CSci 4270 and 6270
Computational Vision,
Fall Semester, 2018-19
Homework 1
Due: Friday, September 14, 5 pm

Homework Guidelines

Your homework submissions will be graded on the following criteria:

- correctness of your solution,
- clarity of your code,
- quality of your output,
- conciseness and clarity of your explanations,
- where appropriate, computational efficiency of your algorithms and your implementations.

Clarity of code includes the usual properties of good code: clear and easy-to-follow logic; concise, meaningful comments; good use of indentation, spacing, variable naming, and blank lines to make the functions easy to read. See the Python PEP 8 style guide.

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 Piazza 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.

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 Piazza 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 the Piazza site.

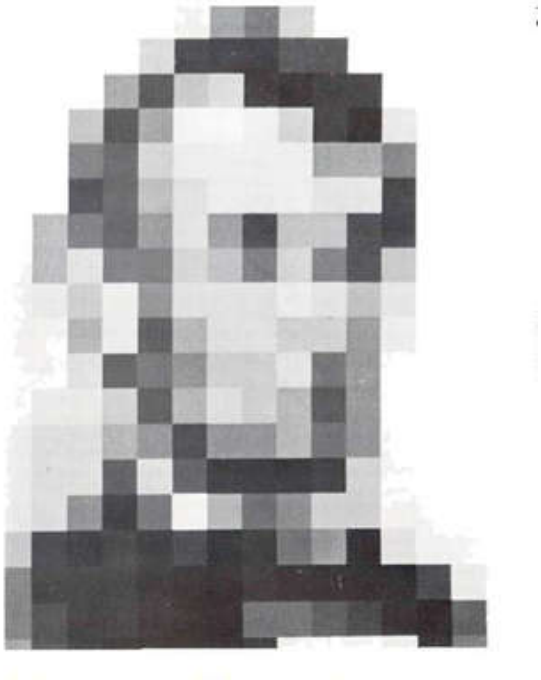
Use of Code from Lecture

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

Problems

Since this is the "warm-up" homework, there is no extra grad-credit problem. If you have no prior experience with

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 are bounded by $\lfloor i * s_m \rfloor$ and

$\lfloor (i+1) * s_m - 1 \rfloor$ and whose column values are bounded by $\lfloor j * s_n \rfloor$ and $\lfloor (j+1) * s_n - 1 \rfloor$, *inclusive*.

(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 at downsized pixels $(m//4, n//4)$, $(m//4, 3n//4)$, $(3m//4, n//4)$, $(3m//4, 3n//4)$, as float values accurate to two decimals.
- 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 20
```

produces the output

```
Downsized images are (25, 18)
Block images are (500, 360)
Average intensity at (6, 4) is 58.97
Average intensity at (6, 13) is 55.55
Average intensity at (18, 4) is 158.26
Average intensity at (18, 13) is 35.05
Binary threshold: 134.6
Wrote image lincoln1_g.jpg
Wrote image lincoln1_b.jpg
```

A few 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. Try to avoid using for loops, however, when converting to a binary image. (If someone figures out an efficient way to avoid these `for` loops altogether I'd like to know.)

(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, create a shaded image, and output the input image and its shaded version side-by-side in a single image file. Supposing the input image, 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).

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.

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,

```
>>> np.tile( a, (3,1) )
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])
```

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.

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

3. **(50 points)** The goal of this problem is to pick two images from a folder of images and create a checkerboard image from them, with one image forming the "white" squares and the other forming 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.

Start by writing a function that takes two color images (3d NumPy arrays) and the values of M, N and s and returns the checkerboard image, which will have Ms rows of pixels and Ns columns of pixels. The first step will be to crop the images so that they are square. For each image, the cropping of the larger dimension should 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. The next step will be to resize the images to be $s \times s$. Finally, form the checkerboard. Do this using NumPy's `concatenate` and `tile` functions. The function should return the resulting image. Diagnostic output from this function should be information about cropping, giving the upper left and lower right pixels where the cropping is occurring, and then information about resizing. Specifically, if the first image is 4000×6000 and the second image is 6000×4000 the output would be

```
Image cropped at (0,1000) and (3999,4999)
Resized from (4000, 4000, 3) to (150, 150, 3)
Image cropped at (1000,0) and (4999,3999)
Resized from (4000, 4000, 3) to (150, 150, 3)
```

If the image is already square and therefore does not need to be cropped, please output a message to this effect. For example,

```
Image does not require cropping
```

If the image is already the correct size, output

```
No resizing needed
```

The second major issue — which actually comes before the first during the execution of your program — is to decide which images from a folder (directory) of images to use to form the checkerboard. The name of this folder will be provided on the command line. Only consider images having the extension `'.jpg'` in some combination of capital and small letters.

Please handle the following four cases:

(a) If there are no images your program should output

```
No images. Creating an ordinary checkerboard.
```

then generate an image of all 255's and an image of all 0's and pass these to the checkerboard creation function.

(b) If there is exactly one image, open this image and make it the image that goes into to the "white" square (i.e. (upper left square) and make the other image all black. The output should be

```
One image: img.  It will form the white square.
```

where `img` is the name of the image file.

(c) If there are exactly two images, open these, output the line

```
Exactly two images: img1 and img2. Creating a checkerboard from them.
```

(where `img1` and `img2` are the image file names) and pass the two images to the checkerboard function.

(d) If there are more than two images, the issue becomes which two images should the program choose. Since we are talking about a checkerboard, it seems natural that the images should be as distinct from each other as possible. There are many possible ways to decide this, but here's a simple one I'd like you to implement: For each image, form a three-component vector of the mean red, green and blue intensities. Now find the two images for which these vectors are furthest apart. These are the two that should form the checkerboard. Of these, the image with the highest magnitude of the vector should be first in the checkerboard. (You are welcome to use nested for loops over the sequence of mean vectors extracted from the images to decide which two are furthest apart.)

For this more-than-two-images case, before calling the checkerboard creation function, output some diagnostic information. Print the name of each image (in the order produced by `os.listdir`) and the vector of three mean values (accurate to one decimal), e.g.

```
example_pic.jpg (45.3, 127.5, 89.1)
```

Then output the names of the two images chosen, along with the distance between them, again accurate to a single decimal point. Output the one with the highest magnitude vector first. For example, your output might be

```
Checkerboard from img1 and img2.  Distance between them is 187.3
```

where `img1` and `img2` are the input file names.

Finally, the command-line will

```
python p3_checkerboard.py directory out_img M N s
```

where `directory` is the path to the folder containing the images, `out_img` is the name of the output image, and `M`, `N` and `s` are the row, column and square values described above. Be careful to write `out_img` to the original working directory rather than the directory containing the input images.