CSci 4270 and 6270
Computational Vision,
Spring Semester, 2022
Homework 1
Due: Tuesday, January 25 at 11:59:59 pm EST

Homework Guidelines

Your homework submissions will be graded on the following criteria:

- correctness of your solution,
- clarity of your code, including:
  - clear and easy-to-follow logic
  - concise, meaningful comments
  - good use of whitespace (indentations and blank lines)
  - self-documenting variable names
  - when needed effective use of functions and/or classes
  - See the PEP 8 Style Guide for more info:
    https://www.python.org/dev/peps/pep-0008/
- quality of your output,
- conciseness and clarity of your explanations,
- where appropriate, computational efficiency of your algorithms and your implementations.

Explanations, when requested, are extremely important. Image data is highly variable and unpredictable. Most algorithms you implement and test will work well on some images and poorly on others. Finding the breaking points of algorithms and evaluating their causes is an important part of understanding image analysis and computer vision.

You must learn to use Python, NumPy and OpenCV effectively. This implies that you will need to work on the tutorials posted on the Submitty site before starting on this assignment. Of particular note, **you should not be writing solutions for this or future assignments that explicitly iterate over each pixel** in a large image, unless otherwise noted.

Submission Guidelines

Your solutions **must be** uploaded to Submitty. Instructions will be posted on the course Submitty site soon. Two things will be extremely important to make the submission and grading processes smooth:

1. Run the programs with command lines **exactly** as specified in the problem descriptions.
2. Make your output **match** our example output as closely as possible.

We will be providing sample data and output several days before the assignment is due, but we will not provide all test cases that we run on Submitty.

Integrity Issues

Two important items:

1. You are free to use without attribution any and all code that I have written for class and posted on Submitty. **Use of my code will not be considered an academic integrity violation.**

2. We will be comparing your submissions to each other and — where problems are repeated — to submissions from previous semesters. Make sure the code you submit is entirely your own!

Problems

Since this is the starting homework, there is no extra grad-credit problem. If you have no prior experience with NumPy, please work on the tutorials we suggested before starting.

1. **(20 points)** Write a script that takes a single image and creates a checkerboard pattern from it. The command-line will look like

   ```
   python p1_checkerboard.py im out_im m n
   ```

   Input image `im` should be cropped to make it square and resized to make it $m \times m$. Next, it should be formed into a 2x2 grid of $m \times m$ images. The 0,0 entry for the grid should show the downsized image, and the 1,1, entry for the grid should show the image upside down. Then the 0,1 entry should show the 0,0 image with the colors of the image inverted so that each color intensity value p is replaced by $255 - p$, and the 1,0 entry should show the 1,1 entry with the colors inverted. Finally, replicate the 2x2 grid of images to make it $2n \times 2n$, generating a final image having $2nm \times 2nm$ pixels. Save the result to `out_im`. Use NumPy functions `concatenate` and `tile` to create the final image. See discussion of `np.tile` below.

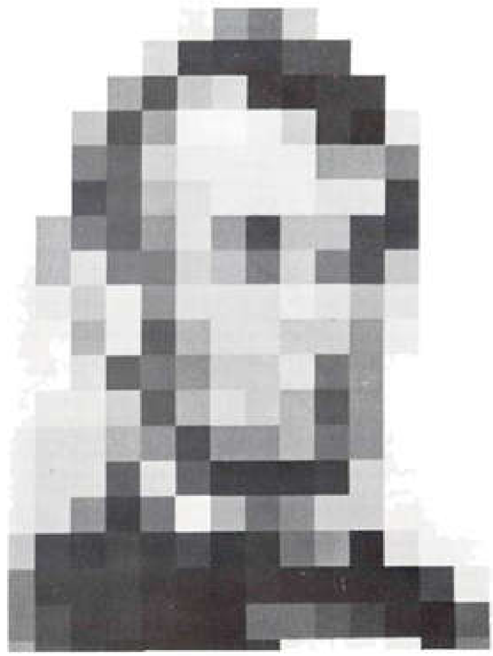
   Here is an example command line

   ```
   python p1_checkerboard.py mountain3.jpg p1_mountain3_checker_out.jpg 120 4
   ```

   and desired output

   ```
   Image mountain3.jpg cropped at (0, 420) and (1079, 1499)
   Resized from (1080, 1080, 3) to (120, 120, 3)
   The checkerboard with dimensions 960 X 960 was output to p1_mountain3_checker_out.jpg
   ```

2. (**20 points**) Do you recognize Abraham Lincoln in this picture?



If you don't you might be able to if you squint or look from far away. Try it now. In this problem you will write a script to generate such a blocky, scaled-down image. The idea is to form the block image from the input image, which you will read as a grayscale: Do this in two steps:

(a) Compute a "downsized image" where each pixel represents the average intensity across a region of the input image.

(b) Generate the larger block image by expanding each pixel in the downsized image to a block of pixels having the same intensity.

(c) Generate a binary image version of the downsized image and make a block version of it as well.

The input to your script will be an image and three integers:

```
python p2_block.py img m n b
```

The values m and n are the number of rows and columns, respectively, in the downsized image, while b is the size of the blocks that replace each downsized pixel. The resulting image should have mb rows and nb columns.

When creating the downsized image, start by generating two scale factors, s_m and s_n. If the input image has M rows and N columns, then we have $s_m = M/m$ and $s_n = N/n$. (Notice that these will be float values.) The pixel value at each location (i, j) of the downsized image will be the (float) average intensity of the region from the original gray scale image whose row values include round$(i * s_m)$ up to (but not including) round$((i + 1) * s_m)$ and whose column values include round$(j * s_n)$ up to (but not including) round$((j + 1) * s_n)$.

You will then create a second downsized image that will be a binary version of the first downsized image. The threshold for the image will be decided such that half the pixels are

0's and half the pixels are 255. More precisely, any pixel whose value (in the downsized image) is greater than or equal to the median value (NumPy has a `median` function) should be 255 and anything else should be 0. Note that this means the **averages should be kept as floating point values before before forming the binary image**.

Once you have created both of these downsized images, you can easily upsample them to create the block images. Before doing this, convert the average gray scale image to integer by **rounding**.

The gray scale block image should be output to a file whose name is the same as the input file, but with `_g` appended to the name just before the file extension. The binary block image should be output to a file whose name is the same as the input file, but with `_b` appended to the name just before the file extension.

Text output should include the following:

- The size of the downsized images.
- The size of the block images.
- The average output intensity (as float values accurate to two decimals) at the following downsized pixel locations:
  - (m // 4, n // 4)
  - (m // 4, 3n // 4)
  - (3m // 4, n // 4)
  - (3m // 4, 3n // 4)
- The threshold for the binary image output, accurate to two decimals.
- The names of the output images.

Here is an example.

```
python p2_block.py lincoln1.jpg 25 18 15
```

which produces the output

```
Downsized images are (25, 18)
Block images are (375, 270)
Average intensity at (6, 4) is 59.21
Average intensity at (6, 13) is 55.46
Average intensity at (18, 4) is 158.30
Average intensity at (18, 13) is 35.33
Binary threshold: 134.68
Wrote image lincoln1_g.jpg
Wrote image lincoln1_b.jpg
```

Important Notes:

(a) To be sure you are consistent with our output, convert the input image to grayscale as you read it using `cv2.imread`, i.e.

```
im = cv2.imread(sys.argv[1], cv2.IMREAD_GRAYSCALE)
```

(b) You are **only** allowed to use `for` loops over the pixel indices of the downsized images (i.e. the 25x18 pixel image in the above example). In addition, avoid using for loops when converting to a binary image.

(c) Be careful with the types of the values stored in your image arrays. Internal computations should use `np.float32` or `np.float64` whereas output images should use `np.uint8`.

3. (**20 points**) Image manipulation software tools include methods of introducing shading in images, for example, darkening from the left or right, top or bottom, or even from the center. Examples are shown in the following figure, where the image darkens as we look from left to right in the first example and the image darkens as we look from the center to the sides or corners of the image in the second example.

The problem here is to take an input image I, create a shaded image I_s, and output the input image and its shaded version (I and I_s) side-by-side in a single image file. Supposing I has M rows and N columns, the central issue is to form an $M \times N$ array of multipliers with values in the range $[0, 1]$ and multiply this by each channel of I. For example, values scaling from 0 in column 0 to 1 in column $N-1$, with $i/(N-1)$ in column i, produce an image that is dark on the left and bright on the right (opposite the first example above). This $M \times N$ array is called an *alpha mask*, or *mask*.

Write a Python program that accomplishes this. The command-line should run as

```
python p3_shade.py in_img out_img dir
```

where `dir` can take on one of five values, `left, top, right, bottom, center`. (If `dir` is not one of these values, do nothing. We will not test this case.) The value of `dir` indicates the side or corner of the image where the shading starts. In all cases the value of the multiplier should be proportional to $1 - d(r, c)$, where $d(r, c)$ is the distance from pixel (r, c) to the start of the shading, normalized so that the maximum distance is 1. For example, if the image is 7×5 and `dir == 'right'` then the multipliers should be

```
[[ 0.  ,  0.25,  0.5 ,  0.75,  1.  ],
 [ 0.  ,  0.25,  0.5 ,  0.75,  1.  ],
 [ 0.  ,  0.25,  0.5 ,  0.75,  1.  ],
```

```
 [ 0.  ,  0.25,  0.5 ,  0.75,  1.  ],
 [ 0.  ,  0.25,  0.5 ,  0.75,  1.  ],
 [ 0.  ,  0.25,  0.5 ,  0.75,  1.  ],
 [ 0.  ,  0.25,  0.5 ,  0.75,  1.  ]])
```

whereas if the image is 5×7 and `dir == 'center'` then the multipliers should be

```
[[0.    0.216 0.38  0.445 0.38  0.216 0.   ]
 [0.123 0.38  0.608 0.723 0.608 0.38  0.123]
 [0.168 0.445 0.723 1.    0.723 0.445 0.168]
 [0.123 0.38  0.608 0.723 0.608 0.38  0.123]
 [0.    0.216 0.38  0.445 0.38  0.216 0.   ]]
```

(I used `np.set_printoptions(precision = 3)` to generate this formatting.) In addition to outputing the final image (the combination of original and shaded images), the program should output, accurate to three decimal places, nine values of the multiplier. These are at the Cartesian product of rows $(0, M//2, M-1)$ and columns $(0, N//2, N-1)$ (where `//` indicates integer division). For example, my solution's output for image `mountain2.jpg` with $M = 1080$ and $N = 1920$ and direction `'center'` is

```
(0,0) 0.000
(0,960) 0.510
(0,1919) 0.001
(540,0) 0.128
(540,960) 1.000
(540,1919) 0.129
(1079,0) 0.000
(1079,960) 0.511
(1079,1919) 0.001
```

These values are the only printed output required from your program.

Important Notes:

(a) Start by generating a 2d array of pixel distances in the row dimension and a second 2d array of pixel distances in the column dimension, then combine these using NumPy operators and universal functions, ending with normalization so that the maximum distance is 1. The generation of distance arrays starts with `np.arange` to create one distance dimension and then extends it to two dimensions `np.tile`. For example,

```
>>> import numpy as np
>>> a = np.arange(5)
>>> np.tile(a, (3,1))
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])
```

After you have the distance array, simply subtract the array from 1 to get the multipliers.

(b) Please do not use `np.fromfunction` to generate the multiplier array because it is essentially the same as nested for loops over the image with a Python call at each location.

(c) Please use (M // 2, N // 2) as the center pixel of the image.

4. **(20 points)** How do you decide how similar two images are to each other? This question is at the heart of the recognition problem that pervades computer vision, and therefore it has been studied for years. Here we will consider a simple method that is a precursor to more sophisticated methods we will see later in the semester.

   Your script will read in each image in a directory. It will reduce each image to a vector of length $3n^2$. It will then find the distance between each pair of images. For each image, in the order produced by `sort`, it must find the closest image, and then it must output the two images and the distance, accurate to 3 decimal places.

   To encode an image in a vector — often called a *descriptor vector* — we divide the image into $n \times n$ regions that are equal in size (perhaps differing by one pixel). Use the same method as you did in Problem 2 when creating the downsized image. In each region, compute the average red, green and blue intensities. Concatenate these in row-major order to form the descriptor. In other words, if $r_{i,j}$, $g_{i,j}$, $b_{i,j}$ are the average RGB values from region i, j (here i represents rows and j represents columns), then the vector should be formed as

$$r_{0,0}, g_{0,0}, b_{0,0}, r_{0,1}, g_{0,1}, b_{0,1}, \ldots r_{0,n-1}, g_{0,n-1}, b_{0,n-1}, r_{1,0}, g_{1,0}, b_{1,0}, \ldots$$

   Finally, normalize this vector (use `np.linalg.norm`) so that its magnitude is 1.0, and then scale all values by 100. The normalization step is intended to correct for brightness differences between images, while the 100 scaling converts to percentages to make the values more intuitive. Call the result the *RGB descriptor*. Output the final values of $r_{0,0}, g_{0,0}, b_{0,0}$ and $r_{n-1,n-1}, g_{n-1,n-1}, b_{n-1,n-1}$ for the first image.

   The command line for your program should be

```
python p4_closest.py img-folder n
```

   where `img-folder` is the file folder containing the images (only consider files whose lower-case extension is `.jpg`) and `n` is the number of regions in the row and column dimensions. Each time the program is run, it should use both the RGB and the L*a*b descriptors, generating two sets of output. All numerical output should be accurate to 2 decimal points. Here is an example based on four images that will be distributed with the assignment.

```
Nearest distances
First region: 20.281 21.207 22.185
Last region: 6.762 6.497 6.520
central_park.jpg to skyline.jpg: 21.65
hop.jpg to times_square.jpg: 24.17
skyline.jpg to central_park.jpg: 21.65
times_square.jpg to hop.jpg: 24.17
```

   In this example, there is symmetry in the closest distances. This will not always be the case.