Jordan Jalufka, Pedro Perin-Cruz, Ross Panning

Image Manipulation in MATLAB

Jordan Jalufka, Pedro Perin-Cruz, Ross Panning

1. Introduction

In this day and age, technological reliance and assistance is a need that is only becoming more prominent as time goes on. In a world where every year, one is able to do more and more just out of their own smartphone or tablet, image and editing apps are, undeniably, in the epicenter of such advancements; the ability to obtain professional quality photos through editing, manipulating and altering a photo you've taken directly from your phone has become so widespread and hassle-free that most people are now able to do it.

Consequently, it is a major industry counting with an ever-growing number of competitors, and one that is certainly not going away anytime soon. Our task is to help a business group build the code needed for an image editing *app* using our knowledge of differential equations and linear algebra. Here, we will manipulate matrices to perform image translation and compression on a few samples.

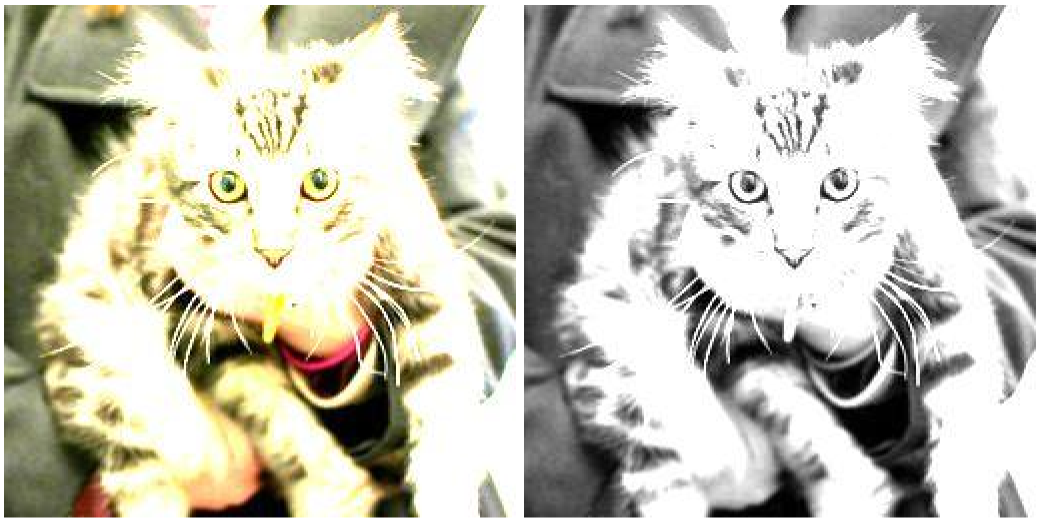
2. Image Translation

First, we are tasked with reading two image files, *photo1.jpg* and *photo2.jpg* and returning the double form of their respective greyscale matrices, to obtain the following two images.

and

Reference Appendix Section #5.1.1 for code and figures.

Then, taking into account that the matrix values represent pixel intensity, we manipulated *photo2.jpg* so as to increase its exposure. It is not clear whether or not it is requested that we over expose the colored image or the grayscale image, so we have done both. The results, along with the original image, are as follows:



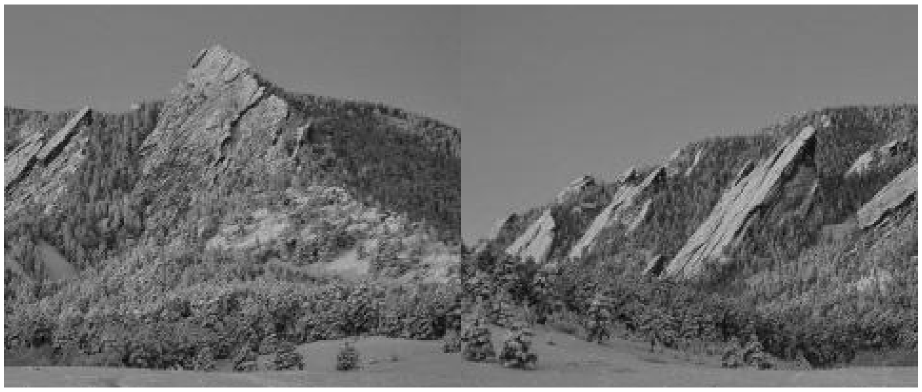
Reference Appendix Section #5.1.2 for code and figures.

Next, in order to switch the leftmost column of pixels with the rightmost column of pixels, we multiply a 4 x 4 matrix A, by E—with E as demonstrated below:

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

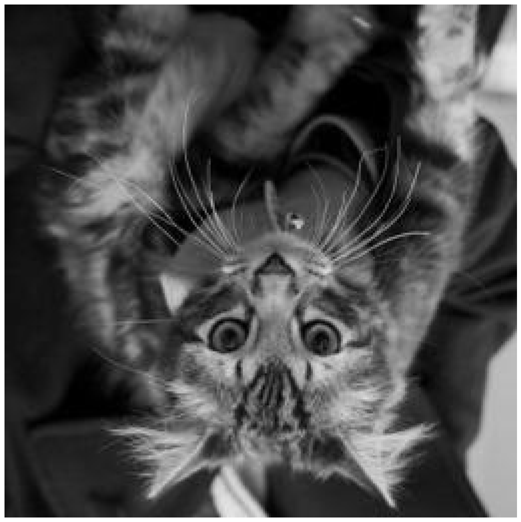
Where E is multiplied to the right of the original matrix. I.e. E* the above matrix equals the matrix where the first and last columns are swapped. It is important to note A*E would not be consistent across all values of E.

Despite the matrix from *photo1.jpg* not being square, we have performed a horizontal shift of 240 pixels and a vertical shift of 100 pixels, through manipulating the columns of I. The resulting matrix yields the following shifted image.



Reference Appendix Section #5.1.4/5 for code and figures.

Using the reverse identity matrix, we managed to flip the image upside down as displayed below. The code and *spy()* function are referenced in the corresponding appendix section.



Reference Appendix Section #5.1.6 for code and figures.

For the next part of the project, we transposed the image matrix. The image below is what we received. Based on the rotation observed, this transposition is indeed what we expected to obtain, given that the operation swaps the matrix's columns with its rows—and what this means for the corresponding image's orientation is a simple counterclockwise rotation.

Reference Appendix Section #5.1.7 for code and figures.

The last part of this section had us reconnecting and using matrix operations to "fix" the distorted *mount2.jpeg*. Using the code referenced in the appendix section, we ended up with the below image—indeed a grayscale reconstruction of *photo1.jpg*.

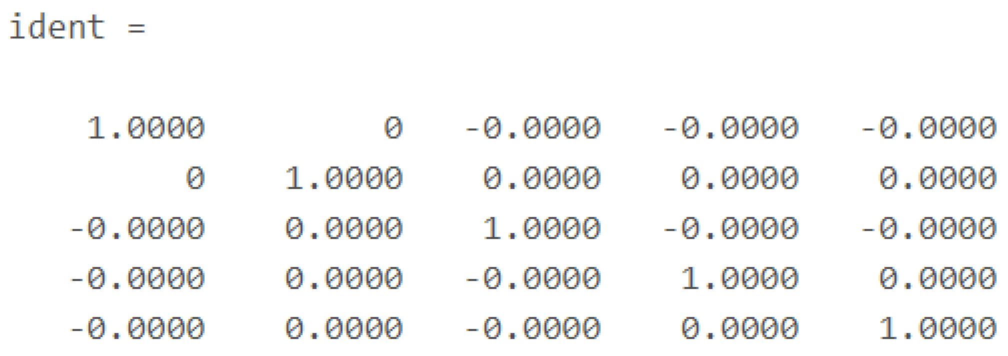
Reference Appendix Section #5.1.8 for code and figures.

3. Image Compression

For the first part of the image compression section, we wrote a function that returns the matrix S of any given size. The code is referenced in the appendix section.

Reference Appendix Section #5.2.9 for code and figures.

We then verified that the S matrix was an inverse of itself, therefore showing that SS equals the identity matrix. The below matrix shows that this is indeed the case. The code for this part is also referenced in the appendix section.



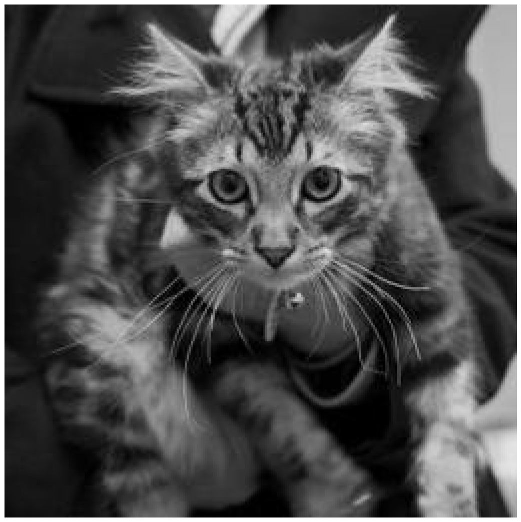
```
ident =

    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
```

Reference Appendix Section #5.2.10 for code and figures.

After that, one can see the DST was applied and then unapplied via multiplying by S(256) on both sides. We received the image below as a result and its respective code is referenced in the appendix section.

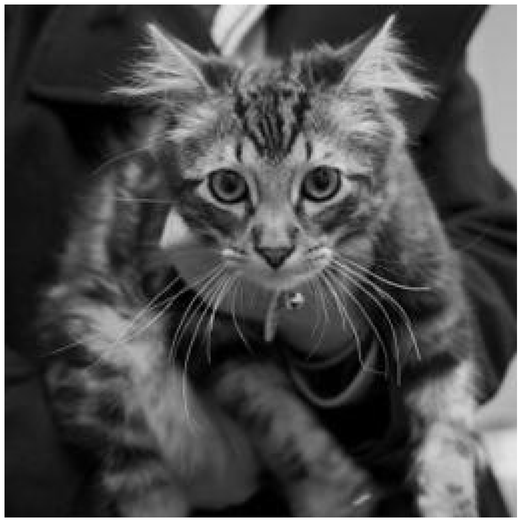
Reference Appendix Section #5.2.11 for code and figures.

For part 12, we used a code template (also referenced in the appendix) for compression. In this example we tested several *p* values and obtained the following images as a result.

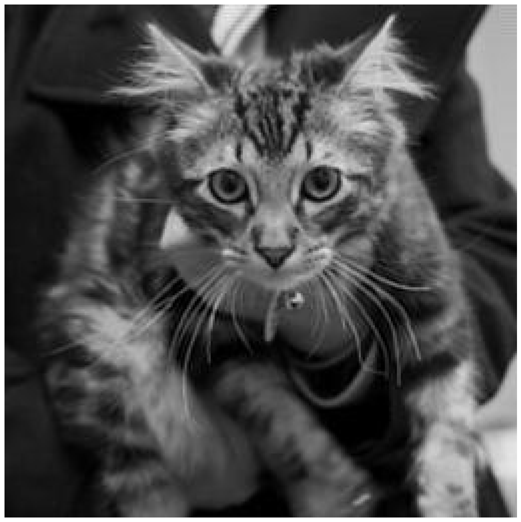


P = 2.0

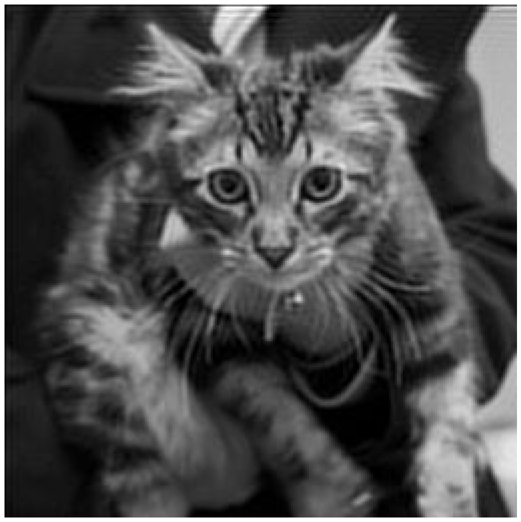
P = 1.0



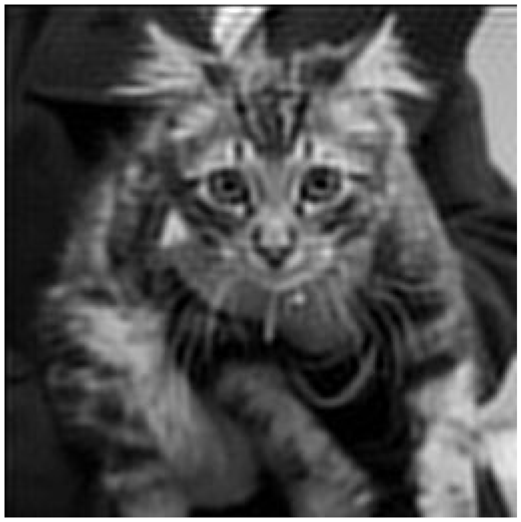
P = 0.8



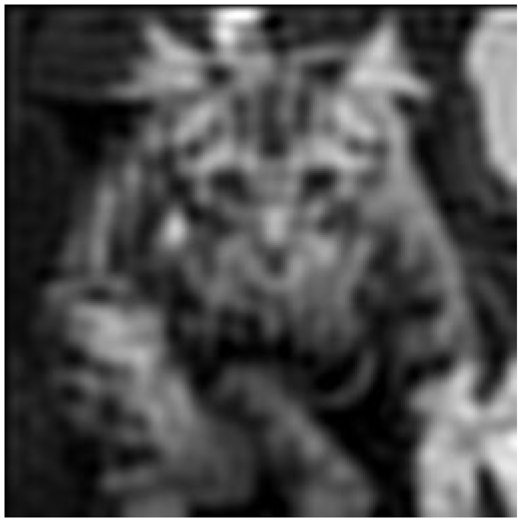
P = 0.5



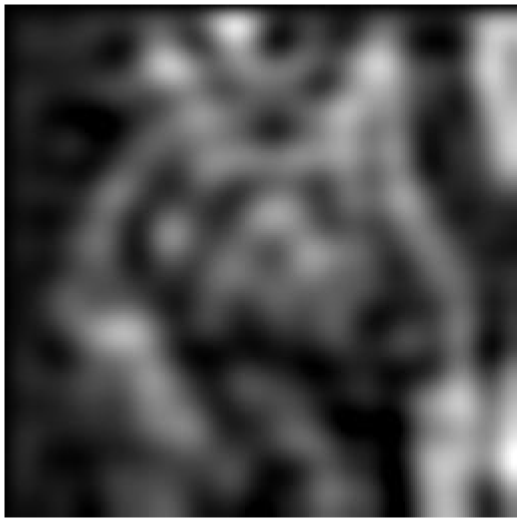
P = 0.3



P = 0.2



P = 0.1



P = 0.05

Reference Appendix Section #5.2.12 for code and figures.

For the next question in this section, we would say that a good *p* value would be around *p* = 0.3. Anything past that value yields an image that is significantly altered and basically ruined. We believe we are able to attain such a small *p* value because of the very way in which the DST operates. The majority of information in the DST is stored in the upper left side of the matrix—thus when we lower *p* and start removing information from it, we begin by removing high frequency sine components, prior to low frequency. That is, information that is located in other extrema of the matrix itself, namely, all locations other than the top left.

Because most information in the image is only stored as low frequency sine components, as we lower *p* there isn't a noticeable change to the image until *p* ~ 0.3, and then the changes happen quickly thereafter. Another factor could be our own limitations in vision. Computers today are now able to produce resolutions that are beyond the capabilities of human visual perception. Thus the results from compressing an image may not be detectable if such compression still holds a high enough resolution.

For the comparison, we will proceed to use *p* values of 0.5, 0.3, and 0.1. To understand how many non-zero entries are in the image, we will once more use logical indexing to save time. For a *p* value of 0.5 we have 32640 non-zero pixels. The total number of pixels is 65536 so we are losing about 32890 pixels of information in our DST. For *p* = 0.3 we got 11628 so we lost 53908 pixels of information. For *p* = 0.1 we got 1275, so we lost 64261 pixels of information. It is safe to say we can afford to lose around 80% of the information in an image without sacrificing much of the resolution and quality. The code used is once again referenced in the appendix section.

Reference Appendix Section #5.2.13 for code and figures.

For the final section of this project, we used the following calculation to determine the different compression ratios (CRs):

CR for p = 0.5 is 32640/65536 which is 0.49804

Cr for p = 0.3 is 0.177429

CR for p = 0.1 is 0.01945

Based on these results, we would certainly use *p* = 0.3 in order to give the best CR while still maintaining good image quality.

4. Code and Figure Appendix

Section 5.1

#5.1.1:

```
function matrix = greyscale(name)
x = imread(name)
xDouble = double(x)
matrix = 0.3*xDouble(:,:,1) + 0.3*xDouble(:,:,2) + 0.3*xDouble(:,:,3);
end
%This is the greyscale function
%It turns a * bit integer Array in adouble array
```

Where the Function call is:

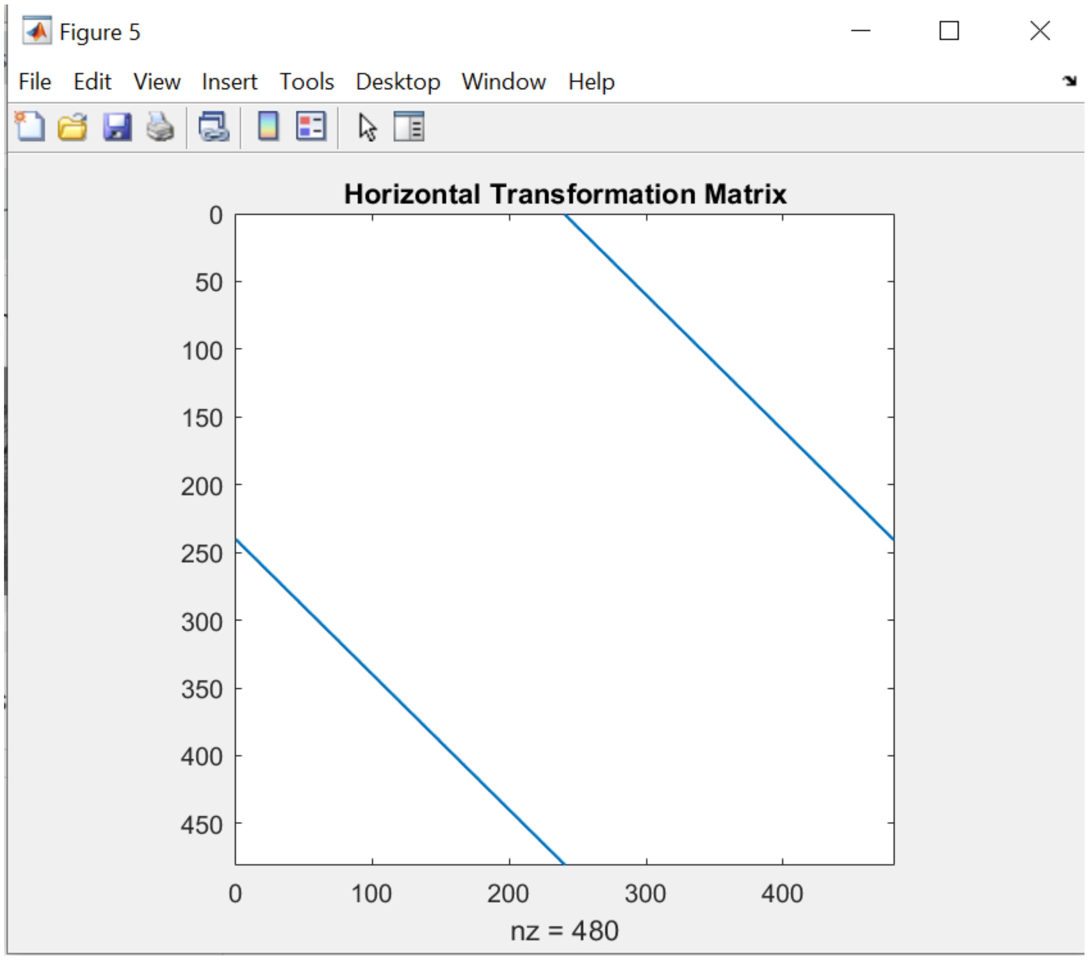
```
greyPhoto2 = greyscale('photo2.jpg')%convert matricies to greyscale
greyPhoto1 = greyscale('photo1.jpg')
figure
imagesc(uint8(greyPhoto1))%prints the image
colormap('gray')
title('Grey Scale Photo1')
imwrite(uint8(greyPhoto1), 'Grey Scale Photo1.jpg')%Writes greyscale image 1 to a file
figure
imagesc(uint8(greyPhoto2))
colormap('gray')
title('Grey Scale Photo2') %Writes greyscale image 2 to a file
imwrite(uint8(greyPhoto2), 'Grey Scale Photo2.jpg')
figure
```

#5.1.2:

```
greyPhoto2 = greyscale('photo2.jpg')%convert matricies to greyscale
greyPhoto1 = greyscale('photo1.jpg')
figure
imagesc(uint8(4*greyPhoto2)) %print over exposed greyscale photo 2
imagesc(uint8(4* double(imread('photo2.jpg'))))%print over exposed colored photo 2
title('Increased Exposure Photo2')
colormap('default')
imwrite(uint8(4* double(imread('photo2.jpg'))), 'Increased Exposure Photo2.jpg')%create image files
imwrite(uint8(4*greyPhoto2),'Increased Exposure Grey Photo2.jpg' )
```

```
greyPhoto2 = greyscale('photo2.jpg')%convert matricies to greyscale
greyPhoto1 = greyscale('photo1.jpg')
figure
[m,n] = size(greyPhoto1);%calculate how be we will need our transformation matrix to be
r = 240;%shifting 240 pixels
E = eye(n);
T = zeros(n);
T(n - r +1: n, :) = E(1 : r, :)%this produces a shift of 240 pixels to the left
T(1:n-r, :) = E(r+1 : n, :)
Xshift = greyPhoto1 * T
imagesc(uint8(Xshift));
colormap('gray');%prints the picture
imwrite(uint8(Xshift), 'Shifted photo1.jpg')
figure
spy(T)%prints the transformation matrix
title('Horizontal Transformation Matrix')
```

Graph of the transformation matrix:



#5.1.4/5:

Code used in this section: Notice that first we shift horizontally. This is achieved by left matrix multiplication. Then we are shifting upwards this is achieved via left matrix multiplications

```
greyPhoto2 = greyscale('photo2.jpg')%convert matricies to greyscale
greyPhoto1 = greyscale('photo1.jpg')
figure
[m,n] = size(greyPhoto1);%calculate how be we will need our transformation matrix to be
r = 240;%shifting 240 pixels
E = eye(n);
T = zeros(n);
T(n - r +1: n, :) = E(1 : r, :)%this produces a shift of 240 pixels to the left
T(1:n-r, :) = E(r+1 : n, :)
Xshift = greyPhoto1 * T %Mutiply the the transformation matrix to fist shift horizontally
figure
h = 100;%shifting vertically 100 pixels
W = eye(m);
M = zeros(m);
M(1:h,:) = W(m-(h-1):m,:);%creates a tranfomation matrix to shift a given matrix up 100 pixels
M(h+1:m,:) = W(1:m-h,:);
newXshift = M*Xshift%Notice that we are not shifting the orginal matrix but the one we previously transformed
imagesc(uint8(newXshift));%prints the image
colormap('gray');
imwrite(uint8(newXshift), 'Shifted Twice photo1.jpg')%writes the image to a file
figure
subplot(2,2,1)%displays 2 transformation maticies we used to shift the image
spy(T)
title('Matrix to Shift the image Horizontally')
xlabel('Matrix Applied to the Right of Photo')
subplot(2,2,2)
spy(M)
title('Matrix to Shift the image Vertically')
xlabel('Matrix Applied to the Left of Photo ')
```

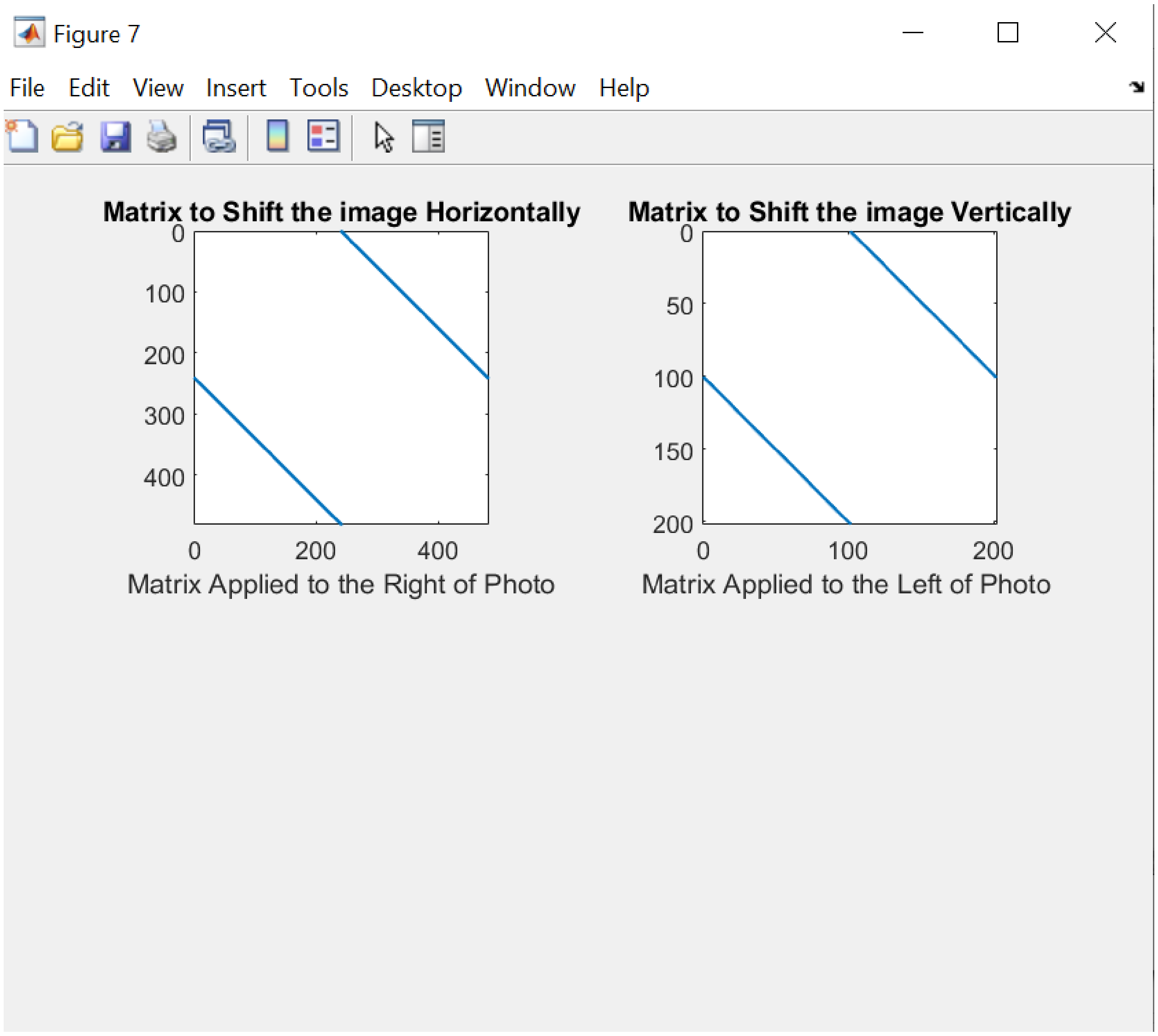


Figure 7
File Edit View Insert Tools Desktop Window Help
Matrix to Shift the image Horizontally
0
100
200
300
400
0
200
400
Matrix Applied to the Right of Photo
Matrix to Shift the image Vertically
0
50
100
150
200
0
100
200
Matrix Applied to the Left of Photo

#5.1.6:

We used the reverse identity matrix to flip the image:

```
greyPhoto2 = greyscale('photo2.jpg')%convert matricies to greyscale
greyPhoto1 = greyscale('photo1.jpg')
figure
[z, y] = size(greyPhoto2)%used to get demensions of needed matrix
revIden = flip(eye(z))%creates reverse identity matrix
flipImage = revIden*greyPhoto2 %use left matrix multiplication to flip the image
imagesc(uint8(flipImage));%print and write the image to a file
colormap('gray');
imwrite(uint8(flipImage), 'Flipped photo2.jpg')
```

#5.1.7:

Transposing the image matrix gives us the rotated photo below.

```
greyPhoto2 = greyscale('photo2.jpg')%convert matricies to greyscale
greyPhoto1 = greyscale('photo1.jpg')
figure
transposeImage = greyPhoto2.' %greyPhoto2 is alrady a double matrix
%thus we can transpose it using the built in matlab commands
imagesc(uint8(transposeImage));%prints and writes image to a file
colormap('gray');
imwrite(uint8(transposeImage), 'Tanspose photo2.jpg')
mountPhoto2 = double(imread('mount2.jpg'))
mountPhoto1 = double(imread('mount1.jpg'))
```

#5.1.8:

After editing and reconnecting, we get the grayscale photo below.

```
mountPhoto2 = double(imread('mount2.jpg'))%read mount1 and mount2 as files
mountPhoto1 = double(imread('mount1.jpg'))
[k, p] = size(mountPhoto2)%mount 1 is good all we need to do is combine it
%with our corrected mount2 image
U = eye(k);
V = zeros(k);
V(1:62, :) = U(k -61 : k, :)%this was done via trial and error to get the
%correct pixel, but essentialy the bottom is where the top should be
V(63 : 139, :) = U(k - 138: k - 62, :)%the middle section is where it
%should be
V(140 : k, :) = U(1 : 62, :)%the top section should be at the bottom
figure%V is just a more advanced form of what we where already doing
tempmount = V * mountPhoto2%uses left matrix mutiplication
mount = [mountPhoto1 tempmount]%combine mount 1 and mount 2 into 1 matrix
imagesc(uint8(mount));%print and write image to a file
colormap('gray');
imwrite(uint8(mount), 'Reassembled Photo 1.jpg')
```

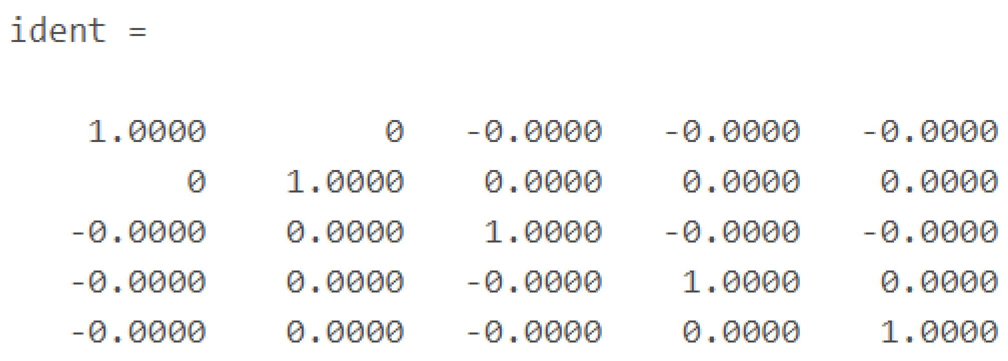
Section 5.2

#5.2.9:

```
function m = S(n)%function called s
m = zeros(n)
y = n
for i = 1 : n%loop goes from 1 to n
    j = 1 : n%unlike the instructions we are going to save alot of time by using
    %instead of nested loops
    m(i, j) = sqrt(2/n)*sin((pi*(i-1/2)*(j-1/2))/n)
end
end
```

#5.2.10:

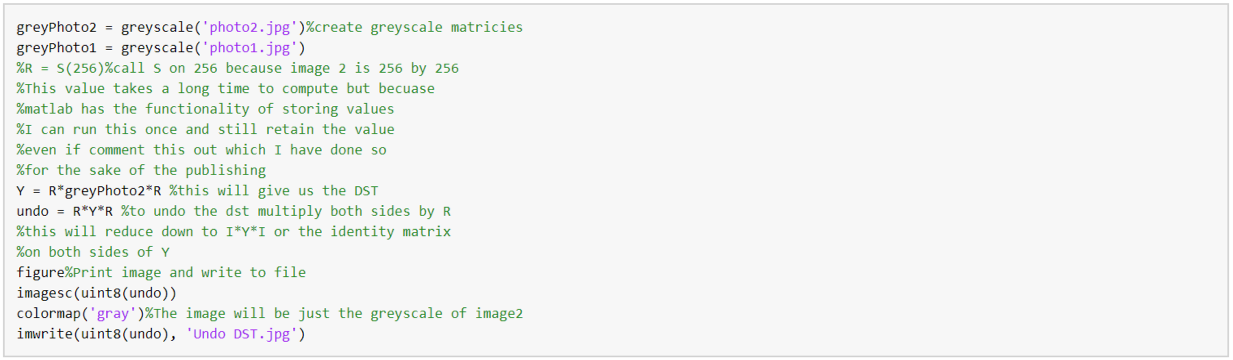
```
problem1 = S(5)%call the function s
ident = problem1*problem1%ident is technically not the identity matrix but
%all other entries exccpt the diagonal are approaching zero
```



```
ident =

    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
```

#5.2.11:

As one can see, the DST was applied and then unapplied via multiplying by S(256) on both sides.



```
greyPhoto2 = greyscale('photo2.jpg')%create greyscale matricies
greyPhoto1 = greyscale('photo1.jpg')
%R = S(256)%call S on 256 because image 2 is 256 by 256
%This value takes a long time to compute but becuase
%matlab has the functionality of storing values
%I can run this once and still retain the value
%even if comment this out which I have done so
%for the sake of the publishing
Y = R*greyPhoto2*R %this will give us the DST
undo = R*Y*R %to undo the dst multiply both sides by R
%this will reduce down to I*Y*I or the identity matrix
%on both sides of Y
figure%Print image and write to file
imagesc(uint8(undo))
colormap('gray')%The image will be just the greyscale of image2
imwrite(uint8(undo), 'Undo DST.jpg')
```

#5.2.12:

Below is the code template used for compression. In this example I used a p value of 0.5

```
greyPhoto2 = greyscale('photo2.jpg')%create greyscale matricies
greyPhoto1 = greyscale('photo1.jpg')
%R = S(256)%call S on 256 because image 2 is 256 by 256
%This value takes a long time to compute but becuase
%matlab has the functionality of storing values
%I can run this once and still retain the value
%even if comment this out which I have done so
%for the sake of the publishing
Y = R*greyPhoto2*R
undo = R*Y*R%gives original image in order to compare the compressed one too
figure%prints og image
imagesc(uint8(undo))
colormap('gray')
n = 256
p = 0.5
% for i = 1:n%note I have modified the fuction give to use logical indexing
%     j = 1 : n %the given nested for loops take approx 1 hr to complete
%     Y(i, j) = (i + j >p*2*n).*(0) +(i + j <= p*2*n).*(Y(i, j))%this uses logical indexing
%     %to make the calculations go by much faster
%     %now it takes around 1 minute
%     %what we are doing is modifying the if statement to occure in one line
%     %this does exactly what the given fuction does
%     %it removes information from the DST inorder to compress the image
% end
% Note I have commented this section out to format into the publisher
jk = R*Y*R %This is two convert the DST back into an image
figure%Prints image
imagesc(uint8(jk))
colormap('gray')
imwrite(uint8(jk), 'Compressed Image2 P = 0.5.jpg')
```

#5.2.13:

```
greyPhoto2 = greyscale('photo2.jpg')%create greyscale matricies
greyPhoto1 = greyscale('photo1.jpg')
%R = S(256)%call S on 256 because image 2 is 256 by 256
%This value takes a long time to compute but becuase
%matlab has the functionality of storing values
%I can run this once and still retain the value
%even if comment this out which I have done so
%for the sake of the publishing
Y = R*greyPhoto2*R
undo = R*Y*R%gives original image in order to compare the compressed one too
figure%prints og image
imagesc(uint8(undo))
colormap('gray')
n = 256
p = 0.5
% for i = 1:n%note I have modified the fuction give to use logical indexing
%     j = 1 : n %the given nested for loops take approx 1 hr to complete
%     Y(i, j) = (i + j >p*2*n).*(0) +(i + j <= p*2*n).*(Y(i, j))%this uses logical indexing
%     %to make the calculations go by much faster
%     %now it takes around 1 minute
%     %what we are doing is modifying the if statement to occure in one line
%     %this does exactly what the given fuction does
%     %it removes information from the DST inorder to compress the image
% end
%Have to comment this out to format in the publisher
counter = sum(Y(:) ~= 0)%counts number of non zero values in matrix
jk = R*Y*R %This is two convert the DST back into an image
figure%Prints image
imagesc(uint8(jk))
colormap('gray')
```

6 References

Farlow, Jerry, et al. *Differential Equations & Linear Algebra*. Pearson Prentice Hall, 2007.