Image Manipulation
APPM 2360
Project 2

Sanjay Kumar Keshava, Ravin Chowdhury, and Grace Cowan

23 November 2020

Introduction

You are a student in the CU business school and you have recently come up with a great idea for a new photo editing application for mobile phones. You hope to employ MATLAB and use image matrix manipulation to edit photos. You also wish to create a scheme that allows image compression. However, you have no background in MATLAB programming, and matrices of all sizes tend to confuse you! Worry not, using the fundamental principles of linear algebra along with our MATLAB programming skills, we will help you develop your photo editing application! In particular, using MATLAB's image manipulation capabilites, we will custom design transformation matrices to help translate and rearrange different portions of images. In addition, we will create ways of changing the brightness of images. Furthermore, we will write code that uses the discrete sine transform to compress images, thereby making it easy and efficient to move image files between different mobile phones. Finally, as a bonus, we will implement a feature that allows images to be blurred.

1 Image Manipulation

MATLAB is our programming environment of choice for the development of the photo editing application. When an image is read in by MATLAB, it is converted into an integer matrix with a bit depth of 8. This means the integers in the image matrix can range from 0 to 255. The MATLAB function *imread* will be used to read in images. After reading in an image, we will be transforming the corresponding image matrix with mathematical matrix operations. Before doing this we need to convert the integers in our matrix to floating-point values. To do this, we will use the *double* function on MATLAB. After image modification, we need to convert back to an 8 bit integer matrix before being able to view our image. To convert back, we will employ MATLAB's *uint8* function, and to view images we will use the *imagesc* function. Once we are satisfied with

how our modified image looks, we can use MATLAB's *imwrite* function to save our image as a *.jpg* file. For the sake of simplicity, we will limit our operations to act only on grayscale images. However, it is worth mentioning that all our functions can be expanded to act on tensors of rank 3 by treating each colour channel as a separate matrix. But for now we will add and normalize the RGB layers to convert our colored images into grayscale images before performing any modifications. In the grayscale matrix, the entry at a given location tells us the intensity of the pixel at that location in the image (0 is black and 255 is white). After conversion, the resulting grayscale matrix can be viewed on MATLAB by enabling the *colormap("gray")* setting.

To begin our journey, we will look at an image of the Flatirons (matrix $\mathbf{F}$) as seen from the CU campus, and an image of the furry (but extremely dangerous) cat known as Albus (matrix $\mathbf{C}$). Using the MATLAB functions mentioned earlier, we read in both the images, converted them to grayscale, viewed them as MATLAB figures and saved them.[1]

Figure 1: The Flatirons (matrix $\mathbf{F}$)

The above image (Figure 1) shows a grayscale photo of the Flatirons. On the next page, we can see the grayscale photo of Albus the cat (Figure 2). As a start, we decided to try and experiment with the exposure of an image, and we tried to make Albus look more "whited out". Since the numerical value stored at each location in the grayscale matrix $\mathbf{C}$ represents the image pixel intensity at that location, we multiplied each matrix entry by 2 to try and make the image look "twice" as bright.[2] A "whited out" version of Albus the cat (matrix $\mathbf{W}$ where $\mathbf{W} = 2\mathbf{C}$) is shown on the next page (Figure 3). We can see that after this matrix manipulation, the exposure of the image has increased significantly.

[1] See Appendix A for the associated code
[2] See Appendix A for the associated code

Figure 2: Albus the cat (matrix $\mathbf{C}$)

Figure 3: "Whited out" image of Albus (matrix $\mathbf{W}$)

After fooling around with image exposure for a while longer, we decided to shift our attention to image manipulation using transformation matrices. To start small, we thought we would try to work with a simple 4×4 matrix. First, we studied column and row swapping. In particular, we tried to find a transformation matrix that swaps the leftmost and rightmost columns of a 4×4 matrix. We used MATLAB's *magic* function to generate the 4×4 magic square matrix **M** (shown below).[3]

$$\begin{bmatrix} 6 & 2 & 3 & 13 \\ 5 & 11 & 10 & 8 \\ 9 & 7 & 6 & 12 \\ 4 & 14 & 15 & 1 \end{bmatrix}$$

After experimenting on pen and paper, we found that, to switch the leftmost and rightmost columns of M, we require matrix **E** (shown below).

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

E was obtained by swapping the the leftmost and rightmost columns of the 4th order identity matrix. We found by matrix multiplication that **ME** (shown below) had the desired property.[4]

$$\begin{bmatrix} 13 & 2 & 3 & 16 \\ 8 & 11 & 10 & 5 \\ 12 & 7 & 6 & 9 \\ 1 & 14 & 15 & 4 \end{bmatrix}$$

In fact, multiplying any 4 x 4 square matrix **A** by **E** from the right side will give the matrix **AE** which is equivalent to the matrix obtained by swapping the leftmost and rightmost columns of **A**. It is worth noticing a couple of things. First, multiplying **A** by **E** from the left side gives us **EA** which is the same as the matrix obtained by switching the topmost and bottom-most rows of **A**. Next, swapping the topmost and bottmom-most rows of the 4th order identity matrix also gives us the matrix **E**. These properties will be useful to us as we proceed with our investigation of transformation matrices.

[3]See Appendix A for the associated code
[4]See Appendix A for the associated code

Using the insight gained from our experiment, we decided to apply a transformation matrix to an image. In particular, we revisited the non-square image of the Flatirons (Figure 1) given by **F** and shifted all the pixels horizontally by 240 units to the right. Our approach was essentially the same as before: we started out with the identity matrix and rearranged its columns in exactly same way that we wished to rearrange the columns of our image matrix. This gave us the horizontal transformation matrix **H** (Figure 4). Since the image matrix was non-square, we made sure that order of the identity matrix used was equal to the number of columns of the image matrix. Next, we multiplied the transformation matrix with the image matrix from the right side to find **FH** (Figure 5).[5] We used the *spy* function to view our transformation matrix.

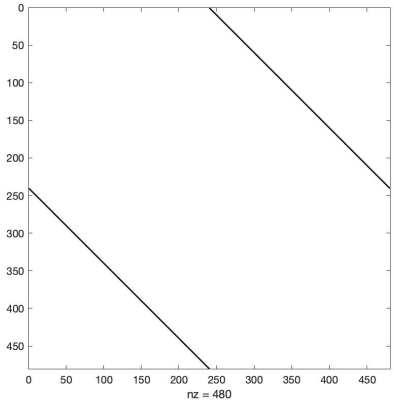


Figure 4: Matrix **H** used to perform a horizontal shift of 240 pixels

Figure 5: The Flatirons after a horizontal shift of 240 pixels (matrix **FH**)

[5]See Appendix A for the associated code

To push things further, we went back our the original image of the Flatirons (Figure 1, matrix **F**) and apply both horizontal and vertical transformations. In particular, as before, we shifted the image horizontally by 240 pixels to the right, and additionally, we shifted it vertically by 100 pixels downwards. We used the same matrix **H** as earlier (Figure 4) to perform the horizontal shift. For the vertical shift, we generated an identity matrix with order equal to the number of rows of our image matrix. Then we shifted the rows of this matrix in the same way that we wished to shift the rows of our image matrix, and this gave us **V**, the vertical transformation matrix (Figure 6).[6] To apply both transformations, we computed **VFH** (Figure 7). From the previous experiments with 4×4 matrices, one might expect **H** and **V** to be identical, but they are not, and this is because **F** is non-square.

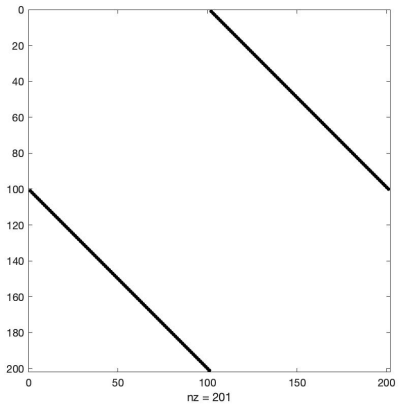


Figure 6: Matrix **V** used to perform a vertical shift of 100 pixels

Figure 7: The Flatirons after a horizontal shift of 240 pixels and a vertical shift of 100 pixels (matrix **VFH**). We notice that $\mathbf{VFH} = \mathbf{V}(\mathbf{FH}) = (\mathbf{VF})\mathbf{H}$

[6]See Appendix A for the associated code

After all this, we were still unsatisfied, and so, we decided to delve deeper into the exciting world of transformation matrices. We also decided to pay our old friend Albus the cat (Figure 2, matrix **C**) a visit and play tricks on him. To make Albus feel dizzy, we first used the transformation matrix **U** to flip him upside down! To find **U**, we started with an identity matrix with order equal to the number of rows in matrix **C**. Then, we reversed the order of elements in each row of this identity matrix to find **U** (Figure 8). Next we simply computed **UC** to flip Albus upside down (Figure 9).[7]

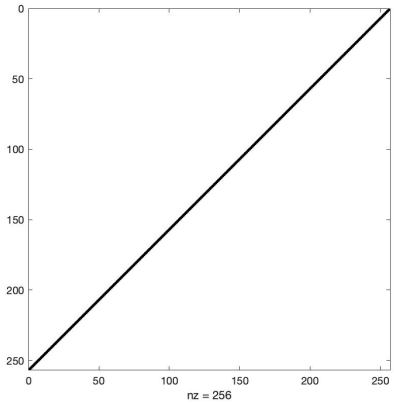


Figure 8: Matrix **U** to flip an image upside down

Figure 9: Upside down image of Albus (matrix **UC**)

[7]See Appendix A for the associated code

Next, To confuse Albus, we rotated him by 90 degrees without any explicit use of transformation matrices! This was done by simply finding the transpose of matrix $\mathbf{C}$ (Figure 10).[8] Overall, this makes sense, because when we take the transpose of a matrix, each row in the original matrix gets converted to a column in the transpose matrix, and this corresponds to a 90 degree rotation.

Figure 10: Albus the cat after a 90 degree rotation (matrix ($\mathbf{C}^T$))

Finally, to test our understanding, we decided to take on a challenge to unscramble an image of a portion of the Flatirons and combine it with another portion to recover the complete image of the Flatirons (Figure 1, matrix $\mathbf{F}$). To begin, we studied the scrambled image (Figure 11, matrix $\mathbf{L}$).

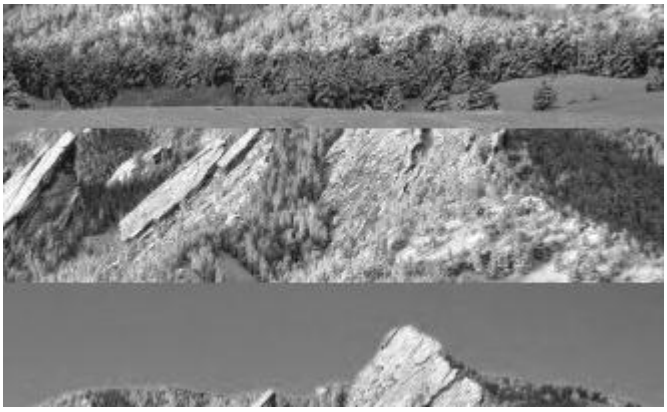


Figure 11: Scrambled image of a portion of the Flatirons (matrix **L)**

[8] See Appendix A for the associated code

We used a combination of multiple matrix manipulations along with some trial and error to unscramble this image. First, we divided the rows of **L** into 3 regions: (a) first row - 62nd row, (b) 63rd row - 139th row, and (c) 140th row - last row. As visible, the current order of the row regions in **L** is (a), (b), (c), while the desired order is (c), (b), (a). So to swap the regions, we first generated an identity matrix with order equal to the number of rows in **L**. Next, we swapped the row regions of this matrix in the same way that we wished to swap the row regions of **L**, and this gave us the transformation matrix **R** (Figure 12).[9]

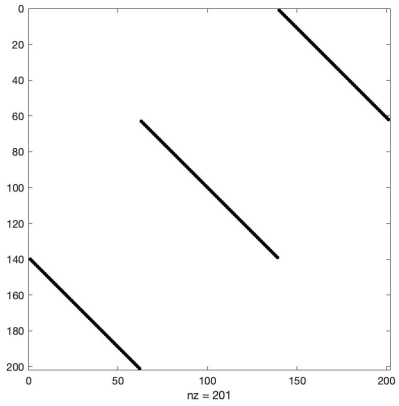


Figure 12: Matrix **R** used to unscramble an image

We then computed **LR** to unscramble **L**, and as we had hoped, the desired result was obtained (Figure 12).

Figure 13: Unscrambled portion of the Flatirons (matrix **LR**)

[9]See Appendix A for the associated code

Finally, we studied the remaining portion of the Flatirons (Figure 14, matrix **P**) and noticed that we simply needed to "place it beside" the image given by **LR** to recover the original image of the Flatirons. To do this, we simply used matrix concatenation to "place" **P** to the left of **LR** (Figure 15).[10]

Figure 14: Remaining portion of the Flatirons (matrix **P**)

Figure 15: Recovered image of the Flatirons (concatenation of **P** and **LR**)

We can see (by comparing Figure 15 to Figure 1) that we have indeed been able to recover the original image of the Flatirons. The only difference is the exposure/brightness which is different because the source images used in the recovery process appear to have been brighter overall. Phew! That was tiring, but there is a lot left to learn, and next, we move on to study image compression using the discrete sine transform.

[10] See Appendix A for the associated code

2 The Discrete Sine Transform

Sometimes, when we come across images that are noisy, or contain digital information at different scales and we want to extract specific features from the image matrix, a function decomposing transformation can come in handy. The most popular transform is of course the Fourier transform, which is used in audio analysis to decompose a signal into a sum of sinusoidal functions. Suppose you were recording an instrument playing at a frequency of 5 Hz, but in the background a high pitched noise at a frequency of 100Hz is audible. The easiest way to remove the noise is to take a Fourier transform of the recording. The resultant function will have two peaks, one corresponding to 5Hz and the other to 100Hz. You can easily remove the peak at 100Hz and take an inverse Fourier transform. The resultant function will be the recording of the instrument with the high pitched noise removed.

A similar method can be used to extract particular features from images. An image can be thought of as a 2D matrix of frequencies where the pixels range from dark to bright based on their intensity. In some regions intensities change quickly, while in other regions there are longer gradients, just like the frequencies in sound. Noise in images can be extracted by the same method - by taking a Fourier transform of the image and cutting out unwanted frequencies. The transformation we will use in our analysis is not the Fourier transform, but a closely related cousin called the sine transform (shown below).

$$F(s) = \int_{-\infty}^{\infty} f(t) \sin(2\pi st)\, \mathrm{d}t.$$

Since we want to use this for a 2D quantized array of pixels, we use the discrete variant in array form. We do this by defining a square transformation matrix $\mathbf{S}$ of size n×n which is known as the discrete sine transform (DST) matrix (shown below). Using MATLAB, we wrote a function that returns the DST matrix of any desired size.[11]

$$s_{i,j} = \sqrt{\frac{2}{n}} \sin[\frac{\pi(i-\frac{1}{2})(j-\frac{1}{2})}{n}]$$

The discrete sine transform of an image matrix $\mathbf{X}_g$ is defined as:

$$\mathbf{Y} = \mathbf{S}\mathbf{X}_g\mathbf{S}$$

Looking at the matrix S, we notice that it is symmetric, and the entry at each location (i, j) is the same as the entry at (j, i). This implies that transpose of the matrix $\mathbf{S}$ is identical to $\mathbf{S}$. Furthermore, we can notice that $\mathbf{S}$ is the also its own inverse matrix. To explore further, we created a DST matrix of size n = 5 to verify this property (Figure 16).[12]

[11] See Appendix A for the associated code
[12] See Appendix A for the associated code

$$\begin{bmatrix} 0.099 & 0.287 & 0.447 & 0.564 & 0.625 \\ 0.287 & 0.625 & 0.447 & -0.099 & -0.564 \\ 0.447 & 0.447 & -0.447 & -0.447 & 0.447 \\ 0.564 & -0.099 & -0.447 & 0.625 & -0.287 \\ 0.625 & -0.564 & 0.447 & -0.287 & 0.099 \end{bmatrix}$$

Figure 16: Matrix $\mathbf{S}$ (5x5)

Next, we computed $\mathbf{SS}$ and got the 5th order identity matrix after rounding to 10 decimal places (Figure 17). This shows that $\mathbf{S} = \mathbf{S}^{-1}$.

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

Figure 17: Matrix $\mathbf{I}_5 = \mathbf{SS}$

A sine transform of an image contains all the original information, and it can be an efficiently used to reduce the information content of an image without causing visible changes by a process called compression. Empirically, it is known that discarding the high frequency components of a transformed image causes the least observable loss in image quality. Since frequencies increase as we go towards the bottom right of the transformed matrix, we can start there and make elements of the matrix 0 to begin the process of image compression. After we have reduced the information content of our matrix, we need to transform it back using an inverse transformation. Mathematically, since $\mathbf{S}$ is its own inverse, we guessed that the inverse discrete sine transform is the same as the discrete sine transform (shown below).

$$\mathbf{X}_g = \mathbf{SYS}$$

To check whether this is correct, we created a randomly generated 5x5 matrix $\mathbf{J}$ (Figure 18) and tried to apply the DST two times in a row.[13]

$$\begin{bmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{bmatrix}$$

Figure 18: Matrix $\mathbf{J}$

[13] See Appendix A for the associated code

Applying a Discrete Sine Transform on **J** using the matrix **S** (Figure 16), we got the matrix **K** (Figure 19).

$$\begin{bmatrix} 54.2191 & 21.3577 & 4.1257 & 5.0813 & 10.2072 \\ 20.4888 & -9.8653 & 10.1520 & 15.2195 & 2.4624 \\ 5.4671 & 10.0245 & 19.4000 & 1.1662 & 6.0745 \\ 5.4205 & 11.2356 & 5.4241 & 9.4102 & -4.6220 \\ 9.8479 & 1.5705 & 9.5464 & -5.2003 & -8.1640 \end{bmatrix}$$

Figure 19: Matrix **K**

Finally, we applied the Discrete Sine Transform on **K** (Figure 19), to get Matrix **O** (Figure 20). This is the same as the original matrix **J** (Figure 18), hence our guess was correct.

$$\begin{bmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{bmatrix}$$

Figure 20: Matrix **O**

Now let us apply our new-found knowledge to compress the image of Albus the cat.[14] We start by applying a DST on the original image of Albus (Figure 21).

(a) Original image (b) Transformed Image

Figure 21: Effect of the Discrete Sine Transform

[14] See Appendix A for the associated code

We can see that the transformed image is basically just a black square, and it tells us nothing. However, it has all the information from the image of Albus contained within it.

We can compress our image by deleting some of high frequency components from this transformed matrix (Figure 21(b)) before applying the inverse DST. To quantify the compression of our image we introduce a variable p between 0 and 1. When p is 1, it corresponds to no compression, and when it is zero it corresponds to the maximum possible compression. Next, we use the following algorithm: if $i + j > 2pn$, we set the entry at location (i, j) in the transformed matrix to 0. We tested the effect of doing this for different values for p to see what values work the best to reduce file size but retain image quality.

To quantify the compression, we defined the compression ratio (CR) for a compressed image as follows (shown below).

$$\text{CR} = \frac{\text{uncompressed size}}{\text{compressed size}}$$

The sizes mentioned in the equation are the number of non-zero entries in the transformed matrix of the image (before and after applying the algorithm). Next we began trying different values of p to see what happened We noticed that it takes a long time for there to be any noticeable change in the file size. Decreasing of p from 1.0 to 0.6 causes nearly no change in the file-size (it only dropped from 10.2 to 10.0 (kilobytes)). If we compare the two images, there is no discernible change to the image quality (Figure 22). The value of the CR for p = 1.0 is 1.0000, and for p = 0.6 it is 1.4753 (to 4dp).

(a) Original image (p = 1.0) (b) Compressed image (p = 0.6)

Figure 22: Compression to p = 0.6

Next, we move on to examine the cases for p = 0.5 and p = 0.4 (Figure 23).

(a) Compressed image (p = 0.5)

(b) Compressed image (p = 0.4)

Figure 23: Compression p = 0.5 and p = 0.4

With p values of 0.5 and 0.4, The corresponding CR values are 2.0078 and 3.1651 respectively. we finally start to see small changes to the image such as an increased 'fuzziness', but at the scale of the image in this report, the changes are nearly indistinguishable from the original (Figure 22(a)).

(a) Compressed image (p = 0.3)

(b) Compressed image (p = 0.2)

Figure 24: Compression p = 0.3 and p = 0.2

Finally, we start to see proper changes to the image (Figure 24). The cat's face is noticeably fuzzier than the original for p = 0.3 and we also see some faint sinusoidal noise at the very top of the image. The CR value here is 5.6356. All these issues in the image quality grow even more pronounced as we go to p = 0.2 (CR = 12.7205). The cat's face and arms are extremely blurry and the black bars at the top are much darker. At this point, it is not worth compressing the image further. For the sake of science, however, we must go on.

Figure 25: Compressed Image p = 0.1

At this point, everything is just blotches and our cat Albus has seen better days. With a CR of 51.3605 when p = 0.1, this image is much more compressed but has lost too much necessary information to be useful (Figure 25). This tells us that we need a more efficient method to compress beyond CR ≈ 6.

We realize that we can make p quite small before effects on the image quality become noticeable. This is likely because the image quality is not a linear function of p. Judging from the information above, in our opinion, the good balance between image size and quality is at $p = 0.3$. To summarize the key information, we created a table of compression ratios for some values of p (shown below).

p	CR
0.5	2.0078
0.3	5.6356
0.1	51.3605

Finally, judging purely based on the value of the CR and the quality of the resultant image, in our opinion, $p = 0.2$ is the best parameter value. The resultant CR is much larger than for $p = 0.3$ (12.7205 as compared to 5.6356), and the image (Figure 24) is not as bad as $p = 0.1$, (Figure 25).

It has been an interesting journey, but we are not done yet. With all that we have learned, we now need to conclude with a summary of our findings.

3 Conclusion

Using our matrix manipulation techniques we were able to transform images in multiple ways after converting them to gray-scale. We changed their exposure, shifted them both horizontally and vertically, flipped them, and even managed to rotate them. Since an image is fundamentally just a matrix of pixels we found that we could perform all these modifications by multiplying the image matrix using different methods. Since the value stored at each pixel location in an image matrix represents the gray-scale intensity, scalar multiplication made an image brighter or darker, thereby changing the exposure. By swapping the columns or rows of identity matrices and then multiplying them with our images, we found that we could swap the corresponding columns or rows in our images. We were also able to undo these changes using a similar process. We found for square matrices that the transformation matrix for the column switch was identical to the transformation matrix for the row switch. We were also able to flip an image of Albus the cat upside down by using an identity matrix that had its rows in reverse order. Finally, because the transpose of a matrix switches the rows and columns, we were able to perform a 90 degree rotation on the image of Albus by transposing the corresponding image matrix. After we were done exploring different types of transformation matrices, we decided to take images and improve them by cancelling background noise and compressing them to decrease their information content. This was done by using a similar system as audio noise reduction which employs the Fourier transform. Since we were editing 2D arrays, we used the Discrete Sine Transform (DST) with the help of matrices. We found that the DST matrix was symmetric, and had the special property that it was identical to its inverse matrix. This allowed us to transform an image using the DST matrix, delete unwanted content, and then use the same matrix to transform it back to obtain an image with some features changed. By removing high frequency elements in the transformed matrix of an image, we were able to compress it while still maintaining its quality. The quality of a compressed image depended on the level of compression, and this was decided by the variable p. We quantified the level of compression using the compression ratio (CR). Through experimentation, we decided that a p value of 0.2 resulted in a good balance between image quality and compression. We also tabulated the CR values for some key p values to show how the CR changed with decreasing p.

4 Appendices

A MATLAB CODE

The following code (shown on the next page) was written on MATLAB and used to generate the images and data that have been used in this write-up. The table of contents has details on the location of different parts of the code. In addition to this, the code has been commented extensively.

APPM 2360 - PROJECT 2 %%

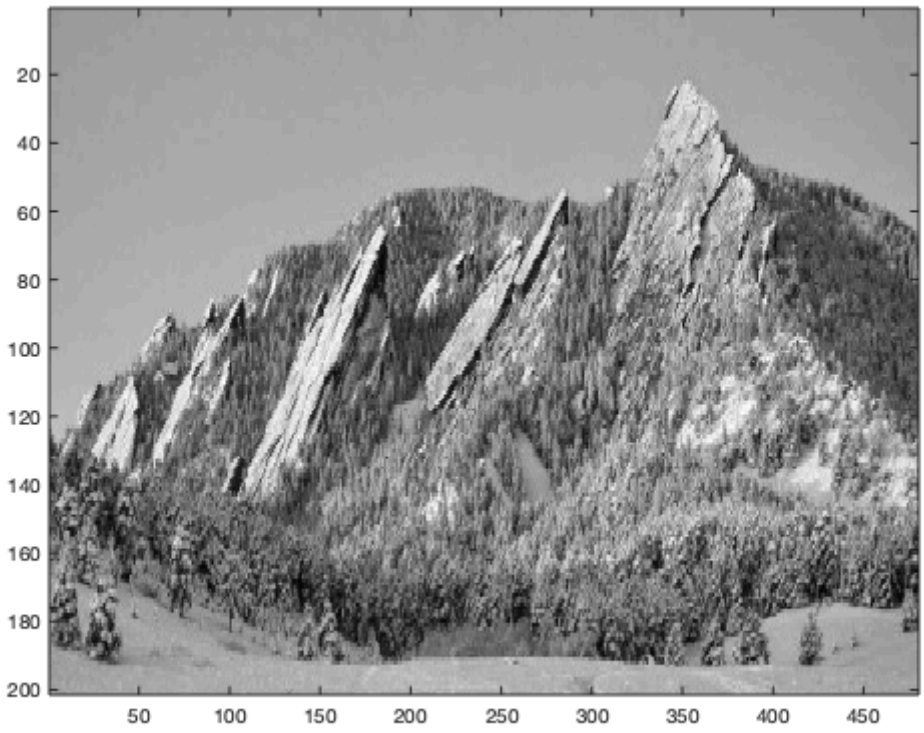
Table of Contents

PART 1

```
photo1 = 'photo1.jpg'; photo2 = 'photo2.jpg'; mount1 = 'mount1.jpg';
 mount2 = 'mount2.jpg';
```

5.11

```
p1_gs = gs_matrix(photo1); p2_gs = gs_matrix(photo2); m1_gs =
 imread(mount1); m2_gs = imread(mount2);
figure(1); imagesc(uint8(p1_gs)); colormap('gray');
 imwrite(uint8(p1_gs), '(5.11a).jpg');
figure(2); imagesc(uint8(p2_gs)); colormap('gray');
 imwrite(uint8(p2_gs), '(5.11b).jpg');
```


20
40
60
80
100
120
140
160
180
200
50
100
150
200
250
300
350
400
450


50
100
150
200
250
50
100
150
200
250

5.12

```
p2_gs_exposed = exposure(p2_gs, 2); % basically multiplies every pixel
 by a scalar to increase intensity
figure(3); imagesc(uint8(p2_gs_exposed));
colormap('gray'); imwrite(uint8(p2_gs_exposed), '(5.12).jpg');
```

5.13

```
A = magic(4); E = swap_matrix(A, 1, 4);
disp('This is A'); disp(A);
disp('This is E'); disp(E);
disp('This is C = A*E'); C = A*E; disp(C);
% E*A will not work. In this case, E*A will swap first and last rows
 of A.
% All this along with the matrices needs to be type out on Latex.
```

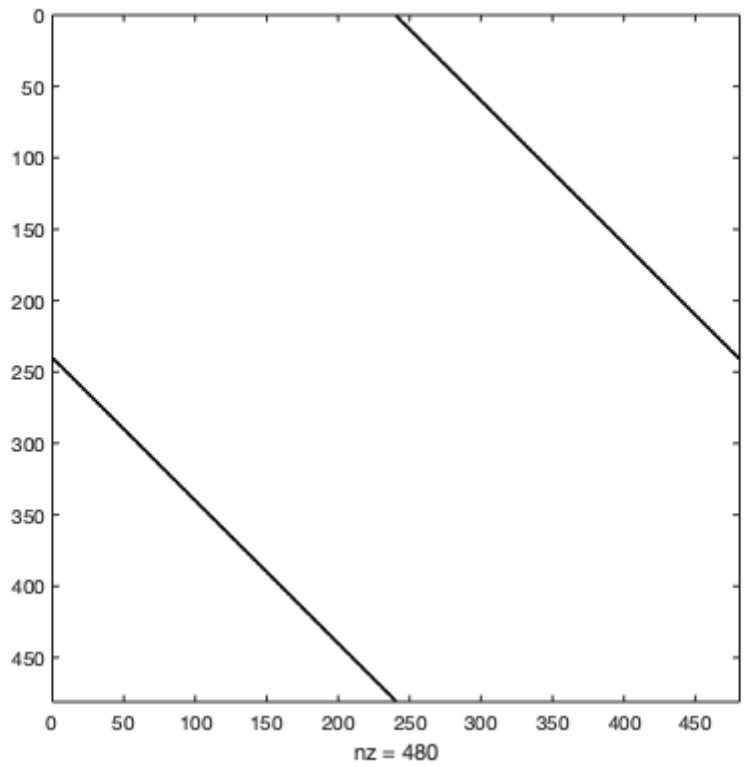
```
This is A
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

This is E
     0     0     0     1
```

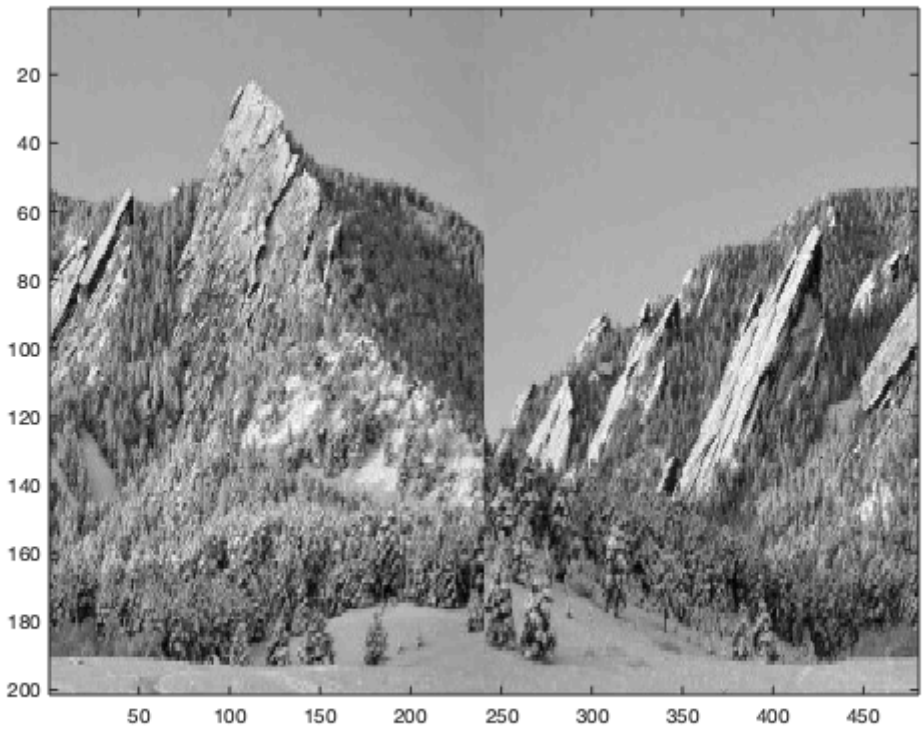
```
     0     1     0     0
     0     0     1     0
     1     0     0     0

This is C = A*E
    13     2     3    16
     8    11    10     5
    12     7     6     9
     1    14    15     4
```

5.14

```
hs_240 = hshift_matrix(p1_gs, 240);
disp('This is the horizontal transformation matrix'); figure(4);
 spy(hs_240,'o','k'); saveas(gcf,'(5.14a).jpg');
p1_gs_hs_240 = p1_gs*hs_240; % post-multiply to apply transformation.
 Then display effect.
figure(5); imagesc(uint8(p1_gs_hs_240)); colormap('gray');
 imwrite(uint8(p1_gs_hs_240), '(5.14b).jpg');
```

```
This is the horizontal transformation matrix
```



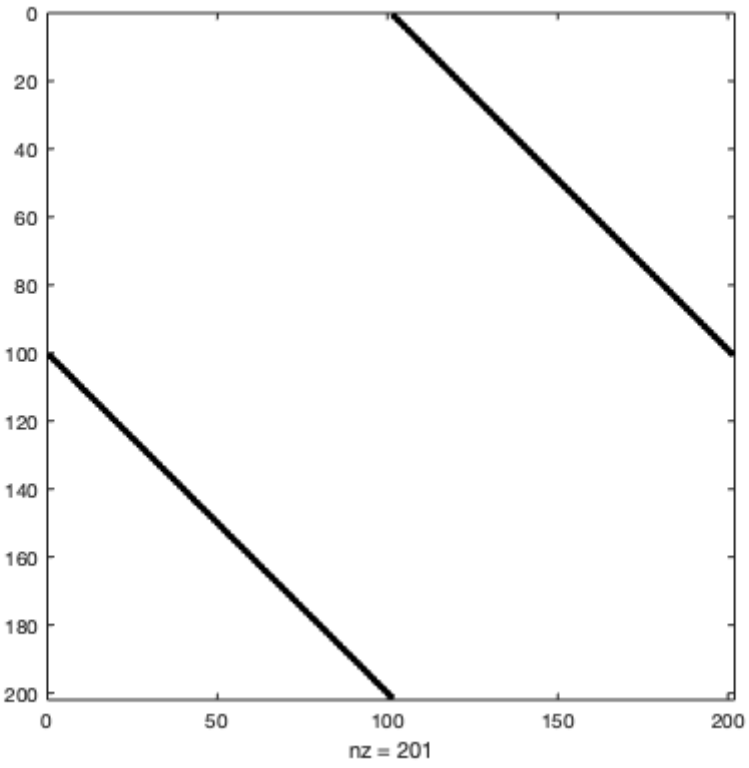


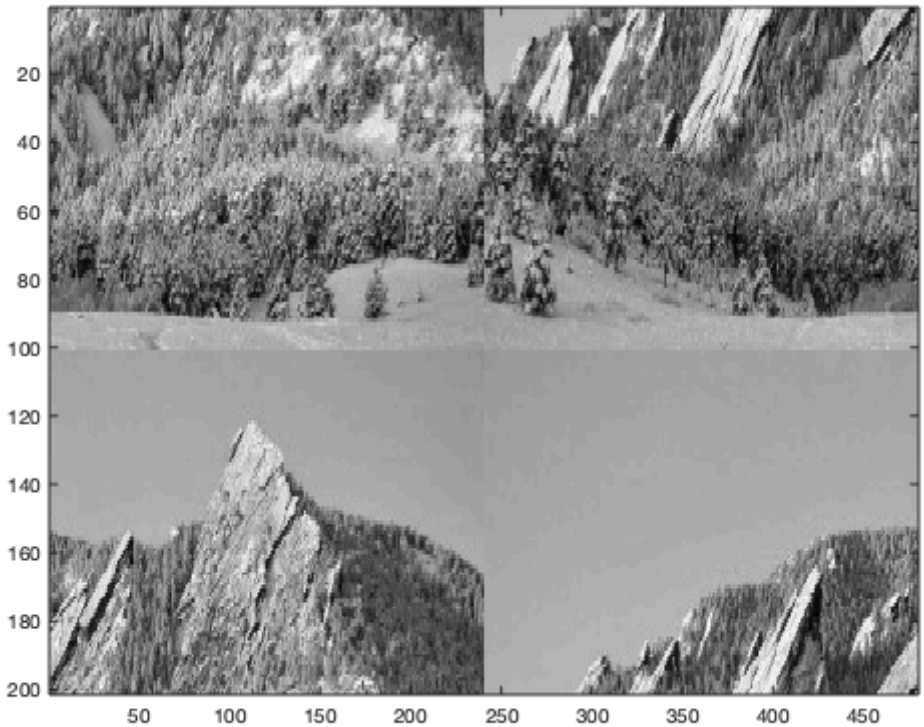
5.15

we can use the same horizontal transformation matrix from part 5.14.

```
vs_100 = vshift_matrix(p1_gs, 100);
disp('This is the vertical transformation matrix'); figure(6);
 spy(vs_100,'o','k'); saveas(gcf,'(5.15a).jpg');
p1_gs_hs240_vs100 = vs_100*p1_gs*hs_240; %pre-multiply and post-
multiply to apply both transformations. Then display effect.
figure(7); imagesc(uint8(p1_gs_hs240_vs100)); colormap('gray');
 imwrite(uint8(p1_gs_hs240_vs100), '(5.15b).jpg');
```

```
This is the vertical transformation matrix
```

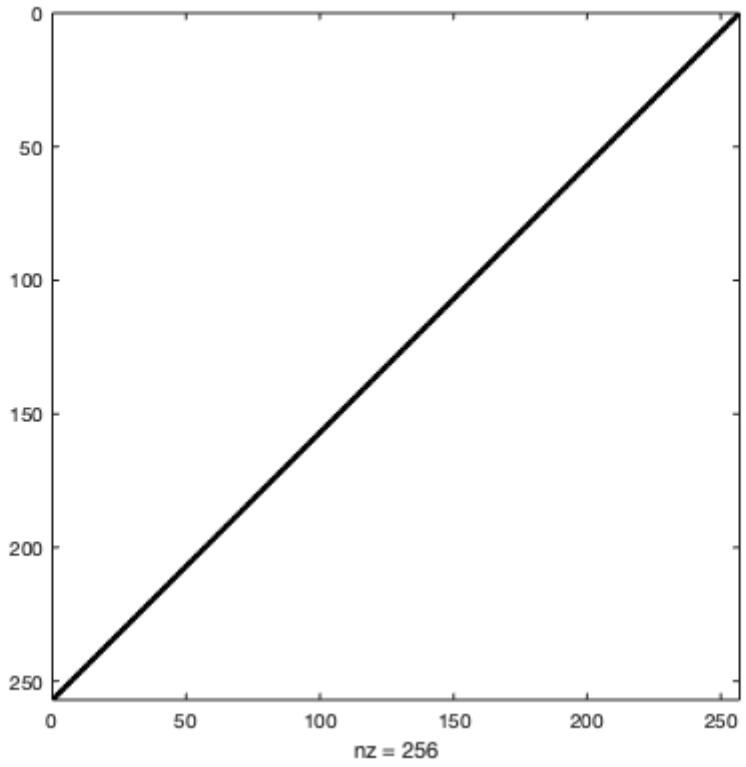

0
20
40
60
80
100
120
140
160
180
200
0
50
100
150
200
nz = 201

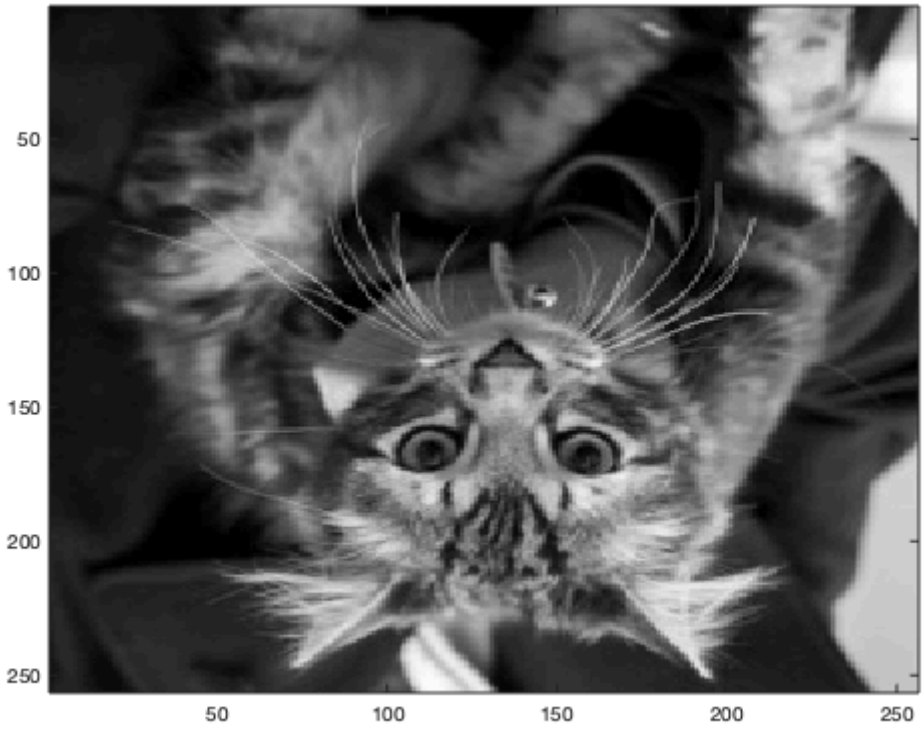

20
40
60
80
100
120
140
160
180
200
50
100
150
200
250
300
350
400
450

5.16

```
flipper = flip_matrix(p2_gs);
disp('This is the transformation matrix to flip an image upside
 down'); figure(8); spy(flipper, 'o','k'); saveas(gcf,'(5.16a).jpg');
p2_gs_flip = flipper*p2_gs; %premultiply by flip matrix. Then display
 effect.
figure(9); imagesc(uint8(p2_gs_flip)); colormap('gray');
 imwrite(uint8(p2_gs_flip), '(5.16b).jpg');
```

This is the transformation matrix to flip an image upside down





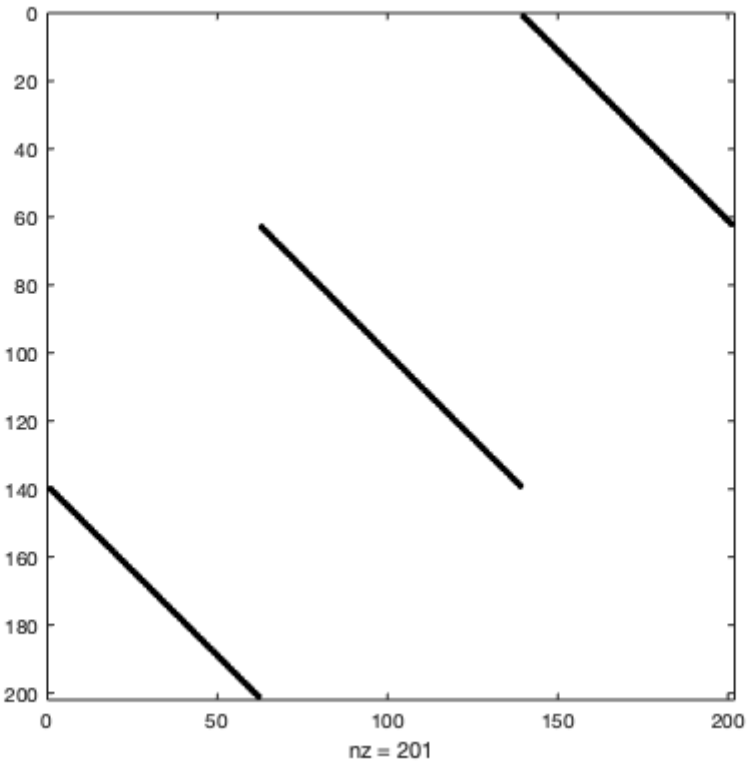
5.17

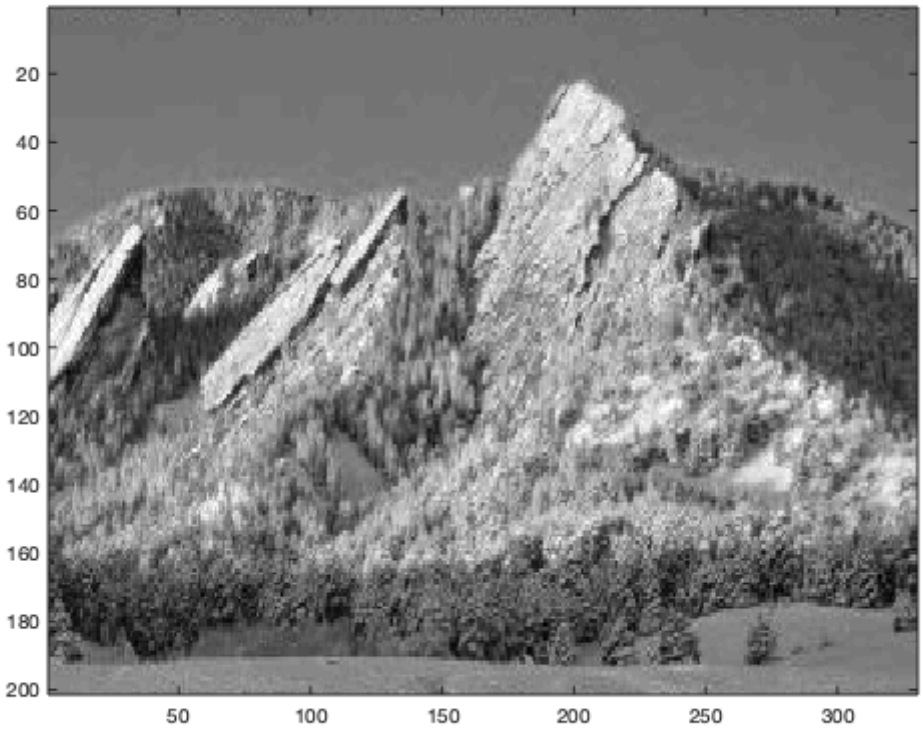
```
p2_gs_transpose = transpose(p2_gs);
figure(10); imagesc(uint8(p2_gs_transpose)); colormap('gray');
 imwrite(uint8(p2_gs_transpose), '(5.17).jpg');
% rotates image/matrix basically.
```

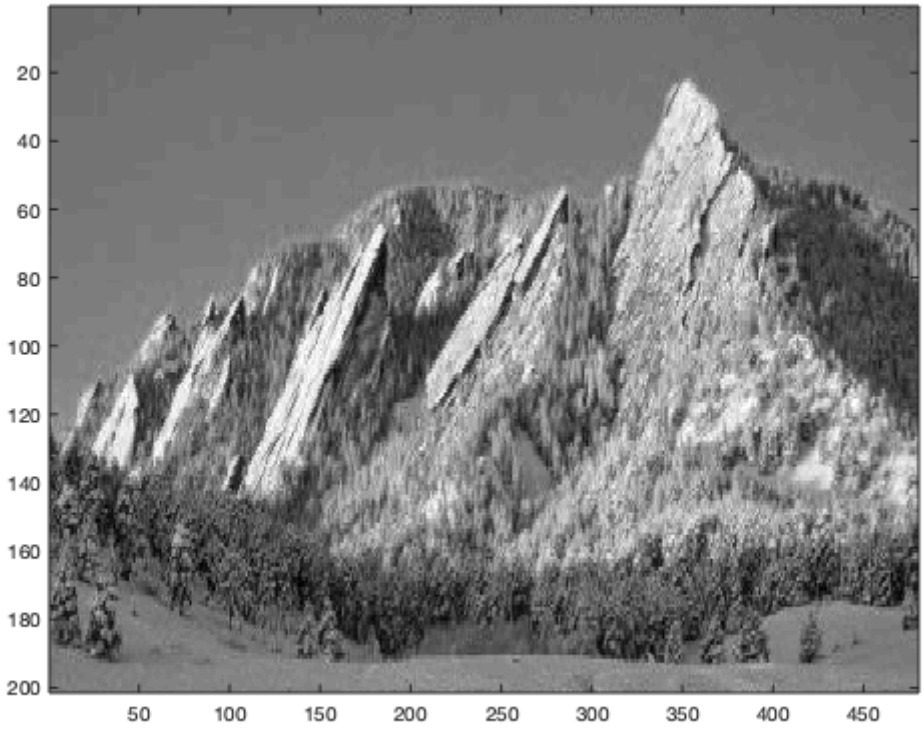
5.18

```
exch = exchange_matrix(m2_gs, 62, 139); % the numbers are the pixel
 locations used to exchange the three regions
disp('This is the transformation matrix to swap the relevant
 regions of the image'); figure(11); spy(exch, 'o','k');
 saveas(gcf,'(5.18a).jpg');
m2_gs_fixed = exch*double(m2_gs); %premultiply by exchange matrix.
 Then display effect.
figure(12); imagesc(uint8(m2_gs_fixed)); colormap('gray');
 imwrite(uint8(m2_gs_fixed), '(5.18b).jpg');
% next we concatenate the two relevant matrices
final_corrected_image_matrix = [double(m1_gs) m2_gs_fixed];
figure(13); imagesc(uint8(final_corrected_image_matrix));
 colormap('gray');
 imwrite(uint8(final_corrected_image_matrix), '(5.18c).jpg');
```

This is the transformation matrix to swap the relevant regions of the image


0
20
40
60
80
100
120
140
160
180
200
0
50
100
150
200
nz = 201


20
40
60
80
100
120
140
160
180
200
50
100
150
200
250
300



PART 2

5.29 and 5.210

```
dst_5 = dst_matrix(5)
disp('This is the order 5 DST matrix'); disp(dst_5);
disp('We now compute the product of the above matrix with itself')
result = dst_5*dst_5; disp('This is the result after rounding to 10
 decimal places');
result = round(result,10); disp(result);
disp('This is the order 5 identity matrix, so it implies that the dst
 matrix is its own inverse.');
```

```
dst_5 =

    0.0989    0.2871    0.4472    0.5635    0.6247
    0.2871    0.6247    0.4472   -0.0989   -0.5635
    0.4472    0.4472   -0.4472   -0.4472    0.4472
    0.5635   -0.0989   -0.4472    0.6247   -0.2871
    0.6247   -0.5635    0.4472   -0.2871    0.0989

This is the order 5 DST matrix
    0.0989    0.2871    0.4472    0.5635    0.6247
    0.2871    0.6247    0.4472   -0.0989   -0.5635
```

```
    0.4472    0.4472   -0.4472   -0.4472    0.4472
    0.5635   -0.0989   -0.4472    0.6247   -0.2871
    0.6247   -0.5635    0.4472   -0.2871    0.0989

We now compute the product of the above matrix with itself
This is the result after rounding to 10 decimal places
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1

This is the order 5 identity matrix, so it implies that the dst matrix
 is its own inverse.
```

5.211

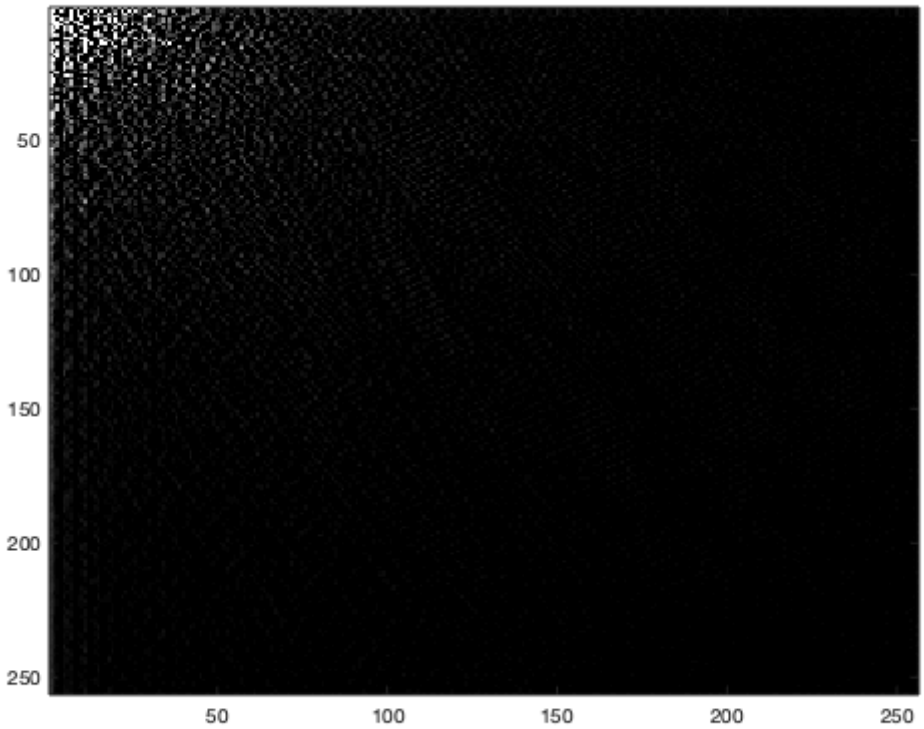
```
disp('We can experiment with an arbitrary square matrix of order 5');
A = magic(5);
disp('Let this be the matrix A')
disp(A);
disp('We can apply the 2D dst onto A to get B. So B = SAS.')
B = apply_dst(A);
disp('This is B')
disp(B);
disp('To undo this effect, we can just apply the 2D dst onto B to get
 C. So C = SBS');
C = apply_dst(B);
disp('This is C after rounding to 10 decimal places. It is the same as
 A.')
C = round(C, 10);
disp(C);
disp('This works because S is its own inverse. So to undo the 2D dst
 we just apply it again.');
% not sure if we have to use an image here. I think this should be
 satisfactory.

We can experiment with an arbitrary square matrix of order 5
Let this be the matrix A
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

We can apply the 2D dst onto A to get B. So B = SAS.
This is B
   54.2191   21.3577    4.1257    5.0813   10.2072
   20.4888   -9.8653   10.1520   15.2195    2.4624
    5.4671   10.0245   19.4000    1.1662    6.0745
    5.4205   11.2356    5.4241    9.4102   -4.6220
    9.8479    1.5705    9.5464   -5.2003   -8.1640
```

```
To undo this effect, we can just apply the 2D dst onto B to get C. So
 C = SBS
This is C after rounding to 10 decimal places. It is the same as A.
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

This works because S is its own inverse. So to undo the 2D dst we just
 apply it again.
```

5.212

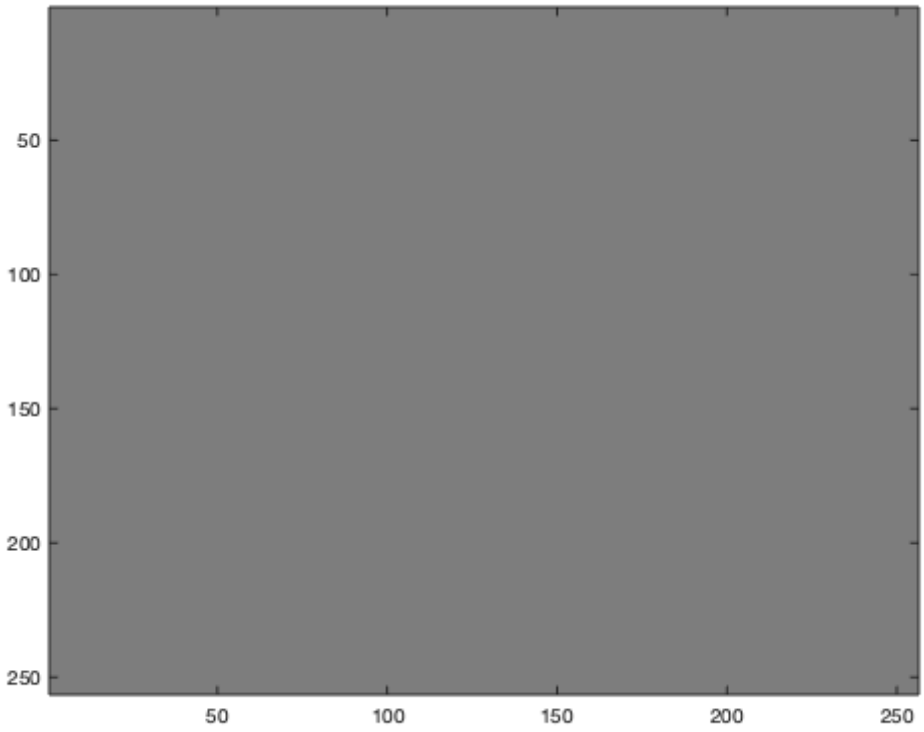
```
Albus = p2_gs;
disp('This is Albus'); figure(14);  imagesc(uint8(Albus));
 colormap('gray'); imwrite(uint8(Albus), '(5.212a).jpg')
Albus_dst = apply_dst(Albus);
disp('This is Albus after a 2D dst'); figure(15);
  imagesc(uint8(Albus_dst)); colormap('gray');
 imwrite(uint8(Albus_dst), '(5.212b).jpg')
disp('Now we will compress the image using different p values, apply
 the inverse DST, and see what happens');

This is Albus
This is Albus after a 2D dst
Now we will compress the image using different p values, apply the
 inverse DST, and see what happens
```


50
100
150
200
250
50
100
150
200
250


50
100
150
200
250
50
100
150
200
250

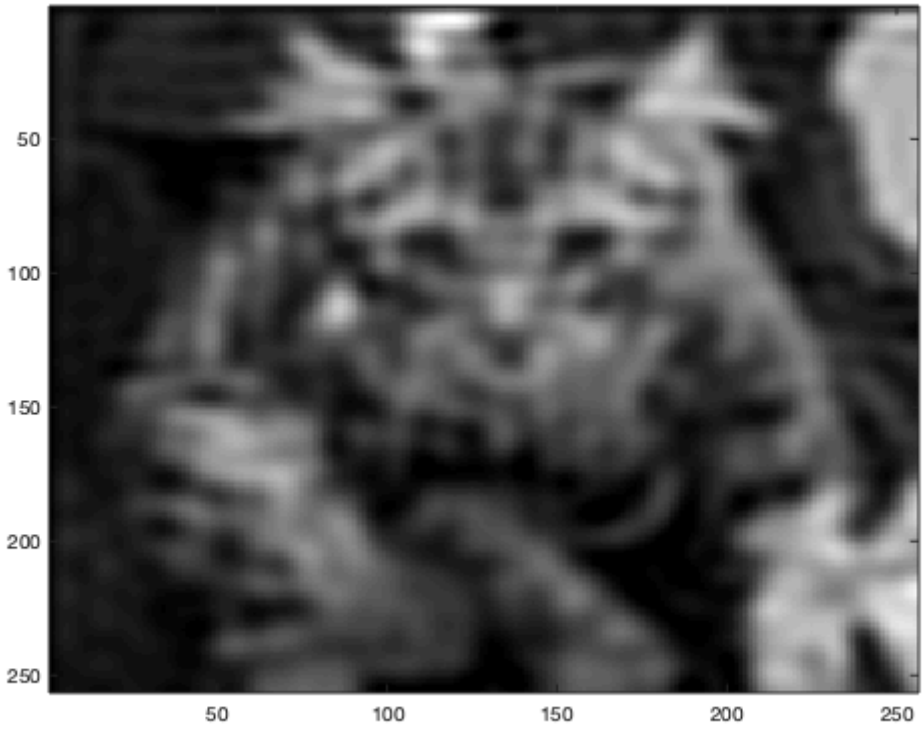
```
disp('p=0')
p = 0;
Albus_00 = apply_dst((zero(Albus_dst,p))); figure(16+(p*10));
 imagesc(uint8(Albus_00)); colormap('gray');
 imwrite(uint8(Albus_00), '(5.212_00).jpg')
```

p=0




```
disp('p=0.1')
p = 0.1;
Albus_01 = apply_dst((zero(Albus_dst,p))); figure(16+(p*10));
 imagesc(uint8(Albus_01)); colormap('gray');
 imwrite(uint8(Albus_01), '(5.212_01).jpg')
```

p=0.1




```
disp('p=0.2')
p = 0.2;
Albus_02 = apply_dst((zero(Albus_dst,p))); figure(16+(p*10));
 imagesc(uint8(Albus_02)); colormap('gray');
 imwrite(uint8(Albus_02), '(5.212_02).jpg')
```

p=0.2


```
disp('p=0.3')
p = 0.3;
Albus_03 = apply_dst((zero(Albus_dst,p))); figure(16+(p*10));
 imagesc(uint8(Albus_03)); colormap('gray');
 imwrite(uint8(Albus_03), '(5.212_03).jpg')
```

p=0.3


```
disp('p=0.4')
p = 0.4;
Albus_04 = apply_dst((zero(Albus_dst,p))); figure(16+(p*10));
 imagesc(uint8(Albus_04)); colormap('gray');
 imwrite(uint8(Albus_04), '(5.212_04).jpg')
```

p=0.4


```
disp('p=0.5')
p = 0.5;
Albus_05 = apply_dst((zero(Albus_dst,p))); figure(16+(p*10));
 imagesc(uint8(Albus_05)); colormap('gray');
 imwrite(uint8(Albus_05), '(5.212_05).jpg')
```

p=0.5


```
disp('p=0.6')
p = 0.6;
Albus_06 = apply_dst((zero(Albus_dst,p))); figure(16+(p*10));
 imagesc(uint8(Albus_06)); colormap('gray');
 imwrite(uint8(Albus_06), '(5.212_06).jpg')
```

p=0.6


```
disp('p=0.7')
p = 0.7;
Albus_07 = apply_dst((zero(Albus_dst,p))); figure(16+(p*10));
 imagesc(uint8(Albus_07)); colormap('gray');
 imwrite(uint8(Albus_07), '(5.212_07).jpg')
```

p=0.7


```
disp('p=0.8')
p = 0.8;
Albus_08 = apply_dst((zero(Albus_dst,p))); figure(16+(p*10));
 imagesc(uint8(Albus_08)); colormap('gray');
 imwrite(uint8(Albus_08), '(5.212_08).jpg')
```

p=0.8


```
disp('p=0.9')
p = 0.9;
Albus_09 = apply_dst((zero(Albus_dst,p))); figure(16+(p*10));
 imagesc(uint8(Albus_09)); colormap('gray');
 imwrite(uint8(Albus_09), '(5.212_09).jpg')
```

p=0.9


```
disp('p=1.0')
p = 1.0;
Albus_10 = apply_dst((zero(Albus_dst,p))); figure(16+(p*10));
 imagesc(uint8(Albus_10)); colormap('gray');
 imwrite(uint8(Albus_10), '(5.212_10).jpg')
```

```
p=1.0
```

5.213

```
us = count(Albus_dst);
cs_00 = count(apply_dst(Albus_00)); cs_01 =
 count(apply_dst(Albus_01)); cs_02 = count(apply_dst(Albus_02));
cs_03 = count(apply_dst(Albus_03)); cs_04 =
 count(apply_dst(Albus_04)); cs_05 = count(apply_dst(Albus_05));
cs_06 = count(apply_dst(Albus_06)); cs_07 =
 count(apply_dst(Albus_07)); cs_08 = count(apply_dst(Albus_08));
cs_09 = count(apply_dst(Albus_09)); cs_10 =
 count(apply_dst(Albus_10));

disp('This is the uncompressed size'); disp(us);
disp('This is the vector of compressed sizes');
cs_vector = [cs_00 cs_01 cs_02 cs_03 cs_04 cs_05 cs_06 cs_07 cs_08
 cs_09 cs_10]';
disp(cs_vector);
disp('This is the vector of compressions ratios');
disp('');
CR = us./cs_vector;
disp(CR);
```

```
This is the uncompressed size
       65536

This is the vector of compressed sizes
```

```
           0
        1276
        5152
       11629
       20706
       32640
       44421
       53601
       60180
       64158
       65536

This is the vector of compressions ratios
         Inf
      51.3605
      12.7205
       5.6356
       3.1651
       2.0078
       1.4753
       1.2227
       1.0890
       1.0215
       1.0000
```

FUNCTIONS

```
% 5.11 - function returns double form of grayscale matrix

function [gs] = gs_matrix(filename)
    gs = double(imread(filename));
    gs = (0.3.*gs(:,:,1)) + (0.3.*gs(:,:,2)) + (0.3.*gs(:,:,3));
end

% 5.12 - function increases exposure of grayscale matrix
function [new_exp] = exposure(img, n)
    new_exp = n.*img;
end

% 5.13 - function returns matrix operator to facilitate column
 swapping
function [swapper] = swap_matrix(mtrx, n1, n2)
    [m,n] = size(mtrx);
    swapper = eye(n);
    temp = swapper(:, n1);
    swapper(:, n1) = swapper(:, n2);
    swapper(:, n2) = temp;
end

% 5.14 - function returns matrix operator to facilitate horizontal
 shifting
function [hshift] = hshift_matrix(mtrx, s)
```

```
    [m,n] = size(mtrx);
    hshift = eye(n);
    hshift = hshift(:, [(end-s+1):end 1:(end-s)]);
end

% 5.15 - function returns matrix operator to facilitate vertical
 shifting
function [vshift] = vshift_matrix(mtrx, h);
    [m,n] = size(mtrx);
    vshift = eye(m);
    vshift = vshift([(end-h+1):end 1:(end-h)],:);
end

% 5.16 - function returns matrix operator to facilitate upsidedown
 flipping
function [flip] = flip_matrix(mtrx)
    [m,n] = size(mtrx);
    flip = eye(m);
    flip = flip([end:-1:1],:);
end

% 5.17 - no function needed for this

% 5.18 - function returns matrix operator to facilitate exchange of
 three special matrix row regions
% in particular, the three regions are connected and they span the
 entire matrix when combined
% this function is specially designed for the given problem
function [exchange] = exchange_matrix(mtrx,n1,n2)
    [m,n] = size(mtrx);
    exchange = eye(m);
    exchange = exchange([(n2+1):end (n1+1):n2 1:n1],:);
end

% 5.29 and 5.210 - function returns square DST matrix of desired size
function [dst] = dst_matrix(n)
    dst = zeros(n);
    for i = 1:1:n
        for j = 1:1:n
            dst(i,j) = (sqrt(2/n))*sin((i-0.5)*(j-0.5)*(pi/n));
        end
    end
end

% 5.211 - Applies 2D dst onto a matrix. This is also the inverse 2D
 dst.
function [res] = apply_dst(mtrx)
    [m,n] = size(mtrx);
    if m == n
        s = dst_matrix(n);
        res = s*mtrx*s;
    else
        disp('error');
        res = 0;
```

```
    end
end

% 5.212 - Zeros regions of matrix based on value of parameter p
function [zeroed] = zero(mtrx, p)
    [m,n] = size(mtrx);
    if m == n
        zeroed = mtrx;
        for i = 1:1:n
            for j = 1:1:n
                if i+j>(p*2*n)
                    zeroed(i,j) = 0;
                end
            end
        end
    else
        disp('error');
        zeroed = 0;
    end
end

% 5.213 and 5.214 - Computes number of nonzero elements in the matrix

function [counted] = count(mtrx)
    counted = 0;
    [m,n] = size(mtrx);
    for i = 1:1:m
        for j = 1:1:n
            if abs(mtrx(i,j)) > (10^(-10))
                counted = counted + 1;
            end
        end
    end
end
```

Published with MATLAB® R2019b