Image Compression Algorithms with Wavelets

Brendan Bruce and Dominic DiPuma

Motivation: Wavelets

- Recall Fourier transform - converts signal into complex exponential basis
  - Do not decay
  - Periodic - real signals are often aperiodic
- Wavelets are any family of signals with the following properties:
  - Must form an orthogonal basis
  - Decay towards zero
  - Aperiodic
- Signals in wavelet domain are often more sparse than in Fourier domain
  - We can exploit this property for compression of signals
- Bonus: fast wavelet transforms are O(N), while fast Fourier transforms are O(N log N)

Motivation: Compression

- The same sentence can be conveyed with less characters than the original
    - The same sentence can be conveyed w/ less chars than the original (Lossless)
        - FLAC audio files
        - PNG images
        - File compression formats: zip, gzip, bz2, etc.
    - Same sentence done w/ less chars (Lossy)
        - MP3 audio
        - GIF or JPEG images

Abstract

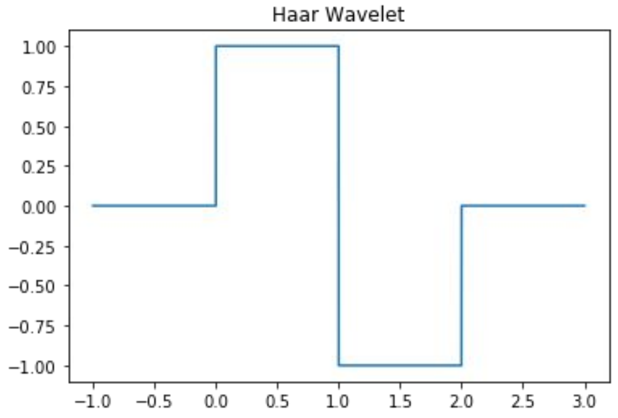
The Fast Wavelet Transformation is an algorithm used to transform a signal in the time domain into a sequence of coefficients whose magnitudes are based on an orthogonal basis of wavelets, similar to the Fast Fourier Transformation.

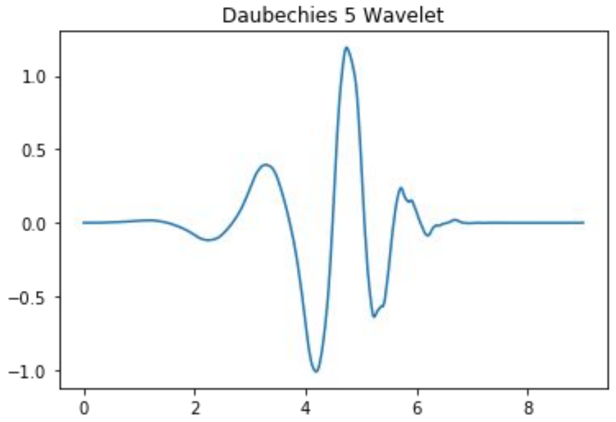
Data compression allows for information to be represented using fewer bits than the original representation. This process can be done in a lossless or a lossy manner based on whether the original representation can be perfectly reconstructed or not.

In this project we would like to implement algorithms for the Fast Wavelet transform and explore using the algorithm for compression of images.

Wavelets

- Any signal with finite power that forms an orthogonal basis
- Shift in time, scale in time, and scale in magnitude
- Haar Wavelet
- Daubechies Wavelets
- Many others



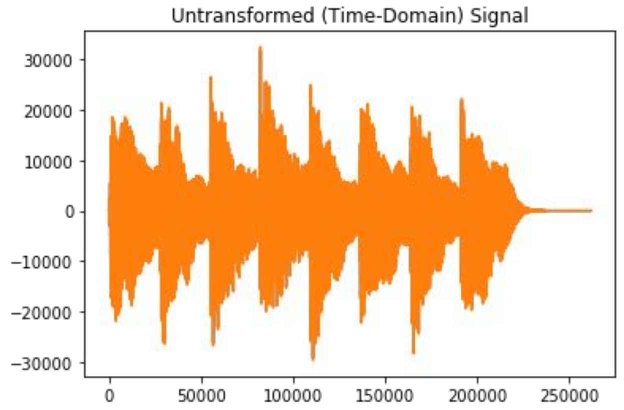


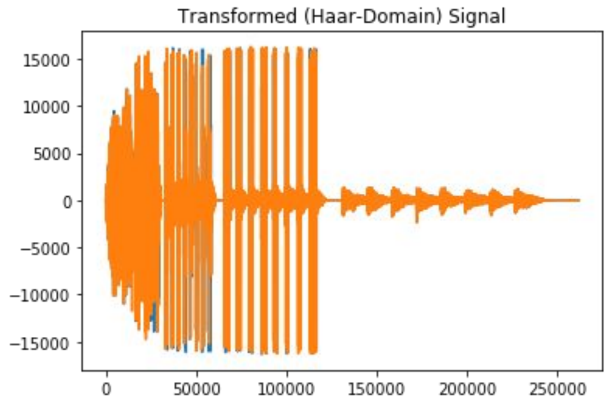
Wavelet Transforms

- Transform a signal into a wavelet basis
- Change of basis is a matrix multiplication
    - Matrix multiplication is slow! $O(N^2)$ for 1D signal of length N
    - Need to do better
- Fast Wavelet Transforms are O(N)
    - Subtly different for each wavelet - not a general algorithm
    - Multiple formulations exist per-wavelet
        - Outputs are different, but still invertible

Haar Wavelet Transform

- Use a single square pulse function as the wavelet
- Assumption: all 1D signals are length power-of-2 (otherwise, zero-pad it)
  - Similar assumption is used for basic fast Fourier transform algorithms
- Assumption: 2D signals are symmetric, and length power-of-2





Haar Wavelet Transform Matrix Algorithm

- Multiply signal by Haar Transform Matrix
    - Generate Haar matrix
        - 2x2 Haar matrix is simple
        - Increase size recursively using Kronecker product
    - Normalize it to find Haar transform matrix
        - Forms orthonormal matrix, which is inverted by a transpose
- 2D transform is identical to 1D transform
- Inverse transform is trivial - just use the inverse (transposed) matrix
- Haar matrix for N = 1 is H_1 = [1 1; 1 -1]
- To calculate larger matrices
    - top = H_{N-1} KRON [1 1]
    - bottom = I KRON [1 -1]
    - stack top and bottom together to form $2^N \times 2^N$ matrix
    - Divide each row by its L2 norm to normalize
    - Return

Complexity Analysis

- Merely a matrix multiplication
- For size NxN Haar transform matrix, size Nx1 signal:
    - $O(N^2)$
- For size NxN Haar transform matrix, size NxN signal (2D case):
    - $O(N^{2.373})$ is possible
    - Naive implementation is $O(N^3)$
- Values of the signal won't change the number of operations performed
    - Best = Average = Worst Case

Fast Haar Wavelet Transform Ordered Algorithm

```
1. Take adjacent pairs from a signal vector
2. do
3.     Populate a new vector, **a**, with the **sum of each pair** divided by 2
4.     Populate another vector, **c**, with the **difference of each pair** divided by 2
5.     Update the signal vector so that it is [**a**, **c**]
6. while (*size*(**a**) > 1)
```

Inverse Fast Haar Wavelet Transform Ordered Algorithm

```
1. size = 2
2. stride = size / 2
3. while size < size(s)
4.     Take pairs from the first size elements of signal, which are stride elements apart
5.     Populate a with the sums and differences of those pairs (sum, difference, sum, …)
6.     Update the signal’s first size elements with a
7.     size = size * 2
```

Complexity Analysis

- First sweep creates an array of N elements after N additions/subtractions
  - Second is N/2, then N/4, then N/8
  - Simple geometric series!
- Sums to 2N calculations
- O(N)
  - Best = Average = Worst Case

Fast Haar Wavelet Transform Inplace Algorithm

For a signal, **s**, of length 2^n

```
1.  $I = 1, J = 2, M = 2^n$
2.  for _ in range(n):
3.      $M = M // 2$
4.      for $K$ in range($M$):                (Sum)                    (Difference)
5.          $\mathbf{s}[J*K], \mathbf{s}[J*K+I] = (\mathbf{s}[J*K] + \mathbf{s}[J*K+I]) / 2, (\mathbf{s}[J*K] - \mathbf{s}[J*K+I]) / 2$
6.      $I = J$
7.      $J = J * 2$
8.  Return s
```

Inverse Fast Haar Wavelet Transform Inplace Algorithm

For a signal, **s**, of length 2^n

```
1. $I = 2^n-1$, $J = 2^n$, $M = 1$
2. for _ in range(n):
3.     for $K$ in range($M$):
4.         s[$J*K$], s[$J*K+I$] = s[$J*K$] + s[$J*K+I$], s[$J*K$] - s[$J*K+I$]
5.     $J = I$
6.     $I = I // 2$
7.     $M$ *= 2
8. Return s
```

(We "undo" the transform through changing indexing variables...

...and changing how they are updated)

Complexity Analysis

- First sweep performs 2 calculations N/2 times (for a total of N calculations)
  - Second performs N/2 total calculations, then N/4, N/8, …
- Same complexity as ordered case, but slightly different arrangement
- O(N)
  - Best = Average = Worst Case

2D Fast Haar Transforms

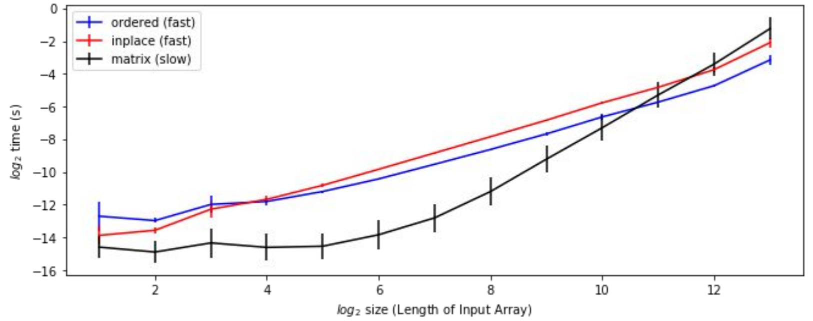
- Apply 1D fast Haar transform to each row
- Apply 1D transform again to each column
- For an NxN signal, perform an O(N) operation 2N times
    - $O(N^2)$

Experimental Configuration

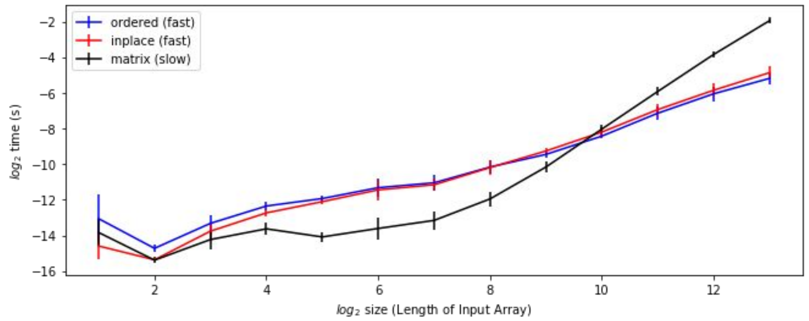
Values of the input do not matter so we used either 1D or 2D arrays of random values

Ran 20 trials for each array size and calculated the mean and standard deviation of the runtime

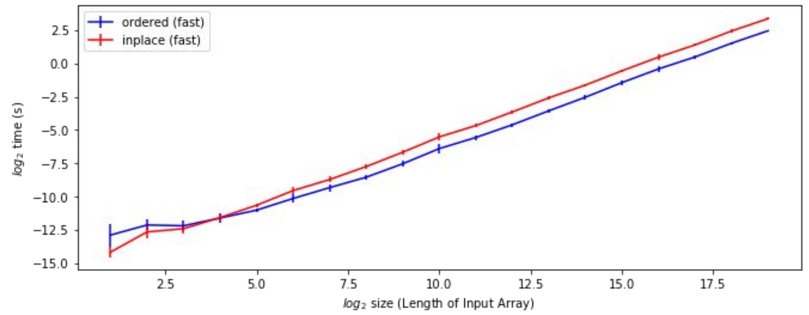
1D Haar Transform Algorithm Runtimes



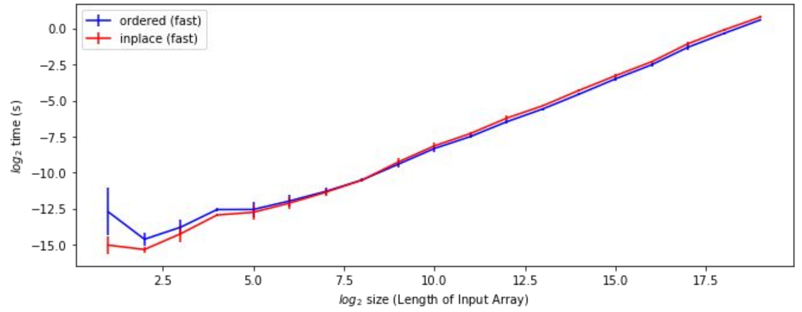
1D Inverse Haar Transform Algorithm Runtimes



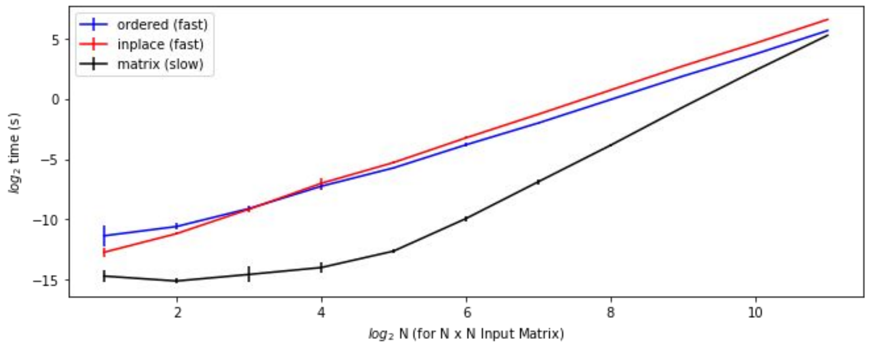
1D Fast Haar Transform Algorithm Runtimes



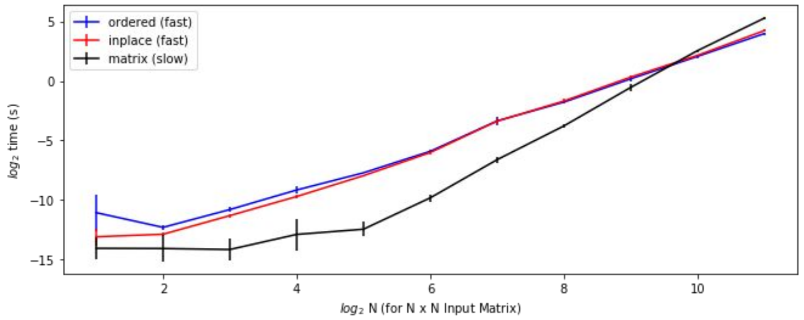
1D Fast Inverse Haar Transform Algorithm Runtimes



2D Haar Transform Algorithm Runtimes



2D Inverse Haar Transform Algorithm Runtimes



Compression using Wavelets

To compress a signal using a wavelet transform, we transform the signal into the basis specified by the wavelet

The result is a signal of the same length whose values are the wavelet coefficients, which represent how much each wavelet basis vector contributes to the signal

This alone will compress the signal and allow for perfect reconstruction **(Lossless)**

Further compression can be performed by setting the small wavelet coefficients to 0, which does not allow for perfect reconstruction **(Lossy)**

Compression ratio is the ratio of nonzero elements in the uncompressed transformed image to the nonzero elements in the compressed transformed image

Compression Algorithm

```
Inputs: image, desired compression ratio

Perform forward FWT on the image to get wavelet coefficient matrix
Initialize threshold variable    # All values in the coefficient matrix under this value will get set to 0
Initialize temporary compression ratio variable
While desired compression ratio != temporary compression ratio:
        Change threshold variable value
        Compute number of nonzero elements with new threshold value
        Update temporary compression ratio with new value
End
Set all values in the wavelet coefficient matrix less than the threshold to 0
Perform inverse FWT to get the resulting image
```

Compression Algorithm: Complexity Analysis

Breaking down the algorithm into stages we have

1. Fast Wavelet Transform stage
2. Coefficient Thresholding stage
3. Inverse Fast Wavelet Transform stage

For an image with **N pixels**, the two FWT stages are **O(N)** and because values of the images don’t matter this is the best, average and worst case

The coefficient thresholding was implemented using a percentile calculation to find a starting point for the threshold value and then a while loop to adjust from there. This operation as a whole should be **O(N)** (with the numpy percentile implementation being O(N)).
In the case where no coefficient thresholding is desired then this stage drops to **O(1).**

As a whole our compression algorithm is **O(N)**, best/average/worst case for images with N pixels

Experimental Configuration

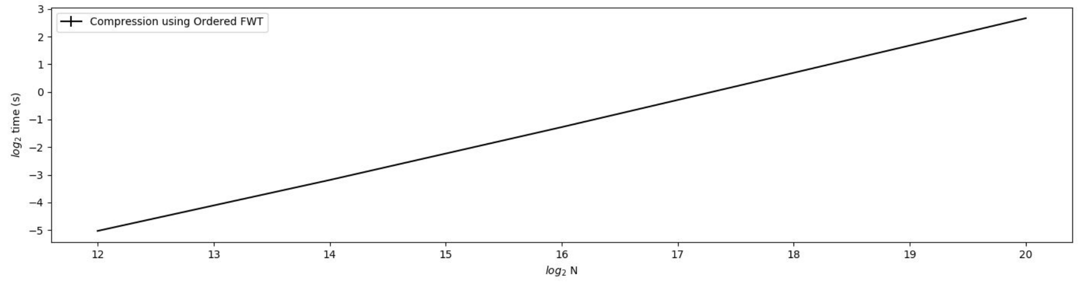
Further experimentation is going to be done tracking runtime as the the desired compression ratio increases

For now we look at runtime for a constant compression ratio of 0 (equivalent to lossless compression)

Used same setup as experiments for testing the 2D transforms

Compression Algorithm Experimental Runtime

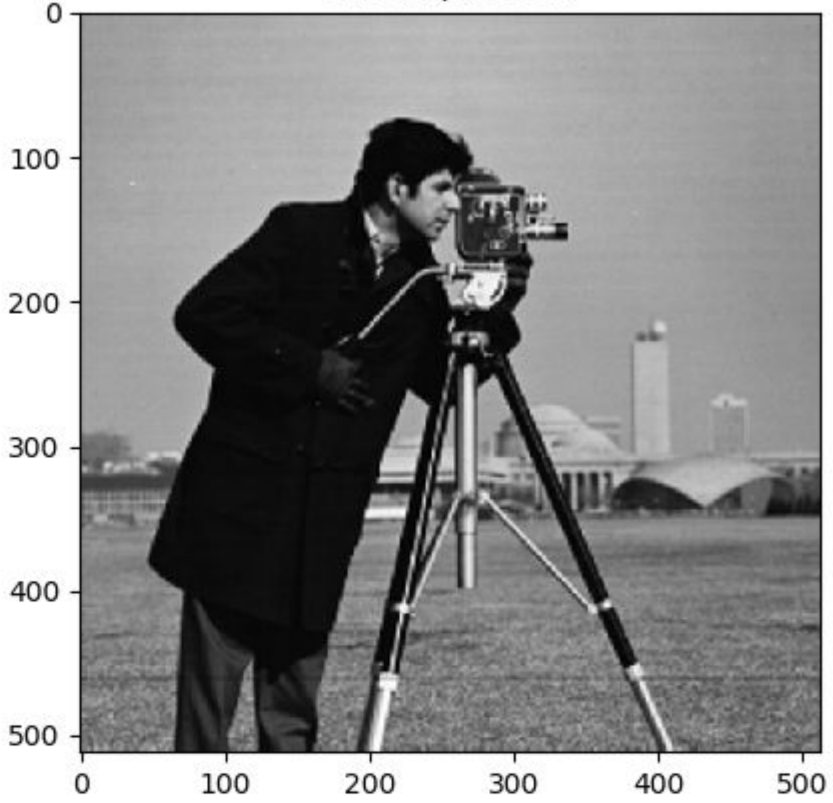
Compression Runtimes



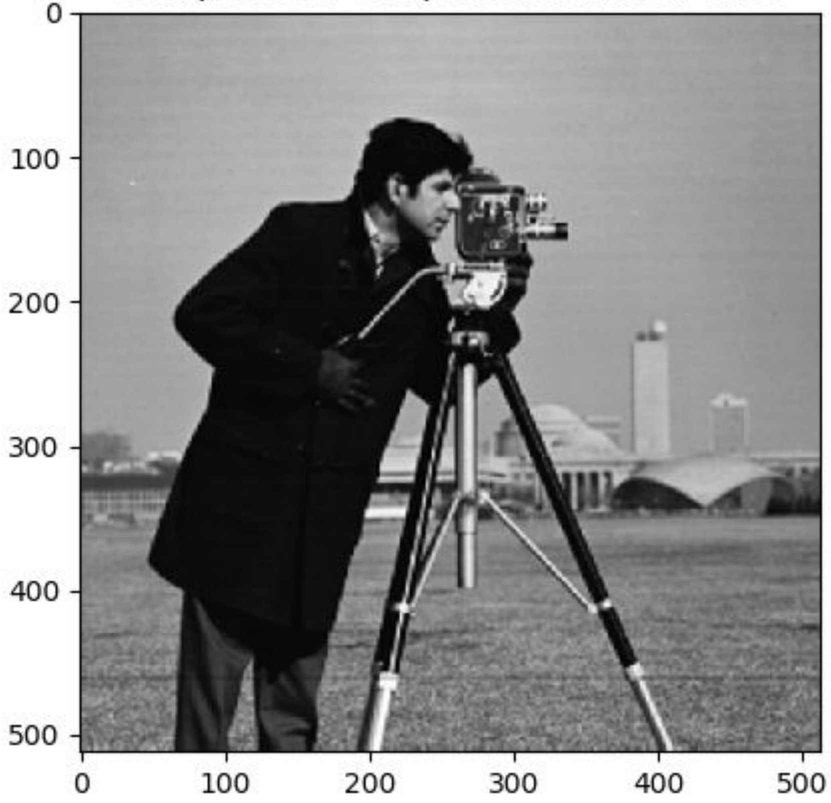
	C
N = 4096 pixels	0.030517898218047835 seconds
N = 16384 pixels	0.10953534985768278 seconds
N = 65536 pixels	0.4133525997311048 seconds
N = 262144 pixels	1.6162768824086031 seconds
N = 1048576 pixels	6.360807482037308 seconds

Image Quality with Varying Compression Ratios

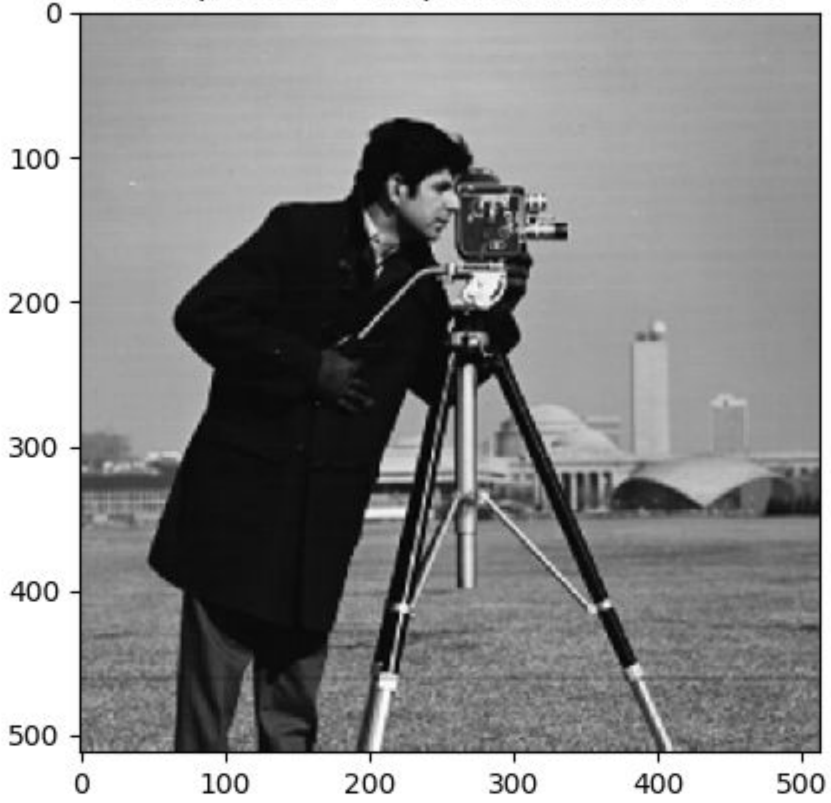
Uncompressed



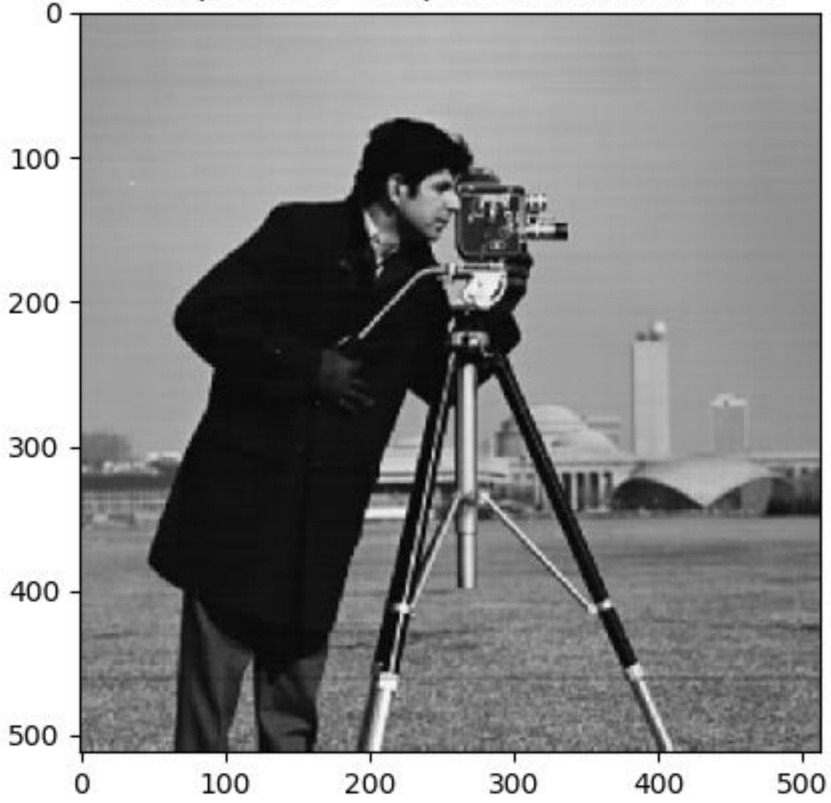
Compressed, compression ratio of 1.98

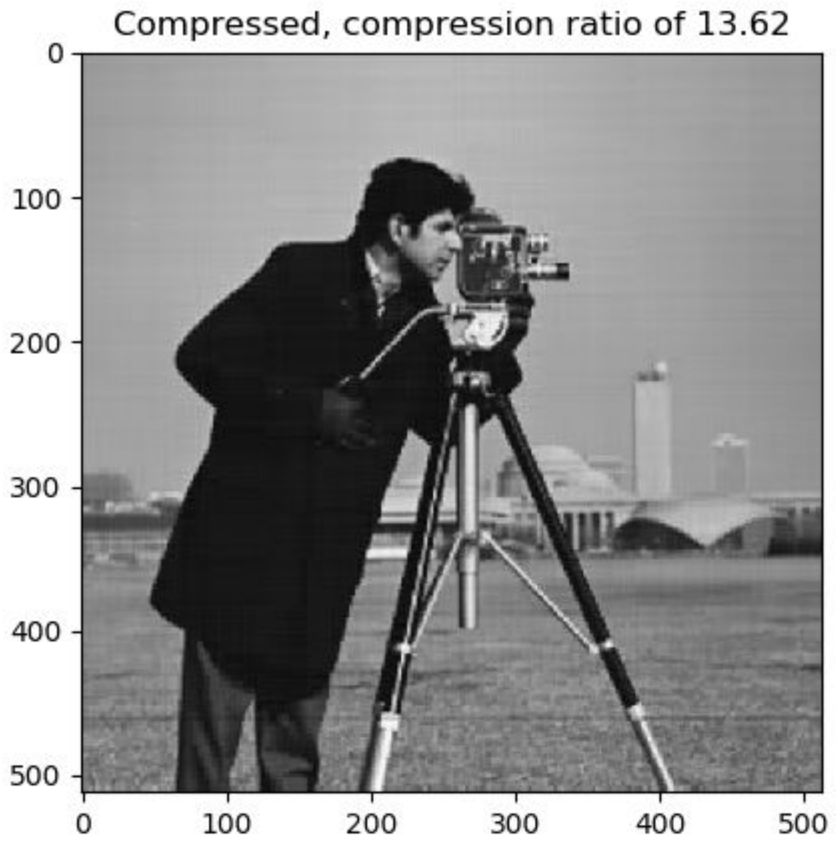


Compressed, compression ratio of 3.78

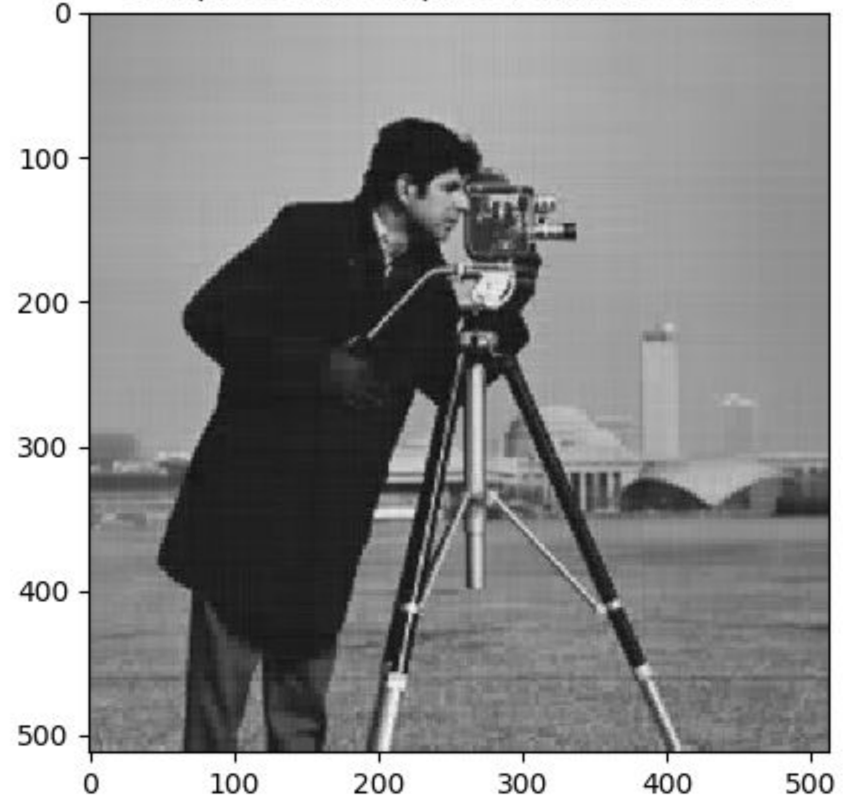


Compressed, compression ratio of 7.28

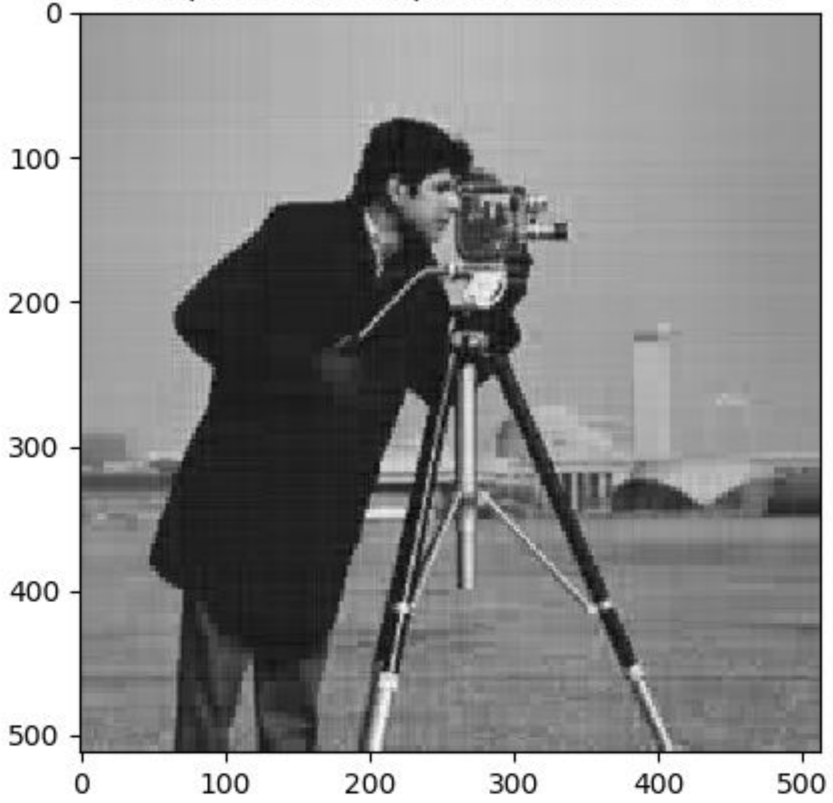



Compressed, compression ratio of 13.62
0
100
200
300
400
500
0
100
200
300
400
500

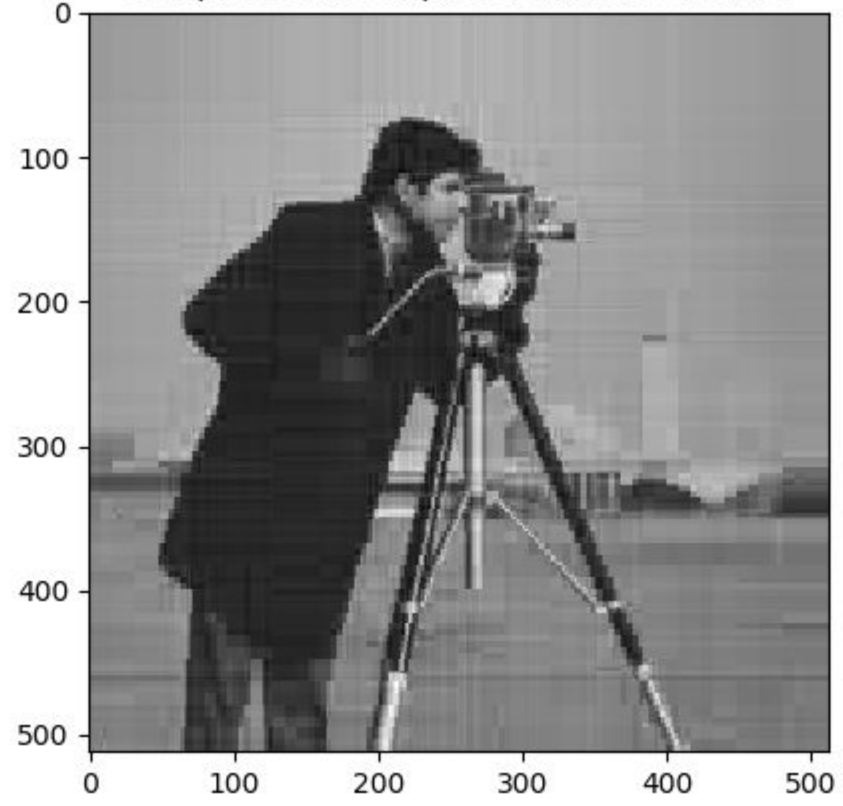
Compressed, compression ratio of 23.15



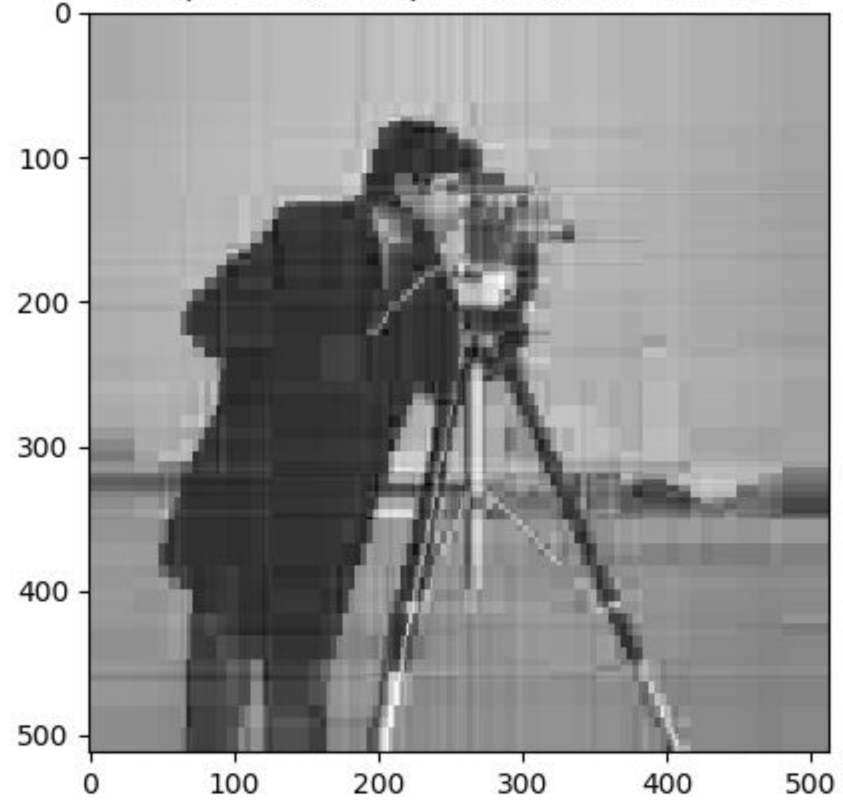
Compressed, compression ratio of 45.06



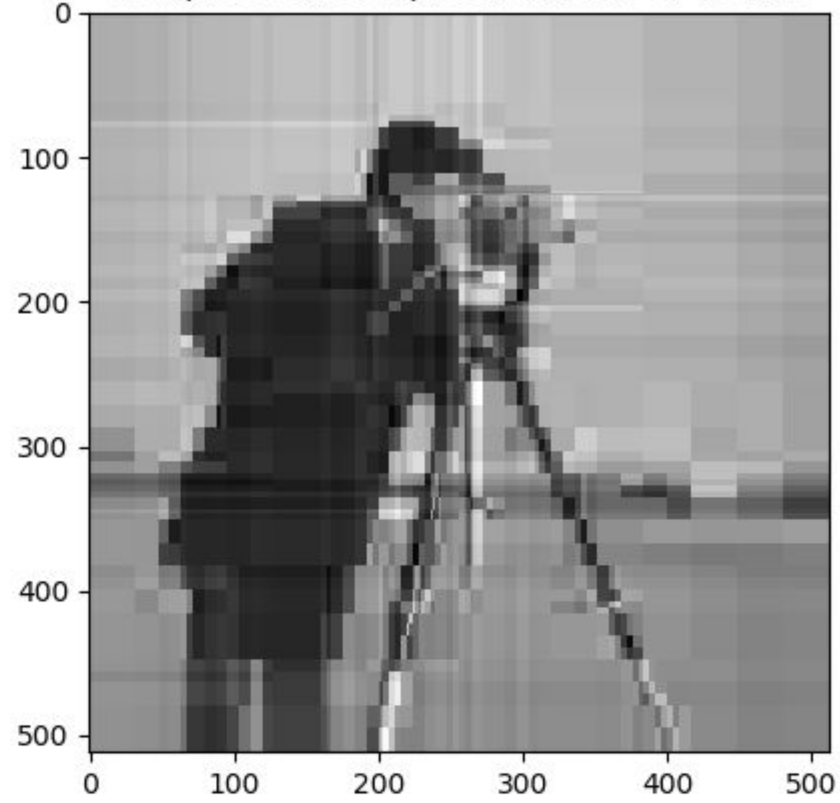
Compressed, compression ratio of 94.26



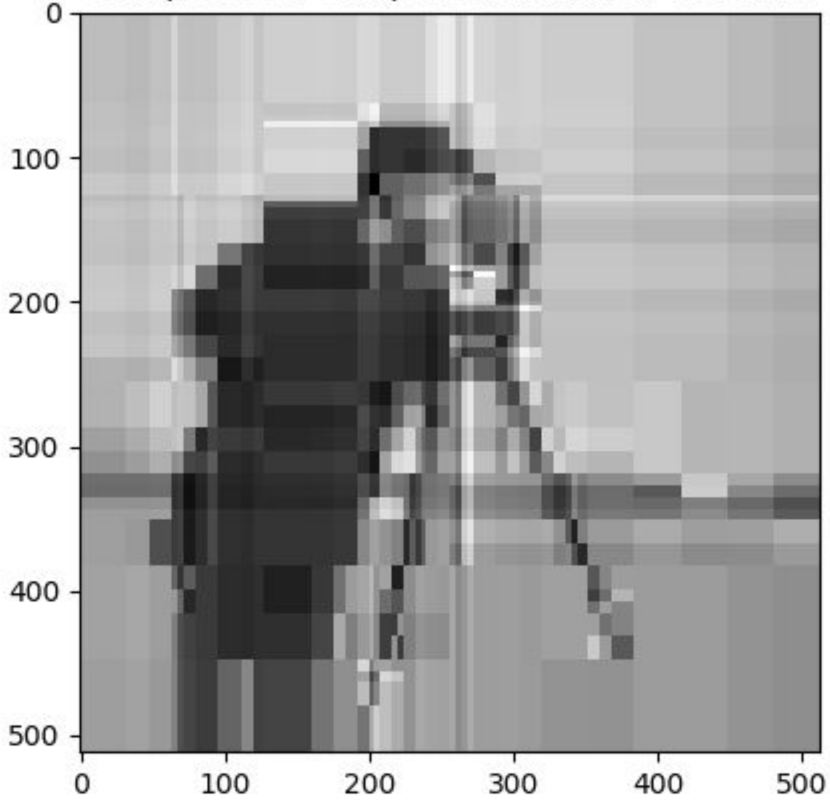
Compressed, compression ratio of 225.31



Compressed, compression ratio of 482.75



Compressed, compression ratio of 1015.72



Discussion

A couple of topics covered in class appeared while we were working on this project

- The performance of the 2D matrix transform was better than the others, despite having a worse complexity growth until the problem size increased
  - Matrix multiplication implementation is well optimized
- Generation of Haar matrices has a nice recursive definition
  - Python’s built-in dynamic programming (lru_cache decorator) makes this practical

Conclusion

- We didn’t realize that the Wavelet transform isn’t as standardized as the Fourier transform
    - Lost a lot of time debugging non-issues
    - Time permitting we are going to look to implement a transform for a different Wavelet series besides Haar
- Compression works for most cases but some edge cases cause infinite looping

Sources

- RIET, Phagwara. "ANALYSIS OF WAVELET TRANSFORM AND FAST WAVELET TRANSFORM FOR IMAGE COMPRESSION."
- Taswell, Carl, and Kevin C. McGill. "Wavelet transform algorithms for finite-duration discrete-time signals." ACM Transactions on Mathematical Software 20.3 (1994): 398-412.
- Nievergelt, Yves, and Y. Nievergelt. Wavelets made easy. Vol. 174. Boston: Birkhäuser, 1999.