ECSE 316 Assignment 2 Group25

Yuhang Zhang 260787441

Charles Liu 260765100

Design

- ### The Discrete Fourier Transform

  First and foremost, we implemented the discrete Fourier and inverse discrete transform. Our idea for this specific section is to convert the mathematical formula we learnt during the class into python functions. It did take a while for us to come up with a way to realize the summation in the formula. We, inspired by the linear algebra dot product, calculated and stored each individual part (without xn) into any array within the range of array size ("N" in code), and dot product the it with xn array. This resulted in each part being multiplied and summed.
  The inverse discrete Fourier transform was exactly the same idea. The only 2 differences are multiplication of 1/N in each term and removing the negative sign in the power term.

- ### The Cooley-Tukey FFT

  To start with this Cooley-Tukey method, we firstly read through the text provided to us. The algorithm is a classic "divide and conquer" idea. Therefore, we write the algorithm by imitating "merge sort" algorithm. First we check if the size of the input array is a power of 2. If not, we return an error. Next we define our base case. By doing some experiments we set the base case to be 32. That is, if the array size is less or equal to 32, we use DFT to calculate directly. The recursive part is quite similar as the "merge sort". We split the array into even and odd, which can be easily done using the slicing operation in python. The merging operation is done using an internal function called "concatenate" in numpy.
  The inverse discrete Fourier transform was exactly the same idea as before. The only 2 differences are multiplication of 1/N in each term and removing the negative sign in the power term.

- The Two-Dimensional Fourier Transform

The Two-Dimensional Fourier Transform is a little tricky than One-Dimensional. We studied the formula for 2D FFT and realized that 2D FFT can just be done by using nested function. That is, functions like this form.

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

Figure 1

Figure 1 is the formula for 2D FFT. We can calculate the inner summation, which is 1D Fourier Transform with input being a 2D array. 2D array can be thought as an 1D array with N^2 elements. Therefore, the inner summation can be easily achieved by using 1D FFT, with a little modification for the input from 1D array to 2D array. We then calculate the outer summation, which can be done using a nested function. The FFT method outputs a 2D array, which becomes an input of another FFT method. This is how we achieve 2D Fourier Transform. The idea of inverse 2D Fourier Transform is the same except that we now need to divide N^2 instead of N.

Testing

- Invalid Input Test

This is the test we test first. We have developed a complete error detection system. There are two inputs, both of which should be checked for validity. The program will print an appropriate exception and terminate when there is an error, such as wrong image location, invalid mode number, etc.

- Algorithm Test

  To test each of our Fourier transform functions is working properly, we came up with an "efficient" idea: after each time we implemented a function, we imported the corresponding library from numpy. For example, we test our FFT algorithm with the build-in function numpy.fft.fft(). In addition, we tested if those two functions would give rise to the same result when we fed them the same array input. We also use the build-in function for testing.

- Mode Test

  To test if each mode is functional, we trigger each mode one at a time to see if each mode is outputting as desired. To make sure our functions are capable of handling generality, we tested 3 other png images and found they would also work.

Analysis

The runtime of the naïve DFT for 1D is O[N2] because for every element in the 1D array we must do the summation. Each summation has N components, N being the length of the array. There are N elements in the array. In total, we have to do N*N computations. Thus the complexity is O[N2].
The derivation for the FFT is shown below.

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

Assume N is a power of 2.

By looking at the formula : $X_k = X_{even} + e^{-\frac{i2\pi k}{N}} X_{odd}$.

we can write the recurrence relation like this:

$$t(N) = cN + 2t\left(\frac{N}{2}\right), \text{ where } c \text{ is a constant.}$$

c is the time for computing $e^{-\frac{i2\pi k}{N}}$, where k lies $[0, N-1]$.

There are N computations, thus cN.

$$\begin{aligned} t(N) &= cN + 2t\left(\frac{N}{2}\right) \\ &= cN + 2\left(c\cdot\frac{N}{2} + 2t\left(\frac{N}{4}\right)\right) \\ &= cN + cN + 4t\left(\frac{N}{4}\right) \\ &= cN + cN + 4\left(c\cdot\frac{N}{4} + 2t\left(\frac{N}{8}\right)\right) \\ &= cN + cN + cN + 8t\left(\frac{N}{8}\right) \\ &\vdots \\ &= cNk + 2^k t\left(\frac{N}{2^k}\right). \\ &= cN\log_2 N + N t(1) \quad \text{when } N = 2^k. \end{aligned}$$

$O(N\log N)$ since this is the dominant term.

The complexities for both 2D DFT and 2D FFT algorithms are also easy to compute. For the 2D FFT algorithm, we can think of it as an 1D array containing N^2 elements. As mentioned earlier, the complexity of 1D FFT is O[N*log N]. Therefore, all we need to do is to replace N, as the number of elements in 1D array, to N^2, as the number of elements in 2D array. Thus, the complexity becomes $O[N^2*\log(N^2)]$.
We simplify a little by moving the exponent inside the log function outside and it becomes $O[N^2 2\log(N)]$ which shows the time complexity of the 2D FFT is $O[N^2\log N]$.
For the 2D DFT algorithm, we apply the same technique as above. Recall the complexity of 1D DFT is $O[N^2]$. We replace N to N^2, as it is now an array with N^2 elements. Therefore, the time complexity of the 2D DFT is $O[N^4]$.

Experiment

- **FFT implementation**

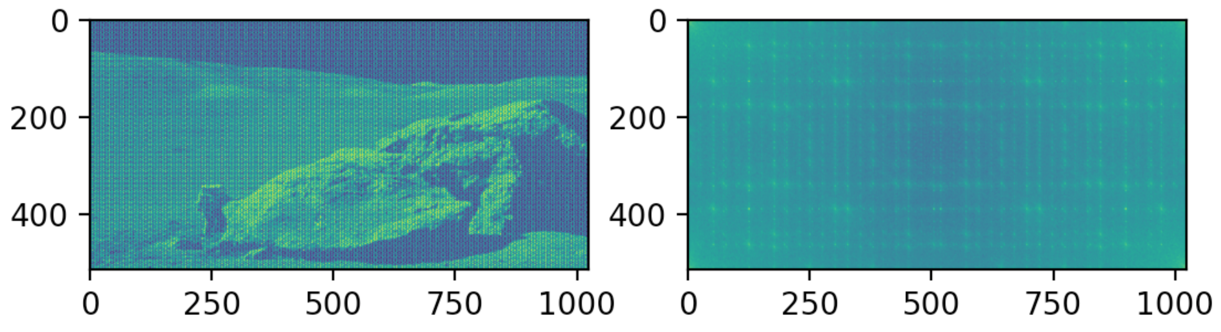


Fig.1 2DFFT implementation result

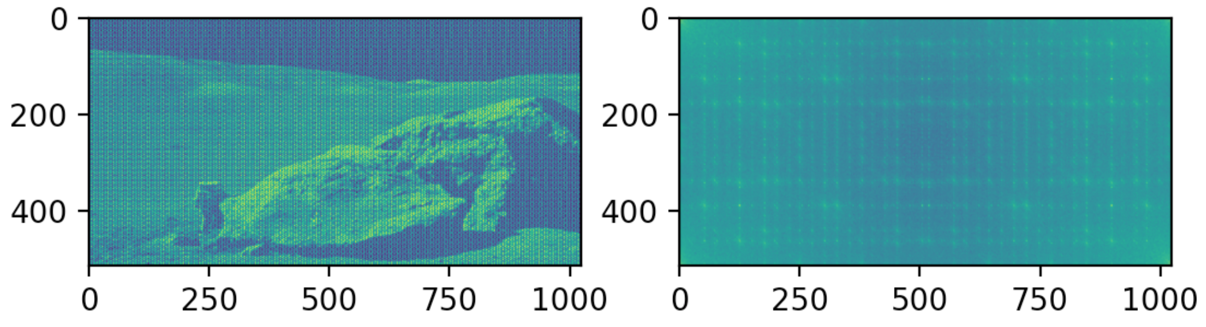


Fig.2 2DFFT result by np.fft.fft2

- **Image Denoise**

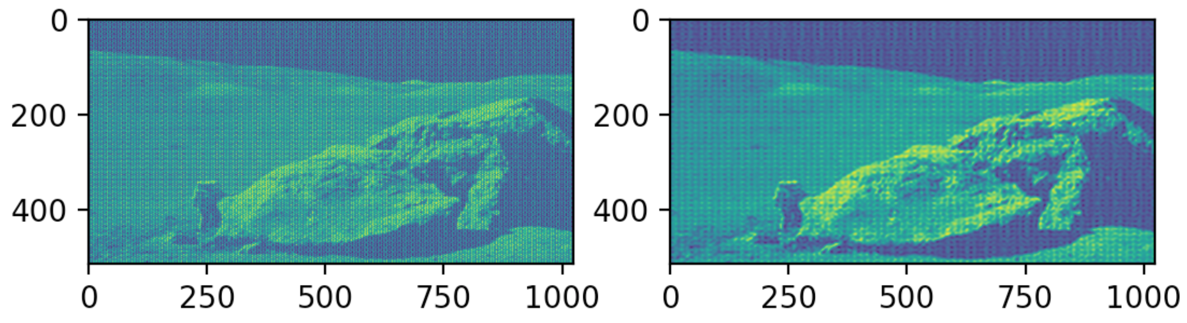


Fig.3 10%high+10%low frequency removed

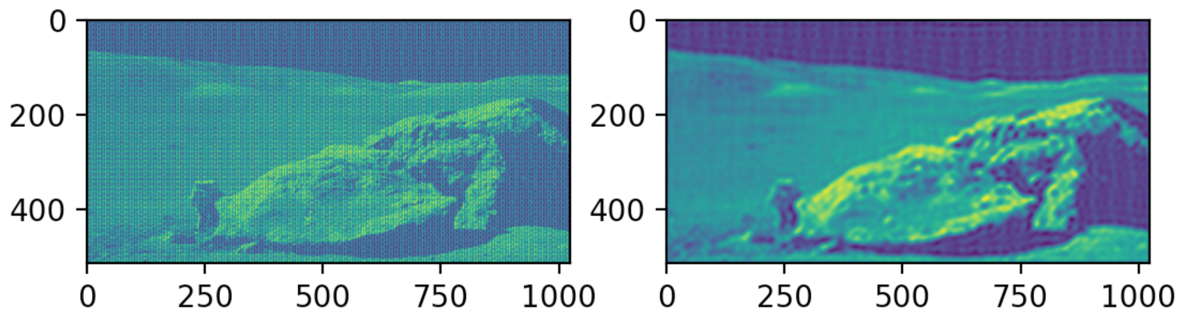


Fig.4 5%high+5%low frequency removed

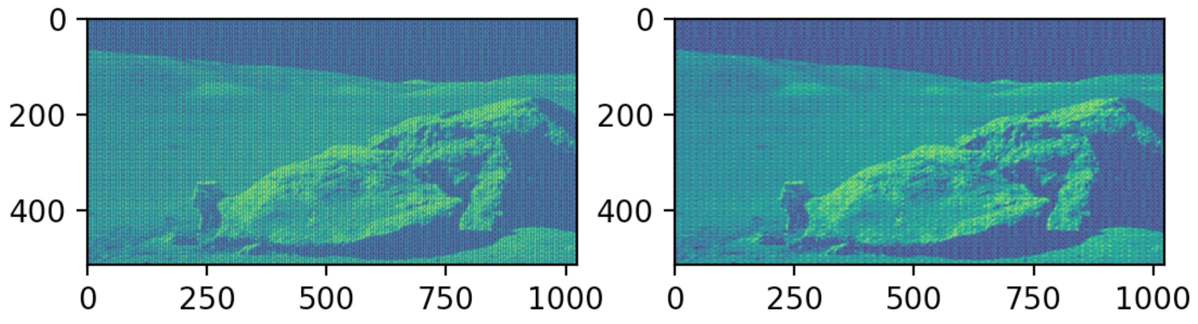


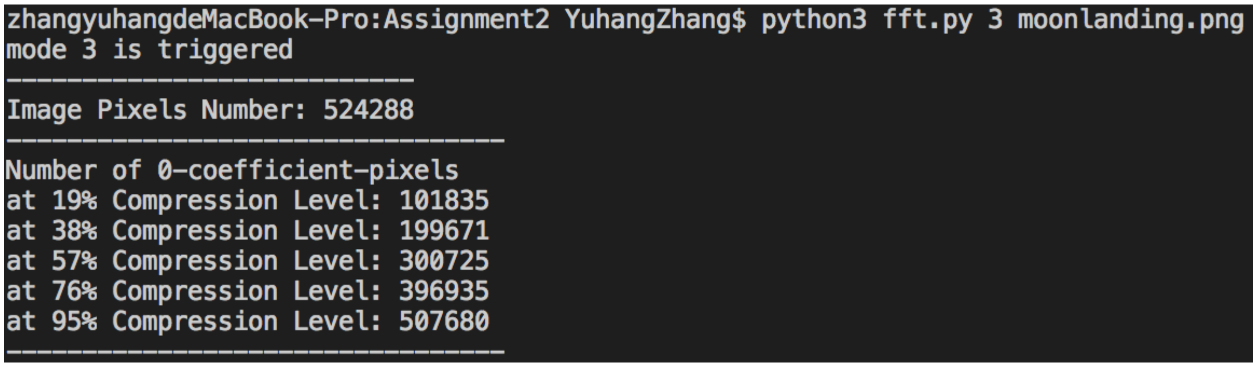
Fig.5 15%high+15%low frequency removed

- **Compression**

  In this part, our strategy is to set a series of threshold boundary for different compression ratios, and for any pixel whose pixel intensity is blow that its corresponding threshold, the pixel is converted to 0.
  We firstly got the total number of pixels by multiplying the length and width size of the image. Furthermore, we calculated the number of pixels shall be converted to 0 at each compression level, 19%, 38%, 57%, 76%, and 95% respectively. The number of pixels shall be turned to 0 are approximately 10,000 for 19%; 20,000 for 38%; 30,000 for 57%; 40,000 for 76%; 50,000 for 95%.

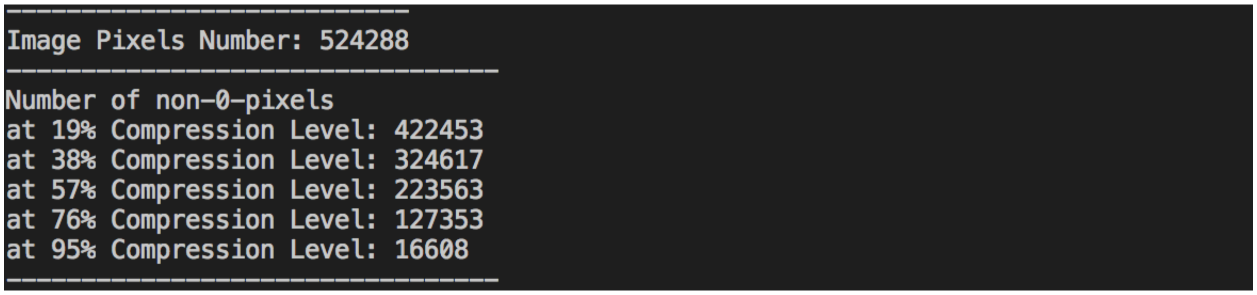
  Experiment_1:
  Apart from using the above strategy, we tried the other way around: to keep the smaller percentile of the pixels, but this result did not turn as we expected.

  Experiment_2:
  In addition, we also tried to keep the coefficient of those abandoned pixels to be 100 instead of 0, hoping the compressed image would have a moderate looking. It also failed our expectation.



```
zhangyuhangdeMacBook-Pro:Assignment2 YuhangZhang$ python3 fft.py 3 moonlanding.png
mode 3 is triggered
---------------------------
Image Pixels Number: 524288
---------------------------------
Number of 0-coefficient-pixels
at 19% Compression Level: 101835
at 38% Compression Level: 199671
at 57% Compression Level: 300725
at 76% Compression Level: 396935
at 95% Compression Level: 507680
---------------------------------
```

Fig.6 Number of pixels whose coefficient is 0 at different compression levels.



```
---------------------------
Image Pixels Number: 524288
---------------------------------
Number of non-0-pixels
at 19% Compression Level: 422453
at 38% Compression Level: 324617
at 57% Compression Level: 223563
at 76% Compression Level: 127353
at 95% Compression Level: 16608
---------------------------------
```

Fig.7 Number of pixels whose coefficient is NOT 0 at different compression levels.

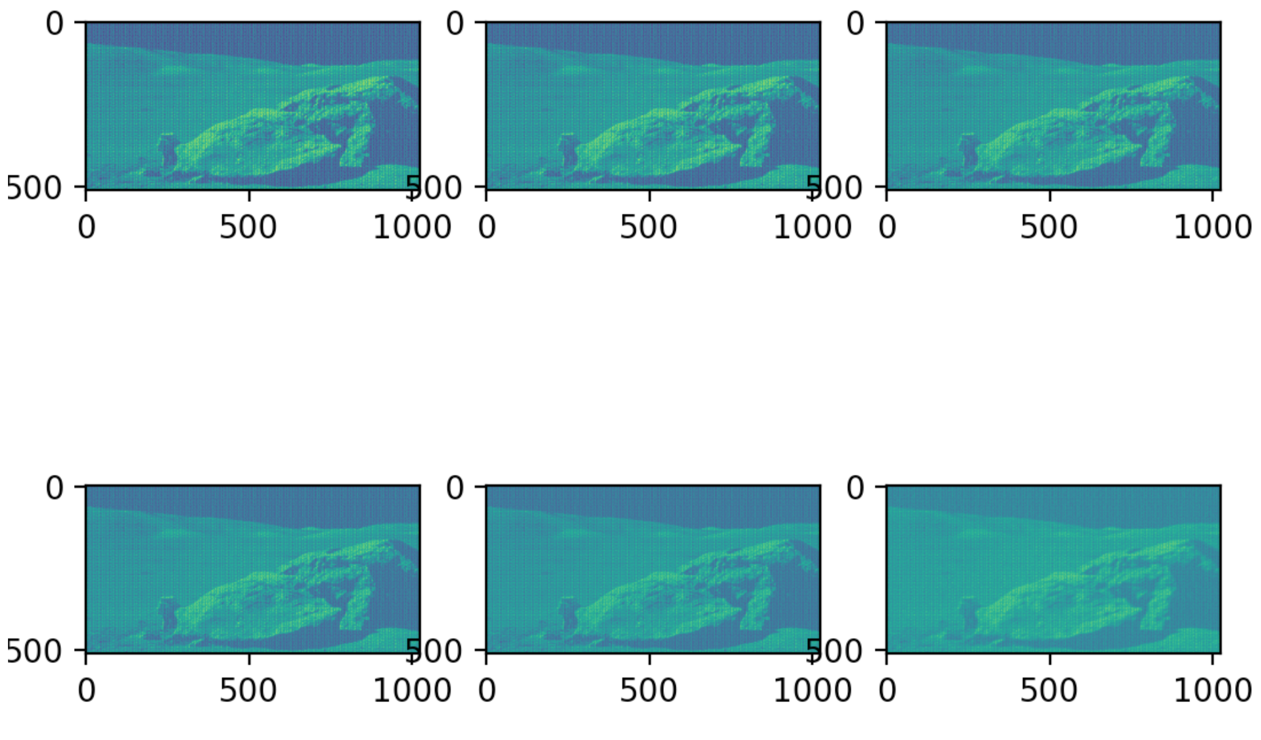


Fig.8 Compressed Images at different levels

- **Runtime plotting**

  In this part, we calculated the runtime of DFT and FFT with different sizes of 2D arrays. We tested 5 different sizes: 32, 64, 128, 256, 512. Each of them had run 15 times. We then gathered data and calculated the mean and standard deviation of both DFT and FFT. From the graph, one can clearly see that FFT is much better than DFT in the long run, at both mean and standard deviation. Figure 8 is the graph.

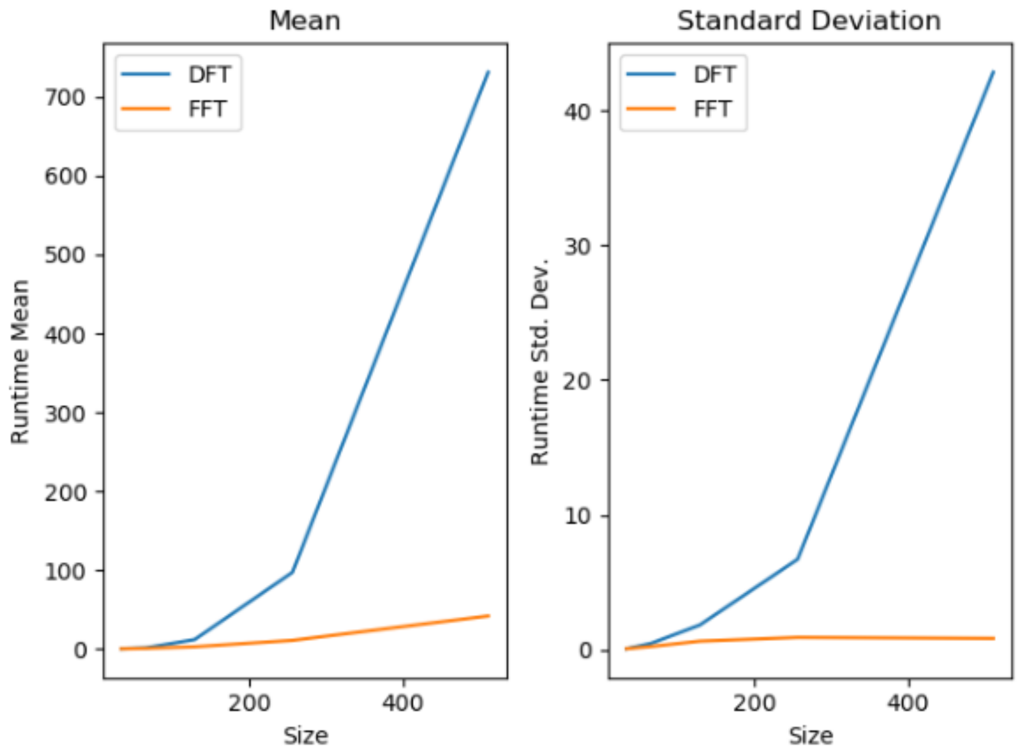


Figure.9 Runtime graph of DFT vs FFT