Introduction – 2:20

In this presentation, I will be discussing and comparing the runtime performances of specific algorithm implementations of the discrete Fourier transform.

Here is a brief overview of the main topics of discussion.

From ECE 205, we learned how to work with the continuous Fourier transform. The input x, is a continuous-time signal, possibly defined over an infinite time interval. However, in the context of embedded systems and digital signal processing, we can’t work with such a signal. Digital signals are limited by the sampling interval, the sampling rate, and quantization. So, from the perspective of a computer we’re working with a quantized discrete-time signal and therefore it is only numerically feasible to use the discrete Fourier transform.

Since an integration is simply the limit of a Riemann Sum, the discrete Fourier transform ends up looking like this.

The mapping takes in as input, a discrete-time signal, x. We can think of x, as an array of length N, and we will treat it as such programmatically. Keep in mind that the input signal may be complex.

The output, y, is the signal represented in the frequency domain. Likewise, it is also an array of length N, and its elements may also be complex.

To compute some element of y, at index k, we need to sum together N complex terms. Each complex term is the product of an element of x with a complex factor called the twiddle factor, which we will talk about later. This process is repeated N times, for each element of y.

Now that we’ve been introduced to the DFT, let’s talk about the problem at hand.

DFT Runtime – 1:20

Say you were a co-op student working at the local telecom company, and your boss asked you to implement a generic DFT function in C/C++, how would you go about doing this? The most straight forward way to implement the DFT would be to do something like this [show pseudo code].

This is very simple and elegant. We only need a pair of nested for loops. For each element of y, we simply need to sum N complex products. Some important things to note are that the arrays are passed in by pointer and that the complex exponential can be computed in constant time.

You do your unit tests do make sure the errors are within tolerance. However, when you do your testing on the board, everything goes down south. As it turns out your function was too slow. Indeed, the downside to this implementation would be the excruciatingly slow runtime.

From the nested for loops, it should be obvious that the runtime is O(n2), which is not good enough, we will see shortly how a faster runtime can be achieved.

Twiddle Factors – 3:40

Looking back at our original implementation, it’s not obvious how we would be able to optimize our code for speed. But by inspection, there’s only one line where all the work happens, the complex multiplication. Indeed, we can take advantage of the periodicity and symmetry of complex exponentials, which are called twiddle factors. This brings us to a very important prerequisite topic, twiddle factors.

The twiddle factor refers to the complex factor in the DFT equation. These numbers are simply phasors that lie on the unit circle. So, for some N-point DFT, the twiddle factors correspond to the N-th roots of unity. These are just N equally spaced points along the unit circle, starting on the positive real axis, and going clockwise. Here’s a nice gif that shows you what I’m talking about. So notice that for some N-point DFT, we have N unique twiddle factors.

Before moving on let’s introduce a change of notation, let wdenote the . Think of the super script (ik) as an index, which I’ll conveniently call the twiddle index. We can think of the set of twiddle factors as a hash table of N elements.

To demonstrate the periodicity of the twiddle factors, let’s look at the 8th roots of unity. Starting with the twiddle index of 0, as we increment the twiddle index by 1, we simply jump from one root unity to the next, going clockwise. When the index reaches 8, we’ve conveniently arrived back where we started. And increasing the index further literally takes us in a circle, this demonstrates the periodic nature of the twiddle factors. Mathematical, this property can be expressed by the following equations.

So in theory, the periodic nature of the twiddle factors should allow us to reduce the number of complex multiplications by reusing intermediate results.

Next we examine the symmetrical property of the twiddle factors. To visualize this, start at some arbitrary twiddle factor, if we go clockwise or counter clockwise by N/2 indices, the result is simply the negation of the twiddle factor where we started.

Assuming that N is even, we can make the following generalization.

So in theory, the symmetrical nature of the twiddle factors should also allow us to reduce the number of complex multiplications by using the negation of intermediate results.

FFTs – 5:00

Now onto the fast Fourier transforms.

There are many FFT algorithms, but today we will be examining Cooley-Tukey’s Radix-2 Decimation in Time Fast Fourier Transform. As the name implies, the “fast” adjective describes that the transform can be done in some time faster than O(n2).

This algorithm uses a divide and conquer approach to recursively break down the problem into two smaller sub-problems, achieving a run time of O(nlog2(n) ).

Recall the mathematical equation that we are trying to implement, without using a nested loop. Taking a divide and conquer approach, let’s split this summation into two smaller summations. Let’s sum together the products involving the even indexed terms (starting at 0). Then, let’s sum together the products involving the odd indexed terms (starting at 1).

In the following line, only two changes happen. I’ve moved the 2 in the exponent’s numerator, such that it appears in the denominator. I’ve also factored out this term here by using exponent laws.

What it looks like here, is that we have the N-point DFT, that is composed of two N/2-point DFTs.

That’s very dense notation, so let’s introduce Ek and Ok, to replace those summations. So Ek and Ok are intermediate results of a smaller DFT from the previous stage of the algorithm.

Next, we can take advantage of the periodicity of the twiddle factor. So if we look at Ek, it’s basically the result from a N/2 point DFT, which uses the N/2 th roots of unity. So for some value of the index k, if we increment it by N/2, we’re still looking at the same root of unity, or the same twiddle factor. If we recall earlier, the index offset simply takes us in a circle, back to where we started. So, I hope that we can agree on this equality which I’ve highlighted in red, which would then imply the following. The same argument can be made for Ok.

Next, this is where the magic happens. We just calculated the result for one element of y. Let’s calculate the result for another element of y, namely, at some index k + N/2. From the periodicity property shown earlier, we can make this substitution. And if we recall the symmetry property, for some Nth root of unity, if we offset the twiddle index by N/2, it’s equivalent to negating it.

Essentially, what’s happening here is that we are using the intermediate results Ek and Ok­ to compute not one, but two elements of y, by taking advantage of periodicity and symmetry. This set of operations is called a 2-point butterfly operation. This operation requires one complex multiply and two complex additive operations, because we would never re-compute that term, we would just use a local variable.

Ek and Ok­ are basically the results from a smaller DFT of the previous stage. So, they are basically the outputs from butterfly operations of the previous stage. Therefore, the inputs to this butterfly operation, are the outputs of butterfly operations from the previous stage.

We apply this recursion until the base case, which is a just a 2-point DFT.

Ultimately this recursive procedure can be summarized with a signal flow chart. For some 8-point FFT this chart describes the computations required for each output element. There are only a few notes that I want to make about this chart.

First, I want to bring attention to the number of stages. There are three stages because 8 is 2 to the power of 3. This chart is meant to be read from left to right. So the first stage, involving the base cases is on the left. The third stage produced the final outputs.

Next, I want us to examine the number of butterflies. The “X”s represent a 2-point butterfly operation. Don’t worry too much about exactly how to interpret the flow chart. If we examine the first stage, it’s clear that 4 butterflies occur. It’s harder to see in the next two stages because of the overlaps but they also have 4 butterflies each. So the number of butterflies per stage is N/2.

Therefore, the total number of butterflies is . Recalling that each 2-point butterfly consists of 1 complex multiply and 2 complex additive operations, we can make a final comparison.

Conclusion – 20 seconds

Therefore, by taking advantage of the periodicity and symmetry of twiddle factors, we reduced the runtime of the DFT from O(n2) to O(nlogn), which is quite the amazing feat. Just to further emphasize the importance it all, here’s a quote from Gilbert Strang of MIT, on FFTs saying it is “the most important numerical algorithm of our lifetime”. Truly, one of the greatest instances disruptive innovations in the 20th century.

Thank you. Any questions?