Image Manipulation Using Matrix Techniques

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

Contents

List of Figures

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.

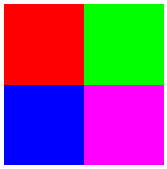


Figure 2: A simple RGB image

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

The Python code the generate these images is below.

```
1 def create_grayscale(image):
2     '''
3     Creates grayscale image from given matrix
4     1) Create ratio matrix
```

```
    2) Dot with image
    '''
    ratio = numpy.array([30., 59., 11.])
    return numpy.dot(image.astype(numpy.float), ratio)
```

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.

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

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

(a) Photo 1 Horizontal Shift

(b) Photo 2 - Horizontal Shift

Figure 6: Horizontally Shifted Images

In the figures (6a) and (6b), we have performed a shift of 240 pixels on each image. Note, for the smaller image (6b), the image itself is only 200 pixels wide, therefore a shift of 240 is equivalent to a shift of 40 pixels.

The python code to shift these images horizontally is below.

```
def shift_hort(image):
    '''
    Shift an image horizontally
    1) Create rolled identity matrix:
        | 0 0 1 |
        | 1 0 0 |
        | 0 1 0 |
    2) Dot with image
    '''
    i       = numpy.roll(numpy.identity(len(image[0])),
                         240, axis=0) # Create rolled idm
```

```
    shifted = numpy.dot(image, i) # dot with image
    return shifted
```

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.

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

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

We can perform both horizontal and vertical translations on our image matrix, but we must do the operations separately for photo1.jpg since they do not involve the same number of iterations. For example, we can first do the horizontal translation by using the same procedure above where the transformation matrix is second in the matrix multiplication (the transformation matrix would be dimension $n \times n$, where n equals the column dimension of photo1.jpg, 408). After performing the 240 iterations of this horizontal translation we can then translate the image matrix vertically. We now place the transformation matrix first in the matrix multiplication; its dimensions must match the row dimension of the image. Therefore, this vertical transformation matrix is 201×201.

The Python code to shift these images vertically is below.

```
def shift_vert(image):
    '''
    Shift an image horizontally
    1) Create rolled identity matrix:
        | 0 0 1 |
        | 1 0 0 |
        | 0 1 0 |
    2) Dot with image
```

```
    '''
    i       = numpy.roll(numpy.identity(len(image)),
                         100, axis=0) # create rolled idm
    shifted = numpy.dot(i, image) # dot with image
    return shifted
```

4 Inversion

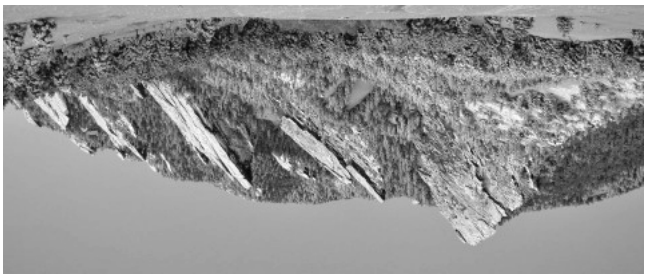
In order to flip a matrix upside down, we first had to generate an identity matrix of the appropriate size where the rows had the opposite diagonal direction.

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

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.

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

This resulted in images that were upside down, as is demonstrated in figure (8)



(a) Image 1 Flipped

(b) Image 2 Flipped

Figure 8: Our Flipped Images

The Python code to flip images is below.

```
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
    for row in il: # Reverses the identity matrix
        row.reverse()
    i       = numpy.array(il) # Turns it into a formal array
    return numpy.dot(i, image) # Dots them together
```

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 9:

(a) Image 1 Transposed

(b) Image 2 Transposed

Figure 9: An example of a transposed image

6 DST

From the plot of the determinant of S as a function of n for n from 1 to 32 shown in Figure 10, it can be seen that the determinant has strictly discrete values of either 1 or -1 alternative every two values, and follows a sinusoidal pattern. It is also noticeable that the plot is an odd function.

The equation used to create each of these matrices is expressed in (1).

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

The Python code used to generate the matrices as well as the graph is shown below.

```
1 def create_S(n):
2     '''
3     Discrete Sine Transform
4     1) Initialize variables
5     2) For each row and column, create an entry
6     '''
7     new_array = []  # What we will be filling
8     size      = n
```

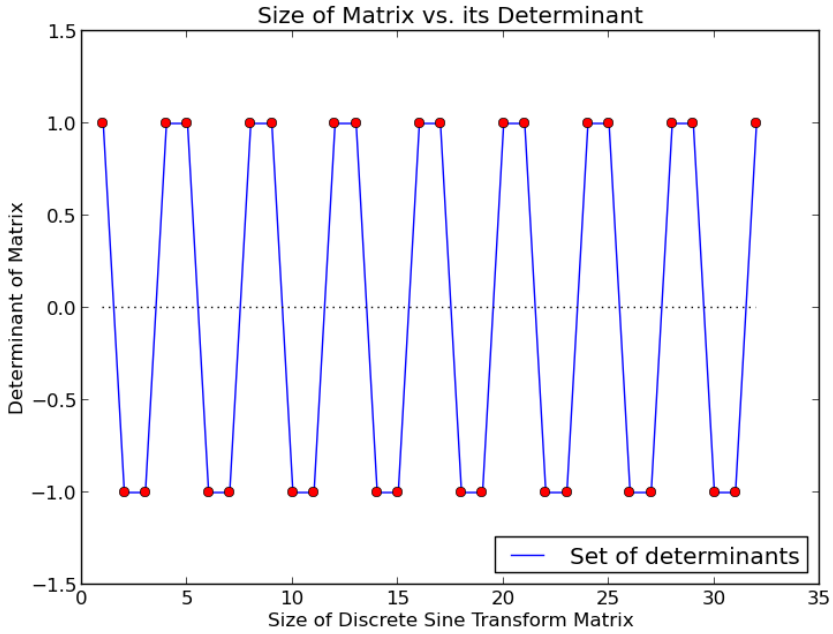


Figure 10: Determinants of $S(n)$

```
    for row in range(size):
        new_row = []     # New row for every row
        for col in range(size):
            S = ((numpy.sqrt(2.0 / size)) * # our equation
                 (numpy.sin((numpy.pi * ((row + 1) - (1.0/2.0)) *
                     ((col + 1) - (1.0/2.0)))/(size))))
            new_row.append(S) # Append entry to row list
        new_array.append(new_row) # append row to array
    return_array = numpy.array(new_array)
    return return_array
```

```
def visualize_s():
    '''
    DST
    Visualize the discrete sine transform equation implemented below.
    Uses matplotlib to create graph
    '''
    nrange   = numpy.arange(1, 33, 1) # Create values range [1,32] stepsize 1
    det_plot = plt.figure() # New matplotlib class instance for a figure
    det_axes = det_plot.add_axes([0.1, 0.1, 0.8, 0.8]) # Add axes to figure
    yrange   = [] # Create an empty y range (we'll be adding to this)
    for number in nrange:
        array = create_S(number)     # Get a new array with size n
        yrange.append(numpy.linalg.det(array)) # append determinant to yrange
    det_axes.plot(nrange, yrange, label='Set of determinants') # Create line
    det_axes.plot(nrange, yrange, 'ro') # Add points
    det_axes.plot(nrange, nrange*0, 'k:')    # Also create line at y=0
    det_axes.legend(loc=4) # Place legend
    plt.xlabel('Size of Discrete Sine Transform Matrix') # Label X
```

```
    plt.ylabel('Determinant of Matrix') # Label Y
    plt.title('Size of Matrix vs. its Determinant') # Title
    plt.savefig('../img/dst_dets.png') # Save as a png
```

7 Restrictions on Compression with the Discrete Sine Transform

With the given equation to transform images using the Discrete Sine Transform (1), there does exist a limitation on the initial image aspect ratio – the image *must* by square. If it is not square, then the dot product will not work, and the image will not be compressed. The reason behind this is that since we are performing a dot product on the same matrix on either side, we know that in order for it to work it needs to be the same size after either operation is performed. The only matrix this holds true for is a square matrix.

That being said, the code below expresses a different algorithm. Instead of being limited to square matrices through the nuances of dot products, the code instead separates the two operations and performs them separately using two differently sized DST matrices. This algorithm is not limited by square matrices since it creates a new DST matrix for each operation.

```
def dst(image):
    '''
    If given a grayscale image array, use the DST formula
    and return the result
    Uses this method:
        image = X
        DST   = S
        Y = S.(X.S)
    '''
    rows    = numpy.dot(image, create_S(len(image[0])))
    columns = numpy.dot(create_S(len(image)), rows)
    return columns
```

8 Compression

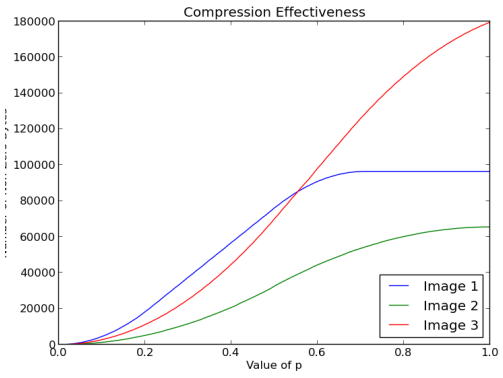
Given that we know the special properties of our Discrete Sine Transform and the algorithm behind it, we also know the to undo it, the same method is used. This is a symmetric algorithm that works the same forwards as backwards. Therefore to compress an image we can apply the algorithm one way, delete the extraneous information, and then apply the same algorithm again to revert it back to a readable image.

9 Optimization

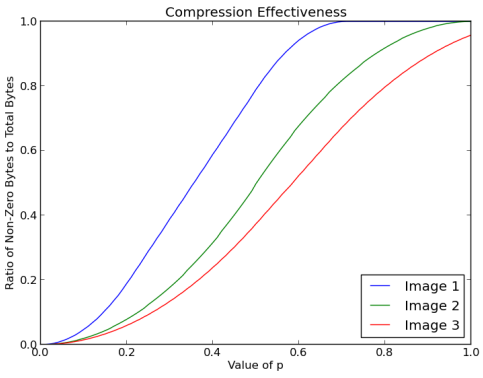
Human vision has noticeable thresholds for the perception of light frequency. We are a lot more sensitive to lower frequencies compared to higher ones. JPEG image compression involves identifying high frequency pixel groupings and removing them from the image; less data in the image matrix means that it takes up less digital storage. We can compress the image by using the DST to identify high frequency data. We can also vary the extent of compression using a variable p, which goes from 0 to 1 where 0 represents a blank image and 1 represents an uncompressed image. To see a demonstration of how different values of p affect the image's quality, please either click this link or download it here.

Because our equation focusses only on the high frequency values of the image, we can eliminate many pixels before our brains register a degradation in quality. Below is a graph that shows how much data is removed from each image as p gets larger.

Images can be compressed to low values of p without being noticeable, especially if the images are primarily uniform and consist of low frequency pixels. This is because of the threshold frequencies in human vision.



(a) Bit Count vs. p Value



(b) Bit Ratio vs. p Value

Due to the inherent nature of images having different amounts of high frequency values, different values of p will be appropriate for different images. For our first image, it initially will not have a large difference in quality as p gets smaller, however for our last image it will immediately start reducing in size. Therefore different values of P are appropriate for these different images.

10 Conclusion

Since digital images are represented as three dimensional matrices, image manipulation involves matrix operations. By using matrix multiplication, transpose, and the Discreet Sine Transformation (DST), we were able to translate, rotate, and compress multiple sized images. These analyses prove the usefulness of matrices when working with large sets of finite data, and they merely serve as an introduction to the powerful information processing that these tools provide.

To see all images produced from this project, please click this link and browse all of the images.

11 Code

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

11.1 Python

The Python code to generate the images is included below.

```
1  #!/usr/bin/env python
2  '''
3  APPM 2360 Differential Equations Project Two
4    |-Will Farmer
5    |-Jeffrey Milhorn
6    |-Patrick Harrington
7
8  This code takes the two given images and performs several
9  mathematical operations on them using matrix methods.
10 '''
11
12 import sys                          # Import system library
13 import scipy.misc                   # Import image processing libraries
```

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

```
14 import numpy                      # Import matrix libraries
15 import matplotlib.pyplot as plt   # Import plotting libraries
16 import pp                         # Library for Parallel Processing
17
18 jobServer = pp.Server() # Create a new jobserver
19 jobs      = []          # List of jobs to complete
20
21 def main():
22     # Open images for manipulation
23     print('Opening Images')
24     image1 = scipy.misc.imread('../img/photo1.jpg')
25     image2 = scipy.misc.imread('../img/photo2.jpg')
26     image3 = scipy.misc.imread('../img/sadfox.jpg')
27
28     # Run manipulations on both images
29     print('Generating Manipulations')
30     manipulate(image1, '1')
31     manipulate(image2, '2')
32     manipulate(image3, '3')
33
34     # Visualize Determinants of DST Matrix
35     print('Generating Determinant Graph')
36     visualize_s()
37
38     # Compress images using DST
39     print('Compressing Images')
40     jobs.append(
41             jobServer.submit(compression,
42                              (image1, '1', 0.5),
43                              (create_grayscale, dst, create_S),
44                              ('numpy', 'scipy.misc'))
45             ) # Add a new job to compress our first image
46     jobs.append(
47             jobServer.submit(compression,
48                              (image2, '2', 0.5),
49                              (create_grayscale, dst, create_S),
50                              ('numpy', 'scipy.misc'))
51             ) # Add a new job to compress our second image
52     jobs.append(
53             jobServer.submit(compression,
54                              (image3, '3', 0.5),
55                              (create_grayscale, dst, create_S),
56                              ('numpy', 'scipy.misc'))
57             ) # Add a new job to compress our third image
58
59     # Analyze Compression Effectiveness
60     print('Generating Compression Effectiveness')
61     comp_effect(image1, image2, image3)
62
63     # Create Picture Grid
64     print('Generating Picture Grid')
65     mass_pics(image1, '1')
66     mass_pics(image2, '2')
67     mass_pics(image3, '3')
```

```
68
69      for job in jobs:
70          job() # Evaulate all current jobs
71      jobServer.get_stats()
72
73  def manipulate(image, name):
74      '''
75      Manipulate images as directed
76      1) Create grayscale image
77      2) Produce horizontal shifts
78      3) Produce Vertical/Horizontal Shifts
79      4) Flip image vertically
80      '''
81      # Create grayscale
82      g = create_grayscale(image.copy())
83      scipy.misc.imsave('../img/gray%s.png' %name, g)
84
85      # Shift Horizontally
86      hs = shift_hort(g)
87      scipy.misc.imsave('../img/hsg%s.png' %name, hs)
88
89      # Shift Hort/Vert
90      hs = shift_hort(g)
91      vhs = shift_vert(hs.copy())
92      scipy.misc.imsave('../img/vhsg%s.png' %name, vhs)
93
94      # Flip
95      flipped = flip(g)
96      scipy.misc.imsave('../img/flip%s.png' %name, flipped)
97
98      # Create Transpose
99      tp = numpy.transpose(g)
100     scipy.misc.imsave('../img/transpose%s.png' %name, tp)
101
102 def flip(image):
103     '''
104     flips an image
105     Essentially just multiplies it by a flipped id matrix
106     '''
107     il = numpy.identity(len(image)).tolist()  # Creates a matching identity
108     for row in il: # Reverses the identity matrix
109         row.reverse()
110     i       = numpy.array(il) # Turns it into a formal array
111     return numpy.dot(i, image) # Dots them together
112
113 def shift_hort(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[0])),
123                          240, axis=0) # Create rolled idm
124     shifted = numpy.dot(image, i) # dot with image
125     return shifted
126 
127 def shift_vert(image):
128     '''
129     Shift an image horizontally
130     1) Create rolled identity matrix:
131         | 0 0 1 |
132         | 1 0 0 |
133         | 0 1 0 |
134     2) Dot with image
135     '''
136     i       = numpy.roll(numpy.identity(len(image)),
137                          100, axis=0) # create rolled idm
138     shifted = numpy.dot(i, image) # dot with image
139     return shifted
140 
141 def create_grayscale(image):
142     '''
143     Creates grayscale image from given matrix
144     1) Create ratio matrix
145     2) Dot with image
146     '''
147     ratio = numpy.array([30., 59., 11.])
148     return numpy.dot(image.astype(numpy.float), ratio)
149 
150 def shift_hort_color(image):
151     '''
152     Shift a color image horizontally
153     1) Create identity matrix that looks as such:
154         | 0 0 1 |
155         | 1 0 0 |
156         | 0 1 0 |
157     2) Dot it with image matrix
158     3) Return Transpose
159     '''
160     # Create an identity matrix and roll the rows
161     i       = numpy.roll(
162             numpy.identity(
163                 len(image[0]))
164             , 240, axis=0)
165     shifted = numpy.dot(i, image) # Dot with image
166     return numpy.transpose(shifted) # Return transpose
167 
168 def compression(image, name, p):
169     '''
170     Compress the image using DST
171     '''
172     g = create_grayscale(image.copy()) # Create grayscale image matrix copy
173     t = dst(g)  # Acquire DST matrix of image
174     (row_size, column_size) = numpy.shape(t) # Size of t
175     for row in range(row_size):
```

```
176             for col in range(column_size):
177                 if (row + col + 2) > (2 * p * column_size):
178                     t[row][col] = 0 # if the data is above a set line, delete it
179         scipy.misc.imsave('../img/comp%s.png' %name, dst(t))
180
181     def dst(image):
182         '''
183         If given a grayscale image array, use the DST formula
184         and return the result
185         Uses this method:
186             image = X
187             DST   = S
188             Y = S.(X.S)
189         '''
190         rows    = numpy.dot(image, create_S(len(image[0])))
191         columns = numpy.dot(create_S(len(image)), rows)
192         return columns
193
194     def create_S(n):
195         '''
196         Discrete Sine Transform
197         1) Initialize variables
198         2) For each row and column, create an entry
199         '''
200         new_array = []  # What we will be filling
201         size      = n
202         for row in range(size):
203             new_row = []    # New row for every row
204             for col in range(size):
205                 S = ((numpy.sqrt(2.0 / size)) * # our equation
206                      (numpy.sin((numpy.pi * ((row + 1) - (1.0/2.0)) *
207                          ((col + 1) - (1.0/2.0)))/(size))))
208                 new_row.append(S) # Append entry to row list
209             new_array.append(new_row) # append row to array
210         return_array = numpy.array(new_array)
211         return return_array
212
213     def mass_pics(image, name):
214         '''
215         Create a lot of compressed Pictures
216         '''
217         answer = raw_input('Create .gif Images? (y/n) ')
218         if answer == 'n':
219             return None # It takes a while, so it's optional
220         domain = numpy.arange(0, 1.01, 0.01) # Range of p vals
221         for p in domain:
222             jobs.append(
223                     jobServer.submit(compression,
224                         (image, 'array_%s_%f' %(name, p), p),
225                         (create_grayscale, dst, create_S),
226                         ('numpy', 'scipy.misc'))
227                     ) # For each value of p, add a new compression job
228
229     def visualize_s():
```

```
    '''
    DST
    Visualize the discrete sine transform equation implemented below.
    Uses matplotlib to create graph
    '''
    nrange   = numpy.arange(1, 33, 1) # Create values range [1,32] stepsize 1
    det_plot = plt.figure() # New matplotlib class instance for a figure
    det_axes = det_plot.add_axes([0.1, 0.1, 0.8, 0.8]) # Add axes to figure
    yrange   = [] # Create an empty y range (we'll be adding to this)
    for number in nrange:
        array = create_S(number)    # Get a new array with size n
        yrange.append(numpy.linalg.det(array)) # append determinant to yrange
    det_axes.plot(nrange, yrange, label='Set of determinants') # Create line
    det_axes.plot(nrange, yrange, 'ro') # Add points
    det_axes.plot(nrange, nrange*0, 'k:')   # Also create line at y=0
    det_axes.legend(loc=4) # Place legend
    plt.xlabel('Size of Discrete Sine Transform Matrix') # Label X
    plt.ylabel('Determinant of Matrix') # Label Y
    plt.title('Size of Matrix vs. its Determinant') # Title
    plt.savefig('../img/dst_dets.png') # Save as a png

def comp_effect(image1, image2, image3):
    '''
    Analyzes compression effectiveness
    If the image already exists, it will not run this
    '''
    try:
        open('../img/bitcount.png', 'r')
        open('../img/bitrat.png', 'r')
        print(' |-> Graphs already created, skipping.\
                (Delete existing graphs to recreate)')
        # If it already exists, don't create it. (It takes a while)
    except IOError:
        g1 = create_grayscale(image1.copy()) # Create grayscale from copy of 1
        g2 = create_grayscale(image2.copy()) # Create grayscale from copy of 2
        g3 = create_grayscale(image3.copy()) # Create grayscale from copy of 2

        domain1 = numpy.arange(0.0, 1.01, 0.01) # Range of p values
        domain2 = numpy.arange(0.0, 1.01, 0.01) # Range of p values
        domain3 = numpy.arange(0.0, 1.01, 0.01) # Range of p values

        # Parallelize System and generate range
        local_jobs = []
        local_jobs.append(jobServer.submit(get_yrange,
                        (domain1, g1),
                        (dst, clear_vals, create_S),
                        ('numpy', 'scipy.misc')))
        local_jobs.append(jobServer.submit(get_yrange,
                        (domain2, g2),
                        (dst, clear_vals, create_S),
                        ('numpy', 'scipy.misc')))
        local_jobs.append(jobServer.submit(get_yrange,
                        (domain3, g3),
                        (dst, clear_vals, create_S),
```

```
                        ('numpy', 'scipy.misc')))
        results = []
        for job in local_jobs:
            results.append(job())
        count_y1 = results[0][0] # Assign variables
        rat_y1   = results[0][1]
        count_y2 = results[1][0]
        rat_y2   = results[1][1]
        count_y3 = results[2][0]
        rat_y3   = results[2][1]

        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.plot(domain3, count_y3, label='Image 3')
        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.plot(domain3, rat_y3, label='Image 3')
        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
```

```
338     return transform
339
340 if __name__ == '__main__':
341     sys.exit(main())
```

11.2 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');
```

11.3 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');
```