Linear Algebraic Methods in Image Compression

Simon Beyzerov[1] and Aryan Jain[1]

[1]Carnegie Mellon University
{sbeyzero,aryanj}@andrew.cmu.edu

Abstract

Data compression is a fundamental problem in information theory, motivated by the rapid transition to digital storage during the infancy of modern computing. At its core, data compression aims to reduce the resources necessary to represent and transmit a piece of data while maintaining its richness. To do so, compression algorithms take advantage of redundancies in the underlying data representation to extract and preserve varying amounts of structure. In this paper, we review various mathematical methods of extracting such structure from common representations. We consider digital images as a dense, highly redundant data medium and focus on algorithms that leverage linear-algebraic structure to achieve sizeable compression ratios. We discuss and motivate the need for effective image compression and go on to define the mathematical tools necessary for the algorithms we describe. Finally, we quantitatively and visually compare the performance of these various compression techniques on a sampled data set and discuss the advantages of these approaches.

1 Introduction

Digital data storage and transmission underlies nearly all modern techniques to persist and share information. Dating back to the 1970s, digital electronics have attempted to represent robust multi-media information *efficiently* through compression [1, 4, 15, 20, 21] (i.e., relay and store the essence of some data in a way that utilizes as least resources as possible). And although the prohibitively high algorithmic (and monetary) costs of storing and retrieving digital memory were critical issues in early systems, the issues of redundancy and speed continue to be central concerns [8, 6]. Storing information takes space, larger pieces of data take more time to process, and in the age of the internet, efficiency frequently takes precedence over precision.

Images are one example of modern-day, data-intensive media. With the rise of multimedia-based web applications, it is critical that bandwidth is properly allocated towards distributing as much media as possible, efficiently [6, 7]. Oftentimes, the less resources used to represent a single image, the more images can be consumed or stored by users. Algorithms that attempt to efficiently represent images reflect the challenges and patterns that have defined the progress of data compression. The foremost task of any such system: within the data, identify systematic redundancy and irrelevant information and either remove it or find ways to minimize its representation.

Following successful compression, there is an inherent trade-off between resources and interpretability. Most data compression incurs some unrecoverable information loss. Such data compression is called *lossy*, and while *lossless* schemes ensure no information is lost, the most significant compression ratios occur in lossy schemes. Images are a practical example of a medium where this loss in precision is acceptable. When viewing large quantities of images, or images that are small, it isn't always necessary to view them in full resolution. It is acceptable (in many applications) to incur some loss in quality.

In this paper, we describe various approaches in image compression and compare their advantages and disadvantages, including empirical observations about the trade-offs in data loss versus compression quality. In the following section, we discuss the mathematical background and approaches necessary to build these compression algorithms. Much of our focus will be on algorithms that are structured around linear algebra. We go on to describe these tools.

2 Background

In this section, we describe further background and the notation we use throughout the rest of this work.

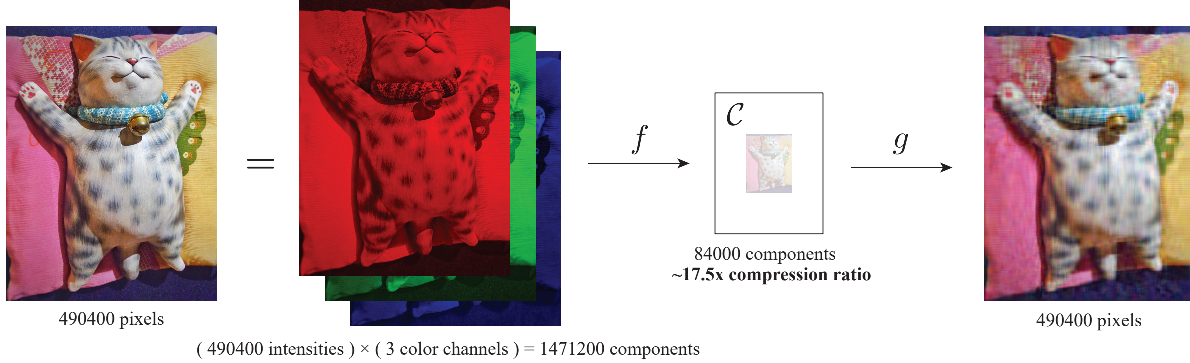


Figure 1: A visualization of our generalized compression model, where an image is split into multiple color channels and each is compressed individually using a compression function f. The compressed image lives in a compression space $\mathcal{C}$, where it occupies a significantly smaller intermediate representation. The compressed image may not be representable visually, but must be recoverable; at a later point, the compressed representation can be decompressed with a decompression function g. It is now representable in the same way as the original image.

Representing Images Since the entire visible spectrum cannot be practically represented by a physical display, pixels and images must be discretized in some capacity. To accommodate physical displays, digital images are commonly represented as a matrix or array of distinct "pixels," or singular units of light on a screen. For the human eye, this approximation of true physical color is perfectly acceptable and each individual pixel is almost imperceptible on a high-density display [16]. Each pixel can be thought of as a single value in $\mathbb{R}$, where image processing software defines some range of over the number of possible colors the pixel can take. In the RGB color system—the most common system for representing modern pixel colors [16]—this pixel value is further broken down into a tuple $(r, g, b) \in \mathbb{Z}_{256}^3$, with each r, g, and b representing the relative intensity of red, green, and blue respectively. While storing each pixel as a single value simplifies compression, we will see that a split into three *color channels* is useful for improving the fidelity of the compressed colors. For demonstrative purposes, we will primarily focus on one (gray-scale) or three (RGB) color channels in the images we compress. The techniques we describe are analogous in higher resolution and color channel image representations. Unless otherwise stated, each color channel is compressed independently before being combined as a single image at decompression time. We summarize this representation in Section 2.

Problem Definition Now that we are able to represent images as matrices, we can state a concrete formulation of what compression and decompression looks like. Define a *compression function* $f : (\mathbb{R}^{m\times n})^\ell \to \mathcal{C}$ that compresses an $m \times n$ image with ℓ color channels into a compression space $\mathcal{C}$. For our purposes, the compression space will often be another matrix that is then used for decompression into the original image. A *decompression function* $g : \mathcal{C} \to (\mathbb{R}^{m\times n})^\ell$ is one that converts from the compression space back into the original compressed image matrix. Finally, consider a norm $\|\cdot\| : (\mathbb{R}^{m\times n}) \to \mathbb{R}$ (we describe a set of norms in section 3 for our analyses). A compression scheme $\mathcal{F} := (f, g, \mathcal{C})$ is thus a pair of compression/decompression functions and a compression space $\mathcal{C}$, such that $\|g(f(X)) - X\| \approx 0$. In an ideal compression we desire $\|g(f(X)) - X\| = 0$, which typically occurs in lossless compression. We'll soon find that this is mostly unachievable in practical lossy compression schemes.

Compression Ratio The goal of compression is to reduce the size of the intermediate representation in $\mathcal{C}$ by exploiting redundancies in the original image matrix. Since defining a single, generalized unit of "information" is difficult and highly dependent on the compression scheme and hardware implementation of primitive types, we will defer to a somewhat nebulous notion for now. Since our surveyed compression schemes will mostly convert the image into an intermediate matrix or array representation, we define the *compression ratio* as the the ratio of non-zero elements in the original matrix to the non-zero elements in the compressed representation. This is motivated by the fact that repeated zero bits in arrays—sparse matrices—can be represented efficiently [2]. A compression ratio acts a good heuristic to the amount of space saved by our compression scheme. We will redefine this quantity as needed for any compression scheme that requires a different notion of the compression ratio.

2.1 SVD (rank approximation) Compression

Consider an initial image matrix $X \in (\mathbb{R}^{m\times n})^\ell$ parameterized as in section 2. Our first compression scheme demonstrates one approach of using linear algebra to isolate structure in the image matrix X.

Motivation & Intuition We start from the fact X can be expressed as a linear combinations of $\operatorname{rank}(X)$ basis vectors. It's not hard to see that the effective "size" of X (the number of real components needed to describe all the pixels) is thus proportional to the square of the image's rank. But the rank of X isn't always going to be low—with images reaching high resolution, the number of vertical or horizontal pixels may be in the thousands. In fact, we have no guarantee (and it is unlikely) that $\operatorname{rank}(X) \ll \min\{m, n\}$. i.e., we can rarely represent complex images with significantly fewer basis vectors than the full rank. One key insight we can make is that under a certain choice of basis, some basis vectors may be over or under-represented in the decomposition of X. In other words, some bases are less prominent than others and may be "ignored." If we express X with only the few most highly represented basis vectors, we can presumably achieve a good approximation of the image while only storing a few basis vectors. Singular Value Decomposition (theorem 2.1) gives us a simple approach for applying this intuition towards a compression procedure.

Theorem 2.1 (Singular Value Decomposition (over $\mathbb{R}$) [10])**.** *Let $A \in \mathbb{R}^{m\times n}$ with rank r. There exists $\Sigma \in \mathbb{R}^{m\times n}$ for which the main diagonal entries are the first r singular values of A, $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r \geq 0$, and there exists an orthogonal matrix $U \in \mathbb{R}^{m\times m}$ and orthogonal matrix $V \in \mathbb{R}^{n\times n}$ such that $A = U\Sigma V^T$.*

Definition 2.1 (SVD_DECOMPOSITION)**.** *Let the function* $\text{SVD_DECOMPOSITION} : \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times m} \times \mathbb{R}^{m\times n} \times \mathbb{R}^{n\times n}$ *return a Singular Value Decomposition of a given real-valued matrix A such that* $\text{SVD_DECOMPOSITION}(A) := (U, \Sigma, V)$ *where the values correspond to those defined in theorem 2.1.*

Compression Compressing an image with SVD is as straightforward as performing a single SVD over a matrix in $\mathbb{R}$. We point to the literature for the most efficient SVD algorithms [12] and assume a black-box instantiation $\text{SVD_DECOMPOSITION}(\cdot)$ as outlined in $definition$ 2.1. Compressing an image X with SVD requires taking its rank k approximation. The parameter k is provided to the compression scheme to indicate the "strength" of the compression: a high rank k will mean more basis vectors are used to represent the image, increasing the size of the intermediate representation but also increasing the amount of information retained when the image is decompressed. If quality is not as critical, k may be reduced to further decrease the size of the compressed image at the cost of the decompressed image's quality. We describe the SVD image compression algorithm in Algorithm 1.

Space Saved & Decompression The compression space presented in Algorithm 1 for compression strength k is $\mathcal{C} = \mathbb{R}^{m\times k} \times \mathbb{R}^{k\times k} \times \mathbb{R}^{n\times k}$. A natural question now is: how much space does SVD compression actually save? Recall that X can be stored naively under its matrix representation in $O(mn)$, where the value of each cell is stored explicitly. We discussed in section 2.1 that finding a minimal basis for X is enough to store X in $O((\min\{m, n\})^2)$ space. Following SVD compression, we are left with U_k, Σ_k, and V_k as our compressed representation, which accounts for $O(mk + k^2 + nk) \subset O(m + n)$ stored values—a significant reduction in size when $k \ll \operatorname{rank}(X)$. To *decompress* from this intermediate representation, it is enough to just multiply the matrices $U_k\Sigma_k V_k^T$.

Algorithm 1 SVD Image Compression (single channel; $\ell = 1$)

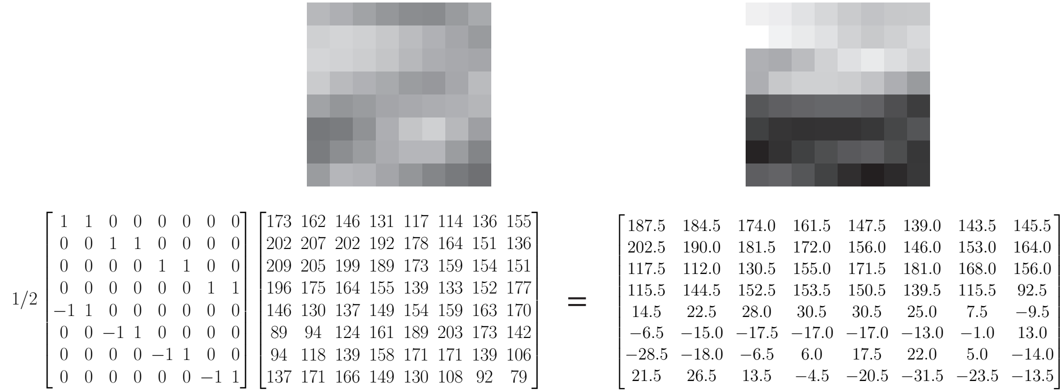
Require: $X \in (\mathbb{R}^{m\times n})^{\ell=1}$ is an image matrix, $m, n, \ell \in \mathbb{N}$ are image parameters according to section 2, and $0 < k \leq \operatorname{rank}(X)$ is the rank approximation.
Ensure: $U_k \in \mathbb{R}^{m\times k}, \Sigma_k \in \mathbb{R}^{k\times k}, V_k \in \mathbb{R}^{n\times k}$ and $X \approx U_k\Sigma_k V_k^T$

1: $(U, \Sigma, V) \leftarrow \text{SVD_DECOMPOSITION}(X)$
2: $(U_k)_{ij} \leftarrow U_{ij} \quad \forall i \in [1, m], j \in [1, k]$
3: $(\Sigma_k)_{ij} \leftarrow \Sigma_{ij} \quad \forall i \in [1, k], j \in [1, k]$
4: $(V_k)_{ij} \leftarrow V_{ij} \quad \forall i \in [1, n], j \in [1, k]$
5: **return** (U_k, Σ_k, V_k)

2.2 Haar Wavelet Transformation

Like SVD and other linear algebraic compression techniques, the Haar Wavelet Transformation (HWT) aims to find an alternate, orthogonal basis for the high-dimensional matrix representing the image, and then remove bases that do not greatly contribute to the final image. In this section we describe the HWT and the linear algebra involved in creating a compression algorithm with it.

Intuition & Motivation The Haar Wavelet Transformation (HWT) comes from a rich area of physics, engineering, and mathematical research called wavelet analysis. Typically adopted in signal analysis and approximation, the Haar wavelet in its most common form aims to take a one-dimensional discrete "signal" (can be thought of as a discrete sequence of values) and decompose it into a set of simpler orthonormal base signals. The central observation of the HWT is that signal values tend to *gradually* fluctuate along a spectrum of intensity. In an image, this means that adjacent pixels will tend to be of very similar intensity, with one pixel being slightly more or less intense than the other. To demonstrate how this could be useful, consider the following alternate way of representing pairs of pixels: Instead of storing the two different pixel values, store the *mean* of their values (a large value), as well as their respective difference from the mean (a small value). While the amount of values being used to represent the pixels is still two, the HWT makes the following observation: the smaller difference value is much easier to *quantize*, or reduce to 0 at a certain threshold. Doing so, we can remove small, redundant differences in pixel intensities.



$$1/2 \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 173 & 162 & 146 & 131 & 117 & 114 & 136 & 155 \\ 202 & 207 & 202 & 192 & 178 & 164 & 151 & 136 \\ 209 & 205 & 199 & 189 & 173 & 159 & 154 & 151 \\ 196 & 175 & 164 & 155 & 139 & 133 & 152 & 177 \\ 146 & 130 & 137 & 149 & 154 & 159 & 163 & 170 \\ 89 & 94 & 124 & 161 & 189 & 203 & 173 & 142 \\ 94 & 118 & 139 & 158 & 171 & 171 & 139 & 106 \\ 137 & 171 & 166 & 149 & 130 & 108 & 92 & 79 \end{bmatrix} = \begin{bmatrix} 187.5 & 184.5 & 174.0 & 161.5 & 147.5 & 139.0 & 143.5 & 145.5 \\ 202.5 & 190.0 & 181.5 & 172.0 & 156.0 & 146.0 & 153.0 & 164.0 \\ 117.5 & 112.0 & 130.5 & 155.0 & 171.5 & 181.0 & 168.0 & 156.0 \\ 115.5 & 144.5 & 152.5 & 153.5 & 150.5 & 139.5 & 115.5 & 92.5 \\ 14.5 & 22.5 & 28.0 & 30.5 & 30.5 & 25.0 & 7.5 & -9.5 \\ -6.5 & -15.0 & -17.5 & -17.0 & -17.0 & -13.0 & -1.0 & 13.0 \\ -28.5 & -18.0 & -6.5 & 6.0 & 17.5 & 22.0 & 5.0 & -14.0 \\ 21.5 & 26.5 & 13.5 & -4.5 & -20.5 & -31.5 & -23.5 & -13.5 \end{bmatrix}$$

Figure 2: Illustrative example of a single-round application of the Haar Wavelet Transformation onto the columns of a two-dimensional image (one channel; grayscale). Observe that each pixel is represented with an intensity from 0 to 255. The HWT is applied in one dimension. Observe that the upper half of the transformed image contains "large" values, while the lower half contains "detail coefficients"—smaller values used for reconstructing the image. When the image is quantized, these small values may be rounded to 0 to reduce the image size.

Definition 2.2 (One-Dimensional Haar Wavelet Transform)**.** *Define the One-dimensional Haar Wavelet Transform over a vector $\vec{v} \in \mathbb{R}^{2^k}$ for a $k \in \mathbb{N}$ as a function $\text{HWT} : \mathbb{R}^{2^k} \to \mathbb{R}^{2^k}$, denoted as $\text{HWT}(\vec{v})$.*

If $\vec{v} = (v_1, v_2, \cdots, v_{2^k-1}, v_{2^k})$,

$$\text{HWT}(\vec{v}) := \left(\frac{v_1 + v_2}{2}, \frac{v_3 + v_4}{2}, \cdots, \frac{v_{2^k-1} + v_{2^k}}{2}, \frac{v_2 - v_1}{2}, \frac{v_4 - v_3}{2}, \cdots, \frac{v_{2^k} - v_{2^k-1}}{2} \right).$$

This is equivalent to the matrix product

$$\text{HWT}(\vec{v}) = \frac{1}{2} W_{2^k} \vec{v} = \frac{1}{2} \begin{bmatrix} 1 & 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & 1 & \cdots & 0 & 0 \\ & \vdots & & & \ddots & 0 & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 & 1 \\ -1 & 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & -1 & 1 & \cdots & 0 & 0 \\ & \vdots & & & \ddots & 0 & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -1 & 1 \end{bmatrix} \vec{v}.$$

Compression While the HWT is intuitive on a single pair of pixels, it is important to understand how it extends to an array of many pixels and even a two-dimensional image. From definition 2.2, we can see that there is some clear linear-algebraic structure in the way we perform the pairwise averaging of each array pair. In fact, the matrix in question has orthogonal columns and can easily be inverted, as described in lemma 2.2. In figure 2 we see how the HWT transforms a single vector into a set of values that can be easily quantized. From here, we can define a threshold $\varepsilon > 0$ such that all values of the compressed array less than ε are rounded to 0. When we reconstruct the array, we see that the values stay mostly the same. From here, we can easily apply the same procedure to each individual row in the image matrix X, transforming them one-by-one. But we can do better. Once we transform each row, we can then transform each resulting (transformed) column. Definition 2.3 illustrates that this operation does indeed have interesting matrix structure. Once again, each small value can be quantized according to some desired compression criteria. Figure 3 illustrates this process of compressing a two-dimensional image with HWT.

Lemma 2.2. *For $k \in \mathbb{N}$, the columns of the compression matrix $\frac{1}{2}W_{2^k}$ are all pairwise orthogonal and thus $\frac{\sqrt{2}}{2}W_{2^k}$ is orthonormal. Its inverse is $\frac{\sqrt{2}}{2}W_{2^k}^T$. The inverse of $\frac{1}{2}W_{2^k}$ is thus $W_{2^k}^T$.*

Definition 2.3 (Two-Dimensional HWT)**.** *Define the Two-Dimensional Haar Wavelet Transform over a matrix $X \in \mathbb{R}^{2^k \times 2^k}$ for $k \in \mathbb{N}$ as a function $\text{2D_HWT} : \mathbb{R}^{2^k \times 2^k} \to \mathbb{R}^{2^k \times 2^k}$, denoted as $\text{2D_HWT}(X)$:*

$$\text{2D_HWT}(X) := \frac{1}{4} W_{2^k} X W_{2^k}^T$$

where W_n is defined in definition 2.2 and the transformation corresponds to applying $\text{HWT}(\cdot)$ onto the columns of X and then applying $\text{HWT}(\cdot)$ onto the rows of the resulting matrix (or vice versa).

Lemma 2.3. *Let $C = \text{2D_HWT}(X)$ for an image matrix $X \in \mathbb{R}^{2^k \times 2^k}$ for $k \in \mathbb{N}$. Reconstructing X from its a single round of compression can be achieved by computing $X = W_{2^k}^T C W_{2^k}$. Decompressing a fully compressed HWT image is done recursively by applying this matrix to the smallest upper-right quadrant repeatedly, just as is shown in algorithm 2.*

Optimization & Decompression We present our final HWT compression procedure in algorithm 2. Observe from 3 that a single two-dimensional transformation with the HWT is equivalent to a "blurring" of the original image into the upper left corner of the compressed image—the remaining values are close or equal to 0 and are less effective to compress. We repeatedly compress the upper left corner of the matrix with the same two-dimensional HWT and then perform quantization on the resulting matrix. To decompress the image we can easily compute a product of each applied matrix transformation's inverse, which can be used to easily invert the entire procedure. Since the transformation matrices do not depend on the actual image pixel intensities, there is no extra storage needed to store the decompression matrix (i.e., all 100×150 images are decompressed with the same decompression matrix, just as all $m \times n$ matrices share the same decompression matrix). In lemma 2.3 we describe what the entire decompression matrix looks like for an image of arbitrary dimension[1]

[1] Note that throughout this section we maintain the invariant that image dimensions are square powers of two. This assists in clarifying the recursive HWT compression procedure, but can easily be generalized by simply performing all operations on a power of two matrix that entirely fits the original image.

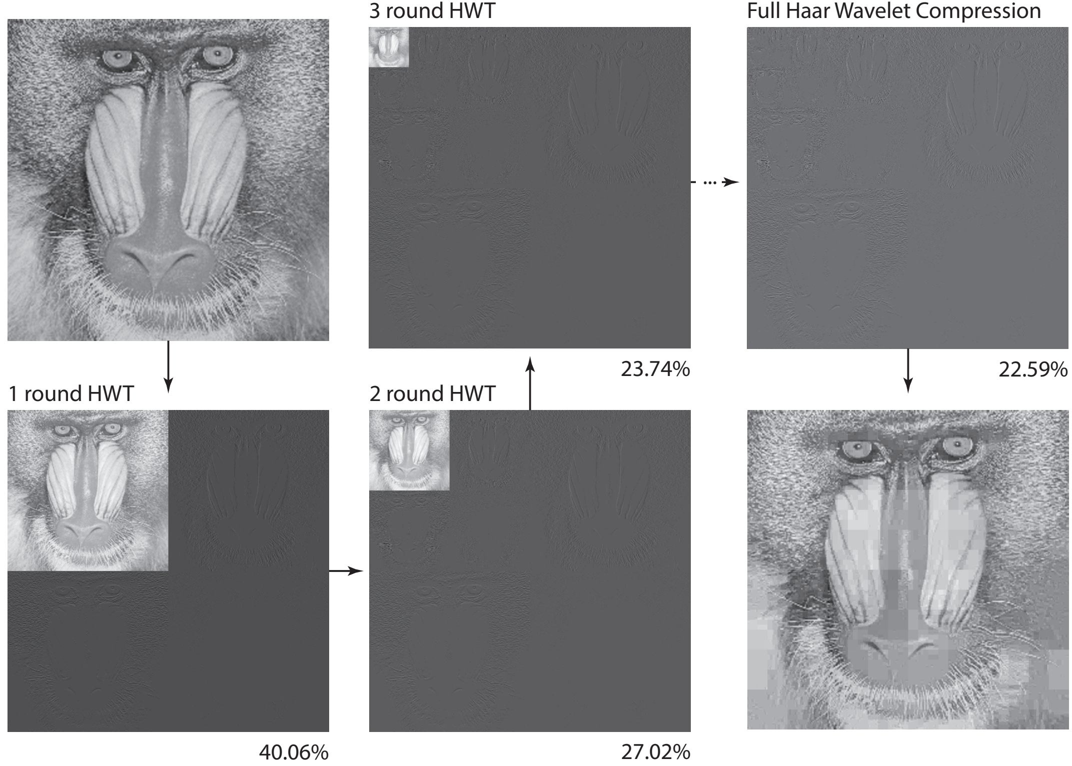


Figure 3: An illustrative example of the two-dimensional Haar Wavelet Transformation on an image, performed in multiple rounds, along with the ratio of compression achieved by quantizing the image at that round (lower right corner of each image). The compression ratio approaches the final full compression ratio, and the image can be decompressed at any step. Observe that at each round, the upper right corner of the image is compressed with respect to its columns and rows, "blurring" the image until it is one pixel in size. The full compression algorithm performs $O(\log n)$ rounds, where n is the width/height of the image. (The compression above is performed on a 512×512 grayscale image, pixels initially normalized between 0 and 1, and $\varepsilon = 0.01$).

Saved Space The compression space in the HWT is the same as the space of the original image, $\mathcal{C} = \mathbb{R}^{2^k \times 2^k}$ for $k \in \mathbb{N}$. As noted in *section 2*, we can measure the amount of space used not by the dimension of the compressed matrix, but by the number of non-zero values in that matrix. The original matrix will tend to have few pixels that are zeros, the matrix size is a good heuristic of its represented size. Following a recursive HWT transformation, quantization will round many of of the values of the compressed matrix to 0. This heavily reduces the amount of space needed to represent the compressed matrix. In section 3 we present empirical results on precisely how much space is saved under varying choices of ε, our quantization factor.

Definition 2.4 (Quantization). *Define* $\textsc{Quantize} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ *and* $\textsc{Quantize_Matrix} : \mathbb{R}^{m \times n} \times \mathbb{R} \to \mathbb{R}$ *for* $m, n \in \mathbb{N}$ *as*

$$\textsc{Quantize}(x, \varepsilon) := \begin{cases} 0 & , |x| < \varepsilon \\ x & , |x| \geq \varepsilon \end{cases}$$

and

$$\textsc{Quantize_Matrix}(X, \varepsilon) := \begin{pmatrix} \textsc{Quantize}(X_{11}, \varepsilon) & \cdots & \textsc{Quantize}(X_{1n}, \varepsilon) \\ \vdots & \ddots & \vdots \\ \textsc{Quantize}(X_{m1}, \varepsilon) & \cdots & \textsc{Quantize}(X_{mn}, \varepsilon) \end{pmatrix}.$$

Algorithm 2 HWT Image Compression (single channel; $\ell = 1$)

Require: $X \in (\mathbb{R}^{2^k \times 2^k})^{\ell=1}$ is an image matrix, $m, n, p, \ell \in \mathbb{N}$ are image parameters according to section 2, and $0 < \varepsilon$ is a quantization threshold.
Ensure: $C \in \mathbb{R}^{2^k \times 2^k}$ is a compressed, quantized image

```
1:  function HWT_COMPRESS(X, k, ε)
2:      if k = 0 then
3:          return X
4:      end if
5:      X' ← 2D_HWT(X)                    ▷ X' = (x1 | x2 / x3 | x4) with x_{1,2,3,4} ∈ R^{2^{k-1}×2^{j-1}}
6:      c1 ← HWT_COMPRESS(x1, k − 1, ε)
7:      ci ← QUANTIZE_MATRIX(ci, ε)   ∀i ∈ {2, 3, 4}
8:      C ← (c1 | c2 / c3 | c4)   ▷ c1 is the compressed upper-left quadrant; c_{2,3,4} are remaining (quantized) quadrants
9:      return C
10: end function
```

1: **function** HWT_COMPRESS(X, k, ε)
2: **if** $k = 0$ **then**
3: **return** X
4: **end if**
5: $X' \leftarrow$ 2D_HWT(X) ▷ $X' = \left(\begin{array}{c|c} x_1 & x_2 \\ \hline x_3 & x_4 \end{array}\right)$ with $x_{1,2,3,4} \in \mathbb{R}^{2^{k-1} \times 2^{j-1}}$
6: $c_1 \leftarrow$ HWT_COMPRESS$(x_1, k-1, \varepsilon)$
7: $c_i \leftarrow$ QUANTIZE_MATRIX$(c_i, \varepsilon) \quad \forall i \in \{2, 3, 4\}$
8: $C \leftarrow \left(\begin{array}{c|c} c_1 & c_2 \\ \hline c_3 & c_4 \end{array}\right)$ ▷ c_1 is the compressed upper-left quadrant; $c_{2,3,4}$ are remaining (quantized) quadrants
9: **return** C
10: **end function**

2.3 Discrete Cosine Transform

Notice that the recursive HWT compression algorithm we describe is essentially equivalent to a single matrix product, followed by quantization. Similar to the HWT, the Discrete Cosine Transform can provide a matrix-based compression algorithm with relatively lightweight mathematical machinery. Its popularization in the JPEG compression algorithm has made it a powerful, efficient tool in modern compression schemes. We point to the literature for a deeper dive into the Discrete Cosine Transform (DCT), but describe its general intuition and structure here. We then compare the performance of the DCT to the previously described compression algorithms.

Much like the HWT, the DCT performs matrix multiplication in square blocks, but unlike the HWT, performs these in fixed sizes instead of half-blocks. The Discrete Cosine Transform uses cosine functions of varying frequencies to construct a basis for the image's pixels. It does this and follows it with quantization, adopting a similar approach to the HWT. Inversion, like HWT, follows from a simple matrix product (as the compression matrices are orthonormal and easy to invert). There have been many iterations of the DCT with different bases. In definition 2.5 we show the DCT transformation matrix to provide better intuition to the compression step, though the spirit of the algorithm matches closely to the HWT, albeit under a cosine wave basis instead of a wavelet basis. Optimized algorithms exist to calculate the DCT without a matrix transformation, but are typically utilized in special cases of compression and under set parameters.

Definition 2.5 (Discrete Cosine Transformation Matrix)**.** *An $m \times m$ Discrete Cosine Transform's transformation matrix is given by,*

$$T_{ij} = \begin{cases} \frac{1}{\sqrt{m}}, & i = 1, 1 \leq j \leq m \\ \sqrt{\frac{2}{m}} \cos\left(\frac{(2j-1)(i-1)}{2m}\pi\right), & 2 \leq i \leq m, 1 \leq j \leq m \end{cases}.$$

Lemma 2.4. *The DCT transformation matrix described in definition 2.5 is orthonormal.*

3 Comparing Compression Techniques

Now that we have defined image compression using SVD/low rank approximation, the Haar Wavelet Transform, and Discrete Cosine Transforms, we compare their empirical performance. We implement these compression algorithms in Julia and include our code in appendix A. Through implementation, we seek to evaluate the practicality, data loss, compression ratio, and best practices for each compression algorithm. To provide a baseline for each compression algorithm, we will start off by compressing the test image shown in figure 4.

Figure 4: The Portland Head Light, a famous lighthouse located in Cape Elizabeth. This image is 512×768 pixels and contains 3 RGB color channels.

3.0.1 SVD & Rank Approximation:

To achieve SVD compression, we split the image into 3 color channels (red, green, and blue), with each channel being a 2d array of normalized pixel values (between 0 and 1) corresponding to the intensity of the respective channel color. Next, we take Singular Value Decomposition (SVD) of every channel independently. Finally, we vary the rank approximation parameter k (as described in algorithm 1) from 0 to 512 (the full dimension of the image). This produces an approximation of our original image to varying degrees, depending on the value of k provided. As a last step, we combine every approximated channel into a final image (using colorview in Julia). For our test image, we see the following result for the rank 200, 150, 100, 50, and 10 approximations, respectively, reported in figure 5.

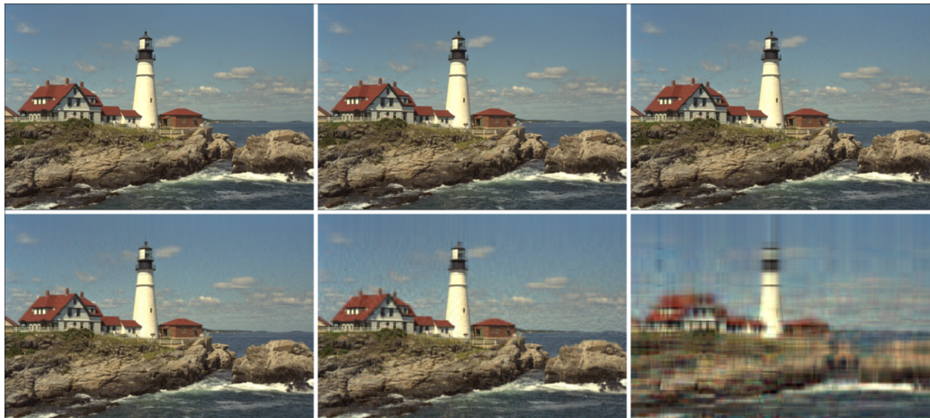


Figure 5: From left to right: original image and rank 200, 150, 100, 50, and 10 approximations.

3.0.2 Haar Wavelet Transformation:

To achieve image compression with the Haar Wavelet Transformation, we perform a slightly optimized version of algorithm 2. First, we split the image into 3 channels (red, green, and blue), with each channel being a 2d array of normalized pixel values (between 0 and 1) corresponding to the intensity of the channel color, just as in our SVD compression implementation. Next, using the Wavelets.jl and WaveletsExt.jl library implementations, we can iterate through every single channel and transform each matrix using the HWT function, `wpt(channel,wavelet(WT.haar))`.

In figure 6, we see the compression results of HWT over three color channels. Unlike our example in figure 3, we combine all three (fully compressed) color channels, and the compression is modified for images that don't necessarily have dimension of 2^k.

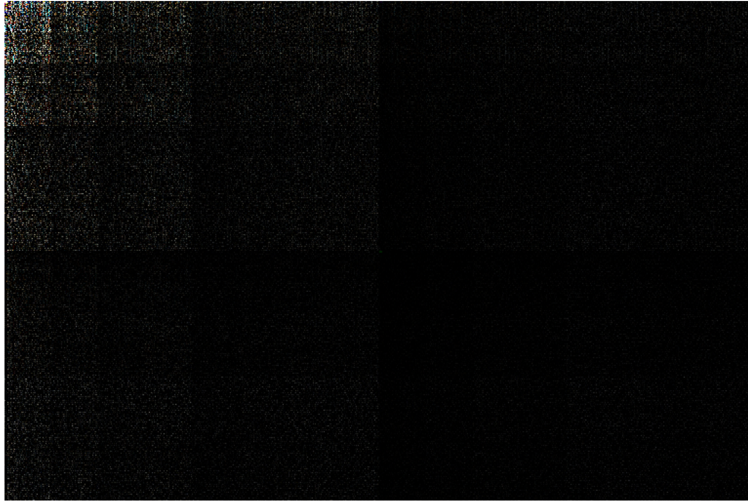


Figure 6: A non-square test image with the Haar Wavelet Transform applied over three color channels (RGB).

Here, we again recall that storage for the image can be significantly reduced by replacing many of the insignificant values near the bottom and right of the transformed image with a 0, thus building a sparse matrix, which can be efficiently represented. We set a variety of quantization vaues ε (also called thresholds; as described in algorithm 2) to test this as a valid compression algorithm (for exhaustive analysis, we test around 100 thresholds between 0 and 1; the quantization threshold can be any real value, but for the normalized pixel intensities we are dealing with, a maximal value of 1 suffices, as seein in figure 9). We seek to zero out every value in the matrix that is at or below each threshold value. To threshold an image, we once again iterate through every channel and call the `threshold!` function on the matrix with the `HardTH()` specification. After thresholding, we can apply the inverse Haar Wavelet transformation on each channel using `iwpt(haar_channel, wavelet(WT.haar))` and reconstruct the image through colorview. In figure 7 we show these reconstructions to illustrate how varying thresholds change the reconstructed image quality.

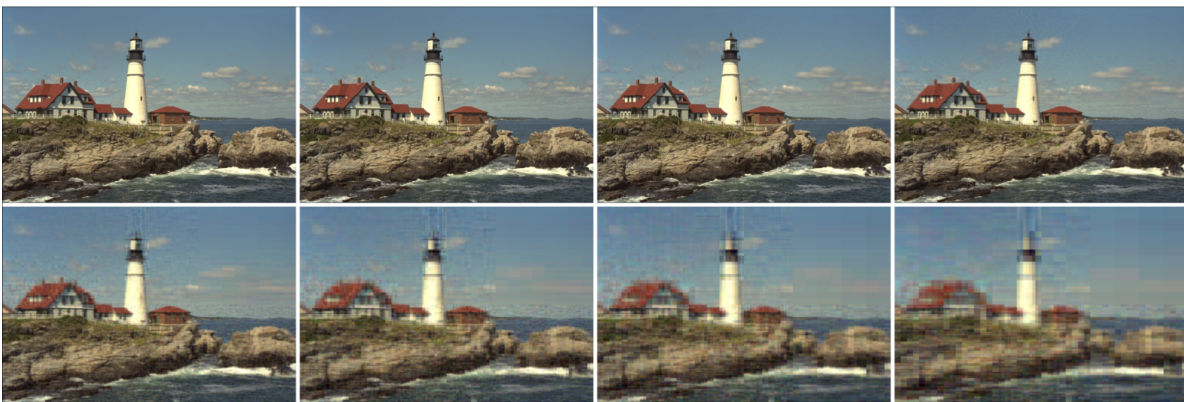


Figure 7: From left to right: original image, then images compressed with thresholds of $\varepsilon = 0.01, 0.05, 0.1, 0.3, 0.5, 0.7$, and 1 using the Haar Wavelet Transform

3.0.3 Discrete Cosine Transform

As the Discrete Cosine Transform functions similar to the Haar Wavelet transform, it has nearly an identical implementation in code, just utilizing a different orthonormal basis. We use the DCT functions from the FFTW.jl library. In short,

the process is that we transform every channel using the `dct(channel)` method (we do not need to specify a dimension to apply the transform along, as we already separate the image into individual channels), then threshold every channel the same way as with Haar Wavelet, and then finally apply the inverse DCT function, `idct(new_channel)` on every channel of an image. One important note is that when taking the inverse DCT of each channel, we must also take the absolute value of the entire resulting channel to ensure that the matrix values are valid normalized pixel values between 0 and 1. The resulting images from the DCT process are shown in figure 8.

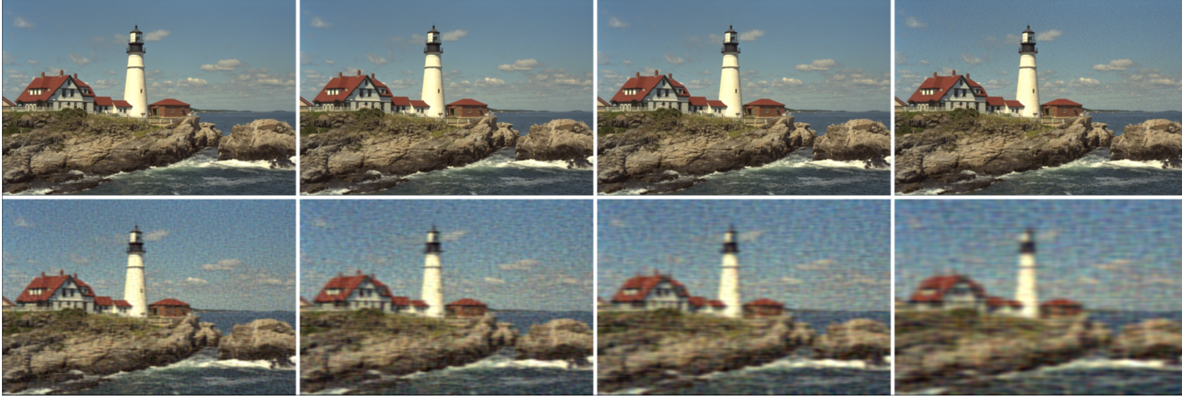


Figure 8: From left to right: original image, then images compressed with thresholds of $\varepsilon = 0.01, 0.05, 0.1, 0.3, 0.5, 0.7$, and 1 under the Discrete Cosine Transformation.

3.0.4 Measuring Compression Ratio and Image Quality

Now that we have implemented each compression algorithm, we want to evaluate, with practical data, the trade-offs in image quality and size as we vary the parameters of each compression algorithm. Concretely, this means considering the change in image quality with respect to some norm (as described in section 2) and the actual compression ratio achieved under a specific parameterization of the algorithm.

Compression Ratio for SVD/Rank Approximation To evaluate the compression ratio that iterations of the rank approximation algorithm achieved, we proceed as follows. To begin, we count the total number of pixel values, across all three channels, that we have to store for the original image. Next, for measuring how much data we have to store for a compressed image with rank approximation k, we add the combined length of the first k columns of U for each channel with k singular values for each channel and the combined length of the first k columns of V^T for each channel. Finally, the compression ratio is simply this number of entries divided by the entries needed for the original image. This is in line with our image representation in section 2, as we make no assumptions about the sparisty of the resulting matrix. Recall that SVD aims to save space in the number of vectors representing the image and not the matrix's sparsity (as in HWT and DCT).

Compression Ratio for Haar Wavelet and DCT To evaluate the compression ratio that HWT and DCT achieved, we proceed as follows. To begin, we count the total number of nonzero pixel values, across all three channels (with the transformation applied), that we have to store for the original image. Next, for measuring how much data we have to store for a compressed image, we count the number of nonzero entries in each transformed and thresholded channel. The idea is that the higher the threshold, the more zero entries exist in a matrix. Thus the matrix would be more sparse, and less space would be needed to store it (lower compression ratio). Finally, the compression ratio is simply this number of entries divided by the entries needed for the original image. Recall this is precisely the formulation of compression ratio we describe in section 2.

Image Quality Metrics - SSIM, PSNR, and Frobenius Norm In order to measure image quality differences between a compressed image and the original image, we will use three metrics: SSIM (Structural Similarity Index), PSNR (Peak Signal-to-Noise Ratio), and the Frobenius Norm. All of these metrics will be used in the same way we

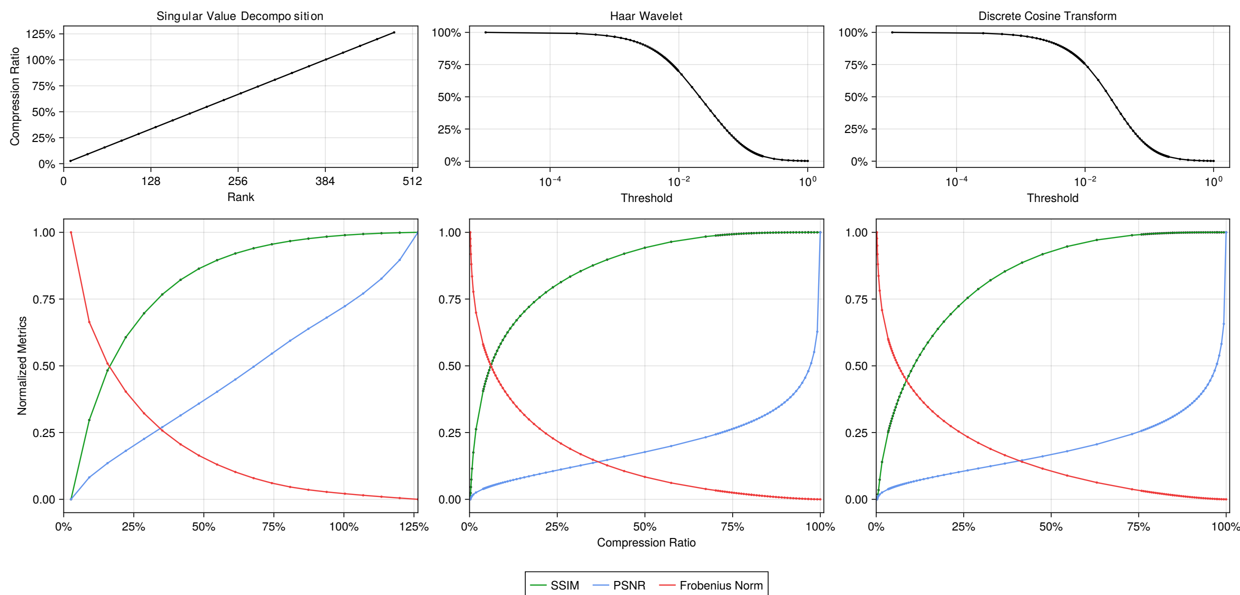


Figure 9: Image compression performance under varied parameter selection on the lighthouse image (figure 4). (Top) Influence of parameterization of each respective compression algorithm on the compression ratio. (Bottom) Influence of compressiion ratio on the overall quality of the compressed image based on SSIM, PSNR, as well as the Forbenius norm (each normalized).

describe norms in section 2: to quantitatively describe the quality of our compression. i.e., to understand whether compression was successful. SSIM is a method for predicting the perceived quality of images by mimicking the human visual perception system, which can identify structural information from a scene. It measures three parts of an image: luminance, contrast, and structure, and combines them to produce a quality score. SSIM has been widely adopted in the literature and has been shown to be a superior objective image quality metric [14]. Peak Signal-to-Noise Ratio (PSNR) is a widely used metric that quantifies the quality of reconstructed images subject to lossy compression. It measures the ratio of the peak signal power to the power of the noise that affects the quality of the signal. Finally, we use the Frobenius Norm as a simplified baseline metric. The Frobenius norm of a matrix is defined as the sum of the squares of all elements of the matrix, or in our case the matrix representing the difference between an original image and compressed image. It is commonly used to measure image noise. For measuring SSIM and PSNR, we will be using the ImageQualityIndexes.jl library and for the Frobenius Norm, we will use Julia's `norm` function.

3.1 Results & Analysis

In figure 9 we report the performance of each of our compression algorithms based on a varied setting of their compression parameters. We particularly focus on the variation in image quality based on the compressed image size, which appears to vary heavily based on the quality/norm metric utilized and the algorithm itself. We go on to compare these algorithms directly to understand the relative strengths and weaknesses of each (on the sample images described).

Consider each metric for each compression algorithms in tables 1 to 3. The parameters (rank and threshold choices) are chosen in a way to make each compression algorithm compress to nearly the same compression ratio (notice the similarities in the rightmost columns), in order to control for compression ratio and observe which algorithm has the best resulting image quality metrics for a given compression ratio range. Figure 9 contextualizes the continuous scale and change in the of parameters.

We consider both discrete data that controls for compression ratio and continuous data. To begin, tables 1 to 3 suggest that HWT and DCT are superior to SVD/Rank Approximation when it comes to reducing image distortion and noise. In particular, images compressed with rank approximation have higher noise (seen in the PSNR and FNorm metrics) compared to DCT and HWT. For example, with a $10 - 11\%$ compression ratio (a desirable compression size for practical image storage), SVD, HWT, and DCT all have similar SSIM metrics at around 0.75 to 0.83. However,

they differ more drastically in the PSNR and FNorm metrics, as DCT and HWT both have a PSNR of around 29 and FNorm of 21, while SVD has a PSNR 26 and FNorm of 30. On average, per entry, DCT and HWT had around a 15% better PSNR and 35% lower FNorm than rank approximation. This suggests that although they produce visually similar images, noise and distortion may be more perceptible in an image compressed with rank approximation compared to one with HWT or DCT.

Rank Approx. (k)	**SSIM**	**PSNR**	**FNorm**	**Compression (%)**
270	0.978762	44.0164	3.94914	70.3812
150	0.921686	34.5915	11.6883	39.1006
100	0.867615	30.8102	18.0642	26.0671
70	0.817009	28.4735	23.6403	18.247
45	0.756492	26.3541	30.1739	11.7302
25	0.682075	24.2682	38.3638	6.51677
10	0.606454	21.8569	50.6393	2.60671
5	0.570184	20.4357	59.6418	1.30335
3	0.550314	19.3081	67.9094	0.782013
1	0.53075	17.5517	83.1281	0.260671

Table 1: SVD/Rank Approximation Compression Performance

Threshold (ε)	**SSIM**	**PSNR**	**FNorm**	**Compression (%)**
0.01	0.994167	50.3861	1.89677	69.6633
0.03	0.955154	39.2629	6.82614	40.1215
0.05	0.916583	34.8262	11.3767	26.566
0.07	0.881999	32.1569	15.4697	18.6637
0.1	0.836712	29.6113	20.7379	11.7697
0.15	0.775591	27.1892	27.4076	6.30781
0.25	0.691328	24.8453	35.898	2.58094
0.35	0.640606	23.6587	41.153	1.3877
0.45	0.608279	22.9208	44.8018	0.875346
0.7	0.563794	21.804	50.9487	0.386726

Table 2: HWT Compression Performance

Threshold (ε)	**SSIM**	**PSNR**	**FNorm**	**Compression (%)**
0.01	0.996107	51.0156	1.76419	75.2769
0.03	0.951103	38.7447	7.24582	42.7772
0.05	0.890736	34.1771	12.2593	26.6921
0.07	0.837194	31.6091	16.4767	17.9412
0.1	0.773141	29.2589	21.5965	10.8636
0.15	0.701652	27.0651	27.8017	5.61676
0.25	0.627718	24.9837	35.3298	2.29874
0.35	0.589338	23.9013	40.0187	1.25512
0.45	0.567068	23.2021	43.3735	0.798459
0.7	0.546328	22.1238	49.1066	0.35926

Table 3: DCT Compression Performance

Next, we directly compare HWT and DCT, as both use the same thresholding process and were run with the same parameters. Although they have similar continuous results, HWT does very slightly outperform DCT in image quality for a fixed compression ratio, specifically in the SSIM metric. In the tables, HWT has a higher SSIM for almost every

given compression ratio. On average, HWT has a 5.1% higher SSIM across every entry compared to DCT and 8.6% improvement over SVD and rank approximation. However, a 5% average difference in SSIM does not necessarily indicate that HWT is a clearly better compression algorithm than DCT. First, DCT does have slightly better FNorm and PSNR metrics compared to DCT (around 0.1%), which contributes to less distortion and noise. However, the main advantage of DCT is a quality unrelated to image quality: speed. In every test, DCT would be significantly faster to compute and use to compress an image compared to HWT. We trade a marginal decrease in SSIM (5%) for a significantly faster compression algorithm and higher computational efficiency.

3.2 Conclusions

In this work, we described the linear algebraic structure and performance of a set of (image) compression algorithms. We identified a common motivating structure in many compression algorithms, that being a change (and reduction) in basis and the consequent removal of redundant information. We saw that the three algorithms in question seemed to fall into two classes of linear algebraic compression: ones that aim to minimize the size of the vector basis representation and those that optimize the sparsity of the intermediate representation. Singular Value Decomposition falls into the former while both the Haar Wavelet and Discrete Cosine Transformations fall into the latter. Our analysis shows that the latter group performs better when trading off image quality and compression size. HWT and DCT turned out to be the best two options and their similar structure corresponded to very similar performance metrics. While SVD/Rank Approximation image compression had passable SSIM metrics, it suffered from high distortion and noise as indicated by PSNR and FNorm metrics. Based on the marginal differences in performance and compression quality, we find that HWT is likely better for applications that require strict lossless compression or high image quality, due to its higher SSIM metrics. It consistently performed well in terms of its image quality vs compression ratio trade-off. On the other hand, we believe DCT could be better for general applications, as it only sees a slight drop in image quality which may be too subtle for the human eye, while having higher performance on FNorm and PSNR metrics than HWT. The DCT algorithm is simpler than HWT and is less computationally intensive, thus is likely better suited for high-throughput, multimedia applications.

Bibliography

[1] N. Ahmed, T. Natarajan, and K.R. Rao. "Discrete Cosine Transform". In: *IEEE Transactions on Computers* C–23.1 (Jan. 1974), pp. 90–93. ISSN: 0018-9340. DOI: 10.1109/t-c.1974.223784. URL: http://dx.doi.org/10.1109/T-C.1974.223784.

[2] Nathan Bell and Michael Garland. *Efficient sparse matrix-vector multiplication on CUDA*. Tech. rep. Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[3] C. *Atomic force microscopy*. [Online; accessed April 27, 2013]. 2013. URL: https://en.wikipedia.org/wiki/File:Atomic_force_microscope_block_diagram.svg.

[4] E. Delp and O. Mitchell. "Image Compression Using Block Truncation Coding". In: *IEEE Transactions on Communications* 27.9 (Sept. 1979), pp. 1335–1342. ISSN: 0096-2244. DOI: 10.1109/tcom.1979.1094560. URL: http://dx.doi.org/10.1109/tcom.1979.1094560.

[5] R. Gray. "Vector quantization". In: *IEEE ASSP Magazine* 1.2 (Apr. 1984), pp. 4–29. ISSN: 0740-7467. DOI: 10.1109/massp.1984.1162229. URL: http://dx.doi.org/10.1109/MASSP.1984.1162229.

[6] Klaus E Holtz, Eric S Holtz, and Diana Kalienky. "Advanced data compression promises the next big leap in network performance". In: *Broadband European Networks and Multimedia Services*. Vol. 3408. SPIE. 1998, pp. 540–557.

[7] Uthayakumar Jayasankar, Vengattaraman Thirumal, and Dhavachelvan Ponnurangam. "A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications". In: *Journal of King Saud University - Computer and Information Sciences* 33.2 (Feb. 2021), pp. 119–140. ISSN: 1319-1578. DOI: 10.1016/j.jksuci.2018.05.006. URL: http://dx.doi.org/10.1016/j.jksuci.2018.05.006.

[8] C.M. Kortman. "Redundancy reduction—A practical method of data compression". In: *Proceedings of the IEEE* 55.3 (1967), pp. 253–263. ISSN: 0018-9219. DOI: 10.1109/proc.1967.5479. URL: http://dx.doi.org/10.1109/proc.1967.5479.

[9] Connor Kuhn. "SVD and DCT Image Compression". In: (2016). URL: https://www.math.utah.edu/~gustafso/s2019/2270/projects-2019/presented/fraser/Linear%20Algebra%20in%20Image%20Compression_%20SVD%20and%20DCT.pdf.

[10] David C Lay, Steven R Lay, and Judi J McDonald. *Linear algebra and its applications*. 5th ed. Upper Saddle River, NJ: Pearson, Dec. 2014.

[11] Y. Linde, A. Buzo, and R. Gray. "An Algorithm for Vector Quantizer Design". In: *IEEE Transactions on Communications* 28.1 (Jan. 1980), pp. 84–95. ISSN: 1558-0857. DOI: 10.1109/tcom.1980.1094577. URL: http://dx.doi.org/10.1109/tcom.1980.1094577.

[12] Aditya Krishna Menon and Charles Elkan. "Fast Algorithms for Approximating the Singular Value Decomposition". In: *ACM Transactions on Knowledge Discovery from Data* 5.2 (Feb. 2011), pp. 1–36. ISSN: 1556-472X. DOI: 10.1145/1921632.1921639. URL: http://dx.doi.org/10.1145/1921632.1921639.

[13] Colm Mulcahy. "Image compression using the Haar wavelet transform". In: *Spelman Science and Mathematics Journal* 1.1 (1997), pp. 22–31.

[14] Jim Nilsson and Tomas Akenine-Möller. "Understanding ssim". In: *arXiv preprint arXiv:2006.13846* (2020).

[15] R. Pasco. "Source coding algorithms for fast data compression (Ph.D. Thesis abstr.)" In: *IEEE Transactions on Information Theory* 23.4 (July 1977), pp. 548–548. ISSN: 1557-9654. DOI: 10.1109/tit.1977.1055739. URL: http://dx.doi.org/10.1109/tit.1977.1055739.

[16] Sabine Süsstrunk, Robert Buckley, and Steve Swen. "Standard RGB Color Spaces". In: *Color and Imaging Conference* 7.1 (Jan. 1999), pp. 127–134. ISSN: 2166-9635. DOI: 10.2352/cic.1999.7.1.art00024. URL: http://dx.doi.org/10.2352/cic.1999.7.1.art00024.

[17] Patrick J Van Fleet. "The discrete Haar wavelet transformation". In: *Joint Mathematical Meetings*. 2007, pp. 25–31.

[18] Sunny Verma and P Krishna. “Image compression and linear algebra”. In: (2013). URL: https://www.cmi.ac.in/~ksutar/NLA2013/imagecompression.pdf.

[19] Gaurav Vijayvargiya, Sanjay Silakari, and Rajeev Pandey. *A Survey: Various Techniques of Image Compression*. 2013. DOI: 10.48550/ARXIV.1311.6877. URL: https://arxiv.org/abs/1311.6877.

[20] J. Ziv and A. Lempel. “A universal algorithm for sequential data compression”. In: *IEEE Transactions on Information Theory* 23.3 (May 1977), pp. 337–343. ISSN: 0018-9448. DOI: 10.1109/tit.1977.1055714. URL: http://dx.doi.org/10.1109/TIT.1977.1055714.

[21] J. Ziv and A. Lempel. “Compression of individual sequences via variable-rate coding”. In: *IEEE Transactions on Information Theory* 24.5 (Sept. 1978), pp. 530–536. ISSN: 0018-9448. DOI: 10.1109/tit.1978.1055934. URL: http://dx.doi.org/10.1109/tit.1978.1055934.

A Appendix

A notebook with our implementation is attached in the following pages. Also available on Github.

Image Compression with SVD (Low Rank Approximation)

To start, we load in an image. We'll be using a test image of a lighthouse.

In [ ]:
```
# load in image
using Images, FileIO, TestImages, LinearAlgebra
img = float.(testimage("lighthouse"))
```

In [ ]:
```
#split image into channels
channels = channelview(img)
#red, green, blue channels
red = channels[1, :, :]
green = channels[2, :, :]
blue = channels[3, :, :]
```

Next, compute the rank of a channel. This is the highest rank approximation we can do (using a rank r SVD). Note, the rank of each channel is equivalent.

In [ ]:
```
rank(red)
```

Now, we define a general rank approximation function that can accept a given SVD and k value to return an approximation. In Line 2 of the function, we multiply the first k cols of U, the first k singular values, and the first k rows of V^T.

In [ ]:
```
using LinearAlgebra

#define a rank approximation function
function rank_approx(F::SVD, k)
    U, E, V = F
    M = U[:, 1:k] * Diagonal(E[1:k]) * transpose(V[:, 1:k])
    clamp01!(M)
end
```

Now, we can define a function to apply this approximation to a given image. Each channel is approximated using its SVD (stored in the vector F), and then a colorview stacking the three channels is returned.

In [ ]:
```
# return an image with low rank k approximation
function img_approx(F::Vector, k)
    rankk = rank_approx.(F, k)
    return colorview(RGB, rankk...)
end
```

We can test out a variety of rank approximations of our test image.

In [ ]:
```
#take SVD of every channel (red, green, blue)
svd_each_channel = svd.(eachslice(channels; dims=1), full = false)

#rank approximation
ranks = [1:1:9;10:25:500]
# ranks = [270, 150, 100, 70, 45, 25, 10, 5, 3, 1]

#store approximated images
imgs = map(ranks) do k img_approx(svd_each_channel, k) end

mosaic(img, imgs...; ncol=3, npad=10, rowmajor = true)
```

From left to right: Original Image, Rank Approximations in increasing order.

Comparing Original Image with Rank K approximations

First, we save each compressed image to our file system.

In [ ]:
```
# save each compressed image
save("./original.png", img)

for k in ranks
    path = "./rank" * string(k) * ".png"
    save(path, imgs[findall(x->x==k, ranks)][1])
end
```

Next, calculate the storage needed for the original image. This is the number of values across all 3 channels times the storage needed for a Float.

In [ ]:
```
# number of bytes to store the original image
num_original_bytes = length(channels)*sizeof(Float32)
Base.format_bytes(num_original_bytes)
```

We can compare this against the storage required for storing the first k U columns, singular values, and V rows. This allows us to calculate a final compression ratio.

In [ ]:
```
for k in ranks
    U1, E1, V1 = svd_each_channel[1]
    # storage needed for storing the first k U cols, singular values, and V rows
    num_compressed_bytes = (3*length(U1[:, 1:k]) + 3*length(V1[1:k, :]) + 3*k)*sizeof(Fl
    println("Rank "*string(k)*" File Size: ", Base.format_bytes(num_compressed_bytes))
    println("Compression Ratio: ", round((num_compressed_bytes / num_original_bytes)*100
    println("Frobenius Norm: ", norm(img-imgs[findall(x->x==k, ranks)][1]))
    println("")
end
```

Assessing Image Quality Differences

We use PSNR (Peak Signal-to-Noise Ratio), SSIM (Structural Similarity Index), and the Frobenius Norm to gauge differences between the original image and the images compressed with rank approximation using a variety of thresholds. This helps us assess which threshold gives the best balance between image quality and compression ratio.

In [ ]:
```
for rankk in imgs
    println("Assessing Image with Rank ", ranks[findall(x->x==rankk, imgs)][1], " Approx
    println("PSNR (Peak Signal-to-noise Ratio)", assess_psnr(rankk, img))
    println("SSIM (Structural Similarity Index): ", assess_ssim(rankk, img))
    println("Frobenius Norm: ", norm(img-rankk))
    println("")
end
```

Data Analysis: Image Quality vs Compression Ratio for SVD (Low Rank Approximation)

First, collect every image quality metric and compression ratio.

In [ ]:
```
using Plots

U1, E1, V1 = svd_each_channel[1]
```

```
#collect metrics
rank_ssim = map(imgs) do i assess_ssim(i, img) end
rank_psnr = map(imgs) do i assess_psnr(i, img) end
rank_fnorm = map(imgs) do i norm(img-i) end
rank_ratios = map(imgs) do h_i
    k = ranks[findall(x->x==h_i, imgs)][1]
    num_compressed_bytes = (3*length(U1[:, 1:k]) + 3*length(V1[1:k, :]) + 3*k)*sizeof(Fl
    (num_compressed_bytes / num_original_bytes)*100
end
```

Use DataTables to make a table of image quality metrics and compression ratios.

In [ ]:
```
using DataTables

rank_d = DataTable(Ranks=ranks, SSIM=rank_ssim, PSNR=rank_psnr, FNorm=rank_fnorm, Compre
```

Normalize each metric to display them alongside each other:

In [ ]:
```
rank_ssim = (rank_ssim .- minimum(rank_ssim)) / (maximum(rank_ssim) .- minimum(rank_ssim
rank_psnr = (rank_psnr .- minimum(rank_psnr)) / (maximum(rank_psnr) .- minimum(rank_psnr
rank_fnorm = (rank_fnorm .- minimum(rank_fnorm)) / (maximum(rank_fnorm) .- minimum(rank_
```

Finally, use Julia Plots to graph our data.

In [ ]:
```
rank_g = plot(rank_ratios, [rank_ssim rank_psnr rank_fnorm], ylabel="Normalized Metrics"
rank_g
```

Image Compression with the Haar Wavelet Transform

In [ ]:
```
img
```

First, apply the Haar Wavelet transform on every channel (red, green, blue).

In [ ]:
```
using Wavelets, WaveletsExt
wt = wavelet(WT.haar)

#apply Haar Wavelet transform to red, green, and blue channels
transformed_channels = map(eachslice(channels; dims=1)) do channel
    transformed = wpt(channel, wt)
end
```

Display channels with Haar Wavelet Transform applied.

In [ ]:
```
colorview(RGB, transformed_channels...)
```

Next, threshold every channel by a certain value. What thresholding does is that every value in a matrix at or below a given threshold is replaced with a zero entry. As we see later, this compresses the image by making the matrix "sparser".

In [ ]:
```
#define a thresholding generation function
function generate_bounded_normal_samples(n, mean, std, lower_bound, upper_bound)
    # Generate normally distributed samples
    samples = mean .+ std * randn(n)

    # Clamp the samples to the specified lower and upper bounds
    samples = clamp.(samples, lower_bound, upper_bound)
```

```
    return sort(unique(samples))
end
```

In [ ]:

```
# thresholds = [0.001:0.001:0.009; 0.01:0.01:1;]
# thresholds = [0.01, 0.03, 0.05, 0.07, 0.1, 0.15, 0.25, 0.35, 0.45, 0.7, 1]
thresholds = generate_bounded_normal_samples(50, 0.1, 0.1, 0.001, 1)
thresholds = unique(sort([0.00001:0.00025:0.01; 0.001:0.005:0.2; 0.2:0.1:1;]))

haar_imgs = map(thresholds) do th
    thresholded_channels = map(transformed_channels) do channel
        new_channel = copy(channel)
        threshold!(new_channel, HardTH(), th)
    end
end
```

Display the hard-thresholded transformed channels:

In [ ]:

```
views = map(haar_imgs) do h_img
    colorview(RGB, h_img...)
end
# optional: display the transformed channels.
# mosaicview(views...; ncol=5, rowmajor=true, npad=10)
```

Finally, apply the inverse Haar Wavelet transform on every channel of every image to recover the thresholded image. We display the original image along with the new compressed images below.

In [ ]:

```
ihaar_imgs = map(haar_imgs) do haar_img
    ihaar_channels = map(haar_img) do haar_channel
        iwpt(haar_channel, wt)
    end
    colorview(RGB, ihaar_channels...)
end

mosaic(img, ihaar_imgs...; ncol=4, rowmajor=true, npad=10, fillvalue=1)
```

From left to right: Original image, and then all thresholded images in increasing order.

Calculating Image Compression with Haar Wavelet Transform

First, note the number of nonzero entries in the original transformed image.

In [ ]:

```
countnz = length(transformed_channels[1][transformed_channels[1] .!= 0]) + length(transf
```

Now, count the nonzero entries in the thresholded transformed matrices. The idea is that the higher the threshold, the more zero entries exist in a matrix. Thus the matrix would be more sparse, and less space would be needed to store it.

In [ ]:

```
haar_nz = map(haar_imgs) do h_i
    nz = length(h_i[1][h_i[1] .!= 0]) + length(h_i[2][h_i[2] .!= 0]) + length(h_i[3][h_i
    nz
end
```

Now, compare the nonzero entries in each matrix to the nonzero entries in the original matrix to get a final compression ratio.

In [ ]:

```
for i in haar_nz
    println("Compression Ratio: ", round(i/countnz * 100, sigdigits =3), "% with thresho
end
```

```
In [ ]: # Example: img 2, with an applied threshold of 0.03 and compression ratio of 40.1%
ihaar_imgs[2]
```

Assessing Image Differences:

We use PSNR (Peak Signal-to-Noise Ratio), SSIM (Structural Similarity Index), and the Frobenius Norm to gauge differences between the original image and the images compressed with Haar Wavelet using a variety of thresholds. This helps us assess which threshold gives the best balance between image quality and compression ratio.

```
In [ ]: using ImageQualityIndexes

for haar_img in ihaar_imgs
    println("Assessing matrix with threshold ", thresholds[findall(x->x==haar_img, ihaar_
    println("PSNR (Peak Signal-to-noise Ratio)", assess_psnr(haar_img, img))
    println("SSIM (Structural Similarity Index): ", assess_ssim(haar_img, img))
    println("Frobenius Norm: ", norm(img-haar_img))
    println("")
end
```

Data Analysis: Image Quality vs Compression Ratio for Haar Wavelet

First, collect every image quality metric and compression ratio.

```
In [ ]: using Plots

haar_ssim = map(ihaar_imgs) do i assess_ssim(i, img) end
haar_psnr = map(ihaar_imgs) do i assess_psnr(i, img) end
haar_fnorm = map(ihaar_imgs) do i norm(img-i) end
haar_ratios = map(haar_imgs) do h_i
    ((length(h_i[1][h_i[1] .!= 0]) + length(h_i[2][h_i[2] .!= 0]) + length(h_i[3][h_i[3]
end
```

Make a table of metrics and compression ratios.

```
In [ ]: using DataTables

haar_dt = DataTable(Thresholds=thresholds, SSIM = haar_ssim, PSNR=haar_psnr, FNorm=haar_
```

Normalize the data for plotting.

```
In [ ]: haar_ssim = (haar_ssim .- minimum(haar_ssim)) / (maximum(haar_ssim) .- minimum(haar_ssim
haar_psnr = (haar_psnr .- minimum(haar_psnr)) / (maximum(haar_psnr) .- minimum(haar_psnr
haar_fnorm = (haar_fnorm .- minimum(haar_fnorm)) / (maximum(haar_fnorm) .- minimum(haar_
```

Finally, plot the data.

```
In [ ]: haar_g = plot(haar_ratios, [haar_ssim haar_psnr haar_fnorm], ylabel="Normalized Metrics"
haar_g
```

Image Compression with DCT

First, apply the DCT transform on every channel (red, green, blue).

```
In [ ]: using FFTW
```

```
#apply DCT transform to red, green, and blue channels
transformed_channels = map(eachslice(channels; dims=1)) do channel
    transformed = dct(channel)
end
```

Next, threshold every channel by a certain value. What thresholding does is that every value in a matrix at or below a given threshold is replaced with a zero entry. As we see later, this compresses the image by making the matrix "sparser".

In [ ]:
```
# thresholds = [0.001:0.001:0.009; 0.01:0.01:1;]

dct_imgs = map(thresholds) do th
    imgs = map(transformed_channels) do channel
        new_channel = copy(channel)
        threshold!(new_channel, HardTH(), th)
    end
end
```

Finally, apply the inverse DCT transform on every channel of every image to recover the thresholded image. We display the original image along with the new compressed images below.

In [ ]:
```
idct_imgs = map(thresholds) do th
    imgs = map(transformed_channels) do channel
        new_channel = copy(channel)
        threshold!(new_channel, HardTH(), th)
        abs.(idct(new_channel))
    end
    colorview(RGB, imgs...)
end

mosaic(img, idct_imgs...; ncol=4, rowmajor=true, npad=10, fillvalue=1)
```

From left to right: Original image, and then all thresholded images in increasing order.

Assessing Compression with DCT

First count the nonzero entries in the original transformed matrix. Then count the nonzero entries in the thresholded transformed matrices. The idea is that the higher the threshold, the more zero entries exist in a matrix. Thus the matrix would be more sparse, and less space would be needed to store it.

In [ ]:
```
countnz = length(transformed_channels[1][transformed_channels[1] .!= 0]) + length(transf

dct_nz = map(dct_imgs) do h_i
    nz = length(h_i[1][h_i[1] .!= 0]) + length(h_i[2][h_i[2] .!= 0]) + length(h_i[3][h_i
end
```

Now, compare the nonzero entries in each matrix to the nonzero entries in the original matrix to get a final compression ratio.

In [ ]:
```
for i in dct_nz
    println("Compression Ratio: ", round(i/countnz * 100, sigdigits =3), "% with thresho
    println("")
end
```

Assessing Image Quality

We use PSNR (Peak Signal-to-Noise Ratio), SSIM (Structural Similarity Index), and the Frobenius Norm to gauge differences between the original image and the images compressed with DCT using a variety of

thresholds. This helps us assess which threshold gives the best balance between image quality and compression ratio.

In [ ]:
```
for dct_img in idct_imgs
    println("Assessing matrix with threshold ", thresholds[findall(x->x==dct_img, idct_i
    println("PSNR (Peak Signal-to-noise Ratio)", assess_psnr(dct_img, img))
    println("SSIM (Structural Similarity Index): ", assess_ssim(dct_img, img))
    println("Frobenius Norm: ", norm(img-dct_img))
    println("")
end
```

Data Analysis: Image Quality vs Compression Ratio for DCT

First, collect every image quality metric and compression ratio.

In [ ]:
```
dct_ssim = map(idct_imgs) do i assess_ssim(i, img) end
dct_psnr = map(idct_imgs) do i assess_psnr(i, img) end
dct_fnorm = map(idct_imgs) do i norm(img-i) end
dct_ratios = map(dct_imgs) do h_i
    ((length(h_i[1][h_i[1] .!= 0]) + length(h_i[2][h_i[2] .!= 0]) + length(h_i[3][h_i[3]
end
```

Make a table with metrics and compression ratios.

In [ ]:
```
dct_dt = DataTable(Thresholds=thresholds, SSIM = dct_ssim, PSNR=dct_psnr, FNorm=dct_fnor
```

Normalize the data for plotting.

In [ ]:
```
dct_ssim = (dct_ssim .- minimum(dct_ssim)) / (maximum(dct_ssim) .- minimum(dct_ssim))
dct_psnr = (dct_psnr .- minimum(dct_psnr)) / (maximum(dct_psnr) .- minimum(dct_psnr))
dct_fnorm = (dct_fnorm .- minimum(dct_fnorm)) / (maximum(dct_fnorm) .- minimum(dct_fnorm
```

Next, plot the quality metrics vs the compression ratio.

In [ ]:
```
using Plots

dct_g = plot(dct_ratios, [dct_ssim dct_psnr dct_fnorm], ylabel="Normalized Metrics", yli
dct_g
```

Final Data Analysis

In summary, we can make a chart of the average metrics across all three compression algorithms.

In [ ]:
```
using Statistics

rank_mean_ssim = mean(rank_ssim)
haar_mean_ssim = mean(haar_ssim)
dct_mean_ssim = mean(dct_ssim)

rank_mean_psnr = mean(rank_psnr)
haar_mean_psnr = mean(haar_psnr)
dct_mean_psnr = mean(dct_psnr)

rank_mean_fnorm = mean(rank_fnorm)
haar_mean_fnorm = mean(haar_fnorm)
dct_mean_fnorm = mean(dct_fnorm)

rank_metrics = [mean(rank_ssim),
```

```
mean(rank_psnr),
mean(rank_fnorm)]

haar_metrics = [mean(haar_ssim),
mean(haar_psnr),
mean(haar_fnorm)]

dct_metrics = [mean(dct_ssim),
mean(dct_psnr),
mean(dct_fnorm)]

categories = ["SVD", "HWT", "DCT"]

bar(categories, [rank_metrics, haar_metrics, dct_metrics], barwidth=0.2, labels = ["SSIM
```

Finally, we can make a larger graph that shows, for each compression algorithm, the relationship between the thresholds/rank parameters chosen and the resulting image quality metrics on a continous scale

In [ ]:
```
using CairoMakie

ssimcolor=:gray7
psnrcolor=:cornflowerblue
fnormcolor=:brown2
ssimline=:solid
psnrline=:solid
fnormline=:solid

xticksrange=0:25:125
xticksall=(xticksrange, [string(val) * "%" for val in xticksrange])
yticksall=0:0.25:1

marksizeplot = 4

fig = Figure(resolution=(1800, 800),margin=(10,10,10,10))
ax1 = CairoMakie.Axis(fig[2, 1], xticks=xticksall, yticks=yticksall)
ax2 = CairoMakie.Axis(fig[2, 2], xticks=xticksall, yticks=yticksall)
ax3 = CairoMakie.Axis(fig[2,3], xticks=xticksall, yticks=yticksall)


ax4 = CairoMakie.Axis(fig[1, 1], xticks=0:128:512,yticks=xticksall, xlabel="Rank", title
ax5 = CairoMakie.Axis(fig[1, 2], yticks=xticksall, xlabel="Threshold", title="Haar Wavel
ax6 = CairoMakie.Axis(fig[1,3], yticks=xticksall, xlabel="Threshold", title="Discrete Co

l1 = Makie.scatter!(ax1, rank_ratios, rank_ssim, color=ssimcolor, marker=:circle, marker
l1 = Makie.lines!(ax1, rank_ratios, rank_ssim, color=ssimcolor, linestyle=ssimline)
l2 = Makie.scatter!(ax1, rank_ratios, rank_psnr, color=psnrcolor, marker=:circle, marker
l2 = Makie.lines!(ax1, rank_ratios, rank_psnr, color=psnrcolor, linestyle=psnrline)
l3 = Makie.scatter!(ax1, rank_ratios, rank_fnorm, color=fnormcolor, marker=:circle, mark
l3 = Makie.lines!(ax1, rank_ratios, rank_fnorm, color=fnormcolor, linestyle=fnormline)

Makie.scatter!(ax2, haar_ratios, haar_ssim, color=ssimcolor, marker=:circle, markersize=
Makie.lines!(ax2, haar_ratios, haar_ssim, color=ssimcolor, linestyle=ssimline)
Makie.scatter!(ax2, haar_ratios, haar_psnr, color=psnrcolor, marker=:circle, markersize=
Makie.lines!(ax2, haar_ratios, haar_psnr, color=psnrcolor, linestyle=psnrline)
Makie.scatter!(ax2, haar_ratios, haar_fnorm, color=fnormcolor, marker=:circle, markersiz
Makie.lines!(ax2, haar_ratios, haar_fnorm, color=fnormcolor, linestyle=fnormline)

Makie.scatter!(ax3, dct_ratios, dct_ssim, color=ssimcolor, marker=:circle, markersize=ma
Makie.lines!(ax3, dct_ratios, dct_ssim, color=ssimcolor, linestyle=ssimline)
Makie.scatter!(ax3, dct_ratios, dct_psnr, color=psnrcolor, marker=:circle, markersize=ma
Makie.lines!(ax3, dct_ratios, dct_psnr, color=psnrcolor, linestyle=psnrline)
Makie.scatter!(ax3, dct_ratios, dct_fnorm, color=fnormcolor, marker=:circle, markersize=
Makie.lines!(ax3, dct_ratios, dct_fnorm, color=fnormcolor, linestyle=fnormline)
```

```
Makie.scatter!(ax4, ranks, rank_ratios, color=:black, marker=:circle, markersize=marksiz
Makie.lines!(ax4, ranks, rank_ratios, color=:black, linestyle=ssimline)
Makie.scatter!(ax5, thresholds, haar_ratios, color=:black, marker=:circle, markersize=ma
Makie.lines!(ax5, thresholds, haar_ratios, color=:black, linestyle=psnrline)
Makie.scatter!(ax6, thresholds, dct_ratios, color=:black, marker=:circle, markersize=mar
Makie.lines!(ax6, thresholds, dct_ratios, color=:black, linestyle=psnrline)


ax1.ylabel = "Normalized Metrics"
ax4.ylabel = "Compression Ratio"
ax5.xscale = log10
ax6.xscale = log10

Makie.xlims!(ax1, (0,maximum(rank_ratios)))
Makie.xlims!(ax2, (0, 100))
Makie.xlims!(ax3, (0, 100))
ax2.xlabel = "Compression Ratio"
Legend(fig[3,2],
    [l1, l2, l3],
    ["SSIM", "PSNR", "Frobenius Norm"],
    margin = (10, 10, 10, 10),
    orientation = :horizontal)

ax4.height=170
save("graphed-compressions.pdf", fig)
```

B HWT Matrix Transformation Implementation

```
using Plots, TestImages, Images, LinearAlgebra
img = testimage("mandril_gray");
x = convert(Array{Float64}, img);
xw0 = dwt(x, wavelet(WT.db2), 1);
p0 = heatmap(xw0,
yflip=true,
color=:greys,
legend=false,
xaxis=false,
yaxis=false, xticks=false, yticks=false, aspect_ratio=:equal);
plot!(p0, title="Standard WT")


function generate_wk_matrix(k)
    if k < 1
        error("k must be greater than or equal to 1.")
    end

    n = 2^k
    matrix = zeros(Int, n, n)

    # Fill the top half of the matrix
    for i in 1:Int(n/2)
        matrix[i, 2*i-1:min(2*i, n)] .= 1
    end
    # Fill bottom half of the matrix
    for i in 1:Int(n/2)
        matrix[Int(n/2)+i, 2*i-1:min(2*i, n)] .= 1
        matrix[Int(n/2)+i, 2*i-1] = -1
    end

    return matrix
end

function hwt(X)
    dims = size(X)
    println(dims[1])
    if ((dims[2] != 1) || (!ispow2(dims[1])))
        error("should be a 2^k x 1 matrix")

    end

    return 0.5*generate_wk_matrix(Int(log2(dims[1]))) * X
end

function quantize(x, eps)
    if abs(x) < eps
        return 0
    else
        return x
    end
end

function quantize_matrix(X,eps)
    quantizer(x)=quantize(x,eps)
    return quantizer.(X)
end
```

```
function hwt2d(X)
    k = Int(log2(size(X)[1]))
    j = Int(log2(size(X)[2]))
    return 0.5*generate_wk_matrix(k)*X*transpose(0.5*generate_wk_matrix(j))
end

function invert2d(X)
    k = Int(log2(size(X)[1]))
    j = Int(log2(size(X)[2]))
    return transpose(generate_wk_matrix(k))*X*(generate_wk_matrix(j))
end

function hwt_compress_helper(X,eps,minsize=0)
    k = Int(log2(size(X)[1]))
    j = Int(log2(size(X)[2]))
    if (k != j)
        error("must be square matrix for compression")
    end

    if k <= minsize
        return X
    end

    res = hwt2d(X)
    quantizer(x)=quantize(x,eps)

    res[1:2^(k-1), 1:2^(k-1)] = hwt_compress_helper(res[1:2^(k-1), 1:2^(k-1)], eps, minsize)
    res[2^(k-1)+1:2^k, 1:2^(k-1)] = quantize_matrix(res[2^(k-1)+1:2^k, 1:2^(k-1)], eps)
    res[1:2^(k-1), 2^(k-1)+1:2^k] = quantize_matrix(res[1:2^(k-1), 2^(k-1)+1:2^k], eps)
    res[2^(k-1)+1:2^k, 2^(k-1)+1:2^k] = quantize_matrix(res[2^(k-1)+1:2^k, 2^(k-1)+1:2^k], eps)

    return res
end

function hwt_invert_helper(X, minsize=0)
    k = Int(log2(size(X)[1]))
    j = Int(log2(size(X)[2]))
    if (k != j)
        error("must be square matrix for compression")
    end

    if k <= minsize
        return X
    end
    res = X
    res[1:2^(k-1), 1:2^(k-1)] = hwt_invert_helper(res[1:2^(k-1), 1:2^(k-1)], minsize)

    res = invert2d(res)
    return res
end

function hwt_compress(x,eps,minsize=-1)
    k = Int(log2(size(x)[1]))
    if minsize == -1
        return hwt_compress_helper(x,eps)
    end
    return hwt_compress_helper(x,eps,k-minsize)
end
```

```julia
function hwt_decompress(x,depth=-1)
    k = Int(log2(size(x)[1]))
    if depth == -1
        return hwt_invert_helper(x)
    end
    return hwt_invert_helper(x,k-depth)
end


function normalize_matrix(matrix)
    return (matrix .- minimum(matrix)) / (maximum(matrix) - minimum(matrix))
end

function visualize_matrix(matrix)
    # Normalize the matrix to be in the range [0, 1]
    normalized_matrix = normalize_matrix(matrix)

    # Convert the normalized matrix to a color image
    color_image = Gray.(normalized_matrix)

    # Display the image using Plots
    plot(heatmap(color_image), size=(500, 500), c=:grays, ticks=false, border=:none)
end
```