Yannick Morin - 260855737
Charles-Antoine Paré - 260930624

ECSE 316 : Assignment 2 - Fast Fourier Transform and Applications

Design:

Discrete Fourier Transform:

For the naive implementation of the dft, we took the equations provided in the assignment description and used an embedded loop to iterate over the k and n variables. As for the fast implementation, again following what was provided in the assignment description, we used a recursive approach similar to merge sort to compute the DFT of even and odd numbers separately using our naive implementation and then merged the results back together.

For the two-dimension FT we used four embedded loops to iterate over the four changing variables in the equation provided in the assignment description. For the fast two-dimensional implementation, we used the fast fourier transform from earlier twice, once per row and once per column.

For the denoising algorithm, we removed the higher frequencies in the two-dimensional FT and then used the inverse of the two-dimensional FT to get back the denoised image.

For image compression, we used a strategy that keeps only the very low and very high frequencies of the FT. To do so, we first computed the FT of the image and then filtered out all unwanted frequencies, then we used the inverse FT to get the compressed image.

Testing:

To make sure that all our DTF algorithms worked properly, we used the one-dimension and the two-dimension fast fourier transform implementation from the numpy library and verified that our outputs matched theirs for randomly generated 1-D and 2-D arrays.

Analysis:

Naive 1d DFT runtime:

By looking at our implementation that uses an embedded loop, we can quickly see that the complexity of our naive DFT implementation is O(n^2).

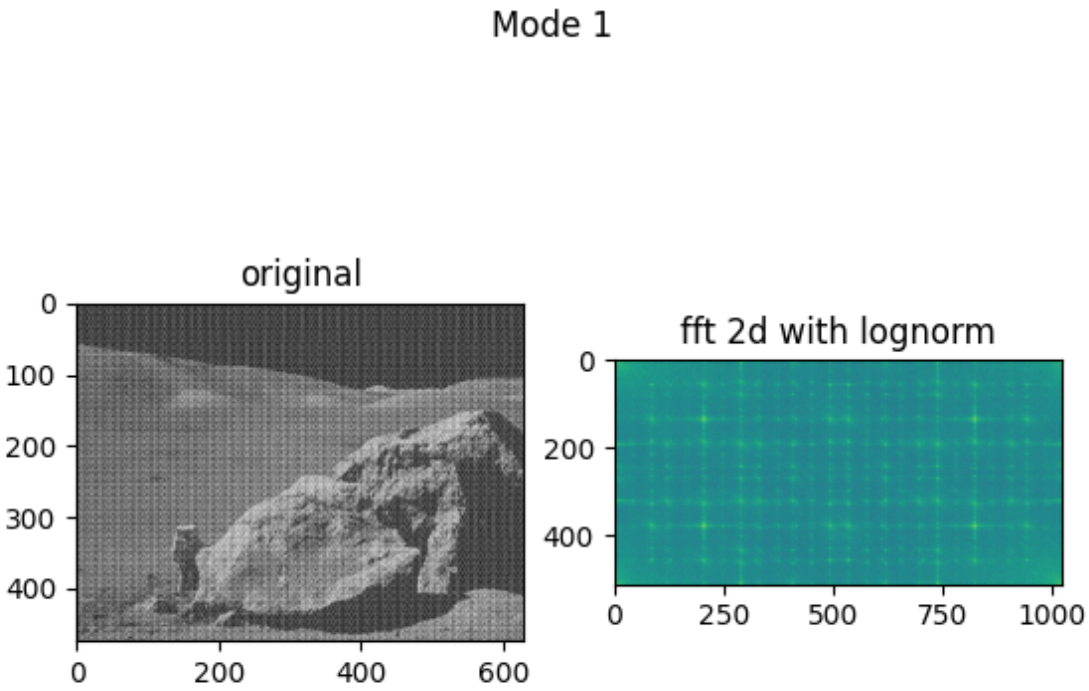
1d FFT runtime:

Let's keep in mind that our FFT algorithm is very close to the merge sort algorithm and since we divide our input into odds and even numbers, we will do at most log(n) divisions (n being the input size) this yields a complexity of O(n log(n)). This is plausible since it is the same time complexity as the merge sort algorithm.

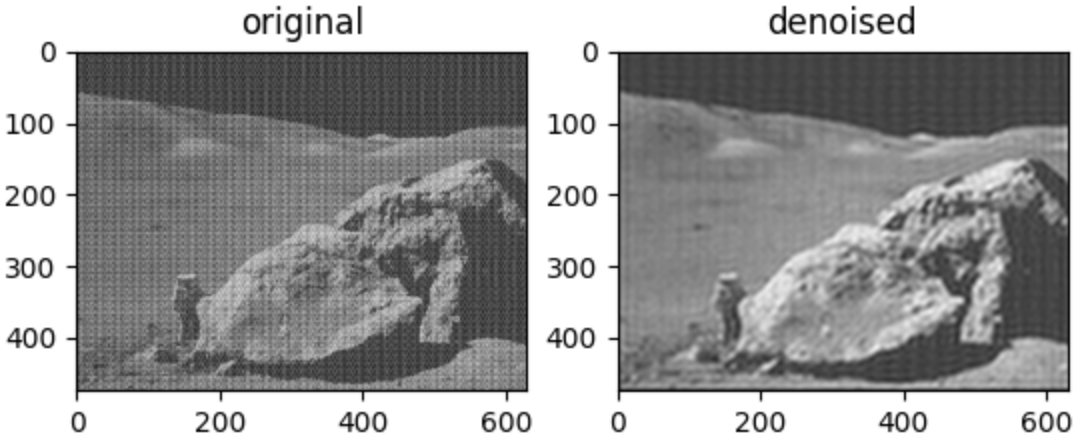
Naive 2d DFT and FFT runtime:

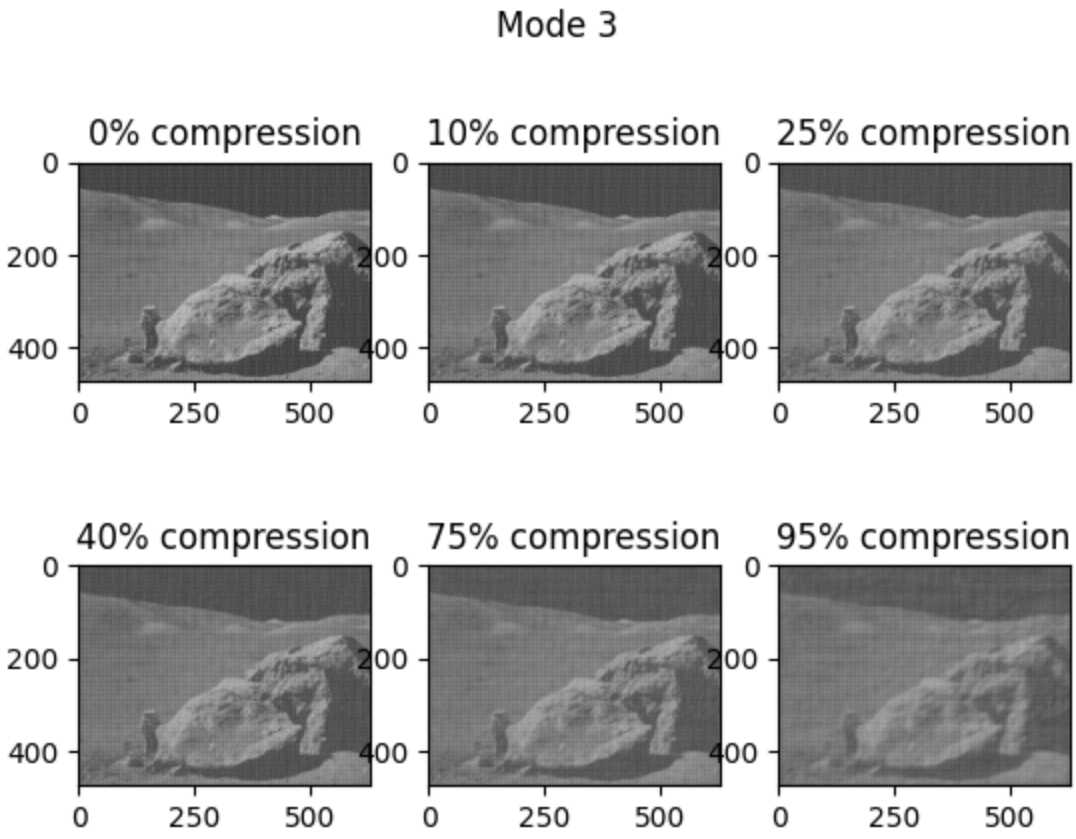
Looking at our implementation using four loops, we get a complexity of O(n^4). Which is basically the naive 1d DFT complexity times itself. Assuming this holds for the 2d FFT, the complexity would be O((n log(n))^2). This assumption is rational, since scaling up the computation should also scale up the complexity proportionally.

Experiments Results:

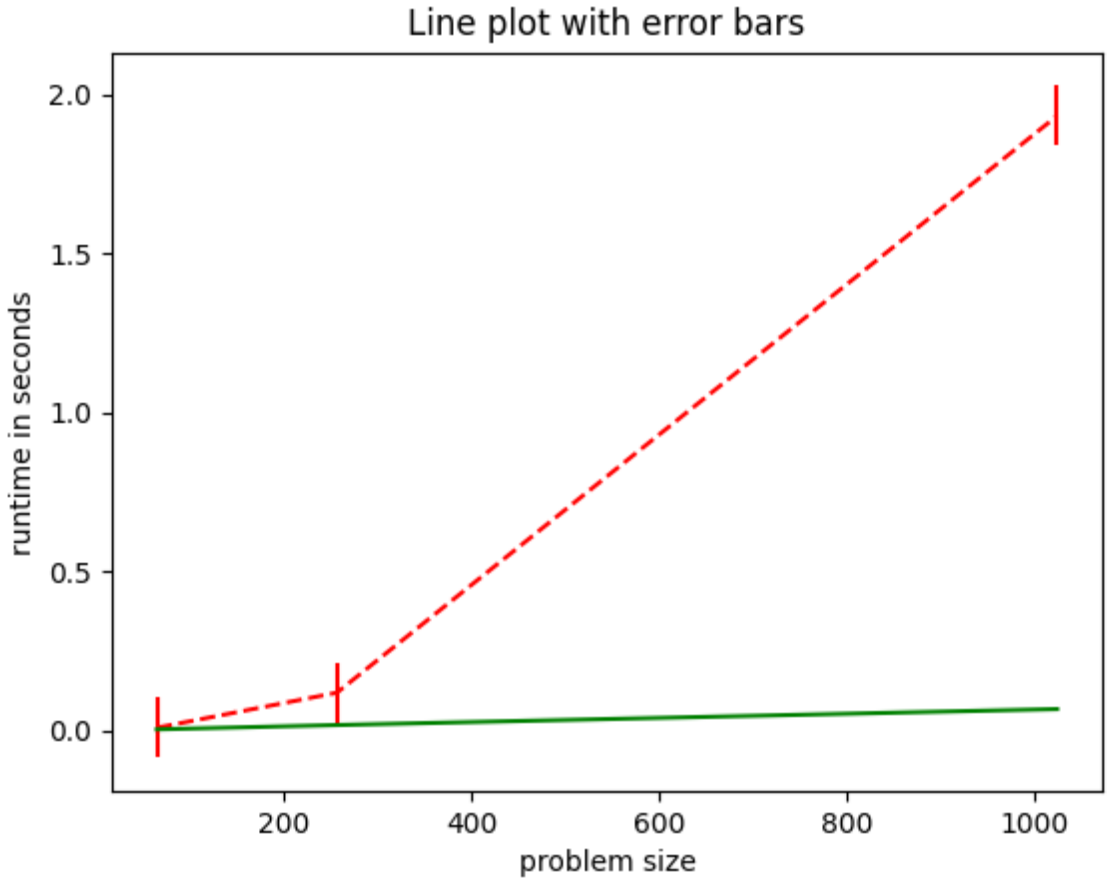


Mode 2





Console Output:
non zero values for level 0% are 298620 out of 298620
non zero values for level 10% are 268758 out of 298620
non zero values for level 25% are 223965 out of 298620
non zero values for level 40% are 179172 out of 298620
non zero values for level 75% are 74655 out of 298620
non zero values for level 95% are 14931 out of 298620



Console Output:
starting measurement for naive_dft_2d
for problem size of 64 over 10 runs: mean 0.007206630706787109, stdev 0.0011363770257610249
for problem size of 256 over 10 runs: mean 0.11756107807159424, stdev 0.0059331355141860895
for problem size of 1024 over 10 runs: mean 1.9339806318283081, stdev 0.09455805473584218
starting measurement for fft_dft_2d
for problem size of 64 over 10 runs: mean 0.0022019147872924805, stdev 0.00042176999509537286
for problem size of 256 over 10 runs: mean 0.01577129364013672, stdev 0.0013385557406787251
for problem size of 1024 over 10 runs: mean 0.06601841449737549, stdev 0.0027087246527760836