Image Manipulation in MATLAB

1 Introduction

Digital images are just matrices of pixels, and any type of matrix operation can be applied to a matrix containing image data. In this project you will explore some ways to manipulate images using MATLAB. We'll start off with transformation matrices, and 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`. `imagesc` is similar to `image`, but for our purposes will work more consistently. 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 run the code from the same folder that contains the image.

```
X = imread('photo1.jpg');
imagesc(X)
```

After reading in an image like this, `X(:,:,1)` is a 2-D matrix with intensity values for the red channel, `X(:,:,2)` for the green and `X(:,:,3)` for the blue.

When images are read in using `imread`, MATLAB stores the data as integers. If we want to perform mathematical operations on the image data using floating point numbers, the integers must be converted to floats as well. If you just read in an image as `X`, you can use `X = double(X);` to perform the floating point conversion. Note that at this point, `imagesc` will no longer work properly on X, since the values in X aren't integers anymore.

If you want to write image data to an image file, you can use `imwrite`. Note that if you converted the image data to the double format, you'll need to convert back to integer values. The command to do this is `uint8`. The following code shows how to read in an image, convert to double, and write to a `.jpg` file. You'll want to build your image manipulation code around this template if you wish to write your output to an image file.

```
X_int = imread('photo1.jpg');
X_double = double(X_int);
%
% manipulate the image
%
imwrite(uint8(X_double),'outputFileName.jpg')
```

3 Image Manipulation

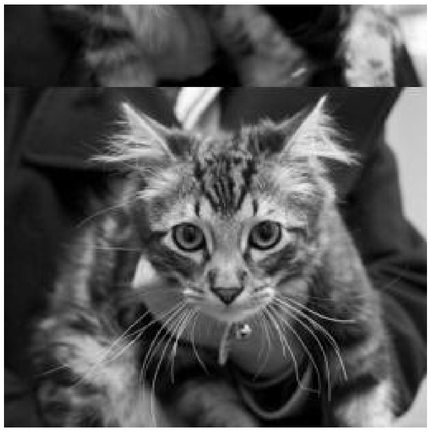
Matrix multiplication allows us to transform a vector in many ways. The following matrix takes the entries of a vector and shifts them down one position, cycling the last entry around to the top.

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

If you notice, the rows in the matrix were transformed in the same way that the product was, if you consider it starting off as the identity matrix. Each of the rows were shifted down one, and the last row cycled around to the top. This is called a Transformation Matrix, and it is obtained by performing the desired transformation on each column of the identity matrix. In this case, column 1 turns into column 2, column 2 turns into column 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 it cycled the rows around in a 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.



Here's the code that produced the shifted image above.

```
[m,n] = size(X_gray);
r = 50;
E = eye(n);
T = zeros(n,n);
%fill in the first r rows of T with the last r rows of E
T(1:r,:) = E(n-(r-1):n,:);
%fill in the rest of T with the first part of E
T(r+1:n,:) = E(1:n-r,:);
X_shift = T*X_gray;

imagesc(uint8(X_shift));
colormap('gray');
```

Rows 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.

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

Just for fun: this code preforms the shift above, but animates it through the entire image.

```
for r = 1:256;
    E = eye(n);
    T = zeros(n,n);
    T(1:r,:) = E(end-(r-1):end,:);
    T(r+1:end,:) = E(1:end-r,:);
    X_shift = T*X_gray;
    imagesc(uint8(X_shift));
    colormap('gray');
    drawnow;
end
```

4 Image Compression

4.1 The Discrete Sine Transform

You can think of the discrete Sine transform (DST) as decomposing a vector into a linear combination of cosine functions with different frequencies. 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 there will be more weight placed on the higher frequency components.

If something is smooth, that is it doesn't wiggle around very much, then most of the information can be retained by only looking at the first few sine modes. Think about Taylor series, smooth functions can be approximated locally pretty well by low order Taylor polynomials. The idea is very similar with a sine transform, except instead of using increasingly higher order (wigglier) polynomial terms we're using increasingly higher frequency (wigglier) sine functions to represent a given set of data.

There are several ways to define the DST matrix, for this assignment use:

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

Where S is an $n \times n$ matrix, and i and j represent the row and column index respectively. There are several ways to construct this matrix, the easiest way is to use nested `for` loops. If you get stuck, use the following skeleton code to get started.

```
S = zeros(n,n); %initialize S

for i = 1:n
    for j = 1:n
        S(i,j) = % fill in matrix entries
    end
end
```

To apply this transform matrix to a vector just multiply. So if y is the transformed version of x, we would obtain it by doing $y = Sx$. Since S is square, the 1-D DST is an operation that takes in a vector of length n and returns another vector of length n. For 1-D data, the output is a vector containing weights for the different frequency components; the higher the weight the more important that frequency component.

This is just the 1-D DST matrix. To transform our image data, we will need the 2-D transform. Thankfully it's very easy to compute the 2-D DST using the 1-D matrix. Let X_g be the grayscale version of the image data[1], then the 2-D DST for the image X_g is:

$$Y = SX_gS^T$$

Intuitively, you can think of SX_g as applying applying the 1-D DST to the columns of X_g, and X_gS^T as applying the 1-D DST to the rows of X_g. So SX_gS^T applies the 1-D transform to both the rows and the columns of X_g. Our DST matrix has the special property that it is equal to its transpose. So for our DST matrix S,

$$S^T = S$$

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

$$Y = SX_gS$$

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

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

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

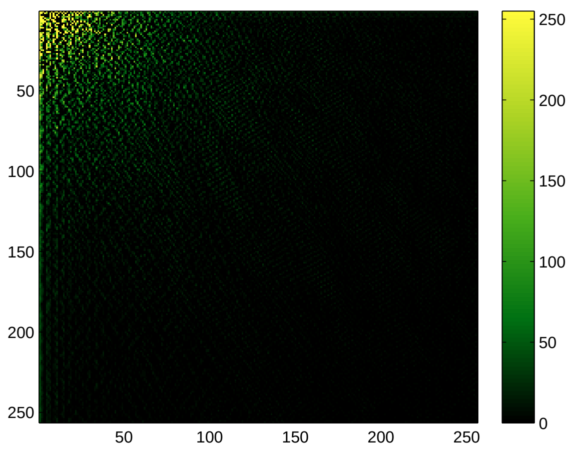


Figure 1: DST coefficients for an image (values in the matrix Y). Values in the upper left are weights on low frequency sine components while values in the lower right are weights on high frequency sine components. Since the values in the upper left are significantly larger than those in the lower right, we can see that low frequencies dominate the overall image.

[1]X_g is a matrix, who's (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.

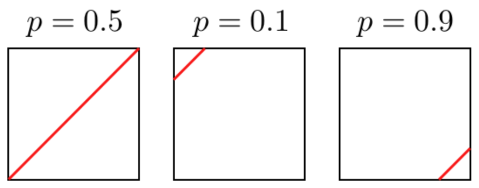
4.2 Compression

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 DST can 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 lowest is stored in the upper left. Therefore, we want to save data in the upper left, and delete data in the lower right. The following code will zero out the matrix below the off diagonal.

```
p = 0.5;
%when p=0, no data are saved
%when p=1, all data are saved
for i = 1:n
    for j = 1:n
        if i+j>p*2*n
            Y(i,j)=0;
        end
    end
end
```

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.



After deleting the high frequency data, the inverse 2-D DST 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 size.

5 Tasks

1. Write a MATLAB function to read in the files `photo1.jpg` and `photo2.jpg` and store them as matrices of doubles. Convert the color arrays into a grayscale matrices formed by the linear combination of 30% red, 59% green, and 11% blue. Include these grayscale images in your writeup.

2. The matrix from photo1.jpg is not square. We can, however, still apply matrices to shift the pixels. Find a matrix that will perform a horizontal shift of 240 pixels to photo1.jpg and include the shifted image in your write up.
   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 to columns of the image matrix to be transformed. For a non-square matrix X, we can take the same approach, but we have to start with the correct identity matrix. Think about the dimensions of the matrix you want to transform and find the matrix I_R such that $XI_R = X$. Manipulate the columns of I_R to obtain the transformation matrix.

3. 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. This matrix isn't square, so think about the dimensions for the appropriate transformation matrices. Shift the image 240 pixels horizontally and 100 pixels vertically.

4. Using what you learned about transformation matrices, determine what matrix would be required to flip an image upside down. Using that transformation, flip photo2.jpg upside down.

5. What should transposing an image matrix do? Try it with photo2.jpg. Does it look the way you expected?

6. Using your own DST matrix code, make a plot of the determinant[2] of S as a function of n for n from 1 to 32 (you will need to create 32 different DST matrices). Do you notice anything interesting about the relationship? What can you say about the rank of S?

7. Given the dimensions of our DST matrix, what restrictions must we impose on the aspect ratio of images we wish to transform?

8. Determine what steps need to be taken to undo the 2-D DST. Remember that our DST is defined by $Y = SX_gS$, and also the special properties of S. You can easily check to see if your inverse transform works by applying it to Y and viewing it with imagesc.

9. Perform our simplified JPEG-type compression on the image of Albus the cat, photo2.jpg

   - Read an image into MATLAB and store as a matrix of doubles
   - Convert the 3-D RGB matrix to a 2-D grayscale matrix
   - Perform the 2-D discrete sine 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 sine 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. (Something like the 4 images at the top of this project writeup)

[2]you may use MATLAB's det command

10. 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)

Note Include all of your code in an appendix. 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.

6 Using Mathematica

You can do all of the tasks in this project using Mathematica as well. You can use the functions `Import` and `Export` to read and write image files. `Table` will allow you to construct the DST matrix. The documentation for `FourierDCT` has an example of reading in an image and storing as an array. You may not use `FourierDST`. You must construct your own DST matrix using the definition listed previously. Do not copy code from the Mathematica documentation or other sources online, as that would be plagiarism.