CSci 4270 and 6270
Computational Vision,
Fall Semester, 2019-20
Homework 1
Due: Friday, September 13, 5 pm

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
  - 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, the department's open source homework submission and grading server. 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:

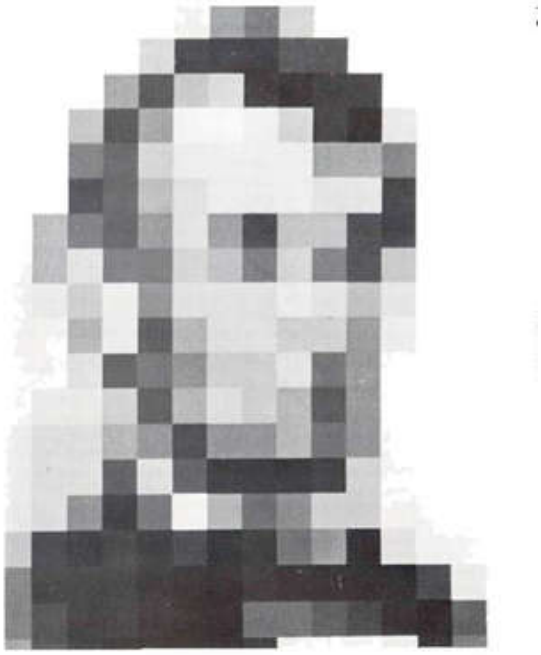
1. You are free to use without attribution any and all code that I have written for class and posted on Submitty. **Use of my code will not be considered an academic integrity violation.**

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 "warm-up" homework, there is no extra grad-credit problem. If you have no prior experience with NumPy, please work on the tutorials we suggested before then.

1. (**25 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. In this problem you will write a script to generate such a blocky, scaled-down image. The idea is to form the block image in two stages:

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

(b) Generate the block image by replacing each pixel in the downsized image with a block of pixels having the same intensity.

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

```
python p1_block 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.) You will actually create two downsized images:

(a) For the first downsized image, the pixel value at location (i, j), where i is the row dimension and j is the column dimension, will be the (float) average intensity of the region from the original gray scale image whose row values include $\lfloor i * s_m \rfloor$ through and $\lfloor (i+1) * s_m - 1 \rfloor$ and whose column values include $\lfloor j * s_n \rfloor$ through $\lfloor (j+1) * s_n - 1 \rfloor$. The $\lfloor x \rfloor$ operation is the floor operation, coercing down to the nearest integer.

(b) The second downsized image will be a binary version of the first 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 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 average gray scale 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 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:
  - $(m \,//\, 3,\ n \,//\, 3)$
  - $(m \,//\, 3,\ 2n \,//\, 3)$
  - $(2m \,//\, 3,\ n \,//\, 3)$
  - $(2m \,//\, 3,\ 2n \,//\, 3)$
- The threshold for the binary image output, accurate to two decimals.
- The names of the output images.

Here is an example.

```
python p1_block.py lincoln1.jpg 25 18 15
```

produces the output

```
Downsized images are (25, 18)
Block images are (375, 270)
Average intensity at (8, 6) is 113.80
Average intensity at (8, 12) is 80.63
Average intensity at (16, 6) is 96.51
Average intensity at (16, 12) is 92.61
Binary threshold: 134.62
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`.

(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, please try to 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`.

2. (**25 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 p2_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 5×7 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 `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 an image with $M = 1080$ and $N = 1920$ and directon `'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]])
```

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

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

3. **(25 points)** How do you decide which images are the most similar to each other? Many ways to do this have been proposed. We will explore just two: the distance between two images' average color vectors, and the distance between their histograms. You will be given a directory of images. For each of the two similarity measures, your script must output the names of the two closest images from the directory and the names of the two images that are furthest apart. Include their distances, accurate to three decimal places.

   To be specific, for the i-th image, let $\mathbf{a}_i$ be its three-component average color vector. Then the first distance between two images is

$$\|\mathbf{a}_i - \mathbf{a}_j\|^2.$$

   For the histogram distance, we use a three-dimensional color histogram, but not at the full $256 \times 256 \times 256$ resolution. Instead each bin of the histogram will cover $S \times S \times S$ colors, where S is a parameter. (If $S = 1$ we get the full resolution histogram.) Entry (r, g, b) in the histogram will count the *fraction* of image pixels whose red value falls into the range $[r \cdot S, (r+1) \cdot S)$, green value falls into the range $[g \cdot S, (g+1) \cdot S)$ and blue value falls into the range $[b \cdot S, (b+1) \cdot S)$. There will be $m = 256/S$ entries along each dimension and $m \cdot m \cdot m$ entries overall. Please use $S = 32$, so that $m = 8$. Note that we increase the size of each bin so that minor variations in intensity do not affect the measure.

   Let h_i be the histogram for image I_j, then the distance between the histograms for two images I_i and I_j is

$$\left[\sum_{r=0}^{m-1}\sum_{g=0}^{m-1}\sum_{b=0}^{m-1}(h_i(r,g,b) - h_j(r,g,b))^2\right]^{1/2}.$$

   There are histogram functions in both NumPy and OpenCV. Here's a bit on how to use NumPy's `histogramdd` function. If `img` is a color image, then

```
reds = np.ravel(img[:,:,0])
greens = np.ravel(img[:,:,1])
blues = np.ravel(img[:,:,2]
```

   gets you one-dimensional arrays of the red, green and blue channel pixels separately. If you then call

```
hist, endpoints = np.histogramdd( [reds, greens, blues )
```

then `hist` will be your desired histogram, and `endpoints` will be the bounds on the entries in each bin. You'll need to study the optional settings to the `histogramdd` function call to get the parameters right.

The command-line for this script will be simply

```
python p3_distance.py image_folder
```

where `image_folder` is the path to a folder of images that are all the same size.

Here is an example of running the solution on the folder `four_images` that we are providing:

```
Using distance between histograms.
Closest pair is (central_park.jpg, hop.jpg)
Minimum distance is 0.182
Furthest pair is (hop.jpg, skyline.jpg)
Maximmum distance is 0.281
```

4. **(25 points)** Given two images, not necessarily the same size, your problem is to create a checkerboard image from reduced resolution versions of these images, where the first image forms the "white" squares and the second image forms the "black" squares. The checkerboard will have M rows and N columns of squares and each square will be s pixels on a side. **You may assume that M and N are both even positive integers.** In the top row, the first image will form square 0, the second image will form square 1, the first image will form square 2, the second image will form square 3, etc. In the next row, this alternating sequence will start with the second image in square 0. The final image will have Ms rows and Ns columns.

   The command-line will look be

   ```
   python p4_checkerboard img1 img2 out_img M N s
   ```

   (Sorry for the large number of arguments.)

   The process you need to follow will look something like this:

   (a) Crop the images so that they are square:
   - For each image, crop the larger dimension to preserve the center section of the image.
   - For example, if the input image is 4000×6000 then the cropped image will be 4000×4000 with columns 0 through 999 cropped out on the left and columns 5000 through 5999 cropped out on the right.

   (b) Resize the images to be $s \times s$

   (c) Form the checkerboard using NumPy's `concatenate` and `tile` functions.

   Diagnostic output from this script should include, for each image, (1) information about the cropping, giving the upper left and lower right pixels where the cropping is occurring, and (2) information about resizing, giving the dimensions before resizing (but after cropping) and the dimensions after resizing. The output should also give information about the size of the final output image. Use the following example to guide you:

   ```
   python p4_checkerboard.py central_park.jpg hop.jpg out.jpg 4 6 150
   ```

```
Image central_park.jpg cropped at (0,200) and (1099,1299)
Resized from (1100, 1100, 3) to (150, 150, 3)
Image hop.jpg cropped at (0,640) and (2559,3199)
Resized from (2560, 2560, 3) to (150, 150, 3)
The checkerboard with dimensions 600 X 900 was output to out.jpg
```