APPM 2360 Project 2
Due Nov. 3 at 5:00 PM in D2L

1 Introduction

Digital images are stored as matrices of pixels. For color images, the matrix contains an ordered triple giving the RGB color values at each pixel; however, in this lab, we will work with grayscale images where only one value, the light intensity, is required for each pixel. Since digital images are 'seen' as matrices, we are able to alter images by simply applying matrix operations, also referred to as linear transformations. In this project, you will explore some ways to manipulate images using Matlab and apply your understanding of Linear Algebra along the way. We'll start off with transformation matrices and then move on to image compression.

2 Basic Matlab commands

Matlab has some nice built in functions for reading and writing image files—the first command we'll be using is `imread`, which reads in an image file saved in the current directory. `imread` works with most popular image file formats. To view an image in a Matlab figure, use `imagesc` (similar to `image` but, this will work more consistently for our purposes). The following code will read in an image with file name `photo1.jpg`, save it as the variable `X`, and display the image in a Matlab figure window. Make sure you save the image to the Matlab folder from where your code is run.

```
X = imread('photo1.jpg');
imshow(X,[0,255])
```

To simplify our matrix computations, we will be working with grayscale versions of photos for much of this lab. To convert the image to the necessary form for the remainder of the project, you can use the following code:

```
X_double = double(X_gray); %converts the image to a matrix of
   doubles
```

You can continue to view the image by typing `imshow(X_double,[0,255])`.

3 Image Manipulation

To gain some intuition, and a better understanding of how matrix multiplication really works, let's practice, in Matlab, by multiplying our image matrix by a permutation of the identity matrix and observing the effect. For example, the following matrix takes the entries of a vector and shifts them down one position, cycling the last entry around to the top.

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

For a larger version of the matrix like the one on the left above, which we'll call P, it can be helpful to visualize it with the Matlab command `spy()`. If P is defined in the Matlab workspace, typing `spy(P)` produces the figure shown below. `spy()` is especially useful for visualizing sparse[1] matrices that have large dimensions.

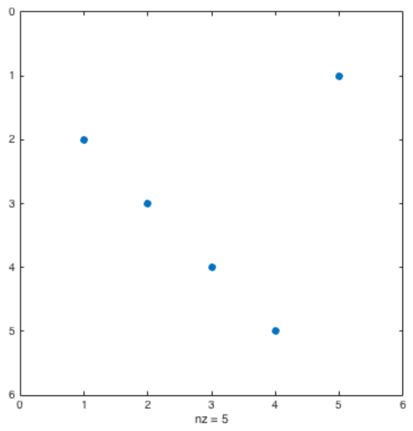


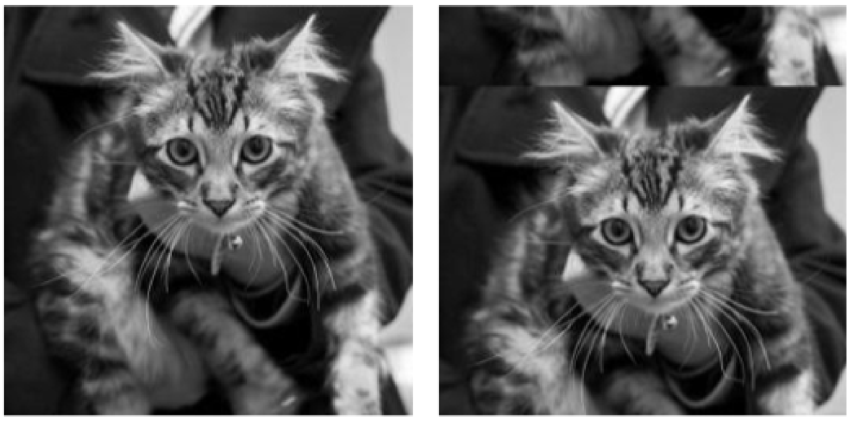
Figure 1: The figure shows the location of the nonzero entries of the matrix P. The axes give the row (y-axis) and column (x-axis) indices for the nonzero entries.

Notice that if you start with the identity matrix and interchange rows until you get the matrix on the left above, multiplying a vector by that new matrix applies the same row interchanges to the vector. Each of the rows were shifted down one, and the last row cycled around to the top. This is called a permutation matrix because it is obtained by permuting the rows and columns of the identity matrix. In this case, row 1 turns into row 6, row 2 turns into row 3 etc. This transformation matrix works on matrices, too.

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

Notice how the matrix multiplication cycled the rows around in the matrix the same way it did in the vector. Since an image is just a matrix, we can transform them using a linear transformation. The following image was transformed using a 256×256 version of the transformation matrix above, shifting the image down by 50 pixels.

[1]Matrices with relatively few nonzero entries.



Here's the code that produced the image above.

```
[m,n] = size(X_gray);
r = 50;
E = eye(m);
T = zeros(m);
% fill in the first r rows of T with the last r rows of E
T(1:r,:)  = E(m-(r-1):m,:);
% fill in the rest of T with the first part of E
T(r+1:m,:)  = E(1:m-r,:);
X_shift = T*X_gray
imagesc(uint8(X_shift));
colormap('gray');
```

Row vectors can be transformed too, just by multiplying by the transformation matrix on the right side. As an example:

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

However, the reordering is not the same as before—the transpose of the transformation matrix must be used to shift the elements so that the last element is first.

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

4 Image Compression

Typically when images are stored one doesn't want to have to store the whole image. Typically one finds a convenient representation of the image so that it can be stored more efficiently. We will investigate two methods for compressing an image.

4.1 The Discrete Cosine Transform

The discrete cosine transform (DCT) is a technique for decomposing a vector into a linear combination of sine functions with different frequencies. The idea is similar to that of a Taylor series, except instead of using polynomials to approximate a function, we are using cosine functions. If the data in a vector are smooth, then the low frequency components will dominate the linear combination. If the data are not smooth (discontinuous, jagged, rapidly increasing or decreasing), then the coefficients on the higher frequency components will have greater magnitude.

In practice, transforming a vector with the DCT means multiplying the vector by a special matrix called (unsurprisingly) the DCT matrix. There are several ways to define the DCT matrix; for this assignment, use:

$$C_{i,j} = \sqrt{\frac{2}{n}} \cos\left(\frac{\pi(i-\frac{1}{2})(j-\frac{1}{2})}{n}\right)$$

Where C is an $n \times n$ matrix, and $S_{i,j}$ is the entry of C in the i-th row and j-th column. There are several ways to construct this matrix, but the simplest way is to use nested `for` loops. If you get stuck, you may find the second half of Worksheet 5 on the APPM 2460 web page helpful.

To apply this transform matrix to a vector, just multiply. So if y is the transformed version of x, we would obtain it by computing $y = Cx$. Since C is square, the 1 dimensional DCT (*i.e.* the DCT that operates on vectors) is an operation that takes in a vector of length n and returns another vector of length n. For 1-D data (vectors), the output is a vector containing weights for the different frequency components; the higher the weight, the more important that frequency is. However, we cannot use C to transform our data because our image data is a matrix, which is two dimensional. So, we will need the 2-D transform. Thankfully, it's very easy to compute the 2-D DCT using the 1-D transform matrix. Let X_g be the grayscale version of the image data[2]. Then the 2-D DCT for the image X_g is:

$$Y = CX_gC^T$$

Intuitively, you can think of CX_g as applying applying the 1-D DCT to the columns of X_g, and X_gC^T as applying the 1-D DCT to the rows of X_g. So CX_gC^T applies the 1-D transform to both the rows and the columns of X_g. Our DCT matrix has the special property that it is *symmetric*, or equal to its transpose. So for our DCT matrix C,

$$C^T = C$$

Now we can define the 2-D transformed image as:

$$Y = CX_gC$$

If we want to get our original image back from the DCT, we'll need to know the inverse of C. Our matrix C also has the property that it is its own inverse. So we have,

$$C^{-1} = C$$

[2]X_g is a matrix whose (i, j) entry represents the grayscale level at pixel position (i, j). In our case, the values range from 0 to 255, with 0 being black and 255 being white.

This is useful, since inverses are often difficult to compute.

4.1.1 Compression with DCT

JPEG is a type of "lossy compression," which means that the compressed file contains less information than the original. Since human eyes are better at seeing lower frequency components, we can afford to toss out the highest frequency components of our transformed image. The more uniform an image, the more data we can throw away without causing a noticeable loss in quality. More complicated images can still be compressed, but heavy compression is more noticeable. Thankfully, the DCT helps us sort out which components of the image are represented by low frequencies, which are the ones we need to keep.

The information corresponding to the highest frequencies is stored in the lower right of the transformed matrix, while the information for the lowest frequencies is stored in the upper left. Therefore, we want to save data in the upper left, and not store data from the remaining entries. We will simulate this effect by zeroing out the high frequency components. The following code will zero out the matrix below the off diagonal:

```
p = 0.5;
%when p = 0, none of the matrix is saved, p=1 means no compression
for i = 1:n
        for j = 1:n
                if i + j > p*2*n
                        X(i,j) = 0
                end
        end
end
```

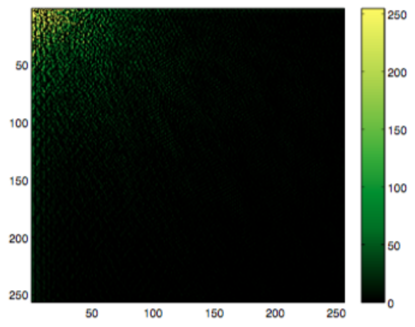


Figure 2: DCT coefficients for an image (values in the matrix Y). Values in the upper left are weights on low frequency sine components while values in the lower right are weights on high frequency sine components. Since the values in the upper left are significantly larger than those in the lower right, we can see that low frequencies dominate the overall image.

Adjusting the value of p moves the diagonal up and down the matrix, affecting how much data are retained. This illustration shows how the off diagonal moves with changes in p in Figure 3. After deleting the high frequency data, the inverse 2-D DCT must be applied to return the transformed image back to normal space (right now it will look nothing like the original photograph). Since none of the zeros need to be stored, this process could allow for a significant reduction in file

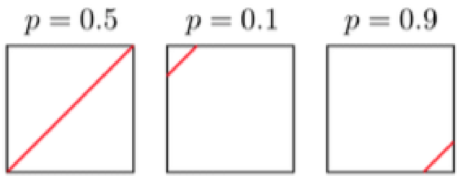


Figure 3: Adjusting the value of p

size. The compression we perform here is a simplified version of the compression involved when storing an image as a JPEG file, which uses a DCT on 8×8 blocks of the image data.

5 Questions

Write a report detailing some of the image manipulation techniques that you can implement using what you have learned in APPM 2360. In your report, you should include examples of images that you have processed with your code, and you should describe the techniques you used to achieve your results. **For this project, submit all your code in an appendix at the end of your report. Make sure to comment your code clearly, so it's very obvious which problem the code was for. Output not supported by code in the appendix will not be counted.**

5.1 Image Translation

1. 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 `photo1.jpg` so that the resulting image appears more "whited out" than the original. Include this increased-exposure image in your report and place the original alongside it so the difference is clear.

2. Given an image that is 4 pixels x 4 pixels, so that the grayscale representation of the image is a 4 x 4 matrix called A, what matrix E would you multiply A by in order to switch the left-most column of pixels with the right-most column of pixels? Should you multiply A by E on the right or the left (*i.e.* would AE, EA, or both give the desired result)?

3. Find a matrix that will perform a horizontal shift of 140 pixels to `photo1.jpg` and include the shifted image in your write up.
Correction: a previous version of this lab had asked for a horizontal shift of 240 pixels. While this will still be accepted, shifting by 140 pixels is much more interesting.
Hint: We saw with $n \times n$ matrices that to perform a horizontal shift we multiply our matrix by a transformation matrix on the right. The transformation matrix on the right was obtained by transforming the columns of the $n \times n$ identity in the same way we wanted the columns of the image matrix to be transformed. Display your matrix using `spy()`.

4. How could you perform a horizontal and vertical shift? That is, what matrix operations would need to be applied to to get an image to wrap around both horizontally and vertically? Apply transformations to the original matrix from `photo1.jpg` that result in both a horizontal and vertical shift. Shift the image 140 pixels horizontally and 100 pixels vertically. Display your

transformation matrix/matrices using `spy()`.
Again, this has been changed from 240 pixels.

5. Using what you learned about transformation matrices, determine what matrix would be required to flip an image upside down. Using that transformation, flip `photo1.jpg` upside down. Use `spy()` once more to display your transformation matrix.

5.2 Image Compression with DCT

1. Verify that, for the 5×5 matrix C, $C = C^{-1}$. That is, show that $C^2 = I_5$ (this can be done in Matlab).

2. Using your own DCT matrix code, compute the determinant of the DCT matrix for 3 value of n.[3] Do you notice anything interesting about the relationship? What does this tell us about the existence of C^{-1}?

3. Using the same DCT matrix code, compute the eigenvalues of C for $n = 512$ using the MATLAB command `eig`. Plot the eigenvalues on the complex plane (plot λ_i as point with the real part of λ_i as the x coordinate and the imaginary part as the y coordinate). Do you notice a pattern? Can you explain anything about the pattern in the determinants given what you now know about the eigenvalues? You'll need to use the following property:

$$\prod_{i=1}^{n} \lambda_i = \det(A)$$

   and you may assume the pattern will continue for larger values of n.

4. Determine what steps need to be taken to undo the 2-D DCT. Remember that our DCT is defined by $Y = CX_gC$, and also the special properties of C. You can easily check to see if your inverse transform works by applying it to Y and viewing it with `imshow(Y,[0,255])`.

5. Perform our simplified JPEG-type compression on the image of CU Boulder, `boulder.jpg`

   - Read the image into Matlab and store as a matrix of doubles
   - Perform the 2-D discrete cosine transform on the grayscale image data
   - Delete some of the less important values in the transformed matrix using the included algorithm
   - Perform the inverse discrete cosine transform
   - View the image or write to a file

   Compress the image with several different values of p. Include sample images for compression values that don't cause an obvious drop in quality, as well as some that do.

6. You should be able to make p pretty small before noticing a significant loss in quality. Explain why you think this might be the case. The point of image compression is to reduce storage requirements; compare the number of non-zero entries in the transformed image matrix (Y, not X_g) to a qualitative impression of compressed image quality. What value of p do you think provides a good balance? (no correct answer, just explain)

[3] you may use MATLAB's `det` command

7. The compression ratio is defined as the ratio between the number of nonzero elements in the original matrix and the number of nonzero elements in the compressed image. The Matlab function `nnz()` will likely be useful to determine the compression

$$CR = \frac{\#\text{ nonzero uncompressed}}{\#\text{ nonzero compressed}}.$$

Determine CR for $p = 0.5, 0.3, 0.1$, using the number of nonzero entries in your transformed image matrix Y as a substitute for file size. In your view, what p value (any value of p, not just $p = 0.5, 0.3, 0.1$) gives the best (largest) CR while still maintaining reasonable image quality?

8. Let's try to use the DCT for more than just image compression. Read the image `boulder_noisy.jpg` and compute the matrix Y of the image for a couple values of p. Does the DCT help much? Comment on how well, or not well, the DCT works on the noisy image.