APPM 2360 Project 2

Ethan Phalen, Savi Singh, Muhannad Ibrahim

April 2021

1 Introduction

In the world of social media and technology, people are taking, sharing, and editing photos relentlessly. Our group of friends require our help in creating an application where they can edit, compress, and manipulate specific images in order for them to pursue a successful business idea. This is where Matlab and our knowledge of matrices' properties comes into play. Digital images are composed of pixels, and we can manipulate these pixels using matrix arithmetics and manipulating their vectors. For example, we can manipulate the identity matrix in order to shift and rearrange pixels that alter the picture. Other important concepts that will help our friends in their digital image application are: grayscaling an image, changing the color of an image, and compressing an image using the Discrete Sine Transform, which uses a decomposition of vectors into linear sine functions to create different frequencies of an image.

2 Image Transformations

Our first task was to simply just read an image file and display the grayscale and original image. In matlab, we accomplished this by using the 'imread' and 'imagesc' functions. In order to develop the grayscale version of the image, we had to make a linear combination of vectors that utilized the red, green, and blue pixel matrices. The grayscale image proved to be quite helpful, as it turns the intensities of pixels into its own n x m matrix. This makes it very easy to work with and manipulate because it does not have a third dimension. The following images were a result of the 'imagesc' function:

Figure 1: Original Image

Figure 2: Grayscale Image

Another important aspect of picture editing is being able to increase the exposure of images. The rectangle image and all other images have three color matrices that are responsible for red, blue, and green color intensities. In order to white out the rectangle image, we added 100 to the red, blue, and green matrices to give it more light because we know images have integer values ranging from 0 to 255. This is a comparison of the whited out image versus the original image:

Figure 3: Original.

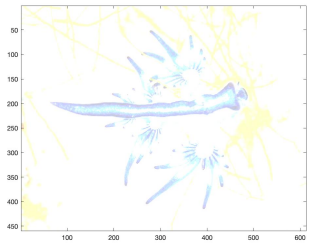

Figure 4: Whited Out

Here is another colored version of the rectangle image with 0 red intensity and a blue intensity added with 80:

Figure 5: No red, plus 80 blue

2.1 Translations

Aside from the color intensity matrices, an image can be shifted or refigured by changing the rows and columns around. This is done by multiplying the image by a certain Identity Matrix. The identity matrix needs to be refigured in the same manner as the desired new matrix. Given a 4x4 matrix A, in order to switch the leftmost and rightmost columns, the identity matrix needs to be altered in such a way that the left and rightmost columns are switched. Then, this can be called matrix 'E'. To get the new matrix, AE is the only correct order of multiplication. Here are the following matrices:

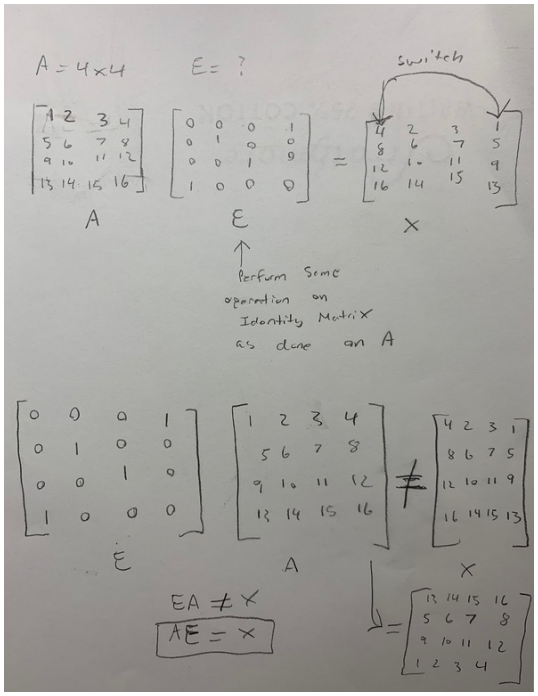

Figure 6: matrix E will swap first and last columns

The square matrix image shift is also applicable to non-square matrices such as the images in this project. By multiplying by the correct transformation matrix, the rectangle image was horizontally shifted:

Figure 7: Horizontal Shift

To preform a vertical shift along with a horizontal shift, we can multiply using the same technique as before, except to implement a vertical shift, we simply change the order of the matrix product; E_1AE_2 will preform both as follows:

Figure 8: Double Shift

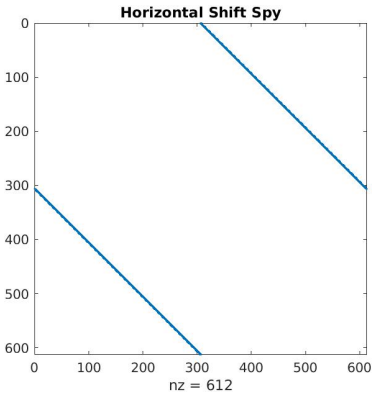

Figure 9: Horizontal Shift

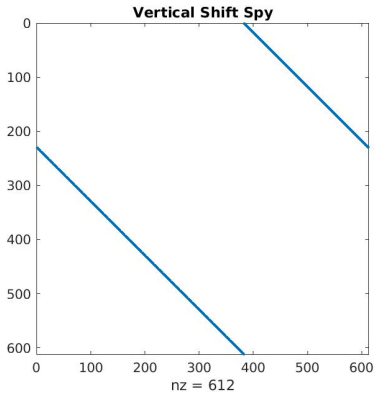

Figure 10: Vertical Shift

2.2 Flipping an Image

Since we've moved the image around, let's now consider how we might flip the picture over vertically, or upside down. To do this, we first see that the "upside down" identity matrix will flip the picture horizontally. To go vertically, we simply multiply the other direction:

Figure 11: Original

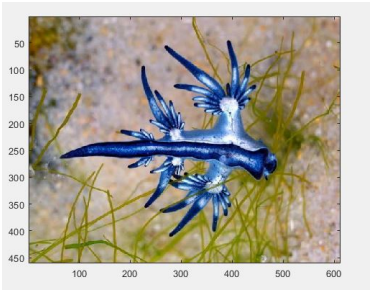

Figure 12: Flipped

In addition to flipping an image vertically, we can also transpose the color matrices, which results in Figure 8 shown below. It appears that we can flip the image matrix across it's diagonal using the transpose. This also appears as a mirror, with a rotation, since by definition we are flipping the rows with the columns.

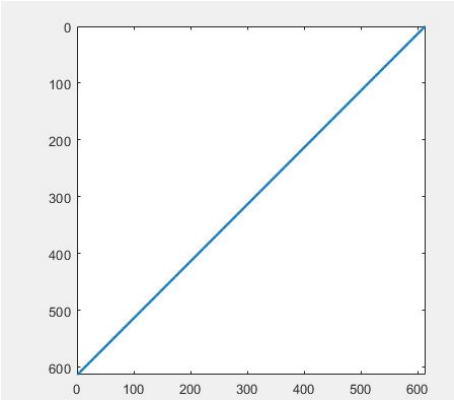

Figure 13: Spy() graph for the transformation matrix, blue is nonzeros (all ones in this case)

Figure 14: Proof for Number 8

2.3 Cropping

Another thing we can do to an image is crop it, which can be done using matrix transformations. We take the 612x612 Identity matrix and remove the ones from the border based on how big of a crop we want, then set that equal to our transform matrix. We then multiply tXt, where t is the transform matrix and X is the double version of the color matrix (We do this separately for each color). We have to multiply by the transformation matrix twice, once on each side, in order to take the top and bottom borders and left and right borders respectively.

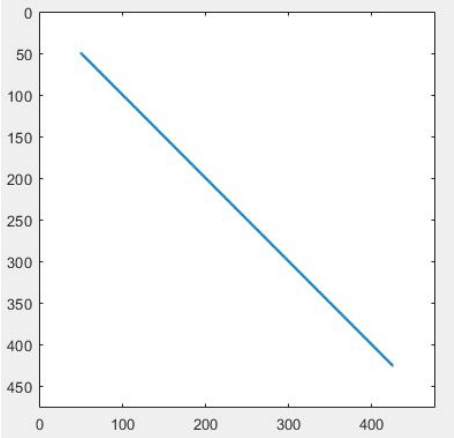

Figure 15: Spy() Graph of Transform Matrix

Figure 16: Image With Border

3 Image Compression

Matrices are useful when manipulating the look of an image. In order to compress an image we begin by creating a matrix based off of the DST function.

3.1 Matrix

The DST functions allows us to create a matrix S, of nxn. This matrix is also the inverse of itself. To create a two dimensional DST, multiply the image by the S matrix. Here we assume X_g is the image.

$$Y = SX_gS^{-1}$$

$$S^{-1} = S$$

$$X_g = SYS$$

Since S is it's own inverse, and we multiply by an inverse to "divide", we can solve the DST for X_g easily, undoing the DST.

3.2 Compressing an Image using DST

Using this discrete sin transform, we can filter out and remove high frequency data, thus making the file size smaller and "compressing" the image. This is our Y value, a matrix that has been transformed. We then multiply this transformation by our original 3 color matrices and reverse the DST using the method shown above. We can then re concatenate these 3 matrices on top of each other to form the compressed image. By changing our value for p, we can adjust how much data we want to remove.

3.3 Compression Ratio

The compression ratio is the uncompressed size divided by the compressed size; essentially by what factor was the file shrunk. We could not get the number of non-zero terms to come out correctly, even using the built in MatLab function nnz(), so for the purposes of this project we will write our images to files and use the actual compression ratio, rather than non-zero term count. We can see the compressed images for different values of p below:

Figure 17: Full Quality, P=1

Figure 18: P = 0.8

Figure 19: P = 0.5

Figure 20: P = 0.3

We tested many different values for p, some of which are displayed below:

For $p = .5$, the compression ratio is 5.8.

For $p = .3$, the compression ratio is 6.2.

$p = .1$, the compression ratio is 8.8.

After experimenting with different values for p, we guess that 0.4 is a good balance between quality and file size, qualitatively. We started at p = 1, and it

stayed close in size until we move lower than p = 0.8 We couldn't even notice a difference in quality until about 0.6, where the file is already significantly smaller. The difference starts to become relatively noticeable around .5, as seen in the above figures, and 0.4 is what we settled on as the best considering the trade offs.

4 Conclusion

Matrices have many uses, one of which can be image manipulation. This can be useful for many reasons, including the compression of images to save file size. We also learned that we can remove a fair bit of data before the image quality decreases significantly. Today many companies such as Instagram implement image compression to save hard drive space to accommodate for a large user-base. And today for our friends, we have shown them how to do the same techniques implemented by large companies using matrix transformations.