ECSE 316 Assignment 2 Report
Signals And Networks

Project Group 1

Elias Al Homsi (elias.alhomsi@mail.mcgill.ca), 260797449
Kamy Moussavi (kamy.moussavikafi@mail.mcgill.ca), 260807441

Date: 3/10/2020

Introduction

In this assignment, we will explore how to programmatically implement fourier transformations in two main approaches. The first approach is the traditional approach which loops over the whole values without recursion and thus has a high time complexity. The other approach exploits the fact that Fourier transformation can be split into two sub arrays of evens and odds which speeds up the calculations and lowers the complexity to a logarithmic factor. The main idea is to split the problem into two subproblems (which is a very familiar concept from divide and conquer) then we can merge those two subproblems solutions in order to generate the solution for the higher level problem and therefore solves the initial problem given the solution of the two subproblems each half the size of the original one. In order to stop the recursion, once we hit a problem size relatively small we can simply just run the slow normal algorithm which should run in constant time given the small constant problem size. Master's approach can be used to calculate the time complexity of tree algorithms like this one in order to prove that the fast algorithm has a lower complexity.

In addition, the assignment also exposes us to image manipulation and processing such as image denoising and compression. We have made the assumption that images handled by this program should be in grayscale as advised by the TA. The image processing pipeline passes through many stages including: image padding to the nearest power 2, image denoising by removing high frequencies, and image compression by selecting only the outlier values. We are going to discuss each of the algorithm design choices by providing a detailed explanation for each.

Program Design

Overview:

DFT algorithm:

The algorithm implementation followed closely the assignment description text. We have designed both algorithms (fast and slow fourier transforms) in such a way to highlight the theoretical information we have covered in lectures. The fast implementation relies on the slow implementation for small problem sizes.

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

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

Figure 1: slow implementation, Source: assignment text

$$\begin{aligned}
X_k &= \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi}{N}kn} \\
&= \sum_{m=0}^{N/2-1} x_{2m} e^{\frac{-i2\pi}{N}k(2m)} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{\frac{-i2\pi}{N}k(2m+1)} \\
&= \sum_{m=0}^{N/2-1} x_{2m} e^{\frac{-i2\pi}{N}k(2m)} + e^{\frac{-i2\pi}{N}k} \sum_{m=0}^{N/2-1} x_{2m+1} e^{\frac{-i2\pi}{N}k(2m)} \\
&= \sum_{m=0}^{N/2-1} x_{2m} e^{\frac{-i2\pi}{N/2}k(m)} + e^{\frac{-i2\pi}{N}k} \sum_{m=0}^{N/2-1} x_{2m+1} e^{\frac{-i2\pi}{N/2}k(m)} \\
&= X_{even} + e^{\frac{-i2\pi}{N}k} X_{odd}
\end{aligned}$$

Figure 2: fast implementation, Source :assignment text

For the two dimensional conversions and using the slow approach we have used 4 for loops to highlight the theoretical implementation. The four for loops comes from the fact that there are 4 variables that we need to iterate over l, k, m, and n.

$$F[k,l] = \sum_{n=0}^{N-1}\sum_{m=0}^{M-1} f[m,n] e^{\frac{-i2\pi}{M}km} e^{\frac{-i2\pi}{N}ln}$$
$$= \sum_{n=0}^{N-1}\Big(\sum_{m=0}^{M-1} f[m,n] e^{\frac{-i2\pi}{M}km}\Big) e^{\frac{-i2\pi}{N}ln}$$
$$\text{for } k = 0, 1, \ldots, N-1 \text{and } l = 0, 1, \ldots, M-1.$$

Figure 3: two dimensional slow implementation, Source: assignment text

For the two dimensional fast implementation we have done a 1 dimensional fast fourier transformation for each column and then a 1 dimensional fast fourier transformation for each row. The result as expected is a two dimensional fast fourier transformation.

Image padding:

We had a challenge of passing 2 dimensional images to fft. However, the dimensions of the input had to be a power 2 number. Therefore, we had two options: either crop to the nearest power 2 or pad with zeros up to the nearest power 2. In other words, round down or round up. We choose the latter to preserve the information in the image.

Image Denoising:

When looking at the fft information from **mode 1** we realized that the best way to denoise an image is to remove the high frequencies in the block around the middle of the fast fourier transformation output which is usually associated with the bad noise along the image. Once we zeroed out those values in the middle the image was denoised and we had good results. We have the idea inspired from [2].

Image compression:

For image compression we have tested multiple techniques and approaches including techniques similar to what we have in denoising. However, the assignment text suggested a technique related to the percentile of frequencies that belong in the image and we were interested in that. We realized that the outlier values are the most important values in keeping the image legible while reducing the size of the image specially the very low frequencies are essential to keep. Therefore we have come up with a technique that keeps only edge values (the values outside the two lines of the cut) similar to z-scores that we learned during statistics.

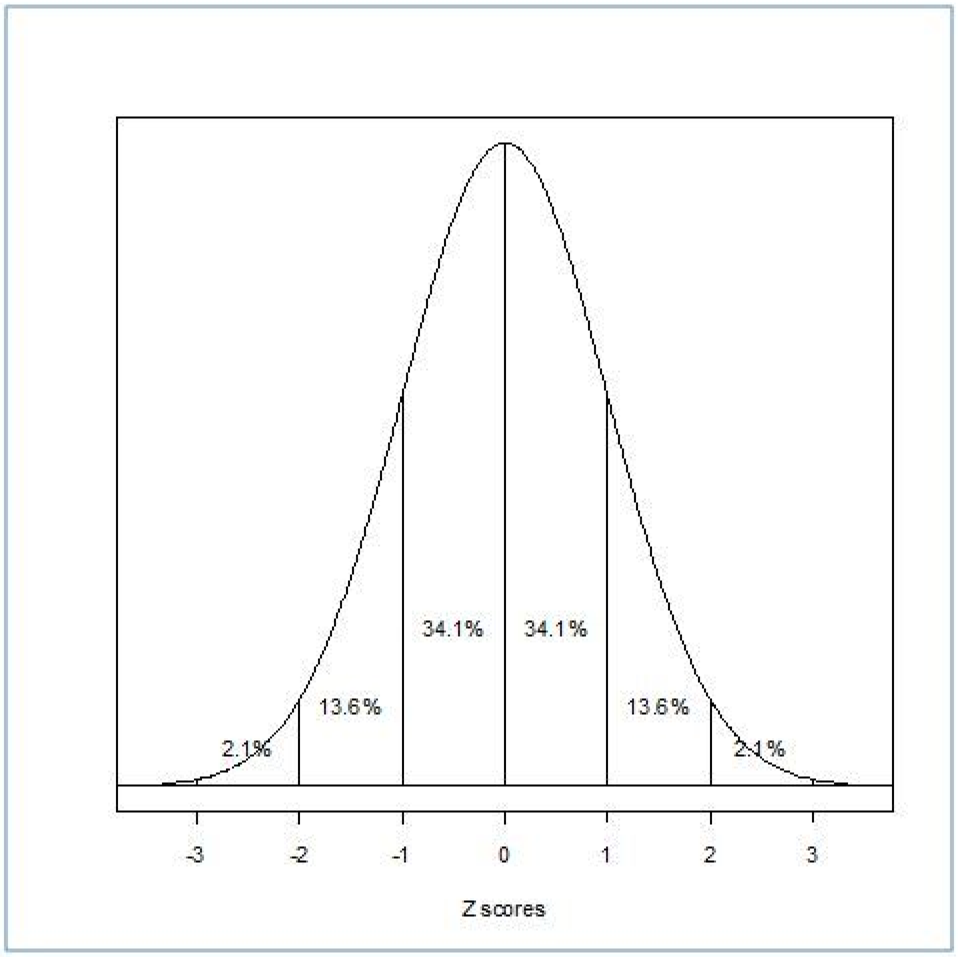


Figure 4: Normal distribution with z scores cuts according to percentiles.

The idea is to keep very low frequencies and very high frequencies. Imagine the image input follows the bell curve hypothetically, then a compression scheme would be simply keeping the low frequencies before the lower line and the higher frequencies after the higher line while disregarding what's in between the lines. The figure above shows such an approach for 3 different compression levels. This is done through the help of **np.percentile**
In the program we have used this technique to provide compression results for the following values: [0, 14, 30, 50, 70, 94]. In this sense 14% compression means that zero out 14% of the data and use the rest. Similarly, 94% compression means only 6% of the data.

Saving csr matrices:

We have using **scipy.sparse** to save sparse matrices in csr format. The matrices are provided in npz file formats that can be loaded through scipy.sparse.load_npz method.
We have noticed dramatical decrease in the sizes of the files required to save the sparse matrices for higher compression which confirms our approach.

Classes Structure:

- **dft.py:** This program contains the mathematical infrastructure necessary for this assignment. It contains 8 different functions for (one, two dimensions), (fast, slow) and finally (forward, inverse). The module also contains a test method that tests the correctness of each of the 8 methods above and ensures the correctness of the transforms.

- **fft.py:** This module contains the logic required for image manipulation and processing. It also uses the argparse library in order to parse command line arguments. This is also where the code for each mode is specified.

- **README.md:** this file contains readme and instructions

Testing

Strategy:

The program consists of two clear modules that are separated by functionality. You can think of them as two layers; the first one is the dft.py which sets up the basics of math functions needed for this assignment, namely: fourier transformations in all kinds (forward, inverse) , (one dimension, two dimensions) and (fast, slow). Therefore, we have 8 different functions to test here. The test strategy for this module is simple. We can use **np.allclose** to measure how close the results of these functions to the standard fft implementation in numpy. A sample test would look like
np.allclose(myoutput, np.fft(input))
Therefore, the module **dft.py** contains a test method that would run tests against all 8 methods to ensure the right implementation of the mathematical infrastructure required for this assignment. Please refer to **test()** in **dft.py** for additional information

Once the first module was tested against different problem sizes and we ensure the right implementation of the fourier transformation we can proceed with building on top of that in order to transform, denoise and compress images using the methods in **dft.py**. The module **fft.py** contains the logic required to run the command line argument as specified in the assignment text, namely:

```
usage: fft.py [-h] [-m MODE] [-i IMAGE]

optional arguments:
  -h, --help  show this help message and exit
  -m MODE     Mode of operation 1-> fast, 2-> denoise, 3-> compress and
save save 4-> plot

  -i IMAGE    image path to work on
```

And hence in order to test **fft.py** we have to test all of the 4 modes manually by passing different image sizes and see if the image gets padded correctly.

To test these modes we have run all four modes against images of different sizes, some of them had dimensions of power 2 while others had to be padded in order to ensure correct running on the dft methods on them. Testing the 4th modes is tricky since the run time is highly dependent on the laptop capability and how much load the machine has at any one point. So we made sure to run the 4th mode on different days with different loads and multiple runs (the assignment suggests 10 runs for each problem size) and we have used 10 runs to have better statistical data to show.

Analysis:

DFT Time Complexity:

The runtime of the naïve 1D DFT algorithm is O(N^2). This is easily derived from the fact that each term in X_k requires a sum of N terms O(N), and the vector X_k contains N terms itself, thus giving a complexity of O(N^2).

In the 2D case, observe that each term in *F[k,l]* is an element of column M multiplied by all elements in row N giving complexity O(M*N), and the output *F[k,l]* itself has size M*N elements, thus resulting in a complexity of O($(M*N)^2$). If M = N, the complexity is O(N^4).

FFT Time Complexity:

The complexity of the Cooley-Tukey FFT algorithm can be derived as follows. Let T(N) be the runtime complexity of FFT with N inputs. Each recurrence of the FFT gives the following:

$$T(N) = 2 * T(N/2) + O(N)$$

Where the term 2 * T(N/2) represents the complexity of X_{even} and X_{odd}, each having half the input size of the initial FFT, and where the O(N) represents the multiplication of the complex term with X_{odd}. Observe that the T(N/2) reaches the base case N = 1 after $\log_2$(N) recurrences, and at each recurrence we are adding a O(N) operation, resulting in a O(N * log(N)) complexity. Furthermore, using Master's algorithm, it is clear to see that this recurrence is O(N * log(N)).

In the 2D case of FFT, we simply have to apply the 1D FFT in each element of rows and columns. Thus, the complexity of a 2D FFT with input size M*N is given by:

$$O(N * (M * log(M)) + M * (N * log(N))$$
$$= O(M * N * (log(M * N)))$$

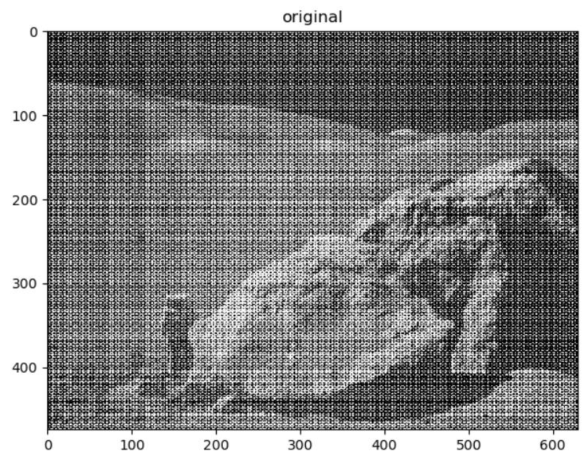
If M = N, then the complexity of 2D FFT is given by:

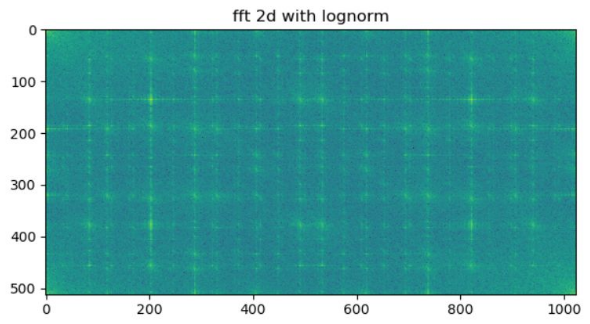
$$= O(N^2 * (log(N^2)))$$

Experiment

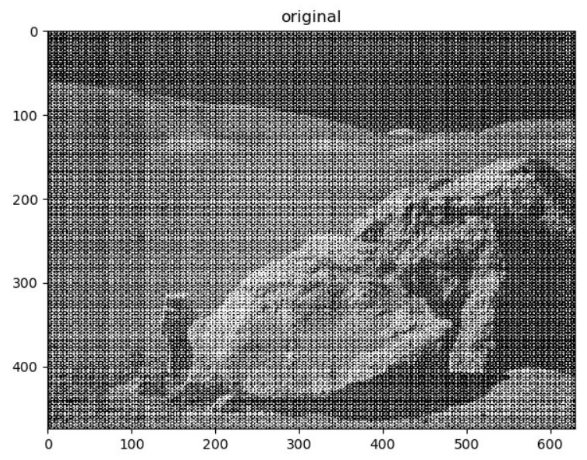
Mode 1:

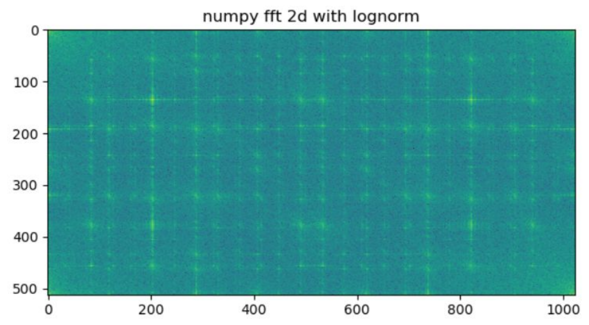
The result of our FFT transform algorithm is shown below:





Below is the result of the numpy np.fft.fft2 algorithm:



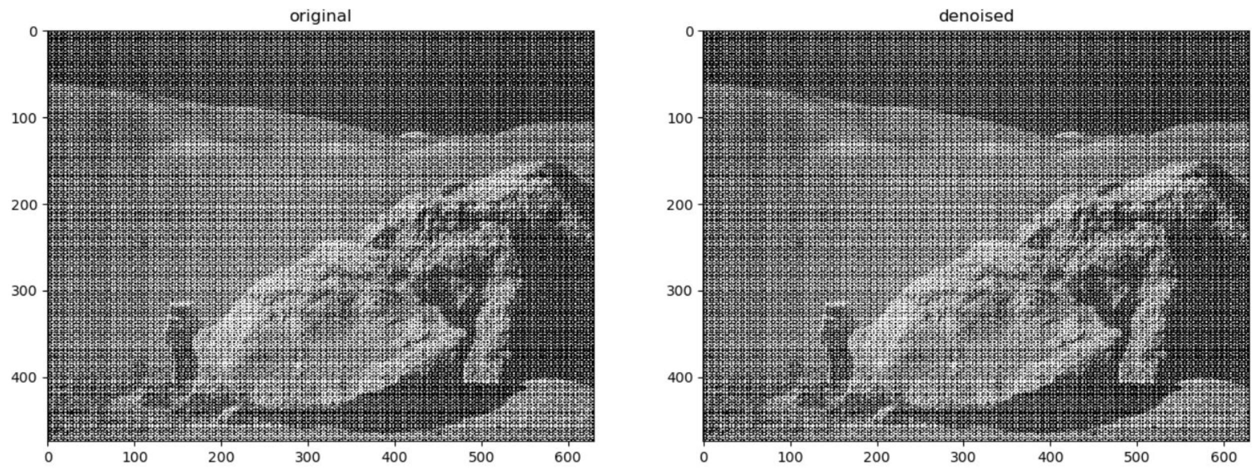


These results are nearly identical, thus we can deduce that our implementation of the Cooley-Tukey FFT algorithm was a success.

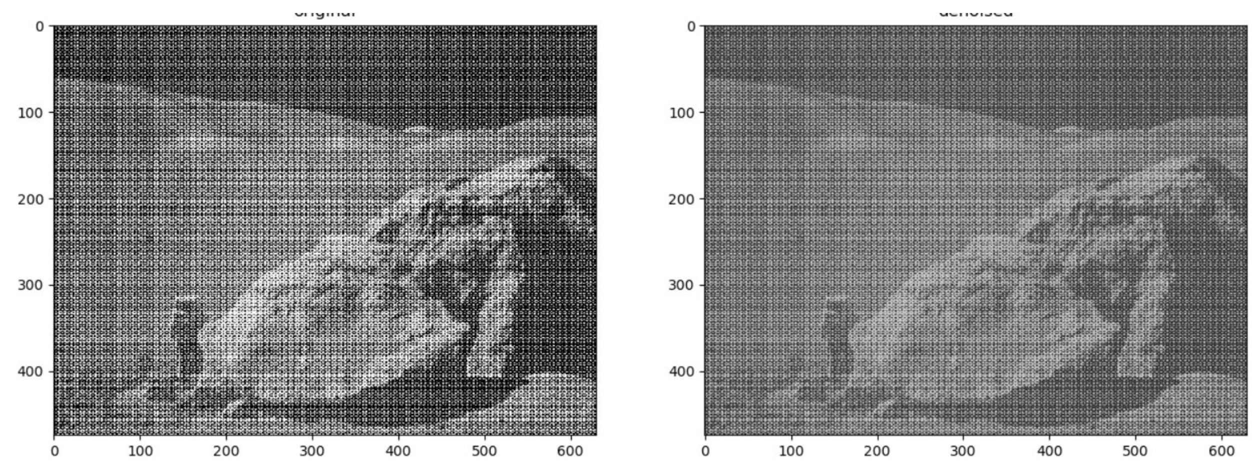
Mode 2:

As explained in the *Program Design* portion of this document, we deduced that the best way to denoise the image is to remove high frequencies in the FFT, as opposed to low frequencies, in the block around the middle of the fast fourier transformation output (defined in our program as *keep_ratio*). Zeroing out these values gave a properly denoised version of the image. Again, this idea was inspired from [2].

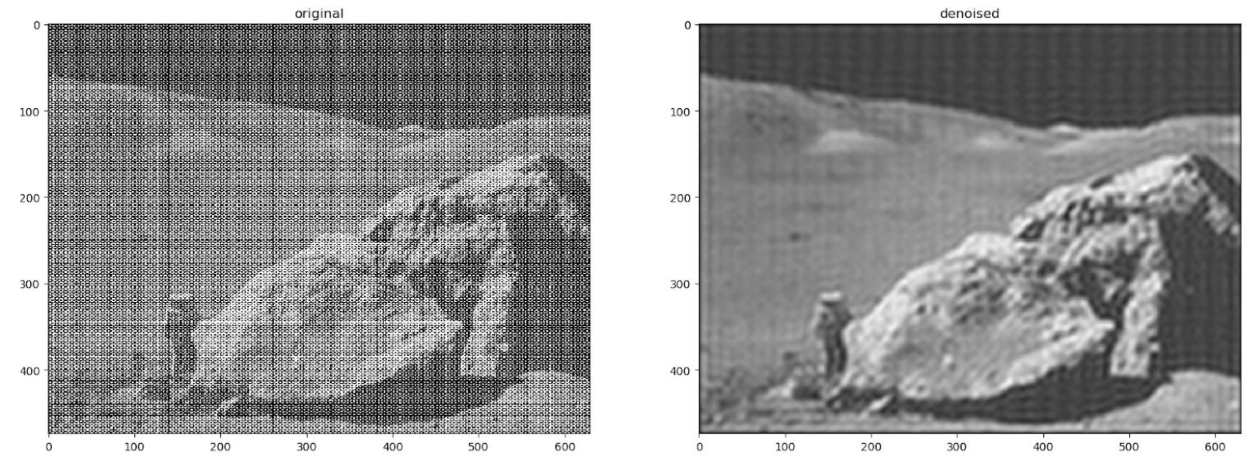
We tested this idea on various thresholds (*keep_ratio*) of high frequencies in the FFT to cut off for an optimally denoised image. Results are shown below for threshold values of 80%, 40%, 8%, and 4% of the pixels in the center of the FFT.



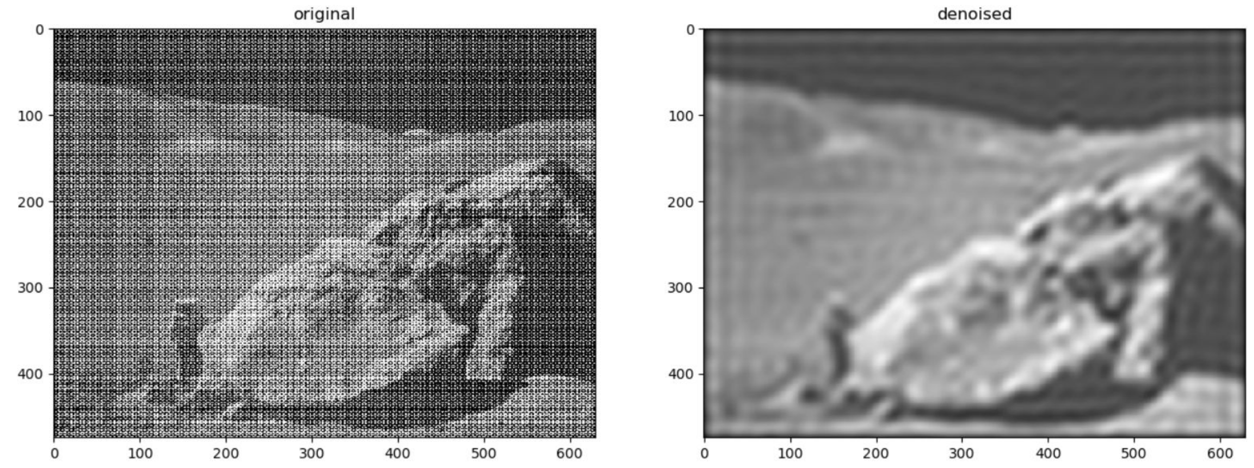
Fraction of pixels used 0.80 and the number is (409, 819) out of (512, 1024)



Fraction of pixels used 0.40 and the number is (204, 409) out of (512, 1024)



Fraction of pixels used 0.08 and the number is (40, 81) out of (512, 1024)

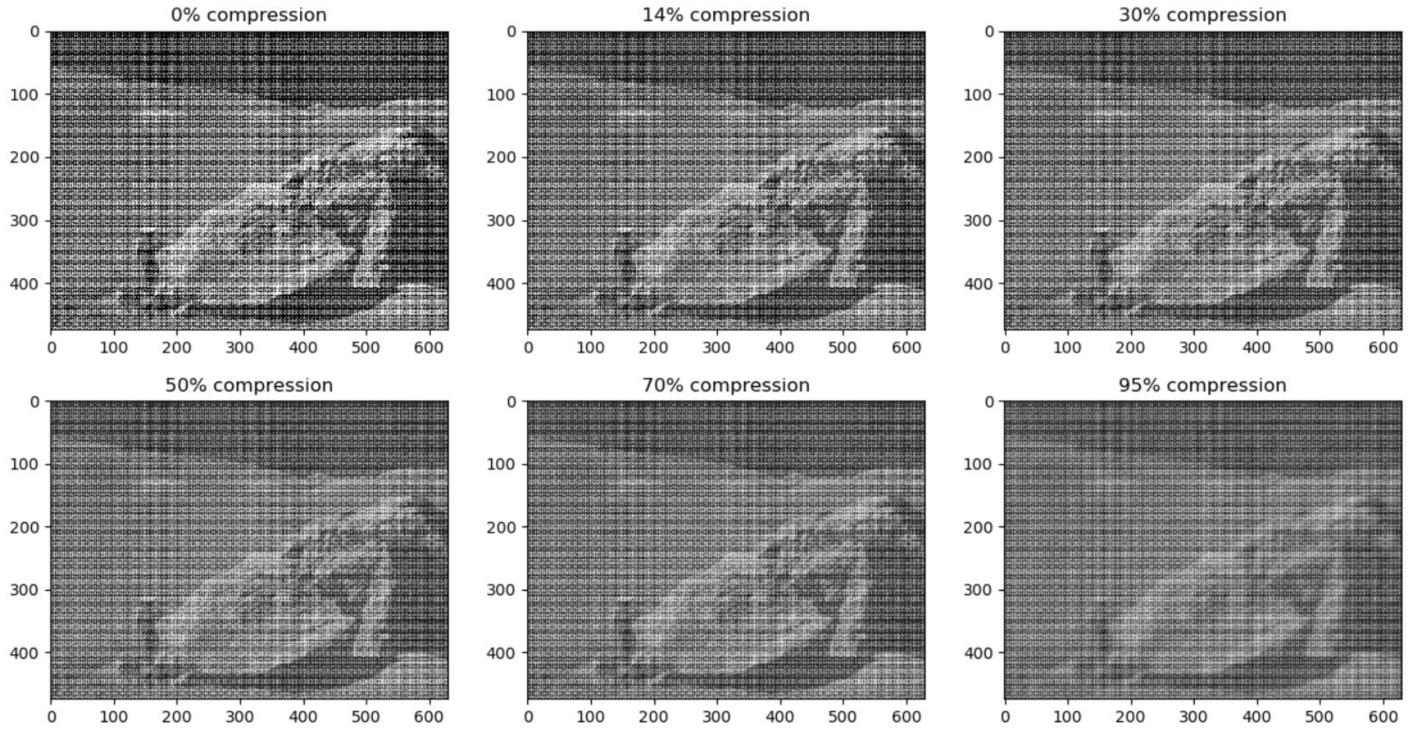


Fraction of pixels used 0.04 and the number is (20, 40) out of (512, 1024)

Clearly as we lower the fraction of pixels used from 80% to 8%, the image noise is substantially diminished. However, it seems that a ratio below 8%, such as the last image shown above at 4%, results in a blurry image as all sharp corners are softened. Therefore, a ratio of around 8% of pixels to keep in the FFT seems like an optimal balance of noise reduction and sharpness in the image.

Mode 3:

The result of the 6 different compression levels (0, 14, 30, 50, 70, 94) % is shown below:



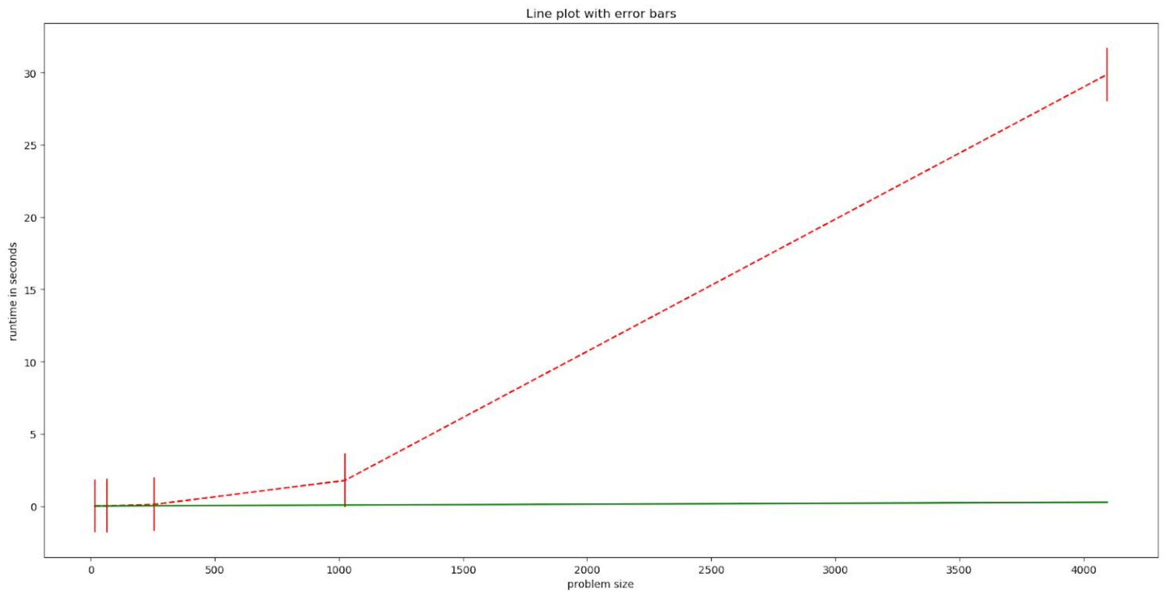
As the compression level is increased, we notice that the contrast between black and white is decreased as well, which results in poorer image qualities at each level. However, we are surprised that the image is mostly distinguishable even at 95% compression level, which has a sparse matrix file size of 4% the size of the original image’s matrix.

Statistics

Compression level	Non-zero values	Matrix file size
0 %	298620	7 859 KB (100%)
13 %	256813	6 879 KB (86%)
30 %	209034	5 683 KB (72%)
50 %	149310	4 117 KB (52%)
70%	89586	2 499 KB (32%)
95 %	14931	327 KB (4%)

Mode 4:

The result of the runtime of 2D DFT (red) and FFT (green) is shown below:



The error bars are shown as red vertical lines. The green line represents fast transformation and the runtime is always in milliseconds. Clearly, FFT is substantially faster than DFT as the size of input grows, which supports our runtime analysis in the previous section. Below are the statistics for each problem size over 10 runs which are used to plot the graph above.

Statistics:

Measurements for slow_two_dimension

for problem size of 16 over 10 runs: mean 0.0006018877029418945, stdev 0.0005183517497861728
for problem size of 64 over 10 runs: mean 0.008092164993286133, stdev 0.0009926399992119393
for problem size of 256 over 10 runs: mean 0.1111147403717041, stdev 0.003130113778472748
for problem size of 1024 over 10 runs: mean 1.7700897216796876, stdev 0.052426385355512015
for problem size of 4096 over 10 runs: mean 29.871770524978636, stdev 1.8563170107825608

Measurements for fast_two_dimension

for problem size of 16 over 10 runs: mean 0.0007979869842529297, stdev 0.0004205766958749998
for problem size of 64 over 10 runs: mean 0.002593207359313965, stdev 0.0009635048810641946
for problem size of 256 over 10 runs: mean 0.01646285057067871, stdev 0.0012582017894080951
for problem size of 1024 over 10 runs: mean 0.06454048156738282, stdev 0.0030856978769300517
for problem size of 4096 over 10 runs: mean 0.25954909324645997, stdev 0.009232139793241804

References:

1. Coates, M. ECSE 316 Lectures Winter 2020 retrieved from mcgill mycourses on 03/11/2020
2. Image denoising by FFT. (n.d.). Retrieved from http://scipy-lectures.org/intro/scipy/auto_examples/solutions/plot_fft_image_denoise.html on 03/11/2020