APPM 2360
Project 2

Peter Braza, Evan Gassiot, Niko Pappas

Chapter 1

1.1 Introduction

Hello again friends! We are here to help you with your latest business venture. We will be using our Matlab skills to manipulate images for your app. We will show you all the code you will need to implement this on the app as well as providing descriptions of what the code we have written is doing.

1.2 Knowns

We will be showing proof of working functions by using the supplied files `rectangle.jpg` and `square.jpg`. We will use rectangle for the image translation sections as it shows that any size of an image can be shifted however the user wants and we will use the square image for the compression since our compression function requires a square image to work.

Figure 1.1: `rectangle.jpg`

Figure 1.2: `square.jpg`

Chapter 2

2.1 Color Modification

We will first focus on modifying image colors. We will be converting an image to gray scale and changing an image's exposure. We will also modify the individual color channels of an image to increase or decrease the intensity of individual colors.

Note:-

When reading an image with Matlabs `imread` command we receive and image whose size is $m \times n \times c$ where m & n are the rows and columns of the image matrix respectively and c is the number of color channels in the image (3). Generally we won't be modifying m & n, we will instead modify the c channel for all of our image recoloring and translation.

We will modify `rectangle.jpg` and convert it into a gray scale image. We will accomplish this by multiplying all of the color channels by $\frac{1}{3}$. After converting the image to a double and then computing the scalar multiplication we are then left with this image:

Figure 2.1: Grayscaled `rectangle.jpg`

Now we have this nice looking gray scale image that we can modify even more to see what other types of changes a person may want to make to their image. We can simulate over exposing the image by scalar multiplying the image by some value s. For this example, we will let $s = 2$, after multiplication and converting back to `uint8` we are left with:

Figure 2.2: Grayscaled `rectangle.jpg` after over-exposure

We can also change the values of the color intensities individually by altering each channel with different values than just using general scalar multiplication. If we were to delete the red channel altogether and increase the blue channel by 80 on the original image we get:

Figure 2.3: `rectangle.jpg` with no red channel and increased blue

2.2 Image Shifting

In this section we will be using matrix multiplication to translate an image side to side and up and down. First we will discuss the types of matrices we will use to accomplish these image shifts. We will use a modified **I** matrix since we don't actually want to change any of the color data stored at these positions. If we wanted to swap the first and last columns of a 4×4 matrix **A** our translation matrix (**T**) would look like this:

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

With the first and last columns of swapped if we were to calculate **AT** our result would be the **A** matrix with its first and last columns swapped. Now we can use a similar strategy on larger matrices to compute massive image shifts.

Note:-

All of the proofs for the translation matrices will be in the appendix as well as explaining other possibilities the app using similar techniques.

If a user wanted to shift their image to the right by 306 pixels they would get an image that would look like this:

Figure 2.4: Shifted `rectangle.jpg`

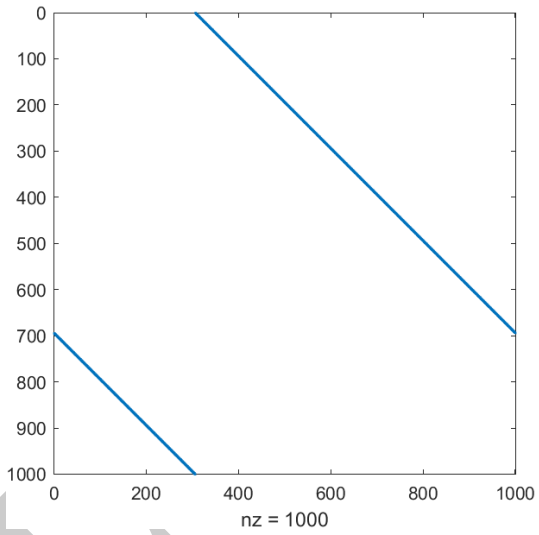

Figure 2.5: Image Translation matrix using `spy()`

If a user wanted to have both a vertical and horizontal shift on their image this could also be done by slightly modifying the code we use for the horizontal shift to include a vertical shift as well. We will need 2 matrices since `rectangle.jpg` is not square, one being the size of the number of rows in the image and another being the size of the number of columns in the image. We can use `spy()` again to view the matrices:

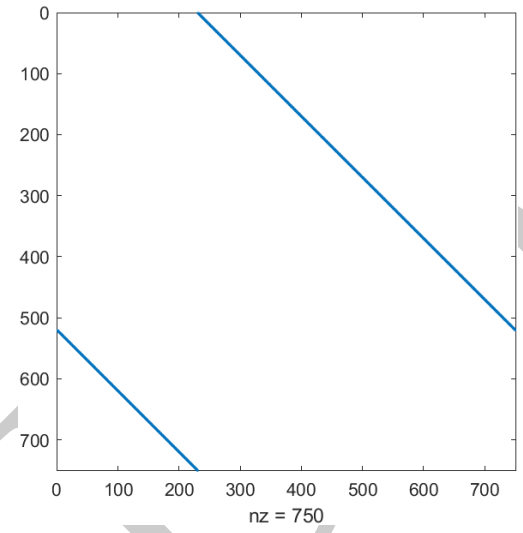

Figure 2.6: Row Translation Matrix

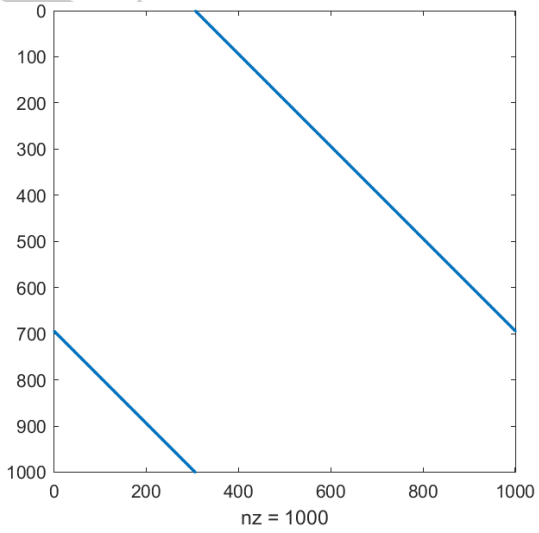

Figure 2.7: Column Translation Matrix

By using both of these matrices we can multiply the left hand side of the image matrix with the row translation matrix and the right hand side of the image with the column translation matrix our result is:

Figure 2.8: `rectangle.jpg` shifted horizontally and vertically

2.3 Other Image Modifications

Some of the other image modifications a user may want to make to their image may include, flipping it, rotating it, and cropping the image. If they wanted to flip their image over itself we can tell Matlab to take an identity matrix and then flip the matrix by reading the identity backwards and then using matrix multiplication to calculate the final image.

Figure 2.9: Flipped `rectangle.jpg`

If the user wanted to flip the image 90° we can transpose the image matrix to accomplish this rotation. This swaps the rows and columns of the matrix so our new matrix is now of size $n \times m \times c$ or in our case $1000 \times 750 \times 3$. The easiest way to visualize the process for a rectangular matrix is to imagine taking a piece of paper and pinning it at the top left hand corner and then flipping the image over the diagonal.

Figure 2.10: Transposed `rectangle.jpg`

The image we get as an output of this makes sense as the top left hand corner has not moved yet the rest of the image has been flipped over the diagonal leaving us with a rotated and flipped image.

In order to get a cropped image we will 'frame' the image by converting all the pixels we do not want to appear in the image to be black. For our example we will take `rectangle.jpg` and delete the first 100`px` from all sides. To modify our image we will take the **I** matrix and remove the first and last 100 entries from the matrix and then multiply the image by our new matrix. Our identity matrices look like this:

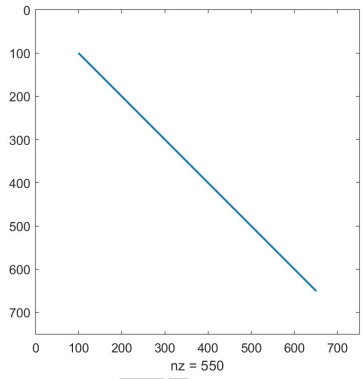

Figure 2.11: Row Modification Matrix

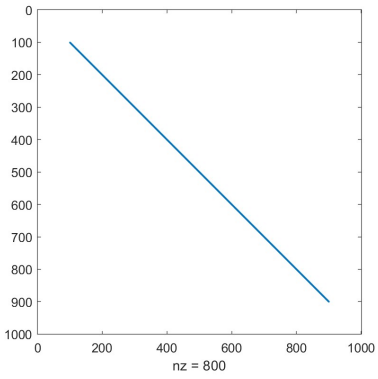

Figure 2.12: Column Modification Matrix

After multiplying our image by the modified identities the user's new image now looks like:

Figure 2.13: Cropped `rectangle.jpg`

2.4 The combinations are endless...

This section is just an example of showing that the user is able to combine many of these image modifications into 1 image like such:

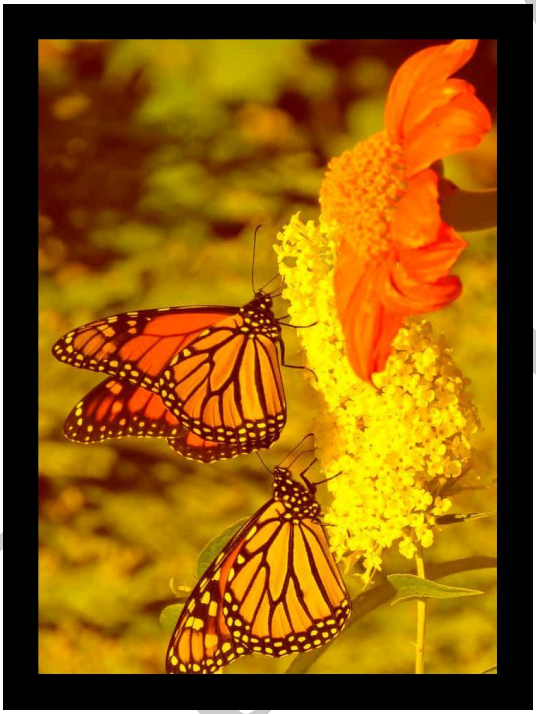

Figure 2.14: Rotated & recolored & cropped image

Figure 2.15: Flipped & shifted image

Chapter 3

3.1 Image Compression

In this section we will be using the Discrete Sine Transform. We will be using a Matlab function to calculate the DST matrix so we do not have to compute it by hand for large matrices. The DST we will be using for our compression is:

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

In order to convert back and forth from the **S** matrix and an image we will prove that **S** is it's own inverse (also called involutory).

Note:-

The full proof showing that **S** is it's own inverse is in the appendix but we will calculate a smaller example here as well.

We will let **E** be some 3×3 matrix. If we were to apply the DST on **E** our result would be some matrix **S**. Computing **S**·**S** yields **I** showing us that **S** is its own inverse which means that when we convert an image back from the DST we can easily replace $\mathbf{S}^{-1}$ with **S**.

Example 3.1.1 ($\mathbf{S} = \mathbf{S}^{-1}$)

$$\mathbf{E} = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix} \xrightarrow{DST} \begin{bmatrix} 0.2113.. & 0.5774.. & 0.7887.. \\ 0.5774.. & 0.5774.. & -0.5774.. \\ 0.7887.. & -0.5774.. & 0.2113.. \end{bmatrix} = \mathbf{S}$$

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

Now that we have our DST we can now process `square.jpg` as an example. Then we will loop through our image matrix using a predefined p value. If an image has a frequency greater than the defined p value we will replace it with 0 since it is outside of the frequency range that humans observe. By converting these values to 0 we can reduce the file size.

Figure 3.1: Compressed Image with p value of 1

Figure 3.2: Compressed Image with p value of 0.3

Figure 3.3: Compressed Image with p value of 0.1

Figure 3.4: Compressed Image with p value of 0.01

From this we can see that the lower the p value the more we prune from image which causes the image quality to drop drastically. When $p = 1$ Matlab tells us that none the image was compressed whereas when $p = 0.01$ Matlab says that over 99.9% of the image contained a pixel that needed to be pruned which contributes to the massive loss in quality on the 4^{th} image.

Note:-

p values greater than 1 do not exist in the image since the values are percentages written as decimals. There is no different between an image with a p value of 4 compared to an image of p value of 1.

The p value can become small before noticing a significant quality loss because the compression removes redundant information. The human eye can detect changes in lower frequencies better than higher frequencies so removing the higher frequencies in the image is the best way to reduce the file size while still allowing the image to look unmodified. A small p value removes a lot more frequencies thus altering the image more drastically and making it quite obvious to the observer.

Chapter 4

4.1 Conclusion

words and then more words and then a period

Appendix A

Question 1: 5.1.1

Write a MATLAB function to read in an image file by name and return the double form of the grayscale matrix representing the picture as well as the red, green, and blue color intensities. Use this code to display grayscale and color versions of the file `rectangle.jpg`

```
X = imread('rectangle.jpg'); %% read in image
X_double = double(X); %% convert image to double form
X_gray= X_double(:,:,1)/3.0 + X_double(:,:,2)/3.0 + X_double(:,:,3)/3.0; %% convert to gray
    scale
imagesc(uint8(X_gray)) %% publish image

imwrite(uint8(X_gray),'grayscaleRectange5_1_1.jpg') %% write new picture to directory
```

Question 2: 5.1.2

When a grayscale image is stored as a matrix, it is a simple task to alter the exposure of the image. Remembering that the values of the matrix represent the pixel intensity, increase the exposure in `rectangle.jpg` so that the resulting image appears more "whited out" than the original.

```
X = imread('grayscaleRectange5_1_1.jpg'); %% Reading the grayscale image
X_double = double(X); %% convert to double

X_2intensity(:,:,1) = X_double(:,:,1)*2; %% changing the RBG channels to have double intensity
X_2intensity(:,:,2) = X_double(:,:,2)*2;
X_2intensity(:,:,3) = X_double(:,:,3)*2;

figure(); %% Creating a figure
hold on
subplot(1,2,1); %% Left Subplot
imagesc(uint8(X_double))
title('Original Image')

subplot(1,2,2); %% Right Subplot
title('2x Image Intensity')
imagesc(uint8(X_2intensity))
title('2x Image Intensity')
hold off

imwrite(uint8(X_2intensity),"increasedIntensity5_1_2.jpg")
```

Question 3: 5.1.3

Now change the colors in the color image. Remove all of the red by scalar multiplying the matrix containing the red intensities by 0. Leave the green intensities unchanged but increase the blue intensity by 80 units.

```
X = imread('rectangle.jpg'); %% Read in the file
[rows, columns, channels] = size(X); %% This call returns the image dimensions as a row vector
```

```
X_double = double(X); %% Convert values to doubles
eighty = 80*ones(rows,columns); %% Matrix where every value is 80

X_modified(:,:,1) = X_double(:,:,1) .* 0; %% Multiplying the red channel values by 0
X_modified(:,:,2) = X_double(:,:,2);
X_modified(:,:,3) = X_double(:,:,3) + eighty; %% Adding 80 to every value in the matrix

imagesc(uint8(X_modified)) %% Printing the image
imwrite(uint8(X_modified),"NoRed80Blue5_1_3.jpg") %% Writing the image to a file
```

Question 4: 5.1.4

Given an image that is 4 pixels by 4 pixels, so that the grayscale representation of the image is a 4×4 matrix called $\mathbf{A}$, what matrix $\mathbf{E}$ would you multiply A by in order to switch the leftmost column of pixels with the rightmost column of pixels? Should you multiply $\mathbf{A}$ by $\mathbf{E}$ on the right or the left (*i.e.*, would $\mathbf{AE}$, $\mathbf{EA}$, or both give the desired result)? It is probably beneficial to experiment with this on paper before providing your answer in your report.

Solution: We will first use the identity matrix to help us solve this problem as we know that anything multiplied by the identity is itself. We will refer to the identity matrix as $\mathbf{I}$ from now on. We will take the standard identity matrix:

$$\mathbf{I}_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We will now modify the columns in $\mathbf{A}$ and the changes made here will affect $\mathbf{A}$ when we compute $\mathbf{AE}$. By swapping columns 1 & 4 of $\mathbf{I}$ we then have:

$$\mathbf{E}_4 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

We can test our work by using the sample matrix:

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

$$\mathbf{AE} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 4 & 2 & 3 & 1 \\ 8 & 6 & 7 & 5 \\ 12 & 10 & 11 & 9 \\ 16 & 14 & 15 & 13 \end{bmatrix}$$

And there we have it, we now have $\mathbf{AE}$ which is $\mathbf{A}$ but with columns 1 & 4 swapped.
The code below demonstrates the proof

```
A = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]; %% Sample Matrix
E = eye(4); %% Identity 4x4

E(:,[1 4]) = E(:,[4 1]); %% Swap columns 1 & 4 with each other, this logic also applies to
    swapping rows

AE = A*E; %% We left multiply E on the right hand side of A which swaps A's columns, we left
    hand multiplication would swap rows
disp(AE) %% Display A*E's product
```

Note:-

If we were to compute **EA** the product would be slightly different, it would be **A** but with rows 1 & 4 swapped instead of the columns. We will make use of LHS and RHS multiplication later when we are looking to swap both rows and columns.

Question 5: 5.1.5

The matrix from `rectangle.jpg` is not square. We can, however, still apply matrices to shift the pixels. Find a matrix that will perform a horizontal shift of 306 pixels to `rectangle.jpg`.

Solution: In order to apply a shift to an image matrix we will again start out with the **I** matrix as it allows us to solely focus on the order of the 1's. We will use a smaller example to order to convey the idea for a much larger example with an actual image. We will use the matrix **A** as our example here. We will let **A**:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

We are going to attempt to alter **A** so we get:

$$\mathbf{A} = \begin{bmatrix} 4 & 5 & 1 & 2 & 3 \\ 4 & 5 & 1 & 2 & 3 \\ 4 & 5 & 1 & 2 & 3 \end{bmatrix}$$

This matrix has had its columns shifted 2 to the right meaning our transformation matrix (**T**) should also have this shift in it. We also need to determine the size of our transformation matrix **T**. First off we need to figure out the size of the **T** matrix. Our current matrix is a size 3×5 but we will generalize this so we can better understand it for the MatLab coding later on.

$$size(\mathbf{A}) = m \times n$$

The size of the matrix we multiply in must be the same size as the output in order for them to be equivalent. Therefore, the number of columns of **A** must be equal to the number of rows in matrix **T**. So the solution for the size of the transformation matrix must be:

$$size(\mathbf{A} \times \mathbf{T}_{\mathrm{n}}) = (m \times n)(n \times n) = m \times n$$

Going back to our example our **T** matrix must be of size 5×5. We will now build our **T** matrix but we will shift it's columns 2 to the right and loop the 4^{th} and 5^{th} columns back to the front of the matrix so we are left with:

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

If we now compute the product of **AT** our result is:

$$\mathbf{AT} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 4 & 5 & 1 & 2 & 3 \\ 4 & 5 & 1 & 2 & 3 \\ 4 & 5 & 1 & 2 & 3 \end{bmatrix}$$

We now have found a solution for our shifted matrix. We can generalized the shifted matrix to be **I** with its columns shifted **r** times to the right making sure to loop the columns back around to the left hand side when they 'fall off'.

When we apply what we learned from our example to our `rectangle.jpg` with an **r** value of 306`px` we get:

Figure A.1: Image shifted 306 pixels to the right

From this we learned that we can shift images horizontally by shifting the identity matrix columns and using RHS multiplication we could produce an image with a vertical pixel shift by using a similar **I** matrix but it's size would need to be $(m \times m)$ and we would then apply LHS multiplication to receive a vertically shifted image.

$$\mathbf{B} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

$$size(\mathbf{T}) = (3 \times 3)$$

$$\mathbf{TB} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \end{bmatrix} = \begin{bmatrix} 3 & 3 & 3 & 3 & 3 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \end{bmatrix}$$

This is a vertical shift of our **B** matrix with an **r** value of 2. This shows that it is also very simple to do a vertical translation. Both of these matricies could be combined into making an image that is both shifted vertically and horizontally at the same time.

The code below shifts by given image by 306 pixels.

```
A = imread('rectangle.jpg'); %% Read the image
X_double = double(A); %% Convert to double
[m, n, c] = size(A); %% Get the size data from the image
r = 306; %% # of pixels to shift
E = eye(n); %% Identity
T = zeros(n);

T(:,1:r) = E(:,n-(r-1):n); %% For columns 1 to r replace them with the identity columns from the
    (n - r - 1)th column to the last column
T(:,r+1:n) = E(:,1:n-r); %% From column r + 1 to the last column replace it with the identity
    from 1 to n - r

writematrix(T,'matrix.txt') %% Write the matrix to a txt file
figure();
hold on
spy(T) %% Plot the matrix on a graph with a vertical line at 306
xline(306,'LineWidth',2,'Color','red')
hold off

X_double(:,:,1) = X_double(:,:,1) * T; %% Modify the image matrix to have the shifted pixels
X_double(:,:,2) = X_double(:,:,2) * T;
X_double(:,:,3) = X_double(:,:,3) * T;

imagesc(uint8(X_double)) %% Print the image

imwrite(uint8(X_double),'shiftedRectangle5_1_5.jpg')
```

Question 6: 5.1.6

Apply transformations to the original matrix from `rectangular.jpg` that result in both a horizontal and vertical shift. Shift the image 306 pixels horizontally and 230 pixels vertical

```
A = imread('rectangle.jpg'); %% Read the image
X_double = double(A); %% Convert to double
[m, n, c] = size(A); %% Get the size data from the image
r = 306; % # of pixels to shift horizontally
v = 230; % # of pixels to shift vertically
E = eye(n); % Identity the size of the columns of A
B = eye(m); % Identity the size of the rows of A
T = zeros(n);
S = zeros(m);

T(:,1:r) = E(:,n-(r-1):n); % For columns 1 to r replace them with the identity columns from the
    (n - r - 1)th column to the last column
T(:,r+1:n) = E(:,1:n-r); % From column r + 1 to the last column replace it with the identity
    from 1 to n - r

S(:,1:v) = B(:,m-(v-1):m); % For columns 1 to v replace them with the identity columns from the
    (m - v - 1)th column to the last column
S(:,v+1:m) = B(:,1:m-v); % From column v + 1 to the last column replace it with the identity
    from 1 to m - v

X_double(:,:,1) = S * X_double(:,:,1) * T; % Modify the image matrix to have the shifted pixels
X_double(:,:,2) = S * X_double(:,:,2) * T;
X_double(:,:,3) = S * X_double(:,:,3) * T;

imagesc(uint8(X_double)) % Print the image

imwrite(uint8(X_double),'HorzAndVert5_1_6.jpg')
```

Question 7: 5.1.7

Determine what matrix would be required to flip an image upside down. Using that transformation, flip `rectangle.jpg` upside down.

```
A = imread('rectangle.jpg');
X_double = double(A);
[m, n, c] = size(A);
I = eye(m);
spy(I)

T = I(:,m:-1:1); %% loop backwards through identity matrix and make T matrix
spy(T)

X_double(:,:,1) = T * X_double(:,:,1); %% Modify the image matrix to have the shifted pixels
X_double(:,:,2) = T * X_double(:,:,2);
X_double(:,:,3) = T * X_double(:,:,3);

imagesc(uint8(X_double))
imwrite(uint8(X_double), 'flipRectangle.jpg')
```

Question 8: 5.1.8

What should transposing an image matrix do? Try it with `rectangle.jpg`.

```
A = imread('rectangle.jpg');

%The image matrix here is m x n x c, what permute does is this:
%...[2 1 3] says to swap m & n but leave c where it already is.

B = permute(A, [2 1 3]); %
imagesc(uint8(B));
imwrite(B,'transposedRectangle.jpg')
```

Question 9: 5.1.9

We next want to crop the image `rectangle.jpg`, that is, eliminate part of the image (turn pixels off, make the color black, change the color intensity to 0) around the edges. This effectively will put a black border around the image.

```
A = imread('rectangle.jpg'); %% Read the image
X_double = double(A); %% Convert to double
[m, n, c] = size(A); %% Get the size data from the image
E = eye(n); % Identity the size of the columns of A
B = eye(m); % Identity the size of the rows of A

l = 100;
r = 100;
t = 100;
b = 100;
for i = 1:l
    E(i,i) = 0;
end
for j = n-r:n
    E(j,j) = 0;
end
for k = 1:t
    B(k,k) = 0;
end
for h = m-b:m
    B(h,h) = 0;
end

X_double(:,:,1) = B * X_double(:,:,1) * E; % Modify the image matrix to have the removed pixels
X_double(:,:,2) = B * X_double(:,:,2) * E;
X_double(:,:,3) = B * X_double(:,:,3) * E;

figure();
imagesc(uint8(X_double)) % Print the image

imwrite(uint8(X_double),'CroppedImage.jpg')
```

Question 10: 5.2.10

Write a function that returns the matrix $\mathbf{S}$ of any given size. Also, verify that, for the (5×5) matrix $\mathbf{S}$, $\mathbf{S} = \mathbf{S}^{-1}$. That is, show that $\mathbf{SS} = \mathbf{I}_5$

```
I = eye(5);
disp(sReturn(I)*sReturn(I))
I2 = eye(5);
disp(sReturn(I2))
function [s] = sReturn(matr)
%since in this project the matrices are guaranteed to be square (nxn), I
%can get the size of just one dimension.
n = size(matr, 1);
%I preallocate the memory here to have an nxn matrix because this is faster
%than reallocating the memory each time I add an element to the array
s = zeros(1, n);
mult = sqrt(2/n);
for i = 1:n
    for j = 1:n
        s(i,j) = mult*sin((pi/n)*(i-.5)*(j-.5));
    end
end
end
```

Question 11: 5.2.13

Perform our simplified JPEG-type compression on the image of the rooster in `square.jpg`. Use multiple p values to compare how the quality can change.

```
img = imread('square.jpg');
double_img = double(img);
imgRed= double_img(:,:,1);
imgGreen = double_img(:,:,2);
imgBlue = double_img(:,:,3);

%Compressing the red channel
imRedComp = sReturn(imgRed)*imgRed*sReturn(imgRed);
newRedFrequencies = filter(imRedComp);
finalRed = sReturn(newRedFrequencies)*newRedFrequencies*sReturn(newRedFrequencies);
%compressing the green channel
imGreenComp = sReturn(imgGreen)*imgGreen*sReturn(imgGreen);
newGreenFrequencies = filter(imGreenComp);
finalGreen = sReturn(newGreenFrequencies)*newGreenFrequencies*sReturn(newGreenFrequencies);
%compressing the blue channel
imBlueComp = sReturn(imgBlue)*imgBlue*sReturn(imgBlue);
newBlueFrequencies = filter(imBlueComp);
finalBlue = sReturn(newBlueFrequencies)*newBlueFrequencies*sReturn(newBlueFrequencies);

finalImg(:,:,1) = finalRed;
finalImg(:,:,2) = finalGreen;
finalImg(:,:,3) = finalBlue;

figure();
hold on
subplot(1,2,1);
imagesc(uint8(double_img));
title('Original Image');

subplot(1,2,2);
imagesc(uint8(finalImg));
title('Compressed image');
hold off
imwrite(uint8(finalImg),"Pvalue=.05.jpg");

function [compressed_frequency] = filter(img)
p = .05; %specific p value
n = size(img,1);
compressed_frequency = zeros(1, n);
for i = 1:n
    for j = 1:n
        if((i+j)>p*2*n)
            compressed_frequency(i, j) = 0;
        else
            compressed_frequency(i, j) = img(i, j);
        end
    end
end
end

function [s] = sReturn(matr)
%since in this project the matrices are guaranteed to be square (nxn), I
%can get the size of just one dimension.
n = size(matr, 1);
%I preallocate the memory here to have an nxn matrix because this is faster
%than reallocating the memory each time I add an element to the array
s = zeros(1, n);
mult = sqrt(2/n);
for i = 1:n
    for j = 1:n
        s(i,j) = mult*sin((pi/n)*(i-.5)*(j-.5));
    end
end
end
```