

JPEG Compression Project

Math 214

Corinne Beemer

Hsien-Yu Lin

Hsiao-Ping (Sabrina) Lin

Ayush Mehra

October 16, 2015

Introduction

As the reliance on computers continues to grow, people need more efficient ways to store large amounts of data. Downloading digital images takes a considerable amount of time and also uses up a large amount of computer memory. There are several methods to compress images, and JPEG compression is one of them. The JPEG compression process is a lossy image compression algorithm that centers around the Discrete Cosine Transform. The DCT works by separating images into parts of different frequencies. A lossy compression represents the image in a way that used up less memory spaces without significant changes from the original data. The JPEG algorithm takes advantage of the nature that humans' eyes are insensitive to high frequencies. During the process of JPEG compression, high frequencies data points in the image are eliminated (Marcus).

In this project, we will treat images as 8×8 blocks and assign a matrix to each block. We will also use linear algebra techniques to maximize compression of the image. We first determine the invertibility of the provided matrices, and determine the effects of matrix multiplications of some 8×8 matrices. For the second part, we used MATLAB to determine how matrices will look like if they are displayed as images. Then we compute a matrix with different threshold values and recorded our discoveries, and MATLAB outputs will be included for illustration. For the next part we write a MATLAB code to compress images. The code will break an image into lots of 8×8 matrices and transform the matrices into a large matrix, then convert the large matrix back to an image. For the last part of the project, we discuss about sparse matrices and the efficiency of using sparse matrices for computer calculation. We conclude the project by discussing what we have learned.

Section 1

H1, H2, and H3 matrices are all invertible because they can all be reduced by Gaussian Elimination to the identity matrix in reduced row echelon form. Additionally, the determinant for all three matrices is nonzero, and this is another check for invertibility.

Multiplying an 8×8 matrix M by $H1$ essentially splits M into two halves. The top half consists of averages of pairs of rows in M . For example, $H1 \times M$ at position (1,1) is the average of (1,1) and (2,1) in M ; $H1 \times M$ at position (1,2) is the average of (3,1) and (4,1) in M and so on, up to position (4,1) in $H1 \times M$. This pattern for the top half continues for subsequent columns in M . The bottom half of $H1 \times M$ is the averages of the differences between pairs of rows in M , in the same pattern as the top half.

h1 =

0.5000	0.5000	0	0	0	0	0	0
0	0	0.5000	0.5000	0	0	0	0
0	0	0	0	0.5000	0.5000	0	0
0	0	0	0	0	0	0.5000	0.5000
0.5000	-0.5000	0	0	0	0	0	0
0	0	0.5000	-0.5000	0	0	0	0
0	0	0	0	0.5000	-0.5000	0	0
0	0	0	0	0	0	0.5000	-0.5000

M =

64	2	3	61	60	6	7	57
9	55	54	12	13	51	50	16
17	47	46	20	21	43	42	24
40	26	27	37	36	30	31	33
32	34	35	29	28	38	39	25
41	23	22	44	45	19	18	48
49	15	14	52	53	11	10	56
8	58	59	5	4	62	63	1

average of
(1,1) and (2,1)

>> h1*M

ans =

36.5000	28.5000	28.5000	36.5000	36.5000	28.5000	28.5000	36.5000
28.5000	36.5000	36.5000	28.5000	28.5000	36.5000	36.5000	28.5000
36.5000	28.5000	28.5000	36.5000	36.5000	28.5000	28.5000	36.5000
28.5000	36.5000	36.5000	28.5000	28.5000	36.5000	36.5000	28.5000
27.5000	-26.5000	-25.5000	24.5000	23.5000	-22.5000	-21.5000	20.5000
-11.5000	10.5000	9.5000	-8.5000	-7.5000	6.5000	5.5000	-4.5000
-4.5000	5.5000	6.5000	-7.5000	-8.5000	9.5000	10.5000	-11.5000
20.5000	-21.5000	-22.5000	23.5000	24.5000	-25.5000	-26.5000	27.5000

H2 and H3 are very similar to H1. They have almost the same effect as H1 except only on the top half of the matrix. For example, the first two rows of H2xM will represent the average of element pairs in the first two rows and the second two rows of M. The second two rows of H2xM will be the averages of the differences of the elements in the first two rows and second two rows of M. The rest of H2xM, the bottom half, will be the same as the bottom half of M because rows 5-8 of H2 are the identity matrix.

H3 has the same effect as H2, just another level deeper, affecting only the top quarter of M (first two rows). The first row in H3xM represents the averages of the elements pairs of first two rows in M and the second row in H3xM represents the differences of the averages of the second two rows in M. The last six rows of H3xM are the same as the last six rows of M because rows 3-8 in H3 are the identity matrix.

Together, H1, H2, and H3

h1 =

0.5000	0.5000	0	0	0	0	0	0
0	0	0.5000	0.5000	0	0	0	0
0	0	0	0	0.5000	0.5000	0	0
0	0	0	0	0	0	0.5000	0.5000
0.5000	-0.5000	0	0	0	0	0	0
0	0	0.5000	-0.5000	0	0	0	0
0	0	0	0	0.5000	-0.5000	0	0
0	0	0	0	0	0	0.5000	-0.5000

h2 =

0.5000	0.5000	0	0	0	0	0	0
0	0	0.5000	0.5000	0	0	0	0
0.5000	-0.5000	0	0	0	0	0	0
0	0	0.5000	-0.5000	0	0	0	0
0	0	0	0	1.0000	0	0	0
0	0	0	0	0	1.0000	0	0
0	0	0	0	0	0	1.0000	0
0	0	0	0	0	0	0	1.0000

>> h3

h3 =

0.5000	0.5000	0	0	0	0	0	0
0.5000	-0.5000	0	0	0	0	0	0
0	0	1.0000	0	0	0	0	0
0	0	0	1.0000	0	0	0	0
0	0	0	0	1.0000	0	0	0
0	0	0	0	0	1.0000	0	0
0	0	0	0	0	0	1.0000	0
0	0	0	0	0	0	0	1.0000

form H = H3xH2xH1 =

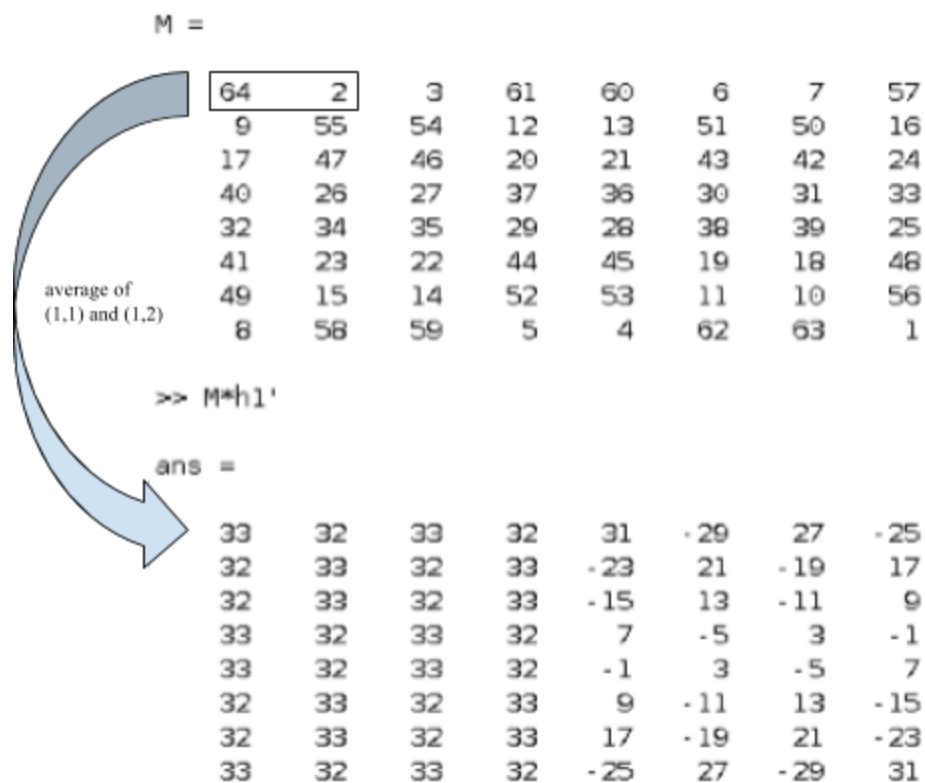
>> H = h3*h2*h1

H =

0.1250	0.1250	0.1250	0.1250	0.1250	0.1250	0.1250	0.1250
0.1250	0.1250	0.1250	0.1250	-0.1250	-0.1250	-0.1250	-0.1250
0.2500	0.2500	-0.2500	-0.2500	0	0	0	0
0	0	0	0	0.2500	0.2500	-0.2500	-0.2500
0.5000	-0.5000	0	0	0	0	0	0
0	0	0.5000	-0.5000	0	0	0	0
0	0	0	0	0.5000	-0.5000	0	0
0	0	0	0	0	0	0.5000	-0.5000

H is invertible, again verified by the fact that it can be reduced to the identity matrix in reduced row echelon form and that the determinant is nonzero.

The cross product of M and the transpose of H1 is much like the cross product of H1 and M however instead of affecting the top half of M, the transpose of H1 has the same effect on the left half of M. For example the first half columns of $M \times \text{transpose}(H1)$ represent averages of pairs of elements in columns in M in matching rows, and the second half of columns of $M \times \text{transpose}(H1)$ represent the averages of the differences between pairs of elements in columns in M in matching rows.



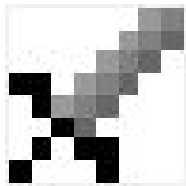
Section 2

A grayscale image represents intensities of a data within some range. MATLAB stores grayscale image as a matrix with each element corresponding to a pixel. The matrices can be stored into class of uint8, uint16, int16, single, or double. For single or double class, the intensity 0 represents black and 1 represents white. For class of uint8, uint16, or int16, the intensity black is shown as `intmin(class(I))` and white is shown as `intmax(class(I))`.

Plotting the matrix

$$M = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & .6 & .5 \\ 1 & 1 & 1 & 1 & 1 & .6 & .5 & .4 \\ 1 & 1 & 1 & 1 & .6 & .5 & .4 & 1 \\ 0 & 0 & 1 & .6 & .5 & .4 & 1 & 1 \\ 1 & 0 & .6 & .5 & .4 & 1 & 1 & 1 \\ 1 & 1 & 0 & .4 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

where values represent the intensity from black to white, as an image using the command `imshow` results in:

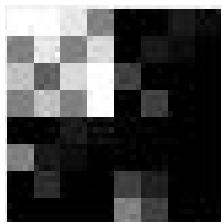


(M using imshow function)

$$M1 = H1 * M * (\text{transpose}(H1))$$

M1 =

1.0000	1.0000	0.9000	0.5000	0	0	0.1000	0.0500
0.5000	0.9000	0.5000	0.8500	0	0.1000	0.0500	-0.1500
0.7500	0.3750	0.8500	1.0000	0.2500	-0.0750	-0.1500	0
0.5000	0.7500	0.5000	1.0000	0	0.2500	-0.5000	0
0	0	0.1000	0.0500	0	0	-0.1000	0
0.5000	0.1000	0.0500	-0.1500	0	-0.1000	0	-0.1500
-0.2500	0.1750	-0.1500	0	0.2500	0.1250	-0.1500	0
0	-0.2500	0	0	0.5000	0.2500	0	0

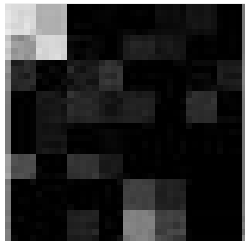


(M1 using imshow)

$$M2 = H2 * M1 * (\text{transpose}(H2))$$

M2 =

0.8500	0.6875	-0.1000	0.0125	0	0.0500	0.0750	-0.0500
0.5938	0.8375	0.0312	-0.1625	0.1250	0.0875	-0.3250	0
0.1500	0.0125	0.1000	0.1875	0	-0.0500	0.0250	0.1000
-0.0312	0.0875	0.1562	0.0875	0.1250	-0.1625	0.1750	0
0	0.0750	0	0.0250	0	0	-0.1000	0
0.3000	-0.0500	0.2000	0.1000	0	-0.1000	0	-0.1500
-0.0375	-0.0750	-0.2125	-0.0750	0.2500	0.1250	-0.1500	0
-0.1250	0	0.1250	0	0.5000	0.2500	0	0

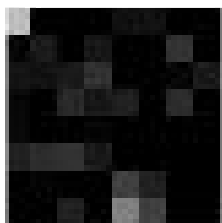


(M2 using imshow function)

M3 = H3*M2*(transpose(H3))

M3 =

0.7422	-0.0203	-0.0344	-0.0750	0.0625	0.0687	-0.1250	-0.0250
0.0266	0.1016	-0.0656	0.0875	-0.0625	-0.0188	0.2000	-0.0250
0.0813	0.0687	0.1000	0.1875	0	-0.0500	0.0250	0.1000
0.0281	-0.0594	0.1562	0.0875	0.1250	-0.1625	0.1750	0
0.0375	-0.0375	0	0.0250	0	0	-0.1000	0
0.1250	0.1750	0.2000	0.1000	0	-0.1000	0	-0.1500
-0.0563	0.0187	-0.2125	-0.0750	0.2500	0.1250	-0.1500	0
-0.0625	-0.0625	0.1250	0	0.5000	0.2500	0	0



(M3 using imshow function)

The final image above, representing M3, looks like the main image of the original sword-shaped figure mapped to the top left corner, leaving everything else close to 0 (black).

Section 3

For this part, we will choose multiple threshold values ε and compare the differences between them.



1. $\varepsilon_1 : 0$

Original	Compress	Decompress
		

M5=

8x8 double								
	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	0.6000	0.5000
2	1.0000	1.0000	1.0000	1.0000	1	0.6000	0.5000	0.4000
3	1	1	1	1	0.6000	0.5000	0.4000	1
4	5.5511e-...	5.5511e-...	1	0.6000	0.5000	0.4000	1.0000	1.0000
5	1.0000	-1.1102e...	0.6000	0.5000	0.4000	1	1.0000	1.0000
6	1	1	-5.5511e...	0.4000	1.0000	1.0000	1.0000	1.0000
7	1	0	1	0	0	1	1	1
8	0	1	1	1	0	1	1	1

2. $\varepsilon_2 : 0.1$

Original	Compress	Decompress
		

M5=

8x8 double								
	1	2	3	4	5	6	7	8
1	0.8438	0.8438	0.8438	0.8438	0.9031	0.7531	0.5531	0.3531
2	0.8438	0.8438	0.8438	0.8438	0.9031	0.7531	0.5531	0.3531
3	1.3438	1.3438	0.9438	0.9438	0.4781	0.3281	0.5281	1.0281
4	0.3437	0.3437	0.7437	0.7437	0.5781	0.4281	0.9281	0.8281
5	0.9594	0.2094	0.6594	0.7344	0.5438	1.1438	0.8438	0.8438
6	0.8844	1.1344	-0.0156	0.5594	0.8438	0.8438	0.8438	0.8438
7	0.9844	0.2344	1.0844	0.2594	0.3438	1.3438	0.8438	0.8438
8	-0.2656	0.9844	0.8344	1.0094	0.3438	1.3438	0.8438	0.8438

1. $\varepsilon_3 : 0.2$


Original	Compress	Decompress
		

M5=

[illegible]

2. $\epsilon_4 : 0.5$

Original	Compress	Decompress
		

$$M5 \equiv$$
 8x8 double[illegible]



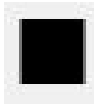
3. $\varepsilon_5 : 0.7$

Original	Compress	Decompress
		


M5=

 8x8 double[illegible]

4. $\epsilon_6 : 0.8$



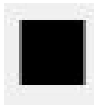
Original	Compress	Decompress
		

M5=

 8x8 double

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

4. $\epsilon_7 : 1$

Original	Compress	Decompress
		

M5=

 8x8 double

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

For M_4 , we chose a non-negative threshold value ε between 0 and 1, and let all the entries in the matrix whose absolute value is less than the threshold becomes 0. When we set the threshold ε equals to zero, there will not be any changes on the image. However, when we increase the the threshold value, some entries will be set to 0. Thus, when the image is compressed or decompressed, some identities and details of the original image will disappear.

When the threshold is zero, the image stays the same. When the threshold is 1, the image is compressed completely and the whole image will be black. We realized that the higher the threshold, the more the image is compressed. The more an image is compressed, the more quickly and efficiently it can be stored and transmitted.

As seen from the above tables of different threshold values, the less the value ε we picked, the higher the quality of the image we got. By contrast, the larger the threshold we chose, the more identity we lose during the process. The most important thing is that we have to choose the most appropriate value that can not only compress the image effectively, but also keep the quality of the image so that people can still identify the image.

Section 4

Using everything we have compiled in sections 1-3, MATLAB code can be written that effectively compresses and decompresses an image. The examples used in this report are all converted to grayscale however the the same logic can be applied to rgb pictures as well, they would just require compression on the three different layers of red, green and blue color intensities. The MATLAB code for this project was split into three files: imageCompression.m which reads the image file and converts it into smaller 8x8 matrices with loops, compress.m which contains logic from sections 1 and 2 to compress a passed in 8x8 matrix with the passed in threshold, and decompress.m which uses the logic from section 3 to decompress the passed in matrix.

imageCompression.m:

```

imageCompression.m  x  +
1  %Author: Ayush Mehra
2
3  RGB_Image = imread('bridge.png');
4  grayscale_image = rgb2gray(RGB_Image);
5
6  image = double(grayscale_image);
7  image = (1/255)*image; %convert from [1,255] range to [0,1] range
8
9  [rows, cols] = size(image);
10
11  numHorizontalSections = cols/8; % need these for proper traversal in loop
12  numVerticalSections = rows/8;
13
14  threshold = 0.0;
15
16  %compress loop
17  for x = 0:numVerticalSections-1 % loops every 8 cells from left to right
18      for y = 0:numHorizontalSections-1 % loops every 8 cells from top to bottom
19
20          image(x*8+1:x*8+8,y*8+1:y*8+8) = compress(image(x*8+1:x*8+8,y*8+1:y*8+8), threshold);
21
22          % gets correct range of matrix to compress and assigns compressed
23          % matrix back to self
24
25      end
26  end
27
28  %decompress loop
29  for x = 0:numVerticalSections-1 %same loop logic as above
30      for y = 0:numHorizontalSections-1
31
32          image(x*8+1:x*8+8,y*8+1:y*8+8) = decompress(image(x*8+1:x*8+8,y*8+1:y*8+8));
33          % gets correct range of matrix to decompress and assigns decompressed
34          % matrix back to self
35
36      end
37  end
38
39  imshow(image) % shows decompressed image
40

```

compress.m:

```

compress.m  x  +
1  %Author: Ayush Mehra
2
3  function [compressed] = compress(m,threshold)
4
5  %create h1, h2, h3 with given values
6  h1 = [1/2 1/2 0 0 0 0 0 0; 0 0 1/2 1/2 0 0 0 0; 0 0 0 0 1/2 1/2 0 0; 0 0 0 0 0 0 1/2 1/2; 1/2 -1/2 0 0
7
8  h2 = [1/2 1/2 0 0 0 0 0 0; 0 0 1/2 1/2 0 0 0 0; 0.5 -0.5 0 0 0 0 0 0; 0 0 0.5 -0.5 0 0 0 0; 0 0 0 0 1 0
9
10 h3 = [1/2 1/2 0 0 0 0 0 0; 1/2 -1/2 0 0 0 0 0 0; 0 0 1 0 0 0 0 0; 0 0 0 1 0 0 0 0; 0 0 0 0 1 0 0 0; 0 0 0
11
12 m1 = h1*m*transpose(h1); % created using formulas from spec
13
14 m2 = h2*m1*transpose(h2);
15
16 m3 = h3*m2*transpose(h3);
17
18 m4 = m3;
19
20 m4(m4>=(-1*threshold) & m4<=threshold) = 0;
21
22 compressed = m4;
23

```


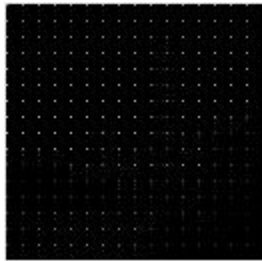


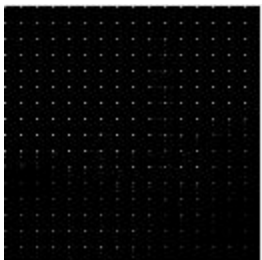


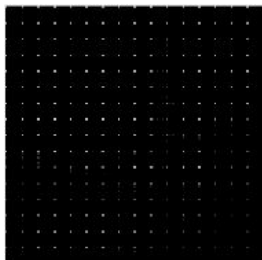

decompress.m:


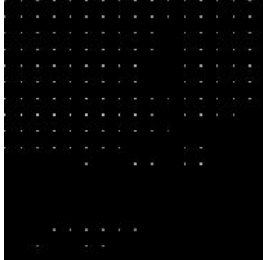
















```





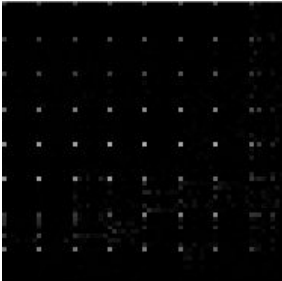



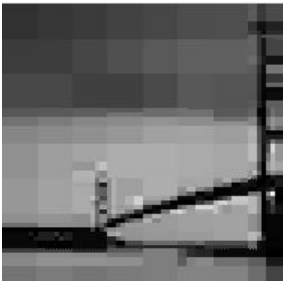




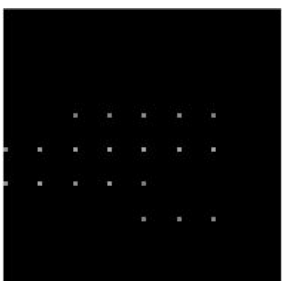
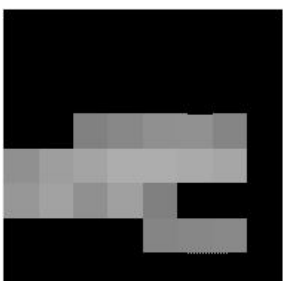
1  %author: Ayush Mehra
2
3  function [decompressed] = decompress(m)
4
5  %create h1, h2, h3 with given values
6  h1 = [1/2 1/2 0 0 0 0 0 0; 0 0 1/2 1/2 0 0 0 0; 0 0 0 0 1/2 1/2 0 0; 0 0 0 0 0 1/2 1/2; 1/2 -1/2 0 0
7
8  h2 = [1/2 1/2 0 0 0 0 0 0; 0 0 1/2 1/2 0 0 0 0; 0.5 -0.5 0 0 0 0 0 0; 0 0 0.5 -0.5 0 0 0 0; 0 0 0 0 1 0
9
10 h3 = [1/2 1/2 0 0 0 0 0 0; 1/2 -1/2 0 0 0 0 0 0; 0 0 1 0 0 0 0 0; 0 0 0 1 0 0 0 0; 0 0 0 0 1 0 0 0; 0 0 0
11
12 h = h3*h2*h1; % created using formulas from spec
13
14 decompressed = (h^-1)*m*transpose((h^-1));
15



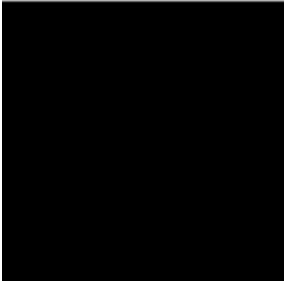
```

The following table shows resulting compression and decompression of a few different images at different threshold values.

Threshold value	Original Image	Compressed Image	Decompressed
0.0			
0.05			
0.1			

0.5			
1.0			
0.0			
0.05			
0.1			
0.5			

1.0			
0.0			
0.05			
0.1			
0.5			

1.0			
-----	---	--	---

It is interesting to note that the second image, due to its higher resolution (500x286) has less loss during compression at low threshold values than lower resolution pictures such as the first and third one. Additionally, from this table it is clear that at threshold 0, there is 0 loss of detail during compression and at threshold 1, the image is completely black so all detail and quality is lost.

Section 5

A sparse matrix is a matrix in which most of the elements in it are zero. There are several computational advantages of sparse matrices, such as memory management and computational efficiency. Using sparse matrices to store data can not only speed up the process but also save a great amount of memory on a computer. The **sparse** attribute in MATLAB allows people to assign a 2D matrix that is composed of double or logical elements. When operating in MATLAB, the **sparse** attribute stores only the nonzero elements and the indices of the matrix, and eliminates the operations on zero elements to reduce computation time. This concept of using sparse matrices to store images efficiently is even more effective for large matrices with a higher percentage of zero-valued elements.

In terms of computational efficiency, operations with sparse matrices do not perform unnecessary low-level arithmetic. This leads to improvements in the operation time for programs with significant amounts of sparse data ("Documentation").

MATLAB code:

```
Matrix_original = magic(1100);
Matrix_original(Matrix_original > 50) = 0;
Matrix_sparse = sparse (Matrix_original);
```

%The steps above creates a 1100 by 1100 matrix, sets the elements to zero if the elements are bigger than 50, and create a sparse matrix same as the original matrix.

whos

Name	Size	storage(bytes)	type
Matrix_original	1100x1100	9680000	double
Matrix_sparse	1100x1100	5004	double

%The **whos** attribute in MATLAB can check the storage of a matrix. While listing out the original matrix and sparse matrix, it is clear that the number of bytes used in the sparse matrix is significantly less ("Documentation").

The method of image compression in this project takes an image and converts it to a matrix of numbers. This matrix assigns a number to each pixel corresponding to where the pixel falls in a range of black to white intensities. In order to compress the image, the image matrix is divided into 8x8 blocks which can then be manipulated using linear algebra functions to form matrices with many zero entries. These matrices are referred to as sparse matrices. The zero elements are created when sections of the original matrix have no variation. Although we cannot expect that many of the elements will be zero, we can set a positive threshold value, as seen in section 3, to compare the numbers of the the transformed matrix to. Any numbers less than this threshold will then be reset to zero. The reasoning behind this is that numbers which fall between 0 and the threshold represent areas of relatively low variation and therefore reassigning these values to zero will not result in the loss of much detail. These sparse matrices take up significantly less space to store and transmit which also means they can be transmitted faster ("Application to Image Compression").

Conclusion

In this project, we used MATLAB to learn about block multiplication, image compression, and sparse matrices. We first researched and learned about the concept and usefulness of JPEG compression, then we coded on MATLAB to figure out the details step by step.

At first, we learned that the invertibility of the three matrices can be determined in its reduced-row echelon form by Gaussian-Elimination method. Next, we discovered some patterns when we performed block multiplication of some 8 x 8 matrices. When multiplying a matrix on the left by H1 matrix, the top and bottom half of H1xM is the averages of the differences between pairs of rows in M. When multiplying a matrix on the left by the transpose of H1 matrix, the effects are the same as the H1 x M block multiplication except that the effects are on the left half of the matrix M rather than on the top and bottom halves of M. Using MATLAB, we again found out that the block multiplication of $H = H_3H_2H_1$ is invertible.

In part two, we learned that MATLAB stores grayscale image as a matrix with each element corresponding to a pixel. We also used MATLAB to show the image of the matrix M, which looks like a sword, as well as some other transformations of M. We realized that the final image of M3 looks like the original sword-shaped image mapping to the top left corner, while the rest of the image filled with black.

In the third section, we compared the differences between multiple threshold values ϵ . We learned that the less the threshold is, the higher the quality of the image will be. These findings were illustrated in the graphics provided. We also realized that in order to compress image but still remain its quality at the same time, it is important for us to determine what the most ideal value is.

For part four, we wrote a MATLAB code that compresses images. We found out by testing an image that when the threshold equals to zero, the image does not change; when the threshold equals to one, the image becomes all black. The concept is the same as the previous section, but instead we proved it by compressing and decompressing an image.

Last but not least, we did some research on the applications of sparse matrices. We learned that sparse matrices can store images more efficiently and using less memory spaces because the **sparse** attribute in MATLAB stores only the nonzero elements of the matrix and eliminates the rest. We also understood this concept by writing the codes on MATLAB to prove that sparse matrices used up significantly less bytes than original matrices. Overall, our this project allowed us to gain a deeper understanding of sparse matrices and their application for image compression.

Citations

- “Application to Image Compression.” 15 Oct. 2015. <http://aix1.uottawa.ca/~jkhoury/haar.htm>
- Cabeen, Ken, Peter Gent. “Image Compression and the Discrete Cosine Function.” 15 Oct. 2015. <http://www.lokminglui.com/dct.pdf>
- "Documentation." *Computational Advantages of Sparse Matrices*. MathWorks, 1994. Web. 15 Oct. 2015. <http://www.mathworks.com/help/matlab/math/computational-advantages-of-sparse-matrices.html>
- Marcus, Matt. “JPEG Image Compression.” 2014. 15 Oct. 2015. <http://www.lokminglui.com/dct.pdf>