SDanial Ludwig

PHYS273H

26 November 2019

**Technical Note: Audio Equalizer**

1. **The Discrete Fourier Transform**
2. Definitions

A close up of a clock

Description automatically generatedThe Discrete Fourier Transform (DFT) decomposes a signal into its frequency components in a way similar to its continuous Fourier counterparts. But while continuous signals may require an infinite basis of frequencies (complex exponentials), discrete, time-limited signals need only a discrete, finite basis. This is shown in the limits of summation in the definition:

A close up of a piece of paper

Description automatically generatedA close up of a logo

Description automatically generatedA picture containing object

Description automatically generatedWhere *F* is the DFT of *f*. The above definition is useful when thinking about *F* and *f* as periodic functions (of period N) on the integers. Sometimes, however, it is easier to understand using vector notation. This is facilitated by the following notation:

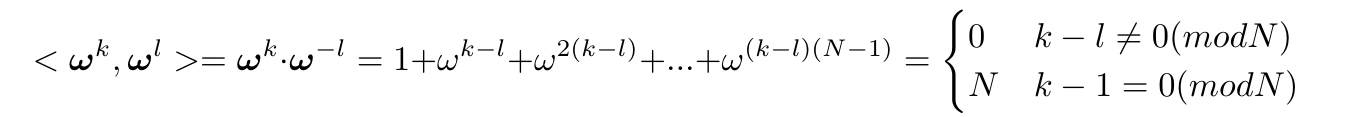
*A picture containing object, clock

Description automatically generated*Now the definition of the transform (called the analysis equation) can be written like this:

A picture containing object, antenna

Description automatically generated This definition makes it clear that we’re just decomposing a regular, finite vector into a basis, and more specifically an orthogonal basis. The orthogonality can be shown using the geometric series summation formula:

Substituting in *w*, and noting that if *k* is a multiple of *N*, we have an integer multiple of 2 pi in the exponent of an exponential:

A picture containing object

Description automatically generatedThus if *k* is *not* an integer multiple of N, the series sums to 0, and if it *is* a multiple of N, all the terms in the series are 1, so it sums to N. Now, taking the complex inner product of two different complex exponential vectors and using the above formula gives:

But *k* and *l* are both between 0 and N-1 (the *F* vector is of length N like the *f* vector), so the only way for *k-l* to be a multiple of N is for *k=l*, so for them to be the same vector. Any two different *w* vectors are thus orthogonal. We can from here derive the second DFT definition using the formula for projecting one vector onto another:

A close up of a logo

Description automatically generated

A screenshot of a cell phone

Description automatically generatedThis inner product presentation lends itself to an easy explanation of the matrix representation of the DFT:

1

--

N

The *rows* of this matrix can be thought of as the conjugate of the *w* vectors, and therefore by the rules of matrix multiplication, each element of *F* is the dot product of *f* and the rows (divided by N).

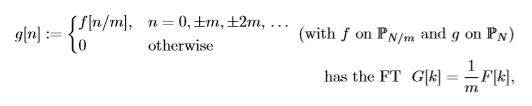
A picture containing object

Description automatically generatedThe past few pages has been about the *analysis* equation, where we break down the original signal into its frequency components. The *synthesis* equation tells us how to build up the signal *from* its frequency components. This is easy to find, though, when thinking of the exponentials as basis vectors—the inverse is just a linear combination of the *w* vectors, where each coefficient tells how much of *f* is “in” that vector, which we already found to be *F[k]*:

1. Important Properties

For the FFT algorithm that I implemented, there are a handful of properties of the DFT that need to be discussed.

1. *Linearity*: the DFT of the sum of two signals is the sum of the 2 DFTs.
2. *A screenshot of a cell phone

   Description automatically generatedTranslation:* the DFT of a shifted signal *modulates* its transform, multiplying it by a complex exponential (analogous to AM—amplitude modulation—radio).
3. *Zero-packing:* up-sampling a signal makes it repeat in the frequency domain, with some scaling

A close up of a logo

Description automatically generatede.g.

1. *Inversion:* the DFT of the DFT of a signal is that signal reflected or time-reversed, with some scaling.

And while it is not necessary for the FFT algorithm, there is another property that is crucial to the interpretation of the DFT: positive and negative frequencies. Unlike the Fourier series, for example, the frequencies of the complex exponential basis vectors don’t monotonically increase with *k*. In fact, the frequencies increase until *k =N/2*, then start *decreasing* back down to 0. *k = N/2* has the maximum frequency of half the sampling frequency, as it oscillates between 1 and -1:

A close up of a logo

Description automatically generated

A screenshot of a cell phone

Description automatically generatedUsing this, we can analyze *F[N/2+1]* and *F[N/2-1]*:

The last column is what gives the first half of the spectrum the name of *positive* frequencies and the second half the name of *negative frequencies*. At the same time, the complex conjugate of the complex exponential is just tags on a negative to its phase, which means that

A picture containing object

Description automatically generatedA picture containing object

Description automatically generatedA close up of a logo

Description automatically generated

**…**

For a real-value signal *f*, that is. *F[0]* is not part of any pair, but it has its own interpretation as the average or DC component of *f*.

What all of this means is that all the information in a DFT is stored in the 1 element, the positive frequencies, and the middle component (we usually assume N to be even); the negative frequencies are redundant, and need to be treated as such.

1. The Fast Fourier Transform

Up until the 1960s, computers had been (approximately) using the matrix representation to compute the DFT of a N-point signal. This amounted to around N^2 complex multiplications, or 4N^2 operations in total. In 1965, James W. Cooley and John W. Tukey published what is now the most frequently cited mathematics paper in the world (Kammler) : “An algorithm for machine computation of complex Fourier series”, which presented a way to compute a DFT using only N log2(N) operations. Ten years later, a researching studying the history of numerical analysis found that Gauss had outlined a similar method in a paper written in 1805.

Basically, the algorithm recursively breaks down the input signal in the “time-domain”, takes the DFT of the pieces, and then builds it back up again in a related way in the “frequency-domain”. The commuting diagrams found in Kammler’s *A First Course in Fourier Analysis* exhibit the basic building blocks:

A close up of text on a white background

Description automatically generated

**Figure 1**

Commuting diagrams for translation and zero-packing

The tau operator represents a shift, the epsilon operator represents modulation, the Z operator represents zero-packing, and the R operators represents repeating.

Say you wanted to compute the DFT of (a,b,c,d). There are two ways you could do it: you could do the DFT straight out, or you could time-shift it, take the DFT of that, and then modulate it. This could be extended to work on a much larger scale, which is what the A close up of a map

Description automatically generatedfollowing two diagrams show:

**Figure 2**

Building up f from components (Kammler)

A close up of a map

Description automatically generated

**Figure 3**

Building up F from components (Kammler)

These parallel trees are just a generalization of the commuting diagrams. Say you have 8-vector *f* at the top of Figure 2 and you want to take the DFT. You could either just take the DFT straight out, or you could break it down in the time-domain, take the DFT of the components, and then build it back up parallel to how it was broken down. The reason why you’d want to do it in the frequency-domain is because the DFT of a single constant is *itself*, so the only true operations would be in the modulation step (all the rest just involves manipulation of data in memory.

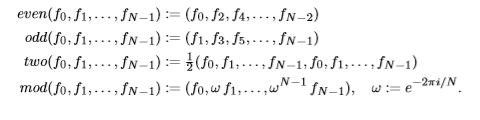
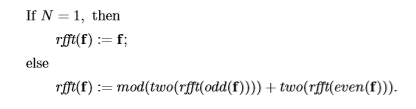
A screenshot of a cell phone

Description automatically generatedImplementing this is relatively simple, although as with all recursive algorithms, it takes a bit of thought. First off, I coded two small helper functions to clean up the *myfft* function:

**Figure 4**

Helper functions for my *myfft* function

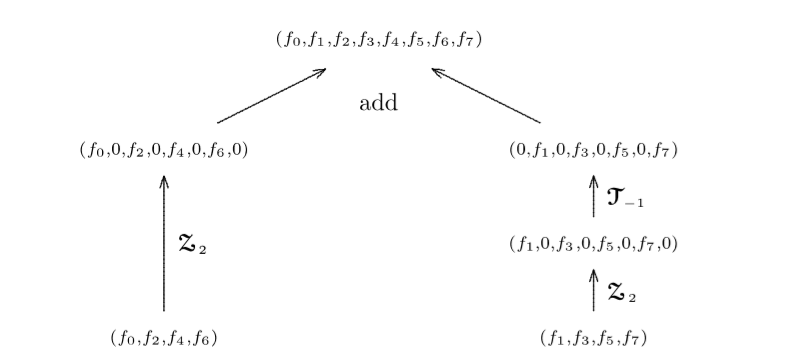
The *repeat* function is the ½ R2 operator found in the tree diagrams, and the *modulate* function is the E-1 operator in the tree diagrams. The comments above them give the examples found in the commuting diagrams (Figure 1).

Before giving my Matlab code for it, I’ll start with the following pseudo-code for the FFT as outlined in an example problem of *A First Course in Fourier Analysis*:

**Figure 5**

Building up F from components (Kammler)

This code exhibits the basic recursive algorithm structure: it calls itself on parts of the input data unless the base condition (input vector is length 1) is met, at which point it starts “working its way up” the function calls. What’s inside the internal *rfft* call (*odd*  and *even*)is represented in the time-domain tree diagram i.e. breaks down the signal, and what’s outside the *rfft* call is represented in the frequency-domain tree diagram i.e. builds up the transform. The *odd* and *even* functions can be seen as shortcuts to doing the inverse zero-packing and inverse shifting that one needs to do to move down one section of the time-domain diagram.



**Figure 6**

Single section of time-domain tree diagram (Kammler)

In order to allow different data-types to be input, I coded a shell function so that the conditions being tested in it don’t get repeated. Additionally, the *odd* and *even* functions in the pseudo-code are easy with Matlab indexing, and therefore did not need separate functions.

The final product:

A screenshot of a social media post

Description automatically generated

**Figure 7**

Code for my FFT function, *myfft*

Using the inversion rule, coding the inverse DFT was straightforward, with the only subtlety being that the time reversal of the vector doesn’t mean simply flipping the vector

A screenshot of a social media post

Description automatically generatedaround; *f[-n]* means that *f[0]* (which is the element at index 1 in Matlab) stays where it is, and the *rest* flip.

**Figure 8**

Code for my inverse FFT, *myifft*

1. Efficiency

As can be expected, the *myfft* function was much more efficient than the matrix DFT, and Matlab’s own *fft* function was much more efficient than that. This is shown in the two plots on the next page. After N = 2^15, the matrix DFT started using up too much memory, so the second plot does not include it, as it plots up to N = 2^20

A close up of a map

Description automatically generatedA close up of a map

Description automatically generated

**Figure 9**

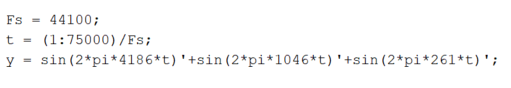
Above: comparisons of all 3 DFT methods

Below: comparisons of my and Matlab’s FFTs

While it looks like *myfft* is growing at a much faster rate, looking at the numerical values of each show that both are roughly doubling at each new point, so growing at O(2^(log2(N)) = O(N). This, however, cannot be correct, as the theoretical efficiency is O(N log2(N)), so probably more data points (and a lot more computer memory) are needed to see the true long-term behavior of the functions.

1. Examples

A screenshot of a cell phone

Description automatically generatedThis section is just for the discussion of some example DFTs. Here’s one for sum of three sine waves at frequencies 261 Hz (C4), 1046 Hz (C6) and 4186 Hz (C8). *Fs* is the *sampling frequency*, representing the rate at which the data points need to be played. The plot shows three distinct peaks with very little noise, as expected.

**Figure 10**

DFT of computer-generated, three-tone signal

A screenshot of a social media post

Description automatically generatedNote that in plotting this, the maximum frequency would be half the sampling frequency *Fs*, or 22050 Hz. After that comes the negative frequencies, which have magnitudes that are a reflection of the first half (positive frequencies). Next we compare the DFT plots of a computer tone 440 Hz (A4) and a violin note 440 Hz (A4):

**Figure 11**

Comparison of normalized spectra for computer and violin A4 (440 Hz)

The spectra shown are normalized, so each signal has the same total energy. The energy of the computer signal is concentrated at 440 Hz, but the energy of the violin is spread out through its harmonics: there’s a peak at around 880 Hz, 1320 Hz, …, all the way up to 3520 Hz. These are *integer multiples* of the fundamental frequency, which we could have expected to appear based on our in-class discussion of the vibrating string. Interestingly, the 7th harmonic does not seem to have a significant contribution, although the 2nd-8th otherwise do.

Here’s the first notes of “Young Forever” by Jay-Z feat. Mr. Hudson:

**Figure 12**

Energy vs. frequency plot for sample song

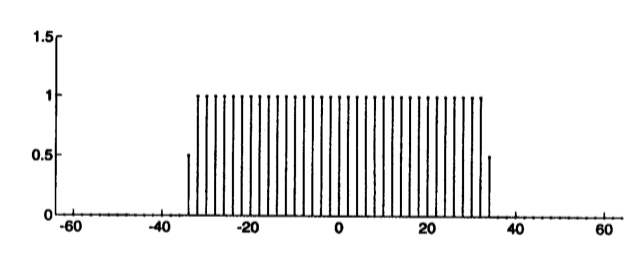
A close up of text on a white background

Description automatically generated

I believe the largest peak represents a G#.

1. **Filtering**
2. Analog and Digital overview

In class, we discussed harmonic circuits—RC, LRC, etc—which have a *natural* frequency, meaning they have the greatest amplitude response when a certain frequency of sinusoidal voltage is applied. In PHYS276 we use various properties of these circuits. When applying a sinusoidal voltage over an RC circuit, taking the output voltage over the capacitor gives you a *low-pass* filter, as at high frequencies the capacitor has a lower relative impedance and therefore the voltage drop mainly occurs over the resistor. Thus, if you have a voltage that is made up of a high and low frequencies, taking the voltage across the capacitor will only reflect the low frequencies. And vice versa when taking the output voltage as the voltage across the resistor, which acts as a ­*high-pass* filter. LRC circuits act more as a *bandpass* filter, letting through not just high or low frequencies, but a certain *range* of frequencies centered at its natural frequency. The width of this range is determined by the Q factor we discussed in class, a higher Q meaning a smaller range of frequencies being responded to (so high “quality” equals high precision).

 The DFT makes it easy to implement basic high-, low-, and band-pass filters. If you have a specific range of frequencies you want to select, you can multiply your transform element-wise with a *window* function, which is identically zero outside of the specified range. The most basic window function is the “box” or “indicator” function:

**Figure 12**

Plot of discrete box function (Osgood)

1. Code

The first goal of this project was to implement the FFT, but the second was to implement a music equalizer that would strengthen or weaken certain ranges of frequencies. This is the first (naïve) version of the code for it:

The function *staticEQ* takes in a music/audio file, a *resolution index*, and a multiplier. Because in a music file, its frequency components are always changing, taking the DFT of the whole file doesn’t make sense. Instead, you have to split it up into sections where the frequency makeup is essentially constant using a *time window*. The size of these sections is the resolution index N. For this first version, I used the box function as the time window (which basically means I just grab every N elements), and where N is a 1024 samples (1024\*Fs is quite small, and 1024 is also a power of 2, which maximizes the efficiency of the FFT).

The *cutoffs* and *bandwidths* vectors contain information about the 4 different frequency ranges I’m using—bass, mids, upper mids, and highs. I combine these with the input multiplier (which contains the desired weighting of each of the ranges) to create the *filterVector*. If N = 8 and the multiplier = [1, 2, 3, 4], then the *filterVector* would then be something like [1 1 2 2 3 3 4 4] (if the frequency ranges were all the same size, which in realitythey’re not). The process, then, is basically

Time window (takes every 1024 samples)

to make yCell

FFT of sections of length 1024 (YCell)

Input signal *y*

Element-wise multiplication by filter vector

Inverse FFT of sections of length 1024

Recombination of pieces into *yOut*

1. Examples

A screenshot of a cell phone

Description automatically generatedThe following three plots show the spectra of the three-frequency computer-generated tone from Figure 10 filtered by the *staticEQ* function:

A screenshot of a social media post

Description automatically generated

A screenshot of a cell phone

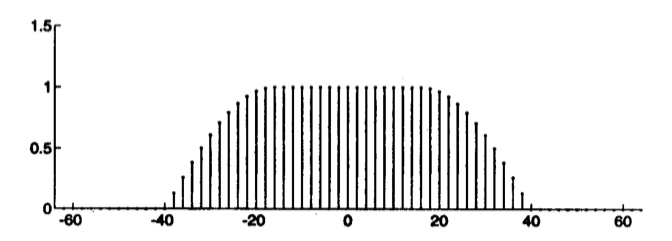
Description automatically generated

While it looks like the original three frequencies are recovered relatively well, the process does add significant noise/artifacts to the audio. This led me to instead of trying to implement a *dynamicEQ* function, that would give a user interface to manipulate in real-time, to see the effect of different window functions. In this first implementation of *staticEQ*, the filter vector introduced discontinuities in the DFT of the signal; ideally, *smooth* functions are the best windows, which led me to implement the following *window* function that looks sort of like the box function but decays sinusoidally to zero:

A screenshot of a social media post

Description automatically generated

Which looks roughly like this figure (Osgood):



This, however, did not decrease the noise added to the signal during processing. In reality, window functions come in pair—if you use one window function in the time domain on analysis, there’s a different one you have to use to synthesize the signal after the processing in the frequency domain. Additionally, as the windows decay to zero, overlapping windows are required in order to have a perfect reconstruction.

References

Kammler, David W. *A First Course in Fourier Analysis*. Cambridge University Press, 2008.

Osgood, Brad. *The Fourier Transform and Its Applications*. Stanford University.