Image Manipulation in MATLAB

Ashish Srivastava, Maya Singh, Alexis Walker

APPM 2360 (Spring 2021) Project 2

November 26, 2025

1 Introduction and Background

A few friends in the business school want to make a new photo editing app for mobile devices and are relying on us for the technical side of things. We will be helping by using our APPM 2360-based matrix manipulation knowledge as well as our programming capability to write some of the code necessary to manipulate the images.

1.1 Image Manipulation

Multiplying any given matrix or vector by the identity matrix $\mathbf{I}$ is an analogue to "multiplying it by one", since nothing changes. However, if we change up the identity matrix in certain ways to make a permutation matrix, it allows us to apply transformations to matrices. Take for example, the following permutation matrix that shifts every row down one, and brings the last row to the top:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 6 & 11 & 16 & 21 \\ 2 & 7 & 12 & 17 & 22 \\ 3 & 8 & 13 & 18 & 23 \\ 4 & 9 & 14 & 19 & 24 \\ 5 & 10 & 15 & 20 & 25 \end{bmatrix} = \begin{bmatrix} 5 & 10 & 15 & 20 & 25 \\ 1 & 6 & 11 & 16 & 21 \\ 2 & 7 & 12 & 17 & 22 \\ 3 & 8 & 13 & 18 & 23 \\ 4 & 9 & 14 & 19 & 24 \end{bmatrix}$$

We can use this property of transformation matrices alongside the fact that we can read images into MATLAB as a three-dimensional matrix, with each slice of the matrix corresponding to the intensity values of a given color (red, green, and blue) in the pixels of an image. We can change colors by altering each color plane individually, or apply the same transformation to each color plane and rejoin them to apply a transformation to the overall colored image. Care must be taken to apply the transformation correctly though, since multiplying the transformation matrix on the left side of a matrix will yield a different transformation than multiplying on the right will.

Another example of a common image manipulation that we will do is converting an image to its grayscale form. We will do so using the provided code in the project assignment document.

Throughout the development of this report, we will demonstrate how matrix manipulations on these color matrices in MATLAB can be used to change the coloring and saturation of image, apply translations to the image, flip the image, and add borders to crop the image.

1.2 Image Compression

Image compression reduces the storage cost of an image file. This helps make both storage and sharing of images more efficient.

1.2.1 The Discrete Sine Transform

The Discrete Sine Transform (DST) allows for us to turn a vector into a linear combination of sin functions with different frequencies. If given an image matrix, we can transform it by multiplying it by the $n \times n$ DST matrix $\mathbf{S}$, which is defined as:

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

where $s_{i,j}$ is the entry in the i-th row and j-th column of $\mathbf{S}$.

This $\mathbf{S}$ matrix has a few special properties, namely that it is symmetric ($\mathbf{S}^{\mathrm{T}} = \mathbf{S}$) and that it is its own inverse ($\mathbf{S}^{-1} = \mathbf{S}$).

If given an image matrix $\mathbf{X}_g$, we can perform the two dimensional DST as $\mathbf{Y} = \mathbf{S}\mathbf{X}_g\mathbf{S}$. Multiplying the image matrix by $\mathbf{S}$ on both the left and right hand sides applies the one dimensional DST to both the rows and columns of $\mathbf{X}_g$.

1.2.2 Compression

After performing the DST on a given image matrix, and decomposing it into a linear combination of sine frequencies, since human eyes can't distinguish higher frequency components that well, we can remove them from the image, thus reducing the storage cost of the file. To get rid of the higher frequency components we will be utilizing the provided code in the project document.

Different levels of compression (using different values of a variable denoted p) will yield different levels of recognizability of the original image, and also different file sizes. Here we can note that if $p = 0$, no data will be saved (the compressed image will just be black), whereas if $p = 1$, all of the data will have been saved (meaning no compression was done).

After performing compression on an image, we can define the compression ratio to be the ratio between the uncompressed file size and the compressed file size:

$$CR = \frac{\text{uncompressed size}}{\text{compressed size}} \tag{2}$$

2 Image Translation

Every image that has been generated for the Image Translation section of the project has associated code which can be found in Appendix E unless specified otherwise (such as definitions of specific functions).

Over the course of this project, we will have to read in images many times, so defining a function that reads in a given image and extracts the color plane matrices we need will make this much more convenient. We defined a function `read_image.m` (that can be referenced in Appendix B) which takes in a given image file name, and converts it to an $m \times n \times 3$ matrix of doubles (essentially decimal values). Each plane of this double matrix is then extracted into distinct color matrices: `X_red, X_green`, and `X_blue` (all $m \times n$ matrices). We then divide each of the color intensity matrices by 3 and recombine them into a new three dimensional image matrix `X_gray` that yields us a grayscale image. This grayscale image can be viewed in Figure 1.

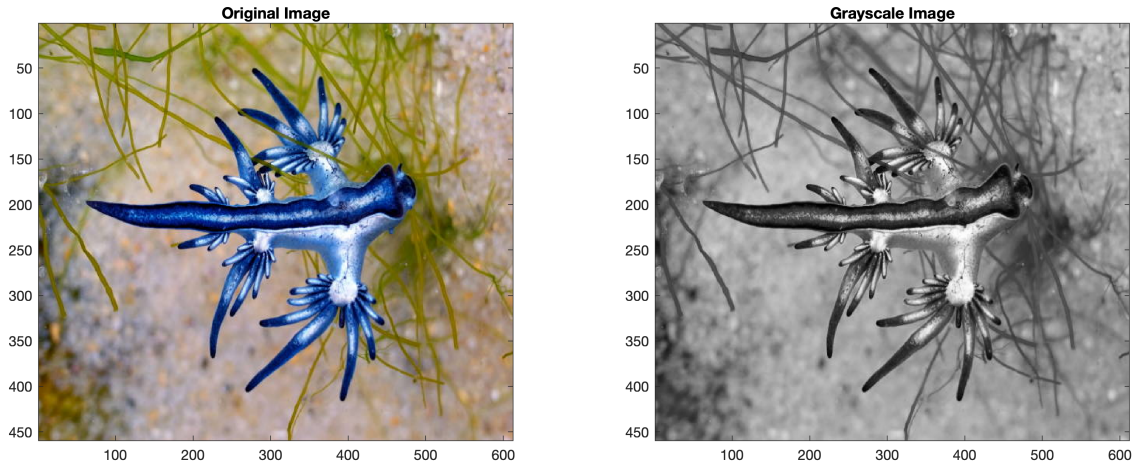

Figure 1: Grayscale of an Image

When we represent a grayscale image as a matrix, the values of the matrix reflect the intensity of the pixel. Given this, it's a simple task to increase the exposure of a grayscale image.

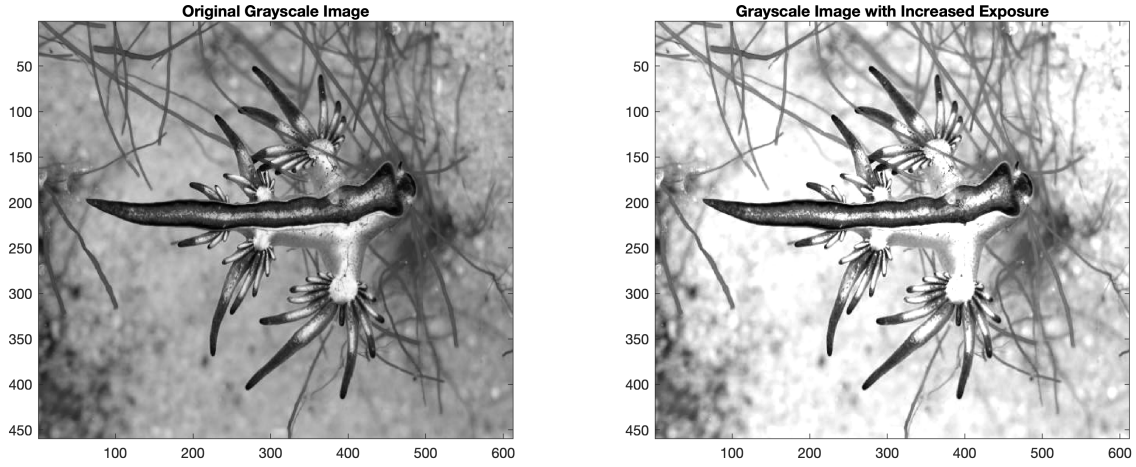

Figure 2: Increasing the Exposure for a Grayscale Image

To do so, as shown in E, once we've read in our image and stored the grayscale image matrix, we can simply add a value (in our case, we chose to increase the intensity of each pixel by 80), and display the new image. The original grayscale image and the grayscale image with increased exposure have been placed side-by-side for comparison in Figure 2.

Grayscale images are only one example of color manipulations that are possible when working with images as matrices. By performing manipulations on each color plane individually, and then recombining the color planes into a single matrix, we can edit the associated color values of the original image. For example, say we want to remove all of the red from the original image, but increase the blue intensity by 80 units (leaving the green intensities unchanged). We can do so by multiplying the entirety of the red color plane by 0, adding 80 to every value in the blue color plane, and not manipulating the green color plane at all, then passing all three of these new color planes into a new $m \times n \times 3$ matrix which can then be visualized. We've included this changed color image below in Figure 3.

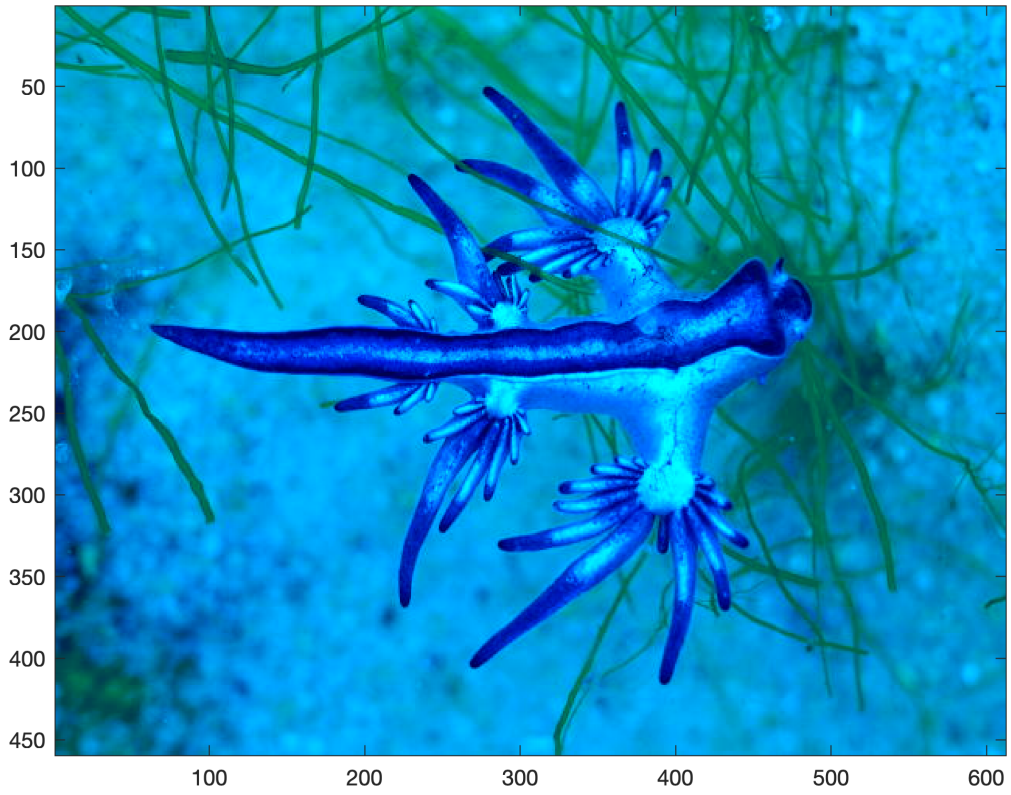

Figure 3: Changing the Color Intensities of the Image

Now we turn our attention to image translations: we can utilize the concept of matrix multiplication to switch around rows/columns of image matrices, translating the image. As a simpler example, if given an image that is 4 pixels by 4 pixels, where the grayscale representation is a 4×4 matrix $\mathbf{A}$, the matrix $\mathbf{E}$ that would switch the leftmost column of pixels with the rightmost column of pixels would take the form of:

$$\mathbf{E} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Care must be taken when applying the transformation matrix. To properly apply this transformation to the original matrix $\mathbf{A}$, you would have to multiply $\mathbf{A}$ by $\mathbf{E}$ on the right: $\mathbf{AE}$. To illustrate this, we have included a sample calculation in Appendix A.1 where it can be shown that if we had instead multiplied $\mathbf{E}$ on the left, it would result in switching the top and bottom rows as opposed to the leftmost and rightmost columns.

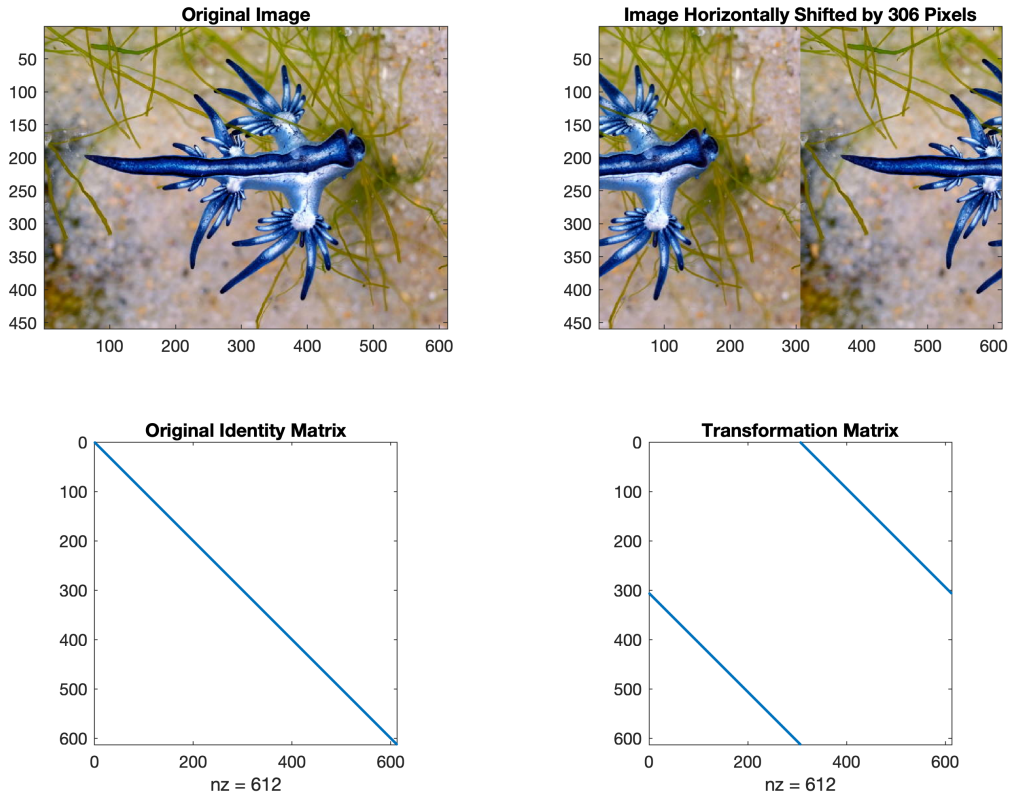

Figure 4: Performing a Horizontal Shift on `rectangle.jpg`

Such transformation matrices are not limited to just square matrices: we can extend this concept of transformation matrices using the entirety of the rectangular image matrix for `rectangle.jpg`. For example, say that we want to perform a horizontal shift of 306 pixels. In order to create the transformation matrix that would produce this shift when applied to the image, the columns of the identity matrix had to be shifted in the same way. Each color plane of our image `rectangle.jpg` was a 459×612 matrix, so since we would be multiplying our transformation matrix on the right ($\mathbf{X}\mathbf{I}_R = \mathbf{X}$), the dimensions of our transformation matrix needed to match. This transformation would result in a transformation matrix that was 612×612.

The appropriate shifts on the identity matrix that would perform this transformation were produced by replacing the first 306 columns of $\mathbf{I}$ with the last 306 columns of the same matrix (and filling the last 306 columns with zeros). Then the now emptied zero columns were replaced with the first part of the original identity matrix. Both the original identity matrix, and the shifted transformation matrix can be viewed above in Figure 4.

In order to apply this transformation to the entire image, we iterated through each of the color planes; applying the transformation matrix to each one by multiplying the color plane matrix on the right hand side, and then recombining all of the shifted color plane matrices into a single $459 \times 612 \times 3$ matrix which could then be viewed. The original `rectangle.jpg` and the `rectangle.jpg` shifted by 306 pixels can also be seen in Figure 4.

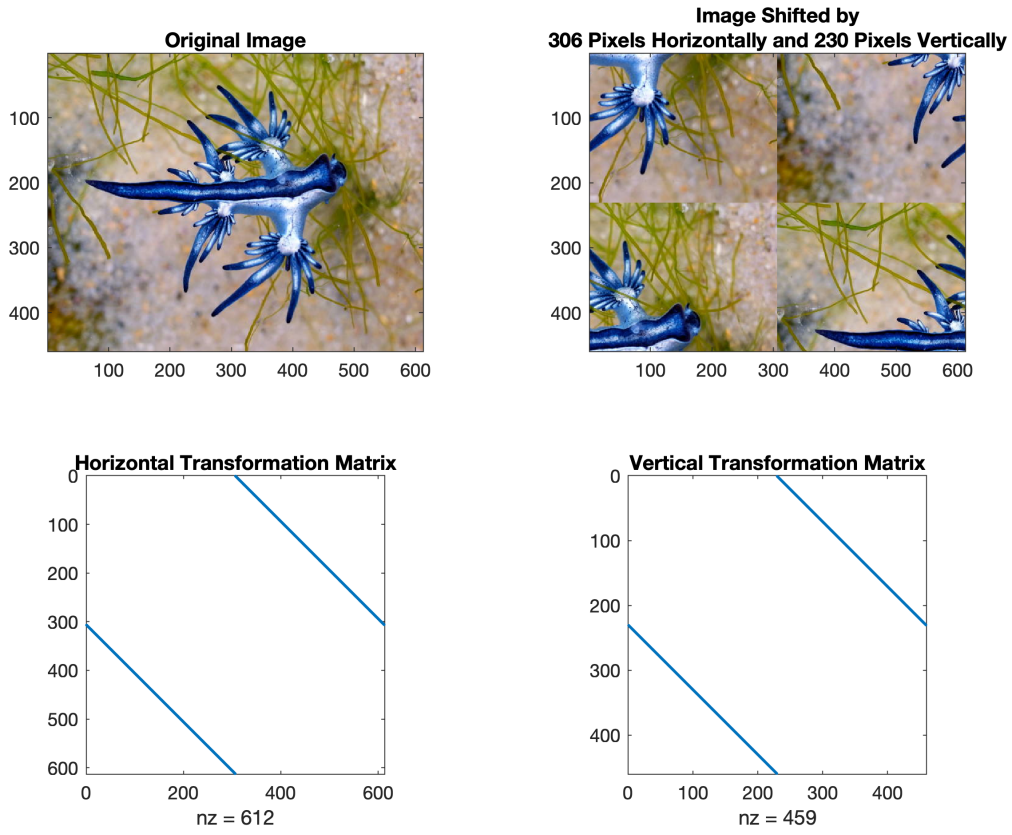

Figure 5: Performing a Horizontal and Vertical Shift on `rectangle.jpg`

This method of shifting images using multiplication with transformation matrices is not limited to horizontal shifts, it can vertically shift images as well. If we wanted to produce both a horizontal and a vertical shift, initially the original image would be multiplied by a horizontal shifting transformation matrix, then the horizontally shifted image would be multiplied by a vertical shifting transformation matrix.

Care must be taken to ensure that the dimensions of the transformation of the matrix match the dimensions of the image. Since our two dimensional image has dimensions of 459×612, the horizontal shift transformation matrix would have dimensions of 612×612, and the image would have to be multiplied by this transformation matrix on the right. The vertical shift transformation matrix would have dimensions of 459×459, and be applied to the left of the image matrix.

The horizontal shifting matrix can be made the same as before, but the vertical shifting matrix is made by manipulating the rows of the identity matrix as opposed to the columns.

Using the code as provided in Appendix E, such a process was taken to shift the `rectangle.jpg` horizontally by 306 pixels and vertically by 230 pixels. The original image, transformed image, horizontal transformation matrix and vertical transformation matrix can be seen above in Figure 5.

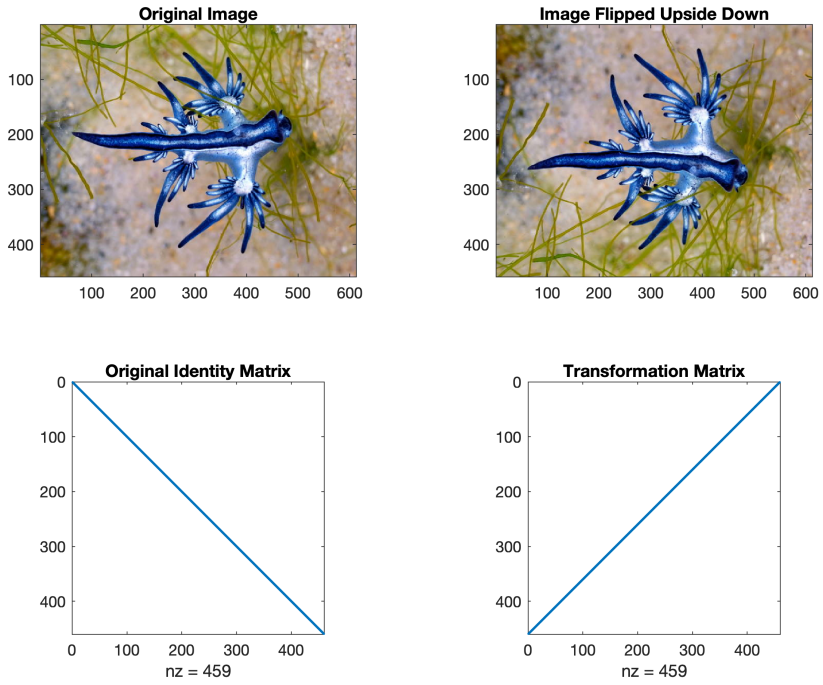

Figure 6: Flipping `rectangle.jpg`

Matrix multiplication levies us not only the ability to shift the image horizontally and vertically, but also to flip the image. If we wanted to flip the image upside down, we would need the first row to switch with the last, the second to switch with the second last, and so on: essentially creating a transformation matrix that is the anti-identity matrix. Since the rows are being manipulated as opposed to the columns, this transformation matrix would have to be multiplied on the left side of the image matrix, and for the case of `rectangle.jpg`, it would have dimensions of 459×459. In MATLAB this transformation matrix was created by creating two 459×459 matrices: one being the identity matrix, and one being a matrix full of zeros. The matrix full of zeros was then populated by iterating through the columns of the identity matrix backwards, and replacing the columns of the zero matrix in the forwards direction, creating the 459×459 anti identity matrix. Both the original identity matrix and the transformation matrix can be seen above in Figure 6

When this matrix was applied to each of the color planes of `rectangle.jpg`, and then recombined into a single image matrix, it produced the image which can also be found in Figure 6.

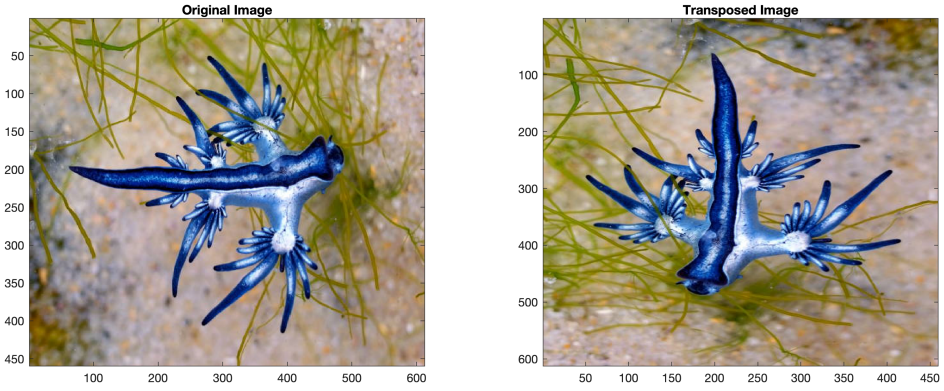

Figure 7: Transposing `rectangle.jpg`

Transposing a matrix essentially "flips" the rows and columns of that matrix. For an image this would

be akin to rotating the image clockwise by 90^{o}, then reflecting it horizontally over its middle axis. Through MATLAB, matrix transposes are simple to implement with the `transpose(A)` function. To produce Figure 7, each color plane was transposed, and then all three color planes were recombined to make the final image. This transposed image may seem horizontally stretched out, but that is only because of the way that MATLAB created the graphics. By observing the axes, it is evident that the horizontal and vertical dimensions have switched, and this image is indeed the transpose of `rectangle.jpg`.

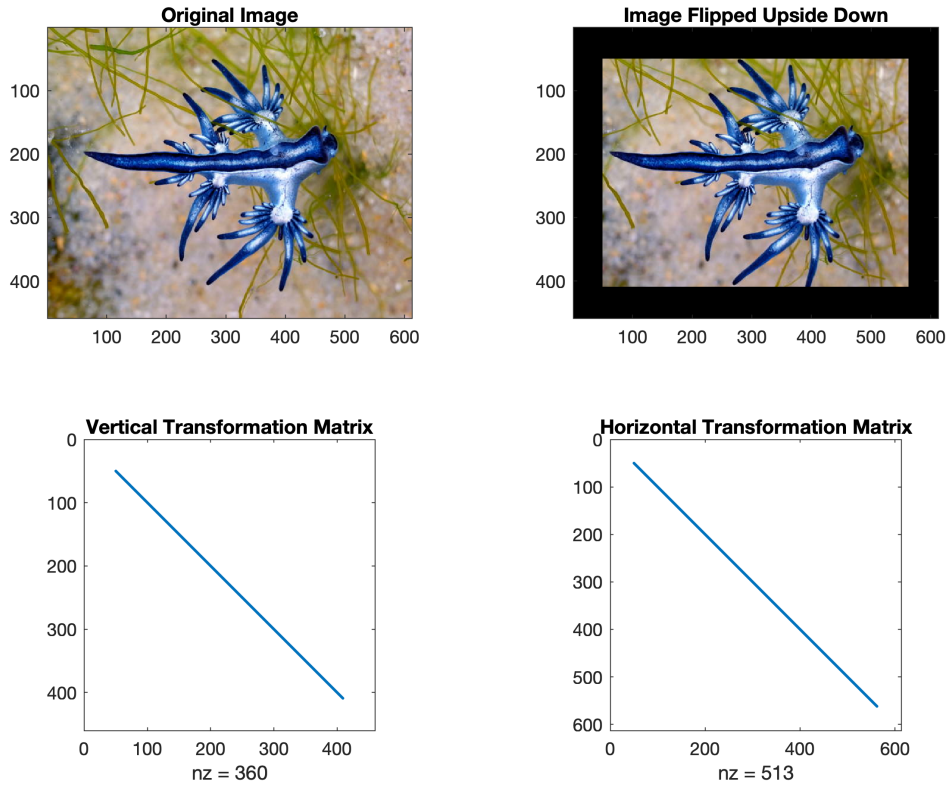

Figure 8: Cropping `rectangle.jpg` with a 50px border

Cropping is a functionality that may not initially seem feasibly by only using transformation matrices, but it is easily accomplished. For our purposes, cropping is the same as getting rid of part of an image (setting their value in the image matrix to 0). To set, for example, a 50px border around `rectangle.jpg`, we can apply both a horizontal and a vertical transformation matrix that will eliminate the top and bottom 50 rows and columns. The vertical transformation matrix will once again be based off of the 459×459 identity matrix, but the first and last 50 columns have been replaced with the first and last 50 columns of the corresponding zero matrix. A similar process is done for the horizontal transformation matrix, but it will have dimensions of 612×612 instead. These two transformation matrices can be seen above in Figure 8.

By multiplying the vertical transformation matrix on the left of the image matrix, and the horizontal transformation matrix on the right, we are yielded with the top right picture in Figure 8.

3 Image Compression

All of the code that has been generated for the Image Compression section of the project has associated code which can be found in Appendix F unless specified otherwise (such as definitions of specific functions).

We have defined a function `S.m` (which can be referenced in Appendix C) that creates the **S** DST matrix for any given size n. Given an input size n, the function creates a square zero matrix of the same size, which it then fills by looping over each row i , then looping over each column element j in each row and applying the DST formula to $s_{i,j}$.

In order to verify that our algorithm indeed creates a matrix **S** such that $\mathbf{S} = \mathbf{S}^{-1}$, we can demonstrate that for the 5×5 case of **S** that $\mathbf{SS} = \mathbf{I}_5$. We performed this calculation in MATLAB and have included the code in appendix F, but for the sake of demonstration, we have included sample output here:

```
    >> S(5)*S(5)

ans =

    1.0000         0   -0.0000   -0.0000   -0.0000
         0    1.0000    0.0000    0.0000    0.0000
   -0.0000    0.0000    1.0000   -0.0000   -0.0000
   -0.0000    0.0000   -0.0000    1.0000    0.0000
   -0.0000    0.0000   -0.0000    0.0000    1.0000
```

This is equivalent to the identity matrix (the negatives and decimal places show up because of the nature of MATLAB's numerical computation).

If we've performed the the DST on a given matrix $\mathbf{X}_g$, and gotten the matrix **Y**, (recall that $\mathbf{Y} = \mathbf{SX}_g\mathbf{S}$), we can utilize a special property of the **S** matrix to undo the DST. Using the fact that $\mathbf{SS} = \mathbf{I}$, we can use the method described in Appendix A.2 to derive that $\mathbf{X}_g = \mathbf{SYS}$ (thus undoing the two dimensional DST). In order to make sure our solution was correct, we verified this formula in MATLAB, which can be seen in Appendix F.

When compressing images with our algorithm, different values of p yield different levels of compression. Recall that $p = 1$ means that all data are saved, and that $p = 0$ means that no data are saved. In order to make the testing of different p values more convenient we have decided to define another function to do the compression: `compress_image.m` which takes a value of p as input, and returns the compressed image. The code in Appendix D works as follows: it reads the matrix into MATLAB and stores it as a matrix of doubles, and takes the dimension of the square matrix (a single color plane). Then, using a `for` loop, it iterates through each of the color planes and performs the two dimensional DST on each color plane matrix, storing it to a new matrix **Y**. Next, it loops over every row i, and every column element in that row j, and performs our compression algorithm for whichever p value was given as an input to the function. In the end, it recollects all of the compressed color plane matrices and undoes the DST by multiplying our **Y** on the left and right by **S**, returning our original image but compressed.

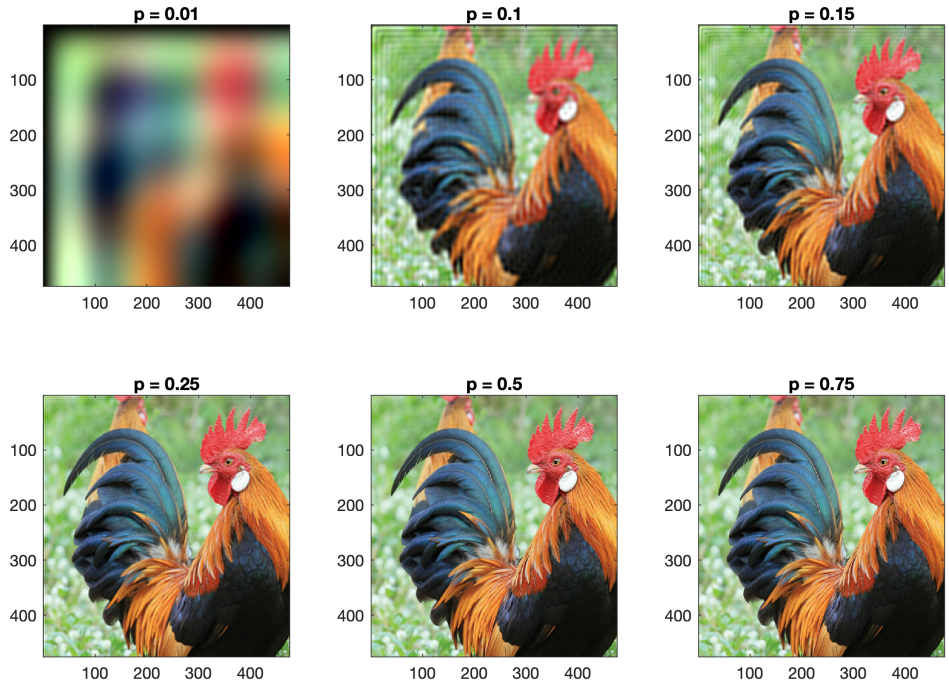

Figure 9: Images Compressed with Different p values

In Figure 9, we have included examples of our original image compressed with 6 different p values. Note here, that even p values as small as 0.15 and 0.25 give us a recognizable image (though the drop in quality is somewhat obvious when viewing the image at a larger scale). When you perform the DST and get the matrix of coefficients $\mathbf{Y}$, the low frequencies tend to dominate the overall image (and the human eye is better at detecting low frequency components), thus much of the high frequency information of an image can be cut out with a surprisingly low p value without causing the image to have a very noticeable drop in quality. For our case, we could start to notice major drops in quality at p values of around 0.25, and smaller drops in quality (when viewed at larger sizes) at $p = 0.3$. We found that having p be around 0.5 struck a good balance so that the image had a significant reduction in file size, but still has some fine details that could be seen.

If we take the number of nonzero entries (given by the `nnz` function in MATLAB) in our transformed image matrix to be the "size" of our file, we can compare it to our uncompressed file size (which has 676,875 nonzero entries) using the compression ratio formula we defined earlier to learn about the effects of different p values on the compression. Below we have tabulated the CR for $p = 0.5$, 0.3, and 0.1

p	0.5	0.3	0.1
compressed size	337725	121410	13395
CR	0.4989	0.1794	0.0198

In our view compression with $p = 0.65$ provides the best CR while still maintaining image quality (it has $CR = 0.7535$, but the image still seems to be high quality).

4 Conclusions

In this project, we have written code capable of recoloring, shifting, cropping, and compressing images using matrix manipulations. Through the development of this report, we implemented our understanding of linear algebra, and how matrices come into play in applications that many of us use every day. Hopefully, the concepts and methods outlined in this report help the readers in their development of their new picture editing application.

References

[1] APPM - CU Boulder. *Project02 - Image Manipulation in MATLAB*. Boulder, Colorado: University of Colorado Boulder, 2021.

[2] MATLAB. *version 9.8 (R2020a)*. Natick, Massachusetts: The MathWorks Inc., 2020.

A Calculations and Derivations

A.1 Switching the leftmost column with the rightmost column

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix}$$

$$\mathbf{AE} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} a_{1,4} & a_{1,2} & a_{1,3} & a_{1,1} \\ a_{2,4} & a_{2,2} & a_{2,3} & a_{2,1} \\ a_{3,4} & a_{3,2} & a_{3,3} & a_{3,1} \\ a_{4,4} & a_{4,2} & a_{4,3} & a_{4,1} \end{bmatrix}$$

$$\mathbf{EA} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} = \begin{bmatrix} a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \end{bmatrix}$$

A.2 Undoing the two dimensional DST

Given that the DST is defined by $\mathbf{Y} = \mathbf{S}\mathbf{X}_g\mathbf{S}$, we can undo the DST on $\mathbf{Y}$ and get back $\mathbf{X}_g$ by noting that $\mathbf{SS} = \mathbf{I}$:

$$\begin{aligned} \mathbf{Y} &= \mathbf{S}\mathbf{X}_g\mathbf{S} \\ \mathbf{S}\mathbf{Y} &= \mathbf{S}\mathbf{S}\mathbf{X}_g\mathbf{S} = \mathbf{I}\mathbf{X}_g\mathbf{S} = \mathbf{X}_g\mathbf{S} \\ (\mathbf{S}\mathbf{Y})\mathbf{S} &= \mathbf{X}_g\mathbf{S}\mathbf{S} = \mathbf{X}_g\mathbf{I} = \mathbf{X}_g \end{aligned}$$

Hence, to undo the two dimensional DST, we apply the formula $\mathbf{X}_g = \mathbf{S}\mathbf{Y}\mathbf{S}$: multiplying our $\mathbf{Y}$ matrix by $\mathbf{S}$ on the left and the right.

B MATLAB: `read_image.m` function

```
function [X_red, X_green, X_blue, X_gray] = read_image(imageFileName)
    X_int = imread(imageFileName);
    X_double = double(X_int);
    X_red = X_double(:,:,1);
    X_green = X_double(:,:,2);
    X_blue = X_double(:,:,3);
    X_gray = X_double(:,:,1)/3.0 + X_double(:,:,2)/3.0 +
 X_double(:,:,3)/3.0;
end
```

Published with MATLAB® R2020a

C MATLAB: `S.m` function

```
function DST_matrix = S(n)
%   Creates the DST Matrix S of any given size n using the definition
 provided in 4.1
DST_matrix = zeros(n); % initializes an nxn square matrix
for i = 1:n % loops over each row
    for j = 1:n % loops over each element in each row
        % applies the given definition for each element (i,j) in S
        DST_matrix(i,j) = sqrt(2/n) * sin( (pi*(i-1/2)*(j-1/2))/n );
    end
end

end
```

Published with MATLAB® R2020a

D MATLAB: `compress_image.m` function

```
function [X_g Y] = compress_image(p)
    X_int = imread('square.jpg'); %reads an image file
    X_double = double(X_int);
    [n n c] = size(X_double);
    % When p=0, no data are saved; when p=1 all data are saved
    for color = 1:1:3 % Looping over each color plane
        % Perform two-dimensional DST
        Y(:,:,color) = S(n)*X_double(:,:,color)*S(n);
        % Delete some of the less important values
        for i = 1:n
            for j = 1:n
                if (i+j > p*2*n)
                    Y(i,j,color) = 0;
                end
            end
        end
        %Perform the inverse diescrete sine transform and store to X_g
        X_g(:,:,color) = S(n)*Y(:,:,color)*S(n);
    end
end
```

Published with MATLAB® R2020a

E MATLAB 5.1: Image Translation

```
% Problem 5.1.1
clear; clc;
X_int = imread('rectangle.jpg');
X_double = double(X_int);
[X_red, X_green, X_blue, X_gray] = read_image('rectangle.jpg');
figure(1)
set(gcf,'position', [0 0 960 360])
    tiledlayout(1, 2,'Padding', 'normal');
    nexttile;
        imagesc(uint8(X_double)); % Display the old image
        title('Original Image')
    nexttile;
        imagesc(uint8(X_gray)); colormap('gray'); % Display the new
 image
        title('Grayscale Image')
exportgraphics(gcf, 'Project2_Q5.1.1.png', 'Resolution', 300)


% Problem 5.1.2
clear; clc;
% Reading in the image
[X_red, X_green, X_blue, X_gray] = read_image('rectangle.jpg');
% Creating a new matrix X_exposure that is an intense X_gray
X_exposure = X_gray + 80;
% Displaying the images
figure(2)
set(gcf,'position', [0 0 960 360])
    tiledlayout(1, 2,'Padding', 'normal');
    nexttile;
        imagesc(uint8(X_gray)); colormap('gray') % Display the old
 image
        title('Original Grayscale Image')
    nexttile;
        imagesc(uint8(X_exposure)); colormap('gray'); % Display the
 new image
        title('Grayscale Image with Increased Exposure')
exportgraphics(gcf, 'Project2_Q5.1.2.png', 'Resolution', 300)


% Problem 5.1.3
clear; clc;
[X_red, X_green, X_blue, X_gray] = read_image('rectangle.jpg');
X_noRed(:,:,1) = X_red * 0;
X_noRed(:,:,2) = X_green;
X_noRed(:,:,3) = X_blue + 80;
figure(3)
imagesc(uint8(X_noRed))
% title('Image with Color Intensities Changed')
exportgraphics(gcf, 'Project2_Q5.1.3.png', 'Resolution', 300)


% Problem 5.1.4
```

```
% Conceptual question: no code for this part


% Problem 5.1.5: performing a horizontal shift of 306 pixels on
 rectangle.jpg
clear; clc;
X_int = imread('rectangle.jpg'); %reads an image file
X_double = double(X_int); % converts the image to double format
X_doubleShifted = double(X_int); % converts the image to double format
 which will be shifted

for color=1:1:3 % will iterate through each of the three color
 intensity matrices 1 (Red), 2 (Green), and 3 (Blue) to apply the
 shift
    X_color = X_doubleShifted(:,:,color); % isolates one color
 intensity matrix
    [m,n] = size(X_color); % takes the size of the matrix
    c = 306; % the number of horizontal pixels that will be shifted
    E = eye(n); % creates an nxn identity matrix
    T = zeros(n); % creates an nxn matrix of zeros
    T(:,1:c) = E(:,n-(c-1):n);
    % fills in the first c columns of T with the last c columns of E
    T(:,c+1:n) = E(:,1:n-c); % fill in the rest of T with the first
 part of E
    X_doubleShifted(:,:,color) = X_color * T; % replaces the old color
 intensity matrix with the new shifted one
end
figure(5)
set(gcf,'position', [0 0 640 480])
    tiledlayout(2,2,'Padding', 'normal');
    nexttile;
        imagesc(uint8(X_double)); % Display the old image
        title('Original Image')
    nexttile;
        imagesc(uint8(X_doubleShifted)) % Display the shifted image
        title('Image Horizontally Shifted by 306 Pixels')
    nexttile;
        spy(E); % Display the original identity matrix
        title('Original Identity Matrix')
    nexttile;
        spy(T); %displays the new identity matrix
        title('Transformation Matrix')
exportgraphics(gcf, 'Project2_Q5.1.5.png', 'Resolution', 300)


% Problem 5.1.6: performing a horizontal shift of 306 pixels and a
% vertical shift of 230 pixels on rectangle.jpg
clear; clc;
X_int = imread('rectangle.jpg'); %reads an image file
X_double = double(X_int); %converts the image to double format
X_xyshift = double(X_int); %will be used for the shifts

for color = 1:1:3 %iterate through each of the three colors
```

```
    X_color = X_xyshift(:,:,color); %isolates one color intensity
 matrix
    [m,n] = size(X_color); %takes the shape of the matrix
    r = 230; %the number of vertical pixels that will be shifted
    c = 306; %the number of horizontal pixels that will be shifted
    Er = eye(m); %makes an mxm identity matrix
    Ec = eye(n); %makes an nxn identity matrix
    Tr = zeros(m); %gives an mxm matrix of zeros
    Tc = zeros(n); %gives an nxn matrix of zeros
    Tr(1:r,:) = Er(m-(r-1):m,:);
    % fill in the first r rows of T with the last r rows of E
    Tc(:,1:c) = Ec(:,n-(c-1):n);
    % fill in the first c columns of T with the last c columns of E
    Tr(r+1:m,:) = Er(1:m-r,:);
    % fill in the rest of T with the first part of E
    Tc(:,c+1:n) = Ec(:,1:n-c);
    % fill in the rest of T with the first part of E
    X_xyshift(:,:,color) = Tr * X_color * Tc;
    %makes the new shifted matrix intensity matrix
end
% Displaying the images and identity matrices
figure(6)
set(gcf,'position', [0 0 640 480])
    tiledlayout(2,2,'Padding', 'normal');
    nexttile;
        imagesc(uint8(X_double)); % Display the old image
        title('Original Image')
    nexttile;
        imagesc(uint8(X_xyshift)) % Display the shifted image
        title(["Image Shifted by", "306 Pixels Horizontally and 230
 Pixels Vertically"])
    nexttile;
        spy(Tc); % Display the original identity matrix
        title('Horizontal Transformation Matrix')
    nexttile;
        spy(Tr); %displays the new identity matrix
        title('Vertical Transformation Matrix')
exportgraphics(gcf, 'Project2_Q5.1.6.png', 'Resolution', 300)


% Problem 5.1.7: flips rectangle.jpg upside down
clear; clc;
X_int = imread('rectangle.jpg'); %reads an image file
X_double = double(X_int); %converts the image to double format

X_flipped = double(X_int); %will be used for the shifts
[m, n, colors] = size(X_double);
E = eye(m);
T = zeros(m);
% fill the first m columns of T with the last m columns of E
T(:,m:-1:1)=E(:,1:1:m);
% Loop through color planes and apply the flip transformation matrix
 to
% each color intensity matrix
```

```
for color = 1:1:colors
    X_flip(:,:,color) = T * X_double(:,:,color);
end
figure(7)
set(gcf,'position', [0 0 640 480])
    tiledlayout(2,2,'Padding', 'normal');
    nexttile;
        imagesc(uint8(X_double)); % Display the old image
        title('Original Image')
    nexttile;
        imagesc(uint8(X_flip)) % Display the shifted image
        title('Image Flipped Upside Down')
    nexttile;
        spy(E); % Display the original identity matrix
        title('Original Identity Matrix')
    nexttile;
        spy(T); %displays the new identity matrix
        title('Transformation Matrix')
exportgraphics(gcf, 'Project2_Q5.1.7.png', 'Resolution', 300)


% Problem 5.1.8: trying to transpose an image matrix
clear; clc;
X_int = imread('rectangle.jpg'); %reads an image file
X_double = double(X_int);
for color = 1:1:3
    X_doubleTranspose(:,:,color) = transpose(X_double(:,:,color));
end
figure(8)
set(gcf,'position', [0 0 960 360])
    tiledlayout(1, 2,'Padding', 'normal');
    nexttile;
        imagesc(uint8(X_double)); % Display the old image
        title('Original Image')
    nexttile;
        imagesc(uint8(X_doubleTranspose)) % Display the transposed
 image
        title('Transposed Image')
exportgraphics(gcf, 'Project2_Q5.1.8.png', 'Resolution', 300)

% Problem 5.1.9: Create a black border around the image
X_int = imread('rectangle.jpg'); %reads an image file
X_double = double(X_int); %converts the image to double format
X_doubleshift = double(X_int); %will be used for the shifts

for color = 1:1:3 %iterate through each of the three colors
    X_color = X_doubleshift(:,:,color); %converts the intensity to one
 matrix
    [m, n] = size(X_color); %takes the shape of the matrix
    r = 50; %horizontal border size
    c = 50; %vertical border size
    Er = eye(m); %makes an mxm identity matrix
    Ec = eye(n); %makes an nxn identity matrix
    Tr = zeros(m); %gives an mxm matrix of zeros
```

```
    Tc = zeros(n); %gives an nxn matrix of zeros
    Tr(r:m-r,:) = Er(r:m-r,:); %creates the black border for the rows
    Tc(:,c:n-c) = Ec(:,r:n-c); %creates the black border for the
 columns
    X_doubleshift(:,:,color) = Tr * X_color * Tc;
    %makes the new shifted matrix intensity matrix
end
figure(9)
set(gcf,'position', [0 0 640 480])
    tiledlayout(2,2,'Padding', 'normal');
    nexttile;
        imagesc(uint8(X_double)); % Display the old image
        title('Original Image')
    nexttile;
        imagesc(uint8(X_doubleshift)) % Display the shifted image
        title('Image Flipped Upside Down')
    nexttile;
        spy(Tr); % Display the original identity matrix
        title('Vertical Transformation Matrix')
    nexttile;
        spy(Tc); %displays the new identity matrix
        title('Horizontal Transformation Matrix')
exportgraphics(gcf, 'Project2_Q5.1.9.png', 'Resolution', 300)
```

Published with MATLAB® R2020a

F MATLAB 5.2: Image Compression

```
% Problem 10
% Function S was declared in a separate .m file


% Problem 11: For the 5x5 matrix S, Show that SS=I_5
S(5)*S(5);

% Problem 12: Check the Undoing of the DST
[X_red, X_green, X_blue, X_gray] = read_image('square.jpg');
[n n] = size(X_gray);
Y = S(n)*X_gray*S(n);
X_t = S(n)*Y*S(n);
imagesc(uint8(X_t)); colormap('gray');

% Problem 13/14: Simplified Compression with several p values
% (function to compress was defined in compress_image.m)
clear; clc;
figure(13)
set(gcf,'position', [0 0 720 480])
    tiledlayout(2, 3,'Padding', 'normal');
    nexttile;
        p=0.01;
        [X_g Y] = compress_image(p);
        imagesc(uint8(X_g)); % p=0.01
        title('p = 0.01')
    nexttile;
        p=0.1;
        [X_g Y] = compress_image(p);
        imagesc(uint8(X_g)); % p=0.1
        title('p = 0.1')
    nexttile;
        p=0.15;
        [X_g Y] = compress_image(p);
        imagesc(uint8(X_g)); % p=0.15
        title('p = 0.15')
    nexttile;
        p=0.25;
        [X_g Y] = compress_image(p);
        imagesc(uint8(X_g)); % p=0.25
        title('p = 0.25')
    nexttile;
        p=0.5;
        [X_g Y] = compress_image(p);
        imagesc(uint8(X_g)); % p=0.5
        title('p = 0.5')
    nexttile;
        p=0.75;
        [X_g Y] = compress_image(p);
        imagesc(uint8(X_g)); % p=0.75
        title('p = 0.75')
exportgraphics(gcf, 'Project2_Q5.2.13.png', 'Resolution', 300)
```

```
% Problem 15
% number of nonzero entries in the uncompressed Y
p = 1;
[X_g Y] = compress_image(p);
nnz(Y)

p = 0.5;
[X_g Y] = compress_image(p);
nnz(Y)
nnz(Y)/676875

p = 0.3;
[X_g Y] = compress_image(p);
nnz(Y)
nnz(Y)/676875

p = 0.1;
[X_g Y] = compress_image(p);
nnz(Y)
nnz(Y)/676875
```

Published with MATLAB® R2020a