ECSE 316 Assignment 2 Report

Mark Coates

April 03, 2023

Group 17:
Joey Koay (260956108)
Mihail Calitoiu (260972537)

Introduction:

The purpose of this assignment is for students to design and test out two different versions of Discrete Fourier Transform (DFT) in a Python application. The first approach is the brute force approach where we simply implement the DFT formula and its inverse. The second approach is the divide-and-conquer approach which is implemented via the Cooley-Tukey FFT.

The main goal of the Python application is to be able to apply and use DFT to solve a real world problem, more specifically in the image processing domain. Our application supports four functionalities:

1. Display the original image and transformed image by FFT
2. Display the original image and denoised image using FFT
3. Display the original image and five compressed images using different compression levels ranging between 0% and 95%
4. Display the average runtime comparison between DFT and FFT with various sizes from 2^5 x 2^5 to 2^8 x 2^8 (most laptops can only handle until 2^8 x 2^8 before reaching their limits)

Design:

To run the program, the user should send in the command line via the following syntax.

python fft.py [-m mode] [-i image]

Note: python is bounded to python version 3

Both the mode and image options are optional. If none are indicated, the default mode is 1 and the default image is the *moonlanding.png* that was provided with the assignment instruction.

In our implementation of the application, we created one file (fft.py) with a few functions to help achieve the result. A high overview of the general structure is provided below

1. Input Parsing: After the user successfully runs the application, the program checks whether a mode and image was indicated. This is done by the parsingArg() function where we pass the system's arguments (with the help of sys library).
2. Mode Selection: A mode is selected and ran based on the result in *Input Parsing* using the modeOption() function.
    a. Mode 1: Two images are plotted: original and image underwent FFT transformation.
    b. Mode 2: Two images are plotted: original and denoised image. In order to transform the original image to the denoised image, the image first underwent FTT. All high frequencies are then set to zero before applying the inverse of FTT. The resulting image is then the denoised version of the original image.
    c. Mode 3: Six images are plotted: original and five images that are compressed. The compression is done by setting some of the Fourier coefficients to zero before applying the inverse of FTT. This process is repeated five times at different compression levels between 0% and 95%.
    d. Mode 4: One line graph is computed with two lines measuring the average runtime in seconds for the brute force algorithm (naive - translation of equation DFT) and the divide-and-conquer (fast - Cooley Tukey FFT). To compare the runtimes, four tests are ran with 2^5 x 2^5, 2^6 x 2^6, 2^7 x 2^7 and 2^8 x 2^8 problem size. A time is started before and after the algorithms to track the amount of time taken. Finally, a graph with two lines is plotted along with error bars proportional to the standard deviation.
3. Displaying Figures: Depending on the mode chosen by the user, the corresponding figures are then displayed for users to see the results using matplotlib.

Below is a more in depth explanation of the program's functions and the motivation behind the design and structure of the code.

In terms of the overall structure of the code, we decided that each mode should have its own function so that the code is easier to understand.

ParsingArg function:
The parsing function takes in one argument (argv) and returns two variables (mode, image). The input is a list of arguments that was used when the user initially ran the program. A for loop is then used to check whether the desired mode and image are entered, if it's not, then mode 1 is used along with the default image (moonlanding.png). If the user entered the wrong mode (error checking), a print statement is used to notify the user that the mode entry is invalid.

ModeOption function:
This function simply redirects the user's request to the correct function to process their request. It takes in two arguments: the mode and the image path.

Mode1 function:
The first mode takes in one argument (image_path - the image's path) and shows two images (the original image and the Fourier transformed Image) using the python library matplotlib. First, the image was turned into an array using matplotlib's pyplot. Then, it underwent fast DFT, which created another array. Since

the Cooley Tukey requires its inputs to be powers of two, the method resizeIMG() is called to ensure that the array is indeed a power of two by calculating the next lowest powers of two on the width and height of the image, and the resizing to the new dimension using cv2.resize. Lastly, both the original and transformed image is displayed.

Mode2 function:

The second mode takes in one argument (image_path - the image's path) and shows two images (the original image and the denoised image) using matplotlib. First, the image is turned into an array using matplotlib's pyplot. Then, it undergoes the fast DFT, which creates another array. Since the Cooley Tukey requires its inputs to be powers of two, the method resizeIMG() is called to ensure that the array is indeed a power of two. The high frequencies are then set to 0 as shown below in Figure 1.

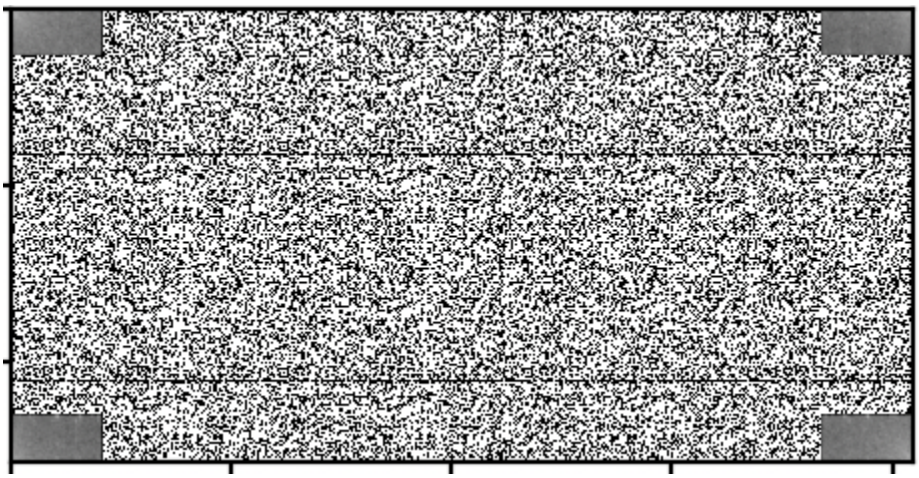


Figure 1: The Image Post-denoising and FFT (Before Applying Inverse FFT)

Then, the image is inverted using the inverse algorithm of fast DFT. After that is done, the algorithm chooses to remove the high frequencies as it is applying a low pass filter. After some trial and error, we determined that the threshold is at 10% as it shows the best result (as shown in our experiment part). Lastly, both the original and denoised image are displayed.

Mode3 function:

The third mode takes in one argument (image_path - the image's path) and shows six images (the original image and five compressed images each with different compression levels) using matplotlib. First, the image is turned into an array using matplotlib's pyplot. Then, it undergoes the fast DFT, which creates another array. Since the Cooley Tukey requires its inputs to be powers of two, the method resizeIMG() is called to ensure that the array is indeed a power of two. Then five compression levels are chosen between 0% and 95%. Since five compressed images are shown, the chosen compression levels are 19, 38, 57, 76, 95 percent to showcase the compression levels linearly. Then, the image is inverted using the inverse algorithm of fast DFT. Lastly, the original and five compressed images are displayed.

Mode4 function:

The fourth mode returns a graph with two lines comparing the average runtime between the slow DFT and fast DFT with 2^5 x 2^5, 2^6 x 2^6, 2^7 x 2^7 and 2^8 x 2^8 problem size. The laptop that this experiment was conducted on could only handle up to 2^8 x 2^8 problem size; hence, the problem size stops at that size. In order to do so, arrays with random values are first generated, and then looped 10 times to calculate the average runtime for each problem size between the two different algorithms. Before each algorithm starts, the current time is saved. After the algorithm has stopped, the current time is saved again. The difference in time will be the time it takes to run one algorithm. Then, after the algorithms have successfully ran ten times, we are able to calculate the average run time along with its standard error. The standard error is calculated using the following formula:

$$standard\ error\ =\ standard\ deviation\ /\ \sqrt{sample\ size})$$

After iterating through all problem sizes, a graph is then produced to compare the naive (slow) and fast DFT algorithm along with the corresponding standard error.

Compression function:

This function takes in three arguments (img - NumPy ndarray, compressedNum - compression level, numPixels - number of pixels in total) and returns the compressed image in the NumPy ndarray format. Then, the upper and lower bounds are determined with the help of finding the midpoint of the compression level and its complements. Then, to compress the image, a NumPy's logical_or function is applied where all pixels that are either below the lower bound or above the upper bound are set to 0.

resizeIMG function:

This function takes in one argument (img - NumPy ndarray) and returns a NumPy ndarray that is a power of 2. The height and width of the image is first obtained, and then the closest power of 2 is obtained before resizing (using cv2) to the new dimensions. This is a helper function that will ensure that the size of the input to the FFT function is a power of two. This is because Cooley Tukey FFT is a divide and conquer approach that will only work on inputs that have a power of two size.

Closest_power_of_2 function:

This function takes in one argument (num - integer) and returns an integer that is the next largest power of 2. The next largest power of 2 is simply $2^{ciel(log_2 num)}$ as the powers and logarithms are inverse of each other, and therefore, rounding up the logarithmic answer will allow us to have the next largest power of 2.

DFT_1d function:

This function takes in one argument (array - NumPy ndarray) and returns a NumPy ndarray that has undergone the naive (slow) DFT. The algorithm is a direct implementation of:

$$X_k\ =\ \sum_{n=0}^{N-1} x_n e^{\frac{-i\,2\,\pi}{N}kn},\ for\ k\ =\ 0, 1, 2, \ldots, N\ -\ 1$$

Since this equation utilizes imaginary numbers, one of the fastest and most efficient way on Python is to first compute the imaginary separately, and then use NumPy's dot function to perform matrix multiplication between x_n and $e^{\frac{-i\,2\,\pi}{N}kn}$.

DFT_1d_inverse function:

This function takes in 1 argument (array - NumPy ndarray) and returns a NumPy ndarray that has undergone the inverse naive (slow) DFT. The algorithm is a direct implementation of:

$$x_n = \frac{1}{N} \sum_{n=0}^{N-1} X_k e^{\frac{i2\pi}{N}kn}, \; for\; k = 0, 1, 2, \ldots, N-1$$

Since this equation utilizes imaginary numbers, one of the fastest and most efficient way on Python is to first compute the imaginary separately, and then use NumPy's dot function to perform matrix multiplication between X_k and $e^{\frac{i2\pi}{N}kn}$.

FFT_1d function:

This recursive function takes in one argument (array - NumPy ndarray) and returns a NumPy ndarray that has undergone the fast DFT (Cooley Tukey). The algorithm implements the Cooley Tukey function as shown in the PDF ($X_k = X_{even} + e^{\frac{-i2\pi}{N}k} X_{odd}$). Our team determined that when the array size is 16 (2^4), the algorithm will use the naive algorithm. This implementation was a translation of a pseudocode from Wikipedia's Cooley-Tukey FFT algorithm.

FFT_1d_inverse function:

This recursive function takes in one argument (array - NumPy ndarray) and returns a NumPy ndarray that has undergone the inverse fast DFT (Cooley Tukey). The algorithm implements the Cooley Tukey function with the consideration that the inverse is multiplied by 1/N and the exponents are opposites in sign. Similar to the FFT_1d function, our team uses the naive algorithm when the array size is 16 (2^4). This implementation was inspired by the pseudocode from Wikipedia's Cooley-Tukey FFT algorithm.

FFT_2d function:

This recursive function takes in one argument (array - NumPy ndarray) and returns a NumPy ndarray that has undergone the fast DFT for both directions. This algorithm implements the 2D DFT equation indicated in the assignment. As mentioned, in the assignment, "the inside brackets is a 1D DFT of the rows of the 2D matrix of values f and that the outer sum is another 1D DFT over the transformed rows performed along each column." Therefore, the approach is similar as reflected in the two for loops seen in the code. Since this is the fast algorithm, when calling the DFT algorithm, it calls the fast 1d algorithm.

DFT_2d function:

This recursive function takes in one argument (array - NumPy ndarray) and returns a NumPy ndarray that has undergone the naive DFT for both directions. The logic for this code is very similar to the FFT_2d function with the exception that the DFT uses the naive 1D algorithm instead of the fast 1D algorithm. This algorithm implements the 2D DFT equation indicated in the assignment. As mentioned, in the assignment, the inside brackets is a 1D DFT of the rows of the 2D matrix of values "f" and that the outer sum is another 1D DFT over the transformed rows performed along each column. Therefore, the approach is similar as reflected in the two for loops seen in the code.

FFT_2d_inverse function:

This recursive function takes in one argument (array - NumPy ndarray) and returns a NumPy ndarray that has undergone the inverse fast DFT for both direction. This algorithm implements the 2D DFT equation indicated in the assignment with the exception that when a DFT algorithm is needed, it calls the fast inverse 1D algorithm. As mentioned, in the assignment, the inside brackets is a 1D DFT of the rows of the 2D matrix of values f and that the outer sum is another 1D DFT over the transformed rows performed along each column. Therefore, the approach is similar as reflected in the two for loops seen in the code. Since this is the fast algorithm, when calling the DFT algorithm, it calls the fast algorithm.

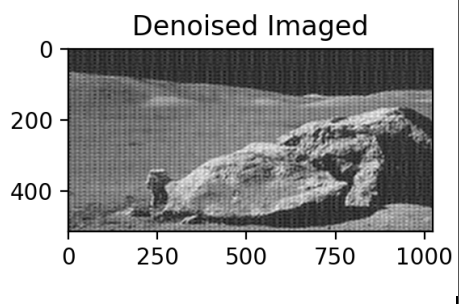
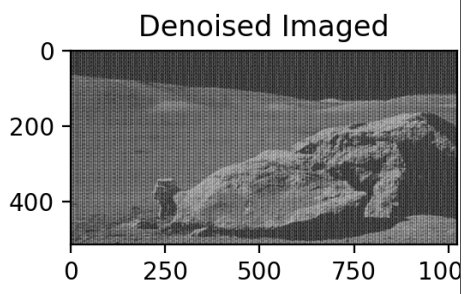
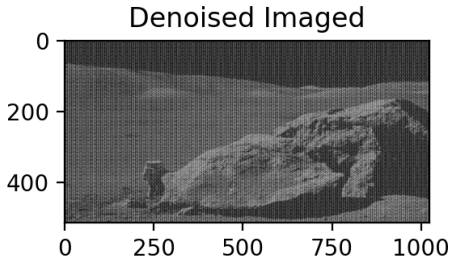
Testing:

The testing phase of our program can be categorized into two different categories: testing against pre-built algorithms and manual testing.

In python's NumPy library, there are built-in discrete Fourier transform algorithms that can be called. The algorithms included 1D discrete fourier transform and its inverse, as well as 2D discrete fourier transform and its inverse. To ensure that our algorithm is implemented correctly, we ran both NumPy's prebuilt algorithm, as well as our program's algorithm. Results were compared and thus this ensured that the algorithms were implemented correctly.

For mode 2, a low pass filter was applied to ensure that high frequencies are removed to denoise the image. We tested three frequencies and noticed a decline in the quality of the denoised image. Table 1 below shows that as the frequency is increased, the denoised image becomes darker (lower luminosity) which decreases the overall quality of the image. As such, choosing a smaller frequency, specifically 0.1, yields a bright and denoised image.

Table 1: Frequencies and Denoised Images

Freq: 0.1	Freq: 0.2	Freq: 0.3
		

To ensure that compression works as expected, manual testing was conducted. The purpose of compression is to be able to save memories. In our code, we decided to space out our 5 compressed levels evenly between 0 and 95. The compression levels chosen are 19, 38, 57, 76, 95. The Fourier transform matrix of coefficients for each compression level are then saved into a txt file. As observed, the coefficients needed for each compression level are 419430, 325058, 220202, 125830, 20972 respectively, compared to 524288 for the original image resized image. Note that the smaller number of coefficients there are, the less amount of memory is needed for the image. Therefore, this ensures that our compression algorithm works.

To test whether the mean runtime for the naive and fast algorithm is computed as intended, results were compared to online searches for DFT vs FFT runtime complexity. As seen below (Figure 2), the naive algorithm (DFT) grows exponentially quicker compared to the fast algorithm (FFT). Additionally, we know that the runtime is as expected because FFT uses a divide and conquer approach with a time complexity of O(nlogn) compared to the brute force approach with a time complexity of $O(n^2)$ (which will be explained more elaborately in the analysis section of the report).

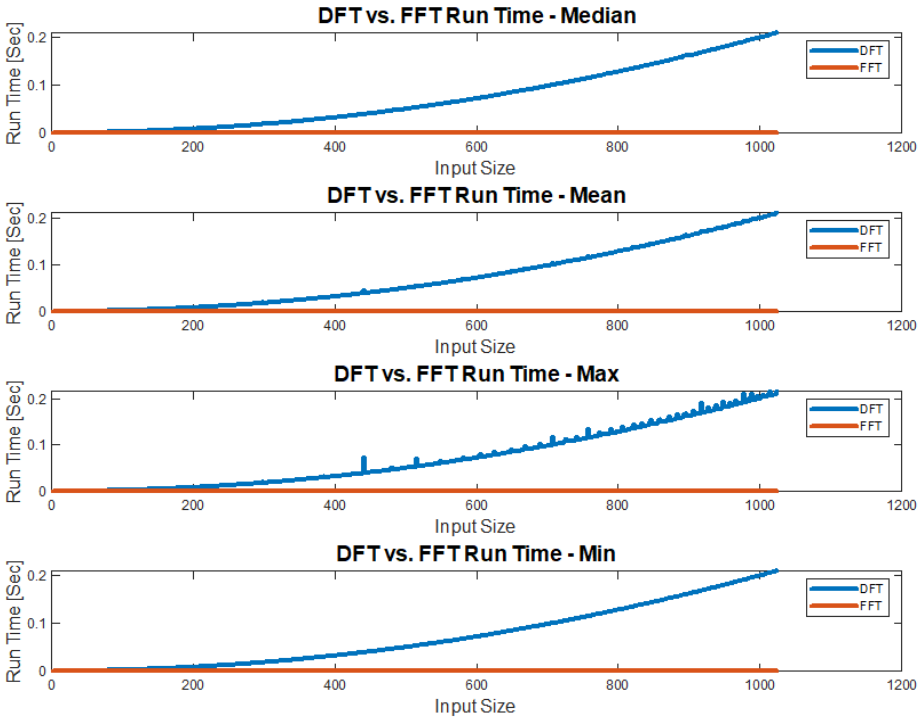


Figure 2: DFT vs FFT runtime complexity taken from [1]

Analysis:

For the brute force method (naive 1D DFT), it is $O(n^2)$ because we need to have a nested loop to compute X_k. The nested loop is caused by the equation $X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-i\,2\pi}{N}kn}$, $for\ k = 0, 1, 2, \ldots, N-1$ having to loop through both n and k for N amount of time. Therefore causing the time complexity to be $O(n^2)$.

For divide and conquer methods, we can use the master theorem to solve for its complexity. The master theorem equation is $T(n) = a \times T(n/b) + f(n)$ where a is the number of subproblems, n/b is the size of each subproblem and $f(n)$ is the cost of dividing and combining the subproblems. [2]

Since the 1D FFT algorithm splits the array into two at each recursion level, $a = 2$, and since each subproblem has halved the length of the previous problem, $n/b = n/2$. Additionally, we know that the cost of dividing and combining the subproblem is O(n), because we need to assemble the resulting back together using a for loop with a range of n/2 rounded down. Therefore, we have that $T(n) = 2 \times T(n/2) + O(n)$. Since the $f(n) = \Theta(n^{log_2 2})$ therefore $T(n) = \Theta(n^{log_2 2} log(n)) = \Theta(nlog(n))$ because master theorem's second statement states that if $f(n) = \Theta(n^{log_b a})$ then $T(n) = \Theta(n^{log_b a} log(n))$.

The 2D FFT algorithm works very similarly to the 1D FFT with the slight difference that we are working with n x m matrix instead where n is the number of columns and m is the number of rows. As mentioned in the PDF, the term inside the brackets for a 2D FFT is a 1D FFT of the rows of the 2D matrix of values "f", and its output sum is another 1D FFT over the transformed rows performed along each column. Therefore, we can conclude that the same value will be visited and undergo 1D FFT twice (once with the column loop, once with the row loop). Since a 1D FFT has the runtime complexity of $O(nlog(n))$ where n is the size, we know that the column loop will take $n\,\Theta(mlog(m))$ and the row loop will take $m\,\Theta(nlog(n))$.

Together, it will take $n\,\Theta(mlog(m)) + m\,\Theta(nlog(n)) = nm\,\Theta(log(m)) + mn\,\Theta(log(n)) = mn\,(\Theta(log(n)) + \Theta(log(n)))$. Therefore, the runtime complexity of the 2D FFT is $mn\,(\Theta(log(n)) + \Theta(log(n)))$.

Experiment:

1. Mode 1:
   Command line input:

```
python fft.py -m 1
```

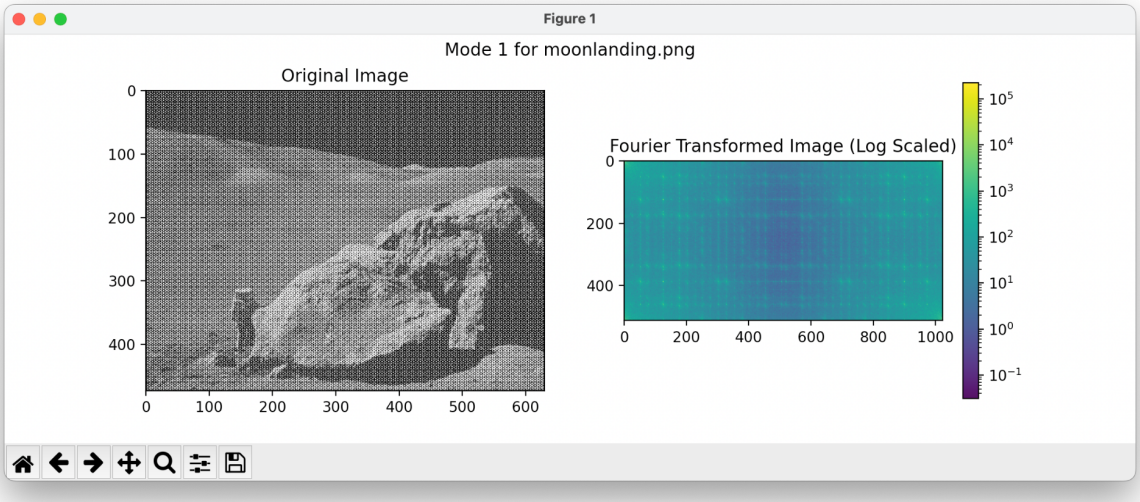


Figure 2: Mode 1 Original Image and Output

Mode 1 shows the original image as well as the FFT Image that is log-scaled to improve visibility. As expected, the log-scaled FFT image is darker in the middle when compared to the corners, showing that the higher frequencies are in the middle.

2. Mode 2:
   Command line input:

```
python fft.py -m 2
```

Command line output:

```
There are 489805 number of zeros, which is 93.42288970947266% of the picture
```

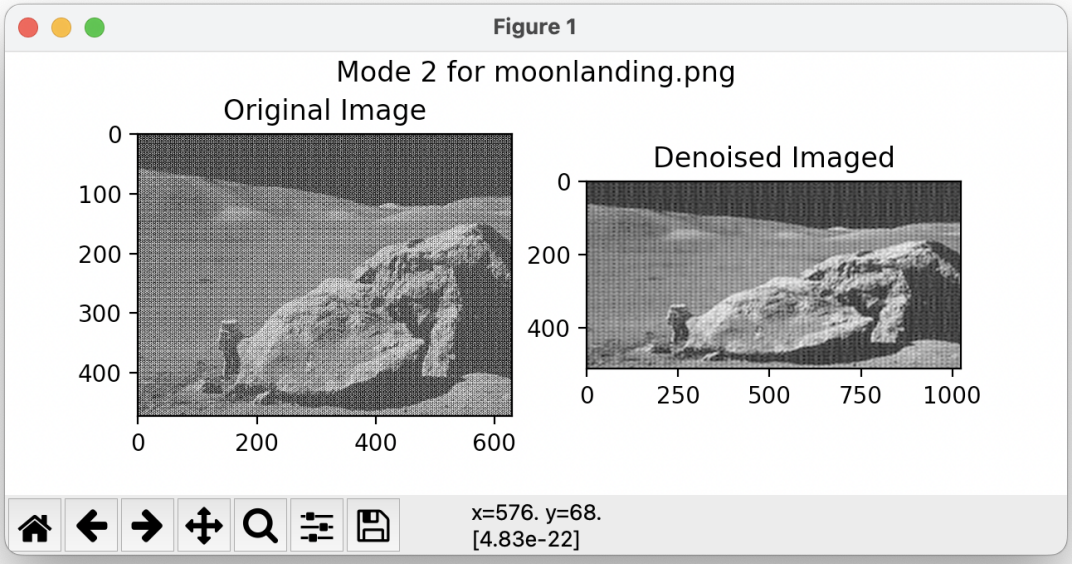


Figure 3: Mode 2 Original Image and Denoised Output

The output of this test is the original and denoised image. The result of the test is as expected, specifically the image is noticeably clearer and has a higher contrast. Additionally, the number of zeros in the original image is outputted in the command line (as per the PDF's instructions).

3. Mode 3:
   Command line input:

```
python fft.py -m 3
```

Command line output:

```
There are 424673 non-zero Fourier coefficients when the compression level is 19
There are 325058 non-zero Fourier coefficients when the compression level is 38
```

```
There are 225443 non-zero Fourier coefficients when the compression level is 57
There are 125829 non-zero Fourier coefficients when the compression level is 76
There are 26214 non-zero Fourier coefficients when the compression level is 95
```

File Size:

```
Mode3_0.txt (original picture): 524288 lines
Mode3_19.txt (compression level: 19%): 419430 lines
Mode3_38.txt (compression level: 38%): 325058 lines
Mode3_57.txt (compression level: 57%): 220202 lines
Mode3_76.txt (compression level: 76%): 125830 lines
Mode3_95.txt (compression level: 95%): 20972 lines
```

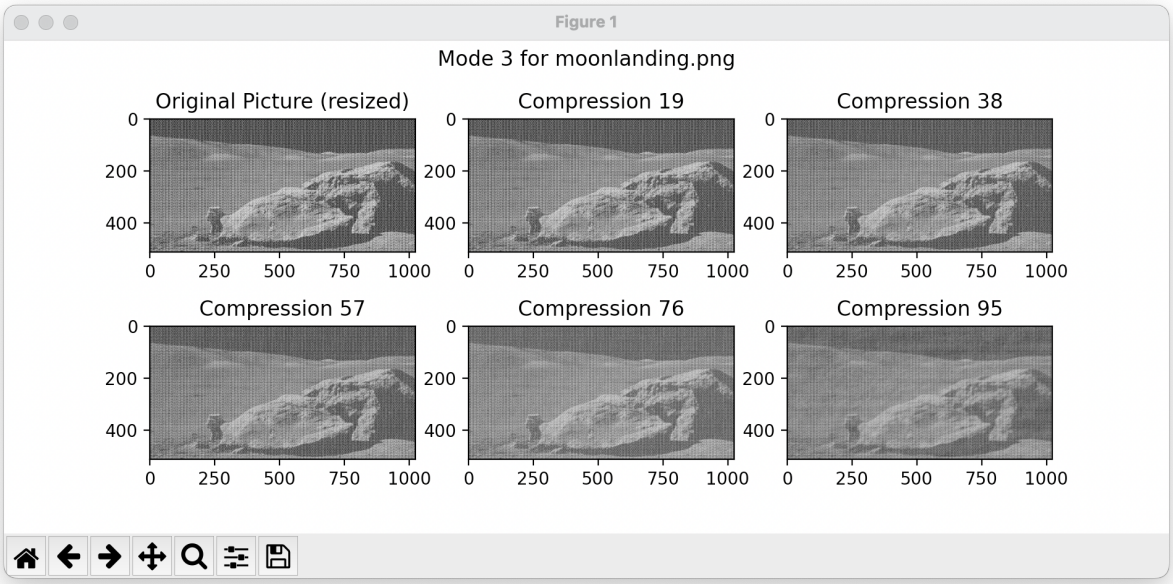


Figure 4: Mode 3 Original Image and Compressed Outputs

The output of this test is the original and five compressed images (increasing compression level) and is as expected. The command line output indicates that the number of lines that make up the image decreases as the compression level is increased, which is logical. Additionally, qualitatively, the level of detail of the images also decreases with higher compression levels. This also is as expected.

4. Mode 4:

Command line input:

```
python fft.py -m 4
```

Command line output:

```
=== 2^5 * 2^5 ===
The naive method's mean: 0.10654304027557374
The naive method's variance: 2.0192942139374283e-06
The FFT method's mean: 0.05672721862792969
The FFT method's variance: 2.635883311086218e-07
The naive error is: 0.0004493655765562632
The fast error is: 0.00016235403632451574

=== 2^6 * 2^6 ===
The naive method's mean: 0.841001009941101
The naive method's variance: 0.00018022775108704535
The FFT method's mean: 0.23654861450195314
The FFT method's variance: 5.260082701624924e-05
The naive error is: 0.004245323910928886
The fast error is: 0.0022934870179761044

=== 2^7 * 2^7 ===
The naive method's mean: 6.656638526916504
The naive method's variance: 0.02410602496755928
The FFT method's mean: 0.9687572479248047
The FFT method's variance: 0.00022296984514241555
The naive error is: 0.04909788688687047
The fast error is: 0.004721968288144421

=== 2^8 * 2^8 ===
The naive method's mean: 53.0943442106247
The naive method's variance: 0.26742556099898185
The FFT method's mean: 3.999457907676697
The FFT method's variance: 0.0005579892966017042
The naive error is: 0.16353151408795244
The fast error is: 0.007469868115313041
```

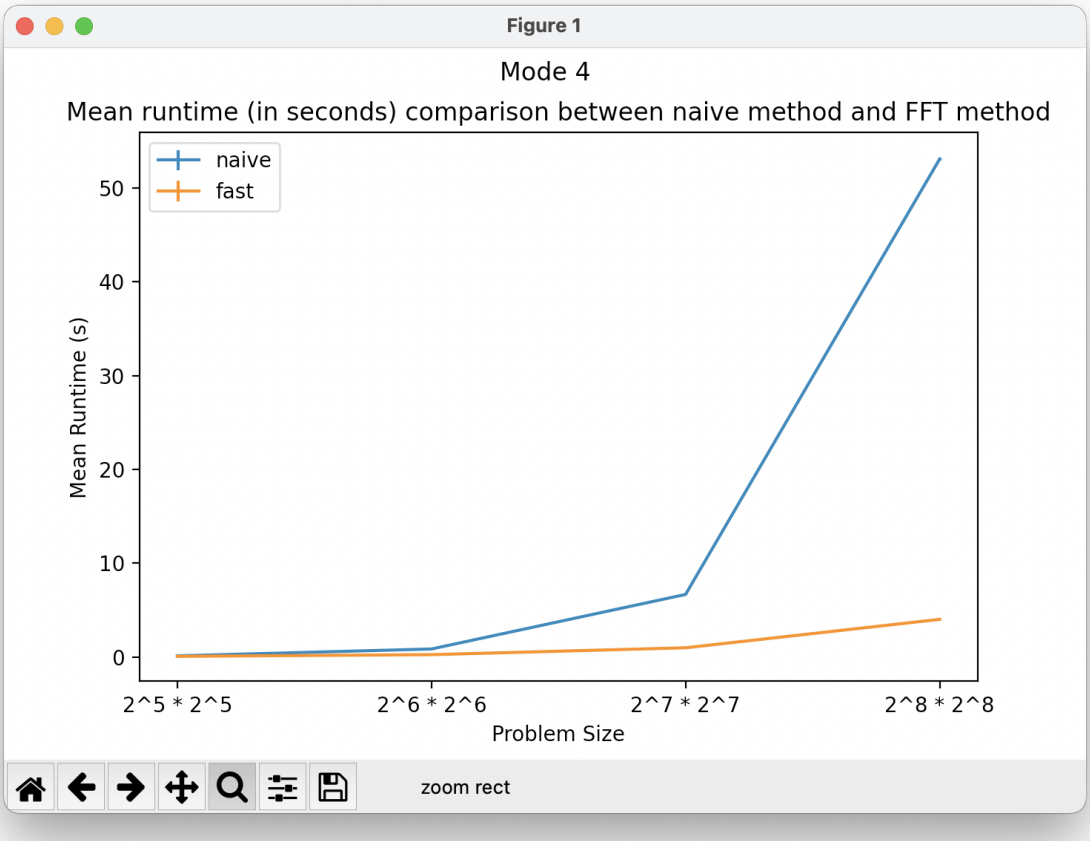


Figure 5: Mode 4 Run Time Analysis of Naive and FFT methods

As seen in the above graph, this shows that the naive (DFT) algorithm runs exponentially faster than the Cooley Tukey's fast algorithm (FFT). This is because a 1D DFT has a complexity of $O(n^2)$ whereas 1D FFT has a complexity of O(nlogn), therefore, causing 2D DFT to have a complexity of $O(n^2m^2)$ and 2D FFT with the complexity of O(nm (log(n) + log(m))).

Discussion:

During this experiment, we were able to observe and apply the Fourier Transform to actual applications.

An observation we had in mode 2 was that the threshold frequency for denoising has a direct correlation to the luminosity of the image. As seen in Table 1, an increase in the threshold frequency will cause the luminosity to be lower.

In addition, we observe that compressing the images in mode 3 helps save the memory needed, however, it comes at a cost of the image's clarity (noticeable decrease in the detail of the image). The larger the compression level is, the less memory space is used and the blurrier the picture gets. As there is no optimal compression level, the user must decide between prioritizing memory space and/or clarity.

An observation we had in mode 4 particularly was that the FFT approach is significantly faster than the naive DFT approach, especially at larger sizes. The closer the problem size is to 2^4, the closer the runtimes are, because the FFT approach's base case is 2^4.

The main challenge for implementing our application was to determine the correct value used for denoising. Thus, the application was run numerous times to ensure that the best value is picked by identifying which values returned the least noisy image. As for the compressing experiment, numerous compression values were used, and it was observed that a higher compression level results in less memory usage, however, it comes at a cost of the image's clarity (i.e. contrast and luminosity). In the end, the user can prioritize either memory usage or image clarity (or try to achieve a combination of the two) - and therefore can pick a compression level that best fits their needs.

Overall, we observed that the Fourier Transformation can be used in a practical way to denoise an image without sacrificing contrast or luminosity. Additionally, we observed that the frequency threshold of FTT was correlated to the image's brightness, which gives the Fourier Transformation another practical use of modifying an image's brightness.

The Fourier Transformation can also be successfully used to compress an image without dramatically lowering its contrast or luminosity (unless the compression level is very high).

References:

[1]: Abhinav JainAbhinav Jain 12377 bronze badges, RoyiRoyi 48.1k44 gold badges169169 silver badges216216 bronze badges, & user28715user28715. (1965, June 1). *FFT vs DFT run time comparison (complexity analysis) in MATLAB*. Signal Processing Stack Exchange. Retrieved April 3, 2023, from https://dsp.stackexchange.com/questions/51516/fft-vs-dft-run-time-comparison-complexity-analysis-in-matlab

[2]: Su, J. (n.d.). *CS 161 Lecture 3 - stanford university*. Retrieved April 3, 2023, from https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture3.pdf