Abdelmadjid Kamli, 260984339
Mircea Gosman, 260983354
Group 27

April 3, 2023
ECSE 316 Assignment 2 Report

Design

The code is contained in three files:

- fft.py (main entry)
- fourier_operations.py (transforms, frequency selection & compression schemes)
- utils.py (argument parsing, image padding, plotting & lambda matrix operations)

The repartition of the code in these different files has been done with modularity & code re-use in mind. That is, we mainly aimed to avoid duplicating any code and to keep the code as flat as possible for readability. As a result of this effort, the longest method we wrote elapses 30 lines of code and serves the purpose of parsing the command line arguments in utils.py. Please additionally note that we do not make use of Classes in this project, as we found no use for Class/Object state and could, thus, make away with simple module imports.

fft.py

The modes executed by the program are listed in fft.py. There are the 4 required by the instructions (FFT, denoise, compress, plot) and each of them is about 4 lines of unsumarizable code. However, we did create 2 additional modes (-m 5 and -m 6) for accuracy measurements & FFT recursive splitting threshold finding. They will be detailed in the later appropriate sections.

fourier_operations.py

The transforms that are found in fourier_operations.py follow the algorithms outlined in the instructions. For the normal transform, we execute the X_k lambda function for each k in a numpy-vectorized manner over the input signal rows. Performing this operation along the rows, and then along the resulting columns yields the 2D transform.

As for the fast transform, the implementation of the divide & conquer algorithm is parametrized to be able to perform both forward and inverse operations. Let's take the example of the forward version. For each dimension, it is performing a sequential, row-by-row FFT. Each 1D FFT recursively splits itself into its odd & even parts whilst computing the joint coefficients. The speedup effectively comes from avoiding to compute the coefficients twice at each stage. We hereby also mention that our final implementation refers to the Python Numerical Methods book for its use of the concatenation method in the conquer part of the algorithm. Before referring to the book, we had tried a standard implementation which we tested to be accurate, but which did not speed up the process. In an original effort to make that first implementation faster, we attempted to vectorize some of the operations, again, without any success. We find that the reason behind these initial failures is that our non-concatenating approaches required carrying non-vectorized operations on the K variable, thus slowing down the otherwise rapid algorithm.

utils.py ()

The syntax to use our fft.py file is the same as outlined in the instructions. Kindly note that the program ignores superfluous invalid arguments (e.g. using a -q switch will be ignored). The utils.py file takes care of argument parsing and uses the matplolib pyplot's image loader. Please note that the operations we carry on the images are single channel only. That is not specifically an assumption we made with respect to the provided moonlanding.png image (which is grayscale). We had, in fact, initially built the naive, fast, and inverse transform operations to operate on an arbitrary number of independent channels. However, we ran into trouble for plotting the 3-channel results with compatibility between matplotlib and openCV's image loader and defaulted back to single channel as provided by matplotlib's loader. Finally (and on another note), we note resing non-power-of-2-sized images with openCV's resize method.

Testing

When we started out, we computed a 2x2 example on paper, and debugged our naive transform until its result matched ours. Then, we moved to a 2x8 toy example for quick tests, whilst keeping the large moonlanding image as a final 'system testing' step. The toy example is available for testing in the first lines of the fft.py/accuracy method.

For systematic reporting of our testing results, we created a 5th mode for accuracy measurements. In this mode, we take our naive transform, our fast transform, our inverse (of the fast) transform, and their numpy.fft equivalents. Then, we compute the root mean squared errors between the pairs and verify, using np.allclose() that the paris are within a specified tolerance (floating point errors fail to allow for exact equality checks). We found our toy example could withold a tolerance of 10e-15 (Logs 1) and the moonlanding image 10e-10 (Logs 2).

Logs 1 : Toy Example

```
PS C:\Users\mirce\OneDrive\Documents\Current\316-A2> python fft.py -m 5
Taking Normal Transform...
Taking FFT...
Taking Inverse...
Checking Numpy Transforms...
Root mean squared errors between our transforms & Numpy's:
        Naive transform is              (1.4574360675722492e-14+8.132524894000631e-15j)         | Within 10^-15 tolerance:      True
        Fast transform is               (1.4574360675722492e-14+8.132524894000631e-15j)         | Within 10^-15 tolerance:      True
        Fast inverse transform is       (3.2765070465752517e-15+1.1255352826311509e-15j)                | Within 10^-15 tolerance:      True
```

Logs 2 : Moonlanding Image

```
PS C:\Users\mirce\OneDrive\Documents\Current\316-A2> python fft.py -m 5
Taking Normal Transform...
Taking FFT...
Taking Inverse...
Checking Numpy Transforms...
Root mean squared errors between our transforms & Numpy's:
        Naive transform is              (2.791571969930938e-11+2.9935412947507184e-12j)         | Within 10^-12 tolerance:      True
        Fast transform is               (2.8100191470474153e-13+3.689155597912162e-14j)         | Within 10^-12 tolerance:      True
        Fast inverse transform is       (3.4223947606983103e-16-4.1526650572740146e-17j)                | Within 10^-12 tolerance:      True
```

Analysis

The naive approach will take O(N^2) of runtime. This is because the algorithm needs to go through N values to compute Xk. For each Xk, xn needs to be calculated which takes another N values.
For the FFT, we can see that each time we are dividing the problem by half and solving it. We have N values which implies that we will be reducing the problem by N/2. This division occurs twice once for even fields and the other for the odd ones.
Then putting N results together will take O(N).

This implies that we end up with : T(N) = N + 2T(N/2).
With the master theorem:

$\text{Log}_b a = \text{Log}_2 2 = 1$

Case 1 : f(n) = N then there is no e > 0 so that $N^{1-e} = N$

NOT case 1.

Case 2 : f(N) = N = N^($\text{Log}_2 2$) then it is clearly case two of the master theorem.

The run time is therefore O(N * log N)

For the 2D one, from the following code, we can observe that we calculate the fft N times for the rows and M times for the columns. This gives O(N*M * log N) + O(N*M * log M) = O(N*M * log N + N*M * log M).

Figure 0: code

```
for n in range(signal.shape[0]):
    d1[n,:] = fft(signal[n,:], j_coef)

for m in range(signal.shape[1]):
    d2[:, m] = fft(d1[:,m], j_coef)
```

Experiment

Fast Fourier Transform (mode 1)

Before starting out the experiments, we sought to find an optimal threshold for the minimal size upon which to recursively split the input (using the -m 6 syntax). We noticed that changing the threshold impacted the accuracy, but we note that this impact is very small in magnitude and thus did not let accuracy impact the selection of the threshold. Instead, we picked a threshold of 32 because it improved the runtime of the algorithm by balancing compute & splitting overhead delays.

Figure 1: RMS & Threshold

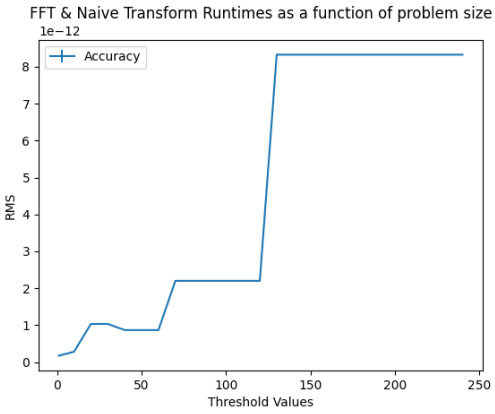
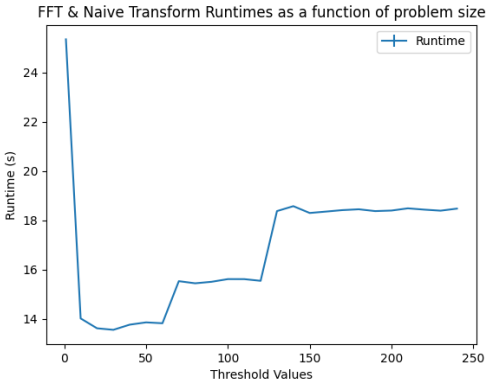


Figure 2: Runtime & Threshold



Following is a graphical (logarithm color-mapped) representation of our transform (Figure 3) and of numpy's (Figure 4). As mentionned before in the accuracy topic discussion, the numbers themselves are close to 10e-12. Thus, the graphical transforms are not very distingushable from one another.

Figure 3: Our Shifted FFT

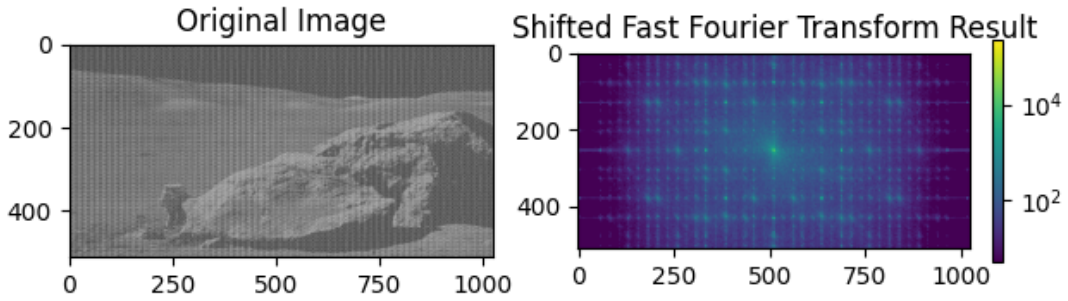
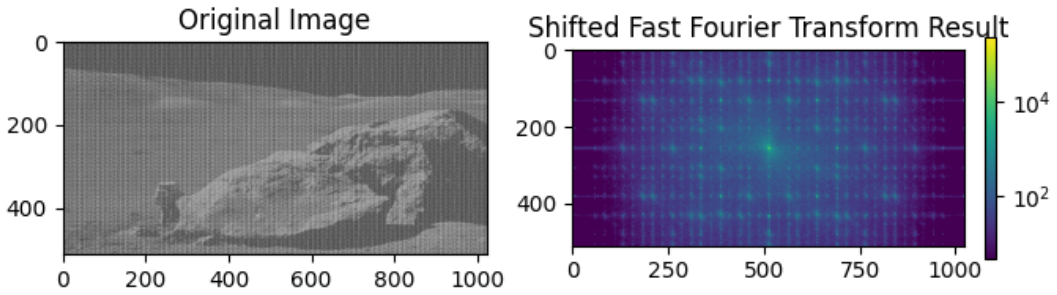


Figure 4: Numpy's Auto-Shifted FFT



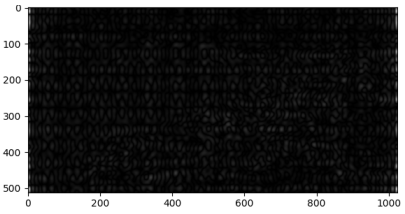
Denoising an Image (mode 2)

We tried both the high frequency and low frequency filtering schemes. For each scheme, we parametrized the thresholds to be axis dependent (i.e. a threshold for first past of 1D transforms on rows, another for the 2nd pass on columns). However, this did parametrized approach did not help the results and is omitted from this section's results. We present here only a few low frequency and high frequency results. As can be observed, the noise signal is of high frequency nature and filtering low-frequencies simply removes the image data (Figure 5). We qualitatively note our best filtering results (in green) are obtained from filtering high frequency with a threshold within the range of [0.05, 0.1] * PI (Figure 6).

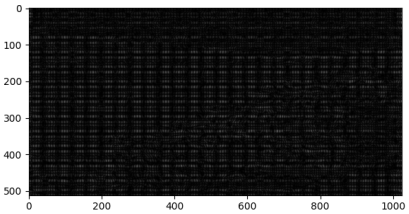
In the end, the optimal value is subjective as we do not have a reference image to optimize for and have to guess a balance between keeping leftover noise & blurring out the image too much. We also mention that we chose to seek out fractions of PI for the threshold values because the cyclic nature of the [0, 2PI] range of the transform means that we only ever filter 1 PI interval twice (once from each end to account for the shift).

Figure 5: Low Freqeuncy Filtering

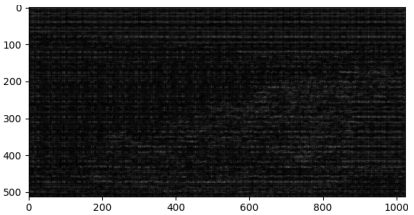
0.9 PI | (1% coeff. left)*



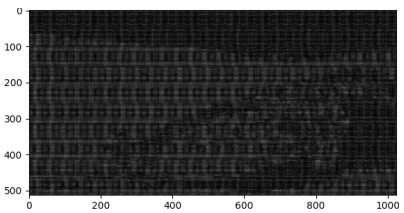
0.8 PI | (4% coeff. left)*



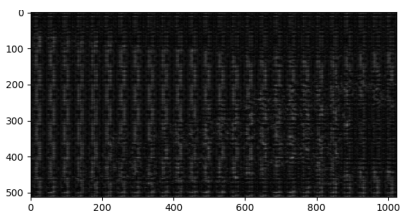
0.7 PI | (9% coeff. left)*



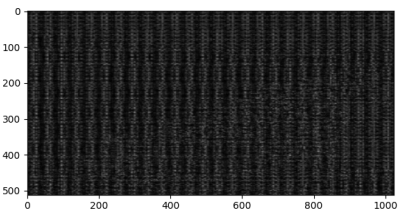
0.6 PI | (16% coeff. left)*



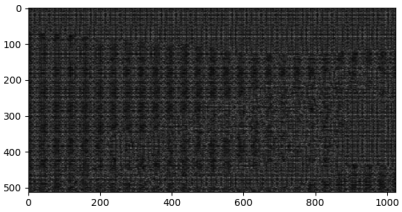
0.5 PI | (25% coeff. left)*



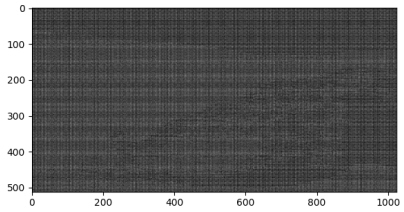
0.4 PI | (36% coeff. left)*



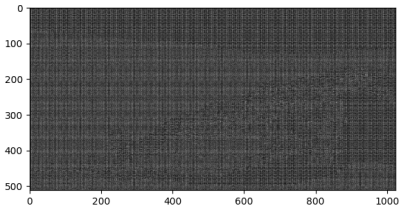
0.3 PI | (49% coeff. left)*



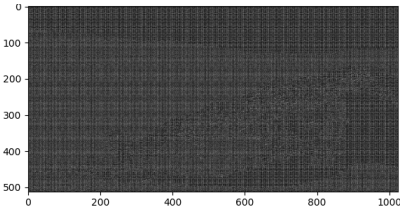
0.2 PI | (64% coeff. left)*



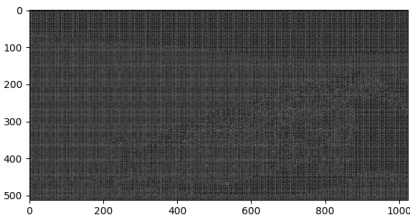
0.1 PI | (81% coeff. left)*



0.07 PI | (87% coeff. left)*



0.05 PI | (90% coeff. left)*



0.01 PI | (98% coeff. left)*

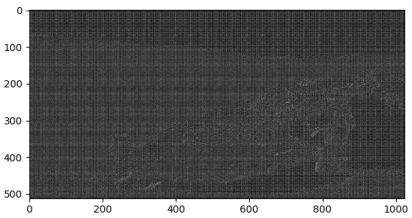
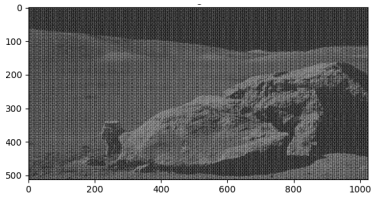
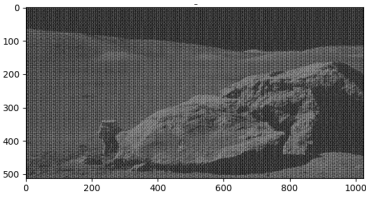


Figure 6: High Frequency Filtering

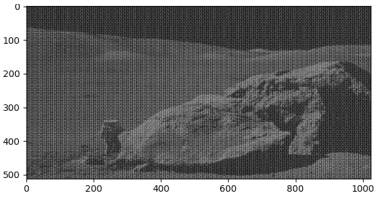
0.9 PI | (81% coeff. left)*



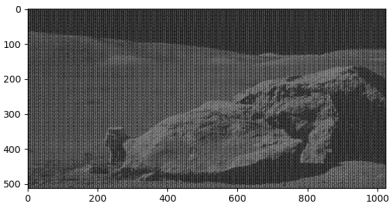
0.8 PI | (64% coeff. left)*



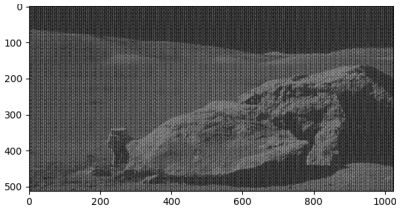
0.7 PI | (49% coeff. left)*



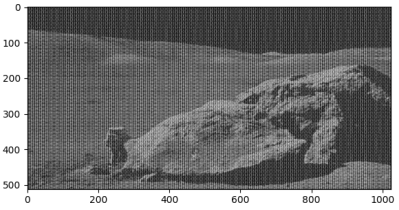
0.6 PI | (36% coeff. left)*



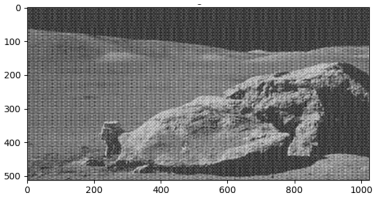
0.5 PI | (25% coeff. left)*



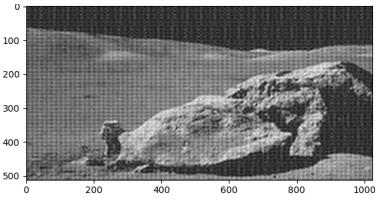
0.4 PI | (16% coeff. left)*



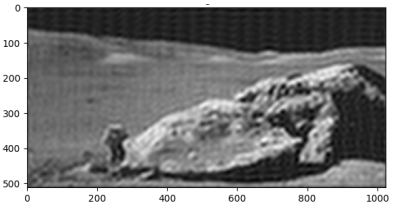
0.3 PI | (9% coeff. left)*



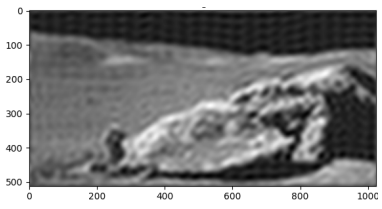
0.2 PI | (4% coeff. left)*



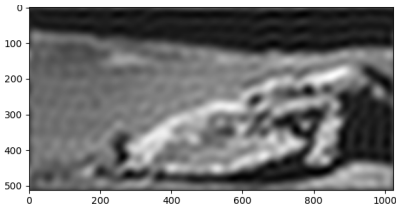
0.1 PI | (1% coeff. left)*



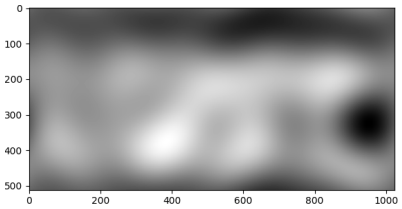
0.07 PI | (0.4% coeff. left)*



0.05 PI | (0.02% coeff. left)*



0.01 PI | (0.01% coeff. left)*

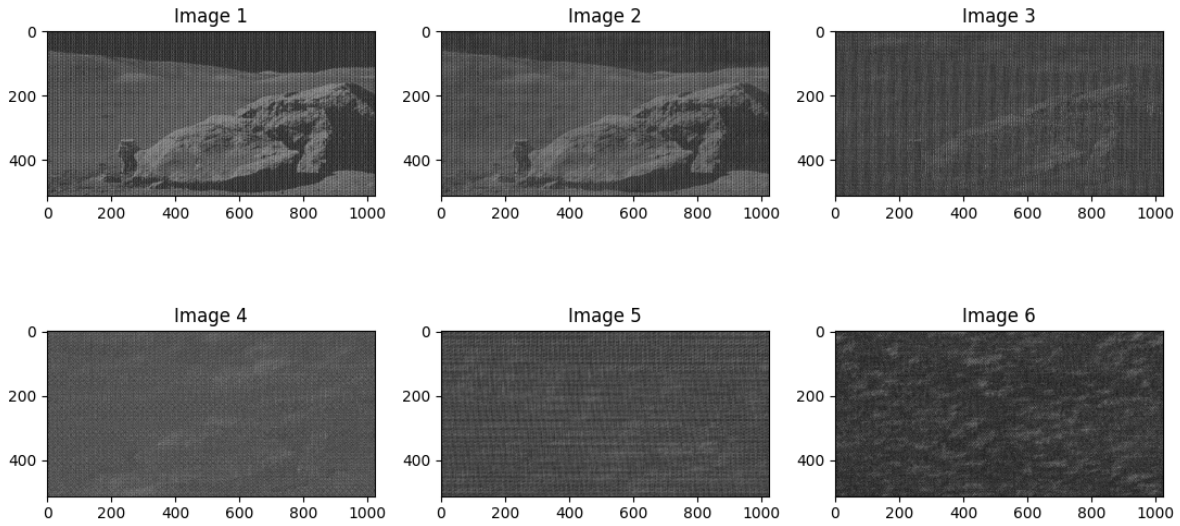


Compression (mode 3)

For compression, we tried 4 different schemes: random, threshold, high frequency, and low frequency. In all cases, the input compression factors were [0%, 19%, 38%, 57%, 76%, 95%] and we have generated an image with each of them. The real, final, obtained compression factor depends, in each case, on the robustness of the compression scheme.

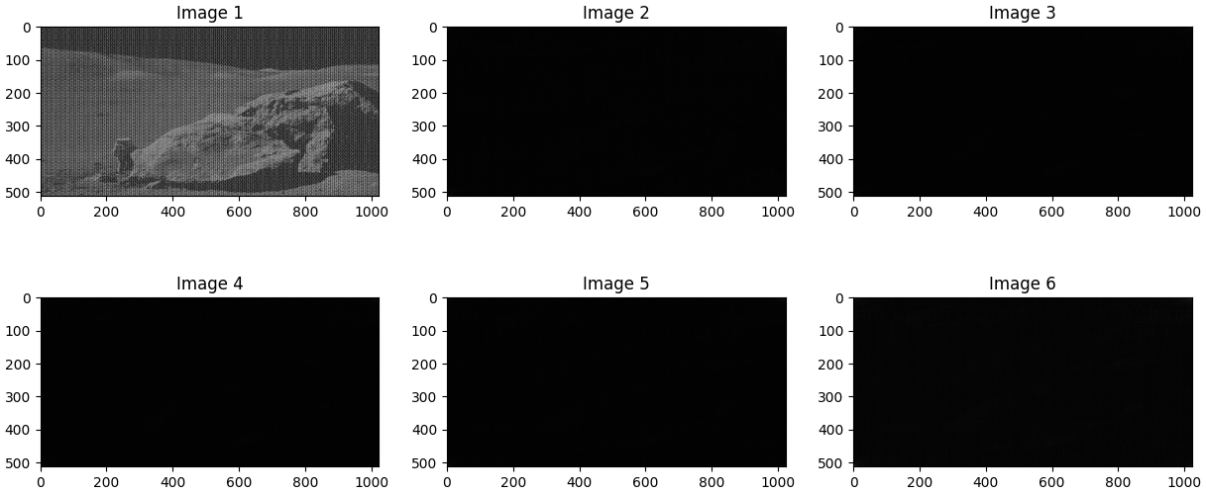
To obtain a baseline for failure, we started out with a random compression scheme which, selects *image.size * factor* coefficients to delete. Unsurprisingly, the visual results (Figure 7) of the image reconstructions are poor as we are likely to delete key image features when sampling coefficients randomly.

Figure 7: Random Compression Scheme | Left Coefs: [100%, 91%, 62%, 43%, 24%, 5%]



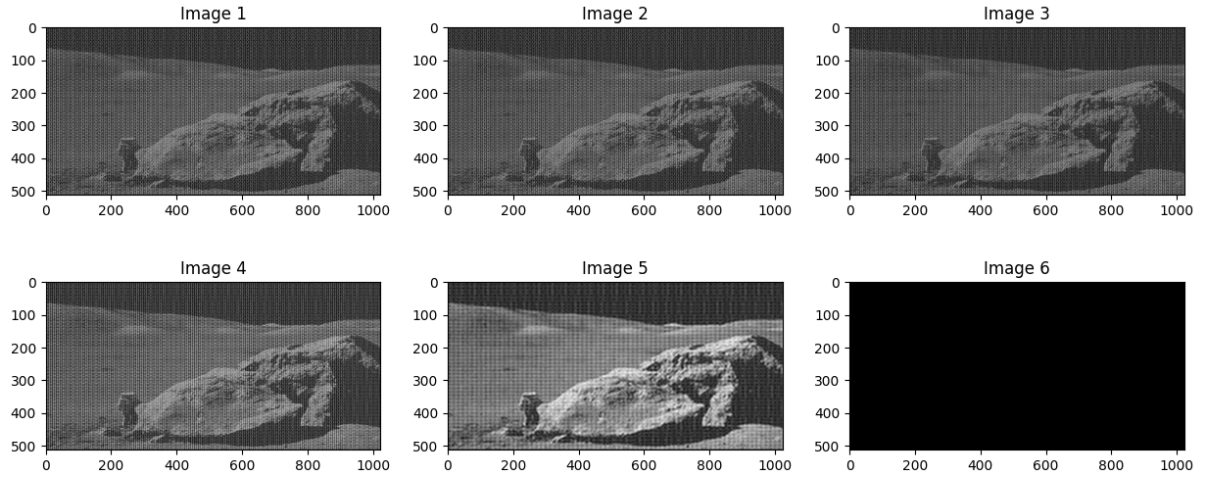
As a first simple scheme, we attempted to filter the *image.size * compression factor* highest magnitude coefficients. Unfortunately, for some obscur reason, we are unable to report complete visual results (Figure 8). When walking through the code with a debugger, we see the correct selection and deletion of coefficients occurring. What's more is that the correct number of non-zero coefficients is output to the CLI afterwards. Strangely again, when computed over our small toy data, the threshold compression scheme selects, deletes, and displays as expected. The leftover issue seems to be a non-throwing error between the compressed moonlanding data and matplotlib.

Figure 8: Threshold Compression Scheme | Left Coefs: [100%, 81%, 62%, 43%, 24%, 5%]



In a previous section, we detailed that high frequency removal helps keeping expected image data. Since the filtering process essentially turns some coefficients to 0, it made sense to try apply frequency selection logic when compressing as a next step. To keep our implementation simple, we assumed an even distribution of frequencies between 0 and 2 PI. Instead of hand filtering the coefficients with the *image.size * compression factor* highest frequencies, we let our filtering threshold be *compression factor * PI*. However, we note that this assumption reduces the robustness of the compression scheme, as the real distribution may not be uniform. As a result, the output ends up with a (often only slightly) different compression scheme than specified in the input. Figure 9 shows the reconstructed compression images, with image 5 corresponding to 97% final compression whilst still maintaining reasonable accuracy. We attribute the success of the later method to the fact that image 5 has 3% leftover coefficients, which is close to the best range of [0.05 - 1] outlined in the filtering section.

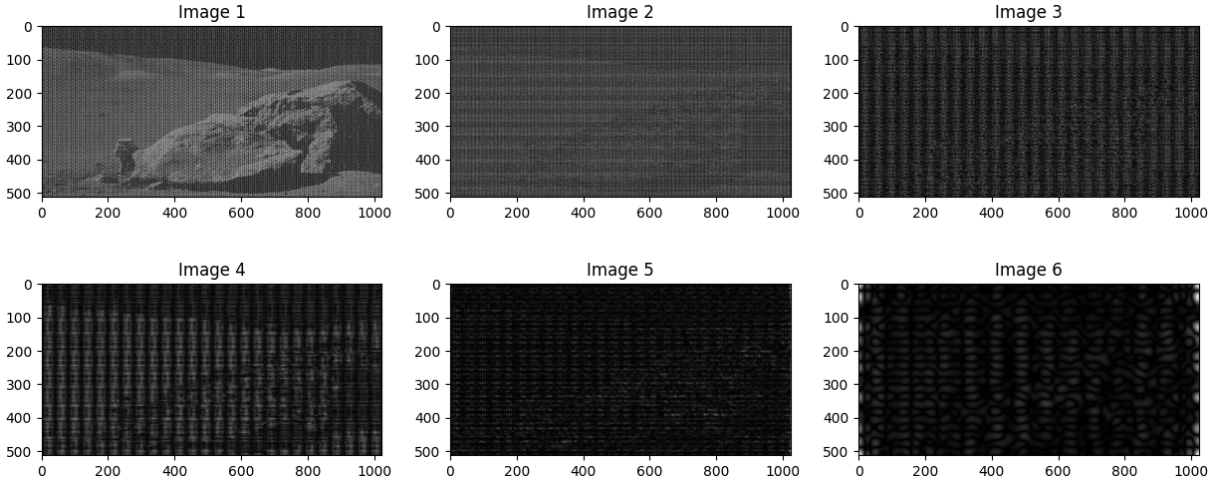
Figure 9: High Freq. Compression Scheme | Left Coefs: [90%, 58%, 32%, 14%, 3%, 2e-4%]



For completeness, we also attempted a low frequency removal compression scheme. However, as was detailed in the filtering section, low-frequency filtering does not increase image quality. Thus, the reconstructed images (Figure 10) turn out to be more of the noise than of the expected image. (Here,

we also not the lack of robustness of this frequency filtering scheme as per the uniform distribution assumption mentionned earlier.)

Figure 10: Low Freq. Compression Scheme | Left Coefs: [100%, 65%, 38%, 19%, 6%, 0.2%]



Finally, we compiled the kilobytes that each compression scheme/factor pair saved *on the transform itself* in Table 1. We generally observe that the compression is very minimal compared to the size of the csv data file (27 908Kb), regardless of the compression scheme or factor. However, we found that the size of a data file that can store a numpy array of zeros is 27 649Kb. Thus, we conclude that the compression scheme can, at most, save 27 908Kb - 27 649Kb = 259 Kb of space. As it turns out, due to rounding errors, the high frequency compression scheme saves the entirety of that space with only a 95% compression factor.

As a result of this realisation and of prior detailed observations, one may conclude that noise filtering may be used to compress a signal without compromising its integrity.

Table 1. Space Saved per Scheme per Factor (27 908 Kb -

	19%	**38%**	**57%**	**76%**	**95%**
Random	24	49	74	98	123
Threshold	0	0	39	136	233
High Freq.	110	176	222	250	259
Low Freq.	89	158	209	243	258

Runtime Plots (mode 4)

We plotted the runtimes of both naive and fast transforms over 10 iterations of the following array of 2D square image sizes: [2^5, 2^6,2^7,2^8,2^9, 2^{10}]. We critically observe that there is no gain to using the FFT algorithm when operating on images under around 252x252 pixels. In such cases, the overhead generated by the recursive splitting is diminishing the merit of any speedup the algorithm may generate in coefficient computing savings. At larger image sizes, of course, the Cooley-Tuckey algorithm saves minutes in runtime when compared to the naive version (Figure 11).

At last, we make a final note about error bars: we have set the errors bars to be the same as specified in the instructions (2*standard deviation). However, the standard deviations are several magnitudes lower than the means. This is due to the mostly deterministic conditions of the system the trials (of the deterministic algorithm) ran on. Figure 12 shows a zoomed-in version Figure 11. It showcases an error bar at size 512 otherwise invisible at the regular zoom level.

Figure 11: Runtime & Problem Size

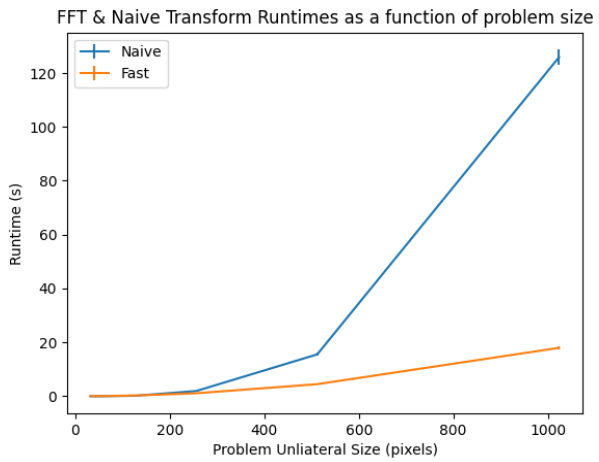


Figure 12: Zoomed in View: Error Bars Exist!