Data Structures & Algorithms

Final Project

Complexity Analysis of Discrete Fourier Transform Algorithms

Group Members:

Siddharth Rupavatharam - siddharth.r@rutgers.edu - sr1090
Peeyush Tambe - peeyush.tambe@rutgers.edu - pgt17
Yilin Yang - yilin.yang@rutgers.edu - yy450

I. Introduction and Motivation

The Fourier theorem was developed by a French mathematician named Jean-Baptiste Joseph Fourier in the late 1700's to better understand and describe heat transfer and wave motion in materials. The fundamental premise of the theorem is that any continuous function or signal can be decomposed as a sum of infinite sines and cosines or complex exponentials ***[1]***. Today this observation, along with a little more mathematical rigour and applied boundary conditions, is widely used in acoustics, digital signal processing, control theory, solving higher order partial differential equations and approximating trigonometric equations.

A Fourier series is a way to represent any continuous function as an infinite sum of sines and cosines or complex exponentials ***[1]***. This means a function in time can be represented as a sum of other functions in time. As shown in Fig 1. a square wave can be generated or approximated using a sum of sines and cosines. Whereas the Fourier Transform is a way to decompose a function in time into its constituent frequencies present with the absolute value representing the amount of frequency present in the original time function ***[2]***. This means a function in time is converted to a function in frequency giving information about the amount of individual frequencies which make up the original function in time as shown in Fig 2. Hence the Fourier transform is called the *frequency domain representation* of the original signal.

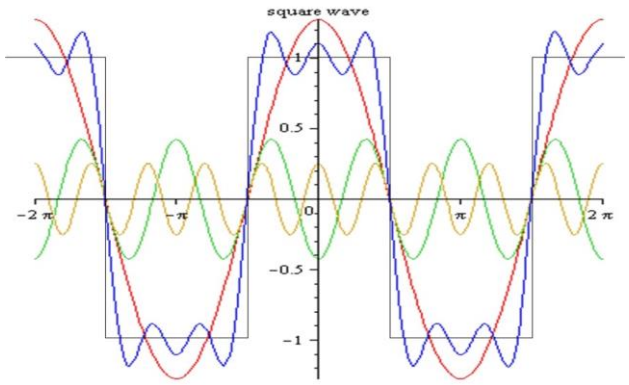

Fig 1. (a) Fourier series using sines and cosines [5]

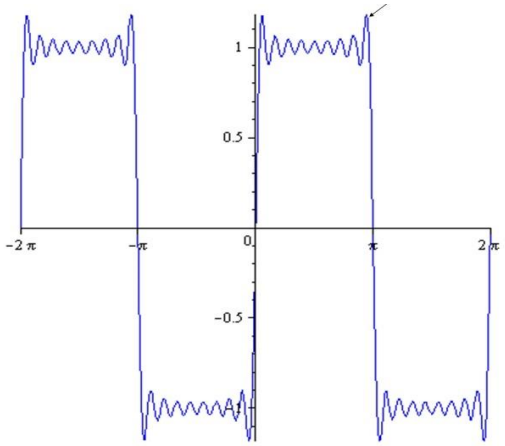

Fig 1. (b) Approximation of a square wave using sines and cosines [5]

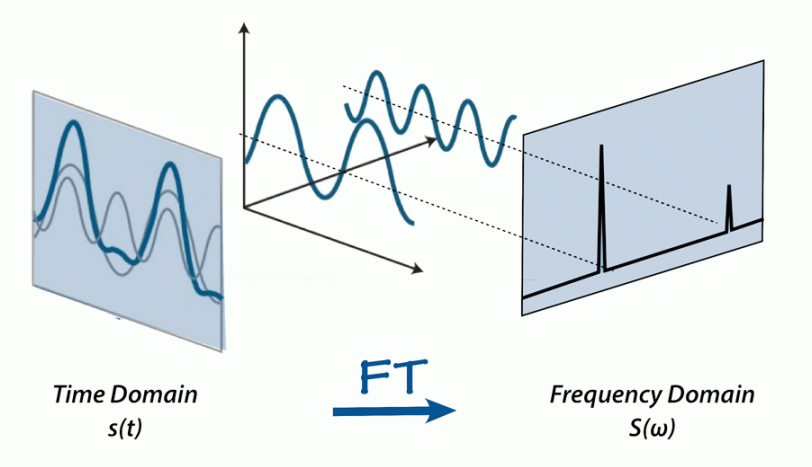

Fig 2. Fourier Transform of a signal s(t) [6]

To give the reader an intuition about the application and working of the Fourier theorem the following paragraph is presented

To give a performance of Verdi's opera Aida, one could do without brass and woodwinds, strings and percussion, baritones and sopranos; all that is needed is a complete collection of tuning forks, and an accurate method for controlling their loudness. This is an application to acoustics of "Fourier's theorem," one of the most useful facts in many branches of physics and engineering.

-Philip. J Davis, Reuben Hersh The Mathematical Experience, Study Edition (1990), Pg. 255

Understanding and appreciating the mathematical framework and potential of the Fourier series and transforms is the first step. The challenge lies in implementing these as parts of a real world system. In the modern world the Discrete Fourier Transform (DFT) was first introduced as the digital counterpart to the Fourier series. The DFT gives the discrete, sampled values of the infinite, continuous Fourier series. From an algorithms perspective the DFT has a time complexity of O(N^2). The Fast Fourier Transform (FFT) is an improvement on the DFT algorithm and reduces the time complexity to O(NlogN).

This report focuses on two real world implementations of the Fourier Transform namely Discrete Fourier Transform (DFT) and Fast Fourier Transform (FFT). The fact that the Fast Fourier Transform reduces the time complexity from an O(N^2) to an O(N log(N)) and implementing both the algorithms on a fixed spec system demonstrates the leaps in performance a good algorithm provides serve as motivation. While some operations are easier to achieve in the Fourier domain, the complexity of the DFT make it incompatible with real-time operations. When several applications require sampling at a rate of hundreds of thousands per second, a time delay of more than a few seconds is impractical. FFT resolves this be offering a drastically faster implementation, enabling many aspects of modern communication. Finally the steps required in understanding, implementing and analyzing the algorithms give us an opportunity to put to practise topics discussed as part of ECE 573 Data Structures and Algorithms.

II. Algorithms Investigated

(i) Discrete Fourier Transform (DFT) [3]

The Discrete Fourier Transform (DFT) is a valuable tool in digital signal processing for performing frequency analysis of signals in the time domain. Given an analog continuous time signal x(t), it is necessary to discretize the signal through sampling to produce a sequence x(n) on which mathematical operations can be performed. The decomposed sequence is comprised of the constituent frequencies of the original signal which can be used for multiple engineering applications. The implementation of a DFT algorithm has an operational complexity of O(N^2).

The DFT algorithm takes a series of N data points represented by x(n) and produces a new sequence X(k) calculated using:

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-j2\pi kn}{N}}$$

Fundamentally this operation requires N complex multiplies inside the summation (i.e., x(n) times the exponential) and this summation operation needs to be repeated N times to generate N X(k) values. Totally it takes N^2 complex multiplies. Next these multiplies need to summed up, each X(k) requires N adds and there are N-1 such points. Totally it takes N(N-1) adds. Hence the time complexity of the DFT is O(N^2).

Implementation Pseudocode

```
Input: fnList (N point list)
Output: FmList (calculated DFT points in a list)

def DFT(fnList):
    SET N = length(fnList)
    SET FmList = Null array

    For m in range(N):
        SET Fm = 0.0
        For n in range(N):
            FmList[n] += fnList[n] * exp(- 1j * pi2 * m * n / N)
    return FmList
```

(ii) Fast Fourier Transform (FFT) [4]

The radix 2 FFT recursively divides a DFT problem of size N into two subproblems each of size N/2, hence the name. The sequence X(k) in radix 2 FFT can be calculated as follows:

Similar to the DFT implementation an input, x(n), is defined as the sampling of the continuous time signal x(t). The output X(k) is the fourier transform of the signal.

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

The exponential is then written as a twiddle factor W and the x(n) points are split into even and odd parts (decimation). The idea is to divide the input samples into two halves and then calculate the N/2 point DFT this reduces the overall required number of multiples and adds. The odd parts are multiplied with a fixed twiddle factor.

$$W_N^k = \exp(-j.2.\pi.\frac{k}{N}), k = 0, 1, ..N-1$$

$$X_k = \sum_{m=0}^{\frac{N}{2}-1} x(2.m).W_N^{2.m.k} \; + \; W_N^k.\sum_{m=0}^{\frac{N}{2}-1} x(2.m+1).W_N^{2.m.k} \; , \; k = 0...N-1$$

The twiddle factor is then further reduced to

$$W_N^{2.m.k} = \exp(-j.2\pi.2.m.\frac{k}{N}) = \exp(-j.2\pi.m.\frac{k}{\frac{N}{2}}) = W_{\frac{N}{2}}^{m.k}$$

Rearranging

$$X_k = \sum_{m=0}^{\frac{N}{2}-1} x(2.m).W_{\frac{N}{2}}^{m.k} \; + \; W_N^k.\sum_{m=0}^{\frac{N}{2}-1} x(2.m+1).W_{\frac{N}{2}}^{m.k}$$

$$X_k = X_{k(even)} \; + \; W_N^k.X_{k(odd)}, \; k = 0 \; ... \; \frac{N}{2} - 1$$

Thus N point DFT has been reduced to two N/2 point DFTs. So from N^2 multiplies it has been reduced to (N/2)^2 multiplies and a final N multiplies with the twiddle. This can further be reduced by dividing each of the N/2 point DFT to two N/4 DFTs and each N/4 DFTs further to two N/8 DFTs recursively whereby reducing the number of multiplies required to log(N) all the while keeping the final N multiples constant. Hence as long as N is a power of 2 the FFT can be calculated with a time complexity of O(N log(N)). This implementation is known as the radix-2 FFT as the N points are divided and grouped in halves.

Example of FFT

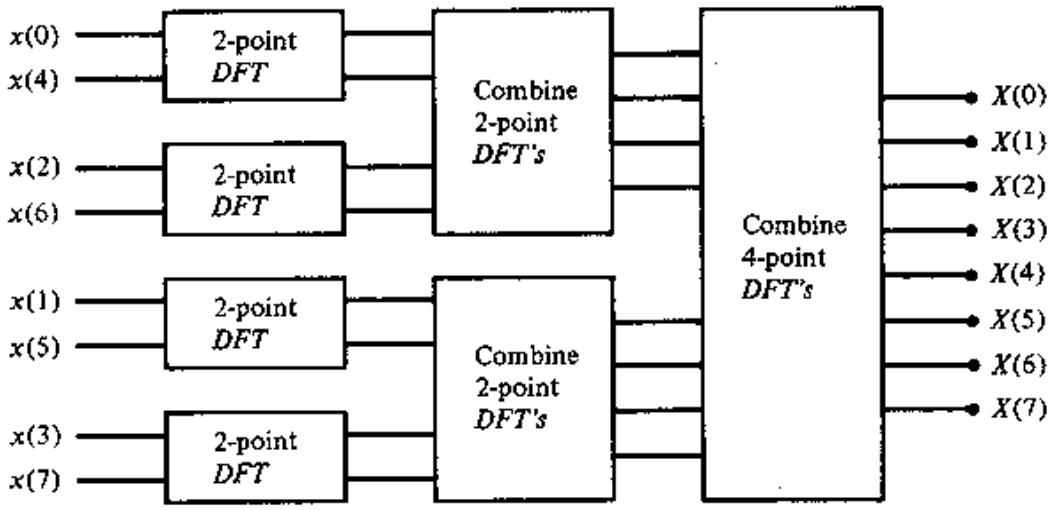

Fig 3. Block diagram of 8 point FFT

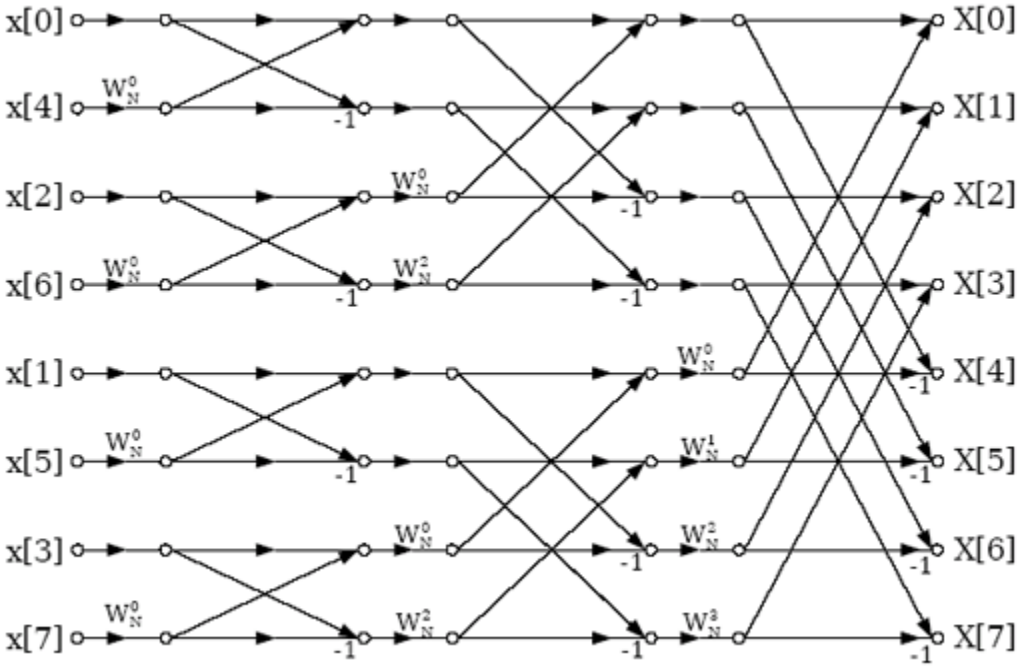

Fig 4. FFT Butterfly

The 8 point FFT is calculated as follows:

The 8 points of x(n) are decimated till pairs of two are formed. It is first divided as x(0), x(2), x(4), x(6) and x(1), x(3), x(5), x(7). These are further grouped as shown in Fig 3. to form the first stage of the FFT and multiplied with the corresponding twiddle factors. The intermediate values

are then multiplied with the next set of twiddle factors and output of the combined 2-point DFTs are multiplied with the next set of twiddle factors and outputs give the FFT.

Implementation Pseudocode

```
Input: seq (N point list)
Output: (calculated DFT points in a list )
def fft(seq):
   SET N_points = length(seq)

    if N_points <= 1:
       return seq

    SET even_seq = seq(even elements)
    even = fft(even_seq)

    SET odd_seq = seq(odd elements)
    odd =  fft(odd_seq)

    For k in range(N_points//2)
        twiddle[k] = [exp(-2j*pi*k/N_points)*odd[k]

   return [even[k] + twiddle[k] for k in range(N_points//2)] + \
          [even[k] - twiddle[k] for k in range(N_points//2)]
```

III. Experiment Methodology

The algorithms were coded and implemented in Python. The two algorithms will be provided data sets of specifically generated signal sequences of varying sizes N. For each problem size, the number of operations performed and execution time were measured and recorded. The two algorithms also have their performance plotted for comparison and discussion. Each group member contributed to the development of the algorithms as well as the generation of input data and execution of the experiments, to be elaborated on further below.

Each algorithm will be implemented separately as two executable files. Each file will in principle function the same; reading a provided input, processing the input through a function that executes the desired algorithm, and calculating statistics regarding the performance of the algorithm execution. These statistics will include the number of operations performed as well as execution time of the overall procedure. This experiment will be repeated for both algorithms across several problem sizes. The input is a generated series of integers ranging in sizes of 2^10, 2^11, 2^12, etc. up to 2^15. An inverse algorithm for both the DFT and FFT are implemented with the goal of obtaining the original input data from the transformed data to verify the original

transform algorithm is performed correctly. It is anticipated that DFT algorithm will exhibit an O(N^2) complexity while the FFT will exhibit an O(N log(N)) complexity. Experimental points of particular interest include observations such as the problem size at which the FFT algorithm surpasses the DFT algorithm in efficiency and problem scales at which FFT is comparable or even worse than DFT. Experiments will be conducted in two environments: (1) an Ubuntu machine and (2) an Ubuntu virtual machine.

Four datasets were generated specifically for the experiment. These datasets represent the different problems that fourier transform algorithms may be applied. The four categories used in this experiment where Zeros (all data points numerically zero), Ones (all data points numerically one), Alternating -1 to 1 (a repeating pattern of -1, 1, -1, 1, etc.), and Cosine (a repeating pattern in the form of a cosine signal). The executable files are to be saved in the same directory as the master dataset folder and will search the folder for the data points to be used in the experiment based on user specification.

Two different testing environments were used, a 64-bit Ubuntu platform running Linux OS and a 64-bit Windows platform running a virtual machine of a 64-bit Ubuntu platform running Linux OS. The usage of two different environments was used to verify the accuracy of the results on more than one machine. Execution of the experiments involved running the files developed for the DFT algorithm and the FFT algorithm. Running either file prompts the user to specify one of the four dataset categories and the problem size (up to 2^15 or 32K data points). A folder containing the datasets must be saved in the same directory as the executable files.

The design of the experiment is modular such that it is trivial to insert or remove new dataset types or problem sizes. The addition of a new dataset simply requires the inclusion of a new folder containing files following the established naming convention “number.txt” where number may be a certain power of 2 such as 4K, 8K, 16K etc. The executable files will also require minor amendment to recognize the introduction or removal of data when prompting the user for experiment parameters.

The contributions of each group members are as follows: Yilin Yang was the developer of the DFT algorithm program, Siddharth Rupavatharam was the developer of the FFT algorithm program, and Peeyush Tambe was the developer the dataset generation program. In addition to their primary responsibilities, each group member contributed to the creation of the final project report, final presentation, and provided assistance for other group members’ primary responsibilities. The distribution of tasks for this project was agreed to be fair by all members.

IV. Results and Analysis

A total of 128 experiments were performed in this analysis. A single experiment consisted of the execution of either the DFT algorithm or FFT on a specific dataset and problem size. Four datasets (Zeros, Ones, -1 to 1, and Cosine) along with eight different possible problem sizes (ranging from 256 to 32 thousand, incremented by powers of two) provided a total of 32 different combinations of parameters to solve. Additionally, the two algorithms were implemented running on two different hardware platforms, offering four unique environments to perform each of the 32 unique experiments. Table 1 through Table 5 present a compilation of the results collected, dictating the execution time and number of operations performed in each experimental setup. An operation is counted as the number of time a complex multiplication is performed. Figure 5 through Figure 14 graph the results into a linear or logarithmic scale. Two figures, one of each scale, are associated with each of the four datasets along with one pair of figures for a comparison of the number of operations performed per problem size.

	Problem Size	DFT (Ubuntu VM)	DFT (Ubuntu)	FFT (Ubuntu VM)	FFT (Ubuntu)
256	2^8	0.12363	0.05373	0.0025	0.0067
512	2^9	0.29977	0.18792	0.0086	0.0125
1k	2^10	1.0552	0.70518	0.0204	0.013
2k	2^11	4.10548	2.81096	0.0503	0.0207
4k	2^12	15.2145	11.1803	0.1013	0.0371
8k	2^13	89.88514	50.29091	0.1536	0.0703
16k	2^14	395.60079	224.86547	0.33671	0.14658
32k	2^15	1265.86637	846.19161	0.7582	0.309

Table 1: Dataset of Zeros

	Problem Size	DFT (Ubuntu VM)	DFT (Ubuntu)	FFT (Ubuntu VM)	FFT (Ubuntu)
256	2^8	0.11174	0.04728	0.00218	0.0063
512	2^9	0.29339	0.181	0.00849	0.001
1k	2^10	1.12745	0.72603	0.02346	0.016
2k	2^11	3.75467	2.85524	0.04848	0.0234
4k	2^12	14.83544	11.36227	0.09605	0.0379
8k	2^13	97.58259	52.01844	0.17482	0.0708
16k	2^14	359.97122	230.09249	0.2375	0.1477
32k	2^15	1160.80973	855.50801	0.4751	0.3079

Table 2: Dataset of Ones

	Problem Size	DFT (Ubuntu VM)	DFT (Ubuntu)	FFT (Ubuntu VM)	FFT (Ubuntu)
256	2^8	0.1307	0.05571	0.00241	0.0054
512	2^9	0.29633	0.17978	0.00841	0.009
1k	2^10	0.96175	0.70308	0.026	0.017
2k	2^11	5.03086	2.77729	0.0589	0.0203
4k	2^12	22.94392	11.08887	0.09496	0.0407
8k	2^13	88.62576	52.68572	0.1955	0.0719
16k	2^14	356.47654	221.95255	0.2547	0.1522
32k	2^15	1204.3287	879.47403	0.4598	0.3054

Table 3: Dataset of Alternating -1 to 1

	Problem Size	DFT (Ubuntu VM)	DFT (Ubuntu)	FFT (Ubuntu VM)	FFT (Ubuntu)
256	2^8	0.10748	0.05634	0.0022	0.0062
512	2^9	0.28475	0.18265	0.0053	0.0038
1k	2^10	1.06186	0.70221	0.00949	0.0158
2k	2^11	4.9707	2.8	0.03841	0.0226
4k	2^12	16.20032	11.1536	0.09208	0.0397
8k	2^13	82.95861	50.31715	0.15306	0.0734
16k	2^14	353.26238	225.3965	0.2559	0.1484
32k	2^15	1454.08398	843.98541	0.4539	0.3219

Table 4: Dataset of Cosine

	Problem Size	DFT	FFT
256	2^8	65536	2048
512	2^9	262144	4608
1k	2^10	1048576	10240
2k	2^11	4194304	22528
4k	2^12	16777216	49152
8k	2^13	67108864	106496
16k	2^14	268435456	229376
32k	2^15	1073741824	491520

Table 5: Number of Operations per Problem Size

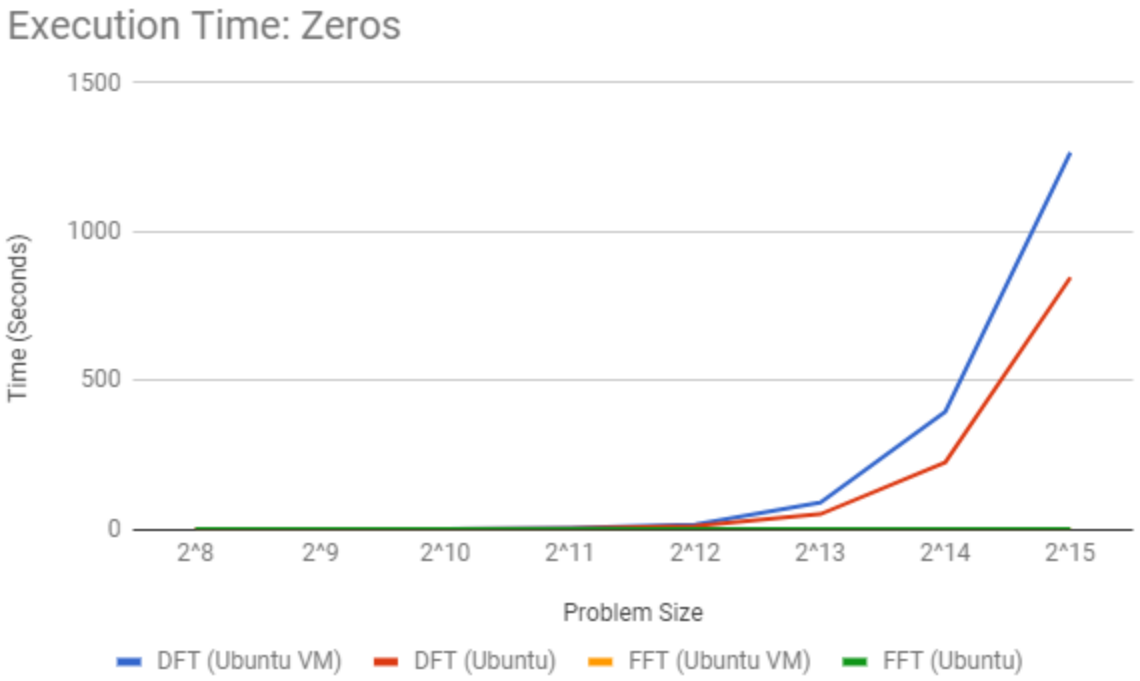

Fig 5: Linear scale representation of execution time for a dataset of zeros

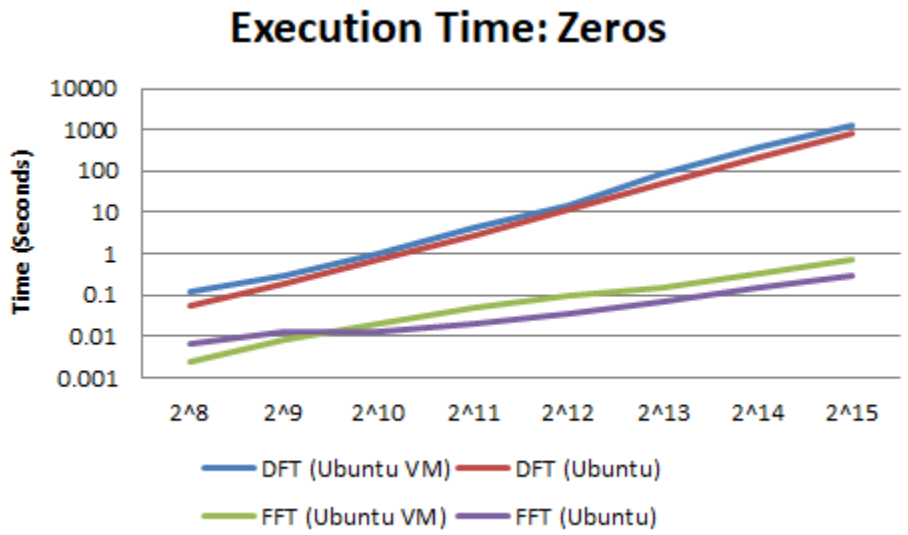

Fig 6: Logarithmic scale representation of execution time for a dataset of zeros

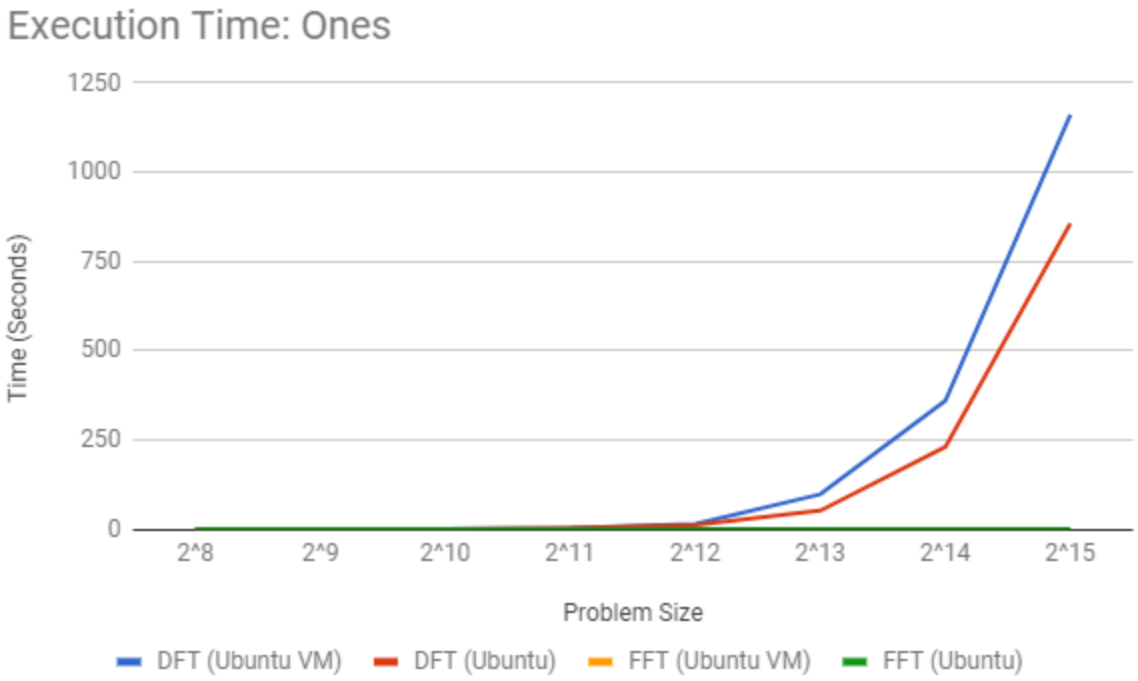

Fig 7: Linear scale representation of execution time for a dataset of ones

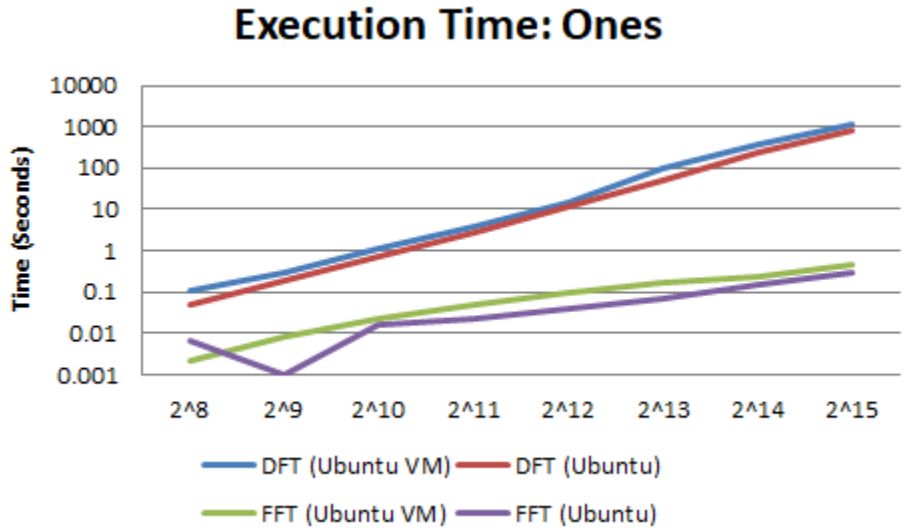

Fig 8: Logarithmic scale representation of execution time for a dataset of ones

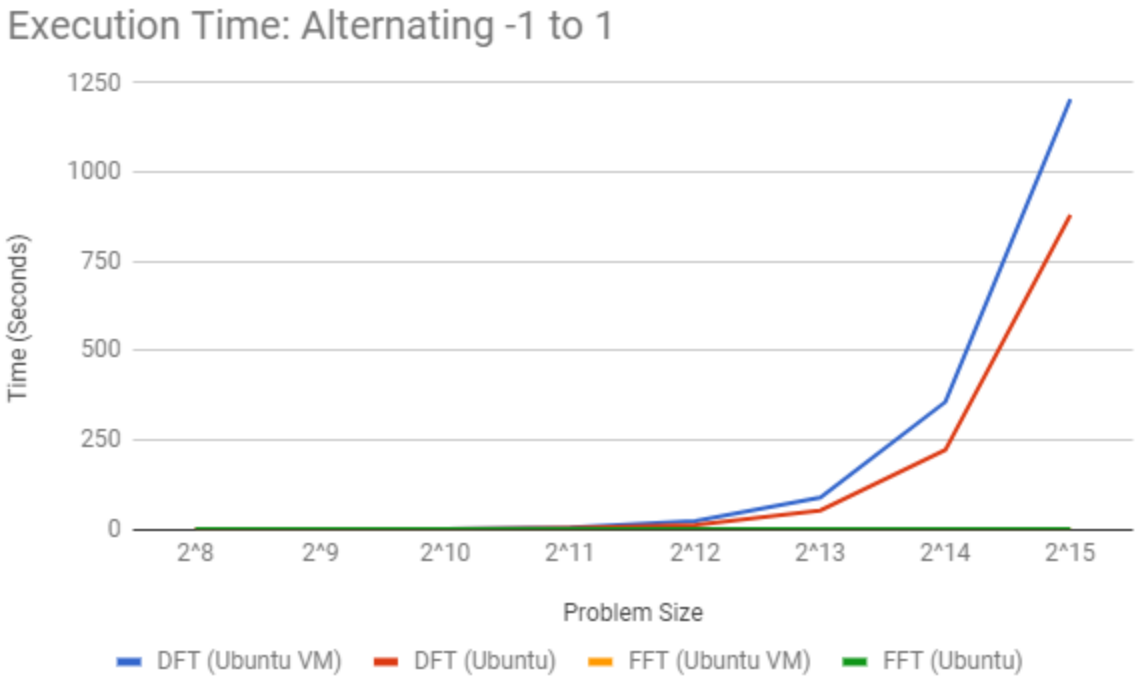

Fig 9: Linear scale representation of execution time for a dataset of Alternating -1 to 1

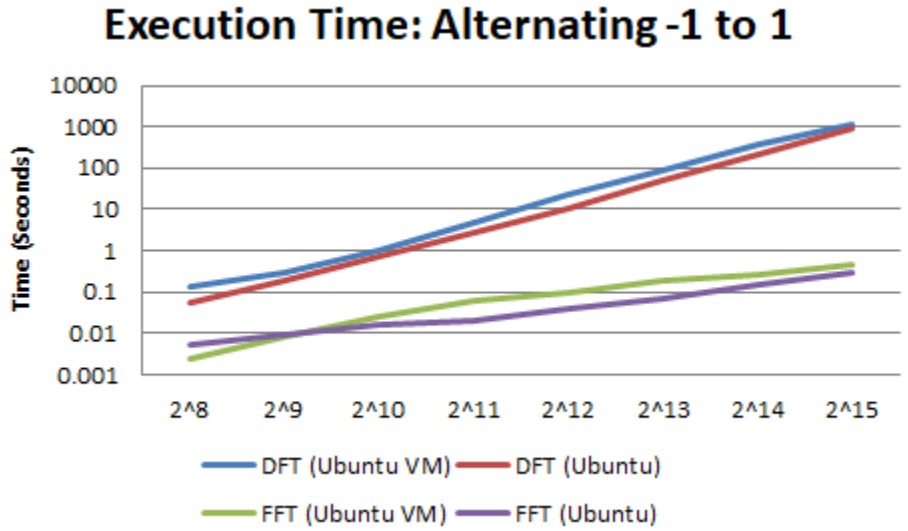

Fig 10: Logarithmic scale representation of execution time for a dataset of Alternating -1 to 1

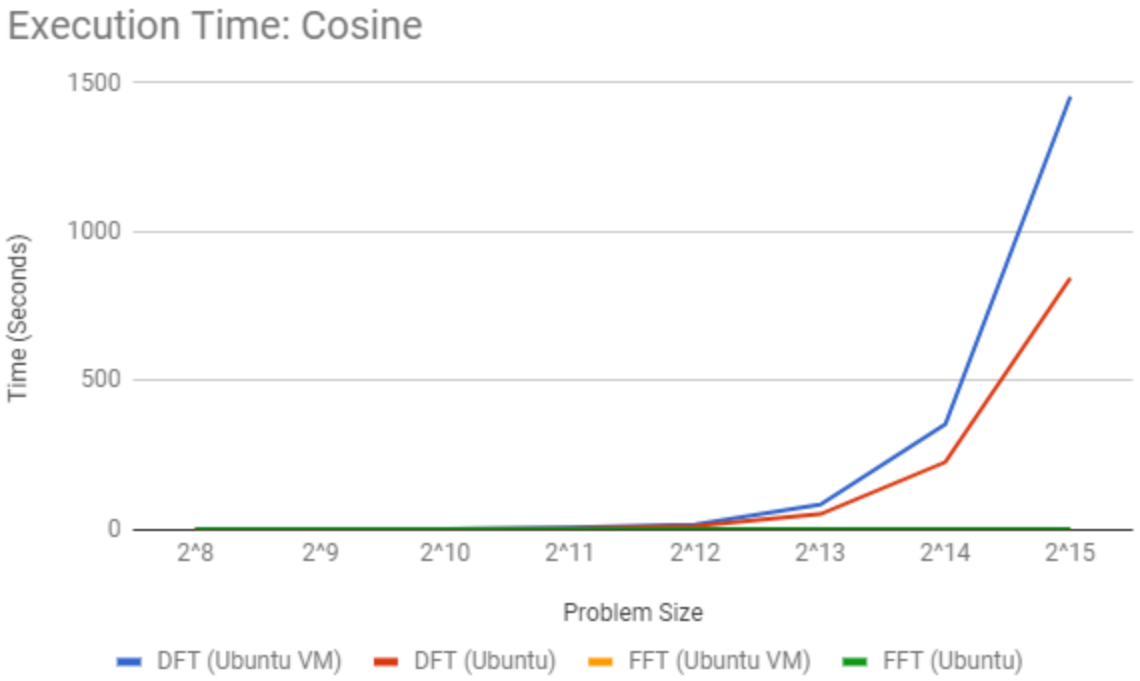

Fig 11: Linear scale representation of execution time for a dataset of cosine

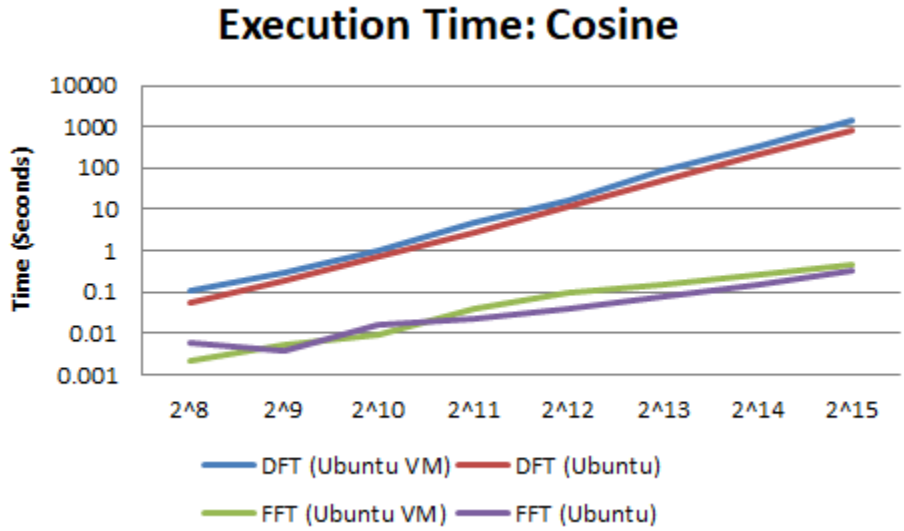

Fig 12: Logarithmic scale representation of execution time for a dataset of cosine

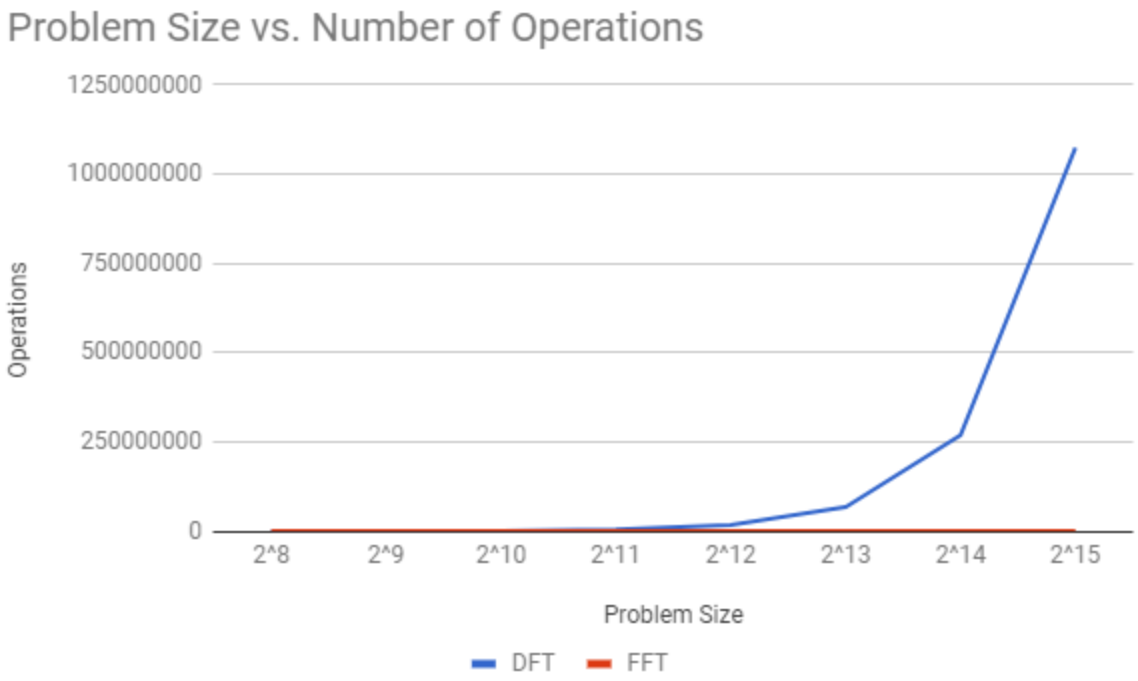

Fig 13: Linear scale representation of operations performed

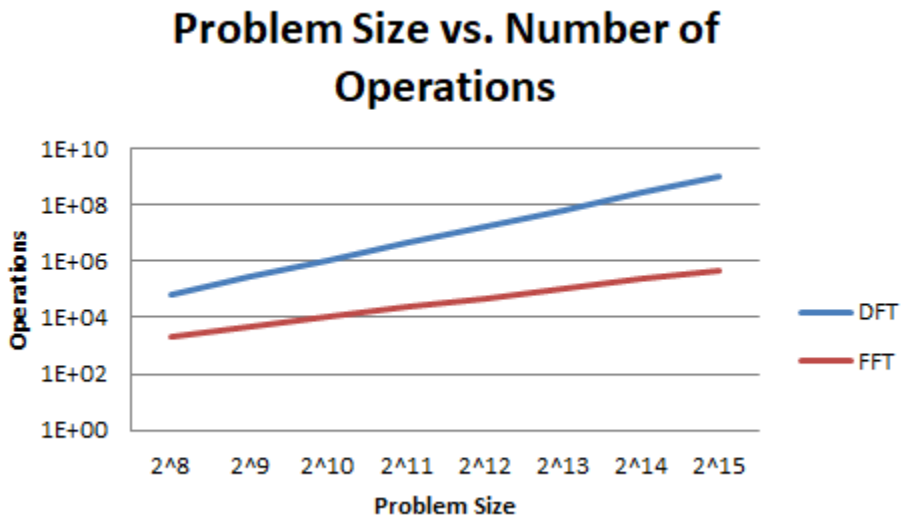

Fig 14: Linear scale representation of operations performed

V. Discussion

The superior efficiency of the FFT algorithm over the DFT algorithm was definitively confirmed in the results collected from the experimental procedure. In all scenarios tested, FFT execution time exceeded DFT by so wide a margin that it could not be accurately perceived on a linear scale. Only when adapted to a logarithmic scale could the speed up be easily conveyed. Across all four datasets, FFT consistently outpaced DFT. Speedups observed ranged from a magnitude of around 10 for smaller problem sizes (256) to over 1,000 for larger ones (32K). Similarly, the number of operations performed ranged from a magnitude of 100 times fewer for smaller problem sizes to approximately 1000 times fewer for larger ones.

The performance of the DFT algorithm is shown to be O(N^2) as the problem size directly correlates to the number of operations performed (ex. 256 data points requires 65536 or 256^2 operations). This is expected since DFT generates an output sequence of length N by calculating a summation on the input sequence of length N for every element of the output. A single summation alone is of order O(N) if it includes the entirety of the input sequence. In other words, DFT must iterate through a list of N elements N times, hence the complexity order of O(N^2). In contrast, the FFT demonstrates a performance complexity of N log(N) (ex. 256 data points requires 2048 or 256*log base 2(256) operations). As a divide-and-conquer strategy, FFT recursively breaks down the problem size in half and operates on them separately. Specifically, the problem size is divided based on even and odd elements on the input sequence, which is then recursively operated on until a base case of size 1 is reached. Similar to merge sort strategies, it requires log(N) divisions of a sequence of length N to reach this base case. Since FFT is computing a sequence of length N, it is performing this strategy N times, hence the O(N log(N)) complexity.

VI. Conclusion

In this project, we proposed an investigation into the performance complexities of two popular algorithms in signal processing systems. These include the Discrete Fourier Transform algorithm (DFT) and the Fast Fourier Transform algorithm (FFT). The DFT is a fundamental tool for the conversion of a problem sample from one domain into another. Specifically, it converts a signal sequence from the frequency domain into the time domain, greatly simplifying the difficulty of calculations needed to be performed when solving frequency analysis, partial differential equations, convolutions, and other related mathematical properties. Closely related to the DFT is the FFT, which can compute the same transformation at a drastically reduced time. The FFT takes a recursive approach to the transformation, breaking down the input sequence into smaller components to be solved.

The two algorithms were implemented using python and used to process sequences of gradually increasing sizes. Sequences were generated ranging from sizes of 2^8 to 2^15 for different types of datasets. These sequences were processed through the two algorithms and the performance was observed. Two criteria were examined during simulations; the number of operations performed and the execution time. From our findings, we confirm the complexity of the DFT and the FFT to be O(N^2) and O(N log(N)), respectively. While useful, the complexity of the DFT transformation is intensive, requiring N operations to be performed for every element in the input sequence of length N. The FFT is able to reduce this by breaking the sequence into smaller components before operating on them. The input sequence, being halved recursively, requires only log(N) operations per element compared to the DFT's N operations per element. This results in a reduced complexity which we have demonstrated to reduce computational run times by magnitudes in the thousands for sufficiently large problem sizes.

VII. References

[1] Baron Fourier, Jean Baptiste Joseph. *The analytical theory of heat*. The University Press, 1878.

[2] Crochiere, R.E.; Rabiner, L.R. (1983). Multirate Digital Signal Processing. Englewood Cliffs, NJ: Prentice Hall. pp. 313–326. ISBN 0-13-605162-6.

[3] Oppenheim, Alan V.; Schafer, Ronald W. (1999). *Discrete-Time Signal Processing* (2nd ed.). Prentice Hall Signal Processing Series. ISBN 0-13-754920-2.

[4] Cooley, James W., and John W. Tukey. "An algorithm for the machine calculation of complex Fourier series." *Mathematics of computation* 19.90 (1965): 297-301.

[5] http://www.skm-eleksys.com/2010/10/fourier-series-in-electrical.html

[6] https://aavos.eu/glossary/fourier-transform/