Image Manipulation Using Matrix Techniques[1]

Names	Student ID	Professor	TA	Recitation
Will Farmer	101446930	Mimi Dai	Amrik Sen	608
Jeffrey Milhorn	100556107	Kevin Manley	Ed Yasutake	605
Patrick Harrington	100411000	Mimi Dai	Ash Same	618

Friday, March 22

[1] Report LaTeXSource Code is attached.

Contents

Table of Contents 1
1 Reading Image Files & Grayscale Conversion 2
2 Horizontal Shifting 4
3 Vertical Shifting 4
4 Inversion 6
5 Transposition 6
6 DST 6
7 Compression 6
8 Optimization 6
9 Code 6
9.1 Python 7
9.2 MATLAB Code 13
9.3 MATLAB Code 14

List of Figures

List of Figures 1
1 Provided Images 2
2 A simple RGB image 2
3 A given image split into its three primary color channels 3
4 A given image split into its three primary color channels, but only intensity of each color is shown. . . 3
5 Grayscale Images 4
6 Horizontally Shifted Images 5
7 Vertically Shifted Images 5
8 An example of a transposed image 6
9 The plot of the Discrete Sine Transform 7

List of Equations

Introduction

Since images stored on computers are simply matrices where each element represents a pixel, matrix methods learned in class can be used to modify images. The purpose of this project was to apply matrix manipulations on given image files, shown below as Figure 1a and Figure 1b.

(a) Photo 1

(b) Photo 2

Figure 1: Provided Images

1 Reading Image Files & Grayscale Conversion

Colored images have an interesting, although problematic property; they do not readily lend themselves to matrix manipulation because in order to get color images, seperate values are used to represent each primary color, which are then mixed together for the final color. For example, in Figure 2, the block represents very simple a 2×2 pixel image.

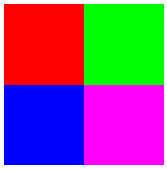

Figure 2: A simple RGB image

This very simple image can be represented as either a trio of primary color matrices where each entry in each primary color matrix coresponds to the same pixel:

$$\underbrace{\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}}_{\text{Red Matrix}}, \underbrace{\begin{bmatrix}0 & 0\\ 1 & 1\end{bmatrix}}_{\text{Blue Matrix}}, \underbrace{\begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}}_{\text{Green Matrix}}$$

A single matrix may be used, with each entry being a submatrix, wherein each element in the submatrix corresponds to a primary color.

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

Using one of the given images, the splitting of color channels gives the following set of images shown in Figure 3.

Figure 3: A given image split into its three primary color channels

While it is possible to manipulate color images, it would be far simpler to manipulate *grayscale* images, where only the final intensity is concerned. To do this, each color is considered independently for its intensity alone as shown in Figure 4, where it may be scaled, and then added together to produce a final black-and-white image, which is a matrix where each entry is a single value. Note how the third panel representing the blue color channel is darker – this implies that blue is a less intense color in the image.

Figure 4: A given image split into its three primary color channels, but only intensity of each color is shown.

Since each primary color is freely editable, it is simple to scale the intensity of each before mixing; in our report, we used 30% of the red channel, 59% of the green channel and 11% of the blue channel. The final outputs for both given images can be seen in Figure 5. Note how the final output is lighter than any of the individual color channels.

(a) Photo 1 - Grayscale

(b) Photo 2 - Grayscale

Figure 5: Grayscale Images

2 Horizontal Shifting

Now that we are working in grayscale, it is far more straightforward to manipulate aspects of the image, such as its horizontal position. Since we are dealing with a normal matrix, transforming the positions of columns requires only that we multiply the image matrix by a transformation identity matrix.

As discussed in the lab instructions, to shift an image horizontally without losing information requires the use of a transformation matrix as shown below.

$$\underbrace{\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}}_{\text{Identity Matrix}} \Longrightarrow \underbrace{\begin{bmatrix}0 & 1 & 0\\ 0 & 0 & 1\\ 1 & 0 & 0\end{bmatrix}}_{\text{Transformation Matrix}}$$

$$\begin{bmatrix}0 & 0 & 1\\ 1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix} \cdot \begin{bmatrix}a & b & c\\ d & e & f\\ g & h & i\end{bmatrix} = \underbrace{\begin{bmatrix}c & a & b\\ f & d & e\\ i & g & h\end{bmatrix}}_{\text{The horizontally shifted matrix}}$$

3 Vertical Shifting

Very similar to the horizontal position change, the vertical position change merely requires the transformation matrix to be shifted row-wise as opposed to column-wise.

$$\underbrace{\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}}_{\text{Identity Matrix}} \Longrightarrow \underbrace{\begin{bmatrix}0 & 0 & 1\\ 1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix}}_{\text{Transformation Matrix}}$$

(a) Photo 1 Horizontal Shift

(b) Photo 2 - Horizontal Shift

Figure 6: Horizontally Shifted Images

Unlike the horizontal matrix shift, the order by which the transformation matrix is applied is reversed:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} = \underbrace{\begin{bmatrix} g & h & i \\ a & b & c \\ d & e & f \end{bmatrix}}_{\text{The vertically shifted matrix}}$$

(a) Photo 1 - Vertical and Horizontal Shift

(b) Photo 2 - Vertital and Horizontal Shift

Figure 7: Vertically Shifted Images

4 Inversion

In order to flip a matrix upside down, we first had to generate an identity matrix of the appropriate size where the rows had the opposite diagonal direction.

$$\underbrace{\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}}_{\text{Identity Matrix}} \implies \underbrace{\begin{bmatrix}0 & 0 & 1\\ 0 & 1 & 0\\ 1 & 0 & 0\end{bmatrix}}_{\text{Transformation Matrix}}$$

This was done by setting up the identity matrix as a two dimensional array; in other words, a list of lists. Then this list was iterated through and each list in the main list was flipped front-to-back. This action had the same effect as flipping the entire matrix on the horizontal axis. Finally, as before with the shifting process, we multiplied the matrix on the appropriate side of the matrix.

$$\begin{bmatrix}0 & 0 & 1\\ 0 & 1 & 0\\ 1 & 0 & 0\end{bmatrix} \cdot \begin{bmatrix}a & b & c\\ d & e & f\\ g & h & i\end{bmatrix} = \underbrace{\begin{bmatrix}g & h & i\\ d & e & f\\ a & b & c\end{bmatrix}}_{\text{The inverted matrix}}$$

5 Transposition

It is simple to visualize the effect of transposing a matrix; it would be a rotation about the main diagonal. The resulting image will be rotated $90°$. Taking the transpose again would give the original image orientation following the properties of transposed matrices:

$$A = (A^T)^T$$

The effect can be seen in Figure 8:

Figure 8: An example of a transposed image

6 DST

7 Compression

8 Optimization

9 Code

The entire codebase for the project follows, and is available for download here.[1]

[1] If you are unable to download these attached files, please go to this link

Figure 9: The plot of the Discrete Sine Transform

9.1 Python

The Python code to generate the images is included below.

```
#!/usr/bin/env python
'''
APPM 2360 Differential Equations Project Two
  |-Will Farmer
  |-Jeffrey Milhorn
  |-Patrick Harrington

This code takes the two given images and performs several
mathematical operations on them using matrix methods.
'''

import sys                          # Import system library
import scipy.misc                   # Import image processing libraries
import numpy                        # Import matrix libraries
import matplotlib.pyplot as plt     # Import plotting libraries
import pp                           # Library for Parallel Processing

jobServer = pp.Server() # Create a new jobserver
jobs      = []          # List of jobs to complete

def main():
    # Open images for manipulation
    print('Opening Images')
    image1 = scipy.misc.imread('../img/photo1.jpg')
    image2 = scipy.misc.imread('../img/photo2.jpg')

    # Run manipulations on both images
    print('Generating Manipulations')
    manipulate(image1, '1')
    manipulate(image2, '2')

    # Visualize Determinants of DST Matrix
    print('Generating Determinant Graph')
    visualize_s()

    # Compress images using DST
    print('Compressing Images')
    jobs.append(
            jobServer.submit(compression,
```

```
                                (image1, '1', 0.5),
                                (create_grayscale, dst, create_S),
                                ('numpy', 'scipy.misc'))
                ) # Add a new job to compress our first image
        jobs.append(
                jobServer.submit(compression,
                                (image2, '2', 0.5),
                                (create_grayscale, dst, create_S),
                                ('numpy', 'scipy.misc'))
                ) # Add a new job to compress our second image

        # Analyze Compression Effectiveness
        print('Generating Compression Effectiveness')
        comp_effect(image1, image2)

        # Create Picture Grid
        print('Generating Picture Grid')
        mass_pics(image1, '1')
        mass_pics(image2, '2')

        for job in jobs:
            job() # Evaulate all current jobs

def manipulate(image, name):
    '''
    Manipulate images as directed
    1) Create grayscale image
    2) Produce horizontal shifts
    3) Produce Vertical/Horizontal Shifts
    4) Flip image vertically
    '''
    # Create grayscale
    g = create_grayscale(image.copy())
    scipy.misc.imsave('../img/gray%s.png' %name, g)

    # Shift Horizontally
    hs = shift_hort(g)
    scipy.misc.imsave('../img/hsg%s.png' %name, hs)

    # Shift Hort/Vert
    hs = shift_hort(g)
    vhs = shift_vert(hs.copy())
    scipy.misc.imsave('../img/vhsg%s.png' %name, vhs)

    # Flip
    flipped = flip(g)
    scipy.misc.imsave('../img/flip%s.png' %name, flipped)

def flip(image):
    '''
    flips an image
    Essentially just multiplies it by a flipped id matrix
    '''
    il = numpy.identity(len(image)).tolist()  # Creates a matching identity
```

```
94      for row in il: # Reverses the identity matrix
95          row.reverse()
96      i       = numpy.array(il) # Turns it into a formal array
97      return numpy.dot(i, image) # Dots them together
98
99  def shift_hort(image):
100     '''
101     Shift an image horizontally
102     1) Create rolled identity matrix:
103         | 0 0 1 |
104         | 1 0 0 |
105         | 0 1 0 |
106     2) Dot with image
107     '''
108     i       = numpy.roll(numpy.identity(len(image[0])),
109                     240, axis=0) # Create rolled idm
110     shifted = numpy.dot(image, i) # dot with image
111     return shifted
112
113 def shift_vert(image):
114     '''
115     Shift an image horizontally
116     1) Create rolled identity matrix:
117         | 0 0 1 |
118         | 1 0 0 |
119         | 0 1 0 |
120     2) Dot with image
121     '''
122     i       = numpy.roll(numpy.identity(len(image)),
123                     100, axis=0) # create rolled idm
124     shifted = numpy.dot(i, image) # dot with image
125     return shifted
126
127 def create_grayscale(image):
128     '''
129     Creates grayscale image from given matrix
130     1) Create ratio matrix
131     2) Dot with image
132     '''
133     ratio = numpy.array([30., 59., 11.])
134     return numpy.dot(image.astype(numpy.float), ratio)
135
136 def shift_hort_color(image):
137     '''
138     Shift a color image horizontally
139     1) Create identity matrix that looks as such:
140         | 0 0 1 |
141         | 1 0 0 |
142         | 0 1 0 |
143     2) Dot it with image matrix
144     3) Return Transpose
145     '''
146     # Create an identity matrix and roll the rows
147     i       = numpy.roll(
```

```
148             numpy.identity(
149                 len(image[0]))
150             , 240, axis=0)
151     shifted = numpy.dot(i, image) # Dot with image
152     return numpy.transpose(shifted) # Return transpose
153
154 def compression(image, name, p):
155     '''
156     Compress the image using DST
157     '''
158     g = create_grayscale(image.copy()) # Create grayscale image matrix copy
159     t = dst(g)  # Acquire DST matrix of image
160     (row_size, column_size) = numpy.shape(t) # Size of t
161     for row in range(row_size):
162         for col in range(column_size):
163             if (row + col + 2) > (2 * p * column_size):
164                 t[row][col] = 0 # if the data is above a set line, delete it
165     scipy.misc.imsave('../img/comp%s.png' %name, dst(t))
166
167 def dst(image):
168     '''
169     If given a grayscale image array, use the DST formula
170     and return the result
171     Uses this method:
172         image = X
173         DST   = S
174         Y = S.(X.S)
175     '''
176     rows    = numpy.dot(image, create_S(len(image[0])))
177     columns = numpy.dot(create_S(len(image)), rows)
178     return columns
179
180 def create_S(n):
181     '''
182     Discrete Sine Transform
183     1) Initialize variables
184     2) For each row and column, create an entry
185     '''
186     new_array = []  # What we will be filling
187     size      = n
188     for row in range(size):
189         new_row = []    # New row for every row
190         for col in range(size):
191             S = ((numpy.sqrt(2.0 / size)) * # our equation
192                  (numpy.sin((numpy.pi * ((row + 1) - (1.0/2.0)) *
193                      ((col + 1) - (1.0/2.0)))/(size))))
194             new_row.append(S) # Append entry to row list
195         new_array.append(new_row) # append row to array
196     return_array = numpy.array(new_array)
197     return return_array
198
199 def mass_pics(image, name):
200     '''
201     Create a lot of compressed Pictures
```

```
202     '''
203     answer = raw_input('Create .gif Images? (y/n) ')
204     if answer == 'n':
205         return None # It takes a while, so it's optional
206     domain = numpy.arange(0, 1.01, 0.01) # Range of p vals
207     for p in domain:
208         jobs.append(
209                 jobServer.submit(compression,
210                     (image, 'array_%s_%f' %(name, p), p),
211                     (create_grayscale, dst, create_S),
212                     ('numpy', 'scipy.misc'))
213                 ) # For each value of p, add a new compression job
214
215 def visualize_s():
216     '''
217     DST
218     Visualize the discrete sine transform equation implemented below.
219     Uses matplotlib to create graph
220     '''
221     nrange   = numpy.arange(1, 33, 1) # Create values range [1,32] stepsize 1
222     det_plot = plt.figure() # New matplotlib class instance for a figure
223     det_axes = det_plot.add_axes([0.1, 0.1, 0.8, 0.8]) # Add axes to figure
224     yrange   = [] # Create an empty y range (we'll be adding to this)
225     for number in nrange:
226         array = create_S(number)    # Get a new array with size n
227         yrange.append(numpy.linalg.det(array)) # append determinant to yrange
228     det_axes.plot(nrange, yrange, label='Set of determinants') # Create line
229     det_axes.plot(nrange, nrange*0, 'k:')   # Also create line at y=0
230     det_axes.legend(loc=4) # Place legend
231     plt.xlabel('Size of Discrete Sine Transform Matrix') # Label X
232     plt.ylabel('Determinant of Matrix') # Label Y
233     plt.title('Size of Matrix vs. its Determinant') # Title
234     plt.savefig('../img/dst_dets.png') # Save as a png
235
236 def comp_effect(image1, image2):
237     '''
238     Analyzes compression effectiveness
239     If the image already exists, it will not run this
240     '''
241     try:
242         open('../img/bitcount.png', 'r')
243         open('../img/bitrat.png', 'r')
244         print(' |-> Graphs already created, skipping.\
245                 (Delete existing graphs to recreate)')
246         # If it already exists, don't create it. (It takes a while)
247     except IOError:
248         g1 = create_grayscale(image1.copy()) # Create grayscale from copy of 1
249         g2 = create_grayscale(image2.copy()) # Create grayscale from copy of 2
250
251         domain1 = numpy.arange(0.0, 1.01, 0.01) # Range of p values
252         domain2 = numpy.arange(0.0, 1.01, 0.01) # Range of p values
253
254         # Parallelize System and generate range
255         count_y1, rat_y1 = jobServer.submit(get_yrange,
```

```
                        (domain1, g1),
                        (dst, clear_vals, create_S),
                        ('numpy', 'scipy.misc'))()
        count_y2, rat_y2 = jobServer.submit(get_yrange,
                        (domain2, g2),
                        (dst, clear_vals, create_S),
                        ('numpy', 'scipy.misc'))()

        count_plot = plt.figure() # New class instance for a figure
        count_axes = count_plot.add_axes([0.1, 0.1, 0.8, 0.8]) # Add axes
        count_axes.plot(domain1, count_y1, label='Image 1')
        count_axes.plot(domain2, count_y2, label='Image 2')
        count_axes.legend(loc=4)
        plt.xlabel("Value of p")
        plt.ylabel("Number of Non-Zero Bytes")
        plt.title("Compression Effectiveness")
        plt.savefig("../img/bitcount.png")

        ratio_plot = plt.figure() # New class instance for a figure
        ratio_axes = ratio_plot.add_axes([0.1, 0.1, 0.8, 0.8]) # Add axes
        ratio_axes.plot(domain1, rat_y1, label='Image 1')
        ratio_axes.plot(domain2, rat_y2, label='Image 2')
        ratio_axes.legend(loc=4)
        plt.xlabel("Value of p")
        plt.ylabel("Ratio of Non-Zero Bytes to Total Bytes")
        plt.title("Compression Effectiveness")
        plt.savefig("../img/bitrat.png")

def get_yrange(domain, g):
    bit_count = [] # Range for image
    bit_ratio = []
    for p in domain:
        t = dst(g.copy()) # Transform 1
        initial_count = float(numpy.count_nonzero(t))
        clear_vals(t, p) # Strip of high-freq data
        final_count = float(numpy.count_nonzero(t))
        bit_count.append(final_count) # Append number of non-zero entries
        bit_ratio.append(final_count / initial_count)
    return bit_count, bit_ratio

def clear_vals(transform, p):
    '''
    Takes image and deletes high frequency
    '''
    (row_size, column_size) = numpy.shape(transform) # Size of t
    for row in range(row_size):
        for col in range(column_size):
            if (row + col + 2) > (2 * p * column_size):
                transform[row][col] = 0 # if the data is above line, delete it
    return transform

if __name__ == '__main__':
    sys.exit(main())
```

9.2 MATLAB Code

Some MATLAB Code was also made that features equivalent functionality

Grayscale

```
function gray_image=grayscale(image)
% This is a function to take an image in jpg form and put it into grayscale

% This reads in the image
image_matrix=imread(image);

% get the dimensions
[rows,columns,~]=size(image_matrix);

% preallocate
gray_image = zeros(rows,columns);
for a=1:rows;
    for b=1:columns;
            gray_image(a,b)=0.3*image_matrix(a,b,1)...
                +0.59*image_matrix(a,b,2)...
                +0.11*image_matrix(a,b,3);
    end
end
imwrite(uint8(gray_image),'name.jpg')

end
```

Horizontal Shifting

```
function [hshifted_image] = hshift(image)

% c is the number of cols we want to shift by
c = 240;

% read in the image and make it a nice little matrix
image_matrix=double(imread(image));

% get the dimensions of the matrix
[rows, cols] = size(image_matrix);

% get the largest dimension for the identity matrix
n = max(rows, cols);

% Preallocate for the id matrix:
T = zeros(n,n);

% generate a generic identity matrix
id = eye(n);

%fill in the first c cols of T with the last c cols of id
T(:,1:c)=id(:,n-(c-1):n);
%fill in the rest of T with the first part of id
```

```
T(:,c+1:n) = id(:,1:n-c);

hshifted_image=uint8(image_matrix*T);

imwrite(hshifted_image,'hshifted.jpg');
```

Vertical Shifting

```
function [vshifted_image] = vshift(image)

% r is the number of rows we want to shift by
r = 100;

% read in the image and make it a nice little matrix
image_matrix=double(imread(image));

% get the dimensions of the matrix
[rows, cols] = size(image_matrix);

% get the largest dimension for the identity matrix
n = min(rows, cols);

% Preallocate for the id matrix:
T = zeros(n,n);

% generate a generic identity matrix
id = eye(n);

%fill in the first c cols of T with the last c cols of id
T(1:r,:)=id(n-(r-1):n,:);
%fill in the rest of T with the first part of id
T(r+1:n,:) = id(1:n-r,:);

vshifted_image=uint8(T*image_matrix);

imwrite(vshifted_image,'vshifted.jpg');
```

9.3 MATLAB Code

Some MATLAB Code was also made that features equivalent functionality

Grayscale

```
function gray_image=grayscale(image)
% This is a function to take an image in jpg form and put it into grayscale

% This reads in the image
image_matrix=imread(image);

% get the dimensions
[rows,columns,~]=size(image_matrix);

```

```
% preallocate
gray_image = zeros(rows,columns);
for a=1:rows;
    for b=1:columns;
            gray_image(a,b)=0.3*image_matrix(a,b,1)...
                +0.59*image_matrix(a,b,2)...
                +0.11*image_matrix(a,b,3);
    end
end
imwrite(uint8(gray_image),'name.jpg')

end
```

Horizontal Shifting

```
function [hshifted_image] = hshift(image)

% c is the number of cols we want to shift by
c = 240;

% read in the image and make it a nice little matrix
image_matrix=double(imread(image));

% get the dimensions of the matrix
[rows, cols] = size(image_matrix);

% get the largest dimension for the identity matrix
n = max(rows, cols);

% Preallocate for the id matrix:
T = zeros(n,n);

% generate a generic identity matrix
id = eye(n);

%fill in the first c cols of T with the last c cols of id
T(:,1:c)=id(:,n-(c-1):n);
%fill in the rest of T with the first part of id
T(:,c+1:n) = id(:,1:n-c);

hshifted_image=uint8(image_matrix*T);

imwrite(hshifted_image,'hshifted.jpg');
```

Vertical Shifting

```
function [vshifted_image] = vshift(image)

% r is the number of rows we want to shift by
r = 100;

% read in the image and make it a nice little matrix
```

```
7  image_matrix=double(imread(image));
8
9  % get the dimensions of the matrix
10 [rows, cols] = size(image_matrix);
11
12 % get the largest dimension for the identity matrix
13 n = min(rows, cols);
14
15 % Preallocate for the id matrix:
16 T = zeros(n,n);
17
18 % generate a generic identity matrix
19 id = eye(n);
20
21 %fill in the first c cols of T with the last c cols of id
22 T(1:r,:)=id(n-(r-1):n,:);
23 %fill in the rest of T with the first part of id
24 T(r+1:n,:) = id(1:n-r,:);
25
26 vshifted_image=uint8(T*image_matrix);
27
28 imwrite(vshifted_image,'vshifted.jpg');
```