Image Compression using Singular Value Decomposition and Discrete Cosine Transformation

Andrew Haas, Mark Van der Merwe, Ann Wilcox
Math 2270-002 Spring 2017

Introduction

Matrices and Linear Algebra are an integral part of many applications in computing such as image representation and manipulation. Using the properties of matrices and their various properties, we can manipulate images in many useful ways, such as the compression of image data. Here we explore two ways of manipulating image matrices for compression. The first is Singular Value Decomposition (SVD) and the second is Discrete Cosine Transformation (DCT). SVD, as will be shown, is less effective than DCT, which is regularly used in JPEG compression.

Singular Value Decomposition

Background

Singular Value Decomposition is similar to Eigenvalue Decomposition in that the singular values of a matrix A are the square roots of the eigenvalues of the matrix A^TA. The same principle for Eigenvalue Decomposition, $A=PDP^{-1}$ is then applied to Singular Value Decomposition. The reason that we cannot use Eigenvalue Decomposition for images is because it only works for square matrices, so we use SVD because A^TA is guaranteed to be square. The general formula for SVD is given as $A=U\Sigma V^T$, where A is an mxn matrix of rank r, U is an mxm matrix, Σ is an mxn matrix, and V is an nxn orthogonal matrix. The Σ matrix is obtained by D, which is a diagonal matrix containing the first r singular values of A in decreasing order, and filling in zeros around D to get the correct mxn dimension. The U matrix is obtained by finding the left singular vectors of A, which are the eigenvectors of AA^T. The V matrix is obtained by finding the right singular vectors of A, which are the eigenvectors of A^TA. This can be used in image compression by "throwing" out values; as the values are in decreasing order, the majority of the image information is contained in the first and largest values. Once the "useless" values are changed to zero, the matrices are multiplied together to get a new image matrix, which corresponds to a smaller file size, thus a compressed image.

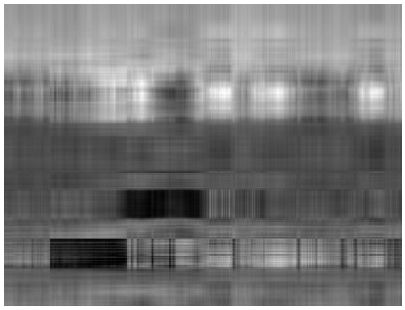
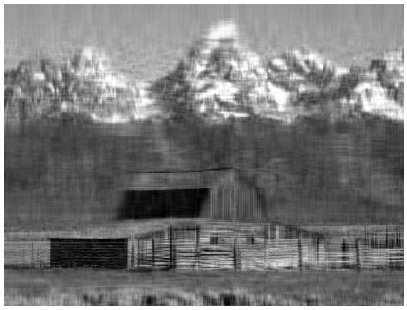
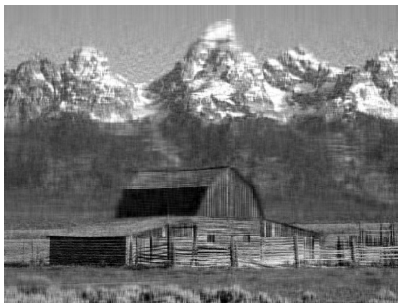
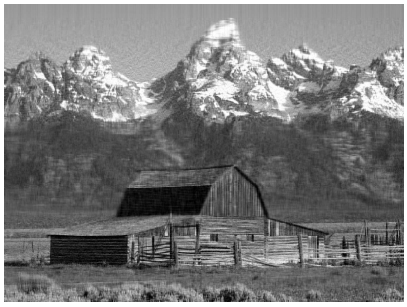
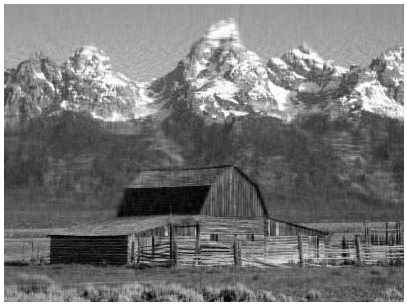
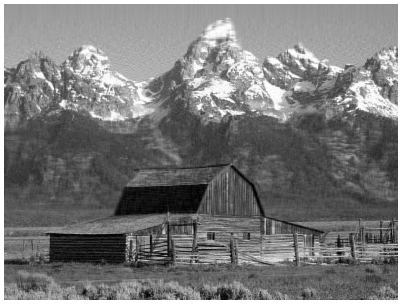
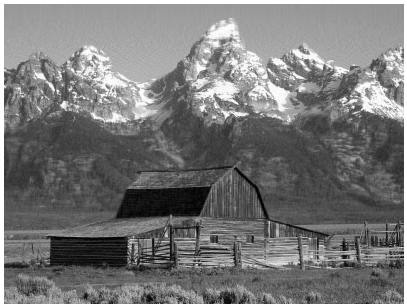
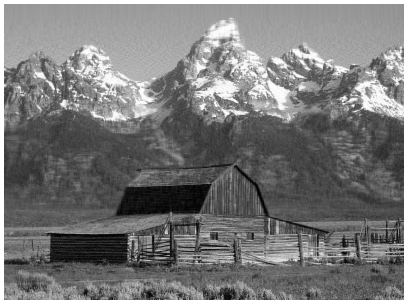
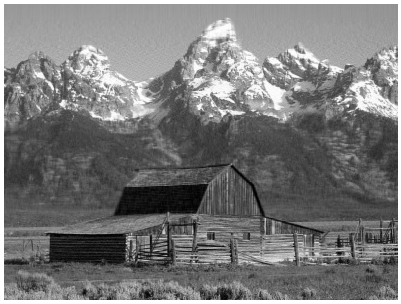
Process

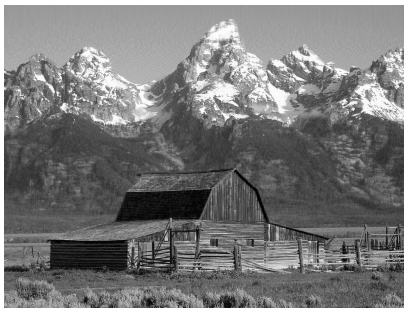
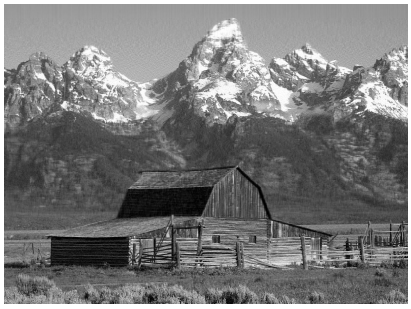
To do the Singular Value Decomposition for our image, we first uploaded the image to Maple. Then, we converted the image to grayscale to get a 2-dimensional image, and converted the image to an image matrix. After we found our image matrix, we performed the Singular Value Decomposition on our image matrix to find the singular values. We had to manipulate matrix and create our own Σ matrix as maple only gave us a vector containing the values. Once we knew our values, we decided on our k, which would determine the amount of values we would keep and which would change to zero. For our k we chose to start at 5 and increase at increments of 20 until we ended at 205. We then did the matrix multiplication for each k, along with running the error, which shows the image quality, and calculated the compression ratio.

Pictures

Original Image:

Compressed Images: K = 5,25,45,65,86,105,125,145,165,186,205

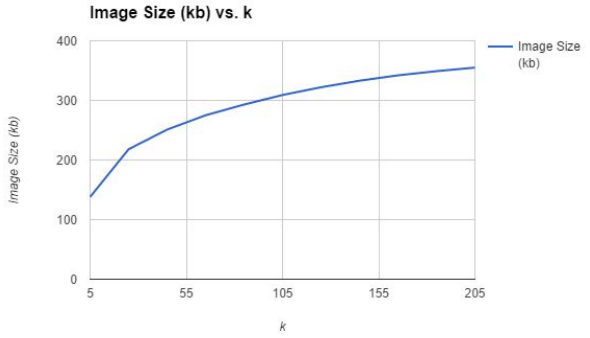



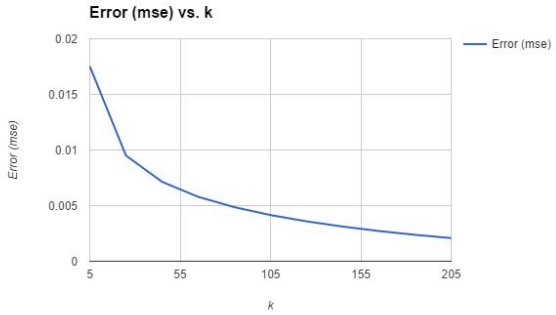
Data Analysis

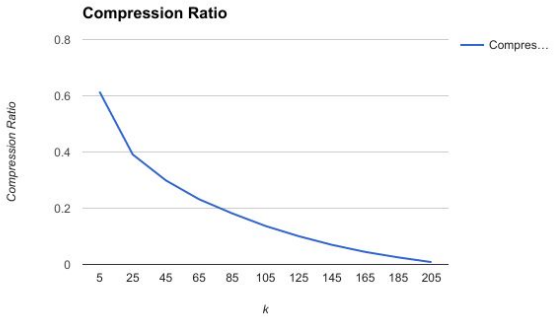
The following is the data that we got from our experiment with Singular Value Decomposition.

k	Image size (kb)	Error (mse)	Compression Ratio
5	138	.01752945165	.6145251397
25	218	.009504274225	.3910614525
45	251	.007137542359	.2988826816
65	275	.005789235824	.2318435754
85	293	.004861137596	.1815642458
105	309	.004151782269	.1368715084
125	322	.003585364946	.1005586592
145	333	.003115425366	.06983240223
165	342	.0027716585017	.04469273743
185	349	.00237558426	.0251396648
205	355	.002081830939	.008379888268

These are the graphs we made to help visualize our data:







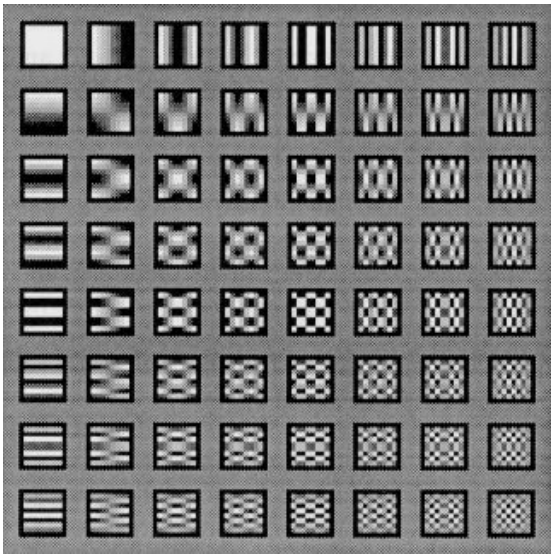
The first graph represents how the image file size changes as k (# of elements we keep) increases. As one could expect, each increase in k changes the file size by less than the previous element. This is due to each consecutive element influencing the image less than the previous one. The reason that we stop our experiment at 205 k is that we found that increasing k beyond that will result in a bigger file size than the original file. The second graph compares the error between the compressed file and the original file. The error decreases as we increase k because we now have more terms that influence the image, and as expected the curve bottoms out due to each k having less impact than the previous one. Mean Square Error(mse) is the way we measured the difference between two images. To find the MSE you square the the difference between a pixel in the original image and the corresponding pixel in the compressed image. By doing this for every pixel, summing the differences, and dividing by the number of pixels we get an approximate error. This number represents the average squared difference for each pixel in the image like the name suggests. Lastly the third graph represents the compression ratio of SVD as k changes. We defined the compression ratio as 1 - the compressed file size over the original file size. This graph is the opposite of the image size graph because as k increases the file size increases which causes the ratio to 0.

Discrete Cosine Transformation

Background

In order to perform a Discrete Cosine Transformation on we take advantage of the addition of sinusoidal waves. In applying DCT, we express images as a series of data points as a sum of cosine functions at different frequencies. The trick is to represent an image using a sinusoidal wave. We simply

let 1 be white and -1 be black and everything else be some gradient between the two. We can then increase or decrease the frequency to change the image. Below is what is called the 8 by 8 DCT matrix which shows how the frequencies of each of the sinusoidal waves and their combinations change.



Here we can see how the changes in the frequencies result in different images. The secret to DCT is that we split a given image into 8 by 8 pixel groups, or submatrices, and then determine how we can combine the above 8 by 8 images from our DCT matrix to form each of our original 8 by 8 matrices. By combining the sinusoidal waves represented in the DCT, it will return to our original image (approximately).

How much each square from the DCT matrix contributes is stored in a corresponding contribution matrix. In order to determine the actual values in our contribution matrix we apply what is called DCT II:

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right] \qquad k = 0, \ldots, N-1.$$

The actual compression comes when we apply a mask upon the resulting contribution matrix. For our purposes we applied a simple mask that removed all but some of the values in the top left corner. The reason we keep the top left is that those simpler matrices almost always contribute more to the 8 by 8 square we are dealing with. So, we remove most of the values and recombine the 8 by 8 matrices with more zeros (thus taking up less space).

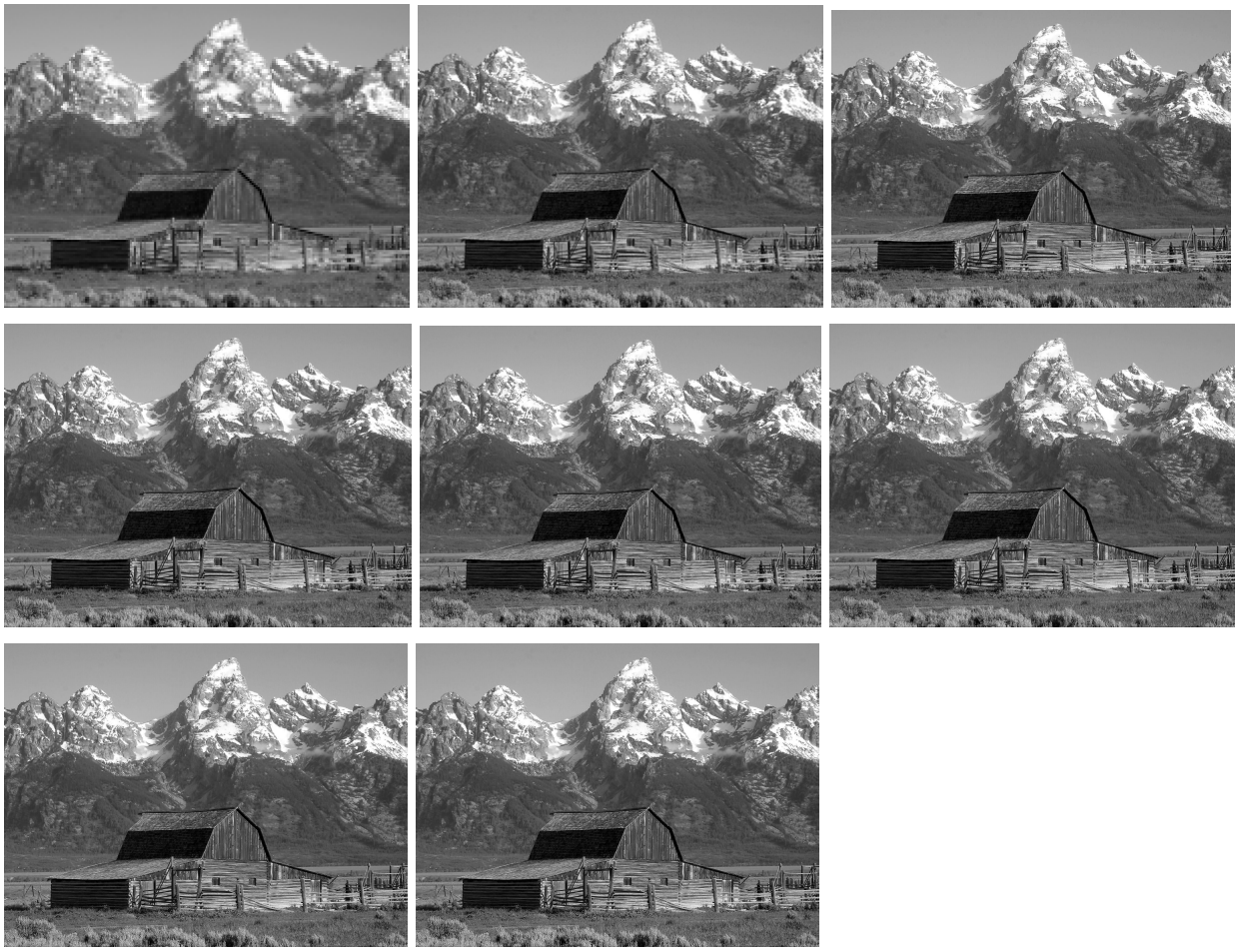
Process

We wrote a simple script using Matlab that uploads the image, then grayscales it (for simplicities sake). We then created our 8 by 8 DCT matrix (like the one above) and determined the contribution matrix for each 8 by 8 sub matrix in our image. We applied our mask to remove the smaller DCT values and recombined. To test different levels of compression, we increased the number of values being kept in each contribution matrix starting at just 1 and increasing by diagonals in our matrix to 34. We then, as in our SVD work, calculated error using Mean Square Error and calculated compression ratio.

Pictures

Original Image is the same as for SVD.

Compressed Images: k=1,3,6,10,15,21,27,34



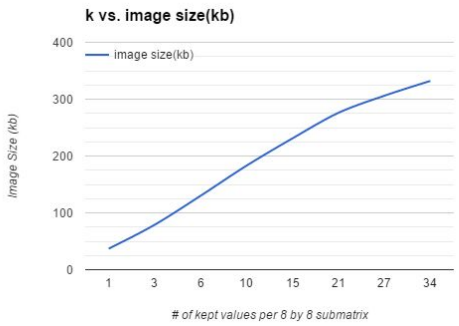
Data Analysis

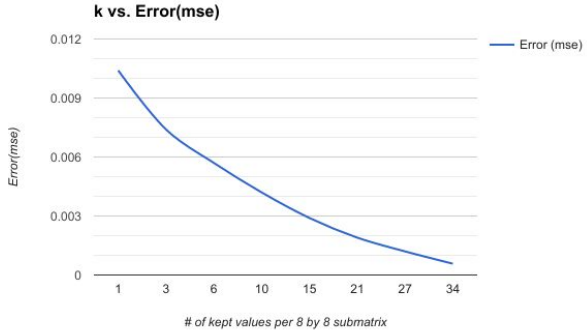
The following is the data we got from our DCT experiment:

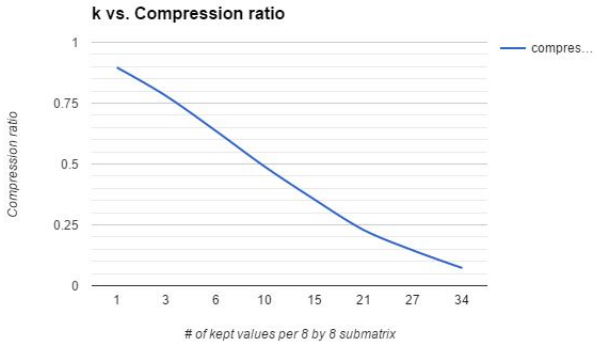
k	Image size (kb)	Error (mse)	Compression Ratio
1	37	.0104	.8966480447
3	79	.0074	.7793296089
6	130	.0057	.6368715084

10	183	.0042	.4888268156
15	231	.0029	.3547486034
21	276	.0019	.2290502793
27	306	.0012	.1452513966
34	332	.00057695	.07262569832

These are the graph we made to help visualize our data:





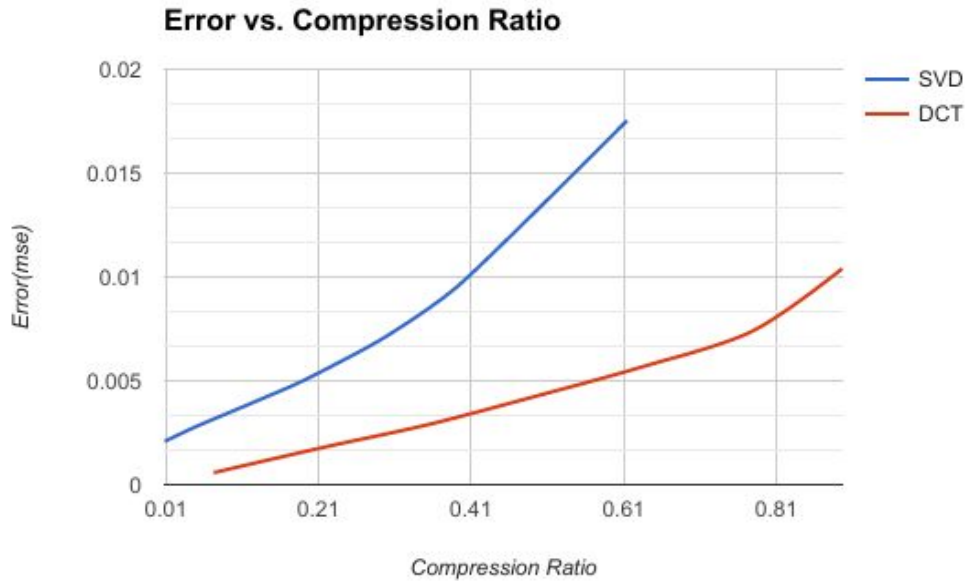


The first graph represents how the image file size changes as k (# of elements we keep in each 8x8 sub matrix) increases. As one could expect, each increase in k changes the file size by less than the previous element. This is due to each consecutive element influencing the image less than the previous one. The second graph compares the error between the compressed file and the original file. The error decreases as we increase k because we now have more terms that influence the image, and as expected the curve bottoms out due to each k having less impact than the previous one. Mean Square Error(mse) is the

way we measured the difference between two images. The third graph represents the compression ratio of DCT as k changes. We defined the compression ratio as 1 - the compressed file size over the original file size. This graph is the opposite of the image size graph because as k increases the file size increases which causes the ratio to 0.

Comparison of SVD and DCT

After experimenting with both SVD and DCT on the same image, they both gave us different data. Although k represented different things in both SVD and DCT, we compared the two by graphing the error vs. compression ratio of each.



Based on the graph, DCT is better at compressing images because it consistently has a lower error for each compression ratios. What this means is that the image quality of a DCT compressed image is much better than the image quality of a SVD compressed image of the same file size. This is because DCT is much more efficient at removing unnecessary values. In conclusion, DCT is a much better way to compress images.