# The DFT and FFT, Program 1

Colton Piper

February 5th, 2018

## 1 Executive Summary

This program discusses about eigen-decomposition for certain linear transformations more specifically being the up-shift matrix and circulant matrices. Then using what we learn in the discussion we will show a more efficient algorithm to solve the $C_n x = b$. We will also be implementing the Discrete Fourier Transform and Fast Fourier Transform and using them in the 2D-Discrete Fourier Transform. Then lastly we will be using our algorithms to solve $C_n x = b$ to use in image processing.

## 2 Statement of the Problem

First we will show that the eigen-decomposition for the permutation matrix for up-shifting a vector ($Z_n$). Then we use our conclusions from that to show the eigen-decomposition of any circulant matrix $C_n$. Finding the eigen-decomposition for the circulant matrix produces a much more efficient algorithm for solving the linear system $C_n x = b$, where $x, b \in \mathbb{R}^n$.

This is the bulk of this program and that is implementing and using the Discrete Fourier Transform (DFT) and the Fast Fourier Transform (FFT). The DFT is a beautiful piece of math, but in practice it is the FFT that makes a lot of things possible in engineering and computing as the FFT can be much fast than the DFT, hence the name. So in this program we will implement both the DFT and the FFT along with the inverses. We mainly will use them for the 2D-DFT for which we will be compressing images and seeing how they compare with the original, depending how much compression the user chooses to do. Then

lastly we will be undoing the blurring of an image and that is going to be done through the use of solving linear systems with circulant matrices.

# 3 Description of the Mathematics

## 3.1 Eigen-Decomposition of Upshift Matrix

First off we are going to show that the eigen-decomposition of the upshift matrix is $Z_n = F_n^H \Lambda_n F_n$. Now we know that $F_n^{-1} = F_n^H$. So we can use this to get

$$\begin{aligned} Z_n &= F_n^H \Lambda_n F_n \\ F_n Z_n F_n^H &= \Lambda_n \end{aligned}$$

Thus we have

$$\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \mu & \mu^2 & \cdots & \mu^{n-1} \\ \vdots & \mu^2 & \mu^4 & \cdots & \mu^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \mu^{n-1} & \mu^{2(n-1)} & \cdots & \mu^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & \vdots & \cdots & 0 & 1 \\ 1 & 0 & \cdots & \cdots & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ \vdots & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix}$$

Now $Z_n$ is just a permutation matrix and so we know that $F_n Z_n F_n^H$ turns to

$$\begin{pmatrix} 1 & 1 & 1 & \cdots & \cdots & 1 \\ \mu^{n-1} & 1 & \mu & \mu^2 & \cdots & \mu^{n-2} \\ \mu^{2(n-1)} & 1 & \mu^2 & \mu^4 & \cdots & \mu^{2(n-2)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \mu^{(n-1)(n-1)} & 1 & \mu^{n-1} & \mu^{2(n-1)} & \cdots & \mu^{(n-2)(n-1)} \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ \vdots & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix}$$

Then we can see that for the $(i-1)^{th}$ row or $F_n Z_n$ we have $\begin{pmatrix} \mu^{i(n-1)} & \mu^0 & \mu^i & \mu^{2i} \cdots \mu^{(n-2)i} \end{pmatrix}$ and for the $(i-1)^{th}$ column of $F_n^H$ we have $\begin{pmatrix} \omega^0 & \omega^i & \omega^{2i} & \omega^{3i} \cdots \omega^{(n-1)i} \end{pmatrix}^T$. Now remember that $\mu = e^{-i\theta}$ and $\omega = e^{i\theta}$ and so $\mu^{-1} = \omega$. Thus we have $\mu^{i(n-1)} = \omega^{i(1-n)} = \omega^i$. So we can see that each diagonal will be $\omega^i + \omega^i + \cdots \omega^i = n\omega^i$. Thus the diagonals of the lambda matrix will be just $\omega^i$ because $F_n$ and $F_n^H$ bring in the scaling of $1/\sqrt{n}$.

Next we can can see that for off diagonal entries of the product we have

$$\begin{aligned}\omega^{j} + \omega^{1k-0j} + \omega^{2k-1j} + \cdots + \omega^{(n-1)k-(n-2)j} \\ &= \omega^{k}\omega^{-1(k-j)} + \omega^{k}\omega^{0(k-j)} + \omega^{k}\omega^{1(k-j)} + \cdots + \omega^{k}\omega^{(n-2)(k-j)} \\ &= \omega^{k}\big(\omega^{(n-1)(k-j)} + \omega^{0(k-j)} + \omega^{1(k-j)} + \cdots + \omega^{(n-2)(k-j)}\big) \\ &= \omega^{k}\big((\omega^{(k-j)})^{n-1} + (\omega^{(k-j)})^{0} + (\omega^{(k-j)})^{1} + \cdots + (\omega^{(k-j)})^{n-2}\big) = 0\end{aligned}$$

So it indeed is an eigenvalue decomposition and we have

$$\Lambda_n = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & \omega & 0 & \cdots & \vdots \\ \vdots & 0 & \omega^2 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \cdots & 0 & \omega^n \end{pmatrix}$$

## 3.2 Eigen-Decomposition of Circulant Matrix

To show that the eigen-decomposition of a circulant Matrix is $C_n = F_n^H \Gamma_n F_n$ we will first need to show that $C_n$ is a a subspace of the vector space created by $nxn$ matrices. To do that let us first find a basis that spans the subspace. Now a circulant matrix is a matrix with all of its diagonals the same element. Thus each row is just a right shift of the row above it or each column is just an bottom shift of the column to its left.

Let us suppose the set of matrices

$$\left\{ \begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \cdots & 0 & 1 & 0 \\ 0 & \cdots & \cdots & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & \vdots & \cdots & 0 & 1 \\ 1 & 0 & \cdots & \cdots & 0 \end{pmatrix}, \cdots, \begin{pmatrix} 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & \cdots & \vdots \\ \vdots & \ddots & \ddots & 0 & 0 \\ 0 & \cdots & \cdots & 1 & 0 \end{pmatrix} \right\}$$

We can see that this is the set permutation matrices which when applied to a matrix on the left or right gives a shift of the row or columns respectively. Notice that each matrix is just the upshift circulant matrix raised to some degree. Thus we could right the set as

$\left\{(Z_n^0, Z_n^1, Z_n^2, \cdots, Z_n^{n-1})\right\}$. This is a basis for the space of circulant matrices. We can see that each element of the set represents one of the $n$ parameters in a circulant matrix. As circulant matrices only have $n$ parameters and each row is just a right shift of the previous row before it. Thus if you want a certain $n$ by $n$ circulant matrix with the first row being $\begin{pmatrix} a_0 & a_1 & a_2 & \cdots & a_{n-1} \end{pmatrix}$, then we can express it as

$$C_n = \sum_{i=0}^{n-1} a_i Z_n^i$$

The set is also linearly independent because it each element of the set represents $n$ specific spots in the an $n$ by $n$ matrix that the others do not. Thus the only way to get the zero vector is if all $a_i$ were identically 0. If we had $aA+bB$ where $A$ and $B$ are circulant matrices we can see that

$$\begin{aligned} aA + bB &= a\left(\sum_{i=0}^{n-1} a_i Z_n^i\right) + b\left(\sum_{i=0}^{n-1} b_i Z_n^i\right) \\ &= \left(\sum_{i=0}^{n-1} a a_i Z_n^i\right) + \left(\sum_{i=0}^{n-1} b b_i Z_n^i\right) \\ &= \sum_{i=0}^{n-1} (a a_i + b b_i) Z_n^i = \sum_{i=0}^{n-1} c_i Z_n^i. \end{aligned}$$

Thus the subspace is closed under vector addition and scalar multiplication. Thus we know that the space spanned by all circulant matrices is a subspace of all $n$ by $n$ matrices and further more that a basis for that subspace is $\left\{(Z_n^0, Z_n^1, Z_n^2, \cdots, Z_n^{n-1})\right\}$.

Recall that we want to find $\Gamma_n$ in $C_n = F_n^H \Gamma_n F_n$ where $\Gamma_n$ is a diagonal matrix corresponding to all the eigenvalues of the circulant matrix $C_n$. Now just up above we found that $C_n$

can be rewritten as a linear combination of upshift matrices. Thus we can see that

$$\begin{aligned}
C_n = \sum_{i=0}^{n-1} a_i Z_n^i &= F_n^H \Gamma_n F_n \\
F_n \left( \sum_{i=0}^{n-1} a_i Z_n^i \right) F_n^H &= \Gamma_n \\
F_n \left( \sum_{i=0}^{n-1} a_i (F_n^H \Lambda_n F_n)^i \right) F_n^H &= \Gamma_n \\
F_n \left( \sum_{i=0}^{n-1} a_i (F_n^H \Lambda_n F_n)(F_n^H \Lambda_n F_n) \cdots (F_n^H \Lambda_n F_n) \right) F_n^H &= \Gamma_n \\
F_n \left( \sum_{i=0}^{n-1} a_i F_n^H \Lambda_n^i F_n \right) F_n^H &= \Gamma_n \\
\sum_{i=0}^{n-1} a_i F_n F_n^H \Lambda_n^i F_n F_n^H &= \Gamma_n \\
\sum_{i=0}^{n-1} a_i \Lambda_n^i &= \Gamma_n
\end{aligned}$$

Thus we have an explicit formula for the eigenvalues of any circulant matrix $C_n$

$$\Gamma_n = \begin{pmatrix}
\sum_{i=0}^{n-1} a_i & 0 & \cdots & \cdots & 0 \\
0 & \sum_{i=0}^{n-1} a_i \omega^i & 0 & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & \vdots \\
\vdots & \cdots & 0 & \sum_{i=0}^{n-1} a_i \omega^{(n-2)i} & 0 \\
0 & \cdots & \cdots & 0 & \sum_{i=0}^{n-1} a_i \omega^{(n-1)i}
\end{pmatrix}$$

This result is wonderful. It allows for easy computation of the eigenvalues and intern we can easily see if a circulant matrix is singular or not as we can just search all the eigenvalues to see if their are any zeros. What this also does for us is makes it much simpler to solve the linear system $C_n x = b$. We can simply insert the eigen-decomposition in for the $C_n$.

Thus we will have

$$\begin{aligned} C_n x &= b \\ F_n^H \Gamma_n F_n x &= b \\ \Gamma_n F_n x &= F_n b \\ F_n x &= \Gamma_n^{-1} F_n b \\ x &= F_n^H \Gamma_n^{-1} F_n b \end{aligned}$$

The inverse of the $\Gamma_n$ is just the reciprocal of each of the diagonals which very easy to compute.

# 4 Description of the Algorithms and Implementation

## 4.1 Discrete Fourier Transforms

The implementations for the algorithms were not extremely difficult in this program. For this program we wanted to implement the Discrete Fourier Transform (DFT), the Inverse Discrete Fourier Transform (IDFT), the Fast Fourier Transform (FFT), and the Inverse Fast Fourier Transform (IFFT). The DFT is just a matrix-vector product and thus we can easily just implement a code that does an inefficient $O(n^2)$ to transform it to the frequency domain. We can exploit the structure of the DFT though and use Horner's Rule. We can see that because the DFT matrix is just

$$\begin{pmatrix} \mu_n^0 & \mu_n^0 & \mu_n^0 & \cdots & \mu_n^0 & \mu_n^0 \\ \mu_n^0 & \mu_n^1 & \mu_n^2 & \mu_n^3 & \cdots & \mu_n^k \\ \vdots & \mu_n^2 & \mu_n^4 & \mu_n^6 & \cdots & \mu_n^{2k} \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ \mu_n^0 & \mu_n^k & \mu_n^{2k} & \cdots & \mu_n^{(k-1)(k-1)} & \mu_n^{(k-1)k} \\ \mu_n^0 & \mu_n^k & \mu_n^{2k} & \cdots & \mu_n^{(k-1)k} & \mu_n^{(k*k)} \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ \vdots \\ f_{k-1} \\ f_k \end{pmatrix}$$

where $\mu_n = e^{-2\pi i/2}$. Thus each row times column is the same as a polynomial raised with the input $\mu_n^k$ with coefficients of $f$. It does become more efficient and faster than without using Horner's Rule, but the FFT is still much faster for higher dimensions as we will see

in the results section. Now the IDFT is almost identical to the DFT, but instead of $\mu_n$ we have $\omega_n = e^{2\pi i/2}$.

The storage for both of these algorithms are only $O(n+1)$ which is for a temporary vector that we will put the transformed $f$s in and a variable for $\mu$ or $\omega$. Now a regular matrix vector product is quadratic complexity, but with Horner's Rule we can make it in quicker quadratic time, but is still not the FFT.

## 4.2 Fast Fourier Transform

The implementations of the FFT is used through recursion. We can see the code itself because it is recursive is very short. The way the Fast Fourier Transform works is that

**Algorithm 1** Fast Fourier Transform

```
if n=2 then
    discreteFourierTransform(n, f);
else
    precision complex feven[n/2];
    precision complex fodd[n/2];
    sortingAlgorithm(n, f, feven, fodd);
    fastFourierTransform(n/2, feven);
    fastFourierTransform(n/2, fodd);
    for int k = 0; k < n; k++ do
        f[k] = (feven[k mod (n/2)] + exp((-2 * π * k/n) * i)*fodd[k mod (n/2)])/√2
    end for
end if
```

we keep factoring a large dense matrix into many large sparse matrices from which we then perform the the DFT on small matrices. The the way we can do this in code is with recursion. We keep sorting the $f$s into even/odds and then sending the even $f$s and odd $f$s through the FFT function again. Every time we go into the FFT we are halving the size of the $f$ vector. We eventually get down to the base case of $n = 2$ and that is we send it to the DFT. Then we combine the $f$ evens and $f$ odds as the factorization sees them and go back up the tree it creates. The IFFT is like the IDFT almost identical to its sister the FFT. The differences are again changing to a positive exponent for the exponential

function and calling the IFFT and IDFT in the function.

The storage and complexity we acquire through all this recursion is worth it as well. We lose a the nice linear storage for $O(n \log_2(n))$ storage, but we speed up substantially in time complexity. We can see in the algorithm that we store $O(n)$ in each recursion. But we have $\log_2(n)$ levels. Now the time complexity comes from the fact that when we solve huge matrix vector products with only 2 non-zero elements in each row of the matrix and the fact that these matrices are highly structured with those 2 non-zero elements we an do it in linear time. Just like storage though we will have $\log_2(n)$ levels of which we will need to do this. Thus the time complexity will decrease from the order of $O(n^2)$ to $O(n * \log_2(n)$ which is amazing to do especially only at the cost of storage increasing a small amount.

## 4.3 Circulant Matrix Linear System Solver

**Algorithm 2** Circulant Matrix System Solver (Cx = b)

```
fastFourierTransform(n, b)
for int k = 0; k < n; k++ do
    precision complex gamma = 0;
    precision complex muk = exp((−2 ∗ π ∗ k/n) ∗ i);
    for int j = n − 2; j ≥ n; j−− do
        gamma = gamma*muk + c[j];
    end forb[k] = b[k]/gamma;
end for
inverseFastFourierTransform(n, b)
```

With the math we did in the Description of Math section and with our Fourier Transform functions we can solve systems with circulant matrices quickly now. As we saw $Cx = b$ can be transformed into $x = F_n^H * \Gamma_n^{-1} * F_n * b = c$. We can see that solving for the eigenvalues can take some time as each is essentially a polynomial, luckily though we can apply Horner's Rule which brings the time it takes to calculate them down to order $O(n^2)$ as each is linear. But everything else is linear or $O(n \log_2(n))$. The algorithm works like this. First take the FFT of $b$, then find the diagonal of $\Gamma_n$ and divide it by the transformed $b$ then lastly use the IFFT on that and done. The storage complexity is constant for this as in the function not including what the FFT and IFFT do in their respective functions.

## 4.4 2-Dimensional Discrete Fourier Transform and Compression

**Algorithm 3** 2D Discrete Fourier Transform ($A' = FAF^T$)

```
precision complex w[n];
for int k = 0; k < n; k++ do
    zeroArray(w);
    w[j] = A[j][k] for j ∈ [0, n − 1];
    fastFourierTransform(w);
    A[j][k] = w[j] for j ∈ [0, n − 1];
end for
for int k = 0; k < n; k++ do
    fastFourierTransform(A[k]);
end for
```

The 2D DFT is for transforming matrices to the frequency domain and is nice for things like compression purposes. From the notes in class we can see that the 2D fourier transform looks like $FAF^T$. Now from this we can see that in reality we are actually first just transforming first the columns of $A$ then after that we are transforming the rows of $A'$. That is what the code does we do need a temporary array for the columns of $A$ as we cannot insert just a column into the FFT. So we transform that temporary array then replace that column of $A$ with the transformed one. Then after all of $A$s columns have been transformed we then transform the rows of $A'$ or the new A. This loop is shorter though as we can insert each row of $A'$ straight into the FFT.

So we can see that for complexity we have on the order of $O(2n^2 \log_2(n))$. That is because we have to do the FFT $2n$ times throughout the function. Storage in the function only uses linear log time $O(n \log_2(n))$ which also comes from the FFT. We also have a compression algorithm that is used once we have transformed $A$ using the 2D-DFT. The compression algorithm goes through and checks the norm of each entry of the transformed $A$ and if it is below a certain tolerance chosen by the user turns that entry to zero.

## 4.5 Focusing Blurred Images

This next section is the algorithm that solves $M = C^T * I * C$ where $C$ is a circulant matrix, $M$ is a blurred image, and $I$ is the original image which is what we have to find.

Solving this is a clever application of our Circulant Matrix System Solver (CMSS), but also very similar to our 2D-DFT algorithm. First we let the matrix $L = I * C$ and then solve $M = C^T * L$ using the CMSS. Then after solve $L = I * C$.

To do this though we will need to solve each column of $I$ separately like we did for the 2D-DFT. First though to do this we will need the parameters of $C^T$. So we initialize an array and fill it with the parameters of $C^T$ and then use a for loop to solve for $L$. After we want to solve $L = I * C$. Now we will need to do this by rows of $I$, so $L_k = I_k C$. Now notice that this is also the same as solving for $C^T * I_k^T = L_k^T$ with CMSS. Again we will need to know the the parameters of $C^T$ and then we can solve for $I$ row by row.

The time complexity of this algorithm is $n$ times the complexity of the our CMSS as we have it in a for loop. Thus the time complexity is on the order of $O(n^3)$ because CMSS is of $O(n^2)$. The storage complexity is better than that at $O(n \log_2(n))$ which comes from using the CMSS. In the function itself only a linear array is initialized.

# 5 Description of the Experimental Design and Results

First let us check our DFT with a simple 4 length vector of $(i, 1+2i, 2+3i, 3+4i)^T$. Doing it out by hand we get that the DFT should be $(3+5i, -2, -1-i, -2i)^T$. The code does get that result and then the IDFT does return it to its original. Then I also checked it with a few examples that I found online. The DFT and IDFT were very simple algorithms and I am from here on going to assume they are correct.

## 5.1 Fast Fourier Transform vs the Discrete Fourier Transform

Next is the Fast Fourier Transform. First off we should check to see if it is in agreement with our DFT and IDFT. Thus let us use an length 8 vector where each element is its row

+(row+1) $* i$. Thus we will have

$$\begin{pmatrix} 0+1*i \\ 1+2*i \\ 2+3*i \\ 3+4*i \\ 4+5*i \\ 5+6*i \\ 6+7*i \\ 7+8*i \end{pmatrix} \begin{pmatrix} 9.9+12.7*i \\ -4.83+2*i \\ -2.83-1.65*10^{-15}*i \\ -2+-0.828*i \\ -1.41+-1.41*i \\ -0.828+-2*i \\ 4.04*10^{-15}-2.83*i \\ 2+-4.83*i \end{pmatrix} \begin{pmatrix} 9.9+12.7*i \\ -4.83+2*i \\ -2.83-1.1*10^{-15}*i \\ -2+-0.828*i \\ -1.41+-1.41*i \\ -0.828+-2*i \\ 6.28*10^{-16}-2.83*i \\ 2+-4.83*i \end{pmatrix} \begin{pmatrix} 1.26*10^{-15}+1*i \\ 1+2*i \\ 2+3*i \\ 3+4*i \\ 4+5*i \\ 5+6*i \\ 6+7*i \\ 7+8*i \end{pmatrix}$$

Table 1: *Left to right the original data vector, the DFT, the FFT, and then the IFFT*

We can see that the only visible difference between the DFT and the FFT is floating point arithmetic error in the terms close to zero. Thus we can see that the FFT works well and the IFFT gets the FFT back to the original data.

Now that we see the two algorithms get the same result lets take a look at how fast each gets the result. Using the same method to populate the vector to be transformed below are two tables that show how many ticks it took for the algorithm to execute. For each data point $t$, corresponding to a $2^t$ dimensional vector, I ran the calculation 3 times and then took the average of those three amount of ticks each time.

| t | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| DFT | 33, 50, 63 | 68, 42, 67 | 69, 67, 73 | 89, 93, 84 | 136, 136, 137 | 340, 268, 293 |
| FFT | 3, 5, 7 | 15, 8, 16 | 36, 29, 44 | 89, 79, 85 | 175, 174, 202 | 399, 343, 390 |

| t | 7 | 8 | 9 | 10 |
|---|---|---|---|---|
| DFT | 594, 893, 893 | 3306, 1691, 2939 | 9253, 7898, 7280 | 22514, 20331, 19725 |
| FFT | 5979, 829, 868 | 1795, 902, 856 | 2878, 1496, 1425 | 2516, 2580, 2702 |

Table 2: *This shows the 3 ticks for each* $2^t$

| t | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DFT | 49 | 59 | 70 | 89 | 136 | 300 | 793 | 2645 | 8144 | 20857 | 62051 | 231638 |
| FFT | 5 | 13 | 36 | 84 | 184 | 377 | 765 | 1184 | 1933 | 2589 | 5605 | 11320 |
| Ratio | 9.8 | 4.54 | 1.94 | 1.06 | 0.74 | 0.8 | 1.04 | 2.23 | 4.21 | 8.06 | 11.1 | 20.5 |

Table 3: *This shows the average of the 3 ticks for $2^t$ and the ratio of the two methods*

We can see that the FFT is faster at the lowest dimensions that it can complete. But from $t = 4$ to $t = 7$ the DFT seems to be a bit faster. Which if we could graph a continuous function of these those two points would be near $2^t$ and $t$ would intersect as $\frac{n^2}{n\log_2(n)} = \frac{n}{\log_2(n)} = \frac{1}{2}2^t$. But as we can see past $t = 7$ the FFT starts being much faster than the DFT.

## 5.2 2D-Discrete Fourier Transform

First off on the webpage "http://fourier.eng.hmc.edu/e161/lectures/fourier/node12.html" their is a nice example to check my code with. You can see the original data and the result below. I also tested my 2D-IDFT on the transformed data as well and it did return it back to the original. With only differences of less than $10^{-14}$.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 70 | 80 | 90 | 0 | 0 | 0 |
| 0 | 0 | 90 | 100 | 110 | 0 | 0 | 0 |
| 0 | 0 | 110 | 120 | 130 | 0 | 0 | 0 |
| 0 | 0 | 130 | 140 | 150 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 4: *This is the original Data*

| 165 | -98.89 | 10 | -21.11 | 55 | -21.11 | 10 | -98.89 |
|---|---|---|---|---|---|---|---|
| -63.11 | -11.34 | 27.68 | 13.23 | -21.04 | 1.59 | -32.68 | 85.66 |
| 15 | -1.516e-15 | -5 | -2.929 | 5 | -8.882e-16 | 5 | -17.07 |
| -41.89 | 16.77 | 2.678 | 6.339 | -13.96 | 4.341 | -7.678 | 33.41 |
| 15 | -8.536 | 2.143e-15 | -1.464 | 5 | -1.464 | -1.531e-15 | -8.536 |
| -41.89 | 33.41 | -7.678 | 4.341 | -13.96 | 6.339 | 2.678 | 16.77 |
| 15 | -17.07 | 5 | -6.727e-16 | 5 | -2.929 | -5 | -9.465e-15 |
| -63.11 | 85.66 | -32.68 | 1.59 | -21.04 | 13.23 | 27.68 | -11.34 |

Table 5: *This is the real part of the 2D-DFT*

| 0 | -88.89 | 55 | 11.11 | -6.736e-15 | -11.11 | -55 | 88.89 |
|---|---|---|---|---|---|---|---|
| -90.53 | 89.19 | -27.07 | 6.945 | -30.18 | 16.77 | 15 | 19.87 |
| 15 | -17.07 | 5 | -1.148e-15 | 5 | -2.929 | -5 | -3.66e-15 |
| -15.53 | 31.95 | -15 | -0.8058 | -5.178 | 4.874 | 12.93 | -13.23 |
| -9.185e-15 | -8.536 | 5 | 1.464 | -3.674e-15 | -1.464 | -5 | 8.536 |
| 15.53 | 13.23 | -12.93 | -4.874 | 5.178 | 0.8058 | 15 | -31.95 |
| -15 | -4.289e-15 | 5 | 2.929 | -5 | 1.364e-15 | -5 | 17.07 |
| 90.53 | -19.87 | -15 | -16.77 | 30.18 | -6.945 | 27.07 | -89.19 |

Table 6: *This is the imaginary part of the 2D-DFT*

I also compared some others with other students and got the same transformations as they did. Thus I think it is safe to say that the my algorithms work correctly. Now part of the other task with the 2D-DFT implementation was to compress the transformed data and then to see how the compressed data fared when put back into the 2D-IDFT. to see if it made huge errors or not.

I checked the error after using compression in a variety of ways. The compression itself was using the modulus of the complex data and if it was below a certain tolerance than we set it to zero. The first way I checked the error once we return using the IDFT was using the modulus on the difference of the original entry and the compressed entry. I realized soon after though that this might not be the best way to determine if the data after being compressed is still legible as the original. Thus my three other ways were looking at the real and imaginary parts separately and also together. So for the reals I would check all the real parts of each entry and see if the relative error was more than 5% then I would count how many entries in the matrix that surpassed that thresh-hold. I also I had a similar

count for the imaginary part of each entry and then if either of the imaginary part or real part are about the thresh-hold of error.

The way I fill in my so called image matrices is by randomly inserting integers from 1 to 10000 for both the real and the imaginary parts of each entry. The table below shows differences in compression levels for the different error techniques discussed above. These will be for 32 size matrices and we can infere a few things from the table below

| Tolerance | 500 | 750 | 1000 | 1500 | 2000 |
|---|---|---|---|---|---|
| Error Max | 120 | 257 | 473 | 975 | 1699 |
| Real Percent | 0.00195 | 0.00195 | 0.00684 | 0.00585 | 0.01367 |
| Imag Percent | 0.00293 | 0.00391 | 0.00293 | 0.00683 | 0.00977 |
| Both Percent | 0.00488 | 0.00586 | 0.00977 | 0.01269 | 0.02344 |
| Compressed | 18 | 34 | 62 | 116 | 220 |

Table 7: *This has the largest error in compressed matrix with respect to the modulus. The percentage of Real parts of the entries that we outside the error tolerance, the same for Imag, if one of the parts is out of the error tolerance, and lastly how many entries were compressed*

So we can see as you start putting more pressure on the compression the amount of entries that stray from the original get higher. That is to be expected but even with 220 out of the $32^2$ entries being compressed to 0 in the frequency domain I would have expected more entries in the time domain to stray and be more than 5% from their original numbers. This just shows that compression can save alot of space because you do not need to store 0 entries and even with them you get a relatively close matrix compared to the original.

## 5.3 Circulant Matrix Solver

This is a short section on of the implementations that we made which solves the linear system $C_n x = b$ where $C_n$ is a circulant matrix. This section is mainly a prelude to what is coming up in the next section as we needed this to be able to achieve the next step of unblurring an image.

The implementation is correct as I did an example with $b = (-3, 0, -1, 4)^T$ and the parameters of $C_n = (1, 2, 3, 4)$. By hand I found that the linear system solves to be

$x = (0.75, -0.25, 1.25, -1.75)^T$. The code does solve this and because we use the code in the focusing an image function if that function does undo the blurring of an image correctly than we know that this system solver must be correct.

## 5.4 Unblurring an Image

Image processing is a wonderful and very cool field. What we are doing today is showing how using circulant matrices we can blur an image and then by the algorithm we developed if we know the circulant matrix and have the matrix of the blurred image we can get the original back in perfect original form. The blurry image we were given is a 512 by 512 matrix and that is quite a few pixels and it is amazing that the code does bring convert it to its original farily quickly for a quite large matrix.

The first image is the original. Then after is

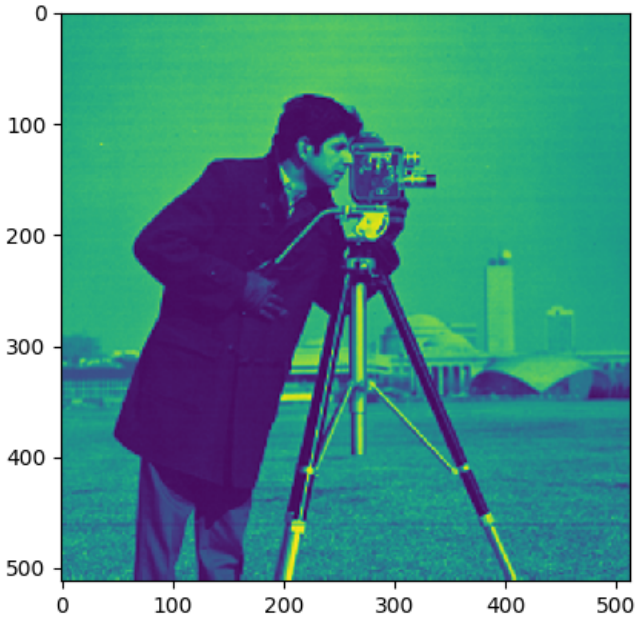


Image 1: *The original image file*

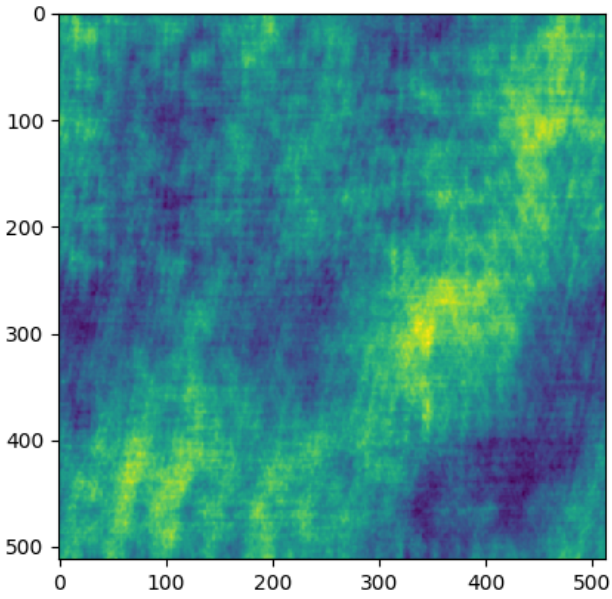


Image 2: *The blurred image file*

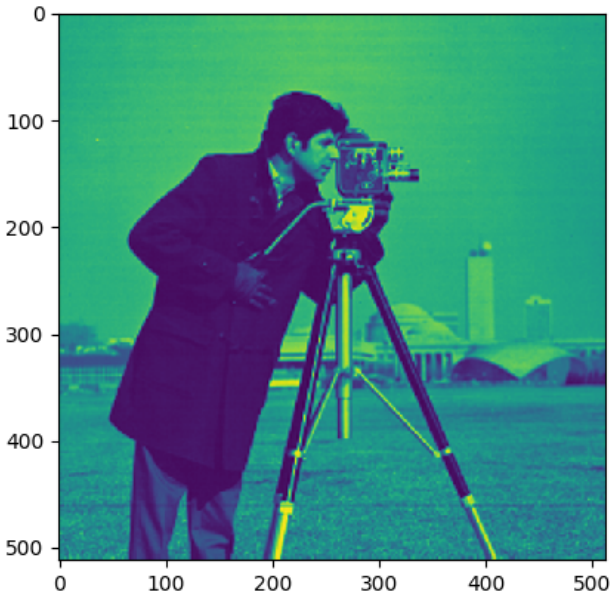


Image 3: *The focused image file*

We can see that the focused and original image look identical and that is because they nearly are. Running an error checking program and I found out that the biggest difference between the focused and the original data is on the order of $10^{-13}$. Thus to the eye they are the same image.

# 6 Conclusions

We did a lot in this program and learned a lot about signal processing and how important the DFT can be in today's world. First we cleverly solved certain eigen-value decomposition's and found the the DFT was part of those decomposition. Next we found clever more efficient ways to solve linear systems which have circulant structure.

Then we started implementing the DFT and FFT. These implementations were small, but oh so powerful in the grand scheme of computers. Most people today enjoy taking photos and especially with cameras on phones being better than ever, everyone can take pictures so easily and really nice ones as well. Thus image processing has become a very important part of society today. Now storage is a luxury we do have, but with photos getting more dense and saturated with pixels we will need to make compression algorithms to store photos efficiently without losing to much data from the photo. That is one of the things we discussed in this report the basics of how image compression works and how far can you compress without to much loss in data.

We also talked about image processing and signal processing in general a bit as well. After all the DFT and FFT are used to transform a digital signal from one domain to a frequency domain of some sort. With the circulant solvers we talked about in the beginning we used those more to learn how to undo blurring in an image. It came out beautiful and can really excite someone when they see math actually being used in everyday life in application such as snapchat filters and others. This discussion was more of an intro to a large topic in digital signal processing and was fun.