A (Very Brief) Overview and Comparison of Some Techniques for Image Compression

Ali Inan
Department of Computer Science
Courant Institute of Mathematical Sciences

May 11, 2018

Abstract

We explore a few techniques for image compression, namely singular value decomposition, the discrete cosine transform, and a discrete wavelet transform and compare their performance. We then consider two transform-based compression techniques, the Discrete Cosine Transform, and a discrete wavelet based transform (specifically using the Haar wavelet). We test these methods on a standard test image and compare the compressed image outputs and error values. The SVD, being the simplest and most naive method, not surprisingly does not give the best performance, giving around only a 5-2.5 compression ratio depending on what quality level is found acceptable. The wavelet and cosine transform methods performed noticeably better than SVD. The difference between wavelet and cosine transforms is a bit harder to discern, but it seems the cosine transform has a slight edge.

Introduction

With the proliferation of cameras, many of them capable of taking very high resolution photos, good compression algorithms are more necessary than ever. The basic idea behind most compression techniques is to transform an image into a form where we can isolate and discard the less important features, resulting in an image that is fairly close to the original. We will explore some methods that allow us to do this, starting with the singular value decomposition, and then the discrete cosine and wavelet transforms. We will compare their efficacy, both in terms of objective performance (error between compressed and original image) and more subjective qualities.

Compression with Singular Value Decomposition

We start with what is probably one of the simplest methods of image compression, based on the well known singular value decomposition. The basic idea is that given the singular value decomposition of a matrix, we can create a matrix that is 'close' to the original but is lower rank and thus takes less information to encode. This idea is formalized by the Eckart-Young-Mirsky theorem, which we briefly summarize below [1].

First we start with an image with matrix representation M. That is, given an image with $m \times n$ pixels, we consider the $m \times n$ matrix where each entry is the value of the corresponding pixel (usually a value from 0 to 255 if standard 8-bit color is used, although for convenience, often these values will be normalized by dividing through by 255). Also for convenience, throughout this paper we will consider only black-and-white images, which are completely represented by a single matrix of grayscale values. Color images on the other hand are usually represented by 3 8-bit integers for each pixel, one for each of the red, green, and blue (RGB) values , thus requiring 3 matrices per image. In theory, all of the techniques we will describe will be able to be applied to color images simply by applying them to each of the RGB color matrices separately. However in practice, algorithms for color compression are often more sophisticated and specialized, exploiting how humans perceive and interpret color. (As an example, the JPEG standard transforms the RGB components to a different space known as the YCbCr space, where Y represents a quantity known as 'luma', roughly meaning brightness, and Cb and Cr represent the blue and red chrominance components, which are the red and blue values subtracted by the luma value [2].)

Now, given the singular value decomposition of the matrix representation of an image M,

$$M = USV^T$$

where U is an $m \times m$ orthogonal matrix where S is the diagonal $m \times n$ matrix of singular values σ_i in descending order, and V^T is an $n \times n$ orthogonal matrix, we first note for convenience that we can rewrite the SVD as a sum of rank-1 matrices as follows:

$$\sum_{i=1}^{n} u_i \sigma_i v_i^T$$

where u_i are the columns of U, σ_i the diagonal elements of S (i.e. the singular values), and v_i the columns of V (or alternatively the rows of V^T). Note that each v_i has m components, u_i n components, and σ_i is just a scalar, so that each term in the sum above requires storing $m + n + 1$ values, for a total of $n(m + n + 1)$ values for the entire matrix.

The crucial insight that allows us to reduce the size of our representation is realizing that often most of the important information for the matrix is contained in the first few terms of the above sum (i.e. terms corresponding to the largest singular values), and thus we can

simply discard them. More precisely, instead of the full rank-1 sum, we keep only the first k terms in the rank-1 sum for the SVD:

$$M_k = \sum_{i=1}^{k} u_i \sigma_i v_i^T$$

Note that M_k is a matrix of lower rank (specifically rank-k, since it has k non-zero singular values), and that the sum above also gives the SVD of M_k, so that the difference between M and M_k is also given in SVD form, and that the norm of this difference (i.e. the error) can be expressed very simply:

$$||M - M_k||_2 = \left|\left| \sum_{i=1}^{n} u_i \sigma_i v_i^T - \sum_{i=k}^{n} u_i \sigma_i v_i^T \right|\right|_2 = \left|\left| \sum_{i=k+1}^{n} u_i \sigma_i v_i^T \right|\right|_2 = \sigma_{k+1}$$

While the 2-norm is nice, in image processing we are usually interested more in how images compare pixel-by-pixel, which is much better represented by the Frobenius norm. Using the easily verified fact that the the Frobenius norm of a matrix A is $\sqrt{\text{tr}(A^T A)}$, we can derive a particularly nice expression for the Frobenius norm of a matrix in terms of its SVD:

$$||A||_F = \sqrt{\text{tr}(A^T A)} = \sqrt{\text{tr}((USV^T)^T USV^T)} = \sqrt{\text{tr}(VS^T U^T USV^T)} = \sqrt{\text{tr}(VSS^T V^T)} =$$

$$= \sqrt{\text{tr}(VSS^T V^T)} = \sqrt{\text{tr}(S^T V^T VS)} = \sqrt{\text{tr}(S^T S)} = \sqrt{\sum_{i=1}^{n} \sigma_i^2}$$

where we have used the fact that $\text{tr}(AB) = \text{tr}(BA)$. Thus we have

$$||M - M_k||_F = \left|\left| \sum_{i=k+1}^{n} u_i \sigma_i v_i^T \right|\right|_F = \sqrt{\sum_{i=k+1}^{n} \sigma_i^2}$$

In either case, we see that the smaller the discarded singular values, the smaller our error. It's not obvious that many of the trailing singular values will always be 'small', but fortunately it is often the case that they are. Note that the original $m \times n$ images requires storing mn values, while the low rank approximation image M_k requires storing $k(m+n+1)$ values. Thus, in order for this procedure to be worthwhile (i.e. actually achieve compression), we must have that $k(m+n+1) < mn$. In the common case of square $n \times n$ images, we have that $k < \frac{n^2}{2n+1} \approx n/2$. In other words, we must throw away at least half the singular values in order to get some compression. Of course, we will ideally want to discard much more than half.

To give a bit more theoretical justification to this method, we now finally prove the rest of the Eckhart–Young–Mirsky theorem. That is, we show that the matrix produced by discarding the singular values is in some sense the optimal matrix of lower rank that approximates M. That is, for all rank-k matrices B_k,

$$||M - M_k||_F \leq ||M - B_k||_F$$

(The result is also true for the 2-norm, but we will only show it for the more relevant Frobenius norm). First we note that if we let $M = M' + M''$, then by the triangle inequality, $\sigma_1(M) = ||M||_2 \leq ||M'||_2 + ||M''||_2 = \sigma_1(M') + \sigma_(M'')$, where by $\sigma_i(M)$ we mean the ith singular value of M. By using the triangle inequality again, and letting M_i denote the rank-i approximation to M as we have defined above, we get

$$\sigma_i(M') + \sigma_j(M'') = \sigma_1(M' - M'_{i-1}) + \sigma_1(M'' - M''_{j-1}) \geq \sigma_1(M' + M'' - M'_{i-1} - M''_{j-1})$$

$$= \sigma_1(M - M'_{i-1} - M''_{j-1}) \geq \sigma_{i+j-1}(M)$$

We also have that for any rank-k matrix B_k, $\sigma_{k+1}(B_k) = 0$, so if $M' = M - B_k$ and $M'' = B_k$, we have that for $i \geq 1, j \geq k+1$

$$\sigma_i(M - B_k) + \sigma_j(B_k) = \sigma_i(M - B_k) \geq \sigma_{i+(k+1)-1}(M) = \sigma_{i+k}(M)$$

This finally gives us

$$||M - B_k||_F = \sum_{i=1}^{b} \sigma_i^2(M - B_k) \geq \sum_{i=k+1}^{n} \sigma_i^2(M) = ||M - M_k||_F$$

which gives the desired result, namely that of all rank-k matrices, M_k gives the smallest Frobenius error.

We now proceed to the experimental results. Here is our original uncompressed 512x512 test image (obtained from `http://sipi.usc.edu/database/database.php?volume=misc`):

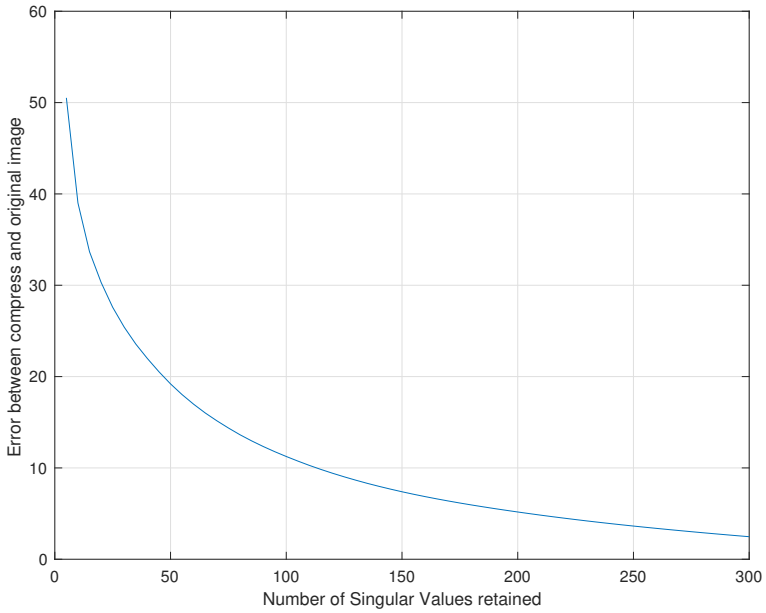
Here are some selected compressed images using the SVD technique:

First row: 5,10,20 singular values retained, Bottom row: 50,100,200 singular value retained

Not surprisingly, just retaining a few singular values is not sufficient for a decent compression for most images. It would be remarkable if most images were simple enough to be characterized by only a few singular values. However, increasing the number of singular values retained even slightly does seem to give significantly better results up to a point. Deciding at what point the image is acceptable is of course a subjective matter and depends on how the image will be used (e.g. for medical or legal purposes we probably would place a premium on quality and fidelity). In this case it would seem reasonable to say that by retaining 50 singular values (which gives a compression ratio of $512^2/50(1025) \approx 5.12$) we already have a reasonable image for many purposes. Of course, at 50 singular values retained, it's not too difficult to see that the image is still fairly distorted. If we are interested only in roughly preserving the shapes and general features well enough so that they can be identified, this might suffice. However, if we want to be a bit more conservative, we can go up to 100 singular values retained, (compression ratio around 2.55). At that point, we need to try very hard to tell the difference between the original and the compressed image.

For completeness, we show a plot of the Frobenius error between the compressed image

and the original for various numbers of retained singular values. (All errors throughout this paper will be the Frobenius norm of the normalized difference between the original image and the compressed image, i.e. all entries of the image matrices will be between 0 and 1).



The graph confirms most of our analysis from looking at the images. Initially, we see a fairly steep decline, as most of the gains in fidelity are from the initial (largest) singular values. Once we get past 50 singular values retained, the curve starts to flatten out, as we see the diminishing returns to each extra singular value retained. We see that even when we get to an acceptable image quality where it is very difficult to tell the compressed image from the original, we can still have a fair bit of error, so just looking at error does not provide a good idea of the efficacy of our techniques.

Compression with the Discrete Cosine Transform

While the SVD approach is fairly simple, and depending on the application, perhaps good enough, not surprisingly there are ways to get better results if we are willing to use more sophisticated tools. One approach often used in the real world is the Discrete Cosine Transform, a relative of the well known Discrete Fourier Transform [4]. The intuition behind using Fourier transforms for compression (and image processing in general) is that they provide us with information about the frequencies present in an image, and allow us to manipulate (and most importantly for compression, remove) these frequencies. While the concept of frequency for signals like sound is fairly obvious, the concept of frequency of an image is a bit murkier. The frequency of an image can be thought of as how rapidly an image is changing in color and intensity. Sharp lines and contrast generally lead to high frequency

areas, while smoother and gentler color gradients suggest low frequency areas. One of the guiding assumptions of many of these frequency-based compression techniques is that human eyes are not very sensitive to high frequencies, or that high frequencies are most often noise, and thus can often be discarded. Of course, this isn't always true, since sharpness is often desired in images, and throwing away too much leads to blurry, poor quality compression.

We again start with some background. First, let us review the standard Discrete Fourier Transform (DFT).Given n complex numbers $x_0, x_1, \ldots, x_{N-1}$, we produce n output numbers $X_0, X_1, \ldots X_{N-1}$ which are given by

$$X_k = \sum_{n=0}^{N-1} x_k e^{-\frac{2\pi i k n}{N}}$$

A particularly convenient way of thinking about the DFT is to consider the inputs and outputs as vectors in $\mathbb{R}^n$. Then the DFT can be written as a linear transformation with the transformation being represented by a vandermonde matrix of the roots of unity, $\omega = e^{-2\pi i/N}$. This representation makes it clear that the DFT is invertible, a crucial property for image compression since we need to be able to decode any compressed image data to get back a useable image.

While the Fast Fourier Transform can be used for image processing, for compression in particular, a variant known as the Discrete Cosine Transform (DCT) is much more popular for a variety of reasons. One reason of course is that the Fast Fourier Transform involves complex numbers which go to waste since we are working with real-valued pixel images. The DCT, being a cosine wave, conveniently only involves real numbers. Empirically, it also just seems that in general, the DCT seems to provide better compression rates. A reason for this might be that the DCT behaves 'nicer' in some sense for images. If we recall, the standard Fourier method is based on assuming a periodic extension to the data it is applied to. Of course, a periodic extension is not guaranteed to continuous. The cosine transform on the other hand is based on assuming an even extension which is produced by reflecting the data at the boundary, which necessarily leads to a continuous extension. This might lead to better convergence, and a property known as the 'energy compaction' property of the DCT, which says that the DCT can capture more of a data set with less coefficients than the standard Fourier transform, thus allowing us to discard more coefficients without losing as much information [5].

There are multiple variants of the DCT, the most commonly used in image processing known as the DCT-II, which is given by the formula:

$$X_k = \sum_{n=0}^{N-1} x_k \cos\left[\frac{\pi}{N}\Big(n + \frac{1}{2}\Big)k\right]$$

Like the Discrete Fourier Transform, the DCT can also be represented as a linear transformation, and thus has a matrix representation (and thus is also invertible). In fact, if we

try to write out the matrix for the DCT (which we will not to save space), it conveniently turns out to be orthogonal (with the right scaling factors which were not given in our above expression for the DCT). This makes computing the inverse transform a much more pleasant prospect, since we can simply take the transpose. Because of this, the inverse transform for DCT-II also has a particularly simple form, up to a normalizing constant, given by:

$$X_k = \frac{1}{2}x_0 + \sum_{n=0}^{N-1} x_k \cos\left[\frac{\pi}{N}\Big(n + \frac{1}{2}\Big)k\right]$$

Of course, we are interested in applying a DCT to a matrix, not a series of values or vector. Thankfully, applying a DCT over 2 dimensions (or even higher) is as simple as you might hope. We simply compose the transforms along each dimension. That is, we can apply a DCT to each row, and then each column. Thus we have that the output coordinates of a 2D DCT are given by:

$$X_{k_1,k_2} = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2=1} x_{n_1,n_2} \cos\left[\frac{\pi}{N_1}\Big(n_1 + \frac{1}{2}\Big)k_1\right] \cos\left[\frac{\pi}{N_2}\Big(n_2 + \frac{1}{2}\Big)k_2\right]$$

The formula for the inverse 2D DCT follows in exactly the same way by applying a 1D inverse DCT along each row of the matrix, and then applying a inverse DCT to the resulting columns.

We now give a brief overview of the exact method we will use to compress our image, which is based on the Joint Photographic Experts Group (JPEG) standard for image compression. Of course the full JPEG standard is fairly complicated, but at its core it involves a DCT transform similar to the one we will describe [3].

1.) First we divide the image into 8×8 pixel blocks. (If the image dimensions are not divisible by 8, then the image can be padded with empty pixels).

2.) We then apply the 2 dimensional DCT to each of the 8×8 blocks. Thus for each block, we get a new 8×8 matrix whose entries are now the transformed coefficients.

3.) We apply a mask to each of the DCT 8×8 blocks to zero out the high frequency entries. A typical mask for an 8×8 block might be the following:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

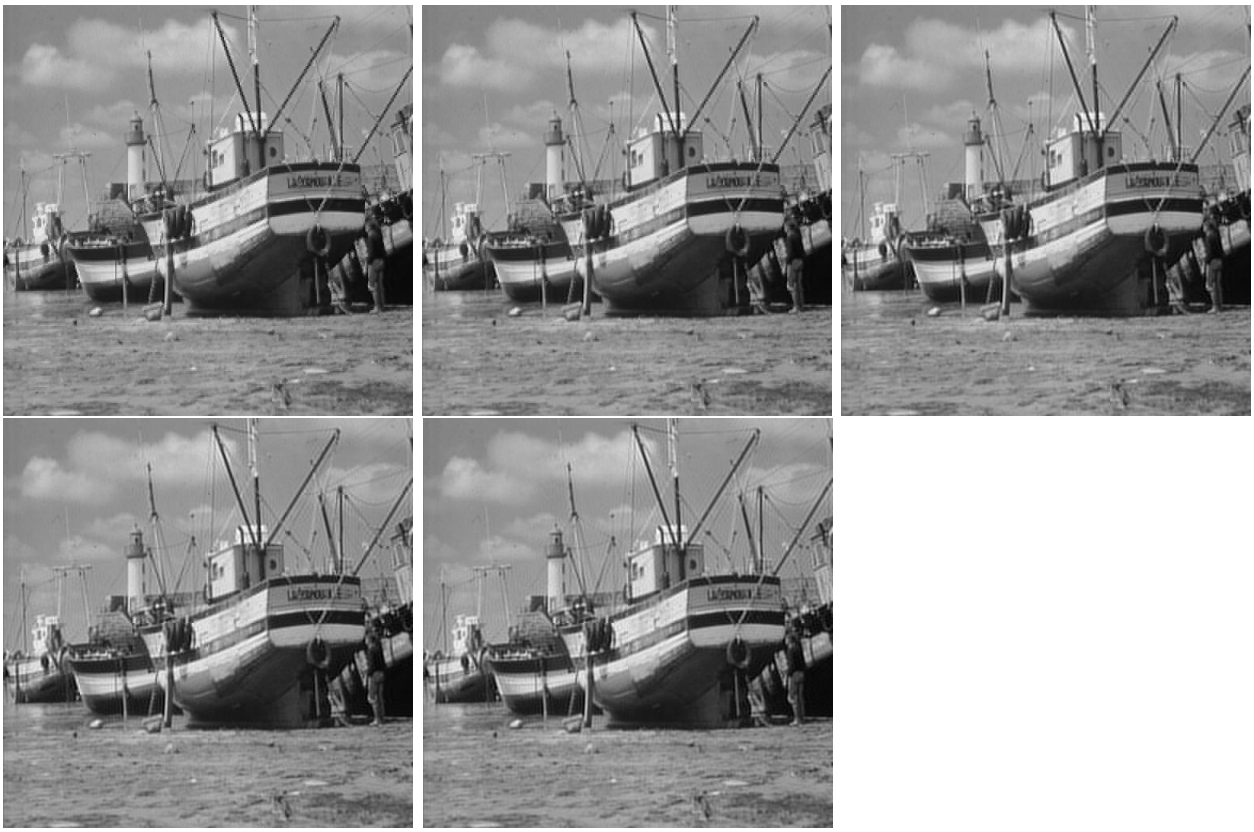
Note that this matrix is multiplied with the DCT coefficients element-by-element (also known as the Hadamard product), not the usual matrix multiplication. This zeros out all entries except for the upper-left triangle, which correspond to the coefficients with low indices, and thus also the low frequencies which we want to retain. This is the step where the actual compression occurs, since the sparser our masked matrix, the more compactly we can represent it. In fact, we can calculate the compression ratio explicitly as the ratio between the number of nonzero elements in the original image and the transformed and masked image. (Of course, just having 0s in a matrix does not lead to compression, since a 0 value still takes an integer number of bits to represent, but sparse matrices in general can be encoded with less bits when using an entropy encoding algorithm such as Huffman encoding. In an actual implementation of a compression algorithm at this step, we would use a standard compression algorithm such as Huffman encoding or run length encoding to encode the matrix, and the exact compression ratio would not be our calculated ratio in terms of nonzero elements, although it should be fairly close).

4.) Once we have our compressed image, we can decode it to convert it into a displayable format by applying the inverse DCT to each block. Of course, since we have applied a mask, applying the inverse DCT will not give us back our original matrix, but since we have selectively only discarded high frequencies which should not be too important, our decoded image should be a fairly accurate reconstruction.

Two things stick out in particular as somewhat arbitrary. One is the choice of mask used, and the other being the choice of dividing the image into 8×8 blocks. The choice of mask is indeed arbitrary. Other than the fact that we want to somehow eliminate high frequencies, there aren't really many guiding principles behind the mask. Many of the coefficients corresponding to the higher frequencies will already be close to zero after applying the DCT, so another possible strategy would be to use a thresholding approach, where we go through the block and set every element whose distance to 0 is within the threshold value to 0. Another commonly used approach instead of multiplying by 0, is to divide each entry by a large number and then round. This involves using what is known as a quantization matrix, where each entry contains the number that the entry in the DCT matrix is divided by. Of course, the higher frequencies have higher weights in the quantization matrix, and there is much work in trying to figure out optimal quantization matrices [3]. However, we will stick to using our basic 1-0 mask since it allows us to easily control how much compression we want by setting the number of entries zeroed out in the mask.

The choice of 8×8 blocks however is specified by the JPEG standard, so presumably three is a good reason for it. However, given that the JPEG standard was originally created in 1992, perhaps there are reasons which may have been valid then (e.g. computational cost) that are not particularly relevant now, so it is probably worthwhile to investigate the effects of block size on performance.

To see exactly how important these factors are, we try applying the algorithm with different block sizes. We try block sizes of 4×4, 8×8, 16×16, 32×32, and 64×64. The mask array used for each block was the default upper-left triangle with sides $n/2$ as shown above in the sample mask.



Different block sizes: Top row: 4x4,8x8,16x16, Bottom row: 32x32, 64x64

From the images it's not clear at all if block size has any significant effect on quality. In fact, even going all the way up to one block of 512x512 didn't seem to make much difference. However, our concern that maybe larger block sizes might be computationally inefficient doesn't seem to hold either, at least for this image. We summarize some of the findings in the table below:

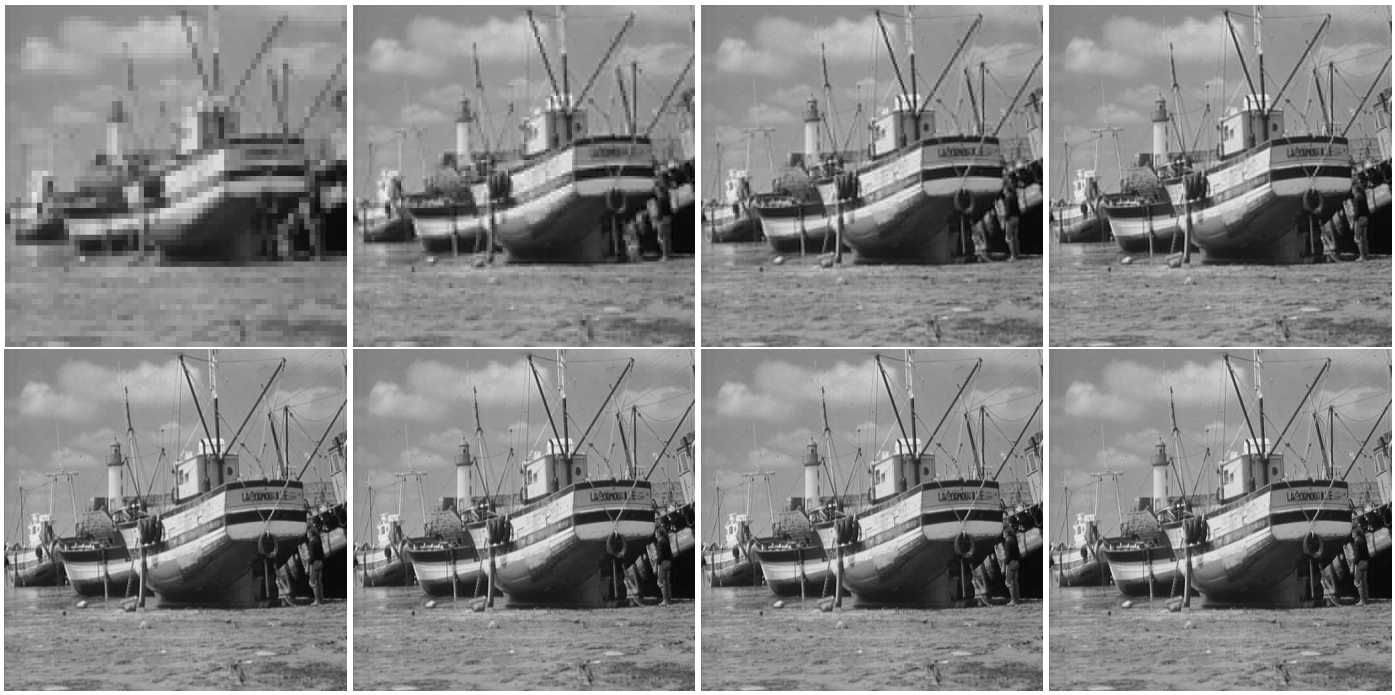
	Compression Ratio	Frobenius Error	Run Time (s)
4x4	5.33	18.44	0.848
8x8	6.40	18.22	0.201
16x16	7.11	18.08	0.095
32x32	7.53	17.78	0.084
64x64	7.76	17.73	0.054
512x512	7.97	17.63	0.096

It seems our concern for larger block sizes being more computationally inefficient was misplaced, since it appears that bigger block sizes lead to faster run times, at least for this image, since the time to run seemed to actually improve as block size increased, up to a point. The difference in errors between different block sizes also seems negligible, particularly when simply looking at the images, in which case they are basically indistinguishable. However, there is a clear trend in decreasing error, as well as increasing compression ratio as we increase the block size. Overall, for this example at least, it seems that there are no downsides to increasing the block size.

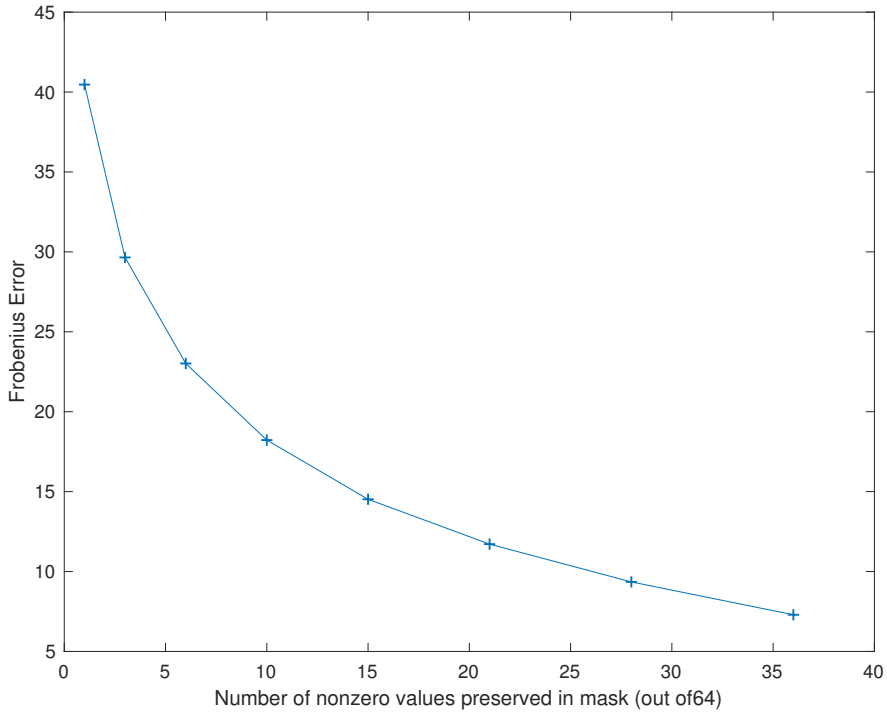
Of course, it is very possible (perhaps likely) that in general, larger block sizes don't behave so nicely. If we view the transform as averaging the contents of a block and consolidating them into a few coefficients, it seems likely that given an image with lots of variance in a block, a large block size would not be very suitable. If we make our blocks too small of course, we will not get very much compression. Thus our goal is to balance the need for finer block granularity to retain acceptable image quality with the desire for more compression. While the JPEG standard suggests a block size of 8, perhaps this should be looked into further. Of course, there are some factors we did not consider that might make 8 a good choice, such as the possibility that with more sophisticated implementations on systems with many processors or graphics processing units, perhaps using smaller block size is more efficient since each block can be processed independently in parallel.

Now we try changing the mask matrix used. The default mask we used was an upper-left triangle of 1s with side length $n/2$. We will simply vary this by first using a $1x1$ upper left triangle mask (i.e. zeroing every pixel except the upper left in each block), then $2x2$ upper-left triangle (i.e. we will zero every entry except (1,1), (1,2) and (2,1)), and so on, until we get to the triangle with side length n. i.e. we will zero out the bottom-right half triangle of the block. Note that we can estimate the compression ratio directly form the mask, since a triangle mask of side length n will retain $n(n+1)/2$ nonzero values, the ratio of nonzero values in the original to the nonzero values after applying the mask should be $64/(n(n+1)/2) = 128/(n(n+1))$.

We now present the results of changing the mask matrix (and thus also changing the compression ratio):



Top row: mask $n = 1, 2, 3, 4$, Bottom row: mask $n = 5, 6, 7, 8$



Just from looking at the images, it seems that the DCT method overall does much bet-

ter than the SVD technique. At the lowest level of compression (corresponding to just a single nonzero value per block being preserved, or a compression ratio of around 64!) we get an image that is still much better than the lowest quality SVD image (which retained 5 singular values, corresponding to a compression ratio of around 51.1) Thus we get a better looking image even with more compression! Looking at the errors corroborates this, as our 64 times compression image has an error of close to 40, while the SVD error for the 51.1 compression ratio was near 50. At all levels, the DCT seems to do beter than the SVD. We reach acceptable image quality with a higher level of compression than SVD.

Interestingly, we can also see qualitative differences in the ways the SVD method and the DCT method distort the images. While both give 'blurry' low images, in the low quality images with SVD, we can almost see the columns of the matrix. Because the SVD gives us a very low rank matrix, we should be able to see that columns are either very similar or multiples/linear combinations of each other. In the lowest quality SVD images, we can make out 'bands' that are similar in shade and shape, representing these linearly dependent columns. On the other hand, the 'blur' of the DCT images is 'blockier' and a bit harder to predict, although we can somewhat make out individual blocks. This is not surprising since our DCT method is based on dividing the image into blocks. If we throw away too much information from each block, we oversimplify it, leading to the blocky appearance.

Compression with a Discrete Wavelet Transform

Finally we consider a somewhat newer approach based on wavelets. While Fourier-based methods such as the Discrete Cosine Transform just described remain popular and widely used, wavelets purport to offer some advantages over Fourier-based transforms, namely that they take into account not only the frequencies found in a signal, but also when/where they occur. Of course, this comes at the cost of added complexity. On an informal level, wavelets, as the name suggest, are little waves, or brief oscillations, that start out with 0 amplitude, increase, and then go back down to zero amplitude. By shifting and translating this original wavelet (often called the 'mother' wavelet) around, we can create other a system of wavelets with which we can construct any other (square integrable) function, just as we do with the Fourier bases. The fact that wavelets are not oscillating over their entire domain is the crucial property that allows them to capture the local properties of functions [6].

Unlike Fourier-based transforms, which all essentially have some variation of the standard trigonometric functions as bases, there are many different possible mother wavelets which can behave very differently. The one we will focus on is a particularly simple wavelet (both conceptually and implementation-wise), the Haar wavelet.

We introduce the Haar wavelet, $\psi(x)$, and it's associated scaling function $\varphi(x)$ given by

$$\psi(x) = \begin{cases} 1 & 0 \leq x \leq \frac{1}{2} \\ -1 & \frac{1}{2} \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}, \varphi(x) = \begin{cases} 1 & 0 \leq x < 1 \\ 0 & \text{otherwise} \end{cases}$$

Given the mother wavelet and scaling function we also introduce the parametrized functions

$$\psi_{j,k}(x) = 2^{j/2}\psi_(2^j x - k)$$

$$\varphi_{j,k}(x) = 2^{j/2}\varphi(2^j x - k)$$

which are simply the original wavelet and scaling functions shifted and dilated. While the theory of wavelets is very interesting, we will not delve into the theory here, but take on faith that it is not too difficult to verify that the Haar wavelet and its child wavelets (produced by shifting and scaling) satisfy the many properties required of a system of wavelets (a crucial property being the admissibility condition which guarantees that we can invert our transformation) [7]. Instead, what we are more interested in is the Discrete Wavelet Transform, which given our functions ψ and φ, and a set of N points $x[0], \ldots, x[N-1]$ are given by somewhat the complicated expressions:

$$W_\varphi(j_0, k) = \sum_{m=0}^{N-1} x[m]\varphi_{j_0,k}[m]$$

$$W_\psi(j, k) = \sum_{m=0}^{N-1} x[m]\psi_{j,k}[m], j > j_0$$

where $j_0 = 0$, j goes from j_0 to $J-1$, $N = 2^J$, and k goes from 0 to $2^j - 1$ (we have also omitted normalizing constants). The $W_\varphi(j_0, k)$ are known as the approximation coefficients, and the $W_\psi(j, k)$ as detail coefficients [8].

While these expressions seem very complicated, fortunately like with the DCT, we can represent them with a surprisingly simple matrix, the Haar transform matrix, which has a fairly simple description of how it acts on a matrix. First we outline how the discrete Haar transform operates on a single row of a matrix. Given a row of an 8×8 matrix, (a,b,c,d,e,f,g,h), the discrete Haar transform groups the elements in pairs (a,b),(c,d),(e,f),(g,h) and returns the 8 element vector whose first 4 entries are the average of the pairs (the approximation coefficients), and the next 4 are half the difference of the pairs (the detail coefficients). This is then repeated on the transformed row, except on the first four elements, replacing the first four with 2 approximation coefficients representing the averages of the pairs, and 2 detail coefficients. This is finally repeated one last time on the first two elements of the transformed vector, replacing the first with the average, and the second with half the difference. Like for the DCT, to extend this to 2D, we simply apply the transform to the rows, and then to the

columns. It is a bit cumbersome, but not hard, to verify that the above operations can be applied to each row in an 8×8 matrix by multiplying with the following matrix:

$$H_8 = \begin{pmatrix} \frac{1}{8} & \frac{1}{8} & \frac{1}{4} & 1 & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{8} & \frac{1}{8} & \frac{1}{4} & 0 & -\frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{8} & \frac{1}{8} & -\frac{1}{4} & 0 & 0 & \frac{1}{2} & 0 & 0 \\ \frac{1}{8} & \frac{1}{8} & -\frac{1}{4} & 0 & 0 & -\frac{1}{2} & 0 & 0 \\ \frac{1}{8} & -\frac{1}{8} & 0 & \frac{1}{4} & 0 & 0 & \frac{1}{2} & 0 \\ \frac{1}{8} & -\frac{1}{8} & 0 & \frac{1}{4} & 0 & 0 & -\frac{1}{2} & 0 \\ \frac{1}{8} & -\frac{1}{8} & 0 & -\frac{1}{4} & 0 & 0 & 0 & \frac{1}{2} \\ \frac{1}{8} & -\frac{1}{8} & 0 & -\frac{1}{4} & 0 & 0 & 0 & -\frac{1}{2} \end{pmatrix}$$

To apply this to each column then, after applying to each row, we simply multiply the resulting matrix on the left by H_8^T. That is, given an 8×8 matrix A, the full 2D discrete Haar transform of A is given simply by $H_8^T A H_8$.

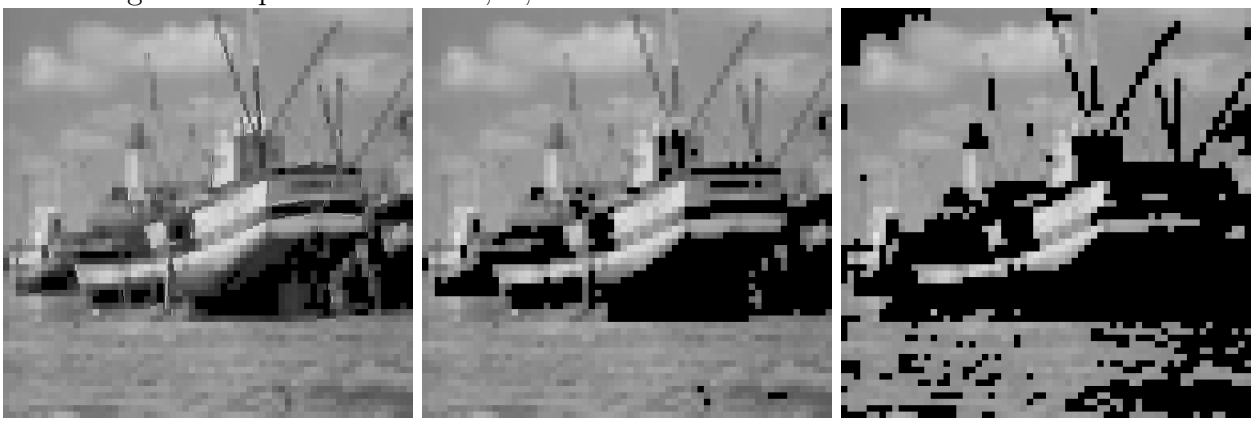
Now that we are equipped with a wavelet transformation matrix, we will apply the same general strategy as we did for the DCT. Like in DCT compression, our goal is to use the transform to get a sparser matrix than we started with. We will divide the image into 8×8 blocks and apply the transform on each block. Due to the repeated halving behavior of the Haar transform, many of the entries in the transformed 8×8 block will already be fairly small. However, instead of using a mask, we will go through the matrix and individually set entries to 0 based on whether or not they are within some threshold distance to 0. By varying this threshold distance, we can be more or less aggressive with our zeroing, and thus control the compression ratio. To reconstruct the image from the compressed matrix, we simply apply the inverse Haar transform (using the inverse of the matrix given above).

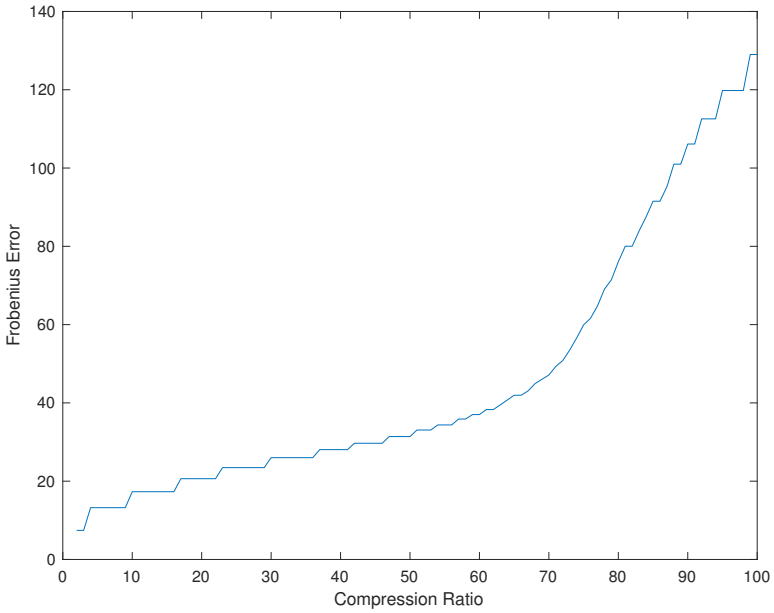
Some selected wavelet transform compressed images:

Left to right: Compression ratio 2,4,8

Left to right: Compression ratio 16,32,50

Left to right: Compression ratio 64,75,100





Unlike the comparison between DCT and SVD, the comparison between the wavelet transform and DCT is not so clear. One interesting thing is due to using a threshold system to zero the transformed coefficients instead of using a block mask, we are able to reach much higher levels of compression since our maximum compression ratio isn't limited by the block size. Although of course as the 100 compression ratio example shows, the results are not nice at all. Due to the fact that we used a block division method as DCT, we still observe some of the same 'blockiness' in the lower quality images. From the graph it is clear that error increases somewhat moderately for compression ratios up to about 70, after which it starts increasing rapidly. This is easily seen in the 75 and 100 compression ratio photos, where entire sections become just solid shades of a single color. Overall it has hard to make a case for the Haar wavelet transform over the DCT. For the lower compression rates, both perform fairly similarly, and the difference seems negligible. However, as we go into the higher compression rates, particularly considering the compression ratio of 64 case, the DCT image seems slightly less blurry. Of course, we should keep in mind that we used one of the simplest and most crude wavelet transforms, so this result is not surprising nor an indictment of wavelet methods overall.

Conclusion

After trying all the methods outlined, it is clear that the frequency transform based methods are more effective than the SVD based technique, which should really only be used as a pedagogical exercise (unless some serious modifications are made to it). However, between the Discrete Cosine Transform and the Wavelet, the results are not so clear, although we are inclined to say that the DCT is slightly better. While we have taken error into account, even when there are differences in error (sometimes non trivial differences), the image with higher error might look better. At the end of the day, subjective image quality should be ultimate arbiter of how successful a compression algorithm is, since our eyes perceive images as wholes, not as sums of differences of pixel values.

Given that JPEG, a standard developed in 1992, is still one of the most popular image compression techniques in the world, it might seem like there is little progress or work to be done in image compression. However, there still seems to be room for improvement and exploration in compression techniques. Even considering a single algorithm, there are almost limitless variations in the permutations of parameters such as block size, masking matrix, threshold strategy/cutoffs, etc. that can be used, and it is unlikely that the optimal ones have been found. Future work in this area could explore this further. Also, ideally, we would have liked to test all our techniques on a very large database of representative images, not just one or a few. To truly see how compression algorithms perform in the real world. they should be subject to a rigorous test containing images that are representative of real world images. Our treatment of wavelets was also undeservedly short, showcasing only a particularly simple wavelet rather than the numerous other, more complicated ones which are probably more suited to image compression. Doing a survey of just the various different

wavelet based compression techniques would be a possible further extension of this work.

References

[1] T. Roughgarden and G. Valiant (2015) The Singular Value Decomposition (SVD) and Low-Rank Matrix Approximations, Stanford CS168 Lecture Notes

[2] E. Hamilton (1992) JPEG File Interchange Format, Version 1.02

[3] G. K. Wallace (1992) The JPEG Still Picture Compression Standard *IEEE Transactions on Consumer Electronics* Vol. 38, No. 1

[4] G. Strang (1999), The Discrete Cosine Transform, *SIAM Review*, Vol. 41, No. 1, 135–147

[5] N. Ahmed, T. Natarajan, and K. R. Rao (1974), Discrete Cosine Transform *IEEE Transactions on Computers*, 90–93

[6] G. Strang (1993) Wavelet Transforms vs Fourier Transforms *Bulletin of the American Mathematical Society*, Vol. 128, No.2, 288–305

[7] I. Daubechies (1992) *10 Lectures on Wavelets*, Springer-Verlag, 10–16

[8] R. Wang, Discrete Wavelet Transform, Harvey Mudd College E161 Lecture Notes

Appendix

SVD Compression Code

```
% read and process image
I = imread('boat.tiff');
I = im2double(I);

% decomposing image with SVD
[U,S,V]=svd(I);

err = [];
numSVals = [];
m = 256;
n = 256;
for n=5:5:300
    % C will be the new low rank approximation diagonal matrix
    C = S;

    % discard the singular values
    C(n+1:end,:)=0;
    C(:,n+1:end)=0;

    % Reconstruct image with new low rank approximation
    D=U*C*V';

    % display and compute error
    if n == 5 || n == 10 || n == 20 || n == 50 || n == 100 ||
    n == 150 || n == 200 || n == 250
    figure;
    buffer = sprintf('Image output using %d singular values', n);
    imshow(D);
    title(buffer);
    out_filename = strcat(strcat('svd', num2str(n)), '.jpg');
    imwrite(D, out_filename);
    end
    error=(norm(I-D,'fro'));

    % store vals for plot
    err = [err; error];
    numSVals = [numSVals; n];
end
```

DCT Compression Code

```
function [I_in, DCT, rDCT, I_comp] = compressimageDCT( image_name, mask_id )
% Function to compress an image and return relevant results
% Some code based on http://www.mathworks.com/help/images/discrete-cosine-
% transform.html

% Get appropriate mask matrix
if length(mask_id) > 1
    mask = mask_id;
else
    mask = getmask(mask_id);
end

% Read image data
I_in = imread(image_name);
I_in = im2double(I_in);

% Take the DCT of each 8x8 block
size = length(mask);
T = dctmtx(size);
dct_function = @(block) T * block.data * T';
DCT = blockproc(I,[size size],dct_function);

% Apply mask matrix to zero out values in the DCT matrix
rDCT = blockproc(DCT,[size size],@(block) mask .* block.data);

% Apply inverse DCT to compressed matrix data and return the decoded compressed image
idct_function = @(block) T' * block.data * T;
I_comp = blockproc(rDCT,[size size],idct_function);
```

Wavelet Compression Code

```
%Apply Haar Transform to 8x8 blocks in matrix
function image_haar=haar_comp(image_in,eps)

h=[1/8  1/8  1/4    0  1/2    0    0    0;
   1/8  1/8  1/4    0 -1/2    0    0    0;
   1/8  1/8 -1/4    0    0  1/2    0    0;
   1/8  1/8 -1/4    0    0 -1/2    0    0;
   1/8 -1/8    0  1/4    0    0  1/2    0;
   1/8 -1/8    0  1/4    0    0 -1/2    0;
   1/8 -1/8    0 -1/4    0    0    0  1/2;
   1/8 -1/8    0 -1/4    0    0    0 -1/2];

haar = @(block) h'*block.data*h;
h_size = length(h);
image_haar= blockproc(image_in,[h_size h_size],haar);

%Zero out entries in the transformed matrix based on threshold
image_haar(image_haar < abs(eps) & image_haar > -abs(eps))=0;

%Apply Inverse Haar Transform to 8x8 blocks in matrix
function image_out=haar_decomp(image_haar)

h=[1/8  1/8  1/4    0  1/2    0    0    0;
   1/8  1/8  1/4    0 -1/2    0    0    0;
   1/8  1/8 -1/4    0    0  1/2    0    0;
   1/8  1/8 -1/4    0    0 -1/2    0    0;
   1/8 -1/8    0  1/4    0    0  1/2    0;
   1/8 -1/8    0  1/4    0    0 -1/2    0;
   1/8 -1/8    0 -1/4    0    0    0  1/2;
   1/8 -1/8    0 -1/4    0    0    0 -1/2];

invhaar = @(block) inv(h)'*block.data*inv(h);
h_size = length(h);
image_out = blockproc(image_haar,[h_size h_size],invhaar);
```