FCM 2 HW 2

Ryan Bausback

February 2021

1 Executive Summary

The Discrete Fourier Transform (DFT) and the Fast Fourier Transform (FFT) were implemented in C++, as well as their inverses, in order to empirically validate their properties and prove that the FFT is actually faster. Several circulant matrices were also analyzed and systems involving them were solved using the DFT and FFT. We found that all properties held as expected and that the FFT performed the Fourier Transform quicker than the regular DFT.

2 Statement of the Problem

The Fourier Transform is a fundamental tool in quantum mechanics and signal processing, meaning it is therefore important to have a way implement it on a computer. This is were the Discrete Fourier Transform (DFT) comes in. However, because the DFT requires $O(n^2)$ computations to compute, when the size of the input data or signal (n) is very large, this can be very costly, leading to creation of the Fast Fourier Transform with only $O(n \log n)$ (which assumes $n = 2^k$).

Our goal is to validate the properties of the DFT / FFT and their inverses (IDFT and IFFT respectively). These properties include: Isometry (the preservations of norms or $||F_n x||_2 = ||x||_2$), Unitary (the fact that $F_n^H F_n = I$), and Reconstruction (applying the DFT and then IDFT or vice versa to a vector x will return the original input). We also would like to verify the fact that the FFT is faster, as its name would imply. This will be done by timing how long each function of either the DFT or FFT takes to evaluate the same input vector of a range of sizes.

Additionally, we will also use the DFT/IDFT and FFT/IFFT pairs in order to solve the system $C_n x = b$, where C_n is a circulant matrix, as well as to determine its eigenvalues and matrix 2-norm.

3 Description of Mathematics

The foundation of the Discrete Fourier Transform and its inverse (DFT/IDFT) are the parameters that define the matrices F_n and F_n^H that are applied to the input vector, x. For the DFT, these parameters are defined as $\mu = e^{-i\theta}$ and $\omega = e^{i\theta}$ for the IDFT. Here, θ is defined as $2\pi/n$, where n is the number of elements of the input vector x. These parameters are then used to define the matrices $F_n = \frac{1}{\sqrt{n}}\Phi$ and $F_n^H = \frac{1}{\sqrt{n}}\overline{\Phi}$ where:

$$\Phi = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \mu & \mu^2 & \dots & \mu^{n-1} \\ \vdots & & & \ddots & \vdots \\ 1 & \mu^{n-1} & \mu^{2(n-1)} & \dots & \mu^{(n-1)(n-1)} \end{bmatrix}$$

and $\overline{\Phi}$ is the element-wise complex conjugate of Φ with all of the μ's replaced by ω's. [1]

This definition of the DFT/IDFT directly leads us to the properties that we want to empirically show, the first being that the DFT is a unitary matrix. We can see this fact by recognizing that $\frac{1}{n}\Phi^H\Phi = I$ where $\Phi^H = \overline{\Phi}$ since Φ is symmetric about the main diagonal, giving us $F_nF_n^H = I$. Based on this observation, it then becomes obvious that if we apply $F_nF_n^Hd$, would should get the original input vector x as the output since $F_nF_n^Hd = Id = d$. This is the property of reconstruction. The final property that we want to show is the property of norm preservation or isometry. This comes from the fact that since we recognized F_n as a unitary matrix, by the definition of a unitary matrix, if we apply $||F_nd||_2$ for example, we should get an identical result to doing $||d||_2$. [2]

The Fast Fourier Transform (FFT) exploits the structure of the DFT in order to produce a computationally more efficient but identical result. However, in order to do this, we need the critical assumption that the number of elements in the input vector is of the form $n = 2^k$. This gives us that $\mu_{2k}^2 = \mu_k$, since μ is a root of unity and is rotating clockwise around the unit circle (the same is true for ω but rotating counter-clockwise). Since, $n = 2^k$, that means $n = 2k$ also, so we can decompose the row equations defining the matrix-vector product $F_nd = s$ into their even and odd parts using the formula $p(x) = p_{even}(x^2) + xp_{odd}(x^2)$.Here, $x = \mu_n^r$ where r corresponds to the row of the equation. Since $(\mu_n^r)^2$ in the formula, we can apply the first fact, giving us a sparse matrix that can be written in the form as:

$$\bar{\Phi}_n f^{(n)} = \begin{pmatrix} \hat{f}_{top}^{(n)} \\ \hat{f}_{bot}^{(n)} \end{pmatrix} = \begin{pmatrix} I_k & \Omega_k \\ I_k & -\Omega_k \end{pmatrix} \begin{pmatrix} \bar{\Phi}_k & 0 \\ 0 & \bar{\Phi}_k \end{pmatrix} \begin{pmatrix} f_{even}^{(n)} \\ f_{odd}^{(n)} \end{pmatrix}$$

$$\Omega_k = \begin{pmatrix} 1 & & & & \\ & \mu_n & & & \\ & & \mu_n^2 & & \\ & & & \ddots & \\ & & & & \mu_n^{k-1} \end{pmatrix}$$

This idea can then be applied recursively to the current even and odd parts by further breaking those into their even and odd parts respectively, until one is left with only a single element for each of the even and odd parts of d. At this point, computation of the matrix-vector products of the small vectors with the sparse matrices can begin. [3]

An additional area that will be explored is solving $C_n x = b$, given a complex vector b and a complex circulant matrix C_n, using the FFT/IFFT pair. A circulant matrix has the structure (for example with n=3):

$$\begin{pmatrix} a_0 & a_1 & a_2 \\ a_2 & a_0 & a_1 \\ a_1 & a_2 & a_0 \end{pmatrix}$$

To use the FFT/IFFT, we first need to recognize that C_n can be written in the form $C_n = F_n \Gamma_n F_n^H$, where Γ_n is a diagonal matrix containing the eigenvalues of C_n. To get this fact, we can decompose the circulant matrix into matrices Z_n^i containing reorderings of the fundamental basis vectors e_i as their columns, such as $C_3 = a_0 Z_3^0 + a_1 Z_3^1 + a_2 Z_3^2$. These Z_n^i can be further decomposed into $Z_n^i = F_n^H \Lambda_n^i F_n$ and if we apply this to all of the Z_n^i in the C_n decomposition, we get $C_n = F_n^H (a_0 \Lambda_n^0 + ... + a_{n-1} \Lambda_n^{n-1}) F_n$. This interior portion $(a_0 \Lambda_n^0 + ... + a_{n-1} \Lambda_n^{n-1})$ is what we define as Γ_n. In practice, to actually compute these eigenvalues, since each diagonal element of Γ_n is $d_{kk} = a_0 + a_1 \omega^k + ... + a_{n-1}(\omega^k)^{n-1}$, we can just apply the distinct elements of the circulant matrix in a vector to the FFT according to $g_n = \sqrt{(n)} F_n^H a_n$ rather than doing a decomposition.[4]

To actually solve the equation, the decomposition of $C_n = F_n^H \Gamma_n F_n$ can now be applied to b, which, given the unitary property of the FFT described above, leads to $x = F_n(\Gamma_n^{-1}(F_n^H b))$. Since Γ_n is diagonal, its inverse is found by taking $1/d_{ii}$. [4]

4 Algorithm and Implementation

For the DFT, a version of the simple accumulation algorithm in [1] was implemented. Since the matrix multiplication is really a polynomial evaluation with the ω's or μ's of increasing power as you move left to right in the rows, you can use Horner's rule to evaluate it in only $O(n)$ computations or a single for-loop. This is then done for each of the rows, resulting in a nested for-loop that gives $O(n^2)$ computations.

For the FFT, either the recursive form or the factored form could be chosen. In this case, the recursive algorithm was implemented that is similar to the one in Golub and VanLoan. [5] The algorithm uses a for loop to break the input vector into its even and odd components based on whether the index is even or odd, and stores them in corresponding even/odd subarrays of size $n/2$. These subarrays are then passed recursively into the FFT function until the input size is 1. At this point, the functions begin to work in reverse, by using a single for-loop to add the odd portion multiplied by the ω's (or μ's) to the even portion for the first $n/2$ indices, and vice-versa with subtraction for the last $n/2$ indices. Since this is done recursively, this happens $O(\log_2 n)$ times as $n = 2^k$, where each level for-loops takes a total of $O(n)$ if all added up, giving a total computational complexity of $O(n \log_2 n)$. Total storage during this process is $O(n)$ since each step in the recursion produces n additional storage and this happens overall less than n times.

The ω's or μ's are computed in the function itself. A boolean parameter dictates whether the ω's or the μ's are to be computed. A seperate function exists that can compute the ω's or μ's so that both could be applied to verify the unitary property of the FFT/DFT. It does not change the overall order of computations for the FFT or DFT to compute them within the routines, just an adjustment of the constant. It also does not change the order of storage since all values are applied directly after computation.

5 Experimental Design and Results

5.1 Validation of Properties

The first property that we would like to validate is the property of isometry. This means that if we apply the DFT/FFT to a data vector and then apply a norm, we should get the same result as applying the norm directly to the data vector (i.e. $||F_n x||_2 = ||x||_2$). As a deterministic, simple test, we will use the input vector $[8, 4, 8, 0]$ (n=4) which has a correct DFT output of $[10, -2i, 6, 2i]$. If we apply the L2 Norm to both, we get $\sqrt{|8|^2 + |4|^2 + |8|^2} = \sqrt{|10|^2 + |-2i|^2 + |6|^2 + |2i|^2} = 144$. This is the same result we got when from applying it in the program.

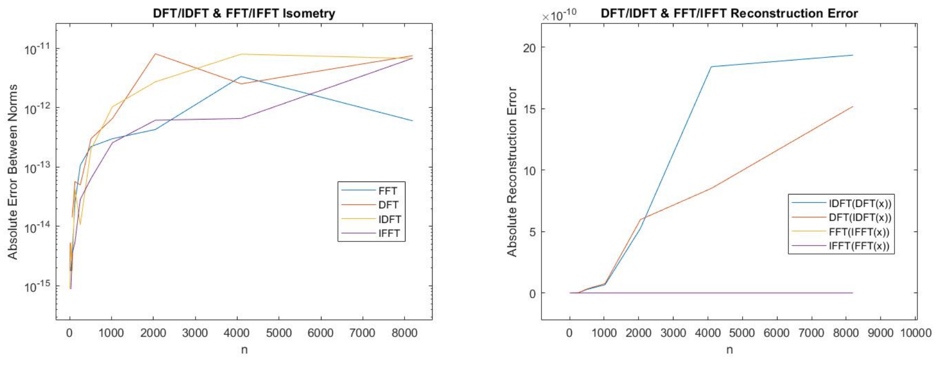


Figure 1: Isometry and Reconstruction for Random Input Vectors, size n

We also applied the DFT/IDFT and FFT/IFFT to a series of randomly generated complex vector of varying sizes n, and compared the difference between the L2 norm on the transform with the L2 norm on the input vector. As the left half of Figure 1 illustrates, even when the size of the random input vector was over 8000, the difference between the L2 norm on the original and the L2 norm on the DFT/FFT output was less than 10^{-11}, which is relatively close to machine epsilon for double. For most data vectors less than 50 elements, it was identically 0 (i.e. less than 10^{-16}). Therefore, the property of isometry is confirmed.

The next property to be validated is the property of reconstruction, or that applying the DFT to a vector x and then applying the IDFT to the output will result in the original x. When we applied IDFT(DFT()) to our test vector of $[8, 4, 8, 0]$, we got $[(8, 7.77156e^{-16})$, $(4, 2.44929e^{-16})$, $(8, -7.42829e^{-16})$, $(0, -1.46958e^{-15})]$ where the left number is the real part and the right number is the imaginary part. This is extraordinarily close to the original input.

To provide further evidence, the DFT and then IDFT and vice versa, as well as FFT(IFFT()) and IFFT(FFT()), were applied to randomly generated complex input vectors of increasing size n. The elements of the output of these processes were then compared with the original input elements in absolute value, and the maximum difference found. As the right half of Figure 1 shows, even for $n > 8000$, this maximum difference was less than 10^{-10}, sufficiently confirming the property of reconstruction.

The final property that we want to confirm is that F_n and F_n^H are unitary matrices, meaning that $F_n F_n^H = I$. This requires us to apply the columns of F_n to F_n^H and vice versa for different sizes of n. For a simple test, the vector of all ω's raised to increasing power based on index and divided elementwise by $\sqrt{n}$ was applied to the DFT with $n = 4$. This is the same as applying the 2nd column of the IDFT matrix to the DFT, which should give us e_1 or $[0, 1, 0, 0]$, since this is the second column of the identity. The output of the algorithm was $[(-5.55112e^{-17}, 2.77556e^{-17})$,$(1, 1.2326e^{-32})$,$(0, -3.06162e^{-17})$, $(0, -6.12323e^{-17})]$ which is within machine epsilon for double of the correct answer.

After confirming it in the n=4 case, all $n = 2^k < 1000$ were tested. The maximum difference for each element from its correct value (1 if on the diagonal, 0 otherwise) was determined for each vector in the IDFT or IFFT matrix when applied to the DFT or FFT matrix, and vice versa. Then the maximum of those maximums was found for each n. As Figure 2 shows, the output of $F_n F_n^H$ was within 10^{-13} or less of the identity for all elements of all transform matrices up to 512x512. There was no difference between applying the columns of the DFT/FFT to the IDFT/IFFT versus the other direction, as Figure 2 illustrates only two of the lines being visible due to overlap.

5.2 Computational Time and Complexity

It was also empirically determined whether the FFT is indeed faster than the DFT, as its name would suggest. However, one issue with accomplishing this goal is that execution time for a single iteration of either function for small values of n gives identical times of 0. Therefore, in order for any discernible difference to be detected, each function will be called 1,000 times for increasing

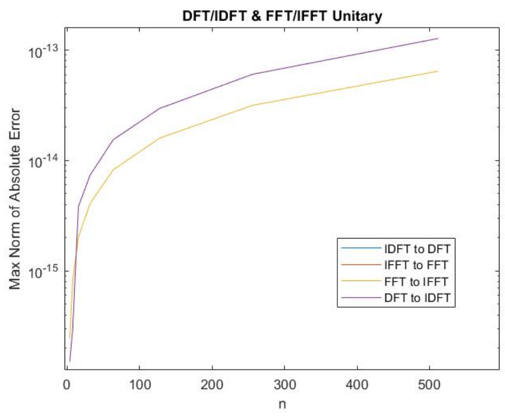
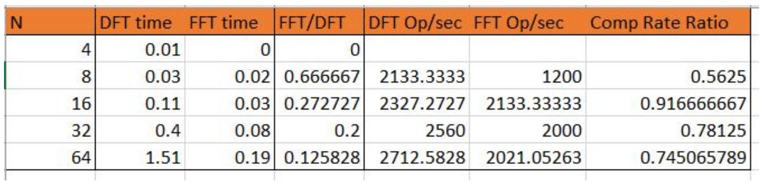


Figure 2: Max Difference of all elements of $F_n F_n^H$ from I for size n

values of n. As columns 2 and 3 of Figure 3 and Figure 4 illustrate, as the size of the input vector (n) increases, the FFT clearly takes less and less time to evaluate than the DFT. By doing an quadratic interpolation of the points for each, we can also see that the DFT is reflecting its $O(n^2)$ computations, while FFT is less than $O(n^2)$ as we expected since the coefficient of x^2 is of the order of 10^{-5}.



N	DFT time	FFT time	FFT/DFT	DFT Op/sec	FFT Op/sec	Comp Rate Ratio
4	0.01	0	0			
8	0.03	0.02	0.666667	2133.3333	1200	0.5625
16	0.11	0.03	0.272727	2327.2727	2133.33333	0.916666667
32	0.4	0.08	0.2	2560	2000	0.78125
64	1.51	0.19	0.125828	2712.5828	2021.05263	0.745065789

Figure 3: Statistics from Timing 1000 DFT and FFT Function Calls

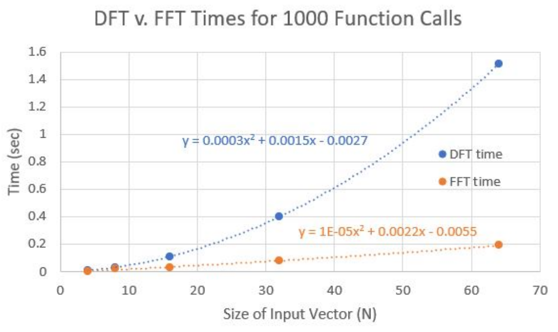


Figure 4: Times for 1000 DFT and FFT Function Calls

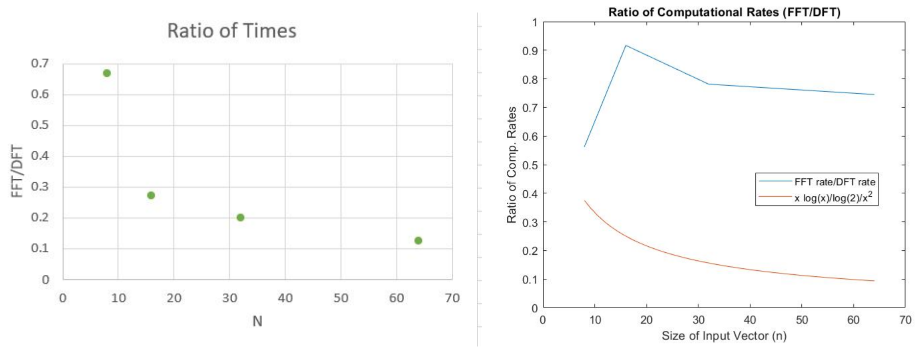


Figure 5

The computational advantage of FFT over DFT is also clearly shown in the ratio of the times (FFT/DFT) as n increases (Figure 5, (left)). The time it takes to evaluate the FFT gets smaller and smaller compared to the time it takes to evaluate the DFT. This makes sense as the DFT is $O(n^2)$ while the FFT is $O(n \log n)$, and n^2 and $n \log n$ begin to diverge as $n \to \infty$.

If we plot the ratio of the computational rates (number of computations/time to evaluate the function) as in Figure 5 (right), we can see that the shape past n=16 effectively mirrors the ratio of the orders of computation, further confirming the correctness of our implementation. This difference in the graphs is likely due to constants introduced by the seconds not fully cancelling out. Additionally, we can see that for n=8, the ratio of the computational rates is lower than predicted by the ratio of orders of computation. This might be attributed to the increased overhead associated with creating separate arrays in which to store the split halves of the input vector. It is more visible for the smaller values of n since the DFT does less computations overall for low n than for high n, meaning the denominator is smaller and less able to dilute its impact. This is also seen in columns 5 and 6 of Figure 3, where the FFT is able to due fewer computations per second than the DFT, since the FFT does additional assignment and declaration, while the DFT does computation almost exclusively.

However, it is obvious that the FFT is computational superior to the DFT, despite a slower computational rate, since it has to perform less computations overall to get an identical result.

5.3 Linear Systems with Circulant Matrices

The final area that will be investigated is the use of the DFT/FFT to solve linear systems involving circulant matrices by finding their eigenvalues, as described in Section 3. A simple, real test case of $a = (1, 2, 3, 4)$ was used where a is the first row of the circulant matrix defined by the values of a. This matrix therefore has the form:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 2 & 3 \\ 3 & 4 & 1 & 2 \\ 2 & 3 & 4 & 1 \end{pmatrix}$$

with the correct eigenvalues of $\gamma = (-2, 10, -2+2i, -2-2i)^T$. The inverse FFT was applied to a and $\sqrt{n}$ i.e. 2 was multiplied elementwise, giving an identical result. The system $C_n x = b$ was then to be solved, were C_n was taken as the above matrix and $b = (5, 6, 7, 8)^T$ giving:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 2 & 3 \\ 3 & 4 & 1 & 2 \\ 2 & 3 & 4 & 1 \end{pmatrix} x = \begin{pmatrix} 5 \\ 6 \\ 7 \\ 8 \end{pmatrix}$$

The correct solution is $x = (0.9, 0.9, 0.9, -0.1)^T$ and the algorithm was able to duplicate this result by applying $\text{FFT}(\Gamma^{-1}(\text{IFFT}(x)))$ where Γ^{-1} is the diagonal matrix consisting of the reciprocals of the eigenvalues. Since this amounts to division of the output of IFFT by the eigenvalues, the matrix cannot be singular or the algorithm will divide by 0 and break the program. Therefore, all of the eigenvalues were compared with 0 after being calculated to prevent a segmentation fault. The algorithm was also able to produce results for circulant matrices and vectors of randomly generated complex numbers of different sizes of n.

The matrix 2-norm was able to be calculated correctly for the test problem above as well. The matrix 2-norm is calculated by finding the square root of the max magnitude eigenvalue of the matrix $C_n C_n^H$. For circulant matrices, the result of this matrix multiplication is also a circulant matrix, and therefore only the first column of the matrix multiplication needs to be calculated. Then the eigenvalue algorithm from above can be used on the output, and the square root of the maximum magnitude eigenvalue can be found. For the test problem, the correct value is 10, which the program was able to find correctly.

6 Conclusions

The key properties of the Discrete Fourier Transform matrix and its inverse where able to be successfully empirically validated using both the normal DFT as well as the FFT algorithms. In terms of property validation, both the DFT and FFT performed with about the same level of error, which was less than 10^{11} in all cases or ver close to zero for double (all complex numbers were represented as 2 doubles).

However, the main difference came when comparing the time to evaluate the same input vector with the DFT versus the FFT. The FFT was significantly faster, as its name implies, and this became even more apparent for large values of n. This difference in computational time is a direct reflection of the difference in order of computation, $O(n^2)$ versus $O(n \log n)$ respectively. One interesting note is that in terms of operations per second, the FFT is actually slower, since the algorithm requires additional array declarations and assignments every time that the recursive input is split into even and odd parts. This might account for some of the error from the true ratio of computations for small n, as well as differences in the times each took to complete the operations. But because the DFT needs to perform almost a full order of computations more than the FFT for the same value of n, the FFT is still significantly faster.

Once the DFT and FFT were implemented, the FFT/IFFT pair was then used to solve a circulant matrix linear system, as well as by being a crucial step in calculating the eigenvalues of that circulant matrix and finding its matrix 2-norm. In all cases, the application of the FFT in that algorithm was able to produce the correct results for that test problem. This was subsequently applied to random circulant matrices of complex values of varying sizes n x n. However, it is difficult to verify the correctness of the output in these cases, since solving a 64x64 system of random complex values without using the approach described here is beyond the scope of this paper. This was done only in order to ensure that an output be achieved if such a system needed to be applied.

7 Program Files

The algorithms were coded in C++ in a .cpp file using notepad++, and compiled and tested using the g++ compiler on "compute2" on the FSU Mathematics servers. This is where all of the timing of the DFT and FFT took place.

References

[1] www.math.fsu.edu/ gallivan/courses/FCM2new/Locked/Sets/set1.pdf

[2] en.wikipedia.org/wiki/Unitary matrix

[3] www.math.fsu.edu/ gallivan/courses/FCM2new/Locked/Sets/set2.pdf

[4] www.math.fsu.edu/ gallivan/courses/FCM2new/Locked/Solutions/solhw2.pdf

[5] "Fast Matrix-Vector Products." Matrix Computations, by Gene H. Golub and Charles F. VanLoan, 4th ed., John Hopkins Univ. Press, 1993, pp. 33–41.