Contents

[Introduction 1](#_Toc481940203)

[Objective 1](#_Toc481940204)

[Fourier Transforms 2](#_Toc481940205)

[DFT 2](#_Toc481940206)

[FFTs 3](#_Toc481940207)

[Body 3](#_Toc481940208)

[Usage in embedded systems and DSP 3](#_Toc481940209)

[Cooley-Tukey FFT Algorithm 3](#_Toc481940210)

[Radix-2 DIT 4](#_Toc481940211)

[Radix-4 DIT 4](#_Toc481940212)

[Overview for usage of radix-4 4](#_Toc481940213)

[Reordering 4](#_Toc481940214)

[Index reversal radix-2 4](#_Toc481940215)

[Index reversal radix-4 4](#_Toc481940216)

[Complex Arithmetic in Butterfly Operations 4](#_Toc481940217)

[2-point Butterfly 4](#_Toc481940218)

[4-point Butterfly 4](#_Toc481940219)

[Use of 4th Roots of Unity 4](#_Toc481940220)

[Twiddle Factors 4](#_Toc481940221)

[Conclusions 5](#_Toc481940222)

[Limitations 5](#_Toc481940223)

[Abstract 5](#_Toc481940224)

# Introduction

## Objective

In this presentation, I will be discussing and comparing the runtime performances of specific implementations (radix-2 vs radix-4) of the (Cooley-Tukey’s) (fixed-point) fast Fourier transform.

Let me elaborate briefly on this very dense objective statement. In the context of embedded systems and digital signal processing, when choosing a function that performs some mathematical operation, there are many aspects that the programmer should consider, such as

* Function size: has to do with the number of instructions stored in memory
* Space complexity: has to do with the local variables, which is important because if you abuse registers, then certain variables will have to be stored on the stack, and we know memory operations tend to cause performance to suffer.
* Accuracy: there are always errors associated with computation, since computers have limited word lengths
* Runtime: how many cycles does the function take to execute

The scope of this presentation will only analyse runtime.

Although we won’t actually see any source code, I will talk about the functions as if they were implemented in C/C++.

The two implementations mentioned, radix-2 and radix-4, refer to the method of divide and conquer; that is, how the problem is recursively broken down into smaller sub-problems. We will see more on that later.

Fixed-point arithmetic is generally used. Using floating-point is much slower because of the instructions required to adjust the mantissa and exponents. I won’t be able to elaborate too much on this topic.

I’ve based the presentation on the expectation that the vast audience has no prior knowledge of fast Fourier transforms, but do have basic knowledge of fourier transforms. The objective of this presentation is not to teach the audience **how** to implement a fast Fourier transform, but rather how optimizations in runtime can be achieved using certain mathematical properties of complex exponentials.

We will go over a brief review of important terminologies and some background information [show Outline].

## Fourier Series

Recall that any periodic continuous- time signal can be decomposed into a summation of cosines and sines.

## Fourier Transforms

We discussed Fourier transforms in ECE 205, and how to compute this operation by hand. But if you were as keen as me, you would still have no idea what we accomplished. So I think it would be appropriate to briefly review this topic.

The Fourier Transform is the mathematical operation that maps a continuous-time signal to a signal represented as a function of frequency.

It’s an extension of the Fourier Series, in the sense that we are now extending the period to approach infinite. The best way to interpret this operation, is that we’re trying to extract from the time domain signal, x(t), the size and phase shift of sinusoids of a particular frequency. Basically, we’re multiplying the signal x(t), with complex signal given by the complex exponential in the integrand. If, …

Let’s say x(t) was just sinwt, and we want to extract the sinusoids of angular frequency w. …

[Show gif of Fourier transform here from wikipedia]

Defining a sinusoid requires three pieces of information. The amplitude, the frequency and the phase shift. The Fourier transform basically lets us extract all this information.

In most cases, it’s much easier to work in the frequency domain.

## DFT

From ECE 205, we learned how to work with the continuous Fourier transform. The signal is continuous over time, and the signal is continuous in “magnitude” (may be complex). This is the idealized real world case.

In our context, the representation of a continuous-time signal is limited by the smpling itnervl, nd the sampling rate, and quantization, so we are dealing with a discrete-time signal.

Nyquist Frequency? tbd

The sampling rate is N/T samples per second. One important thing to keep in mind is, the limitations/constraints associated with the sampling rate. It should be intuitive that our sampling rate should have a minimum value depending on the signal. Consider the following example: x(t) = sin 2t, defined over the time interval 2pi. This signal has a fundamental frequency of 1/pi. But let’s say we only sampled it at 2 points. Once at t = 0, and once at t = pi. That’s a sampling rate of 2/ 2pi = 1/1pi. Just by visual inspection, we could see how this is a bad sampling because we have lost a lot of information. And it would be impossible to reconstruct the signal.

Define Nyquist frequency.

if the signal had two cycles within the sampled time interval.

The topic of quantization is not within the scope of the presentation. But to define it briefly, quantization simply maps a set of (infinite) numbers to a set of finite symbols (e.g ADC) [assumed knowledge].

The DFT is a mapping that can be described by the following mathematical equation.

The mapping takes in as input, a discrete-time signal, x, uniformly sampled over N- points [show graph of example signal], which implies tht it is only defined over a finite time interval, (which we can think of as the fundamental period). 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 (twiddle factor). After the summation, the intermediate result is scaled down by a factor of N. This process is repeated N times, for each element of y. Sometimes, that division is omitted in the computation. The reason being, is first of all division tends to be a very expensive operation, depending on the dt type. Furthermore, it’s not relly necessary becuse the programmer can omit the division operations, and simply reinterpret the elements of y to be scaled up by a factor of N, compared to the expected values.

In our example, we shll see tht we omit the division.

[apriori knowledge?]

What does a complex signal mean? A complex signal can be interpreted as there being two orthogonal components to this quantity that we’re trying to measure. So think of a point on a rotating wheel. Its position vector is represented as an arrow originating from the origin. Consider the shadows that this segment would cast, given two light sources. The projection onto the ground can be interpreted as the real component. And the projection onto the wall can be interpreted as the imaginary component [show diagram?]. These two projections are just scalars; they have a sign and magnitude.

So when the wheel rotates counter clockwise at some constant angular velocity, w0, the position of the point can be represented in the time domain as a complex signal. Namely

) + j sin())

The real part (cosine) would describe the scalar of projection of the point on the ground, and the imaginary part (sine) would represent the scalar projection on the wall.

How do we interpret this output? The best way to examine this is with an example.

Consider the continuous-time signal:

+ 1

Defined over one period T = 2 PI.

For simplicity, let N 4, so sampling at uniform intervals results in these sample points. x = [0, 1, 0, -1]. The corresponding DFT output is

y = [1, 0.5j, 0, -0.5j]

or, in phasor notation

y = [1, 0.5exp(-j2PI ¼), 0, exp(-j2PI ¾)]

A phasor tells gives us two pieces of information, the magnitude of the signal, and phase shift. However in order to reconstruct a sinusoidal signal, recall that we need to angular frequency.

The angular frequency can be deduced from the index of the element. Each index of y, can be thought of as a frequency bin. So each index is associated with a frequency value. This is exactly what we need to reconstruct our signal. So at y index 0, we have 1, this implies that the DC component of the signal is just 1. At index 1, we have this phasor, index 1 is the first fundamental frequency of the signal. Recall that we defined the fundamental period earlier, the fundamental frequency is just 1/T. Index 2 represents the second fundamental frequency, the value is 0, so there is no signal component with that frequency. Now we have the tricky part, the frequency bin given by index 3 cannot be interpreted as the third fundamental frequency. The reason for this is because of the Nyquist limit. The way to interpret this value, is as a frequency bin with a “negative frequency”, which is totally valid. So what we do is we pretend that there’s a zero frequency bin here, and count backwards, towards negative…

Ultimately we have decomposed our signal like so, given this interpretation

x(t) = exp(-j2pi (0) ) + exp(-j2pi f0 (t – T/4) ) + exp(-j2 pi (-f0 (t – T ¾) ) )

= 1 +

And if we do some manipulation, we get to the complex exponential form of sin.

So let’s go ahead and plot this on a graph.

If you were to map these phasors onto the complex plane. We see this. Associating each phasor with its frequency bin value. We can reconstruct the original signal.

Each index is associated with a harmonic frequency. So array y, at index 0, would contain the phasor representing the DC component of the signal. The value of the array y at index 1 is interpreted as the phasor of the fundamental frequency [example].

There’s a special frequency called the Nyquist frequency.

I’m not going to show this, you’ll just have to take my word for it that this is the correct result and interpretation.

## Straight Forward Implementation

Say you were a co-op student working at the local telecom company, and your boss asked you to implement a DFT function for their library, 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.

So this should work well in theory. It has very low code complexity, and it should be very easy debug because there’s very limited logic. You do your unit tests do make sure the errors are within tolerance. However, when your function is integrated into the main program and uploaded onto the board, everything goes down south. It’s not because there was a bug in your code that was miscomputing the outputs, because we just tested it, but it’s because your function was too slow. Indeed, the downside to this implementation would be the excruciating slow runtime.

From the nested for loops, it should be obvious that the runtime is O(n2).

But let’s do a quick runtime analysis anyways. Assume we have a time domain signal, sampled uniformly over N-points, so we’re working with an N-point DFT. The summation sums together N complex terms, the number of complex additions is N-1. Each term is generated by multiplying an element of *x* with a complex factor, this factor is called a twiddle factor and we will discuss more about this terminology later. Let us assume that generating this complex coefficient takes constant time to generate because it can be obtained from a look up table. There are N complex terms within the summation so N complex multiplications are required. We repeat this process a total of N times, because the output *y* has N elements. Therefore a total of N2-N complex additions are required and a total of N2 complex multiplications are required.

Let us remind ourselves that realistically, a complex multiplication actually involves 4 multiplies and 2 additions. And that a complex addition actually requires only two additions. However this really depends on the data type being used to represent our numerical values.

We could probably do better than O(n2), and we will see shortly how this can be achieved.

The other issue is with memory.

If we think about space complexity of the function, that is the maximum amount of space required by the function to do its work, we come to realize that it’s constant because only a couple of local variables are required. Since the arrays are passed in as pointers, there is no reconstruction of the arrays, and therefore space complexity is not affected by the array size.

However, the implication here, is that the main program would then at some point be required to allocate memory for the input array, x and output array, y, either at compile time or at runtime (bad idea). It is also important to mention that these two blocks of memory do not overlap.

The constraint of memory usage would be a case by case basis, depending on the processor application of the program. But let’s say that in this case, memory is indeed a constraint so it would be in everyone’s best interest for the application to save on memory. The requirement implied here is that the solution would need to do away with the output array y, and instead we should simply overwrite elements of the input array x with the final results. Let’s assume that once we apply to Fourier transform, we don’t need the original inputs anymore.

But sadly this straightforward solution is quite possible in this implementation. Our nested loop continuously reads the original values from the array x.

But what if I told you that it was actually possible to avoid the usage of those N memory locations allocated for the output! Indeed, it is actually possible to do the transformation in place! So starting with the input array x, we continuously overwrite its contents with intermediate results, over several stages until we get our final result.

This brings us to our main topic of discussion, fast Fourier transforms, or FFTs.

5:22

## FFTs

As the name implies, the “fast” adjective describes that the transform can be done in some time faster than O(n2). In fact, by using a certain computational framework, the DFT can achieve a run time of O(nlog2(n) ).

Here’s graph to illustrate the difference in runtimes [show graph generated from MATLAB].

Not only tht, but using this implementation, the trnsfomrtion cn lso be done in plce, thereby reducint the memory reuired by the output.

Looking back at our original implementation, it’s not obvious how exactly this runtime optimization would be achieved [show original code implementation]. But just by inspection, there’s relly only one line where all the work happens, so it probably has to do with the complex arithmetic. Indeed, we can take advantage of the periodicity and symmetry of the phasors around the unit circle. This brings us to very importnt prerequisite topic, twiddle factors.

## Twiddle Factors

What is the unit circle? What’s a twiddle factor? We need to establish these terms before moving on.

Consider a unit circle lying on the complex plane, centered at the origin. Each point on the circle is a phasor with a magnitude of 1. Recll tht complex number cn be represented by phsor.

The twiddle factor refers to the complex factor in the DFT equation [show equation]. Twiddle factors lie on the unit circle, and likewise have a magnitude of 1.

So for some N-point DFT we use 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 counter clockwise.

Consider this example with some N-point DFT and the N-th roots of unity.

For example the fourth roots of unity re , j, -, -j…

It’s easier to use a change of notation. Let w­n, ik denote the exp(-j2PI i k /N) [show excellent graph with labels]

If we look at this product i and k, which we can conveniently call the twiddle index, it ranges from 0 to (N-1) x (N-1).

So as we increase the index, we simply jump from one root of Nth root unity to the next, going counter clockwise. When the index reaches N, we’ve conveniently arrived back where we started, since exp(-2Pi j N / N) = 1. And increasing the index above N literally tkes us in circle, thi demonstrates the cyclical/periodic nature of the twiddle factors.

Therefore rather than recomputing certain products, we can just reuse the intermediate results.

For example:

X[0] x wn, 1 is computed once, then at some other point, x[0] w, 1 + N must be computed as well. However, these yield the same result! Therefore it’s completely redundant to recompute this intermediate result, rather we can reuse that first intermediate result.

So in theory, the cyclical nature of the twiddle factors should allow us to reduce the number of complex multiplications. As established earlier, complex multiplications ever so prevalent in the DFT algorithm. Not only would they be expensive (depending on your data type, floating point vs fixed point, and hardware), but they also introduce noise.

Next we examine the symmetrical property of the twiddle factors/ unit circle. Say at some point we compute x[0] wn, 1, then we must also eventually compute x[0] wn, 1 + N/2. Assuming that N is a power of 2, observe that twiddle factors offset by N/2 indices have the following property. They are just the negations of each other. Let’s see an example. This is confirmed mathematically, and visually. The implication here is that we can save even more complex multiplies by negating, or swpping dditions for subtrctions nd vice vers.

Great. Now that we know these mathematical properties of the twiddle factors, let’s move on to the core of the algorithm.

## Cooley-Tukey FFT Algorithm

The most popular computational framework for the FFT is the Cooley-Tukey algorithm. It uses a divide and conquer approach to break down the problem into smaller sub-problems.

## Radix-2 DIT

The radix-2 decimation in time FFT is the simplest and most common form of this divide and conquer algorithm. Don’t worry about the term: decimation in time, we’ll talk about it later.

Recall the mathematical equation that we are trying to implement, without using a nested loop.

In this line, we have the summation of N complex terms. Let’s split this summation into two smaller summations. That’s what radix-2 means, we’re just splitting it into two smaller DFTs. Let’s sum together the even indexed terms (starting at 0). Then, let’s sum together the odd indexed terms (starting at 1). That’s all.

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 E and O, to replace those summations.

Next we can take advantage of the periodicity of the twiddle factor. The twiddle factor is determined by the index k. Recall incrementing the twiddle indices simply takes us around the unit circle, through the roots of unity.

So I hope that we can agree on this fact. So this would imply the following.

So given that E\_k and O\_k have been determined, these two lines here computes two elements of the output y. These element happen to be offset by some index. Essentially, what’s happening here is that we are reusing, an intermediate result O\_k, by taking its negation.

This is called a butterfly operation. You can also think of it as a 2-point FFT…(be careful)

The butterfly operation we looked at, assumed that Ek and Ok were precomputed. Well E\_k and O\_k are just themselves smaller DFTs. So we can continue to break those down into two smaller DFTS each. This is done until the base case, a 2-point DFT.

Ultimately the FFT procedure can be summarized with the signal flow chart.

Talk about how to read the signal flow chart. Make my own, and make it very obvious and easy to read.

Observations to be made, are reordered signals and butterfly operations.

## Radix-4 DIT

Show the formula that breaks it down

Show the signal flow chart.

Observations to be made, are reordered signals and butterfly operations.

Butterflies will be examined later.

## Overview for usage of radix-4

## Reordering

### Index reversal radix-2

Demonstrate with table and examples

Deduce runtimes.

### Index reversal radix-4

Demonstrate with table and examples

Deduce runtimes.

## Complex Arithmetic in Butterfly Operations

### 2-point Butterfly

Show mathematical equation here

### 4-point Butterfly

Show mathematical equation here

## Use of 4th Roots of Unity

## Twiddle Factors

The total number of unique twiddle factors to generate are the same, so that’s a constant.

# Conclusions

When to use radix-4 over radix-2?

Why not just use radix-16 or radix-8?

## Limitations

Recall the other parameters to consider when selecting algorithms.

# Abstract

Provide a TLDR of this presentation.

## Usage in embedded systems and DSP

Why do we care about FFTs in embedded systems and digital signal processing? Of course these are very advanced subjects that we might learn later on but to put it simply.

It’s a lot easier to apply a filter or transfer function once a signal has been mapped to the frequency domain. Demonstrate this with a low pass filter example.

We need it to be fast because the difference in runtime between n square and nlogn gets pretty drastic. Show figure comparing runtimes. Since these systems are real time, we are very concerned with runtime.

Design and algorithm decisions of FFTs. There are a whole lot of different FFT implementations, it’s not a trivial or straightforward task to decide which one to pick because there are many tradeoff parameters

Runtime

Space complexity

Accuracy

Function memory (when the function is compiled into instructions)

Normally done with fixed-point arithmetic rather than floating point or double.

If done with fixed-point arithmetic then “downscaling” is an issue.

**Sources**

[**http://paulbourke.net/miscellaneous/dft/**](http://paulbourke.net/miscellaneous/dft/)

[**https://www.reddit.com/r/math/comments/1wx9wg/nice\_visualization\_of\_the\_fourier\_transform/**](https://www.reddit.com/r/math/comments/1wx9wg/nice_visualization_of_the_fourier_transform/)

**Frequency bin interpretation**

**https://www.mathworks.com/matlabcentral/answers/318595-frequency-bin-of-the-positive-and-negative-frequency**

Nyquist's theorem says that as long as the sampling rate is greater than twice the highest frequency component of the signal, the sampled data will accurately represent the input signal

<http://www.thinksrs.com/downloads/PDFs/ApplicationNotes/AboutFFTs.pdf>

**Explains DFTs very nicely**

**http://complextoreal.com/wp-content/uploads/2013/04/fft5.pdf**