An Implementation and Analysis of Fast Fourier Transform: CSCI 5454 Design and Analysis of Algorithms Project

Lakhan Shiva Kamireddy

December 9, 2016

Contents

1 Introduction

The Fast Fourier Transform (FFT) is one of the most widely used algorithm in the Digital Signal Processing arena. The reason for its popularity is due to its computational efficiency in finding the Discrete Fourier Transform (DFT) of a time series. Almost every continuous signal can be broken down into a combination of simple waves. It can be restated mathematically as follows- Any wave can be written as a sum of sines and cosines. Fourier series is a tool to break the signal into its constituent parts. Fourier transform is a generalization of the Fourier series. Fourier transforms take a signal and express it in terms of the frequencies that make up the signal. Strictly speaking, it only applies to continuous and aperiodic functions, but the use of the impulse function allows the use of discrete signals. Present technology demands that we process signals by discrete methods. Computers and digital processing systems can work with finite sums only. To convert a continuous signal to a discrete signal, we sample the signal at finite regular intervals. While sampling keeping in mind Shannon's sampling theorem, we sample it at a frequency that preserves the signal information. Let us take a time series of length N_o. The Discrete Fourier Transform equation is given by $F[r] = \sum_{k=0}^{N-1} f[k]e^{-j\frac{2\pi kr}{N_o}}$.

To compute one sample of DFT using the DFT equation would take N_o complex multiplications and $N_o - 1$ complex additions. To compute N_o such samples, we require a total of N_o^2 complex multiplications and $N_o(N_o - 1)$ complex additions, the time complexity being $O(N^2)$. Wouldn't it be fascinating if we can solve the same problem with just $N_o \log N_o$ computations ?. FFT algorithm exactly does this, by efficiently computing the Discrete Fourier Transform of a time series (discrete time samples). This makes FFT much faster for large values of N_o. The idea behind FFT is the divide and conquer approach, to break up the original N_o point sample into two $\frac{N_o}{2}$ sequences. We follow this technique because a series of smaller problems is easier to solve than one large problem.

2 Algorithm

The FFT algorithm was invented around 1805 by Carl Friedrich Gauss, but it became popular only after James Cooley and John Tukey have published a paper in 1965 reinventing the algorithm and describing how to perform it conveniently on a computer. Thereon Cooley-Tukey and Sande-Tukey are the popular FFT algorithm variants. These algorithms work best when N is a power of 2, or more generally when N is highly composite.

FFT takes advantage of the symmetry and periodicity of the complex exponential. Let $W_{N_o} = e^{(-j2\pi/N_0)}$.

Symmetry: $W_{N_o}^{k[N_o-n]} = W_{N_o}^{-kn} = \left(W_{N_o}^{kn}\right)^*$

Periodicity: $W_{N_o}^{kn} = W_{N_o}^{k[n+N_o]} = \left(W_{N_o}^{[k+N_o]n}\right)^*$

We can see that two $N/2$ length DFTs take less computation than one length N DFT as $2(\frac{N}{2})^2 < N^2$. FFT exploits the same property. There are two variants of FFT algorithm. One is "Decimation in Time" and other is "Decimation in Frequency". Decimation is the process of breaking down something into it's constituent parts.

2.1 Decimation in Time

A radix-2 decimation in time (DIT) FFT is the most commonly used form of Cooley-Tukey algorithm. Decimation in time involves breaking down a signal in the time domain into smaller signals, each of which is easier to handle. N_o point data sequence f_k is divided into two $\frac{N_o}{2}$ point sequences consisting of even and odd numbered samples respectively. $f_0, f_2, f_4, ..., f_{N_o-2}$ is the even sequence, g_k and $f_1, f_3, f_5, ..., f_{N_o-1}$ is the odd sequence, h_k.

$$F[k] = \sum_{n=0}^{N_o-1} f[n] W_{N_o}^{nk}$$

$$= \sum_{n=even} f[n] W_{N_o}^{nk} + \sum_{n=odd} f[n] W_{N_o}^{nk}$$

$$= \sum_{r=0}^{\frac{N_o}{2}-1} f[2r] \left(W_{N_o}^2\right)^{rk} + W_{N_o}^k \sum_{r=0}^{\frac{N_o}{2}-1} f[2r+1] \left(W_{N_o}^2\right)^{rk}$$

$$= \sum_{r=0}^{\frac{N_o}{2}-1} f[2r] \left(W_{\frac{N_o}{2}}\right)^{rk} + W_{N_o}^k \sum_{r=0}^{\frac{N_o}{2}-1} f[2r+1] \left(W_{\frac{N_o}{2}}\right)^{rk}$$

Let $\sum_{r=0}^{\frac{N_o}{2}-1} f[2r] \left(W_{\frac{N_o}{2}}\right)^{rk}$be G_k and $\sum_{r=0}^{\frac{N_o}{2}-1} f[2r+1] \left(W_{\frac{N_o}{2}}\right)^{rk}$ be the sequence H_k. We have $F[k] = G_k + W_{N_o}^k H_k$ where G_k and H_k are the $\frac{N_o}{2}$ point DFTs of the even and odd numbered sequences, g_k and h_k respectively. G_k and H_k being $\frac{N_o}{2}$ point DFTs are $\frac{N_o}{2}$ periodic. Thus $G_{k+\frac{N_o}{2}} = G_k$ and $H_{k+\frac{N_o}{2}} = H_k$. Moreover $W_{N_o}^{k+\frac{N_o}{2}} = W_{N_o}^{\frac{N_o}{2}} W_{N_o}^k = e^{-j\pi W_{N_o}^k} = -W_{N_o}^k$. Thus $F[k + \frac{N_o}{2}] = G_k - W_{N_o}^k H_k$. This property is used to reduce the number of computations.

2.1.1 Butterfly diagram

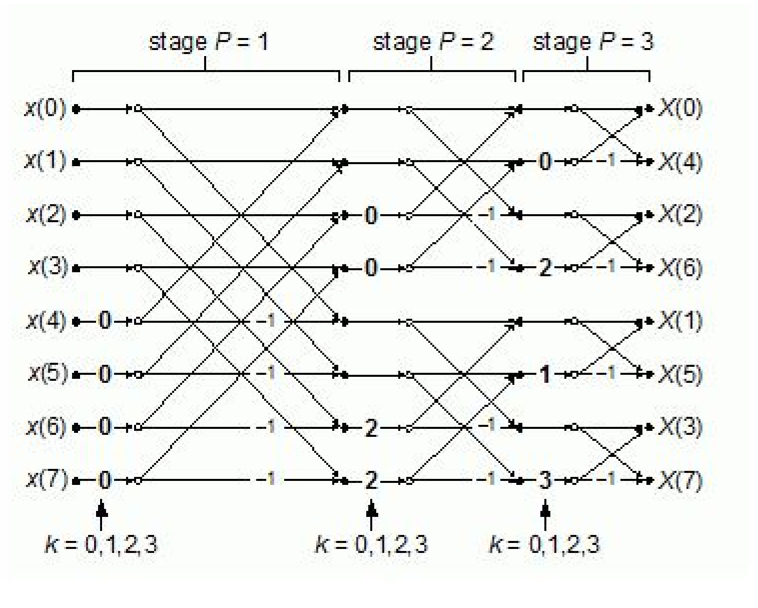

Fig. 1. 8-point DIT FFT butterfly diagram

2.2 Decimation in Frequency

In this algorithm, the output or frequency points are regrouped. Decimation in frequency involves breaking down a signal in the time domain into smaller signals, each of which is easier to handle. We divide the signal points into two equal length sequences with one sequence consisting of first half of the signal samples and the other sequence consisting of other half of the signal samples. N_o point data sequence f_k is divided into two $\frac{N_o}{2}$ point sequences. $f_0, f_1, f_2, ..., f_{N_o/2-1}$ is the sequence g_k and $f_{N_o/2}, f_{N_o/2+1} f_{N_o/2+2} f_{N_o-1}$ is the sequence h_k. We know that $F[k] = \sum_{n=0}^{N_o-1} f[n] W_{N_o}^{nk}$.

The FFT formula for Decimation in Frequency is given as follows.

Replace k with r where $k = 2r$ is even and $k = 2r+1$ is odd.

$F[2r] = \sum_{n=0}^{\frac{N_o}{2}-1} \left(f[n] + f[n+\frac{N_o}{2}]\right)\left(W_{\frac{N_o}{2}}^{nr}\right)$ for $0 \le r \le \frac{N_o}{2} - 1$

$= \sum_{n=0}^{\frac{N_o}{2}-1} a_n \left(W_{\frac{N_o}{2}}^{nr}\right) = A_n$

$F[2r+1] = \sum_{n=0}^{\frac{N_o}{2}-1} \left(f[n] - f[n+\frac{N_o}{2}]\right)\left(W_{N_o}^{n(2r+1)}\right)$

$= \sum_{n=0}^{\frac{N_o}{2}-1} \left(f[n] - f[n+\frac{N_o}{2}]\right)\left(W_{N_o}^{n}\right)\left(W_{\frac{N_o}{2}}^{rn}\right)$ for $0 \le r \le \frac{N_o}{2} - 1$

$= \sum_{n=0}^{\frac{N_o}{2}-1} b_n \left(W_{\frac{N_o}{2}}^{nr}\right) = B_n$

We now have two new subsequences a_n and b_n of length $\frac{N_o}{2}$. We get even-odd indexed DFT coefficients in terms of A_n and B_n respectively. In the next step, the sequences a_n and b_n can be further subdivided into two subsequences of length $\frac{N_o}{4}$ each. This process of dividing the sequence into subsequences of length $\frac{N_o}{2^{m-1}}$ where m is the stage number continues till we are left with only length-2 sequences and DFTs can be found without multiplication.

The total number of stages $m = \log_2 N_o$.

2.2.1 Butterfly diagram

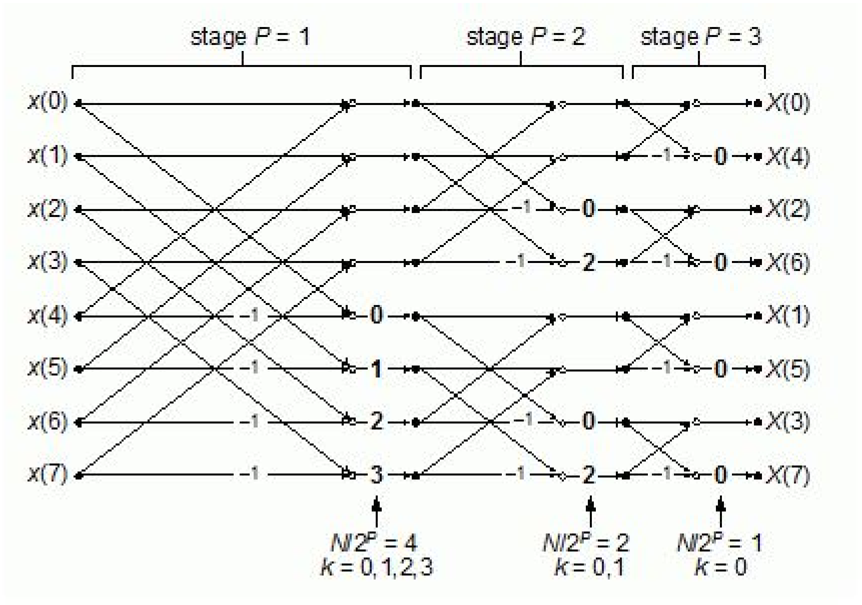

Fig. 2. 8-point DIF FFT butterfly diagram

The N-point DIF FFT has $\log_2 N$ stages, numbered $P = 1, 2, ..., \log_2 N$.

Each stage comprises of $\frac{N}{2^P}$ butterflies.

2.3 Inverse Discrete Fourier Transform

FFT algorithm can also be used to find the Inverse Discrete Fourier Transform (IDFT) of the signal.

IDFT is given by $f[n] = \frac{1}{N_o}\sum_{r=0}^{N_o-1} F[r]\left(W_{N_o}^{-rn}\right)$, $0 \leq k \leq N_o - 1$. Taking the complex conjugate of the function we get $N_o f^*[n] = \sum_{r=0}^{N_o-1} F^*[r]\left(W_{N_o}^{rn}\right)$. But the RHS of the equation is DFT of sequence $F^*[r]$. Thus, $f^*[n] = \frac{1}{N_o} DFT\left(F^*[r]\right)$. Taking complex conjugate on both the sides, $f[k] = \frac{1}{N_o}\left(\sum_{r=0}^{N_o-1} F^*[r] W_{N_o}^{rn}\right)^*$. Therefore, $f[k] = \frac{1}{N_o}\left(FFT\left(F^*[r]\right)\right)^*$. FFT algorithm can be used to compute IDFT when we take the twiddle factors as negative powers of W_{N_o} i.e. powers of $W_{N_o}^{-1}$ instead of powers of W_{N_o}.

IDFT can be found using DIF FFT algorithm where $f[n]$ is replaced by $F[r]$ and converting the twiddle factors from $W_{N_o}^{k}$ to $W_{N_o}^{-k}$ and dividing the final result by N_o.

2.4 Pseudocode

```
FFT($f$):
    $N = length(f)$ //input $f$ is power-of-2 sequence
    $loglength = \log_2 N$
    for $k = 0 : N/2$
        $Wn(k+1) = \exp(\frac{-2\times\pi\times j\times k}{N})$
    $end$
    for $m = loglength : -1 : 1$
        $length = 2^{m-1}$
        $i = 1;$
        $while\,(i \leq N-1)$
            for $k = 0 : 1 : length - 1$
                $F\,(i) = f\,(i) + f\,(i + length)$
                $F(i + length) = (f(i) - f(i + length)) \times (Wn(k+1))^{2^{(loglength-m)}}$
                $i = i + 1$
                $if\,(k = length - 1)$
                    $i = i + length$
                $end$
            $end$
        $end$
        $f = F$
    $end$
    $F = bitreverseorder\,(F)$
```

At a high level, the algorithm calculates FFT in stages according to the DIF butterfly diagram. The output of first stage is given as input to stage 2 and so on. There are $\log_2 N$ stages. At each stage, the computations are done as per the above implementation, which is visualized above in the form of a butterfly diagram. The final output is in the bit reversed order. At the end of the final stage we arrange the outputs by doing a bit reversal operation on the output sequence. In simple terms, bit reversal means mirror imaging the binary numbers. Some signal processors can do this on hardware. In Matlab there is a function for bit reversal namely $bitrevorder()$. For an input sequence of size 8, the following table enumerates bit reversed order.

Input order			Bit reversed order
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Table 1. Illustrating the bit reversed order for input sequence of length 8.

2.5 Chirp-Z Transform

Bluestein's FFT, commonly called the Chirp-Z Transform (CZT) offers a high resolution FFT combined with the ability to specify bandwidth. The reason for exploring the CZT lies in the restrictions to the above FFT implementation. One important limitation is the power-of-two rule, requiring that the number of input samples to be a power of two. Compared to FFT, CZT is much more flexible. Given the sampling rate and the number of samples taken, the resolution can be tailor made by means of adjusting the start and stop frequencies and the number of output samples. However, it is ironic to find that to make the spectrum analysis more flexible, CZT uses FFT itself. Therefore, the faster the FFT becomes, the more efficient the CZT becomes. In spite of this, the CZT will never be faster than the FFT.

The CZT can calculate the spectrum of a signal over an arc of the unit circle and in the usual sense is used for the same purpose, although, not limited to it. CZT can be understood as the convolution of the sampled input and unit circle arc coefficients defined by the user. As can be seen in the following figure, two FFT's and one inverse FFT is used to calculate the CZT. However the CZT code will zero-pad the input samples to a power of two; therefore the user is not restricted to give a power of two sequence.

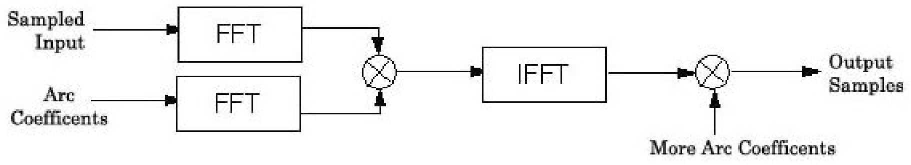

Fig. 3. Block diagram showing the Chirp-Z transform implementation.

I have implemented the chirp-z transform version of FFT for the non power-of-2 input sequences, using the existing routines for FFT and IFFT in matlab. However, it is ironic that to make the spectrum analysis more flexible, CZT uses FFT itself.

2.6 Prime Factor FFT algorithm

Good-Thomas FFT, also known as Prime Factor FFT algorithm expresses the DFT of size N as $N = N_1 \times N_2$ as a two nested DFTs (two dimensions). This works only for the case where N_1 and N_2 are relatively prime (co-primes). The only limitation is that this works when we are able to express N in this sort of a factored format.

Let us take the Discrete Fourier Transform formula, $F[k] = \sum_{n=0}^{N_o-1} f[n]e^{-j\frac{2\pi nk}{N_o}}$.

We re-index the input n and the output k.

$n = n_1 N_2 + n_2 N_1 \bmod N$ and $k = k_1 N_2^{-1} N_2 + k_2 N_1^{-1} N_1 \bmod N$. n_1 runs from 0 through $N_1 - 1$ and n_2 runs from 0 through $N_2 - 1$. N_1^{-1} is the multiplicative inverse of N_1 mod N_2 and N_2^{-1} is the multiplicative inverse of N_2 mod N_1. N_1^{-1} and N_2^{-1} always exist since N_1 and N_2 are co-prime. The re-indexing of n is called Good's mapping and re-indexing of k is called (Chinese Remainder Theorem) CRT mapping.

$F[k_1 N_2^{-1} N_2 + k_2 N_1^{-1} N_1] = \sum_{n_1=0}^{N_1-1} \left(\sum_{n_2=0}^{N_2-1} f[n_1 N_2 + n_2 N_1] e^{-\frac{2\pi j}{N_2} n_2 k_2} \right) e^{-\frac{2\pi j}{N_1} n_1 k_1}$.

$= \sum_{n_1=0}^{N_1-1} \left(\sum_{n_2=0}^{N_2-1} f[n_1 N_2 + n_2 N_1] W_{N_2}^{n_2 k_2} \right) W_1^{n_1 k_1}$The inner sum is a DFT of size N_2 and the outer sum is a DFT of size N_1.

I have not implemented the prime factor FFT algorithm, but I think that it should be present when doing a write-up on FFT, hence have included it here.

3 Analysis

In this section we prove the correctness of the FFT algorithm and explore the complexity in terms of time.

3.1 Correctness

Let us take the FFT Decimation in Frequency formula:

$F[2r] = \sum_{n=0}^{\frac{N_o}{2}-1} \left(f[n] + f[n + \frac{N_o}{2}] \right) \left(W_{\frac{N_o}{2}}^{nr} \right)$ for $0 \le r \le \frac{N_o}{2} - 1$

$F[2r+1] = \sum_{n=0}^{\frac{N_o}{2}-1} \left(f[n] - f[n + \frac{N_o}{2}] \right) \left(W_{N_o}^{n} \right) \left(W_{\frac{N_o}{2}}^{rn} \right)$ for $0 \le r \le \frac{N_o}{2} - 1$

We need to prove that the above formula indeed calculates the Discrete Fourier transform of the input signal.

Let us take the DFT formula, $F[k] = \sum_{n=0}^{N_o-1} f[n] W_{N_o}^{nk}$

$F[k]$ can be rewritten as $F[k] = \sum_{n=0}^{\frac{N_o}{2}-1} f[n] \left(W_{N_o}^{nk} \right) + \sum_{n=\frac{N_o}{2}}^{N_o-1} f[n] \left(W_{N_o}^{nk} \right)$

$F[k] = \sum_{n=0}^{\frac{N_o}{2}-1} f[n] \left(W_{N_o}^{nk} \right) + \sum_{n=0}^{\frac{N_o}{2}-1} f[n + \dfrac{N_o}{2}] \left(W_{N_o}^{[n+\frac{N_o}{2}]k} \right)$

$= \sum_{n=0}^{\frac{N_o}{2}-1} f[n] \left(W_{N_o}^{nk} \right) + (-1)^k \sum_{n=0}^{\frac{N_o}{2}-1} f[n + \dfrac{N_o}{2}] \left(W_{N_o}^{nk} \right)$ as $W_{N_o}^{(\frac{N_o}{2})k} = (-1)^k$

$= \sum_{n=0}^{\frac{N_o}{2}-1} \left(f[n] + (-1)^k f[n + \frac{N_o}{2}] \right) \left(W_{N_o}^{nk} \right)$ for $0 \le k \le N_o - 1$

We replace k with r where $k = 2r$ is even and $k = 2r + 1$ is odd.

$F[2r] = \sum_{n=0}^{\frac{N_o}{2}-1} \left(f[n] + f[n + \frac{N_o}{2}] \right) \left(W_{\frac{N_o}{2}}^{nr} \right)$ for $0 \le r \le \frac{N_o}{2} - 1$

$F[2r+1] = \sum_{n=0}^{\frac{N_o}{2}-1} \left(f[n] - f[n + \frac{N_o}{2}] \right) \left(W_{N_o}^{n(2r+1)} \right)$

$= \sum_{n=0}^{\frac{N_o}{2}-1} \left(f[n] - f[n + \frac{N_o}{2}] \right) \left(W_{N_o}^{n} \right) \left(W_{\frac{N_o}{2}}^{rn} \right)$ for $0 \le r \le \frac{N_o}{2} - 1$

3.2 Runtime

The DFT equation is given by $F[k] = \sum_{n=0}^{N_o-1} f[n] e^{-j\frac{2\pi nk}{N_o}}$. DFT runs N_o times for each element of the output sequence. For N_o such elements, the complexity of DFT is $O\left(N_o^2\right)$.

The FFT is calculated by the above given formula. If we look at the implementation, the algorithm's outer loop runs $\log N_o$ times. The inner loop does N_o complex additions and multiplications. Therefore the running time complexity of the above implementation is $O\left(N_o \log N_o\right)$. By using FFT implementation, we were able to reduce the complexity from $O\left(N_o^2\right)$ to $O\left(N_o \log N_o\right)$ which is a significant improvement.

The FFT algorithm's runtime is compared with that of DFT for various input lengths. The figure illustrates the comparison of FFT with DFT, with log of length of input sequence on the x-axis and the clock time for computing the Fourier Transform on y-axis.

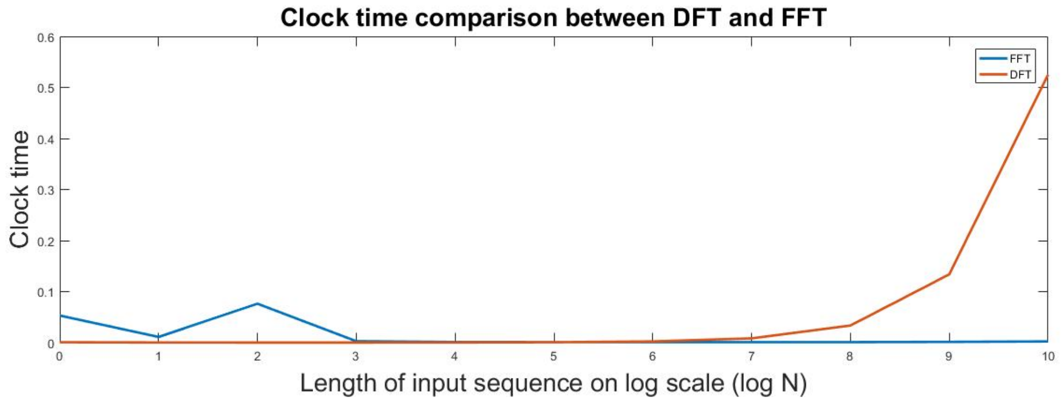

Fig. 4. Clock time comparison between DFT and FFT algorithm for different lengths of input sequences.

We have written a simple DFT function in matlab to facilitate the above comparison. To make the above comparison, we have created a script that calls simple dft function and the current function on a randomly generated input sequence. It then plots the clock time for both the methods in the above graph. We have run this on Matlab R2016a (9.0.0.341360) 64 bit. The results were generated on a PC running Intel Core i5-6200U CPU processor with 8 GB RAM.

3.3 Random Analysis

The input sequences for the FFT implementation were generated randomly and input to the algorithm. The outputs generated were compared with the outputs of matlab's FFT implementation. The results match, strengthening the current implementation's credibility.

Pseudocode:

```
$N$= length of input sequence // Plots for both power-of-2 and non powe-of-2 input sequences
$sequence$=[]
for $i$ = $1 : N$
    $sequence$= [$sequence\ rand(i)$] //randomly generated number is added to the sequence
end
$f$=fft_algo($sequence$);
$plot(abs(f))$//Current implementation Magnitude plot
$F$=$fft(sequence)$;
$plot(abs(F))$// Matlab FFT function Magnitude plot
//Angle plots
$fft_algo(angle(f))$
$fft(angle(f))$
```

The following figure illustrates the results of performing random analysis on the current fft implementation.

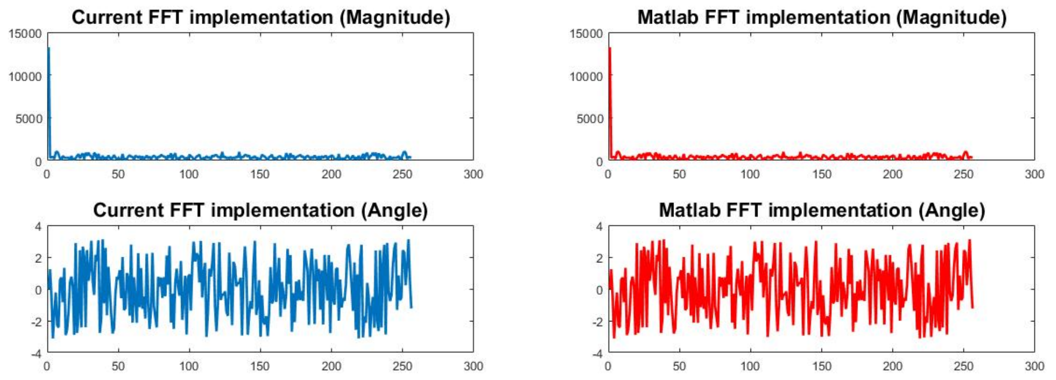

Fig. 5. Random analysis on power-of-2 input sequence

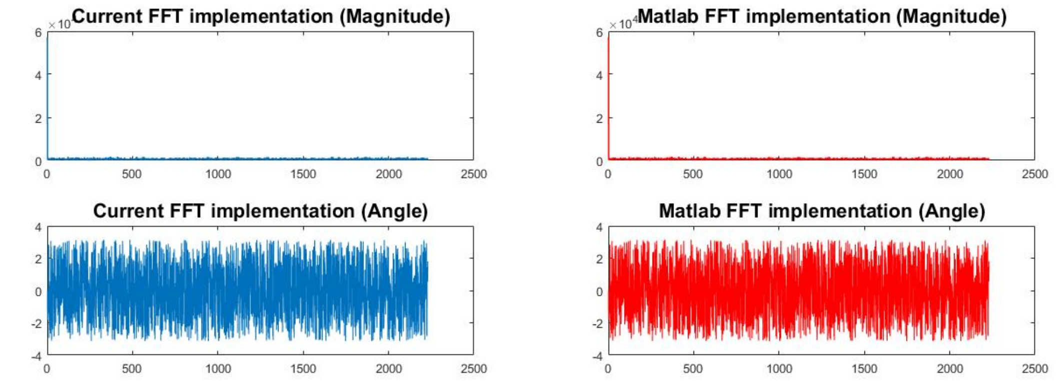

Fig. 6. Random analysis on non-power-of-2 input sequence

4 Conclusion

Apart from being rather an intriguing technique to speed up the Fourier analysis, the Fast Fourier Transform is one of the most important algorithms in signal processing. Fast Fourier Transforms have applications to literally several hundreds of subfields of computer science. They can be used in doing the large integer multiplications, computing the Fast Discrete Cosine Transform, etc. The impact this had on the scientific field is truly significant. For example, the fourier transforms are the backbone of image/audio compression algorithms in pictures (png, mpeg), audio (mp3) and video (avi) formats which are used in millions of applications worldwide.

The improvement Fast Fourier Transform algorithm has over the simple discrete fourier transform computation is truly significant in terms of its runtime (computation time). It was shown that it has improved from $O(N^2)$ to $O(N \log N)$.

In this paper we briefly outlined the history of FFT algorithm and presented implementation for power-of-two and non-power-of-two sequences. The algorithm was analysed in terms of runtime complexity and the experimental results were discussed.

References

[1] E Oran Brigham and Elbert Oran Brigham. *The fast Fourier transform*, volume 7. Prentice-Hall Englewood Cliffs, NJ, 1974.

[2] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[3] I. J. Good. The interaction algorithm and practical fourier analysis: An addendum. *Journal of the Royal Statistical Society. Series B (Methodological)*, 22(2):372–375, 1960.

[4] Mathworks. The chirp z-transform matlab. *https://www.mathworks.com/help/signal/ref/czt.html*, 2010.

[5] Lawrence R. Rabiner, Ronald W. Schafer, and Charles M. Rader. The chirp z-transform algorithm and its application. *Bell System Technical Journal*, 48(5):1249–1292, 1969.

[6] B.R. Sekhar and K.M.M. Prabhu. Radix-2 decimation-in-frequency algorithm for the computation of the real-valued fft. *Trans. Sig. Proc.*, 47(4):1181–1184, April 1999.