Fast Algorithms for computing the Discrete Fourier transform

Rajarshi Basak

A report presented for the course
Design and Analysis of Algorithms (CSCI - 5454)
Fall 2016

Department of Computer Science
University of Colorado Boulder
United States of America

9.12.2016

Contents

Chapter 1

Introduction

1.1 Fourier Transforms

1.1.1 The Fourier Transform

The Fourier transform fragments a signal (a function of time) into the frequencies that constitute it. It is essentially a complex-valued function of frequency, whose absolute value represents the amount of that frequency present in the original function, while the complex argument is the phase offset of the basic sinusoid in that frequency. Even though the original signal (function) is usually a function of time, and its Fourier transform is called the frequency domain representation of the signal, the Fourier transform is not limited to functions of time; to have a unified language, the domain of the original function is customarily referred to as the time domain. [1]

1.1.2 The Discrete Fourier Transform

The Discrete Fourier Transform converts a finite sequence of equally spaced samples of a signal (function of time) into an equivalent length sequence of equally spaced samples of the discrete-time Fourier transform (DTFT), which is a complex-valued function of frequency, and the interval of sampling the DTFT is the reciprocal of the time period of the input signal. [2]

The inverse DFT is a Fourier series, which uses the DTFT samples as coefficients of complex sinusoids at the corresponding DTFT frequencies. Since it has the same sample-values as the original input sequence, the DFT is said to be a frequency domain representation of the original input signal or sequence. [2]

As a DFT deals with a large (but finite) amount of data, certain computer algorithms can implement it, which generally employ the fast Fourier transform. [2]

1.1.3 The Fast Fourier Transform

A Fast Fourier Transform (FFT) computes the discrete Fourier transform(DFT) of a signal or sequence, or its inverse. An FFT swiftly computes these Fourier transformations by factorizing the DFT matrix into a product of sparse factors. This helps it to reduce the running time of computing the DFT from *O(n*^{2}*)* to *O(nlogn)*, where n is the size of the input signal. [3]

1.1.4 FFT vs. DFT

We have seen earlier that the DFT is obtained by fragmenting a sequence of values into components of different frequencies. Computing the DFT from definition, however, can be too slow for practical purposes. An FFT can expedite this process. The complexity improvement of *O(nlogn)* of an FFT over the *O(n*^{2}*)* of the naive DFT computation can lead to a huge difference in speed, especially when the size of the data sets are in thousands or millions. [3]

1.2 Algorithms for the FFT

The most popular FFT algorithms rely on the factorization of N (size of dataset). However, there exists FFTs with *O(nlogn)* complexity for all N including prime N. Several FFT algorithms are hinged on the fact that $e^{-2\pi i/N}$ is an Nth primitive root of unity, and can hence be applied to comparable transforms over any finite field, *e.g.* number-theoretic transforms. [3]

Cooley and Tukey, whose work was published in 1965, are predominantly credited for the invention of the modern generic FFT algorithm. [3]

Chapter 2

The Cooley-Tukey FFT algorithm

Named after James William Cooley and John Tukey, the Cooley-Tukey algorithm is the most common fast Fourier transform (FFT) algorithm. Since it re-expresses the discrete Fourier transform (DFT) of an arbitrary composite size $N = N_1 N_2$ in terms of N_1 smaller DFTs of sizes N_2 recursively, it reduces the computation time to *O(NlogN)* for highly composite *N*.

2.1 History

The algorithm was inventede by Carl Friedrich Gauss who employed it to interpolate the trajectories of the asteroids Pallas and Juno. However, Gauss did not analyze the asymptotic computational time, and his work was not published. [–tukey$_f ft_a lgorithm$]

The idea occurred to John Tukey of Princeton during a meeting of President Kennedy's Science Advisory Committee when he was discussing ways to select nuclear-weapon tests in the Soviet Union by using seismometers (to generate seismological time series) located outside the country. The analysis of this data required fast algorithms for computing DFT owing to number of sensors and length of time and this task was pivotal for the ratification of the proposed nuclear test ban. This allowed any violations to be detected without the need to visit Soviet facilities. [–tukey$_f ft_a lgorithm$]

Richard Garwin of IBM, who was another participant at the meeting, recognized the prospects of the method and connected Tukey with Cooley, but he made sure Cooley was unaware of the original purpose. Cooley was told that the motivation for the algorithm wast to determine periodicities of the spin orientations in a 3-D crystal of Helium- 3. In 1965, Cooley and Tukey published their joint paper, and a wide adoption followed owing to the simultaneous development of Analog-to-Digital Converters which were able to sample rates up to 300 kHz. [–tukey$_f ft_a lgorithm$]

Even though Gauss had described the same algorithm, the fact was realized several years after Cooley and Tukey's 1965 paper, which cited as inspiration I.J. Good's work now known as the Prime-factor algorithm (PFA). Initially Good's algorithm was thought to be equivalent to the Cooley-Tukey algorithm; soon people realized that the PFA was radically different. It only worked for sizes having relatively prime factors and depending on the Chinese Remainder Theorem. [–tukey$_{f}ft_{a}lgorithm$]

2.2 The radix-2 DIT case

The most elementary and commonplace form of the Cooley-Tukey algorithm is a radix-2 decimation-in-time (DIT) FFT, which divides a DFT of size N into two interleaved DFTs of size $N/2$ with each recursive stage. [–tukey$_{f}ft_{a}lgorithm$]

The formula for the discrete Fourier transform is:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk} \tag{2.1}$$

where k is an integer ranging from 0 to $N-1$. [–tukey$_{f}ft_{a}lgorithm$]

The radix-2 DIT first computes the DFTs of the even-indexed inputs ($x_{2m} = x_0, x_2, ..., x_{N-2}$) and of the odd-indexed inputs ($x_{2m+1} = x_1, x_3, ..., x_{N-1}$) following which it merges these two results to generate the DFT of the whole sequence. This process is then executed recursively to decrease the overall runtime to O($NlogN$). It is to be noted here that this simplified form assumes that N is a power of (i.e. $N = 2^k$ where $k = 0, 1, 2, ...$). [–tukey$_{f}ft_{a}lgorithm$]

The algorithm rearranges the DFT of the function into two parts: a sum over the even-numbered indices $n = 2m$ and a sum over the odd-numbered indices $n = 2m+1$: [–tukey$_{f}ft_{a}lgorithm$]

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N}(2m)k} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N}(2m+1)k} \tag{2.2}$$

Factoring out a common multiplier $e^{-\frac{2\pi i}{N}}$ from the second sum, as shown below, it is observed the two sums are the DFT of the even-indexed part x_{2m} and the DFT of the odd-indexed part x_{2m+1} of the function x_n. If we use E_k to denote the DFT of the even-indexed inputs x_{2m} and O_k to denote the DFT of the odd-indexed inputs x_{2m+1}, we have [–tukey$_{f}ft_{a}lgorithm$]

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N/2}mk} + e^{-\frac{2\pi i}{N}} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N/2}mk} = E_k + e^{-\frac{2\pi i}{N}} O_k \tag{2.3}$$

Using the periodicity of the DFT, we have $E_{k+\frac{N}{2}} = E_k$ and $O_{k+\frac{N}{2}} = O_k$ and the equation above can be rewritten as [–tukey$_{f}ft_{a}lgorithm$]

$$X_k = E_k + e^{-\frac{2\pi i}{N}} O_k for 0 \leq k \leq N/2 \tag{2.4}$$

$$X_k = E_{k-N/2} + e^{-\frac{2\pi i}{N}} O_{k-N/2} for N/2 \leq k \leq N \tag{2.5}$$

Since the twiddle factor $e^{-\frac{2\pi ik}{N}}$ obeys the following relations: [–tukey$_f ft_a lgorithm$]

$$e^{-\frac{2\pi i}{N}(k+N/2)} = e^{-\frac{2\pi ik}{N} - \pi i} = e^{-\pi i} e^{\frac{-2\pi ik}{N}} = -e^{-\frac{-2\pi ik}{N}} \tag{2.6}$$

,

the number of twiddle factor calculations are cut in half. For $0 \leq k < N/2$,

$$X_k = E_k + e^{-\frac{2\pi i}{N}k} O_k \tag{2.7}$$

$$X_{k+N/2} = E_k - e^{-\frac{2\pi i}{N}k} O_k \tag{2.8}$$

The above results, which expresses the DFT of length N recursively in terms of two DFTs of size $N/2$ is the crux of the radix-2 DIT fast Fourier transform and the algorithm gains its speed by re-using the results of intermediate computations to compute multiple DFT outputs. To attain the final outputs, a $+/-$ combination of E_k and $O_k e^{-\frac{2\pi ik}{N}}$ are taken, which is a size-2 DFT (called a butterfly). This size-2 DFT is replaced by a larger DFT when generalized to larger radices. The process mentioned above falls in the category of divide-and-conquer algorithms. However, in many conventional implementations, the computational tree is traversed in breadth-first fashion instead of doing recursion. [–tukey$_f ft_a lgorithm$]

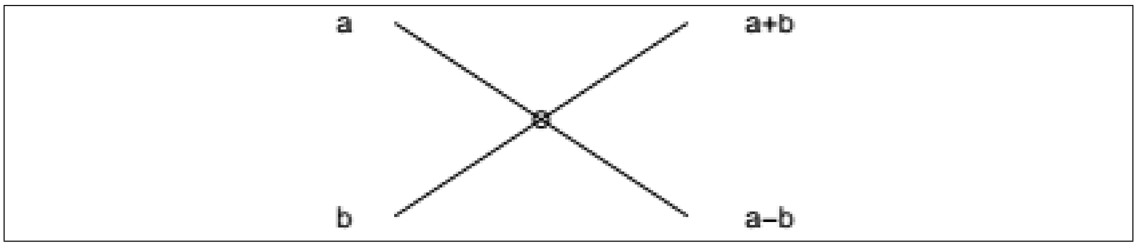

Figure 2.1: The butterfly: the flowgraph for a sum and difference operation [4]

Since the above identity was noted by G.C. Danielson and Cornelius Lanczos, the above re-espression of a size-N DFT as two size-$N/2$ DFTs is referred to as the Danielson-Lanczos lemma. Applying their lemma in a reverse fashion, they repeatedly doubled the DFT size until the transform spectrum converged, even though they failed to notice the linearithmic (i.e. order $NlogN$) asymptotic complexity they had achieved. [–tukey$_f ft_a lgorithm$]

2.3 General factorizations and the radix-2 DIF

In general, Cooley-Tukey algorithms re-espress a DFT of a composite size $N = N_1 N_2$ recursively by performing the following steps:

(a) Calculate N_1 DFTs of size N_2 (b) Compute the twiddle factors (complex roots of unity) and multiply by them (c) Calculate N_2 DFTs of size N_1 [–tukey$_f ft_a lgorithm$]

Usually, either N_1 or N_2 is a small factor called the radix and it can differ at different stages of the recursion. The algorithm is called a decimation-in-time algorithm if N_1 is the radix; if N_2 is the radix instead, it is called a decimation in frequency algorithm (also called the Sande-Tukey algorithm). [–tukey$_f ft_a lgorithm$]

In the version of the algorithm presented above (the radix-2 DIT algorithm)the phase multiplying the odd transform in the final expression is the twiddle-factor, and the $+/-$ combination (butterfly) of the even and odd transforms is a size-2 DFT (Owing to the shape of the dataflow diagram for the radix-2 case, the radix's small DFT is called a butterfly). [–tukey$_f ft_a lgorithm$]

2.3.1 Other variations

A mixed-radix implementation handles composite sizes with a range of factors, in which it typically employs the $O(N^2)$ algorithm for the prime base cases of the recursion. Bluestein's or Rader's algorithm can employ an $O(NlogN)$ algorithm for the prime base cases. [–tukey$_f ft_a lgorithm$]

Harnessing the fact that he first transform of radix-2 requires no twiddle factor, the split radix method combines radices 2 and 4 to attain the lowest known arithmetic operation count for power-of-two sizes (although recent variations have managed to achieve an even lower count). Since performance is determined more by cache and CPU pipeline considerations compared to strict operation counts on present day computers, well optimized FFT implementations often often use larger radices and/or hard-coded base-case transforms of significant size. [–tukey$_f ft_a lgorithm$]

The Cooley-Tukey algorithm can also be seen as an algorithm that re-expresses a size N one-dimensional DFT as an N_1 by N_2 two-dimensional DFT and twiddles, where the output matrix is transposed. As a result of these transpositions, for a radix-2 algorithm, a bit reversal of the input (DIF case) or the output (DIT case) indices. If we employ a radix of approximately $\sqrt{N}$ and explicit input/output transpositions instead of using a smaller radix, it is called a four-step or six-step algorithm (depending on the number of transpositions). Whilst this was initially proposed to enhance memory locality, it was later shown to be an optimal cache-oblivious algorithm. [–tukey$_f ft_a lgorithm$]

2.3.2 The general Cooley-Tukey factorization

The general Cooley-Tukey factorization rewrites the indices k and n as $k = N_2k_1 + k_2$ and $n = N_1n_2 + n_1$, respectively, with k_a and n_a running from $0...N_a - 1$ (for a of 1 or 2). In other words, it re-indexes the input (n) and output (k) as N_1 by N_2 two-dimensional arrays in column-major and row-major order, respectively; the difference between these indexings being a transposition. on substitution of this re-indexing into the DFT formula for nk, since the exponential of the $N - 1n_2N - 2k_1$ cross term is unity, the cross term vanishes, and the remaining terms yield [–tukey$_f ft_a lgorithm$]

$$X_{N_2k_1+k_2} = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x_{N_1n_2+n_1} e^{-\frac{2\pi i}{N_1N_2}(N_1n_2+n_1)(N_2k_1+k_2)} \tag{2.9}$$

$$= \sum_{n_1=0}^{N_1-1} \left[e^{-\frac{2\pi i}{N} n_1 k_2} \right] \left(\sum_{n_2=0}^{N_2-1} x_{N_1n_2+n_1} e^{-\frac{2\pi i}{N_2} n_2 k_2} \right) e^{-\frac{2\pi i}{N_1} n_1 k_1} \tag{2.10}$$

where each inner sum is a DFT of size N_2, each outer sum a DFT of size N_1 and the bracketed term ([...]) is the twiddle factor. [–tukey$_f ft_a lgorithm$]

2.4 Bit reversal algorithms

Bit reversal is an arrangement in which the data at index n, written in binary with digits $b_2b_1b_0$ (e.g. 3 digits for $N = 8$ inputs), is transferred to the index with reversed digits $b_0b_1b_2$. In the last stage of a radix-2 DIT algorithm, the output is written in place over the input; when E_k and O_k are combined with a size-2 DFT, these two values are overwritten by the inputs. The output values should instead go in the first and second halves of the output array, corresponding to the most significant bit b_2 (for $N = 8$); while the two inputs E_k and O_k are interleaved in the even and odd elements, corresponding to the least significant bit b_0. Therefore, in order to get the output in the correct place, b_0 should take the place of b_2 and the index becomesv $b_0b_1b_2$. For the next recursive stage, those 2 least significant bits should become b_1b_2. If one includes all of the recursive stages of the radix-2 DIT algorithm, all the bits must be reversed and thus one must pre-process the input (or post-process the output) with a bit-reversal to get in-order output. If each size-$N/2$ subtransform is to operate on adjacent data, the DIT input is pre-processed by bit-reversal. If one performs all of the above-mentioned steps in reverse order, on obtains a radix-2 DIF algorithm with bit-reversal in post-processing, the details of which has been discussed in the next chapter. [–tukey$_f ft_a lgorithm$]

Chapter 3

The radix-2 Decimation in frequency (DIF) algorithm

The DFT of an N-point signal [$x[n], 0 \leq n \leq N-1$] is given by [4]

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{-kn}, 0 \leq k \leq N-1 \tag{3.1}$$

where

$$W_n = e^{j\frac{2\pi}{N}} = cos\left(\frac{2\pi}{N}\right) + jsin\left(\frac{2\pi}{N}\right) \tag{3.2}$$

is the principal N-th root of unity.

Compared to the direct computation of X_k for $0 \leq k \leq N-1$ which requires $(N-1)^2$ complex multiplications and $N(N-1)$ complex additions, the radix-2 FFT algorithms (used for data sets of lengths $N = 2^k$) proceed by dividing the DFT into two DFTs of length $N/2$ each, and iterating. [4]

To develop the DIF FFT, we employ the following properties, [4]

$$W_N^2 = W_{N/2} \tag{3.3}$$

$$W_N^{k+\frac{N}{2}} = -W_N^k \tag{3.4}$$

3.1 The Derivation

For an N-point signal $x[n]$ of even length, first we split the DFT coefficients X_k into even and odd indexed values. The even values X_{2k} are: [4]

$$X_{2k} = \sum_{n=0}^{N-1} x_n W_N^{-2kn} \tag{3.5}$$

$$= \sum_{n=0}^{N-1} x_n W_{N/2}^{-kn} \tag{3.6}$$

Splitting the above sum into the first $N/2$ and second $N/2$ terms, we have [4]

$$X_{2k} = \sum_{n=0}^{N/2-1} x_n W_{N/2}^{-kn} + \sum_{n=N/2}^{N-1} x_n W_{N/2}^{-kn} \tag{3.7}$$

$$= \sum_{n=0}^{N/2-1} x_n W_{N/2}^{-kn} + \sum_{n=N/2}^{N-1} x_{n+\frac{N}{2}} W_{N/2}^{-k(n+\frac{N}{2})} \tag{3.8}$$

$$= \sum_{n=0}^{N/2-1} x_n W_{N/2}^{-kn} + \sum_{n=N/2}^{N-1} x_{n+\frac{N}{2}} W_{N/2}^{-kn} \tag{3.9}$$

$$== \sum_{n=0}^{N/2-1} \left(x_n + x_{n+\frac{N}{2}} \right) W_{N/2}^{-kn} \tag{3.10}$$

$$= DFT_{\frac{N}{2}} \left\{ x_n + x_{n+\frac{N}{2}} \right\} \tag{3.11}$$

This implies that the even DFT values X_{2k} for $0 \leq 2k \leq N-1$ are given by the $\frac{N}{2}$-point DFT of the $\frac{N}{2}$-point signal $x_n + x_{n+N/2}$. [4]

A similar analysis for the odd values X_{2k+1} yields

$$X_{2k+1} = \sum_{n=0}^{N-1} x_n W_N^{-n} W_{N/2}^{-kn} \tag{3.12}$$

and splitting this sum into the first $N/2$ and second $N/2$ terms, we get

$$X_{2k+1} = DFT_{\frac{N}{2}} \left\{ W_N^{-n} \left(x_n - x_{n=\frac{N}{2}} \right) \right\} \tag{3.13}$$

or the odd DFT values X_{2k+1} for $0 \leq 2k+1 \leq N-1$ are given by the $\frac{N}{2}$-point DFT signal $W_N^{-n}(x_n - x_{n+N/2})$. [4]

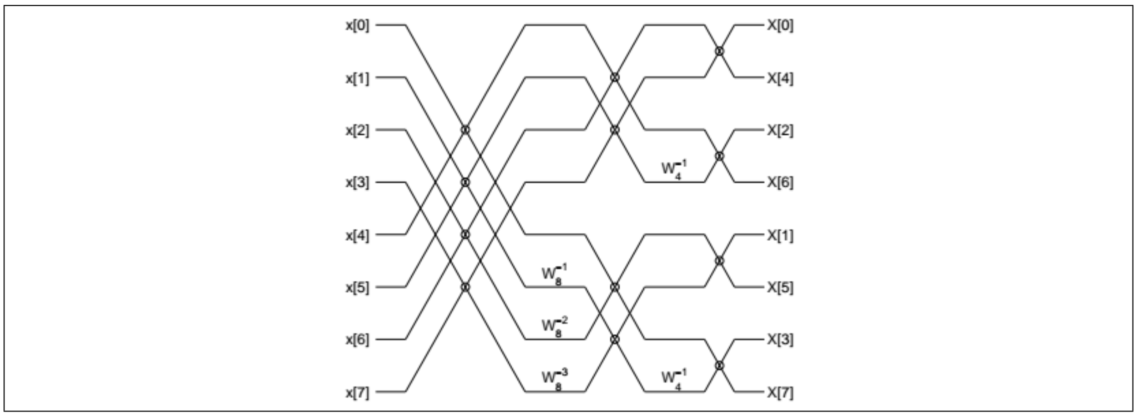

Figure 3.1: The full flowgraph for a radix-2 8-point DIF FFT [4]

3.2 The Implementation

3.2.1 Correctness

The class FFT has two methods: a void method named 'fourier', and a method padWithZeros which returns a String.

The method fourier takes as input a double array named 'signal' which contains the input data. Integer 'l' is assigned the value of the length of the input array, or the number of data points in the array, which should be a power of 2. Next we define a new array 'signalcomp' (by calling the Complex class) which stores the input signal as it's real coefficients, and the imaginary coefficients are initially set to 0. We also define an array 'output' (from the Complex class), initialized to zero coefficients (real and imaginary parts) to store the transformed signal. Integer 'st' is assigned the value of the number of stages, which equals $log_2N = log_2l$

The first for loop iterates over the number of stages, backwards; i.e. from $st = log_2l$ to $st = 1$, decreasing the index i by 1 at each step. We assign the integer variable 'xs' the number of butterflies to be calculated at that stage, given by $2^{stagenumber-1}$ and initialize $j = 0$. In the nested while loop, which runs as long as j does not exceed the input signal length (here j runs from 0 to $l - 1$), we nest another for loop which runs from $k = 0$ to $k = xs - 1$ (i.e. 'k' iterates over the number of butterflies until j equals the signal length); inside this we update the jth entry of the 'output' array to the sum of the jth entry and the $j + xs$th entry of the input array ('signal comp'). The $j + xs$th entry of the 'output' array is updated to the difference between jth entry and the $j + xs$th entry of the input array times the twiddle factor, which is calculated and stored in the Complex (class) variable c. After these additions and multiplications (for each butterfly in a particular stage), j is incremented inside the k index for loop, and we repeat the above process for the next value of k. When the value of k matches the number of butterflies $= xs$, j is updated to $j = xs$, and we repeat the process for the next value of j. Once j equals the input signal length l, i.e. all the calculations are done for the first stage and they have been stored in 'output' array, we overwrite the 'signalcomp' array the values of the 'output' array, as in the next stage, the input values are the output values from the previous stage.

Once i has gone through all the iterations of the initial for loop and exited it, the 'output' array, which stores the fourier transform of the input array 'signalcomp' is printed in bit-reversed order.

The method padWithZeros pads the binary representation of the 's'th index with the requisite number of zeros from the left, and returns the String to the method fourier, so that the number of digits after padding matches the number of digits for the binary representation of the largest value of 's'.

The class Complex (borrowed from http://introcs.cs.princeton.edu/java/97data/Complex.java.html), helps deal with complex numbers. We use it here to store the real and imaginary parts of the input array 'signal', and for the output array.

3.2.2 Complexity Analysis

To evaluate the arithmetic complexity of this algorithm, we let $A_c(N)$ and $M_c(N)$ denote the number of complex additions and multiplications respectively for computing the DFT of an N -point complex sequence x_n. Since $N = 2^k$, we have

$$A_c(N) = 2A_c(N/2) + N \tag{3.14}$$

$$M_c(N) = 2M_c(N/2) + \frac{N}{2} + 1 \tag{3.15}$$

as N complex additions and $\frac{N}{2} - 1$ complex multiplications are required to put the two $N/2$-point DFTs together. As a 2-point DFT is simply a sum and a difference,

$$X_0 = x_0 + x_1 \tag{3.16}$$

$$X_1 = x_0 - x_1 \tag{3.17}$$

the initial conditions are $A_c(2) = 2$ and $M_c(2) = 0$. Solving the recursion equation, we get $A_c(N) = Nlog_2N$ complex additions and $M_c(N) = \frac{N}{2}log_2N - N + 1$ complex multiplications. [4]

As a single complex multiplication can be performed with 4 real multiplications and 2 real additions and a single complex addition can be performed with 2 real additions, we have [4]

$$M_r(N) = 4M_c(N) \tag{3.18}$$

$$A_r(N) = 2M_c(N) + 2A_c(N) \tag{3.19}$$

which yields

$$M_r(N) = 2Nlog_2N - 4N + 4 \tag{3.20}$$

$$A_r(N) = 3Nlog_N - 2N + 2 \tag{3.21}$$

Since the number of stages of the algorithm is log_2N and each stage has a complexity $O(N)$, the overall complexity is $O(Nlog_2N)$. [4]

3.2.3 The output

For an input signal (1.0, 2.0, 3.0, 4.0), which is a 4-point DFT, the output generated by the code is

```
10.0
-1.9999926535897934 + 1.9999999999865075i
-2.0
-2.000007346410207 - 1.9999999999865075i
```

and for an input signal (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0), an 8-point DFT, the outout generated by the code is

```
36.0
-3.9999645284318515 + 9.656843860024743i
-3.999985307179587 + 3.999999999973015i
-3.9999731351722936 + 1.6568646387501231i
-4.0
-4.000006085927322 - 1.656843860078713i
-4.000014692820414 - 3.999999999973015i
-4.000056250468534 - 9.656864638696153i
```

3.3 Simulations and Numerical Characterization

3.3.1 The randomized input generator

For numerical characterization, we implemented a randomized input generator, which given the size of the input array/signal, generates a random signal (with entries within a range). We then run the actual FFT method (which we call 'fourier' here) on this signal and measure the time taken to do so. By repeating this process several times, (by generating random signals a number of times, say 50), we compute the total time taken for several iterations of the FFT method. Finally we take the mean time for all these iterations to get the average time taken the fourier method to compute an FFT of a signal for an input of a given size.

3.3.2 Implementation and correctness of the random input generator

The class FFTAnalysis has two methods: the method fourier which returns a long variable, and the void method randominputgenerator.

The method fourieranalysis is almost identical to the 'fourier' method from the FFT class (explained in the previous section). We have removed the print statements (since they add to execution time) and the bit reversal portion (which is not part of the actual algorithm for computing the FFT), and introduced three new variables: long 'startTime', long 'endTime' and long 'totalTime'. These variables store the time when the procedure starts, ends, and the difference between the start and the end respectively; the difference being the time taken by the procedure, which we are interested in.

The void method randominputgenerator takes as input the size of the input signal to be generated (given the integer 'length'), and the number of such signals (given by the integer 'times'). We define a new long variable 'grandtime' to 0 to store the time taken for the iterations of the 'fourieranalysis' method. Next we initiate an instance of the Java class Random, and create a new variable 'rand'. We also create a new double array 'input', to store the randomly generated signals.

The first for loop iterates over the number of signals we need to generate (from i = 0 to i = times - 1), and the nested for loop creates each of these signals (j runs from j = 0 to j = length - 1) by calling the Java command nextInt from the Random class. It is to be noted here that the entries generated by nextInt are between 1 and 1000. Once each input signal has been created randomly, we run the FFT algorithm 'fourieranalysis' on it and cumulatively store the time taken to do so in the variable 'grandtime' (the method 'fourieranalysis' takes the input signal, computes its DFT, and returns the time taken). Finally we print the total time taken for all the runs of the 'fourieranalysis' method, and the average time taken for one run.

3.3.3 Simulations

For input sizes ranging from 2^7, which is 128 entries, to 2^{20}, which is 1048576 entries, we generated entry values ranging from 1 to 1000, and ran the FFT algorithm 50 times (on 50 different randomly generated signals) for a particular size. The results of the simulations have been tabulated at the end of this section.

Here $T(N)$ represents the running time per run of the FFT algorithm ('fourieranalysis' method). Both the total running time and $T(N)$ are in milliseconds. Since the running time per run of the FFT was negligible for input sizes smaller than 128 entries (2^7), we considered sizes above that.

3.3.4 Plots

The plots shown below were generated for input data sizes varying from 1024 to 524288 entries, the entry values ranging from 1 to 1000, chosen randomly. Figure 3.3 is a zoomed-in

(k)	Input signal size ($N = 2^k$)	Total running time (ms)	$T(N)$ (ms)
7	128	9	0.18
8	256	17	0.34
9	512	42	0.84
10	1024	86	1.72
11	2048	147	2.94
12	4096	287	5.74
13	8192	621	12.42
14	16384	1384	27.68
15	32768	2909	58.18
16	65536	6684	133.68
17	131072	14220	284.4
18	262144	30573	611.46
19	524288	66996	1339.92
20	1048576	148158	2963.16

Table 3.1: Table 1: Data for random runs of the FFT algorithm

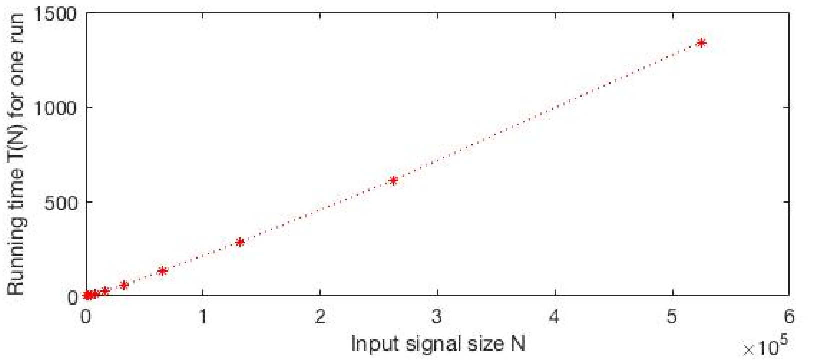

Figure 3.2: Plot of T(N) vs. N for input signal sizes from 1024 to 524288, randomized entries between 1 and 1000, averaged over 50 runs of the FFT algorithm

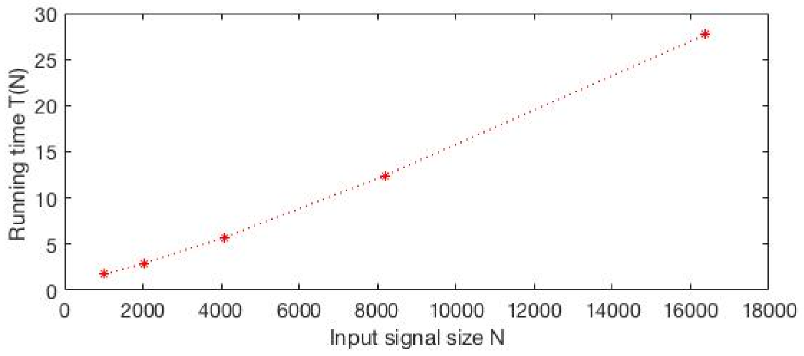

Figure 3.3: Plot of T(N) vs. N for input signal sizes from 1024 to 32768, randomized entries between 1 and 1000, averaged over 50 runs of the FFT algorithm

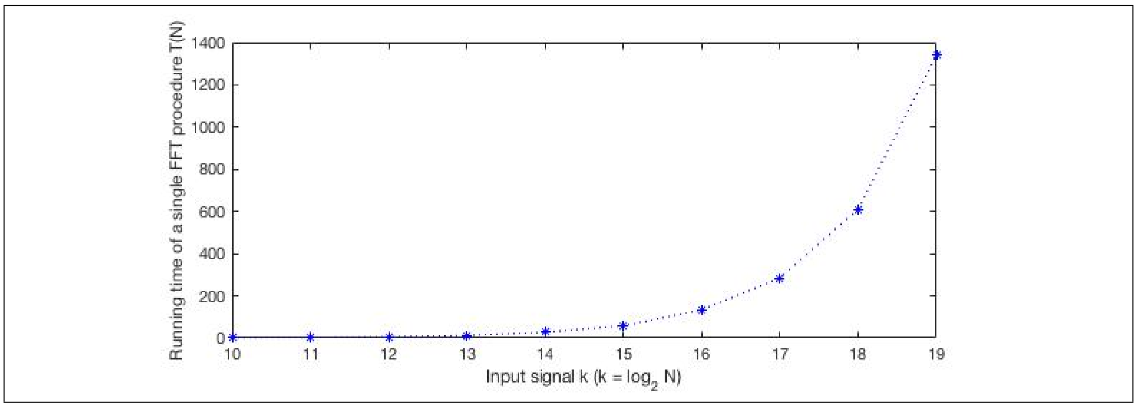

Figure 3.4: Plot of T(N) vs. input signal k value ($log_2 N$) for input signal k's from 10 to 524288, randomized entries between 1 and 1000, averaged over 50 runs of the FFT algorithm

view of Figure 3.2

3.4 Conclusions from the simulations

It is evident from the plots that our algorithm behaves better than $O(N^2)$ behaviour of the straightforward calculation time of the DFT. The complexity is slightly worse than linear; to be precise, it is $O(NlogN)$. This is clear from the table above, where as the input size grows by a factor of exactly 2, the running time grows by a factor of slightly more than 2.

3.5 Comparison with other built-in implementations

In this section, we compare our simulation results to a built-in implementation, namely the Matlab fft computer.

We used the following commands to built random arrays ('r') of length specified by 'length' whose values ranged from 1 to 1000.

```
a = 1;
b = 1000;
r = (b-a).*rand(length,1) + a;
```

Then we used the 'fft' command in Matlab to compute the FFT, and sandwiched it between 'tic' and 'toc' commands as shown below to record the time taken.

```
tic
y = fft(r);
toc
```

(k)	Input signal size ($N = 2^k$)	Running time per run of 'fft' in Matlab: $T(N)$ (ms)
10	1024	13.201
11	2048	13.354
12	4096	13.5075
13	8192	13.6395
14	16384	13.6274
15	32768	14.684
16	65536	16.7206
17	131072	19.1572
18	262144	23.3749
19	524288	32.349
20	1048576	35.839

Table 3.2: Table 1: Data for random runs of the built-in fft implementation in Matlab

For each input array size, we generated 10 different arrays and ran the fft on them; and computed the average time taken for a given size.

The results of the simulations have been tabulated above, and the plots shown below.

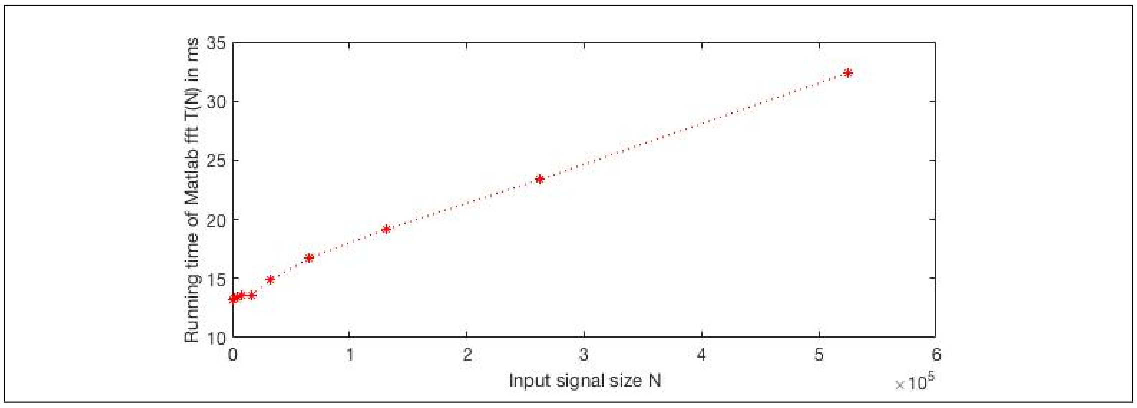

Figure 3.5: Plot of T(N) vs. Input signal k value ($log_2 N$) for input signal sizes from 10 to 524288, randomized entries between 1 and 1000, averaged over 10 runs of the built-in Matlab fft algorithm

It is clear from the figures below that Matlab's built-in fft performs far better (runs faster) for large input signals (size ¿ 2^{14}), but our implementation is faster for signals smaller than size 2^{13}. For input sizes between 2^{13} and 2^{14}, the running times coincide for Matlab and our code. In fact, for large signals, the running time of Matlab's version does not vary considerably.

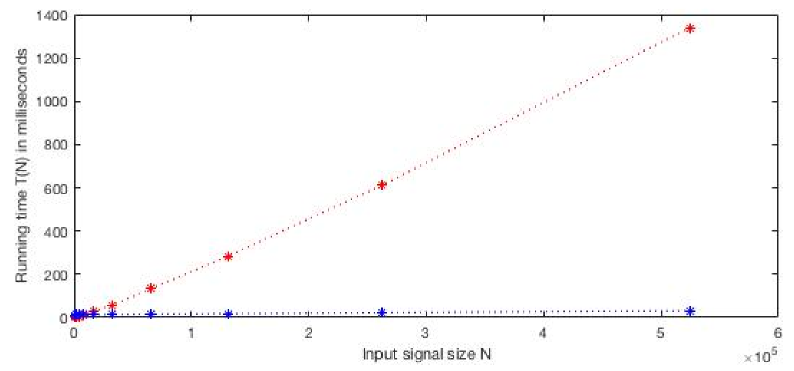

Figure 3.6: Plot of T(N) vs. Input signal k value (log_2N) for input signal sizes from 10 to 524288, randomized entries between 1 and 1000. Here the red line represents our java FFT implementation, and the blue line is Matlab's in-built fft.

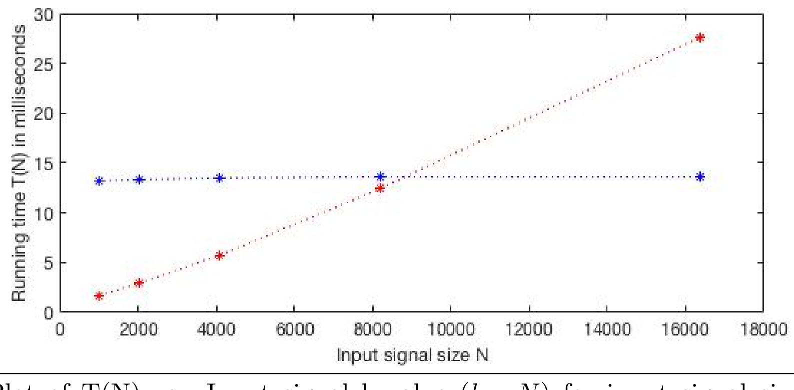

Figure 3.7: Plot of T(N) vs. Input signal k value (log_2N) for input signal sizes from 10 to 16384, randomized entries between 1 and 1000. Here the red line represents our java FFT implementation, and the blue line is Matlab's in-built fft.

Chapter 4

Algorithms for composite N

If N is a composite number, there are several algorithms for computing the Fourier transform: the Prime factor algorithm (PFA, Bluestein's algorithm and Winograd's Fourier transform algorithm (which is an extension of Rader's FFT algorithm).

In this chapter, we discuss the Prime factor algorithm.

4.1 The Prime-factor FFT algorithm

4.1.1 The derivation

If $N = r_1 r_2$, where r_1 and r_2 are relatively prime, or the greatest common divisor of r_1 and r_2, represented by GCD(r_1, r_2), is equal to 1, the indices n and k of Eqn. (3.1) can be expressed as [5]

$$n = n_1 r_1 + n_2 r_2 (modN), n_1 = 0, 1, ..., r_2 - 1, n_2 = 0, 1, ..., r_1 - 1 \tag{4.1}$$

$$k = k_1 r_2 s_1 + k_2 r_1 s_2 (modN), k = 0, 1, ..., N - 1 \tag{4.2}$$

where

$$k_1 = k(modr_1), k_1 = 0, 1, ..., r_1 - 1 \tag{4.3}$$

$$k_2 = k(modr_2), k_2 = 0, 1, ..., r_2 - 1 \tag{4.4}$$

and s_1, s_2 are solutions of

$$s_1 r_2 = 1(modr_1) \tag{4.5}$$

$$s_2 r_1 = 1(mod r_2) \tag{4.6}$$

Representing the sequences X_k and x_n as two-dimensional arrays, Eqn. (3.1) can be rewritten as [5]

$$X_{k_1,k_2} = \sum_{n_2=0}^{r_1=1} \sum_{n_1=0}^{r_2-1} W_N^{(n_1 r_1 + n_2 r_2)(k_1 r_2 s_1 + k_2 r_1 s_2)} x_{n_1,n_2} \tag{4.7}$$

Since

$$W_N^{r_1 r_2 n_1 k_1 s_1} = W_N^{r_1 r_2 n_2 k_2 s_2} = 1 \tag{4.8}$$

Eqn. (4.7) is simplified to

$$X_{k_1,k_2} = \sum_{n_2=0}^{r_1-1} W_{r_1}^{k_1 n_2} A_{k_2,n_2} \tag{4.9}$$

where

$$A_{k_2,n_2} = \sum_{n_1=0}^{r_2-1} W_{r_2}^{n_1 k_2} x_{n_1,n_2} \tag{4.10}$$

In other words, the N-point DFT can be viewed as r_1 r_2-point DFT's followed by r_2 r_1-point DFT's. If the r_1 and r_2-point DFT computations require M_1 and M_2 operations respectively, the N-point DFT can be calculated in $N\left(\frac{M_1}{r_1} + \frac{M_2}{r_2}\right)$ operations. [5]

There are several differences between the PFA and the FFT. Firstly, the factors of N must be mutually prime in the PFA. The twiddle factors have disappeared in the PFA. Finally, the index mapping which converts the one-dimensional arrays in Eqn (3.1) into two-dimensional arrays (multidimensional in general) is different for the two cases. [5]

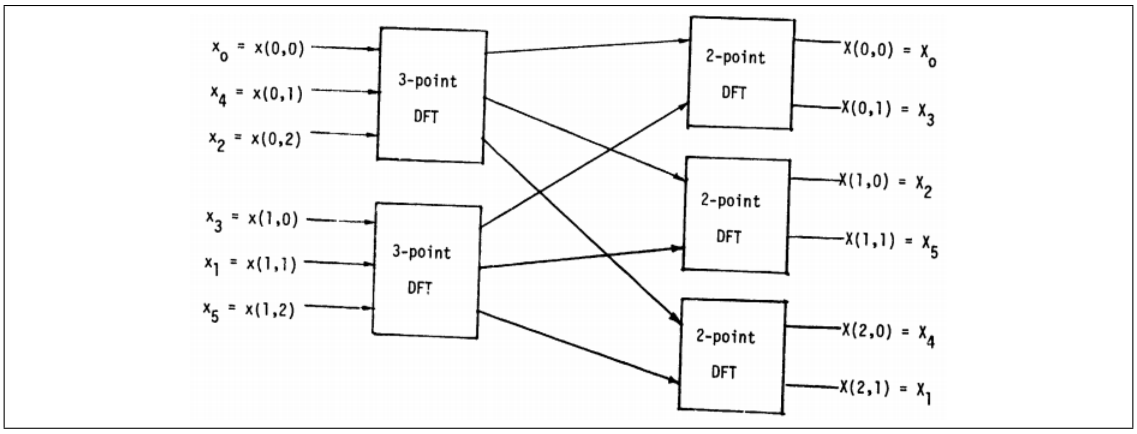

Figure 4.1: The full flowgraph of a 6-point DFT computation using PFA [5]

4.1.2 Complexity Analysis

Let $N = r_1 r_2 .... r_L$ where $r_1 r_2 .... r_L$ are relatively prime.

If A_i and M_i are the number of additions and multiplications respectively, required to compute r_i-point DFT, then in the PFA, the number of real additions is [5]

$$2N \sum_{i=1}^{L} \frac{A_i}{r_i}$$

, and the number of real multiplications is

$$2N \sum_{i=1}^{L} \frac{M_i}{r_i}$$

Chapter 5

Applications

The Fourier transform or the Discrete Fourier transform has widespread practical applications. Some of these include digital signal processing, image processing, solving partial differential equations and performing operations such as convolutions or multiplying large integers. [2]

5.1 Digital Signal Processing

Firstly, the fourier transform of a signal informs us what frequencies are present in the signal and in what proportions.

The following is a list of reasons why the Fourier transform has extensive significance in signal processing: [6]

(1) The magnitude of the Fourier transform tells us how much power the signal $x(t)$ has at a particular frequency f.

(2) We have

$$\int_R |x(t)|^2 = \int_R |X(f)|^2 df$$

,from Parseval's Theorem, which implies the total energy in a signal across all time equals the total energy in the transform across all frequencies, which in turn means the transform conserves energy.

(3) For two signals $x(t)$ and $y(t)$, if

$$z(t) = x(t) \times y(t)$$

, where $\times$ denotes convolution, the Fourier transform of $z(t)$ is

$$Z(f) = X(f).Y(f)$$

. In other words, convolutions in time domain are equivalent to multiplications in the frequnecy domain. With the evolution of efficient FFT algorithms, it is almost always faster to implement a convolution operation in the frequency domain than in the time domain.

(4) Similar to the convolution operation, cross-correlations are also implemented with ease in the frequency domain than in the time domain.

(5) Splitting signals into their constituent frequencies allows one to block out certain frequencies selectively by nullifying their contributions.

(6) A shifted (delayed) signal in the time domain exhibits as a phase change in the frequency domain. This is used frequently in imaging and tomography applications.

(7) Fourier transforms can be used to calculate derivatives of signals (including n^{th} derivatives).

Note: The link

https://en.wikipedia.org/wiki/cooley–tukey$_fft_algorithm$

was added to the bibliography file in sharelatex, but wherever it has been cited, it shows up as [$-tukey_fft_algorithm$]

5.2 Addendum

(1) Section 3.3 (Simulations and Numerical Characterization) in CHapter 3

(2) Section 3.4 (Conclusions from the simulations)

(3) Section 3.5 (Comparison with other built-in implementations)

(4) Java class FFTAnalysis attached to the zip file along with the classes Complex.java and FFT.java

Bibliography

[1] https://en.wikipedia.org/wiki/fourier$_{t}ransform$.

[2] https://en.wikipedia.org/wiki/discrete$_{f}ourier_{t}ransform$.

[3] https://en.wikipedia.org/wiki/fast$_{f}ourier_{t}ransform$.

[4] http://eeweb.poly.edu/iselesni/el713/zoom/fft.

[5] http://www.dtic.mil/dtic/tr/fulltext/u2/a058049.pdf.

[6] http://dsp.stackexchange.com/questions/69/why-is-the-fourier-transform-so-important.