Project 2 - Image Manipulation in MATLAB

Due in Gradescope 29 November 2022 at 11:59 PM MST

1 Introduction

A couple of your friends in the business school have a concept for a great new picture editing application for mobile devices, but they need help on the technical side of things. They've asked you to come on board and help write some of the code necessary to manipulate the images. It just so happens that you have been studying matrix manipulations in your differential equations/linear algebra class and this knowledge, along with some basic programming skills, will allow you to make a significant contribution to the application.

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 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 will 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 `inputFileName.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('inputFileName.jpg');
imagesc(X)
```

After reading in an image like this, `X(:,:,1)` is a two dimensional matrix with intensity values for the red channel, `X(:,:,2)` for the green and `X(:,:,3)` for the blue. You can think of `X` as a three-dimensional cube consisting of three two-dimensional matrices lying of top of each other.

When images are read in using `imread`, MATLAB stores the data as 8-bit integers, or integers that can range from 0 to 255. 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 = double(X);` to perform the floating point conversion.

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 will 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 will want to build your image manipulation code around this template if you wish to write your output to an image file.

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

Note that if you want to view the image after you have converted the image data to double format, you may need to convert back to `uint8` just as you did to use `imwrite`. In other words, the command you will need to view an image once you've converted it to double format is `imagesc(uint8(X_double))`.

For some of this lab, we will be working with grayscale versions of photos to keep the matrix manipulations simpler. To make a grayscale image out of our color image, we want to combine the matrices that store the information for the red, green, and blue colors into one matrix whose (i, j)-th entry tells the *intensity* level of the pixel at that position in the image, with 0 being black and 255 being white. Since we want the information from each of the matrices storing the red, green, and blue values for the image to contribute to the intensity of each pixel, we will create the matrix for the grayscale image by making a linear combination of each of `X_double(:,:,1)`, `X_double(:,:,2)`, and `X_double(:,:,3)`. The following lines of code will convert the image to grayscale and display this converted image in a MATLAB figure:

```
X_gray= X_double(:,:,1)/3.0 + X_double(:,:,2)/3.0 + X_double(:,:,3)/3.0;
imagesc(uint8(X_gray))
colormap('gray')
```

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.

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

Matrices like the one on the left above, which we will call $\mathbf{P}$, can be helpfully visualized with the MATLAB command `spy()`. If $\mathbf{P}$ is defined in the MATLAB workspace, then typing `spy(P)` gives the figure shown below. `spy()` is especially useful for visualizing sparse[1] matrices that have large dimensions.

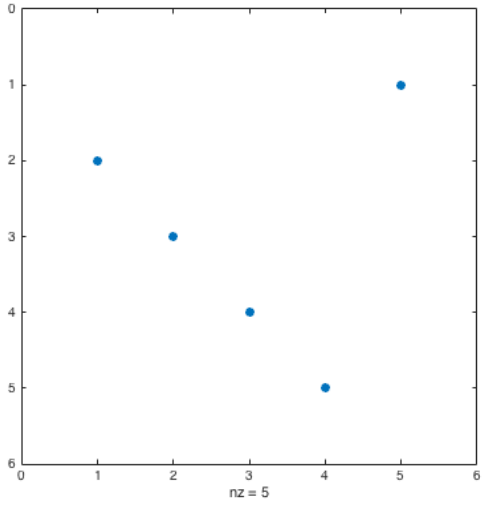
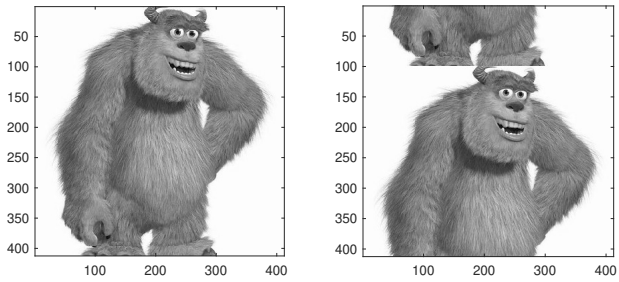


Figure 1: The figure shows the location of the nonzero entries of the matrix $\mathbf{P}$. The axes give the row (y-axis) and column (x-axis) indices for the nonzero entries.

Notice that if you start with the identity matrix ($\mathbf{I}$) 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 was 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 5, row 2 turns into row 1, *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 matrix multiplication. The following image was transformed using a 412×412 version of the transformation matrix above, shifting the image down by 100 pixels.



[1] Matrices with relatively few nonzero entries.

Here's the code that produced the image above.

```
[m,n] = size(X_gray);
r = 100;
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

4.1 The Discrete Sine Transform

The Discrete Sine Transform (DST) 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 sine 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 DST means multiplying the vector by a special matrix called (unsurprisingly) the DST matrix. 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 $\mathbf{S}$ is an $n \times n$ matrix, and $s_{i,j}$ is the entry of $\mathbf{S}$ 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.

To apply this transform matrix to a vector, just multiply. So if $\vec{\mathbf{y}}$ is the transformed version of $\vec{\mathbf{x}}$, we would obtain it by computing $\vec{\mathbf{y}} = \mathbf{S}\vec{\mathbf{x}}$. Since $\mathbf{S}$ is square, the one dimensional DST (*i.e.* the DST that operates on vectors) is an operation that takes in a vector of length n and returns another vector of length n. For one dimensional 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 $\mathbf{S}$ to transform our data because our image data is a matrix, which is two dimensional. So, we will need the two dimensional transform. Thankfully, it's very easy to compute the two dimensional DST using the one dimensional transform matrix. Let $\mathbf{X}_g$ be the grayscale version of the image data[2]. Then the two dimensional DST for the image $\mathbf{X}_g$ is:

$$\mathbf{Y} = \mathbf{S}\mathbf{X}_g\mathbf{S}^{\mathrm{T}}$$

[2] $\mathbf{X}_g$ 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.

Intuitively, you can think of $\mathbf{S}\mathbf{X}_g$ as applying the one dimensional DST to the columns of $\mathbf{X}_g$, and $\mathbf{X}_g\mathbf{S}^{\mathrm{T}}$ as applying the one dimensional DST to the rows of $\mathbf{X}_g$. So $\mathbf{S}\mathbf{X}_g\mathbf{S}^{\mathrm{T}}$ applies the one dimensional transform to both the rows and the columns of $\mathbf{X}_g$. Our DST matrix has the special property that it is *symmetric*, or equal to its transpose. So for our DST matrix $\mathbf{S}$,

$$\mathbf{S}^{\mathrm{T}} = \mathbf{S}$$

Now we can define the two dimensional transformed image as:

$$\mathbf{Y} = \mathbf{S}\mathbf{X}_g\mathbf{S}$$

If we want to get our original image back from the DST, we'll need to know the inverse of $\mathbf{S}$. Our matrix $\mathbf{S}$ also has the property that it is its own inverse ($\mathbf{S}$ is *involutory*). So we have,

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

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

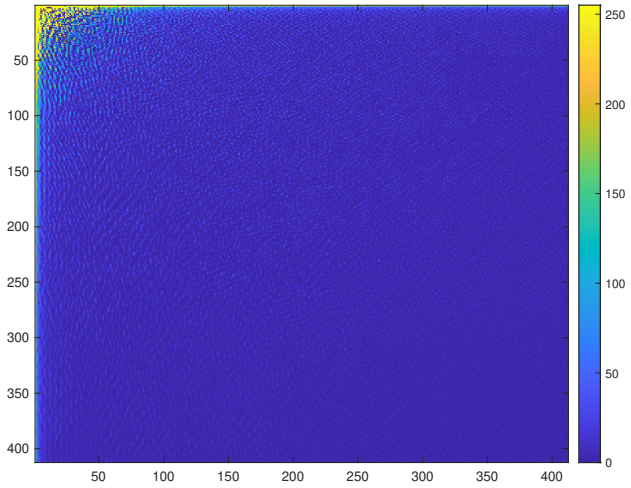


Figure 2: DST coefficients for an image (values in the matrix $\mathbf{Y}$). Values in the upper left are weights on low frequency sine components while values in the lower right are weights on high frequency sine components. Since the values in the upper left are significantly larger than those in the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

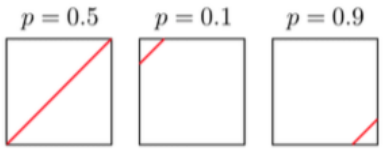
4.2 Compression

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 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 *antidiagonal*, the diagonal running from the lower left to the upper right of the matrix:

```
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 antidiagonal up and down the matrix, affecting how much data are retained. This illustration shows how the antidiagonal moves with changes in p.



After deleting the high frequency data, the inverse two dimensional 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. The compression we perform here is a simplified version of the compression involved when storing an image as a JPEG file, which uses a Discrete *Cosine* Transform on 8×8 blocks of the image data.

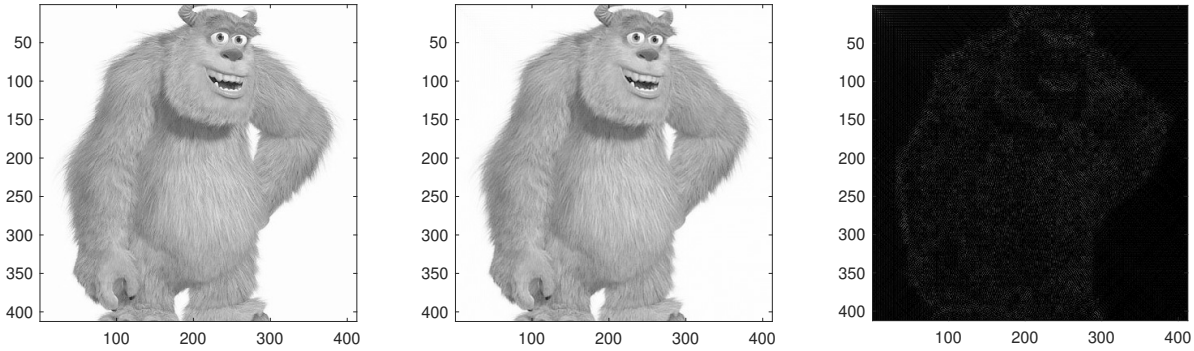


Figure 3: The left figure is the original image. The middle figure was obtained by performing the DST on the image ($\mathbf{Y} = \mathbf{S}\mathbf{X}_\mathbf{g}\mathbf{S}$), zeroing out the bottom right "half" of the matrix $\mathbf{Y}$ (using the above algorithm with $p = 0.5$), and then applying the inverse DST. The right figure was obtained by zeroing out the top left "half" of $\mathbf{Y}$ and applying the DST. This shows that the values in the bottom right "half" of the matrix store the image data that corresponds to quickly oscillating pixel intensities, or fine textures in the image. The values in the top left of $\mathbf{Y}$ correspond to image data with coarser, smoother texture. We also clearly see that the values in the top left contain the vast majority of the information in the image.

5 Questions

Your friends have asked you to write code capable of shifting, cropping, and compressing images (image compression can also double as a blurring effect because removing high frequency information smooths sharp edges). To do so, follow the steps laid out in the questions below. 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. 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`, which can be found in the Project Module in Canvas. These tasks can be completed by piecing together the blocks of code given in Section 2. Note that in what follows, when dealing with color images, you need to perform matrix operations on all three (red, green, blue) of the color intensity $m \times n$ matrices separately. To display the new images after the operations have been performed, place the three $m \times n$ matrices back into an $m \times n \times 3$ matrix to pass into `imagesc`.
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. Include this increased-exposure image in your report and place the original alongside it so your friends can clearly see the difference between the two. (Consider having a look at the MATLAB command `subplot`).

CONTINUED - THERE ARE MORE QUESTIONS BELOW

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 (simply add 80 to the matrix containing the blue intensities). Include this changed color image in your report.

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 $\mathbf{A}$ 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.

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` 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 matrix in the same way we wanted the columns of the image matrix to be transformed. For a non-square matrix $\mathbf{X}$, 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 $\mathbf{I}_R$ such that $\mathbf{X}\mathbf{I}_R = \mathbf{X}$. Manipulate the columns of $\mathbf{I}_R$ to obtain the transformation matrix. Display your matrix using `spy()`.

6. How could you perform a horizontal and vertical shift? That is, what matrix operations would need to be applied to get an image to wrap around both horizontally and vertically? Apply transformations to the original matrix from `rectangular.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 306 pixels horizontally and 230 pixels vertically. Display your transformation matrix/matrices using `spy()`.

7. Using what you learned about transformation matrices, determine what matrix would be required to flip an image upside down. Using that transformation, flip `rectangle.jpg` upside down. Use `spy()` once more to display your transformation matrix. (Note: do not use any built-in MATLAB functions to do this)

8. What should transposing an image matrix do? Try it with `rectangle.jpg`. Does it look the way you expected?

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. There are different ways to do this, but here you will want to start with an appropriately sized identity matrix or matrices, change it/them slightly, and then apply appropriate matrix operations to the color intensity matrices to generate the cropped image. Use `spy()` to include these transformation matrices in your report. It may be helpful to experiment with this on paper prior to writing any code.

5.2 Image Compression

10. Write a function that returns the matrix $\mathbf{S}$ of any given size. Write this yourself. Do not simply use a built-in MATLAB function.

11. Verify that, for the 5×5 matrix $\mathbf{S}$, $\mathbf{S} = \mathbf{S}^{-1}$. That is, show that $\mathbf{SS} = \mathbf{I}_5$ (this can be done in MATLAB).

12. Determine what steps need to be taken to undo the two dimensional DST. Remember that our DST is defined by $\mathbf{Y} = \mathbf{S}\mathbf{X}_g\mathbf{S}$, and also the special properties of $\mathbf{S}$. You can easily check to see if your inverse transform works by applying it to $\mathbf{Y}$ and viewing it with `imagesc`.

13. Perform our simplified JPEG-type compression on the image of the rooster in `square.jpg`, which can be found in the Project Module in Canvas.

    - Read the image into MATLAB and store as a matrix of doubles (you should be able to use the function that you wrote for question 1)
    - On each of the color intensity matrices
      - Perform the two dimensional discrete sine transform
      - Delete some of the less important values in the transformed matrices using the included algorithm
      - Perform the inverse discrete sine transform
    - View the image or write to a file (remember to put the manipulated red, green and blue $m \times n$ matrices into a single $m \times n \times 3$ matrix prior to displaying or writing the transformed image)

    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.

14. 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 nonzero entries in the transformed image matrix ($\mathbf{Y}$, not $\mathbf{X}_g$) to a qualitative impression of compressed image quality. What value of p do you think provides a good balance? (no correct answer, just explain)

CONTINUED - THERE ARE MORE QUESTIONS BELOW

15. The compression ratio is defined as the ratio between the uncompressed file size and compressed file size

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

Determine CR for $p = 0.5, 0.3, 0.1$, using the number of nonzero entries in your transformed image matrix $\mathbf{Y}$ 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?