Fast Wavelet Transform and Image Compression

Brendan Bruce and Dominic DiPuma

23 April 2019

1 Introduction

As our society continues transferring more data each year, it is increasingly important for engineers and scientists to develop compression algorithms which allow us to transmit the same messages with fewer bits. This can save time and bandwidth, but compression does not come for free. Compression requires processor time to complete and media compression often incurs losses in fidelity which cannot be recovered. As such, it is important for compression algorithms to be fast, and to preserve as much quality as possible.

Some older and more naive compressed formats rely on frequency-domain transforms, such as the discrete Fourier transform or closely related discrete cosine transform, to compress their data. In short, these algorithms transform the time-domain data, pick off the most significant frequency components, and encode those frequencies alone. By removing the frequencies which have little impact on the data, file size is reduced immensely. Moreover, algorithms such as the FFT allow these operations to be performed in O(N log N), so the time complexity is not a major hurdle.

However, representing data in the frequency domain is not appropriate for all forms of data. This is where wavelet transforms come in. Wavelet transforms use sets of orthonormal basis vectors, dubbed wavelets, as the target domain for the transformation. This should allow for improved compression over the Fourier and cosine transform approaches.

Wavelet transforms, performed naively, are $O(N^2)$ operations. However, fast wavelet transform algorithms exist, which exhibit O(N) performance by using filter banks.

We intend to study fast wavelet transforms and use them to implement a compressed image format. This includes exploring runtime complexity of the transformation algorithms, the differences between lossy and lossless implementations, and analysis of compression ratios.

1.1 Background on Haar Wavelets

During our project, we found that wavelet transform algorithms are not as general as we thought they were. For this reason, we decided to focus on the Haar wavelet. The Haar wavelet is the most simple version of a wavelet being an orthogonal set of square pulses. The power behind the simplicity of the wavelet is that operations involving the wavelet can be done with simple addition and subtraction operations. Compare to the Fourier transform where multiplications using complex exponential factors are necessary. The disadvantage that comes with being composed of square pulses is that they are not differentiable. This limits their effectiveness in applications involving certain types of real-world signals.

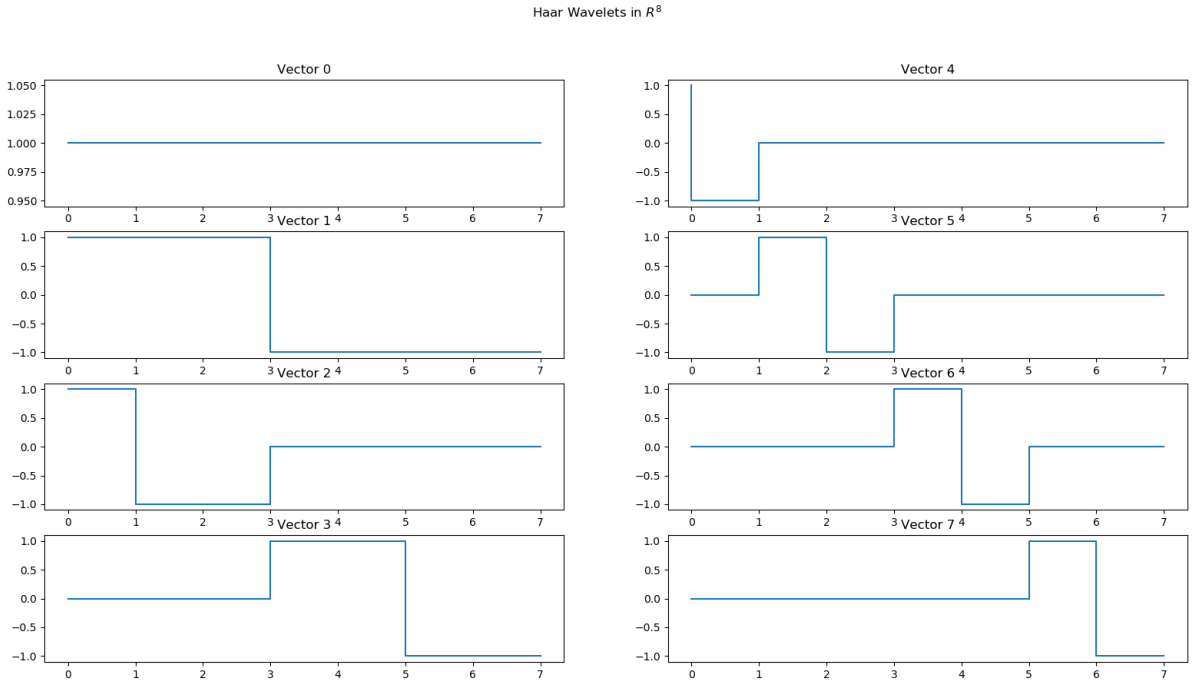


Figure 1: Haar Wavelet Basis

An important property of the Haar wavelet can be seen in 1. The Haar basis vectors must be generated in powers of 2 in order to maintain orthogonality. When the desired basis is length 2^0, the corresponding basis is spanned by Vector 0. When the desired basis is length 2^1, the corresponding basis is spanned by Vector 0 and Vector 1. When the desired basis is length 2^2, the corresponding basis is spanned by Vector 0-3. When the desired basis is length 2^3, the corresponding basis is spanned by Vector 0-7. Taking a look at each individual basis, we can see the important property in play. As more basis vectors are incorporated, the resolution of the new basis vectors increases.

This property is key for enabling the use of the Haar wavelet for image compression.

1.2 Background on Image Compression

In general, we are interested in two forms of image compression: lossy and lossless. Lossy compression refers to compression that causes the reconstructed signal to contain less information than the original signal. Lossless compression refers to compression that causes the reconstructed signal to contain the same information as the original signal.

Lossless compression can be achieved through a wavelet transform when the coefficients generated by the transform are smaller to store than the original signal. Now instead of sending an image, we would send the coefficients generated by the transform and tell the recipient to reconstruct using the Haar basis.

Lossy compression is the more interesting case. In the previous section, it was mentioned that the key property for the Haar wavelet with regards to image compression was the increasing resolution in the basis vectors. The reason this property allows us to implement efficient compression is because as the resolution of the basis vectors increase, the coefficients generated for the basis vectors tend to be smaller. In lossy compression, we choose threshold values where any coefficient smaller than the threshold value is set to 0. This causes our coefficient matrix to be sparse which allows for greater compression. The trade off is that as the threshold increases, the image quality decreases. For our project, we use the metric of compression ratio, the number of nonzero elements in the uncompressed matrix:the number of zero elements in the compressed matrix, and implement an image compression algorithm that selects the threshold according to the desired compression ratio.

2 Algorithms

2.1 Haar Matrix Generation

A reference "naive" implementation of the Haar wavelet transform uses the notation of linear algebra to express a signal in terms of Haar wavelet basis vectors. In linear algebra, a change of basis operation is performed by a matrix multiplication, using a transformation matrix. In the case of the Haar wavelet transform, this transformation matrix is well-defined for sizes equal to powers of two.

2.1.1 Recursive Generation

The Haar transformation matrix H_N, where 2^N is the size of the square matrix, is most easily expressed using recursion. This relies on a base case:

$$H_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Moreover, the representation uses I_N as the $N \times N$ identity matrix, and relies on the Kronecker product for its computation. The Kronecker product is a linear algebraic operator between two matrices, and is represented by $\otimes$:

$$A \otimes B := \begin{bmatrix} a_{11}B & a_{12}B & \dots & a_{1n}B \\ a_{21}B & a_{22}B & \dots & a_{2n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & a_{m2}B & \dots & a_{mn}B \end{bmatrix}$$

which assumes that A is of size $m \times n$ and that B is of arbitrary size.

With this notation in place, the Haar matrix can be readily defined:

$$H_N = \begin{bmatrix} H_{N-1} \otimes \begin{bmatrix} 1 & 1 \end{bmatrix} \\ I_{N-1} \otimes \begin{bmatrix} 1 & -1 \end{bmatrix} \end{bmatrix}$$

For the purposes of making this transformation easily invertible, it is desirable to represent the Haar transform matrix as an orthonormal matrix. (The above definition is for an orthogonal matrix, which can be made orthonormal by dividing each element in the matrix by the L_2 norm of its row.) The advantage of using an orthonormal matrix is that its inverse is achieved by a transpose operation, which is trivial.

The theoretical performance of this algorithm is the same in the best, worst, and average cases because it is a purely arithmetic operation. That is, a problem of size N will always exhibit the same performance, because the output and intermediate calculations will always be the same.

The time complexity of this algorithm is related to the time complexity of the Kronecker product. For the calculation of $A \otimes B$ with A of size $m \times n$ and B of size $p \times q$, each element of the matrix B is scaled by each element of A. This is therefore a $O(mnpq)$ operation. For this particular algorithm, 2 Kronecker products are calculated, for A of size $2^{N-1} \times 2^{N-1}$ and B of size 1×2. Thus, assuming H_{N-1} is known, the calculation of H_N is $O(2 * 2^{N-1} 2^{N-1})$. Removing constant coefficients, this is $O(2^N)$. Recall that this generates a matrix of size $2^N \times 2^N$, which means that this algorithm is linear time with respect to the size of the matrix, again, provided that H_{N-1} is already known. If H_{N-1} is not known, as is generally the case, the algorithm requires the computation of a square matrix with size 2^N, 2^{N-1}, 2^{N-2}, and so on. However, this is still linear time, as the smaller operations are $O(M/2)$, $O(M/4)$, and so on, (where M is the matrix size 2^N) which results in an $O(2M)$ overall time complexity. This reduces down to $O(M)$, so that the recursive calculation of H_N is possible in linear time.

Normalizing a matrix requires quadratic time, however. For an $M \times M$ matrix, the norm of each row must be calculated (which requires M multiplications, $M - 1$ additions, and 1 square root operation), and this must be done M times (once per row), resulting in $O(M^2)$*complexity*. Each of the M^2 elements must also be divided by its rows norm.

As a result, the calculation of a given Haar transform matrix is quadratic in the size of the matrix.

2.1.2 Iterative Generation

An iterative implementation of the Haar matrix generation was experimented with during this project, following a MATLAB implementation. Due to the lack of documentation and no references to how the algorithm works, we went with the recursive implementation. A runtime comparison of the two implementations can be found in our result section.

2.2 Haar Matrix Wavelet Transform

The change-of-basis operation using the Haar transform matrix is simple to express.

In one dimension, for a signal $\vec{x}$ of size $M = 2^N$, the transform requires one to multiply $\vec{x_H} = H_{N,norm}\vec{x}$, where $\vec{x_H}$ is the Haar-transformed signal. This multiplication requires M outputs to be calculated, each of which is the result of M multiplications and $M-1$ additions. As such, it is an $O(M^2)$ operation, and the Haar wavelet transform in matrix notation is quadratic-time in one-dimension.

For the two dimensional transform, to transform a signal X which is $M \times M$ for $M = 2^N$, the calculation is $X_H = H_{N,norm} X H_{N,norm}^T$. Naively, at each of the two stages of matrix multiplication, the output matrix has M^2 elements, each of which is calculated via M multiplications and $M-1$ additions, yielding cubic time complexity, or $O(M^3)$. However, optimized linear-algebra algorithms allow this matrix multiplication to be performed with slightly smaller coefficients, such as $O(N^{\log_2 7})$[1].

2.3 Inverse Haar Matrix Wavelet Transform

Since the Haar transform matrix $H_{N,norm}$ is an orthonormal matrix, its inverse is merely $H_{N,norm}^T$. As a result, the 1D Haar transform is $\vec{x_H} = H_{N,norm}^T \vec{x}$, which is still a quadratic-time algorithm. In addition, the 2D Haar transform is represented as $X_H = H_{N,norm}^T X H_{N,norm}$, which is again a cubic-time algorithm, or a quadratic-time algorithm with optimized matrix multiplication.

2.4 Ordered Fast Wavelet Transform

The ordered version of the forward Fast Haar Wavelet Transform uses adjacent pairs of values from the signal, computes their sum and difference and puts them in new vectors. These vectors grow until they both reach full length, equal to half of the signal. Then the signal vector is updated to be equal to the two vectors appended. This is repeated for each of the values in the a vector, which results in new a and c vectors of length that is now a quarter of the original signal and half of the a vector. This repeats until the a vector is size 1.

Algorithm 1 Ordered Fast Wavelet Transform

```
1: procedure OrderedHaarFWT(signal vector of length 2^n)
2:     Initialize vector a, that is a copy of the signal vector
3:     do
4:         Populate vector a_new with the sum of each pair of elements in a, divided by 2
5:         Populate vector c with the difference of each pair of elements in a, divided by 2
6:         Update the a so that it is [a_new, c]
7:         On the next iteration, treat elements of a_new as a
8:     while size(a) < 1
       return a
```

Each while loop iteration performs a number of operations equal to the size of *veca*. This size decreases by a factor of two with each iteration of the while loop, as $\vec{a}$ is halved on each iteration. Therefore, the algorithm performs $N + \frac{N}{2} + \frac{N}{4} + \frac{N}{8} + \cdots + 1 = N\sum_{i=0}^{N}(\frac{1}{2})^i = 2N$ operations, which is $O(N)$.

2.5 Inverse Ordered Fast Wavelet Transform

The ordered version of the inverse Fast Haar Wavelet Transform looks to undo the transform by striding down the array with different lengths and finding the sum and differences of the pairs.

Algorithm 2 Ordered Inverse Fast Wavelet Transform

```
1: procedure InverseOrderedHaarFWT(signal vector of length 2^n)
2:     size = 2
3:     stride = size / 2
4:     while size < 2^n do
5:         Take pairs from the first size elements of signal, which are stride elements apart
6:         Populate a⃗ with the sums and differences of those pairs (sum, difference, sum, ...)
7:         Update the signal's first size elements with a⃗
8:         size = size * 2
       return signal
```

The inverse ordered fast transform has the same analysis and time complexity as the forward ordered transform.

2.6 In-place Fast Wavelet Transform

The in-place version of the forward and inverse Fast Haar Wavelet Transform allows for the transform to be done in constant space by computing the sums and differences of pairs on different chunks of the signal and then updating the values of the pairs with the sums and differences.

Algorithm 3 In-place Fast Wavelet Transform

```
1: procedure InPlaceHaarFWT(s, signal of length 2^n)
2:     I ← 1, J ← 2, M ← 2^n
3:     for each in [1...n] do
4:         M = M // 2
5:         for K in [1...M] do
6:             s[J*K], s[J*K+I] = (s[J*K] + s[J*K+I]) / 2, (s[J*K] - s[J*K+I]) / 2
7:         I = J
8:         J = J * 2
       return s
```

The outer for loop is run N times, but the number of operations performed in that loop decreases with each iteration. That is because the inner for loop's dataset is halved with each iteration. Therefore, the algorithm performs $N+\frac{N}{2}+\frac{N}{4}+\frac{N}{8}+\cdots+1 = N\sum_{i=0}^{N}(\frac{1}{2})^i = 2N$ operations, which is $O(N)$.

2.7 Inverse In-place Fast Wavelet Transform

The in-place version of the inverse Fast Haar Wavelet Transform has the same inside loop as the forward transform but changes the indexing variables to allow for the transform to be undone.

Algorithm 4 Inverse In-place Fast Wavelet Transform

```
1: procedure InverseInPlaceHaarFWT(s, signal of length 2^n)
2:     I ← 2^n − 1, J ← 2^n, M ← 1
3:     for each in [1...n] do
4:         for K in [1...M] do
5:             s[J*K], s[J*K+I] = (s[J*K] + s[J*K+I]) / 2, (s[J*K] - s[J*K+I]) / 2
6:         J = J * 2
7:         I = I // 2
8:         M = M * 2
       return s
```

The analysis of this algorithm is identical to that of the forward in-place algorithm.

2.8 Image Compression Algorithm

To implement image compression we perform a FWT to get the wavelet coefficients, threshold the coefficients based on the desired compression ratio, then perform an inverse FWT to get the compressed image. Because we have already analyzed the FWT and inverse FWT, we know that it scales linearly with the number of pixels in the image. In order to be efficient with our algorithm, we had to use a method to find and threshold coefficients that scaled with the number of pixels. What we came up with was to use the Quick Select algorithm studied in class. The Quick Select algorithm allows us to select the kth smallest element from an unsorted array. This algorithm is O(n) in the average-best case and, with a shuffle step, becomes incredibly unlikely to ever reach the $O(n^2)$ worst case. To select the value for our threshold, we compute the number of nonzero elements in the uncompressed wavelet coefficient matrix and using that we can compute the number of nonzero elements in the compressed wavelet coefficient matrix based on the desired compression ratio. Then we use Quick Select to select the value from the matrix that is the kth smallest, where k is the number of nonzero elements we desire in the compressed wavelet coefficient matrix. We then apply this threshold to the coefficient matrix.

Algorithm 5 Image Compression Algorithm

```
1: procedure HAAR_IMAGE_COMPRESSION(image: size of N pixels, target_compression_ratio)
2:     wavelet_coefficients = FWT(image)
3:     Find number of nonzero elements in wavelet_coefficients
4:     target_nonzero = uncompressed_nonzero // target_compression_ratio
5:     threshold = QuickSelect(wavelet_coefficients, target_nonzero)
6:     Set elements of wavelet_coefficients that are less than threshold to 0
7:     compressed_image = iFWT(wavelet_coefficients)
```

The FWT is O(N) as analyzed in the previous sections. Finding the number of nonzero elements in the coefficient matrix is an O(N) operation, looking at each pixel to see if it is 0 or not. Computing the target nonzero elements is an O(1) operation, a single division (and rounding to the nearest integer). The Quick Select is O(N) as mentioned previously. Applying the threshold is an O(N) operation, looking at each pixel and modifying it if it is less than the threshold or not. The iFWT is O(N) as analyzed in the previous sections. Giving our image compression algorithm an O(N) runtime complexity where N is the number of pixels in the image to be compressed.

3 Experimental Setup

All algorithms and experiments are implemented using Python 3.7 and Python libraries NumPy (scientific computation), imageio (handling image input/output) and matplotlib (visualization).

3.1 Wavelet Transform Algorithms

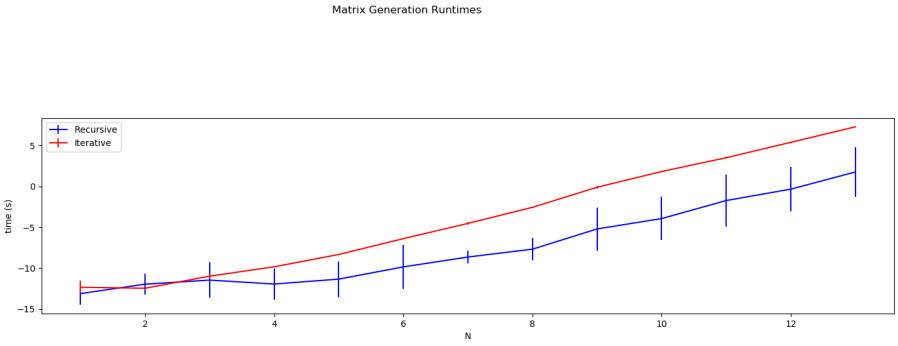
The values of the input signal do not matter so we used either 1D or 2D arrays of random values depending on what was being tested, running 20 trials of randomly generated signals for each array size and calculated the mean and standard deviation of the runtime. This setup allows us to capture the runtime complexity of the algorithms with respect to input size.

3.2 Image Compression Algorithm

We also use a similar experimental setup to investigate the runtime complexity of our image compression algorithm. In addition, we do a qualitative investigation on how the compression ratio changes the quality of the reconstructed images and track the runtime for the algorithm for different compression ratios on the same image.

4 Results

4.1 Haar Matrix Generation

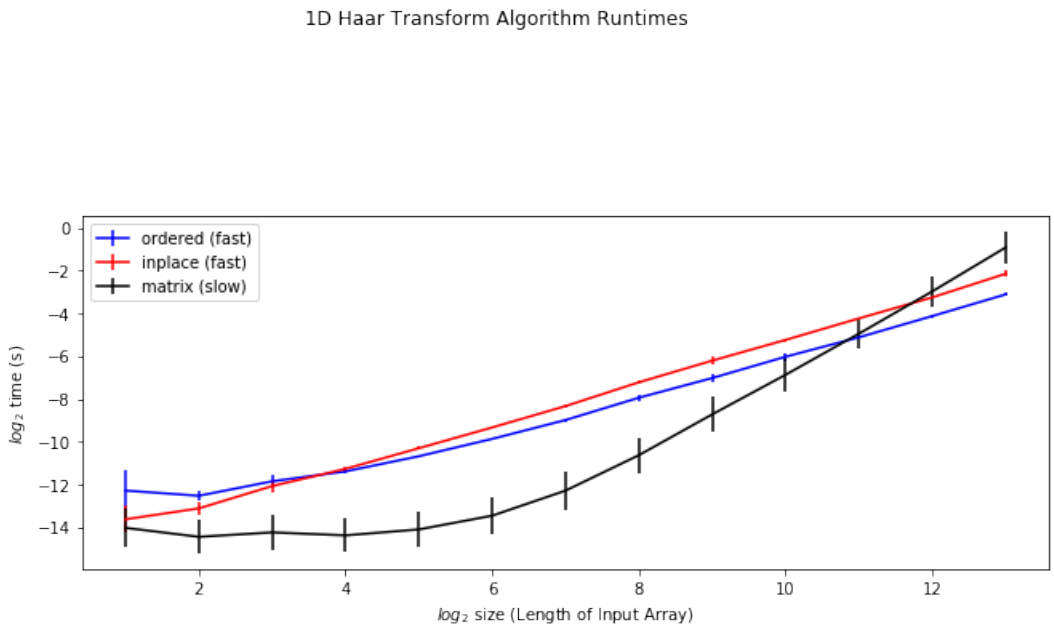


	Recursive Matrix Generation	Iterative Matrix Generation
Matrix Size = N x N = 2 x 2	0.0001126401601993389 5 seconds	0.0001931380524630079 seconds
Matrix Size = N x N = 4 x 4	0.000250311467109642 seconds	0.0001774935857686553 seconds
Matrix Size = N x N = 8 x 8	0.0003544182818393338 seconds	0.0004946495923905314 seconds
Matrix Size = N x N = 16 x 16	0.0002540092501464889 4 seconds	0.0010948282237556962 seconds
Matrix Size = N x N = 32 x 32	0.0003848538806810750 5 seconds	0.0031212133279478436 seconds
Matrix Size = N x N = 64 x 64	0.001086010433437059 seconds	0.011922790290190633 seconds
Matrix Size = N x N = 128 x 128	0.0025082346783782093 seconds	0.043524897457631946 seconds
Matrix Size = N x N = 256 x 256	0.0048924514025975585 seconds	0.17059039372856 seconds
Matrix Size = N x N = 512 x 512	0.027164767523224914 seconds	0.9344110000512 seconds
Matrix Size = N x N = 1024 x 1024	0.06554604877660264 seconds	3.55265387044106 seconds
Matrix Size = N x N = 2048 x 2048	0.302917550816072 63 seconds	11.460027249816534 seconds

Figure 2: Timing of the Iterative and Recursive Haar Matrix generation

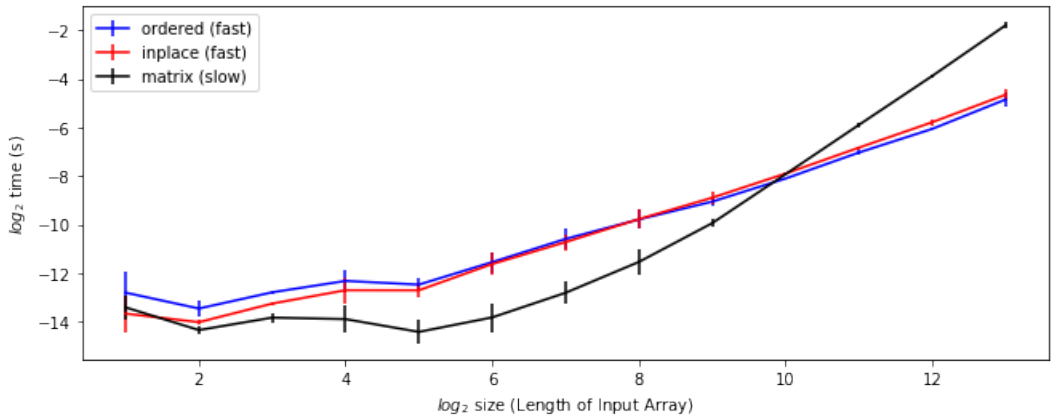
Both methods seem to exhibit the same runtime growth. The recursive method runs faster than the iterative method. The recursive method uses a LRU-caching dynamic programming which is likely one of the reasons for its better performance.

4.2 Wavelet Transform Algorithms



	Ordered Haar Transform Runtime	In-Place Haar Transform Runtime	Matrix Haar Transform Runtime
Input Length = 2	0.00020236352003848878 seconds	7.996664004167542e-05 seconds	6.058908000341034e-05 seconds
Input Length = 4	0.00017115512000600574 seconds	0.00011382476002836484 seconds	4.535644000497996e-05 seconds
Input Length = 8	0.00027339520001987 69 seconds	0.00023433031996319186 seconds	5.231860001003952e-05 seconds
Input Length = 16	0.00037650399999620274 seconds	0.00040898595994804055 seconds	4.75724799616728e-05 seconds
Input Length = 32	0.0006144606800262409 seconds	0.000797141399980319 seconds	5.762364002293907e-05 seconds
Input Length = 64	0.001077969279995159 seconds	0.0015611099199850287 seconds	8.987824006908341e-05 seconds
Input Length = 128	0.001984025719957572 seconds	0.003114384600048652 seconds	0.000202208119972056 seconds
Input Length = 256	0.00408197171995198 seconds	0.006723005600504225 seconds	0.0006365459200787882 seconds
Input Length = 512	0.007713032359988574 seconds	0.01354521211998872 seconds	0.002383619399915915 seconds
Input Length = 1024	0.015365268120058317 seconds	0.026487715719995322 seconds	0.008570279959967593 seconds
Input Length = 2048	0.028783329079997203 seconds	0.05309298959999069 seconds	0.03238574776007226 seconds
Input Length = 4096	0.05720692700004292 seconds	0.10471900108001136 seconds	0.12711525947994232 seconds
Input Length = 8192	0.11535811216001093 seconds	0.22493702211997516 seconds	0.5261772978399313 seconds

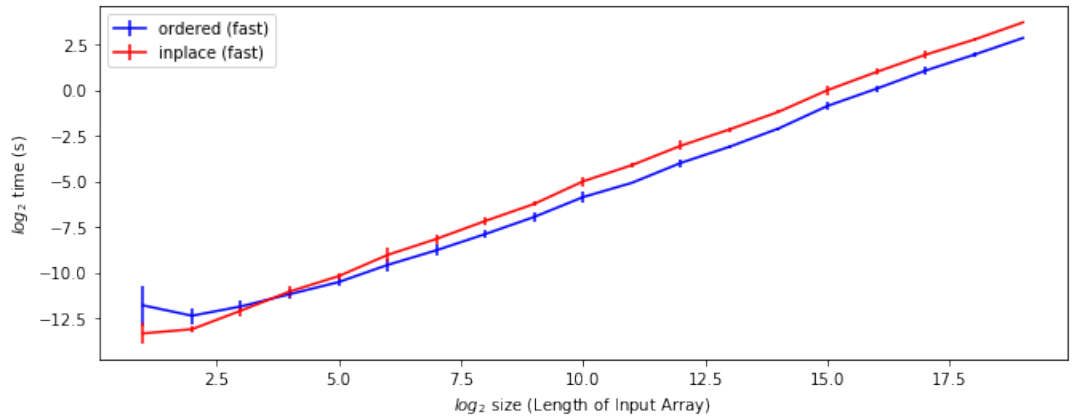
Figure 3: Timing of the 1D Forward Haar Wavelet Transform



	Ordered Haar Transform Runtime	In-Place Haar Transform Runtime	Matrix Haar Transform Runtime
Input Length = 2	0.00014060899999094546 seconds	7.722039998043329e-05 seconds	9.230171999661252e-05 seconds
Input Length = 4	8.928924002248095e-05 seconds	6.050283996955841e-05 seconds	4.831316004128894e-05 seconds
Input Length = 8	0.0001419129599162261 seconds	0.00010281603996190824 seconds	6.889496002258966e-05 seconds
Input Length = 16	0.00019638595995274954 seconds	0.00015062803995533614 seconds	6.60190799862903e-05 seconds
Input Length = 32	0.00017674336004347424 seconds	0.00015037748002214358 seconds	4.580803997669136e-05 seconds
Input Length = 64	0.0003359135999926366 seconds	0.00031809775999136036 seconds	6.932031996257138e-05 seconds
Input Length = 128	0.0006509488400115515 seconds	0.000595010400029423 2 seconds	0.00013995424002132494 seconds
Input Length = 256	0.0011364781199154094 seconds	0.0011454370399587787 seconds	0.000337664919989038 seconds
Input Length = 512	0.0018875324000509864 seconds	0.002111699840024812 seconds	0.0010221861999343673 seconds
Input Length = 1024	0.0036496306799926968 seconds	0.004208432400027959 seconds	0.004131753199999366 seconds
Input Length = 2048	0.007669552079951245 seconds	0.008812405000007857 seconds	0.016871948120042363 seconds
Input Length = 4096	0.015048339640034101 seconds	0.018215938080029446 seconds	0.06823519464001947 seconds
Input Length = 8192	0.03484643951995167 seconds	0.03992487983994579 seconds	0.28962511092002385 seconds

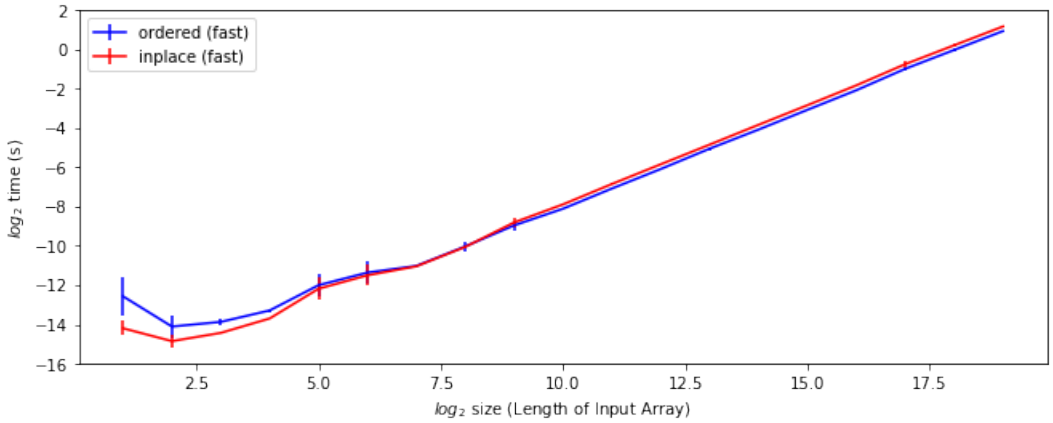
Figure 4: Timing of the 1D Inverse Haar Wavelet Transform

When doubling the input size, the fast transforms (both forward and inverse) take roughly twice as long to execute on average, which empirically confirms their linear-time performance. The matrix multiplication transform algorithm takes roughly four times as long when the input size is doubled, which experimentally shows its quadratic complexity.



	Ordered Haar Transform Runtime	In-Place Haar Transform Runtime
Input Length = 2	0.0002803390399822092 seconds	9.653807999711717e-05 seconds
Input Length = 4	0.0001889259600284276 6 seconds	0.00011320735999106546 seconds
Input Length = 8	0.0002687534000142477 4 seconds	0.00022899307998159202 seconds
Input Length = 16	0.0004280744800416869 seconds	0.00047464255998420414 seconds
Input Length = 32	0.0006776859999808949 seconds	0.0008490143599919975 seconds
Input Length = 64	0.0012984906399833563 seconds	0.0018981233600425184 seconds
Input Length = 128	0.002267770959988411 seconds	0.0034862881199660477 seconds
Input Length = 256	0.004194204159975925 seconds	0.006897529719917656 seconds
Input Length = 512	0.008034861519990954 seconds	0.013162284920017555 seconds
Input Length = 1024	0.017063507159964502 seconds	0.031041960879974795 seconds
Input Length = 2048	0.029352204719980363 seconds	0.057432553360122256 seconds
Input Length = 4096	0.06251376631995299 seconds	0.12000593711996771 seconds
Input Length = 8192	0.11573409844000708 seconds	0.22317200044002675 seconds
Input Length = 16384	0.23104704667995976 seconds	0.43499466980003487 seconds
Input Length = 32768	0.5435333653600355 seconds	0.9892970566400254 seconds
Input Length = 65536	1.0428921914399871 seconds	1.9685757369999737 seconds
Input Length = 131072	2.072119701319971 seconds	3.7853183630400054 seconds
Input Length = 262144	3.7970212727599937 seconds	6.727240836880046 seconds
Input Length = 524288	7.174207487080021 seconds	12.96213676256004 seconds

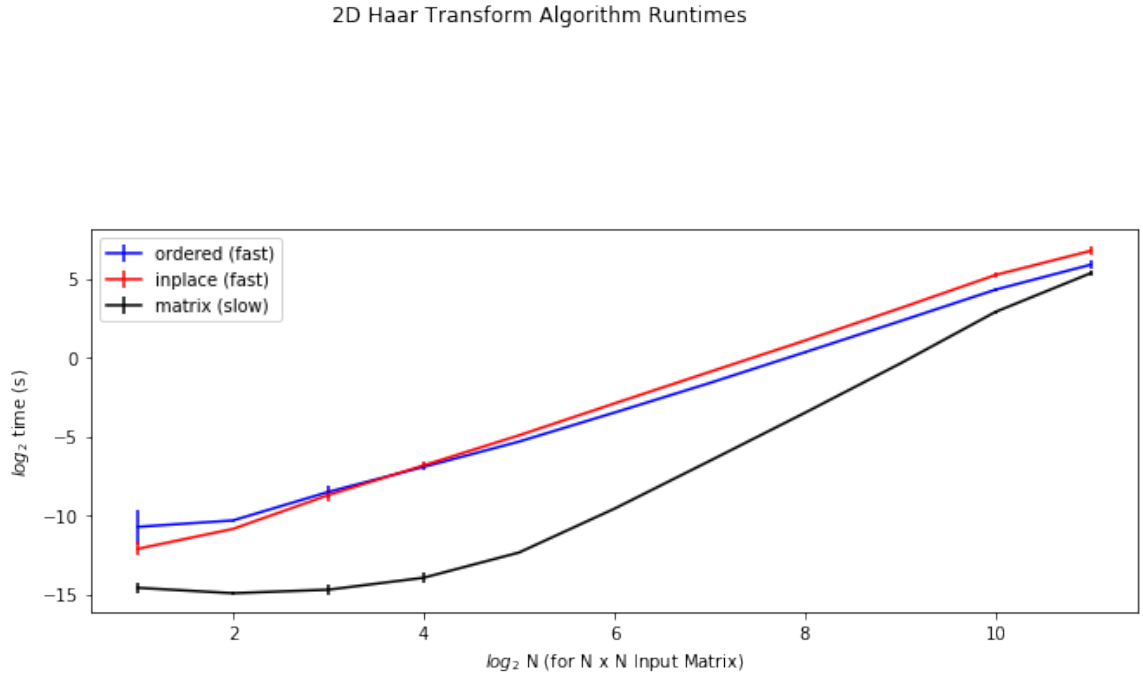
Figure 5: Timing of the Fast 1D Forward Haar Wavelet Transform



	Ordered Haar Transform Runtime	In-Place Haar Transform Runtime
Input Length = 2	0.0001642845600144937 6 seconds	5.32623999970383e-05 seconds
Input Length = 4	5.6633960084582214e-05 seconds	3.383459996257443e-05 seconds
Input Length = 8	6.671976003417512e-05 seconds	4.526296001131414e-05 seconds
Input Length = 16	9.986824003135552e-05 seconds	7.529451992013491e-05 seconds
Input Length = 32	0.00024354788009077312 seconds	0.00021452372006024233 seconds
Input Length = 64	0.00037942928003758424 seconds	0.000344541039885371 seconds
Input Length = 128	0.0004800959599378984 seconds	0.00047296296004788017 seconds
Input Length = 256	0.0009543203600333072 seconds	0.0009381509199738503 seconds
Input Length = 512	0.002001658439949 0966 seconds	0.002223220960004255 seconds
Input Length = 1024	0.0036027957199621596 seconds	0.004198309280036483 seconds
Input Length = 2048	0.007339204720010457 seconds	0.008612564360155374 seconds
Input Length = 4096	0.014614519279966771 seconds	0.017273601120032255 seconds
Input Length = 8192	0.02993987615984224 seconds	0.03455065896003361 seconds
Input Length = 16384	0.05869009775997256 seconds	0.06944415660003869 seconds
Input Length = 32768	0.11771236147989839 seconds	0.13849865375999798 seconds
Input Length = 65536	0.23472625392005284 seconds	0.2789331606799533 seconds
Input Length = 131072	0.49948891183983507 seconds	0.588723058000105 seconds
Input Length = 262144	0.969841905999928 7 seconds	1.1440722638800072 seconds
Input Length = 524288	1.8899592044799647 seconds	2.221835234479986 seconds

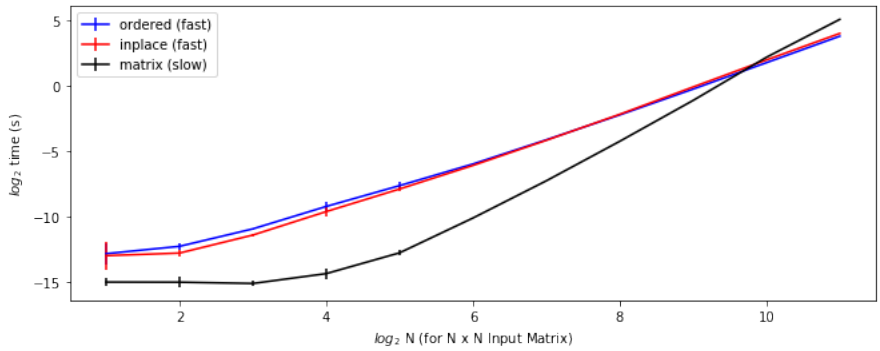
Figure 6: Timing of the Fast 1D Inverse Haar Wavelet Transform

Large input signals are not feasible with the matrix multiplication algorithm, because the Haar transform matrices become impractically large (several gigabytes, in fact). In order to further demonstrate the performance of the fast transforms, larger inputs were used. The linear complexity is still visible at these sizes, and the forward ordered algorithm is about twice as fast as the in-place algorithm. This is likely because the ordered algorithm relies on a list comprehension in a for loop, whereas the in-place algorithm uses a double nested for loop. Python in generally exhibits better performance with comprehensions than loops. Moreover, the inverse algorithms perform far better than the forward algorithms. This is because the forward algorithms must perform many floating-point divisions, while the only floating point operations required for the inverse algorithms are additions/subtractions.



	Ordered Haar Transform Runtime	In-Place Haar Transform Runtime	Matrix Haar Transform Runtime
Input Length = 2	0.0005906797000534425 seconds	0.00022484399987661164 seconds	4.0673800049262356e-05 seconds
Input Length = 4	0.0007825581998076814 seconds	0.0005394390998844755 seconds	3.207499994459795e-05 seconds
Input Length = 8	0.0027381454998248954 seconds	0.0023793461001332616 seconds	3.744679988813004e-05 seconds
Input Length = 16	0.008206627200161165 seconds	0.008794502000000648 seconds	6.325419999484439e-05 seconds
Input Length = 32	0.024891212199781874 seconds	0.03282567390006079 seconds	0.00019135469983666553 seconds
Input Length = 64	0.08885146810016523 seconds	0.13284982960003616 seconds	0.0013079457000458206 seconds
Input Length = 128	0.3259306134999861 seconds	0.5312987304998387 seconds	0.010665671099650354 seconds
Input Length = 256	1.2601621616000558 seconds	2.1035823079001603 seconds	0.08832514010000522 seconds
Input Length = 512	4.934814939600074 seconds	8.719905810399997 seconds	0.7688229667000996 seconds
Input Length = 1024	19.655235581000305 seconds	37.42572318030015 seconds	7.417439684299916 seconds
Input Length = 2048	58.944622316200004 seconds	108.30766497709992 seconds	40.65924634499997 seconds

Figure 7: Timing of the 2D Forward Haar Wavelet Transform



	Ordered Haar Transform Runtime	In-Place Haar Transform Runtime	Matrix Haar Transform Runtime
Input Length = 2	0.00013880670003345584 seconds	0.00012488870006563957 seconds	3.077770006711944e-05 seconds
Input Length = 4	0.0002048620999630657 seconds	0.00014333189983517514 seconds	3.059659993596142e-05 seconds
Input Length = 8	0.0005176892999770643 seconds	0.0003729893001036544 seconds	2.8567699973791606e-05 seconds
Input Length = 16	0.0016995822000353655 seconds	0.001293923499997618 seconds	4.8182899990933945e-05 seconds
Input Length = 32	0.005127349299891648 seconds	0.0042874056999608005 seconds	0.00014524179978252504 seconds
Input Length = 64	0.016155090799929894 seconds	0.014966355899923655 seconds	0.000919732800048223 seconds
Input Length = 128	0.0582279250999818 seconds	0.05675908619978145 seconds	0.006621342100061156 seconds
Input Length = 256	0.21783687769975585 seconds	0.2249762561999887 seconds	0.05373814329996094 seconds
Input Length = 512	0.8574051956000404 seconds	0.9595823066000776 seconds	0.46957083959996454 seconds
Input Length = 1024	3.482055859600041 seconds	4.028583801400055 seconds	4.63274106880026 seconds
Input Length = 2048	14.18517795680009 seconds	16.445310705099747 seconds	34.62487372400001 seconds

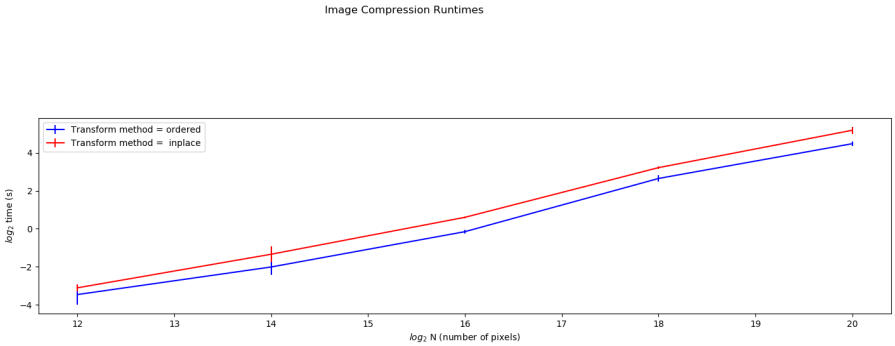
Figure 8: Timing of the 2D Inverse Haar Wavelet Transform

The 2D transform performance is interesting. Doubling the number of rows and columns in the array roughly quadruples the length of time it takes to calculate the transform when using either of the fast algorithms. This is indicative of quadratic time complexity with respect to side-length of the array. Moreover, the increase in running time for the matrix algorithm is roughly a factor of 8 as the matrix size quadruples, suggesting cubic time complexity for this algorithm.

Although the growth rates of the fast transforms are vastly better than that of the matrix algorithm, the fast transforms take longer to calculate than the matrix transform does for the array sizes tested here. This is because the leading coefficient for these algorithms is large. For small arrays, the matrix algorithm is the better choice for 2D transforms, including those used in image compression.

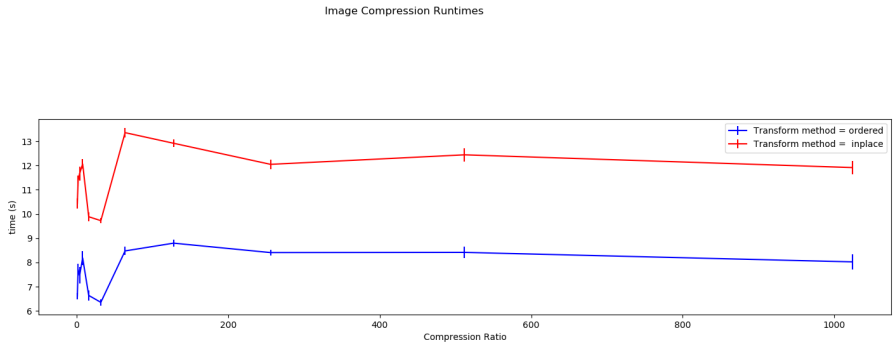
4.3 Image Compression

4.3.1 Compression Runtime



	Ordered Haar Transform Runtime	In-Place Haar Transform Runtime
Number of Pixels (N) = 4096	0.0905736873442939 seconds	0.11580318692008805 seconds
Number of Pixels (N) = 16384	0.24762175661760943 seconds	0.39421609844067323 seconds
Number of Pixels (N) = 65536	0.8982682997593594 seconds	1.5166788977951733 seconds
Number of Pixels (N) = 262144	6.267127277692128 seconds	9.301976512440667 seconds
Number of Pixels (N) = 1048576	22.277766652823676 seconds	36.37935632026973 seconds

Figure 9: Timing of the Compression Algorithm for images of different numbers of pixels



	Compression w/ Ordered Haar Transform Runtime	Compression w/ In-Place Haar Transform Runtime
Target Compression Ratio = 1	6.618562870844973 seconds	10.441187329688645 seconds
Target Compression Ratio = 2	7.708749978666637 seconds	11.374447352102909 seconds
Target Compression Ratio = 4	7.463185099641026 seconds	11.669200596196385 seconds
Target Compression Ratio = 8	8.184893873626857 seconds	12.063770145806462 seconds
Target Compression Ratio = 16	6.638968034087884 seconds	9.887490058652542 seconds
Target Compression Ratio = 32	6.354055045767188 seconds	9.725400754347708 seconds
Target Compression Ratio = 64	8.476177118118517 seconds	13.35859700422684 seconds
Target Compression Ratio = 128	8.796606686729524 seconds	12.920435257507938 seconds
Target Compression Ratio = 256	8.407492334211293 seconds	12.05002793248416 seconds
Target Compression Ratio = 512	8.41559793151705 seconds	12.443241021498398 seconds
Target Compression Ratio = 1024	8.024213026436337 seconds	11.91677320074416 seconds

Figure 10: Timing of the Compression Algorithm for varying target compression ratios

The compression algorithm requires a forward fast Wavelet transform, a threshold selection and an inverse fast Wavelet transform. Therefore, we expected the algorithm to grow linearly with the number of pixels in the image. The slope of the log-log plot appears to be linear, suggesting this to be the case.

Varying compression ratios appears to not impact the amount of time it takes for the algorithm to complete. This makes sense because it is implemented using the Quick Select algorithm on a flattened copy of the image, selecting a threshold value that would give a compression ratio that is close to the target compression ratio. This means the threshold is found with an

O(N) operation where N is the number of pixels.

4.3.2 Image Quality

For image quality we use a qualitative approach to look at how the image quality decreases as the compression ratio grows. If we were creating a compressed image format, this would need to be studied to find a compression ratio that gives an acceptable compression to image quality trade off.

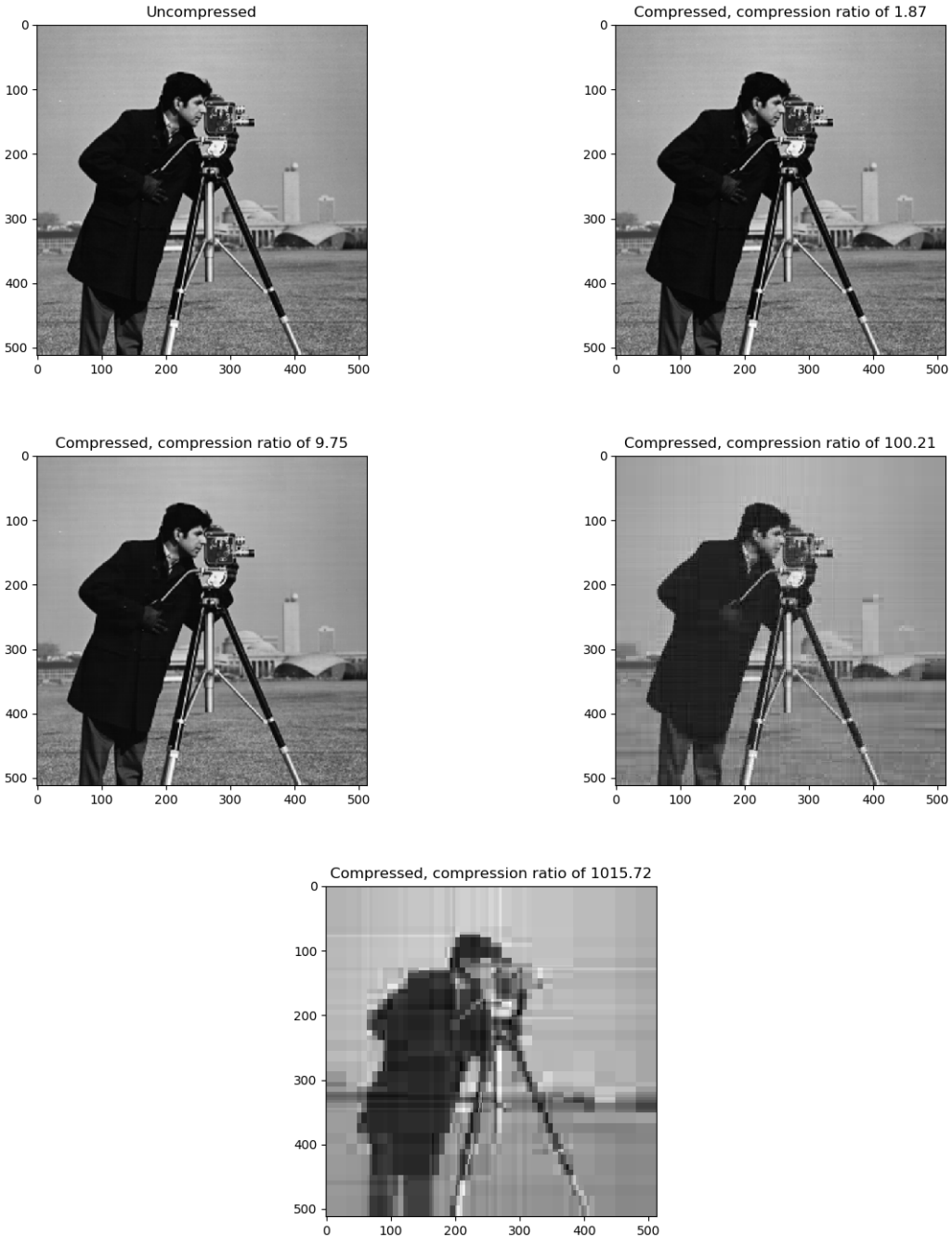


Figure 11: Reconstructed images after applying a threshold to the wavelet coefficients for a compression ratio

High compression ratios introduce blocky artifacts, as a result of the step-like nature of the Haar wavelet. These artifacts are less noticeable at modest compression ratios.

5 Conclusions

The performance metrics we observed in our experiments reinforced a number of concepts which were introduced in Data Structures and Algorithms. Namely, that growth rates should generally drive algorithm choices; although the naive implementation of the 1D Haar wavelet transform (via matrix multiplication) was simple and elegant, it does not scale nearly as well as the slightly more involved 1D fast transform algorithms. However, the pursuit of better growth rates is not always the best path forward. Fast 2D wavelet transforms have better growth rates than the matrix multiplication 2D wavelet transform, but the leading coefficients are so disparate that the naive implementation is actually faster for our application set.

With respect to image compression, we were able to implement an elegant image compression algorithm by building around our 2D wavelet transform. This algorithm scaled linearly with image size, and performed approximately O(1) with respect to compression ratios. With that said, this algorithm's lossy compression quality is not ideal. At high compression ratios, the compression introduces blocky artifacts into the compressed image. This is intuitive, given that the Haar wavelet is essentially a square pulse function scaled and shifted over time. Using just a few coefficients, the wavelet's step-like nature becomes apparent in the image. This is presumably why real-world image compression algorithms avoid the Haar wavelet, and instead opt for smoother wavelets. With more time, this work could be replicated on more sophisticated wavelets to observe their performance and compression quality. Moreover, an objective compression quality metric should be developed if one of us decides to pursue this work going forward.

References

[1] Grey Ballard, Austin R. Benson, Alex Druinsky, Benjamin Lipshitz, and Oded Schwartz. Improving the numerical stability of fast matrix multiplication algorithms. *CoRR*, abs/1507.00687, 2015.

[2] G. Beylkin, R. Coifman, and V. Rokhlin. Fast wavelet transforms and numerical algorithms i. *Communications on Pure and Applied Mathematics*, 44(2):141,183, 1991-03.

[3] D. Labate, G. Weiss, and E. Wilson. Wavelets. *Notices of the American Mathematical Society*, 60(1):66,76, 2013.

[4] S. Mallat. *A Wavelet Tour of Signal Processing*. Elsevier Inc., 2009.

[5] Yves. Nievergelt. *Wavelets Made Easy*. Modern Birkhauser classics. Springer New York, New York, NY, 2013.

[6] Anthony. Teolis. *Computational Signal Processing with Wavelets*. Modern Birkhäuser Classics. Springer International Publishing, Cham, 2017.

[7] Wikipedia contributors. Fast wavelet transform — Wikipedia, the free encyclopedia, 2018. [Online; accessed 22-March-2019].

[8] Wikipedia contributors. Haar wavelet — Wikipedia, the free encyclopedia, 2018. [Online; accessed 30-March-2019].