this is?

Step 1.4 Consider a feedforward filter with complex conjugate zeros at $z_{1,2} = -0.5 \pm j0.5$.

a. Determine the filter coefficients.

b. Use Matlab to plot the frequency response of the filter.

c. What are the effects of the zeros on the frequency response? What kind of filter would you call this?

3.3 Linearity and Cascading Filters

Step 2.1 A system is called *linear* if a sum of different inputs produces an output that is the sum of the outputs for the inputs taken individually. Perform a simple test of the linearity of the filter from step 1.2 by doubling the input amplitude in Matlab ($X' = 2X = X + X$). How does the new output amplitude compare to the old one?

Step 2.2 In one of the self-test exercises in the textbook, two filters with transfer functions $H_1(z) = b_0 + b_1 z^{-1}$ and $H_2(z) = b'_0 + b'_1 z^{-1}$ were connected in series, and it was shown that they could be connected in either order to produce the same composite effect (the same overall transfer function). Redo this exercise using the *defining equations* for the two filters, i.e., $y_1[n] = F_1(x[n])$ for the filter with transfer function $H_1(z)$ and $y_2[n] = F_2(x[n])$ for the filter with transfer function $H_2(z)$. In other words, show that $F_2(F_1(x[n])) = F_1(F_2(x[n]))$.

Step 2.3 Use Matlab to implement a 50% duty cycle square wave with amplitude 1, frequency 2Hz, and duration 1s. Sample and quantize it appropriately (to make your figures look nicer, feel free to chose a sampling rate much higher than the minimum). Send the resultant digital signal through the previously-defined three-point averager filter, and then the output of that filter through the first difference filter. Plot the input and output. What does the output of this combined filter look like?

Step 2.4 Now, switch the order you apply the filters so that the first difference filter is first and the three-point averager is second. Plot the input and output. How does the output of this configuration compare to that of the preceding step? Does this match what you expected? Why or why not?

4 Let's Catch Some Z's

This lab covers the z-transform, used to convert arbitrary digital signals to the frequency domain. It also exercises the relationship between a filter's transfer function and impulse response and how the operations of multiplication and convolution, respectively, can be used to compute a filter's output.

4.1 The z-transform, Transfer Function, & Impulse Response

A discrete signal $x[n]$ has a z-transform $X(z)$ defined by the following equation:

$$X(z) = \sum_{n=0}^{\infty} x[n]z^{-n}$$

With this definition lets investigate a feed forward filter with ten coefficients, $\{b_0, b_1, \cdots, b_9\}$. Recall that the Matlab `filter` function allows us to specify a filter in terms of its *coefficients*, but we can also think of it as being defined in terms of its *transfer function.* Considering the b_k coefficients of the above feed forward filter, the `filter` function implements the transfer function:

$$H(z) = \sum_{k=0}^{9} b_k z^{-k} \tag{16}$$

In previous labs we have computed the transfer function using the delays of the *defining function.* Mathematically, we were actually taking the z-transform of the *impulse response*! In this example, the impulse response is:

$$h[n] = \sum_{k=0}^{9} b_k \delta[n-k] \tag{17}$$

where $\delta[k]$ is the unit impulse and only has a non-zero value at $k = n$. $H(z)$ and $h[n]$ form a z-transform pair, $h[n] \overset{z}{\leftrightarrow} H(z)$. It should now be obvious why feedforward filters are also known as finite impulse response filters — their impulse response only has a *finite* number of values. To compute the output, $y[n]$, using the impulse response we use *convolution.* Namely, we *convolve* the input, $x[n]$, by the impulse response, $h[n]$,

$$y[n] = x[n] * h[n] = \sum_{k=0}^{9} x[k]h[n-k] \tag{18}$$

And, indeed, Matlab has a `conv` function to do this convolution. Alternatively, we can compute a filter's output by multiplying the transfer function by the z-transform of the input to yield the z-transform of the output:

$$Y(z) = H(z)X(z) \tag{19}$$

From a practical point of view, of course, it makes more sense to implement a filter in terms of its impulse response. However, for filters with long impulse responses, it is sometimes more convenient to represent them mathematically using the transfer function (which we now know is just the z-transform of the impulse response!).

4.2 Z-Transforms

Step 1.1 On paper, compute the z-transform, $X(z)$, of

$$x[n] = \begin{cases} (-1)^n & n \geq 0 \\ 0 & n < 0 \end{cases} \tag{20}$$

Note that this is an infinite geometric series. What are the locations of any pole(s) (roots of the denominator polynomial) or zero(s) (roots of the numerator polynomial)?

Step 1.2 Evaluate the frequency response of $X(z)$ from step 1.1, $X(z)\big|_{z=e^{j\hat{\omega}}}$, by sketching it by hand. What kind of filter is this?

Step 1.3 Consider the z-transform:

$$X(z) = 1 - 2z^{-1} + 3z^{-3} - z^{-5} \tag{21}$$

Write the inverse z-transform, $x[n]$, as a table of values for corresponding n values.

4.3 Impulse Response

Step 2.1 Consider a filter with a transfer function

$$H(z) = 1 + 5z^{-1} - 3z^{-2} + 2.5z^{-3} + 4z^{-8} \tag{22}$$

What is the defining equation for this filter, $y[n] = F(x[n])$?

Step 2.2 What is the output sequence of the filter of Step 2.1 when the input is $x[n] = \delta[n]$? Verify this using Matlab.

Step 2.3 The impulse response of a filter is $h[n] = \delta[n] + 2\delta[n-1] + \delta[n-2] - \delta[n-3]$, or equivalently, $h[n] = \{1, 2, 1, -1\}$, $n = \{0, 1, 2, 3\}$. Determine the response of the system to the input signal $x[n] = \{1, 2, 3, 1\}$, $n = \{0, 1, 2, 3\}$ by hand. Use Matlab to check your results. Include a figure that shows both the input and output signals; make sure the reader can clearly see what the signal values are (the `stem` plot function should help to ensure this is the case).

Step 2.4 Change the input to the filter of Step 2.3 to be $\delta[n]$. What are the output values? How do they compare to the impulse response? Include plots of the filter input and output values in your report.

Step 2.5 Use Matlab to determine the output of the filter {1/3,1/3,1/3}, $n = \{0, 1, 2\}$ for the input:

$$x[n] = 4 + \sin[0.25\pi(n - 1)] - 3\sin[(2\pi/3)n] \tag{23}$$

Include a listing of your Matlab code and a figure with plots of the filter input and output in your report. Is the result expected? Why or why not?

Step 2.6 Create your own Matlab *function*, `convolution`, to implement a convolution function. To test your function, make sure it works exactly like the Matlab `conv` and `filter` functions by providing the same input to each and subtracting their outputs. Use the filter $\{1/3, 1/3, 1/3\}$, $n = \{0, 1, 2\}$ from step 2.5.

4.4 Canceling Sinusoidal Components

Filters can be designed to cancel sinusoids. Implement a filter in Matlab with the following impulse response:

$$h[n] = \delta[n] - 2\cos(\pi/4)\delta[n - 1] + \delta[n - 2] \tag{24}$$

Step 3.1 Plot the frequency response for this filter. What are the zero locations?

Step 3.2 Use as an input to this filter the signal $x[n] = \sin \hat{\omega} n$, using the two frequencies $\hat{\omega} = \pi/2$ and $\hat{\omega} = \pi/4$. You will need to choose appropriate analog signals, with convenient frequencies and durations, and then sample them appropriately so they have the correct digital frequencies. Make sure to verify that you get the correct digital frequencies and that plots you make are convenient for the reader (for example, neither too many nor too few cycles)! Compute the filter in Matlab for each of these two inputs, plotting the input and output of each. When do you get cancellation?

Step 3.3 Can you modify the filter coefficients to cancel the other sinusoid? If so, show your work.

5 To Infinity and Response!

By the end of this lab you should feel comfortable manipulating and using feedback filters for simple problems. You should also be comfortable with the concept of a filter with an infinite impulse response. All feedback filters have an infinite impulse response and are also known as IIR filters. Feedback filters use the previous outputs of the filter, feeding them back to compute the output for the current sample. The "fed back" outputs are weighted by coefficients, a_ℓ.

5.1 A Note About Matlab Filter Coefficients

Note that the Matlab `filter` function uses *negative* values for the a_ℓ (feedback) coefficients, from the transfer function. In other words, in the text, a second-order feedback filter's defining equation might be:

$$y[n] = a_1 y[n-1] + a_2 y[n-2] + b_0 x[n] \tag{25}$$

$$y[n] - a_1 y[n-1] - a_2 y[n-2] = b_0 x[n] \tag{26}$$

This yields the transfer function:

$$Y(z)(1 - a_1 z^{-1} - a_2 z^{-2}) = b_0 X(z) \tag{27}$$

$$Y(z)/X(z) = \frac{b_0}{1 - a_1 z^{-1} - a_2 z^{-2}} \tag{28}$$

$$H(z) = \frac{b_0}{1 - a_1 z^{-1} - a_2 z^{-2}} \tag{29}$$

The coefficients used by `filter`, rather than being the a_ℓ from the defining equation (25) are the *negative* a_ℓ from the transfer function (29) — the ratio of two polynomials. In other words, to properly compute the above filter in Matlab, you will need to use:

```
a = [1.0 -a1 -a2];
b = [b0];
y = filter(b, a, x);
```

Note also that in this example the filter includes the $a[0]$ coefficient (of course, per Matlab one-based indices, as `a(1)`), which we will always leave as 1.0 (it's the first "1" in the denominator of the transfer function).

And finally, note that one form of the `filter` function takes a fourth argument, which is the initial conditions for the feedback delays (used in computing the output values that have delay terms which come before the first value in the x vector). These default to all zero if not specified.

5.2 Feedback Filters as Recurrence Relations

You may notice that the defining equation for a feedback filter is in the form of a recurrence relation. In fact, we can use a feedback filter to implement a recurrence relation if we set the input to be an impulse, $x[n] = C\delta[n]$, with amplitude C being the initial value for the iteration. Let's start out with the Fibonacci sequence, which you'll remember to be:

$$F[n] = \begin{cases} 1 & n < 2 \\ F[n-1] + F[n-2] & n \geq 2 \end{cases} \tag{30}$$

We can rewrite this recurrence relation as:

$$y[n] = y[n-1] + y[n-2] + x[n] \tag{31}$$

and we will get the Fibonacci sequence *if* we input an impulse (hence, the appearance of the $x[n]$ on the right hand side, which serves only to initialize the filter). In Matlab, this can be done trivially by taking a vector of all zeros — let's call this vector `x` — and setting its first value only (`x(1)`) to C. This is also a very good demonstration of the first "I" in the acronym "IIR": the impulse response of this filter has infinite duration.

Step 1.1 If we set $x[n] = \delta[n]$ in (31), we should see that the impulse response of this filter is indeed the Fibonacci sequence. Implement this filter in Matlab and verify that its impulse response is the Fibonacci sequence. What are the values of the coefficients that you used?

Step 1.2 What is the value for $n = 19$ (`y(20)` in Matlab)?

Step 1.3 Is this filter stable?

Step 1.4 Let's do something similar with the recurrence relation for computing the series $y[n] = 1/3^n$ in the text (as always, remember that Matlab indices start at 1). Set the coefficients for a feedback filter to implement equation (5-43) in the text, $y[n] = 1/3y[n-1] + x[n]$. What are the filter coefficients?

Step 1.5 What are the pole location(s) for this filter?

Step 1.6 Now use Matlab to calculate the impulse repsonse. Set the amplitude of the input impulse to be 0.99996. Is this filter stable? Is its impulse response consistent with the result of iterating equation (5-43) in the textbook?

5.3 Telephone Touch Tone Dialing

Telephone touch pads generate dual tone multi frequency (DTMF) signals to dial a telephone. When any key is pressed, the tones of the corresponding column and row in the table below are generated, hence it is a "dual tone" code. As an example, pressing the 5 button generates the tones 770Hz and 1336Hz summed together.

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

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

It is possible to decode such a signal by first using a *filter bank* composed of seven bandpass filters, one for each of the frequencies above. When a button is pressed, it will produce a combination of two tones, and thus, at the decoder end, two of the bandpass filters will produce significantly higher outputs than the others. A good measure of the output levels is the average power at the filter outputs. This is calculated by squaring the filter outputs and averaging over a short time interval.

Step 2.1 First of all, please write a Matlab `DTMFCoder` function. This function should take in one argument — a telephone key number — and return a digital waveform containing the appropriate summed tones. Internally, it should do this by generating AnalogSignals, summing them, and then sampling them at 8kHz and quantizing them at 16 bits. DTMF signal duration should be 1s. For each of the seven tone frequencies in Hz, what is the corresponding digital frequency in the range $[0, \pi]$?

Step 2.2 In this step, please construct a bandpass filter for the 697Hz tone. Use a feedback filter with complex conjugate poles. Locate these complex conjugate poles at the correct location for ±697Hz. Use equation (5-38) of section 5.1.4 of the text to set the radius of those poles so that the closest other tone frequency, 770Hz, lies outside the passband (in other words, to set the bandwidth so that it is significantly smaller than twice the difference between 697Hz and 770Hz). What were your pole locations?

Compute the corresponding filter coefficients. You can verify your filter performance by plotting its frequency response.

Verify that the filter output for `DTMFCoder` output for keys 1, 2, and 3 are pretty much identical, and that all other buttons produce much lower amplitude output. In your report, include a plot of the filter output for one of the buttons 1, 2, or 3 and a plot for one of the buttons 4, 5, or 6.

Step 2.3 Now we are ready to decide whether a particular frequency is present. Write a Matlab `RMS` function that takes a vector as input and returns a scalar root mean squared value for it — this function should square each value of the vector, take the mean of those

[1]More information can be found at: http://en.wikipedia.org/wiki/DTMF

squares, and then take the square root of that mean. Determine the RMS filter output for telephone buttons 1, 2, and 3 and compare them to the other phone buttons. You should see a much higher value for 1, 2, and 3 than the other buttons; additionally, the RMS values for those three buttons should be almost identical. What are the RMS values you get for pressing 1 versus 4?

Step 2.4 Now we will assemble a filter bank. Implement filters in Matlab for each of the six other DTMF frequencies and verify that they work as expected. Now, write a Matlab function `DTMFDecoder` that takes a single input — a vector (for which you will use the `DTMFCoder` output). `DTMFDecoder` should compute the RMS value of the output of each of the seven filters in the filter bank. It should output (to the Matlab console) these RMS values, and then detect the two highest values. It should use a lookup table or equivalent logic to decode which button was "pressed," and output (to the Matlab console) that button.

6 Joe Fourier Was Not a Discrete Fellow

6.1 Lab Background

By the end of this lab you should have a firm understanding of how the Discrete Fourier Transform (DFT) can be implemented exactly using the Fast Fourier Transform (FFT). In addition you should be able to identify common problems using the DFT to analyze signals. You will also be familiar with a new tool, the spectrogram, that uses the DFT as a function of time.

6.2 Implementing the DFT

Recall that the DFT can be implemented directly from the analysis equation. For a length N signal $x[n]$,

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi}{N}nk} \qquad \text{for } k = 0, 1, 2, \cdots N-1 \tag{32}$$

The order of the implementation is $O(N) = N^2$. The following Java code outlines implementation of a 256-point DFT. It is written without any algorithmic speedup (i.e., it exactly mirrors equation 32).

```java
public class MyDFT
{
  // x is the input and y is the magnitude of the complex DFT
 public void computeDFT(double[] x, double[] y)
  {
  double[] yImag = new double[256];
  double[] yReal = new double[256];

  double twoPiOverN = 2*Math.PI/256;

  for (int k = 0 ; k < 256 ; k++)
  {
    yReal[k] = 0;
    yImag[k] = 0;
    for (int n = 0 ; n < 256 ; n++)
    {
      yReal[k] += x[n]*Math.cos(n*k*twoPiOverN);
      yImag[k] += -x[n]*Math.sin(n*k*twoPiOverN);
    }
    y[k] = Math.sqrt(yReal[k]*yReal[k] + yImag[k]*yImag[k]);
  }
 }
}
```

Note that, unlike in Matlab, there is no native support in Java for complex numbers so this arithmetic is written out explicitly in the code above. For example, the equation $y = x \times e^a$ (where x is a real number) must be explicitly written out using Euler's formula, and the real and imaginary portions saved in separate variables, $y_{real} = x \times \cos(a)$ and $y_{imag} = x \times \sin(a)$.

The FFT algorithm can be used to reduce the computation time of the DFT to $O(N) = N \log_2 N$ — a significant speedup for even modest length signals.

6.3 The FFT in Matlab

Matlab includes a `fft` function (there are many more related operations in the Signal Processing Toolbox, but we are sticking to "vanilla" Matlab here). You can look at the documentation for this function; pay especial attention to the frequency values that correspond to each element of the vector that this function returns, and to how to specify the number of points in the FFT it computes.

Step 1.1 Implement your own `myFFT` function in Matlab that takes a real-valued vector as input, computes a 256-point FFT, and returns a real-valued vector that is the magnitude of the (single-sided, i.e., only positive frequencies) FFT. Implement this function using loops (i.e., do not use recursion). The first part of this code performs a bit reversal on the input array. You can use the following code to perform the bit reversal efficiently (efficient bit reversal algorithms in other languages are typically more complex):

```
% Assume that you want to do a bit reversal of the contents of the vector x
indices = [0 : length(x)-1];                   % binary indices need to start at zero
revIndices = bin2dec(fliplr(dec2bin(indices, 8))); % bit reversed indices
revX = x(revIndices+1);                        % Add 1 to get Matlab indices
```

This code converts the vector of in-order indices to an array of 8-character strings (representing those indices as binary 8-bit numbers). Each string is then reversed, and converted back to numbers. The resultant vector of indices (which is what they are after we add one to each) is applied to the signal to pull its entries out into a new vector, with the order of the entries in bit-reversed order.

Once you've done this, all you need to do is iterate over the array $\log_2 N$ times! Remember that you're performing complex arithmetic and to compute the magnitude of the output array once the FFT is computed. Include a copy of your Matlab code in your report.

Step 1.2 Check your results using the Matlab `fft` function. Your results should be quite similar, if not identical; this should be apparent by comparing graphs of the outputs. Take the FFT of a sinusoid with a frequency of $\pi/4$ radians per second using your FFT implementation and the `fft` function.

Step 1.3 Prove that the number of complex multiplies performed by your code is $O(N \log N)$.

6.4 Using the DFT

Step 2.1 Create a sum of two sinusoids. Use the built-in Matlab `fft` function (it will allow you, among other things, to apply an n-point FFT to a signal with more than n samples) to compute the FFT of the sum and then plot the FFT magnitude. Use frequencies of 0.13π and 0.19π for the two sinusoids. Make sure that there are at least 256 samples in each waveform, as you'll want to use a 256-point FFT! Also, make sure you correctly compute the corresponding frequency values (either from 0 to π or $-\pi$ to π, depending on whether you prefer plotting the single-sided magnitudes or not) for the FFT X-axis, and label it appropriate in your graph. What does the result look like? Does this make sense?

Step 2.2 At what index (or indices) does the FFT magnitude reach its peak value(s)? What frequency (or frequencies) does this correspond to?

Step 2.3 Change the frequencies of the sinusoids to 0.13π and 0.14π. Repeat steps 2.1 and 2.2. Do the results still make sense?

Step 2.4 Replace the sum of two sinusoids with your `DTMFCoder` function from lab 6. Input a selection of button values. Do the FFT function and plot show the separate frequencies for each button?

6.5 Spectrograms

Comparing FFT graphs (as in the step 2.4 above) can be difficult to do. But what if we could analyze the frequency content of a signal as a function of time? That would make it easier to see differences in frequency if a signal started changing (like a string of DTMF keys pressed in turn). To do this we will need a new tool called the *Spectrogram.* The spectrogram is simply an algorithm for computing the FFT of a signal at different times and plotting them as a function of time. The spectrogram is computed in the following way:

1. A given signal is "windowed." This means that we only take a certain number of points from the signal (for this example assume we are using a window of length 128 points). To start out, we take the first 128 points of the signal (points 1 through 128 of the input vector).

2. Take the FFT of the window and save the it in a separate array.

3. Advance the window in time by a certain number of points. For instance we can advance the window by 64 points so that we now have a window of indices 65 through 192 from the input signal array.

4. Repeat steps 1–3, saving the FFT of each window, until there are no longer any points in the input array.

5. Form a 2-D matrix whose columns are the FFT magnitudes of each window (placed in chronological order). In this way, each row represents a certain frequency, each column represents a given instant in time, and the value of the matrix represents the magnitude of the FFT.

The result is called a *spectrogram* and is usually displayed as an RGB image where blue represents small FFT magnitudes and red represents larger FFT magnitudes (the *Jet* colormap if you are familiar with color visualizations). There is an art to choosing the correct parameters of the spectrogram (i.e., window size, FFT size, how many points to advance the FFT, etc.). Each parameter has tradeoffs for the time and frequency resolution of the resulting spectrogram. For our purposes here, we will not be concerned with these tradeoffs. Instead we will be more interested in getting familiar with analysis using spectrograms.

The Matlab Signal Processing Toolbox has a `spectrogram` function, but you can retrieve a drop-in replacement for it from `http://www.ee.columbia.edu/ln/rosa/matlab/sgram/` that will work without that toolbox. See the online Matlab documentation for the `spectrogram` function to understand what the parameters to that function are (though, for our purposes, you should just be able to use the call `y = myspecgram(x)`.

Step 3.1 Use your `DTMFCoder` function again. Instead of computing its FFT, compute its spectrogram instead. What does the spectrogram look like for button 1? Are both frequencies present?

Step 3.2 For this step, use `DTMFCoder` to generate the codes for multiple buttons — at least three — and concatenate them together to form a single vector. Does the spectrogram make it easier to judge the frequency content of the keys? Can you clearly see when the signal changes from one key to another?

7 I'm So Compressed

7.1 Sound and images in MATLAB

MATLAB has a number of functions that can be used to read and write sound and image files as well as manipulate and display them. See the `help` for each function for details. Ones that you'll likely find useful are:

audiorecorder Perform real-time audio capture. This may or may not work on your system; it is critically dependent on your hardware and MATLAB's support thereof. You may find it simpler to record and edit audio files with other software and then save it to a file later read into Matlab; this might give you greater control over the recording.

audioinfo Get information about an audio file, including number of channels, how compressed, sampling rate, bits per sample, etc.

audioread Read part or all of an audio file, returning sampled data and sampling rate.

audiowrite Write an audio file; the format is indicated by the filename provided. All platforms support `wav`, `.ogg`, and `.flac`; Windows and OS X support `.m4a` and `.mp4`, too.

sound Play vector as a sound. This allows you to create arbitrary waveforms mathematically (e.g., individual sinusoids, sums of sinusoids) and then play them through your speakers. This is the function to use if you're values are scaled within the range of -1 to +1.

soundsc This function works like `sound`, but first it scales the vector values to fall within the range of -1 to +1. Much of the time, you won't have your waveforms pre-scaled, and you'll use this function.

imfinfo Provides information about an image file, including dimensions, bit depth, etc.

imread Read image from graphics file. This function (and `imwrite`) supports many file types.

imwrite Write image to graphics file.

image Display image. The image is displayed within normal Matlab axes, which you can label or otherwise modify as usual. Any 2D array can be displayed as an image (for example, this is what's done with the output of `myspecgram`). You can alter the colormap used for this display, if you like. There are other functions that plot 2D data as wire meshes and surfaces.

imagesc Like image, but scales the image values first so that they span the full range of the current colormap.

There are a number of other functions available; *read the Matlab documentation to get the full scoop on what is available and how it works.*

Within MATLAB, audio is a 1-D vector (stereo is $n \times 2$, but we won't deal with stereo) and gray-scale images are 2D arrays (color images are 3-D arrays). For simplicity's sake, make sure any sound files you use are mono.

7.1.1 Data Types

MATLAB supports a number of data types, and the type that the file I/O functions return often depends on the kind of file read. Regardless, almost all of the MATLAB functions you'll use to process data return doubles. For example, consider the following commands that read a color image, convert it to gray-scale, and display it:

```
>> A = imread('/tmp/ariel.jpg');
>> B = mean(A,3);
>> imagesc(B)
>> colormap(gray)
>> axis equal
>> whos
  Name      Size                               Bytes  Class

  A       100x55x3                             16500  uint8 array
  B       100x55                               44000  double array
```

The (color) image is read into `A`, which, as you can see from the output of the `whos` command, is a 100x55x3 unsigned, 8-bit integer array (the third dimension for the red, green, and blue image color components). I convert the image to gray-scale by taking the mean of the three colors at each pixel (the overall brightness), producing a `B` array that is `double`. The `imagesc` function displays the image, the `colormap` function determines how the array values translate into screen colors, and the `axis equal` command ensures that the pixels will be square on the screen.

Much of the time, we will just perform our calculations using `double` data types. However, we can use functions like `uint8` to convert our data.

At the very least, you can get real audio files from `http://faculty.washington.edu/stiber/pubs/Signal-Computing/`; I assume that you'll have no trouble locating interesting images (please keep your work G-rated). Don't use ones that are too big, to avoid issues with Canvas uploads of large lab report files.

7.2 Lossless image coding

The simplest way to compress images is *run-length coding* (RLE), a form of repetitive sequence compression in which multiple pixels with the same values are converted into a count and a value. To implement this, we need to reserve a special image value — one that will never be used as a pixel value — as a flag to indicate that the next two numbers are a (count, value) pair, rather than just a couple pixels. Let's apply RLE to three different kinds of images: color photographs, color drawings, and black-and-white images (e.g., a scan of text). You can choose images you like, or get these from the book web site: http://faculty.washington.edu/stiber/pubs/Signal-Computing/ariel.jpg (color photo), http://faculty.washington.edu/stiber/pubs/Signal-Computing/cartoon.png (color drawing), and http://faculty.washington.edu/stiber/pubs/Signal-Computing/text.png (black-and-white).

Step 1.1 Write a MATLAB script to read an image in, convert it to gray-scale if the image array is 3-D, and scale the image values so they are in the range [1, 255] (note the absence of zero). Verify that this has worked by using the `min` and `max` functions. Convert the results to `uint8` and display each using `imagesc`.

Step 1.2 At this point, you should have in each case a 2-D array with values in the range [1, 255] inclusive. To simplify matters, we will treat each array as though it were one-dimensional. This is easy in MATLAB, as we can index a 2-D array with a single index ranging from 1 to $N \times M$ (the array size). Write a RLE function that takes in a 2-D array and scans it for runs of pixels with the same value, producing a RLE vector on its output. When fewer than four pixels in a row have the same value, they should just appear in the vector. When four or more (up to 255) pixels in a row have the same value, they should be replaced with three vector elements: a zero (indicating that the next two elements are a run code, rather than ordinary pixel values), a count (should contain 4 to 255), and the pixel value for the run. Verify that your RLE function works by implementing a RLD function (RLE decoder) that takes in a RLE vector, N, and M and outputs a 2-D $N \times M$ array. The RLD output should be identical to the RLE input (subtracting them should produce an array of zeros); verify that this is the case. For each image type, compute the compression factor by dividing the number of elements in the RLE vector by the number of elements in the original array. What compression factors do you get for each image?

7.3 Lossy audio coding: DPCM

In *differential pulse-code modulation* (DPCM), we encode the differences between signal samples in a limited number of bits. In this section, you'll take an audio signal, apply DPCM with differing numbers of bits, see how much space is saved, and hear if and how the sound is modified. A slight complicating factor is that all of the data will be represented using `double`, however, we will limit the values that are stored in each double to integers in the range $[0, 2^{\text{bits}} - 1]$ (where "bits" is the number of bits we're using).

Step 2.1 Find a sound to work with; you can use http://faculty.washington.edu/stiber/pubs/Signal-Computing/amoriole2.mat if you like. Likely, it will have values that are not integers; convert the values to 16-bit integer values by scaling (to $[0, 2^{16} - 1]$) and rounding (reminder: the vector's type will still be `double`; we're just changing the *values* in each element to be integers in that range). Verify that this conversion produces no audible change in the sound. What is the MATLAB code to do this initial quantization?

Step 2.2 Now write a DPCM function, `DPCM`, that takes the sound vector and the number of bits for each difference and outputs a DPCM-coded vector. Note that the MATLAB `diff` function will compute the differences between sequential elements of a vector. Your DPCM function should:

1. compute the differences between the samples,
2. limit each difference value to be in the range $[-2^{\text{bits}-1} - 1, 2^{\text{bits}-1} - 1]$, producing a quantized vector and a vector of "residues" (you will generate two vectors). For each quantized difference, the "residues" vector will be zero if the difference, Δx_i, is within the above range. Otherwise, it will contain the amount that Δx_i exceeds that range (i.e., the difference between Δx_i and its quantized value, either $-2^{\text{bits}-1} - 1$ or $2^{\text{bits}-1} - 1$).
3. "make up for" each nonzero value in the "residues" vector. For each such value, scan the quantized difference vector from that index onward, and modify its entries, up to the quantization limits above, until all of the residue has been "used up".
4. The final result is a single coded vector that your function should return.

For example, let's say that we're using 4 bit DPCM and that some sequence of differences is $(\Delta x_1 = 10, \Delta x_2 = 5, \Delta x_3 = -3)$. The range of quantized differences is $[-7, +7]$, and so the quantized differences are $(7, 5, -3)$ and the residues are $(3, 0, 0)$. Since the first residue is nonzero, we proceed to modify quantized differences starting with the second one, until we've added three to them. The resulting final quantized differences are $(7, 7, -2)$.

Step 2.3 Implement an inverse DPCM function `IDPCM` that takes the first value from the original sound vector and the DPCM vector and returns a decoded sound vector. There's no way for us to know where the losses in the encoding occurred, so we just use the DPCM values as differences. Note that the MATLAB `cumsum` function computes a vector with elements being the cumulative sum of the elements of its input vector, and that you can add a constant to all of the elements of a vector using the normal addition operator.

Step 2.4 Test your DPCM and IDPCM functions by coding your sound using 15, 14, 12, and 10 bit differences. At what point does the coding process produce noticeable degradation? To investigate further, see how few bits you can use to encode the sound and still detect some aspect of the original sound. Is this surprising to you?

7.4 Lossy image coding: JPEG

Step 3.1 In this sequence of steps, we will use frequency-dependent quantization, similar to that used in JPEG, to compress an image. Start with your gray-scale, continuous-tone image from step 1.1 (if you used a color image, convert it to gray-scale as you did in step 1.1). The MATLAB image processing or signal processing toolboxes are needed to have access to DCT functions, so we'll use the `fft2()` and `ifft2()` functions instead. To do a basic test of these functions, write a script that loads your image, converting it to gray-scale if necessary, and then computes its 2D FFT using `fft2()`. The resulting matrix has complex values, which we will need to preserve. Display the magnitude of the FFT (remember to use the `abs` function to get the magnitude of a complex number) using `imagesc()`. Do this for each image type. Can you relate any features in the FFT to characteristics in the original image?

Step 3.2 Use `ifft2()` to convert the FFT back and plot the result versus the original gray-scale image (use `imagesc()`) to check that everything is working fine. Analyze the difference between the two images (i.e., actually subtract them) to satisfy yourself that any changes are merely small errors attributable to finite machine precision). Repeat this process for the other images.

Step 3.3 Let's quantize the image's spectral content. First, find the number of zero elements in the FFT, using something like `origZero=length(find(abs(origX)==0));`, where `origX` is the FFT. *Remember to exclude the DC value in the* `fft2` *output in figuring out this range.* Then, zero out additional frequency components by zeroing out all with magnitudes below some threshold. You'll want to set the threshold somewhere between the min and max magnitudes of `origX`, which you can get as `mn=min(min(abs(origX)));` and `mx=max(max(abs(origX)));`. Let's make four tests, with thresholds 5%, 10%, 20%, and 50% of the way between the min and max, i.e., `th=0.05*(mx-mn)+mn;`. Zero out all FFT values below the threshold using something like:

```
compX = origX;
compX(abs(origX)<th) = 0;   % Uses logical array indexing
```

You can count the number of elements thresholded by finding the number of zero elements in `compX` at this point and subtracting the number that were originally zero (i.e., `origZero`). This is an estimate of the amount the image could be compressed with an entropy coder. Express the number of thresholded elements as a fraction of the total number of pixels in the original image and make a table or plot of this value versus threshold level.

Optional. Note that the FFT may have very high values for just a few elements, and low values for others. You might plot a histogram to verify this. Not including the DC value likely will eliminate the highest value in the FFT. However, some of the other values may still be large enough to produce too large of a range. Can you suggest an approach that will take this into account? How does the JPEG algorithm deal with or avoid this problem?

Step 3.4 Now we will see the effect of this thresholding on image quality. Convert the thresholded FFT back to an image using something like `reconstructed = abs(ifft2(compX));`. For each type of image and each threshold value, plot the original image and the final processed image. Compute the mean squared error (MSE) between the original and reconstructed image (mean squared error for matrices can be computed as `mean(mean((original-reconstructed).^2))`). What can you say about the effects on the image and MSE? Collect your code together as a script to automate the thresholding and reconstruction, so you can easily compute MSE for a number of thresholds. Plot MSE vs. threshold percentage (just as you plotted fraction of pixels thresholded vs. threshold in step 3.3).