Fast Fourier Transform

Adam Lail Jacob Hellenbrand

May 7, 2024

Abstract

In this report we will take an indepth view of the Fast Fourier Transform. The FFT is a numerically efficient method to calculate Discrete Fourier Transform. These methods are commonly used in signal and frequency domain processing. We will work through a simple example of how FFT can be used. Furthermore we will explore the formal uses and ensuring a deep examination of its correctness and accuracy.

1 Introduction

1.1 Overview

Called one of the most important algorithms of the 20th century, the Fast Fourier Transform algorithm (from here on FFT) became well established by J. Cooley and J. Tukey in 1965 with their ingenious divide and conquer algorithm. FFT is a fast solution to finding the Discrete Fourier Transform (DFT) as opposed to computing it directly.

A DFT is a Fourier Transform on a finite set of data and boils down to taking any quantity or signal that varies over time and decomposing into its components. Fourier transform is generally described as a mathematical tool that allows us to understand the frequency content of a signal. It takes a function of time (or space) and expresses it in terms of the frequencies of the waves that make it up. Or put differently, it can be used to transform a time-domain signal into its frequency-domain representation. FFT shows up in many fields where situations this described situation shows up, such as digital signal processing, training models used in convolutional neural networks, solving partial differential equations, and multiplying large polynomials.

2 A Worked Example

Let's consider a simple example of FFT, using a polynomial. Consider the polynomial $A(x)$,

$$A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$$

From here we will split $A(x)$ into two parts, one being the even coefficients the other the odd,

$$A(x)_{even} = a_0 + a_2x + a_4x^2 + a_6x^4 \text{ and } A(x)_{odd} = a_1 + a_3x + a_5x^2 + a_7x^4$$

And we see that these can be rewritten as,

$$A(x) = A(x^2)_{even} + xA(x^2)_{odd}$$

From here we now want to evaluate $A(x)$ at the n complex nth roots of unity, them being: $w_n^0, w_n^1, \ ... \ , w_n^{n-1}$ where $w_n = e^{2\pi i/n}$. We then can observe two conditions,

$$\text{if } k < \frac{n}{2} \ , \ A(w_n^k) = A_{even}(w_n^{2k}) + w_n^k A_{odd}(w_n^{2k})$$
$$\text{if } k \geq \frac{n}{2} \ , \ A(w_n^k) = A_{even}(w_n^{2k}) - w_n^k A_{odd}(w_n^{2k})$$

Now we can evaluate A_{even} and A_{odd} twice while only calculating for them once!

3 A Formal Problem Specification and Pseudocode

FFT is an application of the design and conquer strategy. This method is a numerically efficient way of computing DFT. The idea behind it is splitting the input into even and odd halves then recursively solving. It takes advantage of the symmetrical properties of DFT and complex roots of unity to efficiently solve.

The recursion continues until we reach our base case where the results allow for DFT to be computed directly. We then can take the answers from these sub problems and combine them to obtain the final DFT results.

Below we can examine Algorithm 1 that accomplishes this.

Algorithm 1 Fast Fourier Transform

```
function FFT(f)
    n = len(f)
    if n == 1 then
        return f
    end if
    W_n = e^(-2πi/n)
    W = 1
    f_even = f[0 :: 2]
    f_odd = f[1 :: 2]
    F_even = FFT(f_even)
    F_odd = FFT(f_odd)
    for k ← 0 to (n/2 − 1) do
        F[k] = F_even[k] + W * F_odd[k]
        F[k + n/2] = F_even[k] − W * F_odd[k]
        W = W_n
    end for
    return F
end function
```

Note: If the number of elements in our input are not a power of 2, we will pad the input and add 0's until we have a power of 2. Padding the input brings additional computational needs, but ensuring our input is a power of 2 guarantees the efficiency of our algorithm.

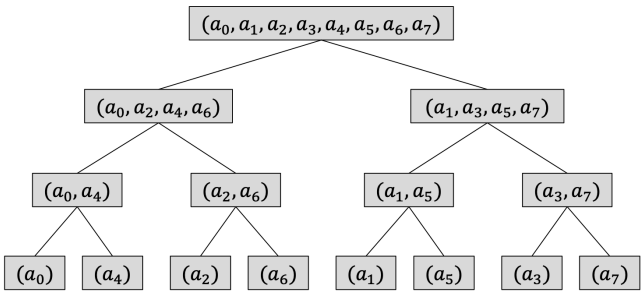

Figure 1: Example of how FFT divides.

4 A Runtime Analysis

Looking at our FFT alogirthm we see a rather similar recurrance relation to that of MergeSort. Thus, Let $T(n)$ be the number of assignments to the array A in FFT for an input of length n. There are 2 recusive calls of size $\frac{n}{2}$ as well as n to be able to compute with DFT and we see,

$$T(n) = 2T(n/2) + O(n)$$

Which is bounded by $\theta(n \log(n))$ by the Master Theorem. Therefore we know the algorithm runs in $n \log(n)$ time.

5 An Argument for Correctness

An argument for correctness must begin with the fact that this is an extremely researched and rigorously validated algorithm for computing the DFT. A formally rigorous proof would be far outside the breadth of our personal knowledge and the math required for this course. The idea behind an argument for correctness would come from an inductive proof. We would establish a base case of an input sequence with just one or two elements. From there we would assume the algorithm correctly computes the DFT for input sizes of $N = 2^k$, where k is a non-negative integer. From there we would show that it also works for input sequences of size $N = 2^k + 1$. To do so we would need to possibly use a loop invariant and prove that the recursive nature of the algorithm works correctly. Fundamentally, the algorithm's mathematical foundations, algorithmic design, empirical validation, and practical impact collectively attest to its correctness and reliability.

6 Implementation

Here is the link to our githhub repo: `https://github.com/jacobhellenbrand/Lail-Hellenbrand-Comp221-Final`

References

[1] Steve Brunton. The fast fourier transform algorithm, 2020. YouTube video.

[2] Paul Heckbert. Fourier transforms and the fast fourier transform (fft) algorithm, 1998.

[3] Stephen Roberts. Lecture 7 - the discrete fourier transform, 2012.

[4] T. Sidhant. Tutorial on fft/ntt - the tough made simple. (part 1), 2016.

[5] Aashirwad Viswanathan Anand. A brief study of discrete and fast fourier transforms, 2015.

[6] Wikipedia contributors. Cooley–tukey fft algorithm — Wikipedia, the free encyclopedia, 2024. [Online; accessed 4-May-2024].