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 LaTeX Source Code is attached.

Contents

List of Figures

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.

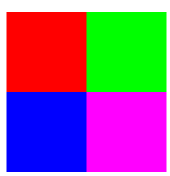


Figure 2: A simple RGB image

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.

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

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

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.

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

(a) Photo 1 Horizontal Shift

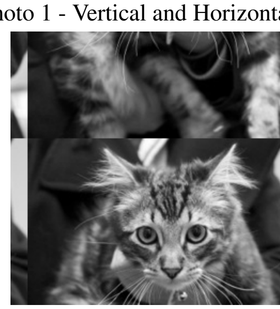
(b) Photo 2 - Horizontal Shift

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

From the plot of the determinant squared of S as a function of n for n from 1 to 32 shown in Figure 9, it can be seen that the determinant has strictly discrete values of either 1 or -1, and follows a sinusoidal pattern. It is also noticeable that the plot is an odd function.

The Discrete Sine Transform has the following equation:

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

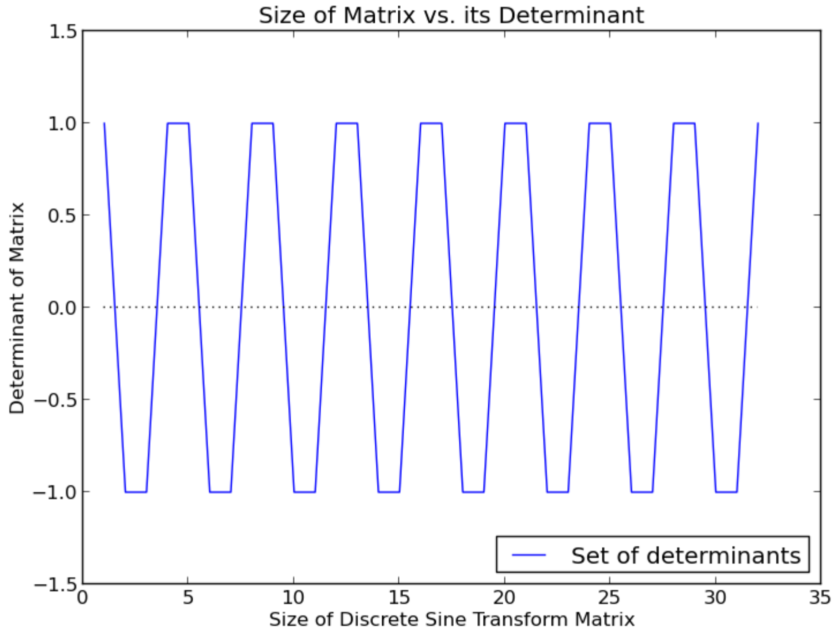


Figure 9: Δ^2 of $S(n)$

Figure 10: The plot of the Discrete Sine Transform

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.

```
    columns = numpy.dot(create_S(len(image)), rows)
    return columns

def create_S(n):
    '''
    Discrete Sine Transform
    1) Initialize variables
    2) For each row and column, create an entry
    '''
    new_array = []  # What we will be filling
    size      = n
    for row in range(size):
```

8 Compression

9 Optimization

10 Code

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

10.1 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')
```

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

```

    # 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
    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

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

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

def create_grayscale(image):
    '''
    Creates grayscale image from given matrix
    1) Create ratio matrix
    2) Dot with image
    '''
    ratio = numpy.array([30., 59., 11.])
```

```
    return numpy.dot(image.astype(numpy.float), ratio)

def shift_hort_color(image):
    '''
    Shift a color image horizontally
    1) Create identity matrix that looks as such:
         | 0 0 1 |
         | 1 0 0 |
         | 0 1 0 |
    2) Dot it with image matrix
    3) Return Transpose
    '''
    # Create an identity matrix and roll the rows
    i       = numpy.roll(
            numpy.identity(
                len(image[0]))
            , 240, axis=0)
    shifted = numpy.dot(i, image) # Dot with image
    return numpy.transpose(shifted) # Return transpose

def compression(image, name, p):
    '''
    Compress the image using DST
    '''
    g = create_grayscale(image.copy()) # Create grayscale image matrix copy
    t = dst(g)  # Acquire DST matrix of image
    (row_size, column_size) = numpy.shape(t) # Size of t
    for row in range(row_size):
        for col in range(column_size):
            if (row + col + 2) > (2 * p * column_size):
                t[row][col] = 0 # if the data is above a set line, delete it
    scipy.misc.imsave('../img/comp%s.png' %name, dst(t))

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

def create_S(n):
    '''
    Discrete Sine Transform
    1) Initialize variables
    2) For each row and column, create an entry
    '''
    new_array = []  # What we will be filling
    size      = 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 mass_pics(image, name):
    '''
    Create a lot of compressed Pictures
    '''
    answer = raw_input('Create .gif Images? (y/n) ')
    if answer == 'n':
        return None # It takes a while, so it's optional
    domain = numpy.arange(0, 1.01, 0.01) # Range of p vals
    for p in domain:
        jobs.append(
                jobServer.submit(compression,
                    (image, 'array_%s_%f' %(name, p), p),
                    (create_grayscale, dst, create_S),
                    ('numpy', 'scipy.misc'))
                ) # For each value of p, add a new compression job

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, 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):
    '''
    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

        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

        # Parallelize System and generate range
        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())
```

10.2 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');
```

10.3 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
```

```
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');
```