2020BTEIT00041

Om Vivek Gharge

Fractal Image Compression

CODE:

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from scipy import ndimage
from scipy import optimize
import numpy as np
import math

# Manipulate channels

def get_greyscale_image(img):
    return np.mean(img[:,:,:2], 2)

def extract_rgb(img):
    return img[:,:,0], img[:,:,1], img[:,:,2]

def assemble_rbg(img_r, img_g, img_b):
    shape = (img_r.shape[0], img_r.shape[1], 1)
    return np.concatenate((np.reshape(img_r, shape),
np.reshape(img_g, shape),
        np.reshape(img_b, shape)), axis=2)

# Transformations

def reduce(img, factor):
    result = np.zeros((img.shape[0] // factor, img.shape[1] //
factor))
    for i in range(result.shape[0]):
        for j in range(result.shape[1]):
            result[i,j] =
np.mean(img[i*factor:(i+1)*factor,j*factor:(j+1)*factor])
    return result

def rotate(img, angle):
    return ndimage.rotate(img, angle, reshape=False)

def flip(img, direction):
    return img[::direction,:]

def apply_transformation(img, direction, angle, contrast=1.0,
brightness=0.0):
    return contrast*rotate(flip(img, direction), angle) + brightness

# Contrast and brightness

def find_contrast_and_brightness1(D, S):
    # Fix the contrast and only fit the brightness
    contrast = 0.75
```

```
    brightness = (np.sum(D - contrast*S)) / D.size
    return contrast, brightness

def find_contrast_and_brightness2(D, S):
    # Fit the contrast and the brightness
    A = np.concatenate((np.ones((S.size, 1)), np.reshape(S, (S.size,
1))), axis=1)
    b = np.reshape(D, (D.size,))
    x, _, _, _ = np.linalg.lstsq(A, b)
    #x = optimize.lsq_linear(A, b, [(-np.inf, -2.0), (np.inf,
2.0)]).x
    return x[1], x[0]

# Compression for greyscale images

def generate_all_transformed_blocks(img, source_size,
destination_size, step):
    factor = source_size // destination_size
    transformed_blocks = []
    for k in range((img.shape[0] - source_size) // step + 1):
        for l in range((img.shape[1] - source_size) // step + 1):
            # Extract the source block and reduce it to the shape of
a destination block
            S =
reduce(img[k*step:k*step+source_size,l*step:l*step+source_size],
factor)
            # Generate all possible transformed blocks
            for direction, angle in candidates:
                transformed_blocks.append((k, l, direction, angle,
apply_transformation(S, direction, angle)))
    return transformed_blocks

def compress(img, source_size, destination_size, step):
    transformations = []
    transformed_blocks = generate_all_transformed_blocks(img,
source_size, destination_size, step)
    i_count = img.shape[0] // destination_size
    j_count = img.shape[1] // destination_size
    for i in range(i_count):
        transformations.append([])
        for j in range(j_count):
            print("{}/{} ; {}/{}".format(i, i_count, j, j_count))
            transformations[i].append(None)
            min_d = float('inf')
            # Extract the destination block
            D =
img[i*destination_size:(i+1)*destination_size,j*destination_size:(j+1
)*destination_size]
```

```
            # Test all possible transformations and take the best one
            for k, l, direction, angle, S in transformed_blocks:
                contrast, brightness =
find_contrast_and_brightness2(D, S)
                S = contrast*S + brightness
                d = np.sum(np.square(D - S))
                if d < min_d:
                    min_d = d
                    transformations[i][j] = (k, l, direction, angle,
contrast, brightness)
    return transformations

def decompress(transformations, source_size, destination_size, step,
nb_iter=8):
    factor = source_size // destination_size
    height = len(transformations) * destination_size
    width = len(transformations[0]) * destination_size
    iterations = [np.random.randint(0, 256, (height, width))]
    cur_img = np.zeros((height, width))
    for i_iter in range(nb_iter):
        print(i_iter)
        for i in range(len(transformations)):
            for j in range(len(transformations[i])):
                # Apply transform
                k, l, flip, angle, contrast, brightness =
transformations[i][j]
                S = reduce(iterations[-
1][k*step:k*step+source_size,l*step:l*step+source_size], factor)
                D = apply_transformation(S, flip, angle, contrast,
brightness)
                cur_img[i*destination_size:(i+1)*destination_size,j*d
estination_size:(j+1)*destination_size] = D
        iterations.append(cur_img)
        cur_img = np.zeros((height, width))
    return iterations

# Compression for color images

def reduce_rgb(img, factor):
    img_r, img_g, img_b = extract_rgb(img)
    img_r = reduce(img_r, factor)
    img_g = reduce(img_g, factor)
    img_b = reduce(img_b, factor)
    return assemble_rbg(img_r, img_g, img_b)

def compress_rgb(img, source_size, destination_size, step):
    img_r, img_g, img_b = extract_rgb(img)
    return [compress(img_r, source_size, destination_size, step), \
```

```
        compress(img_g, source_size, destination_size, step), \
        compress(img_b, source_size, destination_size, step)]

def decompress_rgb(transformations, source_size, destination_size,
step, nb_iter=8):
    img_r = decompress(transformations[0], source_size,
destination_size, step, nb_iter)[-1]
    img_g = decompress(transformations[1], source_size,
destination_size, step, nb_iter)[-1]
    img_b = decompress(transformations[2], source_size,
destination_size, step, nb_iter)[-1]
    return assemble_rbg(img_r, img_g, img_b)

# Plot

def plot_iterations(iterations, target=None):
    # Configure plot
    plt.figure()
    nb_row = math.ceil(np.sqrt(len(iterations)))
    nb_cols = nb_row
    # Plot
    for i, img in enumerate(iterations):
        plt.subplot(nb_row, nb_cols, i+1)
        plt.imshow(img, cmap='gray', vmin=0, vmax=255,
interpolation='none')
        if target is None:
            plt.title(str(i))
        else:
            # Display the RMSE
            plt.title(str(i) + ' (' +
'{0:.2f}'.format(np.sqrt(np.mean(np.square(target - img)))) + ')')
        frame = plt.gca()
        frame.axes.get_xaxis().set_visible(False)
        frame.axes.get_yaxis().set_visible(False)
    plt.tight_layout()

# Parameters

directions = [1, -1]
angles = [0, 90, 180, 270]
candidates = [[direction, angle] for direction in directions for
angle in angles]

# Tests

def test_greyscale():
    img = mpimg.imread('F:/Assignments/CA/Fractal Img
compression/fractal-image-compression/monkey.gif')
```

```
    img = get_greyscale_image(img)
    img = reduce(img, 4)
    plt.figure()
    plt.imshow(img, cmap='gray', interpolation='none')
    transformations = compress(img, 8, 4, 8)
    iterations = decompress(transformations, 8, 4, 8)
    plot_iterations(iterations, img)
    plt.show()

def test_rgb():
    img = mpimg.imread('lena.gif')
    img = reduce_rgb(img, 8)
    transformations = compress_rgb(img, 8, 4, 8)
    retrieved_img = decompress_rgb(transformations, 8, 4, 8)
    plt.figure()
    plt.subplot(121)
    plt.imshow(np.array(img).astype(np.uint8), interpolation='none')
    plt.subplot(122)
    plt.imshow(retrieved_img.astype(np.uint8), interpolation='none')
    plt.show()

if __name__ == '__main__':
    test_greyscale()
    #test_rgb()
```

Results:

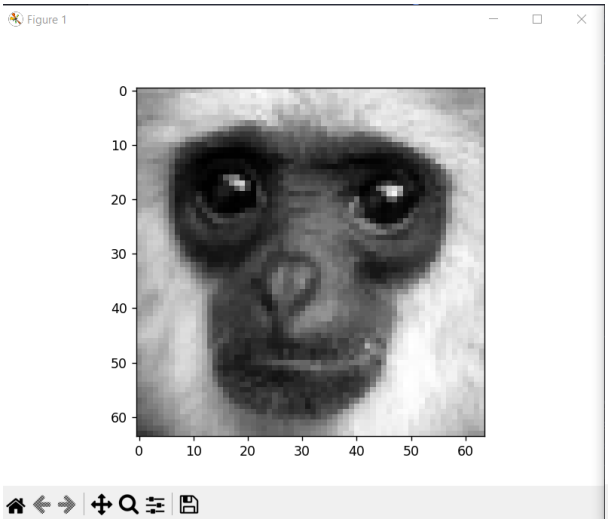
Figure 1
0
10
20
30
40
50
60
0
10
20
30
40
50
60

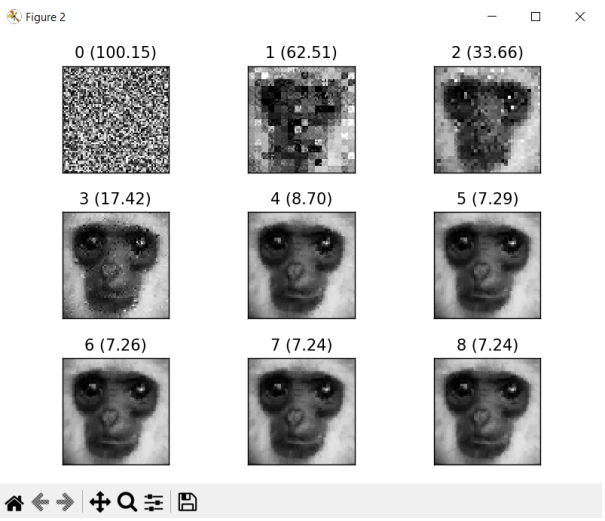
Figure 2
0 (100.15)
1 (62.51)
2 (33.66)
3 (17.42)
4 (8.70)
5 (7.29)
6 (7.26)
7 (7.24)
8 (7.24)