APPM 2360 Lab #2: Image Manipulation

Alex Mault, Evan Mori, John Dunn

November 1, 2013

0 Introduction

Pictures are made up of pixels which are, in essence, matrices where each data point is a pixel. These matrices can be manipulated using normal matrix operations. In this lab, we use MATLAB to perform different operations on the image matrices to manipulate them into new forms.

1 Color Manipulation to Achieve a Greyscale Photo

Color Manipulation is used everywhere in the photography industry. Whether it be a professional photography in photoshop or a casual shutterbug on instagram, color manipulation and image filters are quickly becoming more mainstream. Using MATLAB, we can achieve some of the same effects as high quality programs like photoshop. A simple example is conversion of an image to greyscale. To convert to greyscale, we first have to read in the image file and separate each pixel into it's red, green and blue contributions using imread('photo1.jpg'). Next, we create our greyscale matrix. This matrix contains the percent of the colors we want to achieve a greyscale.(R:30% G:59% B:11%). By multiplying the linear combination of the greyscale and image matrices we can create a greyscale image shown in Figure 1.

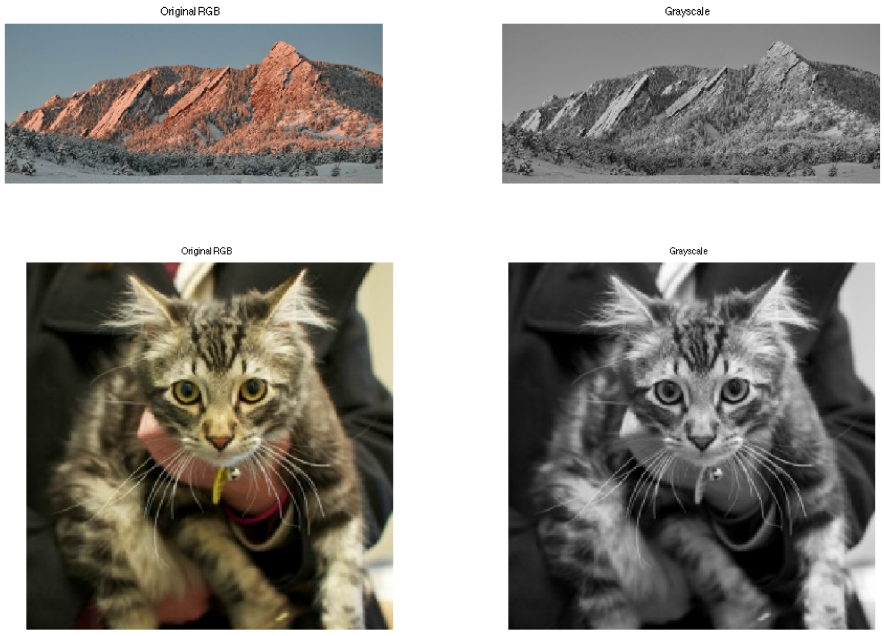


Figure 1: Image Greyscale

2 Transformation Matrices to Perform Horizontal Shifts

To perform image shift transformations,(i.e. horizontal and vertical shifts) we can utilize matrix multiplication of a transformation matrix and an image pixel matrix. The transformation matrix is a manipulated identity matrix that reflects the transformation of the desired image. In order to create a horizontal shift we simply multiply the image matrix by a column shifted identity matrix, shown in Figure 2.

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

Figure 2: Image Matrix $*$ Transformation Matrix = Shifted Image Matrix

We can utilize this method to manipulate image pixel matrices and achieve horizontal shifts, shown in Figure 3.

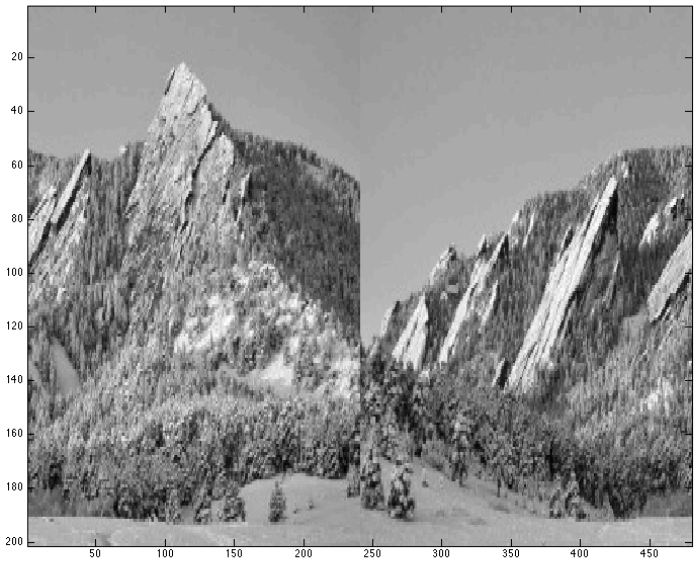


Figure 3: By using matrix multiplication with a shifted identity matrix, we can create a horizontally shifted image.

3 Transformation Matrices to Perform Both Horizontal and Vertical Shifts

Furthermore, we can utilize multiple transformation matrices to reflect multiple image shifts. To preform a vertical shift, we have to multiply a row shifted transformation matrix by an image matrix, in that order. For a horizontal shift, we multiply an image matrix by a column shifted transformation matrix, again, in that order. Logically, to preform a vertical and a horizontal shift on an image, we multiply a row shifted transformation matrix by an image matrix and then by a column shifted transformation matrix.

$$\begin{vmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{vmatrix} \times \begin{vmatrix} 2 & 3 & 4 & 4 & 4 \\ 4 & 2 & 2 & 1 & 3 \\ 6 & 7 & 3 & 1 & 0 \\ 4 & 7 & 0 & 3 & 1 \\ 1 & 2 & 5 & 7 & 2 \end{vmatrix} \times \begin{vmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{vmatrix} = \begin{vmatrix} Transformed \\ Image \end{vmatrix}$$

Figure 4: Vertical Shift $*$ Image $*$ Horizontal Shift = Transformed Image

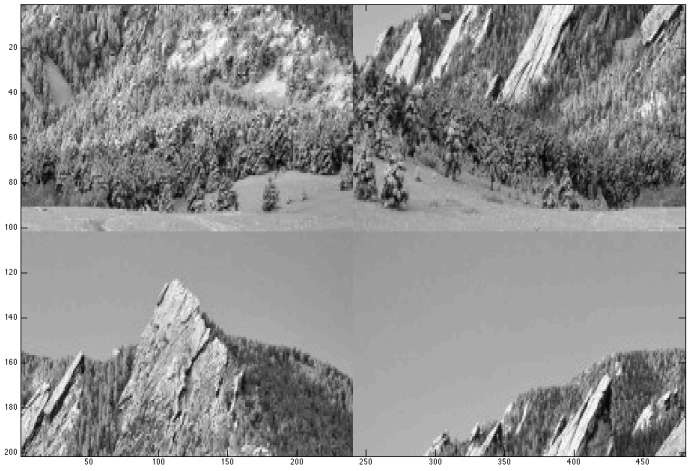


Figure 5: Using the same technique we manipulated the image to create a transformed image shifted 240px horzontally and 100px vertically.

4 Transformation Matrices to Flip an Image

Another way to manipulate an image is to flip it. To accomplish this, we will again use the power of transformation matrices. If we flip the columns of an identity matrix,creating a transformation matrix, the resulting product between the transformation and image matrices will be a vertically flipped image. This process is exemplified in Figure 6.

$$\begin{vmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{vmatrix} \times \begin{vmatrix} 2 & 3 & 4 & 4 & 4 \\ 4 & 2 & 2 & 1 & 3 \\ 6 & 7 & 3 & 1 & 0 \\ 4 & 7 & 0 & 3 & 1 \\ 1 & 2 & 5 & 7 & 2 \end{vmatrix} = \begin{vmatrix} Flipped \\ Image \end{vmatrix}$$

Figure 6: Row flipped Transform Matrix $*$ Image $* =$ Vertical Flipped Image

Figure 7: Using the same technique we manipulated the image to create a flipped image.

The same technique could be used to do a horizontal flip. The only difference would be that the transformation matrix would be the after the image matrix. See Figure 8 for example.

$$\begin{vmatrix} 2 & 3 & 4 & 4 & 4 \\ 4 & 2 & 2 & 1 & 3 \\ 6 & 7 & 3 & 1 & 0 \\ 4 & 7 & 0 & 3 & 1 \\ 1 & 2 & 5 & 7 & 2 \end{vmatrix} \times \begin{vmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} Flipped \\ Image \end{vmatrix}$$

Figure 8: Image $*$ Row flipped Transform Matrix $* =$ Horizontal Flipped Image

5 Image rotation by Transposition

The last image manipulation is rotation, which is the easiest of all manipulations because it does not require additional transformation matricies. We need only to transpose the image in MATLAB by saying rotatedImage = imageMatrix'. The resulting image follows the idea of image rotation. See Figure 9.

Figure 9: A transposed image matrix results in a rotated image.

6 Image Correction Using Transposition and Transformation Matrices

We were tasked with correcting an image using the manipulations detailed above (Horizontal/Vertical Shifts, Horizontal/Vertical Flips, and Rotation). Given the image shown in Figure 10, we observed that a horizontal shift of 249 pixels and a vertical shift of 299 pixels, followed by a vertical flip, would give a corrected image also shown in Figure 10.

Figure 10: Image Correction using pixel manipulation.

7 2-D Discrete Cosine Transforms

Figure 11: Image Correction using pixel manipulation.

The Discrete Cosine Transform is a image manipulation technique that is used to take the lower frequency values in an image matrix and push them to the top left corner, and move the higher frequency values along the main diagonal. This is demonstrated on the righthand side of Figure 11. Using equation 1 below, we can create a matrix that, when used in equation 2, changes the matrix into the form described above.

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

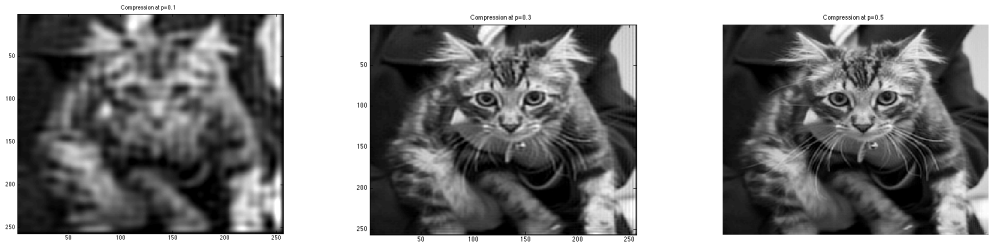
$$Y = CX_gC^T \tag{2}$$

In order to reverse the Discrete Cosine Transform that resulted from equation, we must ensure that the aspect ratio of the image is 1:1, that is, the image is perfectly square. If the condition is satisfied, then we can use equation 3 in order to reverse the DCT and extract the original image see on the lefthand side of Figure 11.

$$Image = C^T * Y * C^T; \tag{3}$$

8 Simplified JPEG-type compression

Using the Discrete Cosine Transform above we can arrange image data in such a way that it is trivial to remove the higher frequency data that is not visible or apparent to the human eye. This way it takes less room to store, and depending on the amount of data deleted, does not noticeably change the visible quality of the compressed image. Figure 11 shows the same image with different amount of data removed. The lower the value "p" is, the more data has been removed.



(a) Compression P=0.1 (b) Compression P=0.3 (c) Compression P=0.5

Figure 12: Images with varying compression values.

9 Analysis of Image Compression Techniques

Much of the data in a digital image is high frequency noise, unseen by the human eye. Things such as extremely small differences in color or brightness between adjacent pixels can not be seen, but yet take up space in the image. To reduce the image size while minimally impacting the visual quality of the image, we need to remove the unnoticeable data. To determine how much data we remove, we define P as a rough approximate to the "percent of original data remaining." Because a majority of the picture is made up of this high-frequency noise, due to large patches of similar colors such as the sky and ground, we should be able to reduce the P value greatly before there is a noticeable change.

To quantitatively analyze how much data we lose, we will count the nonzero elements of the image's DCT each time it is compressed. Starting with the original image having 65536 nonzero elements, we begin decreasing the 'p' value. For example, at P=0.5 the nonzero counts to roughly half of the original weighing in at 32640 nonzeros. Contrary to natural intuition, even though half of the data has been lost, viewing the image shown on the right of Figure 12 yields what appears to be an image incredibly similar to the original (pictured at the bottom of Figure 1). P=0.5 appears to be a good compression ratio for those willing to trade slightly less sharp image for a 50% saving in disk space.

10 Conclusion

Image manipulation is a huge part of photography, and is becoming widely accessible in the form of programs like Photoshop. In this lab we looked behind the scenes of Photoshop and found that basic image edits can be done using matrices. First, we learned how to change the colors in a photo by separating an image into its RGB values, then using linear combinations to adjust the values of each color. Next, we found that we could shift and flip an image into any form by using transformation matrices, which are simple row/column modified identity matrices. We also found that image rotation could be achieved by performing a transpose directly on an image matrix. Furthering the transformations of images, we then applied these modifications sequentially to a single broken image to fix it. Stepping away from direct transformations of images, we learned the Discrete Cosine Transform as a way to quickly organize an image by the frequency of the change in its pixels. Using the resulting sorted image, we were able to then discard unseen data from the image, resulting in a smaller image without noticeable loss in visual quality.

11 Appendix

11.1 Code

Code is organized with the primary FullProject function that then calls to the functions printed below it. All files have been concatenated into this appendix for convenience of the reader. A zip file of all .m files may be requested from Alex.Mault@colorado.edu

Please See Next Page for beginning of code.