**CSci 4270 and 6270**
**Computational Vision,**
**Fall Semester, 2017-18**
**Homework 2**
**Due: Tuesday October 10, 5 pm**

Please refer back to HW 0 for guidelines.

This homework is divided into two parts. The first part consists of programming problems, two relatively short and one somewhat long. The second consists of analytical / mathematical problems. You must upload your solutions to these to Submitty as a single pdf file. No hand-written solutions please, and LaTeX typesetting is preferred.

## Programming Problems

1. **(25 points)** This problem is about constructing a camera matrix and applying it to project points onto an image plane. The command line of your program should simply be

   ```
   python p1_camera.py params.txt points.txt
   ```

   Here `params.txt` contains parameters that can be used form the $3 \times 4$ camera matrix. Specifically, the following ten floating point values will appear in the file on three lines:

   ```
   rx ry rz
   tx ty tz
   f  d  ic jc
   ```

   Here's what these mean: Relative to the world coordinate system, the camera is rotated by `rz` **degrees** about the z-axis, then `ry` degrees about the y-axis, then `rx` degrees about the x-axis. Then it is translated by vector `(tx,ty,tz)` millimeters. The focal length of the lens is `f` millimeters, and the pixels are square with `d` microns on each side. The image is 4000x6000 (rows and columns) and the optical axis pierces the image plane in row `ic`, column `jc`. Use this to form the camera matrix **M**. In doing so, please explicitly form the three rotations matrices (see lecture notes) and compose them. Overall on this problem, be very, very careful about the meaning of each parameter and its units. My results were obtained by converting length measurements to millimeters.

   Please output the 12 terms in the resulting matrix **M**, with one row per line. All values should be accurate to 2 decimal places. I have provided two examples and in my example I've also printed **R** and **K** – the test cases will not have these.

   Next, apply the camera matrix **M** to determine the image positions of the points in `points.txt`. Each line of this file contains three floating points numbers giving the x, y and z values of a point in the world coordinate system. Compute the image locations of the points and determine if they are inside or outline the image coordinate system. Output six numerical values on each line: the index of the point (first point has index 0), the x, y and z values that you input for the point, and the row, col values. Also output on each line, the decision about whether the point is inside or outside. For example, you might have

   ```
   0: 45.1  67.1 89.1 => 3001.1  239.1 inside
   1: -90.1  291.1 89.1 => -745.7  898.5 outside
   ```

All floating values should be accurate to just one decimal place.

One thing this problem does not address yet is whether the points are in front of or behind the camera, and therefore are or not truly visible. Addressing this requires finding the center of the camera and the direction of the optical axis of the camera. Any point is considered visible if it is in front of the plane defined by the center of projection (the center of the hypothetical lens) and the axis direction. As an example to illustrate, in our simple model that we started with, the center of the camera is at $(0, 0, 0)$ and the direction of the optical axis is the positive $z$-axis (direction vector $(0, 0, 1)$) so any point with $z > 0$ is visible. To test that you have solved this, as a final step, print the indices of the points that are and are not visible, with one line of output for each. For example, you might output

```
visible: 0 3 5 6
hidden: 1 2 4
```

If there are no visible values (or no hidden values), the output should be empty after the word `visible:`. This will be at the end of your output.

2. (**15 points**) You are given a series of images (all in one folder) taken of the same scene, and your problem is to simply determine which image is focused the best. Since defocus blurring is similar to Gaussian smoothing and we know that Gaussian smoothing reduces the magnitude of the image's intensity gradients, our approach is simply to find the image that has the largest average squared gradient magnitude across all images. This is value is closely related to what is referred to as the "energy" of the image. More specifically, this is

$$E(I) = \frac{1}{MN} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} \left( \frac{\partial I}{\partial x}(i,j) \right)^2 + \left( \frac{\partial I}{\partial y}(i,j) \right)^2 .$$

Note that using the **squared** gradient magnitude is important here. In order to ensure consistency across our implementations, use the two OpenCV Sobel kernels to compute the x and y derivatives and then combine into the square gradient magnitude as in the above equation.

The command-line of your program will be

```
python p2_best_focus.py image_dir
```

where `image_dir` is the path to the directory that contains the images to test. Assume all images are JPEGs with the extension `.jpg` (in any combination of capital and small letters). Sort the image names using the python list sort function. Read the images as grayscale using the built-in `cv2.imread`. Then output for each image the average squared gradient magnitude across all pixels. (On each line output just the name of the image and the average squared gradient magnitude, accurate to two decimal places.) Finally output the name of the best-focused image.

3. (**40 points**) Ordinarily, image resize functions, like the one in OpenCV, treat each pixel equally — everything gets reduced or increased by the same amount. In 2007, Avidan and Shamir published a paper called "Seam Carving for Content-Aware Image Resizing" in SIGGRAPH that does the resizing along contours in an image — a "seam" — where there is not a lot of image content. The technique they described is now a standard feature in image manipulation software such at Adobe Photoshop.

Here is an example of an image with a vertical seam drawn on it in red.

A vertical seam in an image contains one pixel per row, and the pixels on the seam are 8-connected between rows, meaning that pixels in adjacent rows in a seam differ by at most one column. Formally a vertical seam in an image with $M$ rows and $N$ columns is the set of pixels

$$\mathbf{s}_r = \{(i, c(i))\}_{i=0}^{M-1} \text{ s.t. } \forall, i|c(i) - c(i-1)| \leq 1. \tag{1}$$

In reading this, think of $i$ as the row, and $c(i)$ is the chosen column in each row. Similarly, a horizontal seam in an image contains one row per column and is defined as the set of pixels

$$\mathbf{s}_c = \{(r(j), j)\}_{j=0}^{N-1} \text{ s.t. } \forall, j|r(j) - r(j-1)| \leq 1. \tag{2}$$

Here, think of $j$ as the column and $r(j)$ as the row for that column.

Once a seam is selected — suppose for now that it is a vertical seam — the pixels on the seam are removed from the image, shifting the pixels that are to the right of the seam over to the left by one. This will create a new image that has $M$ rows and $N-1$ columns. (There are also ways to use this to add pixels to images, but we will not consider this here!) Here is an example after enough vertical seams have been removed to make the image square.

The major question is how to select a seam to remove from an image. This should be the seam that has the least "energy". Energy is defined in our case as the sum of the derivative

magnitudes at each pixel:

$$e[i,j] = \left| \frac{\partial I}{\partial x}(i,j) \right| + \left| \frac{\partial I}{\partial y}(i,j) \right|.$$

for $i \in 1 \ldots M-2$, $j \in 1 \ldots N-2$. (Note that this is neither the gradient magnitude nor the sum-squared gradient magnitude.) The minimum vertical seam is defined as the one that minimizes

$$\sum_{i=0}^{M-1} e[i, c(i)]/M$$

over all possible seams $c(\cdot)$. Finding this seam appears to be a hard task because there is an exponential number of potential seams.

Fortunately, our old friend (from CSCI 2300) dynamic programming comes to the rescue, allowing us to find the best seam in linear time in the number of pixels. To realize this, we need to recursively compute a seam cost function $W[i,j]$ at each pixel, that represents the minimum cost seam that runs through that pixel. Recursively this is defined as

$$\begin{aligned} W[0,j] &= e[0,j] \qquad \forall j \\ W[i,j] &= e[i,j] + \min\big(W[i-1,j-1], W[i-1,j], W[i-1,j+1]\big) \qquad \forall i > 0, \forall j \end{aligned}$$

Even if you don't know dynamic programming, computing $M[i,j]$ is pretty straightforward (except for a few NumPy tricks — see below).

Once you have $M$, you have to trace back through it to find the actual seam. This is also defined recursively. The seam pixels, as defined by the function $c(\cdot)$ from above, are

$$\begin{aligned} c(N-1) &= \operatorname*{argmin}_{1 \le j \le N-1} W[N-1,j] \\ c(i) &= \operatorname*{argmin}_{j \in \{c(i+1)-1, c(i+1), c(i+1)+1\}} W[i,j] \qquad \text{for } i \in N-2 \text{ downto } 0 \end{aligned}$$

In other words, in the last row, the column with the minimum weight (cost) is the end point of the seam. From this end point we trace back up the image, one row at a time, and at each row we choose from the three possible columns that are offset by -1, 0 or +1 from the just-established seam column in the next row.

A couple of quick notes on this.

- You need to be careful not to allow the seam to reach the leftmost or rightmost column. The easiest way to do this is to introduce special case handling of columns 0 and $N-1$ in each row, assigning an absurdly large weight.
- The trickiest part of this from the NumPy perspective is handling the computation of the minimum during the calculation of $W$. While you clearly must explicitly iterate over the rows (when finding the vertical seam), I don't want you iterating over the columns. Instead, use slicing in each row to create a view of the row that is shifted by +1, -1 or 0 and then take the minimum. For example, here is code that determines at each location in an array, if the value at that is greater than the value at either its left or right neighbors.

```
import numpy as np

a = np.random.randint( 0,100, 20 )
```

```
print(a)
is_max = np.zeros_like(a, dtype=np.bool)
left = a[ :-2]
right = a[ 2: ]
center = a[ 1:-1 ]
is_max[ 1:-1 ] = (center > right) * (center > left)
is_max[0] = a[0] > a[1]
is_max[-1] = a[-1] > a[-2]
print("Indices of local maxima in a:", np.where(is_max)[0])

'''
Example output:

[93 61 57 56 49 40 51 85  5 13 28 89 31 56 11 10 60 93 26 86]
Indices of local maxima in a: [ 0  7 11 13 17 19]
'''
```

**Your actual work:** Your program should take an image as input and remove enough rows or columns to make the image square. The command-line should be

```
python p3_seam_carve.py in_img out_img graph_file
```

You must compute the sum of the magnitudes of the x and y derivatives of the image produced by the OpenCV function `Sobel`. For the 0th, 1st and last seam, please print the index of the seam (0, 1, ...), whether the seam is vertical or horizontal, the starting, middle and end pixel locations on the seam (e.g. if there are 10 pixels, output pixels 0, 5 and 9), and the average energy of the seam (accurate to two decimal places). At the end, output the final resized image to `out_img`. Finally, output a PyPlot plot to `graph_file` that shows the seam index along the horizontal axis and the energy of the seam along the vertical axis.

## Written Problems

1. **(15 points)** Give an algebraic proof that a straight line in the world projects onto a straight line in the image. In particular

   (a) Write the parametric equation of a line in three-space.

   (b) Use the simple form of the perspective projection camera from the start of the Lecture 5 notes to project points on the line into the image coordinate system. This will give you equations for the pixel locations $x$ and $y$ in terms of $t$.

   (c) Combine the two equations to remove $t$ and rearrange the result to show that it is in fact a line. You should get the *implicit form* of the line.

   (d) Finally, under what circumstances is the line a point? Show this algebraically.

2. (**15 points**) Evaluate the quality of the results of seam carving on several images. In particular, find images for which the results are good, and find images for which the results are poor. Show these images before and after applying your code, and include the energy graphs. Explain the results both intuitively in what you see in the images and more quantitatively in terms of what you see in the the graphs.

3. **(10 points, Grad Only!)** Suppose you are given three pairs of points in 2d, $(x_i, y_i)^\top, (u_i, v_i)^\top$, $i = 1, 2, 3$. If these points are not collineaer there is a unique 2x3 affine transformation matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix}$$

such that

$$\mathbf{A} \begin{pmatrix} x_i \\ y_i \\ 1 \end{pmatrix} = \begin{pmatrix} u_i \\ v_i \\ 1 \end{pmatrix}.$$

Derive a matrix equation that gives the parameters of $a_{i,j}$ from the three pairs of points, showing the details of your derivation. At some point you will end up with an inverse of either a $3 \times 3$ or a $6 \times 6$ matrix. You do not have to explicitly compute the inverse.

4. (**2 points**) Please estimate the number of hours you spend on this assignment.