Laboratory 5: The Discrete Fourier Transform (DFT)

1. Objectives

- Learn how to compute and interpret the discrete Fourier Transform (DFT) of a DT signal
- Learn how to use the DFT to compute a very fast DTFT
- Gain real-world experience by examining the frequency distribution within a vocal recording

2. Introduction

In the previous lab you learned how the DTFT was the discrete-time version of the continuous-time concept of the Fourier Transform. You discovered that it was periodic in 2π, and was typically plotted over the region 0 to π, where the highest frequency (corresponding to a signal like [1 -1 1 -1]) occurs at $\omega = \pi$ rads/sample. You ended by examining its use for analyzing the frequency content of a sampled EKG waveform.

In this lab you will work with the Discrete Fourier Transform (DFT), which is the discrete-time version of the continuous-time Fourier Series concept; it is not a continuous function of frequency like the DTFT, but rather is a sequence itself, of the same length as its input. Unlike the DTFT, algorithms exist to calculate the DFT in a blindingly fast manner, and as explored in this lab the DFT can be used to approximate the DTFT to arbitrary precision. Recall the length of time it took to analyze the 2000 sample EKG in the previous lab? With DFT's (and a little more programming effort) we cut the processing time to less than 1% of our previous time.

3. Discrete Fourier Transform (DFT)

A. Definition of the DFT

- The N-point DFT of the a finite-length sequence x[n] defined for $0 \le n \le N-1$ is given by:

$$X[k] = \sum_{n=0}^{N-1} x[n]\, e^{-j2\pi kn/N}$$

  Note the convention of using the variable n as the time-domain index and k as the frequency-domain index.

- The N-point DFT X[k] of the signal x[n] defined for $0 \le n \le N-1$ is simply the frequency samples of its DTFT $X(e^{j\omega})$ evaluated at N uniformly-spaced frequency points between 0 and 2π, or in other words,

$$X[k] = X(e^{j\omega})\big|_{\omega=2\pi k/N}$$

- The inverse DFT can reconstruct the time-domain signal x[n] from the frequency domain sequence X[k] as follows:

$$x[n] = \sum_{k=0}^{N-1} X[k]\, e^{j2\pi kn/N}$$

B. Intuition about the DFT

First, a common point of confusion: the name Discrete Time Fourier Transform, or DTFT, sounds almost identical to the name Discrete Fourier Transform, or DFT. Some will cite historical reasons for this, but I believe it's just because EE sounds harder than it really is. Because the names sound so similar, practicing engineers never call them by the full names but instead just by the acronyms DTFT and DFT. Remember the difference because the DTFT with the longer acronym generates an output much longer than its input sequence length (its output is a continuous function of ω, even though only given a finite number of data samples), but the DFT with the shorter acronym returns a frequency-domain data vector exactly as long as its time-domain input sequence.

The DFT is the digital equivalent of the continuous-time Fourier Series coefficients, just as the DTFT is the digital equivalent of the continuous-time Fourier Transform. Recall in EE230 that the Fourier Series was used to analyze the frequency components of periodic signal x(t). If the signal x(t) had a period of T (meaning it had a fundamental frequency of $\omega_o = 2\pi/T$), the trigonometric version of the Fourier Series decomposition found the amount of sine and cosine waves that would create x(t). Specifically, it found coefficients a_k and b_k that when multiplied by $\cos(k\omega_o t)$ and $\sin(k\omega_o t)$ summed to create x(t). Think of it this way: the Fourier Series decomposition transformed a continuous-time signal to a discrete series of coefficients a[k] and b[k].

In EE230 we also studied the complex exponential form of the Fourier Series decomposition, which although less-intuitive than the trigonometric form of the Fourier Series, was simpler from the perspective that it only generated a single sequence c[k] of coefficients representing the periodic waveform x(t). Specifically, it generated a series of complex coefficients c[k] that when multiplied by the complex exponential $e^{-j2\pi\omega/T}$ and summed over all k results in the original signal x(t).

To intuitively understand what the DFT looks like, realize these facts:

- **It is as long as the input sequence**. The DFT of x = [1 4 2] will be of length 3, although the DTFT will be a continuous function of ω.
- **It will be complex.** The DFT of [1 4 2] is roughly [7 -2-j1.7 -2+j1.7]. Like the DTFT it is usually graphed twice, once to show the magnitude and once to show the phase. The magnitude of the above signal, for example, is about [7 2.6 2.6].
- **The DFT is a sampled version of the DTFT**. The magnitude of the DTFT of a random signal of length 10 and the magnitude of the DFT of the same signal is shown in Figure 1 and 2 below. Specifically, the sampling starts at ω = 0, continues every 2π/N where N is the signal length, and thus the last number in the DFT (the at n=N-1) corresponds to the DTFT just before 2π frequency (exactly, 2πN/(N-1)).

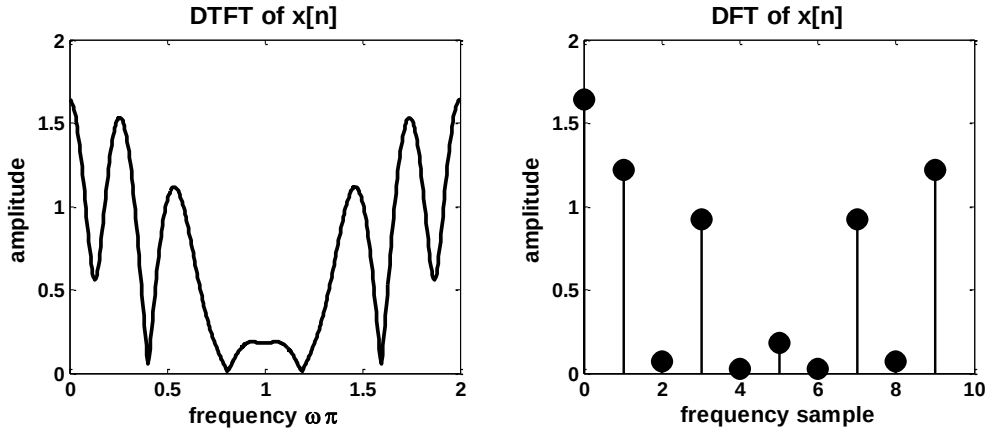

Figure 1: The DTFT and DFT of a random signal of length 10. Note that the DTFT is continuous, and is plotted here as it varies between 0 and 2π. The DFT is discrete, the same length of the signal, and is the sampled version of the DTFT. The DFT sample k

corresponds to the DTFT frequency $\omega = 2\pi k/N$, where N is the length of the signal. You can select the frequencies of the DTFT that you wish to plot, since it is defined over all values of ω (of course, it has a period of 2π). You cannot choose the frequencies of the DFT to plot; a length N signal will have a length N DFT, corresponding to DTFT frequencies from 0 to one sample less than 2π.

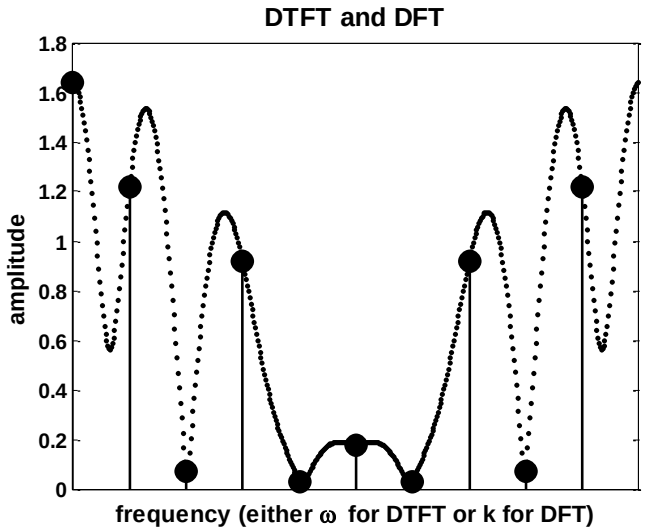

Figure 2: The DTFT and DFT of Figure 1 superimposed. Note that there is no x-axis scale on this figure because it doesn't make sense to have one; the DTFT is continuous from 0 to 2π and the DFT is discrete from 0 to N-1. To create this plot I linspaced the DTFT between 0 and 10 to force it to line up with the DFT.

- **The first number of the DFT is the sum of x[n]**, which of course is proportional to x[n]'s DC component. The index k in the DFT X[k] of length N corresponding to the highest frequency in the time-domain signal x[n] is at k=1+N/2. If this number is fractional then the highest frequency corresponds to that value of k both rounded up and down (e.g. if N=5, 1+N/2 = 3.5, so the highest frequencies correspond to X(k=3) and X(k=4)). All this comes from the intuition that a typical DFT magnitude will look like this X=[a b c d c b], where a is proportional to the zero frequency energy, d is proportional to the highest-frequency energy at π, and b, c are evenly-spaced between these extremes. Since the DTFT is symmetric around π, the DFT values after d are mirrored from the indexes before it. As another example, X2=[a b c d e d c b], or X2 = [a b c d d c b]. In words this is confusing; read it while looking at Figure 1 to make it intuitive.

- **The benfits of zero padding**. We said the DFT is just a sampled version of the DTFT. It is far preferable computationally to compute the DFT using very fast commonly-available algorithms such as the Fast Fourier Transform (FFT) algorithm, rather than the DTFT which takes far more time to compute. How can we make the DFT more finely sample the DTFT to obtain finer frequency resolution? Since the length of the signal and its DFT are the same, pad the end of the signal with zeros. See Figure 3 for an example.

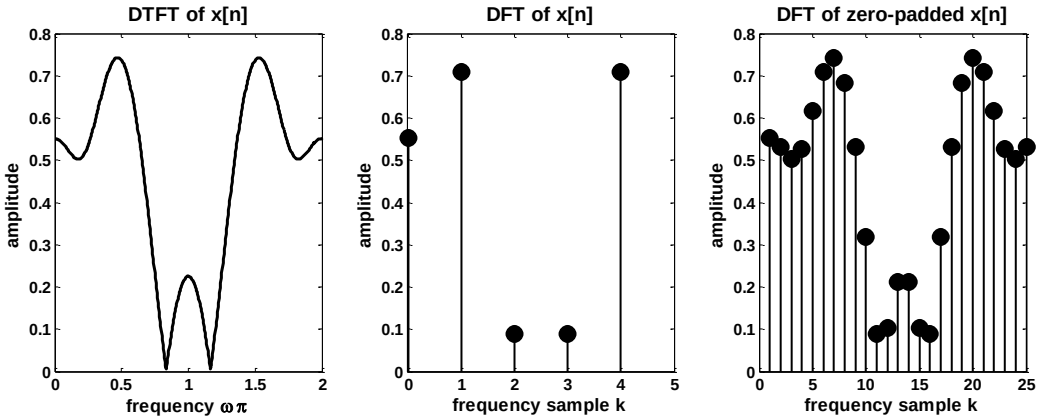

Figure 3: Zero-padding x[n] to allow the DFT to more finely sample the DTFT. The left panel shows the DTFT of a length 5 sequence x[n]. The center panel shows its length 5 DFT, which is a crude approximation to the true DTFT. The right panel shows the DTFT of the x[n] padded with 20 additional zeros. In Matlab terms, if `x = rand(1,5)` the center panel shows `stem(abs(fft(x)))`, and the right panel shows `stem(abs(fft( [ x zeros(1,20) ] )))`.

- **Parseval's Relation**. Parseval's relation is a precise statement comparing the energy in a signal x[n] to its DFT X[k]. If the signal is a sampled voltage connected to a 1Ω resistor then its power is $x^2[n]$ (think $P = IV = V^2R$). Just as in the continuous-time domain the total energy is the integral of the power over all time, in the discrete-time domain the total energy is the sum of the power over all n. So, the total energy in the signal is the sum of $x^2[n]$ over all n. Parseval's Relation intuitively states that the total energy in a signal x[n] of length N is equal to N times the total energy in the DFT of the signal, or in equation form,

$$\sum_{n=0}^{N-1} (x[n])^2 = N \sum_{k=0}^{N-1} (X[k])^2$$

C. Matlab and the DFT: the command `fft`

- Matlab computes the DFT using `fft`. For example, to compute X[k] = the DFT of x[n], the Matlab command would be `X = fft(x)`. Recall that Matlab is case-sensitive, so that x and X are two different variables.

- A potentially-confusing name: FFT vs. DFT. The DFT is the name of the transform; the FFT is one algorithm used to compute the DFT. FFT stands for "Fast Fourier Transform", which is too bad because it is an algorithm, not a transform in itself, and worse it doesn't compute the Fourier Transform (which as we said was a continuous-time concept you learned in EE230) but rather the DFT. Even worse, the folks at Matlab decided to call their DFT operation "FFT", naming it after the algorithm rather than the operation. This is ECE.

Problem 1: DFT

a. Write a Matlab program that takes a vector x and an integer N, zero-pads the end of x to make it N long, and then computes its DFT (using Matlab's `fft` command). The first line should look like:

```
function X = Lab5_Prob1a(x, N)
```

To test it, use x = [-1 2 1 -1] as data, and use the program to stem plot the magnitude of the DFT of x after padding it to make it 10 long and again after padding to make it 50 long. The indexing for DFT's always starts at 0. Use subplot to stem plot them in a single column of two rows. Comment on the effect of the more extensive zero padding.

b. **Challenge**: Repeat part a, but this time compare the same sequence padded to length 5 with the sequence padded to 50. Their relationship looks at first glance to be different than the relationship between the sequences padded to 10 and 50 you found in part a. Explain.

D. Circular Shifting

- A circular shift is more easily shown by example than by definition. If the sequence x[n] that begins at n=0 is {1, 2, 3, 4, 5, 6} is circularly-shifted to the right by 2, it remains beginning at 2 but becomes {5, 6, 1, 2, 3, 4}. The mathematical notation for this intuitively-obvious operation is awkward: $x[\langle n-2\rangle_6]$, where the 6 signifies that we wrapped digits shifted beyond the 6th place back to the zero index position. This means $\langle n\rangle_N$ = n modulo N = n with however many multiples of N removed from it so that it is between 0 and N-1. Again by example, $\langle 0\rangle_6 = 0$, $\langle 1\rangle_6 = 1$, $\langle 5\rangle_6 = 5$, but $\langle 6\rangle_6 = 0$ (since it wraps around to become 0), $\langle 11\rangle_6 = 5$, $\langle 12\rangle_6 = 0$, and $\langle 13\rangle_6 = 1$.

- The N-point *circular convolution* of two length-N sequences x[n] and h[n], $0 \le n \le N-1$, is denoted by $y_c = x[n] \circledast h[n]$ and is defined by

$$y_c[n] = \sum_{m=0}^{N-1} x[m]h[\langle n-m\rangle_N]$$

where $\langle n\rangle_N$ = n modulo N. That's the mathematical definition, but is far easier for a human to compute: to create a N-length circular convolution, just take the linear convolution, and then wrap the end to make it only N long. For example, if a linear convolution resulted in [1 2 3 4 5 6], to create a 4 point circular convolution, wrap the 5th position around to add the first, and the 6th position around to add to the second, resulting in [1+5 2+6 3 4] = [6 8 3 4]. To create an 8 point circular convolution, just zero pad to create [1 2 3 4 5 6 0 0]. It's important to realize: linear convolution is usually what is desired, but using the DFT as described below, circular convolution is what you get unless you zero pad (also described below).

A fast way to circularly convolve two signals x[n] and h[n] using a computer is to compute their DFT's, multiply them, and then compute the resulting inverse DFT, since multiplying in the frequency domain is like convolving in the time domain. Although fast, it cannot be done in real-time (you need to take the DFT of the full length signal x[n]).

- The fastest way possible to get the linear convolution y[n] = x[n]*h[n], where x is N samples long and h is M samples long, is to
  1) Pad the end of x[n] with M-1 zeros (to make the wrapped numbers in circular convolution all zeros, therefore making it equivalent to linear convolution)
  2) Similarly, pad the end h[n] with N-1 zeros. Now both x[n] and h[n] will be N+M-1 long.
  3) Take the DFT of both, multiply together, and then take the IDFT.

Matlab example: if `x = [1 2]` and `h = [1 2 3]`,
`conv (x, h)` is the same as
`ifft( fft([x 0 0]) .* fft([h 0]) )`
but the ifft method is much faster. Entering the data by hand on a PC you won't notice the difference; they each take less than a millisecond to compute. On a wireless networking system that has to process 100Mbytes = 800Mbits of data each second in small data packets, this difference is huge.

- The DFT has a number of useful properties frequently exploited in real-world applications. We will explore the two most common properties in this lab: the circular time-shifting property and the circular frequency-shifting property.

  - *Circular Time-Shift Property*: If X[k] is a N-point DFT of a length-N sequence x[n], then the N-point DFT of the circularly time-shifted sequence $x[\langle n-n_0 \rangle_N]$ is $X[k]e^{-j2\pi kn_0/N}$. This should look vaguely familiar, since in EE230 you learned that time-shifting a continuous signal x(t) to make it $x(t-t_0)$ was the same thing as multiplying the Fourier transform of $X(e^{j\omega})$ by $e^{-j2\pi t_0}$.

  - *Circular Frequency-Shift Property:* If X[k] is a N-point DFT of a length-N sequence x[n], then the N-point DFT of the sequence $x[n]e^{j2\pi nk_0/N}$ is the circularly frequency-shifted sequence $X[\langle k-k_0 \rangle_N]$

Problem 2: Circular Shifting

The function below circularly shifts a signal. It is available on the course web page.

```
function y = circularshift(x,M)
% CIRCULARSHIFT circularly-shifts sequence x by M to the right
L = length(x);
M = rem(M,L);
if M < 0
    M = M + L;
end
y = [x(L-M+1:L) x(1:L-M)];
```

a. What is the purpose of the command rem in the function circularshift?
b. Explain how the function circularshift works.

The function below illustrates circular shifting. It uses the function circularshift developed above.

```
% Lab5_Prob2
% Illustration of Circular Shifting of a Sequence
%
clf;
M = 2;
a = 0:9;
b = circularshift(a,M);
L = length(a)-1;
n = 0:L;
%
subplot(2,1,1)
stem(n,a); axis([0,L,min(a),max(a)]);
title('Original Sequence')
%
subplot(2,1,2)
stem(n,b); axis([0,L,min(a),max(a)]);
str = sprintf('Sequence Circularly Shifted by %g Samples',M);
title(str)
```

c. What parameter determines the amount of time-shifting?
d. What happens if the amount of time-shift is greater than the sequence length?
e. What does the function `sprintf` do?

Problem 3: The Circular Time-Shifting Property of the DFT

The following script illustrates the effects of the DFT on a circularly time-shifted signal. Download it from the EE431 homepage and run it. It uses the function circularshift developed in a previous problem.

```
% Lab5_Prob3
% Circular Time-Shifting Property of the DFT
%
clf;
x = [-3 -2 -1 0 1 2 3 2 1 0 -1 -2]
n = 0:length(x)-1;
y = circularshift(x,1);
X = fft(x);
Y = fft(y);
%
subplot(3,2,1)
stem(n,x);
title('The Original Sequence')
%
subplot(3,2,2)
stem(n,y);
title('The Circularly-Shifted Sequence')
%
subplot(3,2,3)
stem(n,abs(X)); grid
ylabel('amplitude')
title('Magnitude DFT of Original Sequence')
%
subplot(3,2,4)
stem(n,abs(Y)); grid
ylabel('amplitude')
title('Magnitude of Circularly Shifted Sequence')
%
subplot(3,2,5)
stem(n,unwrap(angle((X)))*180/pi); grid
ylabel('phase (degrees)')
title('Phase DFT of Original Sequence')
%
subplot(3,2,6)
stem(n,unwrap(angle((Y)))*180/pi); grid
ylabel('phase (degrees)')
title('Phase of Circularly Shifted Sequence')
```

a. Run this program and include the plot. What amount of right circular time-shift is shown?
b. Comment/explain what the time-shift does to the signal's DFT.
c. Modify the program to run with a time-shift of -1. Comment/explain.
d. **Challenge:** Again, let x = [-3 -2 -1 0 1 2 3 2 1 0 -1 -2]. Why does a right shift of 1 appear to leave many of the phase values of the DFT as zero? Why does a right shift of 2 appear to make even more of the phase values equal to zero?

Problem 4: Circular Convolution using the DFT

Begin by analyzing the following Matlab function that finds the circular convolution of two sequences, available on the course homepage.

```
function y = circularconv(x1,x2)
% CIRCULARCONV circularly-convolves sequences x1 and x2
L1 = length(x1); L2 = length(x2);
if L1 ~= L2, error('Sequences of unequal lengths'), end
y = zeros(1,L1);
x2tr = [x2(1) x2(L2:-1:2)];
for n=1:L1
    sh = circularshift(x2tr,n-1);
    h = x1.*sh;
    y(n) = sum(h);
end
```

a. What is the purpose of the ~= operator in the function `circularconv`?
b. Explain how the function `circularconv` works.
c. For the rest of this problem, let sequence x1 = [1 1 -1 1] and sequence x2 = [1 2 3 4]. Let y equal the linear convolution of x1 and x2. Find y either by hand (using graphical methods) or by using Matlab.
d. Using the function `circularconv` above, find the circular convolution of x1 and x2. Explain how you could find the circular convolution directly if you knew the linear convolution you found in part c.
e. Another way to find the circular convolution directly is to note that multiplication of DFT's in the frequency domain is like circular convolution in the time domain. Compare your answer from part d above with this: `ifft(fft(x1) .* fft(x2) )`.
f. **Challenge:** Taking the DFT's of signals appears to be a needlessly complex way of doing circular convolution; the function `circularconv` seems to be much simpler since it does not involve complex exponentials. Which is faster? Create two signals x1 and x2 that each have 50,000 random elements and compare the length of time it takes to circularly convolve the two using the `circularconv` function used in part a and the DFT method given in part e. To create a long sequence of random numbers use the command `rand`. To measure the length of time it takes to perform an action using `circularconv` or the `fft` method, use `tic` and `toc` as follows:

```
>> tic, circularconv(x1,x2); toc
```

or

```
>> tic, ifft( fft(x1) .* fft(x2) ); toc
```

`Tic` saves the system clock into a hidden global variable and `toc` reads that variable, compares it with the current time, and displays the difference. Record how long it takes with each method, and be prepared to be surprised. It may take a while for this program to run!

Problem 5: Linear Convolution using the DFT

Program Lab5_5 shows how one can compute linear convolution using DFTs by padding the end of each input sequence with zeros so that the circular convolution of the DFT multiplication is equivalent to linear convolution.

```
% Lab5_Prob5
% Linear Convolution Using the DFT
%
x = [1 2 3 4 5];
h = [2 2 0 1];
%
xPadded = [x zeros(1,length(h)-1)];
hPadded = [h zeros(1,length(x)-1)];
y = ifft(fft(xPadded) .* fft(hPadded));
y = real(y);
disp('Fast linear convolution using the DFT =')
disp(y)
disp('Slow direct linear convolution using conv =')
disp(conv(x,h))
```

a. Run the program. Does convolution using zero-padded DFT's produce exactly the same answer as direct linear convolution?
b. Repeat the above for a different set of sequences for x and h.
c. Write a Matlab function called `fastconv` that takes two sequences and returns their linear convolution using DFTs (it does exactly what the built-in function `conv` does, but in a much faster manner by using DFTs). Use the script given above to guide you. The first line should be
   `function y = fastconv(x, h)`
d. **Challenge**: Are there reasons why Matlab does not use this faster method of convolution?

Problem 6: DTFT and DFT problem

Let x = [1 2 3 4 3 2]. In a single-column four-row subplot, plot its DTFT over $0 \le \omega \le 2\pi$, its DFT using stem, the DFT of the signal replicated 10 times (e.g. [x x x…]), and the DFT of x with 60 zeros after it. Comment on the results.

4. Real-World Problems – Extract A Pitch From A Vocal Recording

Music formats

There are two fundamentally different ways to record music: as a sampling of the time-varying sound pressure sensed by a microphone (e.g. CD recordings, WAV files, MP3's) or as a list of sound events where each event has an on-time, off-time, pitch, and sound type associated with it (e.g. sheet music, MIDI files, electronically-generated music). These two ways are analogous to the difference between a .jpg picture of a handwritten letter and a Word document containing the letter's text. The .jpg can reveal subtle nuances in the way the handwriting was printed, but takes a huge amount of room and except for cropping or amplitude scaling can't be edited. The text file can be easily edited, but its regular typefaces can't capture the way that a handwritten letter reflects the mood of the writer. Similarly, an MP3 can perfectly capture the performance, but cannot be edited beyond simple cropping or sound scaling.

Sheet music is the standard way that musicians record compositions (Figure 4). It is comparatively easy to play an instrument from sheet music (not necessarily play well, but play accurately), but quite difficult to do the reverse: create sheet music after hearing a melody. A program that could hear a sound recording, for instance in a WAV file, and determine the MIDI or sheet music representation of the song, would enable a powerful set of new software applications that could, for instance
1) allow anyone to compose their own melodies in sheet-music form by humming into a microphone
2) allow anyone to create sheet music from a CD recording
3) permit truly incredible compression of instrumental recordings. For example the music in Figure 4 takes 5 seconds to play. Using MP3 compression at a fixed rate of 128kb/s this would take 640kb to store. The GIF image of the sheet music took only 13kbytes to store. A MIDI file that records the frequency and on/off times of each note for these bars takes only 215 bytes to store…a savings of about 3000x over MP3 format, or about 30,000x over WAV files!

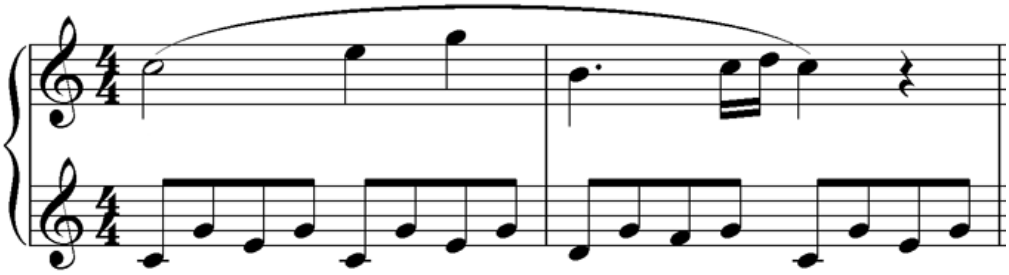

Figure 4: The opening two bars of Mozart's Sonata in C. The vertical axis corresponds to frequency and the horizontal axis corresponds to time. The shape of the note indicates how long it is held. Since there are two staves of music, two different notes play at once. This representation needs an additional specifier to let the musician know what instrument should reproduce the sounds; as written it could be the piano, a violin, or a tuba.

This section of the lab will have you tackle a subset of this problem: you will use the DFT to determine the DTFT of a note you will sing, and from that determine what pitch it was that you sang. The problem is stated at the end of the section, after the software you will need to use is introduced.

Music Theory from an Engineer's Perspective

Music is written with 12 unique notes, increasing in frequency as follows: A, A#, B, C, C#, D, D#, E, F, F#, G, G#. The first A is defined to occur at 440Hz. After G#, the scale wraps to start again at A, this time at exactly twice the frequency, or 880Hz. We differentiate between these two A's by calling the first A440 and the second A880. The frequency ratio between each note is a constant k, and to have 12 geometrically-increasing intervals fit in a span of 2, k must equal the 12^{th} root of 2, or k = 1.059463. For instance, since A440 is defined to occur at 440Hz, the next note A# must occur at 440 * k = 466.2 Hz, and B occurs at 440 * k * k = 493.9 Hz. A880 could have been determined directly by calculating 440 * k * k * (for all 12 notes) = 440 * k^12 = 880.

The notes are circularly wrapped in that as you continue up the scale, after the upper A at 880Hz, the next note is A# at 880 * k = 932.4 Hz, and similarly the next B occurs at 880 * k * k = 987.8 Hz. Each note occurs again at double and half frequencies; this span of frequency is called an "octave". One could also have derived the A# an octave above the already-calculated A#466.2 note to be: 466.2 * 2 = 932.4 Hz, or similarly the higher B as 493.9 * 2 = 987.8 Hz.

Notes also circularly wrap downwards on the scale. The A one octave below A440 occurs at 220Hz. Just as A# above A220 occurs at 220*k, the note below A220, a G#, occurs at 220/k = 207.7 Hz.

Example: What note is 60Hz powerline noise? The A one octave below 220 is 110, and an octave below that is 55. One note above A55 is 55*k = 58.3, and two notes above is 55*k^2 = 61.7Hz. Powerline noise is therefore by chance exactly in between the notes A#58.3 and B61.7, guaranteeing that no matter what key music is written in, powerline hum will sound highly irritating.

To Record Your Voice To A Matlab Array

Use a Matlab audiorecorder object.

1. Create an instance of the audiorecorder and define the sampling frequency, bit depth, and number of channels. For this lab, use a sampling frequency of 22050 Hz, which is a fairly standard frequency for radio-quality sound. Bit depth refers to how many bits are used to store each sample; a 16 bit depth is standard for CD quality. Channels means number of microphones, and can be 1 for mono, 2 for stereo, or 5 or more for surround, and then one will record 1, 2, 5, etc. simultaneous vectors of data, one per microphone.

```
recordObject = audiorecorder(22050, 16, 1);
```

2. Tell Matlab to record 5 seconds of data. To let you know that Matlab is starting and stopping, include disp(‘’) commands.

```
disp('Recording session starting')
recordblocking(recordObject, 5);
disp('Recording session ending');
```

3. Extract the data stored in the audiorecorder object to a vector (or if multiple channels are recorded, into a matrix).

```
v = getaudiodata(recordObject);
```

4. Listen to the sound

```
sound(v, 22050);  % data vector, then sample frequency
```

To Read a WAV File Into Matlab

1. If the wav file is called Filename.wav, you can read it into Matlab as a vector with the command
   `[x, Fs] = audioread('Filename.wav');`
   The wav file will be stored in vector x and the sampling frequency in scalar Fs.

2. To hear the wav file, recall from Lab 3 that you can use the command
   `sound(x, Fs)`

An Improved DTFT Program

The Lab4_DTFT program you used in the previous lab is too slow to use in this exercise. It took several seconds to analyze a 2,000 sample signal; for this project you will record about a one second length (about 22,000 samples). Since the Lab4_DTFT uses an order N^2 algorithm, this increase in size of a factor of about 10 will cause it to require 100 times the processing time, all for a single second of sound. To analyze an entire song this way would be impractical. The new method, which we will call simply DTFT and is available on the course webpage, uses the fast DFT to finely-sample the DTFT. The DFT is implemented using the Fast Fourier Transform (FFT) algorithm, which is order $N \log_2(N)$. For very short sequences of about length 8, both the direct DTFT method of Lab 4 and the DFT method using the FFT take about the same length of time to analyze. A 1.2 second WAV recording that is 32,000 samples long, or 4,000 times longer, will take the DTFT 16,000,000 times as long to analyze, but the DFT only $4 \cdot 10^3 \log_2(4 \cdot 10^3) \approx 50{,}000$ as long, which is over 300 times faster than the direct DTFT method! In practice, this means the DFT can process a 32,000 length sequence well under a second.

The provided program computes at least 1001 samples of the DTFT using the DFT for speed. It first zero pads the end of short input signals to ensure they are at least 1001 in length. It then adds an extra 0 to the end of even-length sequences to ensure the resulting DFT has an odd number of digits to guarantee that the the (L+1)/2 index of the DFT will be exactly the π sample of the DTFT. This is important because the DFT always samples the DTFT from 0 to 2π, and we only want to sample it from 0 to the highest discrete frequency of π. To accomplish this we simply remove the upper half of both the frequency vector and the DFT vector.

Lastly, the function changes the frequency vector from the discrete frequency 0 to π to the continuous frequency 0 to $F_s/2$, where F_s is the sampling frequency. You may find use for this program in industry or graduate school; it is quite useful.

```
function [w,X]=dtft(x,Fs)
% DTFT samples the DTFT of a sequence or sampled signal using a DFT
% [w,X]=dtft(x) returns the dtft of x at the frequencies w
% [f,X]=dtft(x,Fs) returns the DTFT of  x sampled at Fs.
% If no output arguments are requested, it plots the  result

if nargin==1  % If only 1 input argument (ie not sampled)
    Fs=2*pi;  %  2pi is the highest digital signal frequency
end

% zero pad x if needed to make it long enough
x = x(:)';    % guarantee it is a row vector

L=length(x);
if length(x)<1001     % guarantee it is at least 1001 long
    x = [x zeros(1,1001-L)];
end
L=length(x);
if rem(L,2)==0 % guarantee it is an odd length
    x = [x 0];
end
L=length(x);

% take fft of x and drop the high freq mirror symmetry part
X1 = fft(x);
X1((L+1)/2+1:end)=[];

% make the appropriate omega vector
omega = linspace(0,Fs*(L-1)/L,L);
omega((L+1)/2+1:end)=[];

% if no output requested, plot it
if nargout == 0
    plot(omega, abs(X1))
    xlabel('frequency')
    ylabel('amplitude')
    title('magnitude of the DTFT of x[n]')
    figure(gcf)
else % pass the result to the output variables
    w = omega;
    X = X1;
end
```

Problem 7: Identifying musical pitch in sound recordings

a. Create about a one-second .WAV recording of you singing the vowel sound "ahhh" at a comfortable speaking frequency. Crop it as needed to remove any unwanted noise from the start or end of the recording, and compute its DTFT using the above program. Most of the useful information is contained in the lower-frequency part of the graph; zoom in until you see the fundamental and the next four harmonics (the upper frequency will be in the 400Hz – 1kHz range depending on the note you sung) and print the result.

b. Do you see 60Hz noise contamination? (You may need to zoom in or out to see 60Hz). What is your signal to noise ratio in dB, where the "signal" of interest is the amplitude of the fundamental frequency of your voice? How many harmonics of the 60Hz noise do you observe (if any)?

c. What is the fundamental frequency of your note? Zoom as needed to get a precise reading. What is the musical note name?

d. Vowels are strongly periodic, and fairly sinusoidal in appearance; this is why they have such strong fundamental frequencies and clear harmonics. Consonant sounds such as "sh" are very different; they sound like white noise (listen to yourself say "shhhh"). True white noise has a flat magnitude spectra across all frequencies (the opposite of the magnitude spectra of a sine wave); when we say "shhhh" our oral cavity makes a band-pass filter that limits the frequency to a high-frequency band centered somewhere between typically 1kHz and 5kHz. The lack of harmonics indicates the sound is white noise, and not periodic; it has no tone. Analyze a one-second recording of yourself saying "shhhhh" and plot its magnitude spectrum, zoomed to show its sound energy. Identify the approximate low and high cutoff frequency limits of your mouth's bandpass filter.

e. **Challenge**: Write an m-file that, given a 1 second recording at the sampling frequency you worked with, could identify whether the sound is a consonant "sh" or a vowel "ahh". Have it take as an argument the vector sound file and sampling frequency, and display to the screen whether or not it is a vowel or consonant. (Hint: Some of your approaches may require the command `find`, from an earlier lab, to help with indexing).

5. Matlab Commands Used in Laboratory 5

Extracting parts of complex signals

abs(X)	returns the magnitude of complex signal X
angle(X)	returns the phase angle of complex signal X
unwrap(angle(X))	returns the phase angle of X, "unwrapped" so that as a complex number travels smoothly counterclockwise around the complex plane, rather than having the phase suddenly jump from 1∠179.9° to 1∠-179.9° it is unwrapped to go from 179.9° to 180.1°.
real(X)	returns the real part of complex signal X
imag(X)	returns the imaginary part of complex signal X

General Matlab commands

sprintf	combines numbers and text into a single string, e.g. title(sprintf('%g samples',n)) creates a title reading "5 samples" if n=5.
tic, <operation>,toc	returns the time in seconds needed to do the operation. For example, try tic, fft(rand(100)); toc
rand(n,m)	create an n row m column matrix of random numbers, uniformly distributed between 0 and 1

Sound commands

[x,Fs] = audioread('filename.wav')	reads a wav file and stores the samples in x and the sampling frequency in Fs
sound(x,Fs)	plays the sound sampled in x at sampling frequency Fs