CSci 4270 and 6270
Computational Vision,
Spring Semester, 2025
Homework 1
Due: Friday, January 17 at 11:59:59 pm EST

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 study (and manipulate) the Lecture 02 examples and work on tutorials and practice problems 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

All of your solutions **must be** combined in a single Jupyter notebook and uploaded to the course Submitty site. Here are some important notes:

- A starting notebook containing testing code is provided. **Please add your solutions to the notebook, but do NOT modify the cells containing the testing code!**
- **Example input and output for each problem are in the starting notebook! Save a copy before starting work so that you can refer back to it!**
- The images referenced in the testing code are provided as a folder with this assignment. This folder must be stored in the folder that contains your solution notebook. Do NOT modify the relative paths to the files specified in the notebooks.

- Please be sure to run each of the indicated tests and include the results in your notebook before submission. We will rerun some of the tests.
- We have provided example output below and sample solution images as well. Please try to match our output as closely as possible.

Reminder About Integrity Issues

Two important reminders:

1. You are free to use any and all code that I have written for class and posted on Submitty, but please also provide appropriate attribution (as a matter of good professional and scientific practice). **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

For reference, each of my solutions to the these problems is less than 40 lines of Python.

1. **(25 points)** Write a function that takes a single image and creates a checkerboard pattern from it. As you will see in the template notebook, this will require three parameters to be set: the (relative) image path, and the values of two integers m and n. Input image `im` should be cropped to make it square (the cropping should be centered) and the cropped version should be resized to make it $m \times m$. Call the resulting image `im'`. Next, `im'` should be should be formed into a $2m \times 2m$ image. The upper left quadrant of this image should contain `im'`. The upper right quadrant of this image should contain `im'` rotated clockwise by 90 degrees. The lower right quadrant should contain `im'` upside down and the lower left quadrant should contain `im'` rotated counterclockwise by 90. (You'll need to use combinations of `np.transpose` and coordinate reversals to accomplish this.) Finally, thinking of this as a 2x2 grid of images, replicate it to make a $2n \times 2n$ grid, generating a final image having $2nm \times 2nm$ pixels.

   As output, print the actual shape of the final image, and display the final image using `pyplot`.

   Rubric:

   - 9 points: the 2x2 grid correct (watch for 180 degree rotations instead of flips)
   - 6 points: the final image is correct
   - 4 points: the final shape is correct and output
   - 6 points: code and (brief) documentation

   Penalties:

   - Up to -8 for use of for loops

2. (**25 points**) Vignetting in an image is a systemic reduction in the brightness of pixels as their distance from the center increases. Often this is thought of as a sign of low quality

of a camera's lens system, but vignetting may be created as an artificial effect in photo manipulation software. That is what we are going to do in this problem.

The important variables are the image, and r_0 and c_0, the row and column position of the center of the vignetting (which may not be exactly the center of the image).

(a) Your first step is to calculate the maximum distance for any pixel from r_0, c_0. Suppose M and N are the number of rows and columns, then the maximum distance is

$$d_m = \sqrt{\max(r_0, M-1-r_0)^2 + \max(c_0, N-1-c_0)^2}$$

(b) Next create a 2d array that is the size of the image ($M \times N$) where the value at location (i, j) is the pixel distance to (r_0, c_0), specifically

$$D_{i,j} = \sqrt{(i-r_0)^2 + (j-c_0)^2}.$$

Importantly, this can be done entirely with array programming and without explicit for loops.

(c) Create a weight array, also of size $M \times N$:

$$W_{i,j} = 1 - \sqrt{D_{i,j}/d_m},$$

or, more compactly,

$$\mathbf{W} = 1 - \sqrt{\mathbf{D}/d_m}.$$

(d) Finally, multiply the input image by $\mathbf{W}$ to create the output vignetted image. Assuming the input image is color (three dimensions), you will need to expand $\mathbf{W}$ to three-dimensions as well before the multiplication. To do this, look up `np.expand_dims`.

(e) Display the final resulting "vignetted" image in your notebook.

Text output should include the following:

- The value of d_m, accurate to one decimal place
- The values of $\mathbf{W}$, accurate to two decimal places, at the following pixel locations
  - $(M \,//\, 4,\ N \,//\, 4)$
  - $(M \,//\, 4,\ 3N \,//\, 4)$
  - $(3M \,//\, 4,\ N \,//\, 4)$
  - $(3M \,//\, 4,\ 3N \,//\, 4)$
- The final, vignetted image pixel values (RGB) at the same four locations.

As a side note: there are many other ways to model the vignetting weight as a function of the image distance. This problem has explored a simple one.

Rubric:

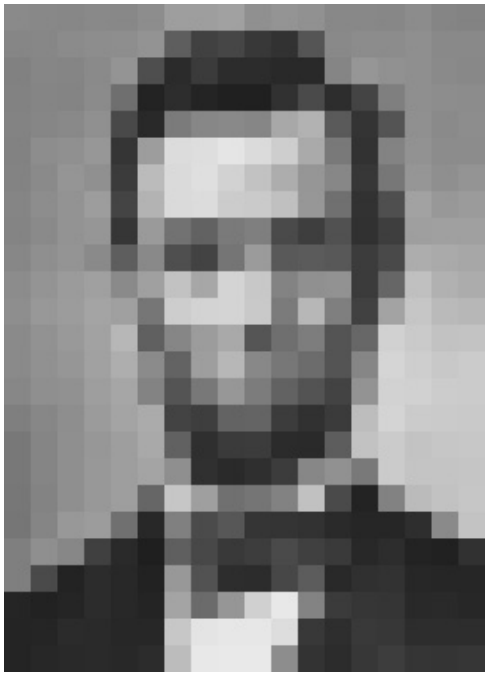
- 3 points: value of d_m is correct
- 5 points: good method calculating D
- 4 points: values of W are correct
- 4 points: values of I are correct

- 4 points: final image is output correctly
- 5 points: code and (brief) documentation

Penalties:

- Up to -8 for use of for loops

3. (**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 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 important variables for doing this are the image file name and three integers, m, n and 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 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**.

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 one decimal place) 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 one decimal place.

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(fname, 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`.

Rubric:

- 3 points: downsized and upsampled images are the right size
- 6 points: average intensities are correct
- 3 points: binary threshold is correct
- 4 points: gray scale image looks right
- 4 points: binary image looks right
- 5 points: code and (brief) documentation

Penalties:

- Up to -8 for use of for loops