FFT Fourier Project

KATHRYN GRAY, KSENIA LEPIKHINA, AMELIA WESTERDALE

1 INTRODUCTION

The Fast Fourier Transform (FFT) was created as an optimized way to calculate the Discrete Fourier Transform (DFT) of a finite sequence of numbers. [7] For years the FFT was considered the most important algorithm in engineering and applied sciences. The FFT is especially effective and important in one and multidimensional systems theory and signal processing. In this paper we outline the history of the Fast Fourier Transform, its mathematical background, various applications of the FFT, comparable algorithms to the FFT, and its setbacks. The mathematics of the DFT and the FFT will be stated and explained, and then an experiment will be conducted to demonstrate an application in detail.

2 HISTORY

In 1805, Carl Friedrich Gauss was working on determining the orbit of asteroids in certain locations. Through the process, he developed the DFT 17 years before Fourier even published his results. [18]. Gauss was influenced by the analysis of trigonometric series of Euler (1707-1783), the first iteration of the DFT (restricted to finite cosine Fourier Series) by Clairaut (1713-1813) and the second iteration of the DFT (restricted to finite sine Fourier Series) by LaGrange. Clairaut and Lagrange were focused on "orbital mechanics and the problem of determining the details of an orbit from a finite set of observations." [7] The two used interpolation for orbit determination. Consequently, Gauss extended this from trigonometric interpolation to periodic functions that are not necessarily odd or even.

Principal Discoveries of Efficient Methods of Computing the DFT

Researcher(s)	Date	Sequence Lengths	Number of DFT Values	Application
C. F. Gauss [10]	1805	Any composite integer	All	Interpolation of orbits of celestial bodies
F. Carlini [28]	1828	12	—	Harmonic analysis of barometric pressure
A. Smith [25]	1846	4, 8, 16, 32	5 or 9	Correcting deviations in compasses on ships
J. D. Everett [23]	1860	12	5	Modeling underground temperature deviations
C. Runge [7]	1903	$2^n k$	All	Harmonic analysis of functions
K. Stumpff [16]	1939	$2^n k$, $3^n k$	All	Harmonic analysis of functions
Danielson and Lanczos [5]	1942	2^n	All	X-ray diffraction in crystals
L. H. Thomas [13]	1948	Any integer with relatively prime factors	All	Harmonic analysis of functions
I. J. Good [3]	1958	Any integer with relatively prime factors	All	Harmonic analysis of functions
Cooley and Tukey [1]	1965	Any composite integer	All	Harmonic analysis of functions
S. Winograd [14]	1976	Any integer with relatively prime factors	All	Use of complexity theory for harmonic analysis

Fig. 1. Discoveries of computational methods for calculating DFT and their applications [7]

Gauss's compiled work is titled "Theoria Interpolationis Methodo Nova Tractata". Gauss's work with the Discrete Fourier Transform was not published while he was alive but rather remained unpublished as a part of a collected work. The year 1805 is the presumed composition date of his treatise. This predates Fourier's work with harmonic series by two years. Parts of this treatise outline trigonometric interpolation algorithms. The articles were written in neo-Latin and were difficult to translate because Gauss used unusual notation (i.e. π instead of n as the length of a sequence, a, a', and a" as indices of time series, etc.). [7] In this treatise, Gauss outlined a form of what is now called the DFT. In order to calculate the DFT, Gauss invented an algorithm which would now be recognized as the FFT, but did not recognize its importance at the time. [18]

In 1965, J. W. Cooley and J. W. Tukey were able to optimize the DFT algorithm (similarly to Gauss's work) and created the FFT. It was later found that the two authors independently rediscovered the FFT that Gauss had discovered nearly a century and a half ago. Formerly, it was thought that the DFT required N^2 arithmetic operations, but Cooley and Tukey were able to prove that the new FFT algorithm could be completed in $NlogN$ arithmetic operations. In Cooley and Tukey's paper, they claimed their work was influenced only by I.J. Good's paper, "The Interaction Algorithm and Practical Fourier Analysis" written in 1958 [7], however Cooley-Tukey's algorithm was labeled the FFT and Good's algorithm was labeled the Prime Factor Algorithm (PFA). The difference between DFT and PFA are that the DFT is able to work with any composite integer sequence length and the PFA is able to work only with any integers with relatively prime factors as shown in Figure 1. This paper will not elaborate on the PFA, but will instead focus on the DFT and it's implementation, specifically, the FFT.

3 MATHEMATICAL BACKGROUND: WHAT IS A FFT? HOW DOES IT WORK?

3.1 Discrete Fourier Transform

Often, questions in mathematics are difficult to solve by hand. Such problems are better solved with computers. One such problem that often arises is the Fourier Transform. However, because computers can only deal with discrete values, an approximation to the Fourier Transform is necessary, hence the Discrete Fourier Transform is used. The DFT approximates the Fourier Transform and is suitable for computation. To show how the DFT is derived, let $f(t)$ be the original function and $\hat{f}(k)$ be the Fourier Transform. This derivation is reformatted and expanded upon from the derivations from [2] and [6].

3.1.1 Derivation of the DFT. The first step is to sample $f(t)$, because we can only hold a finite amount of values, a sampling function is necessary. This sampling function will be described more in the issues section as there are different forms, but for now, let $\Delta(t)$ be the sampling function, and for simplicity, assume this function takes samples every T seconds. Also, note that δ is the

dirac delta function, so Δ samples by repeated use of the δ function.

$$
\begin{aligned}
f(t)\Delta(t) &= f(t)\sum_{n=-\infty}^{\infty}\delta(t-nT) \\
&= \sum_{n=-\infty}^{\infty} f(nT)\delta(t-nT)
\end{aligned}
$$

The last line comes from an identity that we will not show here.

Then, since the function is assumed to be periodic, we can construct an interval that will encapsulate the function. Let this be given as $-\frac{T}{2} < t < T_0 - \frac{T}{2}$, where T_0 is the period. This interval is chosen to help avoid aliasing (an issue discussed in Section 8). Then, assuming the sampling function gives N samples in the interval, we can write the function again as follows:

$$
\sum_{n=0}^{N-1} f(nT)\delta(t-nT)
$$

Then, if we wanted a total approximation of $f(x)$, we arrive at the following, because we merely repeat what was found before across the whole axis. This gives the m summation, with offsets given by

$$
f(x) \approx \sum_{m=-\infty}^{\infty} T_0 \sum_{n=0}^{N-1} f(nT)\delta(t-nT-mT_0)
$$

Now we can set up the Fourier Transform of the approximation to $f(x)$. We will take one period of the function, but this will extend to all of them. However, this means that we can deal only with the n summation.

$$
\begin{aligned}
\hat{f} &\approx \int_{-\frac{T}{2}}^{T_0-\frac{T}{2}} \sum_{n=0}^{N-1} f(nT)\delta(t-nT)e^{-ikt}dt \\
&= \sum_{n=0}^{N-1} f(nT) \int_{-\frac{T}{2}}^{T_0-\frac{T}{2}} \delta(t-nT)e^{-ikt}dt \\
&= \sum_{n=0}^{N-1} f(nT)e^{-ikT}
\end{aligned}
$$

3.1.2 Inverse DFT. The Inverse Discrete Fourier Transform follows a similar proof, so we will state it below, but omit the derivation.

$$
f(x) =\approx \frac{1}{N}\sum_{n=0}^{N-1} \hat{f} e^{i2\pi nx/N}dk
$$

3.1.3 DFT Naive Algorithm. Since we motivated the DFT with approximating the Fourier Transform using a computer, an algorithm should be put forward to solve this. The first algorithms to attempt this were implemented without taking advantage of any properties of the Fourier Transform. This led to an algorithm, that although simpler to understand, has a run time of $O(N^2)$.

To solve the DFT, we set up matrices to solve, for ease of notation, let $\omega = e^{2\pi i/N}$

$$
\begin{bmatrix}
\omega^0 & \omega^0 & \omega^0 & \dots & \omega^0 \\
\omega^0 & \omega^1 & \omega^2 & \dots & \omega^{N-1} \\
\omega^0 & \omega^2 & \omega^4 & \dots & \omega^{2(N-1)} \\
\omega^0 & \omega^3 & \omega^6 & \dots & \omega^{3(N-1)} \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
\omega^0 & \omega^n & \omega^{2n} & \dots & \omega^{n(N-1)}
\end{bmatrix}
\begin{bmatrix}
f(0) \\ f(1) \\ f(2) \\ f(3) \\ \vdots \\ f(N-1)
\end{bmatrix}
=
\begin{bmatrix}
\hat{f}_0 \\ \hat{f}_1 \\ \hat{f}_2 \\ \hat{f}_3 \\ \vdots \\ \hat{f}_{N-1}
\end{bmatrix}
$$

Then, solve for the $\hat{f}$ values, each one will take N multiplications and $N-1$ additions (this is accounts for each row). Then there are N rows and we must solve for each of the $\hat{f}$, giving a naive implementation taking $O(N^2)$ operations to complete. While this algorithm works, a faster algorithm is preferable, especially for large N.

Storing the DFT Matrix can be efficient when factorizing it into "a short product of sparse matrices" that can be represented by fewer numbers than its size because much of their entries would be zero; only the non-zero entries and their locations need to be stored [16]. Not only is the DFT matrix square, it is invertible. This follows that the inverse DFT can be computed. Likewise, there exists an Inverse FFT.

3.2 Fast Fourier Transform

The Fast Fourier Transform takes advantage of the symmetries in the DFT to have a much faster algorithm. First, we must show that the transform is periodic.

$$
\begin{aligned}
\hat{f}(m+N) &= \sum_{n=0}^{N-1} f(n)e^{-i2\pi n(m+N)/N} \\
&= \sum_{n=0}^{N-1} f(n)e^{-i2\pi n(m/N+1)} \\
&= \sum_{n=0}^{N-1} f(n)e^{-i2\pi nm/N)}e^{-i2\pi n)} \\
&= \sum_{n=0}^{N-1} f(n)e^{-i2\pi nm/N)} \\
&= \hat{f}(m)
\end{aligned}
$$

The Fast Fourier Transform is a recursive algorithm used to compute the Discrete Fourier Transform, but with an $O(NlogN)$ runtime complexity. By the Danielson-Lanczos Lemma, if N is a power of 2, the DFT, approximated as the sum $\sum_{n=0}^{N-1} f(nT)e^{-ikT}$, can be split into two different summations of half the length (we are using the definition that $\omega = e^{2\pi i/N}$ again):

$$
\begin{aligned}
\sum_{n=0}^{N-1} f(n)\omega^{kn} &= \sum_{n=0}^{N/2-1} f(2n)\omega^{2kn} + \sum_{n=0}^{N/2-1} f(2n+1)\omega^{k(2n+1)} \\
&= \sum_{n=0}^{N/2+1} f(2n)\omega^{2nk} + \omega^k \sum_{n=0}^{N/2-1} f(2n+1)\omega^{2kn} \\
&= \hat{f}_1 + \omega^k \hat{f}_2
\end{aligned}
$$

The last line shows the power of this decomposition. The original problem has now been split into two different versions of the same problem. This gives a runtime of $O(N \log N)$, since each split must be evaluated, giving the N term and the splitting can be shown to give the $\log(N)$ term.

If N is not a power of two, then this splitting can be continued until the sum consists of one data point.

The FFT can be performed on subsets of points with lengths corresponding to the prime factors of N. This results in a slower, yet asymptotically identical, run-time.

A nice visualization of how the Cooley-Tukey algorithm for calculating the Fast Fourier Transform can be seen in the butterfly diagram in Figure 2.

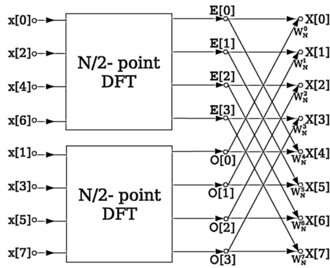

Fig. 2. Butterfly diagram of Radix-2 FFT algorithm [3]

The algorithm breaks the length N DFT into two N/2 length DFTs by splitting up the even and odd indices. It then combines the results through multiple butterfly operations. This method of splitting and combining saves a factor of two in complex multiplies.

3.3 Is it an actual transform?

The mathematical definition of a transform is a function that maps to itself [17]. As discussed, many real functions are mapped to the complex Fourier space and then transformed back into the original space with inverse FFT. The result would be complex, as opposed to real, and only the real part is considered. On the other hand, the DFT is a discretization of a continuous function–applying the inverse to the transformed function would yield a signal. Therefore, the DFT is not a proper transform under many mathematical definitions.

4 APPLICATIONS OF THE FAST FOURIER TRANSFORM

There are various applications for the Fast Fourier Transform. Some areas include applied mechanics (aircraft wing flutter suppression), sonics and acoustics (architecture acoustic measurement), bio- medical engineering (diagnostics of airway obstruction), signal processing (speech synthesis and recognition), instrumentation (microscopy), and communication (speech scrambler system). [2]

A specific application of the FFT is the deblurring (deconvultion) of an image. To deblur an image, it is necessary to have the blurred image itself and the Point Spread Function (PSF) of the original image. The PSF "describes the response of an imaging system to a point source [single identifiable localized source] or point object" [11]. In other words, the point spread function essentially describes how "blurred" an image really is. It describes the quality of an imaging system. Fast Fourier Transforms play a role in blurring like so:

$$FFT(\text{clean image}) + FFT(\text{point spread function}) = FFT(\text{blurred image})$$

so, in order to get the clean image, the equation comes [5]:

$$\text{clean image} = FFT^{-1}\left[\frac{FFT(\text{blurred image})}{FFT(\text{point spread function})}\right]$$

Since the image is represented as a matrix, then using the Fast Fourier Transform to find the clean image is much faster than the Discrete Fourier Transform. The visualization can be seen in Figure 4 below.

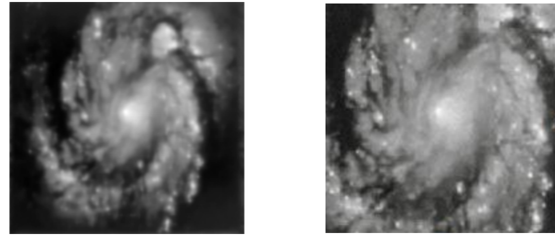

Fig. 3. Blurred galaxy photo [5] Fig. 4. Less blurry galaxy photo [5]

It's faster because the matrix can be broken down into multiple matrices with lots of zero entries. On each of those smaller matrices, the DFT can be applied. Being able to break down a matrix in such away, allows for the number of operations to go down from O(N^2) to O($NlogN$). [9]

Another application of the Fast Fourier Transform is in forensics. When looking at fingerprints for example, the background to the image of the print might make it difficult to analyze.

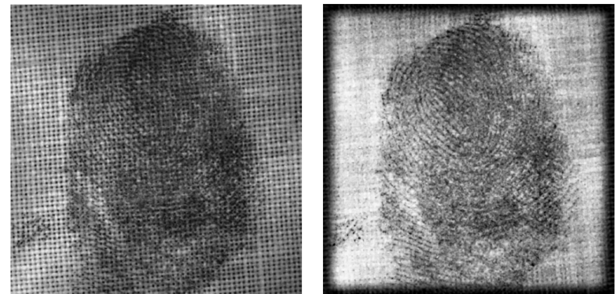

Fig. 5. Original print[10] Fig. 6. Cleaned up print [10]

In this case, the Fast Fourier Transform helps remove the ridges from the background of the original image. "The FT has peaks at spatial frequencies of repeated texture." [10] In Figure 5, the Fast Fourier Transform helps remove the texture of the background by removing the periodic background.

5 EXPERIMENTS

The Fast Fourier Transform has many implementations. The Cooley-Tukey implementation is typically known as the Radix-2 DIT (decimation in time) FFT. Paired with this algorithm is the Radix-2 DIF (decimation in frequency) FFT (Sande-Tukey algorithm) which we

will not elaborate on in this paper. The appendix of this paper includes a MATLAB implementation of the Cooley-Tukey implementation of the Fast Fourier Transform. This version of the algorithm implements a divide and conquer strategy that recursively splits a DFT of size N into multiple smaller DFTs. [4] The function we implemented takes an array of values and returns the DFT of the array of values using the FFT. The FFT implementation applies when the length of the array is an integer power of 2. The power of 2 length assumption is not necessary but allows for a simpler implementation by hand. An example of the FFT of an array look something like so:

```
fft([1 2 3 4 5 6 7 8])
ans = 36 -4+9.6569i -4+4i -4+1.6569i -4 -4-1.6569i -4-4i
     -4-9.6569i
```

If we wanted to apply the Fast Fourier Transform to a matrix of size $[2^n, 2^n]$ where n is any positive integer, then an example of our `fft_matrix` function would look like so:

```
fft_matrix([1 5 9 13; 2 6 10 14; 3 7 11 15; 4 8 12 16])
ans = 10 26  42  58
 -2+2i  -2+2i  -2+2i  -2+2i
 -2  -2  -2  -2
 -2-2i  -2-2i  -2-2i  -2-2i
```

A quick example of how the FFT works on an image can be seen below. This example makes use of MATLABs `mat2gray` function as well as the three functions in the appendix of this paper.

```
A = [1 5 9 13; 2 6 10 14; 3 7 11 15; 4 8 12 16]
I = mat2gray(A)
disp(I)
>> 0    0.2667    0.5333    0.8000
   0.0667    0.3333    0.6000    0.8667
   0.1333    0.4000    0.6667    0.9333
   0.2000    0.4667    0.7333    1.0000
FFTI = fft_matrix(I)
disp(FFTI)
>> 0.4   1.467   2.533   3.6
 -0.133+0.133i -0.133+0.133i -0.133+0.133i -0.133+0.133i
 -0.133  -0.133  -0.133  -0.133
 -0.133-0.133i -0.133-0.133i -0.133-0.133i -0.133-0.133i
FFTBack = fft_inverse_ksenia_matrix(FFTI)
disp(FFTback)
>> 0    0.2667    0.5333    0.8000
   0.0667    0.3333    0.6000    0.8667
   0.1333    0.4000    0.6667    0.9333
   0.2000    0.4667    0.7333    1.0000
imshow(I)
imshow(FFTI)
imshow(FFTBack)
```

and the three `imshow` commands return first the original gray scale image, then the FFT of image I, FFTI (note MATLAB does not plot complex numbers), and finally, the inverse FFT of FFTI, FFTBack.

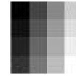

Fig. 7. Image

Fig. 8. FFT of Image

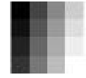

Fig. 9. Inverse FFT

After computing the FFT, we are essentially transforming from a function of time to a function of frequency. Many of those values are converted to the complex plane. Taking the inverse Fast Fourier Transform, we are returning back from the frequency domain to the time domain.

An example of how the Fast Fourier Transform changes an image from the time space to the frequency space follows below. The image was taken from `https://amath.colorado.edu/faculty/segur/` and was converted to black and white. The image was cropped and padding was added in order to create a matrix of size $2^n x 2^n$.

Fig. 10. I = Image

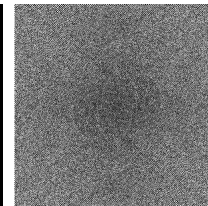

Fig. 11. F = FFT(I)

Fig. 12. FFT^{-1}(F)

Note that this also works if you do the inverse first.

Fig. 13. I = Image

Fig. 14. F2 = FFT^{-1}(I)

Fig. 15. FFT(F2)

The first transform of the original image is the Fast Fourier Transform. This is essentially assuming that the first image is a time matrix and the FFT converts it to a frequency matrix. When we take the inverse FFT of the new frequency matrix, we get the original time matrix back.

In the second group of images, we start by taking the inverse Fast Fourier Transform first. When we do so, we are assuming that the original matrix is a matrix of frequencies and converting it to a time matrix. Taking the FFT of the time matrix converts it back to a frequency matrix.

6 FFT COMPETITORS

6.1 Image Compression Competitors

One of the applications of a Fast Fourier Transform is image compression. A comparable competitor for the FFT for image compression is the Discrete Cosine Transform (DCT).The DCT is preferred over the FFT in this case because it "incorporates more information from the image in fewer coefficients." [8] Unlike the FFT, the DCT assumes an even extension outside of the domain on which we have a sample which means that the function is continuous on the

boundary. When images are compressed using the FFT, the quality of the image deteriorates significantly more than the DCT when for both 80% of the coefficients are discarded. Both the DFT and the DCT decompose a discrete time vector into a sum of scaled and shifted basis functions, however the DFT uses complex exponential functions and the DCT uses real valued cosine functions. Since the DFT works with complex numbers and the DCT works with real numbers and audio and image data are usually represented with real numbers, the natural choice for working with audio and image compression is the DCT. So in this case, the FFT is less accurate than the DCT.

6.2 DFT Competitors

When looking for only a few frequencies, the Goertzel algorithm[14] can be used. This algorithm takes less time than the FFT when the number of frequencies is less than $\log(N)$. The Goertzel algorithm takes $O(N)$ time to find one frequency. The algorithm also has no restrictions on how many samples are given, so it does not need to be a multiple of 2^n (n an integer).

Another algorithm is Bluestein's Fast Fourier Transform or the Chirp-Z Transform (CZT) is used to find some of the spectral frequencies over some range[15]. While the complexity of CZT is larger than the FFT, the CZT is sometimes used as it has better resolution than the FFT. The equation for the CZT is found through convolutions of the DFT.

7 PROBLEMS WITH FFT

7.1 Sampling

One of the issues that may arise when using the DFT to approximate the Fourier Transform is how the function is sampled and how many samples are taken. A lower bound on the amount of samples that is necessary has been given as 2 times the bandwidth[1] (bandwidth being the difference between the upper and lower frequencies). This is known as the Nyquist sampling rate. If the sampling rate is lower than this, the samples could come from different initial functions, adding uncertainty that the solution is correct. Sampling is such an important topic, we cannot cover everything that has been formulated on it here.

7.2 Aliasing

Aliasing can arise when the function is undersampled. Aliasing refers to having two solutions for given sample points or having multiple frequencies covered by one data point. When the sampling rate is too low, the larger frequencies "fold back" on itself, which leads to another name this can go by, spectral folding. Often, when using real data, it is not possible to take enough samples to fulfill the Nyquist sampling rate. In this case, care must be taken so the important frequencies to the problem are still sampled well enough. In general, a solution will have any number of solutions at $2\pi n$ higher frequency. To mitigate this problem, FFT and other DFT solvers will use the lowest one.

7.3 Multidimensional FFT

The Fast Fourier Transform does not necessarily scale well for higher dimensions. In this paper we have mentioned how the FFT is an accurate approximation of the Fourier Transform for one dimension and for two dimensions (matrices). The two dimensional version of the FFT has an asymptotic run time of $O(N^2\log(N))$. However, when we scale to n-dimensions, there are some runtime problems. Although the FFT is surprisingly stable [12], for high dimensional FFTs (N-D FFT) "complexity increases with an increase in dimension, leading to costly hardware and low speed of system reaction." [13] The runtime when scaling to n-dimensions becomes $O(N^n log(N))$. In general, the FFT can only reduce the runtime along one dimension.

8 CONCLUSION

The Fast Fourier Transform is held in high regards for 20th and 21st century applied sciences, and is used in fields ranging from image processing to bio-medical engineering. This algorithm modernizes the useful Discrete Fourier Transform, invented by mathematicians as early as the 1700s, so that larger amounts of data can be computed and manipulated. There are nearly countless applications of the FFT, but a simple, yet effective, experiment on image processing was provided to demonstrate not only its usefulness, but its substantial accessibility through popular computing environments, such as MATLAB. Although there are competitors and setbacks to the FFT, the creation of a divide-and-conquer algorithm that improves the run-time complexity of the DFT is impressive and remains employed to scientists, engineers, and mathematicians today.

9 APPENDIX

The first and most important function we wrote was the `fft` function. This function takes an array or vector of size 2^n and computes the Fast Fourier Transform and returns the resulting array or vector.

```
function [y] = fft(x) % function for calculating fft of
    array or vector
    len = (length(x)); % find length of array or vector
    half = ceil(len/2);
    exponential = exp(-2 * pi * 1i / len) .^ (0 : half -
        1); % fft exponential piece
    if len == 1 % if the length of the array is 1, the FFT
        = array
        y = x;
    else
        y1 = fft(x(1: 2 : (len - 1))); % recursively call
            on odd incides
        y2 = fft(x(2: 2 : (len))); % recursively call on
            even indices
        y = [y1+exponential .* y2,y1-exponential .* y2];
            %create results
    end
end
```

The second function we wrote is `fft_matrix`. This function takes a matrix of size 2^nx2^n and returns the Fast Fourier Transform of it which will also be a matrix of size 2^nx2^n. This algorithm will run in O(N^2log N) time because it is calling `fft` for each column in the

matrix. Since fft runs in O(N log N) time, this one runs in O(N^2 log N) time since fft is called n times.

```
function [V] = fft_matrix(x) % function for calculating
     fft of matrix
    ncol = size(x,2); % get size of 2^n x 2^n matrix
    V = zeros(ncol,ncol); % pre allocate empty matrix for
         result
    for k = 1:ncol % iterate through each column and call
         fft
       result = fft(x(:,k));
       V(:,k) = result; % add column to V
    end
end
```

This final algorithm, fft_inverse_matrix was written to calculate the inverse Fast Fourier Transform of a 2^nx2^n matrix. The algorithm follows an example on page 3 in: [9].

```
function [V] = fft_inverse_matrix(x) % function for
     finding inverse fft of matrix
  y = conj(x); %take the complex conjugate of all values
       in x
  temp = fft_matrix(y); % call fft_matrix on the conjugate
       matrix
  c = conj(temp); % take the conjugate again
  for k = 1:size(c,2) % call on each column
      V(:,k) = c(:,k)/size(c,2); % normalize
  end
end
```

REFERENCES

[1] Anders Brandt and Kjell Ahlin. 2010. Sampling and time-domain analysis. *Sound and Vibration*, 44, 5, 13.

[2] E. Oran Brigham. 1988. *The Fast Fourier Transform and its Applications*. Prentice-Hall, Inc., New Jersey, US.

[3] [n. d.] Butterfly diagram. https://en.wikipedia.org/wiki/Butterfly_diagram. Accessed: 2018-12-05. ().

[4] [n. d.] Cooley–tukey fft algorithm. https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm. Accessed: 2018-12-04. ().

[5] Earl F. Glynn. 2007. Fourier analysis and image processing. Stowers Institute for Medical Research. (February 14, 2007). Retrieved 12/01/2018 from http://research.stowers.org/mcm/efg/Report/FourierAnalysis.pdf.

[6] J. Fessler. [n. d.] Digital signal processing and analysis. (). https://web.eecs.umich.edu/~fessler/course/451/l/pdf/c5.pdf.

[7] 1984. Gauss and the history of the fast fourier transform. *IEEE ASSP Magazine*, (October 1984).

[8] [n. d.] Image compression using fourier techniques. http://www.maths.usyd.edu.au/u/olver/teaching/Computation/ExampleProject.pdf. Accessed: 2018-12-03. ().

[9] [n. d.] MIT Lecture 26: Complex Matrices; Fast Fourier Transform. URL: https://ocw.mit.edu/courses/mathematics/18-06sc-linear-algebra-fall-2011/positive-definite-matrices-and-applications/complex-matrices-fast-fourier-transform-fft/MIT18_06SCF11_Ses3.2sum.pdf. Last visited on 2018/12/03. ().

[10] [n. d.] Oxford Lecture 2: 2D Fourier transforms and applications. URL: http://www.robots.ox.ac.uk/~az/lectures/ia/lect2.pdf. Last visited on 2018/12/05. ().

[11] [n. d.] Point source function. https://en.wikipedia.org/wiki/Point_spread_function. Accessed: 2018-12-01. ().

[12] James C. Schatzman. 1996. Accuracy of the discrete fourier transform and the fast fourier transform. *SIAM Journal of Scientific Computing*, 17, 5, 1150–1166.

[13] Athina Petropulu Shaogang Wang Vishal M. Patel. 2016. An Efficient High-Dimensional Sparse Fourier Transform. Technical report. IEEE.

[14] [n. d.] Single tone detection with the goertzel algorithm. (). https://www.embedded.com/design/real-world-applications/4401754/Single-tone-detection-with-the-Goertzel-algorithm.

[15] S. Sirin. 2003. Czt vs fft: flexibility vs speed. (2003). https://www.osti.gov/servlets/purl/816417.

[16] [n. d.] The fft via matrix factorizations. https://www.cs.cornell.edu/~bindel/class/cs5220-s10/slides/FFT.pdf. Accessed: 2018-12-06. ().

[17] Graham Wilkinson Leland. 2005. *The Grammar of Graphics (2nd ed) p. 29*. Springer.

[18] Stefan Worner. [n. d.] Fast Fourier Transform. Technical report. Swiss Federal Institute of Technology Zurich.