CSci 4270 and 6270
Computational Vision,
Fall Semester, 18
Homework 3
Due: Friday, October 11, 5 pm

Please refer back to HW 1 for guidelines.

This homework is divided into two parts. The first part consists of three programming problems. The second consists of a single analytical problem. **You must upload your solutions to the second part as a single pdf - absolutely no hand-written solutions, Word/LibreOffice Docs, etc will be accepted.** The preferred method is LaTeXtypesetting, though this is not strictly required. Two additional notes: (1) This will also be the first assignment that we will test your programs on images that you have not seen. (2) There is no difference between the graduate and undergraduate versions of this assignment.

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 to form the 3×4 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 first 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. (**Note: the rotation about the z axis will be first and therefore it is the right most of the rotation matrices.**) 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**, but *you should not do this in your final submission.*

   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 outside the image coordinate system. Output six numerical values on each line: the index of the point (the first point has index 0), the x, y and z values that you

input for the point, and the row, column values. Also output on each line, the decision about whether the point is inside or outside. (Anything with row value in the interval $[0, 4000]$ and column value in the interval $[0, 6000]$ is considered inside.) 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. (Note: in this case, a point is considered "visible" even if it is not "inside" the image coordinate system.) 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.

To summarize your required output:

(a) Matrix **M** (one row per line, accurate to two decimals)

(b) Index and (x, y, z) position of input point, followed by transformed (r, c) location and whether it's inside the $4,000 \times 6,000$ frame

(c) Visible point indices (sorted ascending)

(d) Hidden point indices (sorted ascending)

2. (**50 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 as 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 pixel locations 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

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

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

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

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 $i \in 1 \ldots M-2$, $j \in 1 \ldots N-2$. 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 $W[i, j]$ is pretty straightforward (except for a few NumPy tricks — see below).

Once you have matrix W, you must 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 few 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]
'''
```

- Recompute the energy matrix e after each seam is removed. Don't worry about the fact that the energy of most pixels will not change.
- The seam should be removed from the color image, but the energy is computed on a gray scale image. This means you will have to convert from color to gray scale before each iteration energy matrix computation and seam removal.
- Convert the image to float immediately after reading it (and before any derivative computation). This will ensure the greatest consistency with our results. In particular, if `fname` stores the name of the image file, use

```
img = cv2.imread(fname).astype(np.float32)
```

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 p2_seam_carve.py img
```

Compute the sum of the absolute values of the x and y derivatives of the image produced by the OpenCV function `Sobel`. or 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 11 pixels, output pixels 0, 5 and 10), and the average energy of the seam (accurate to two decimal places). Finally, output the original image with the first seam drawn on top of image in red and output the final, resized image. If `foo.png` is the input image, the output images should be `foo_seam.png` and `foo_final.png`.

Finally, you may not be able to reproduce the exact output of my code. Do not worry about this too much as long as your energies and seams are close. Especially important is that the final square image looks close.

3. **(40 points)** In class we started to implement an edge detector in the Jupyter notebook `edge_demo.ipynb`, including Gaussian smoothing and the derivative and gradient computations. The code is posted on Submitty.

   In this problem, you will implement the non-maximum suppression step and then a thresholding step, one that is simpler than the thresholding method we discussed in class. Here are the details:

   - For non-maximum suppression, a pixel should be marked as a maximum if its gradient magnitude is greater than or equal to those of its two neighbors along the gradient

direction, one "ahead" of it, and one "behind" it. (Note that by saying "greater than or equal", edges that have ties will be two (or more) pixels wide — not the right solution in general, but good enough for now.) As examples, if the gradient direction at pixel location (x, y) is $\pi/5$ radians (36° degrees) then the ahead neighbor is at pixel $(x+1, y+1)$ and the behind neighbor is at pixel $(x-1, y-1)$, whereas if the gradient direction is $9\pi/10$ (162°) then the ahead neighbor is at pixel $(x-1, y)$ and the behind neighbor is at pixel $(x+1, y)$.

- For thresholding, start from the pixel locations that remain as possible edges after non-maximum suppression and eliminate those having a gradient magnitude of lower than 1.0. Then, for the remaining pixels, compute the mean, μ, and the standard-deviation, s, of the gradient magnitudes. The threshold will be the minimum of $\mu + 0.5s$ and $30/\sigma$, the former value because in most images, most edges are noise, and the latter value to accommodate clean images with no noise. Dividing by σ is because Gaussian smooth reduces the gradient magnitude by a factor of σ.

The command-line should be

```
python p3_edge.py sigma in_img
```

where

- `sigma` is the value of σ used in Gaussian smoothing, and
- `in_img` is the input image.

(I have posted an example on line with `sigma = 2` and `in_img = disk.png`)

The text output from the program will be:

- The number of pixels that remain as possible edges after the non-maximum suppression step.
- The number of pixels that remain as possible edges after the gradient threshold of 1.0 has been applied.
- μ, s and the threshold, each on a separate line and accurate to 2 decimal places.
- The number of pixels that remain after the thresholding step.

Three output images will be generated, with file names created by adding a four character string to the file name prefix of the input image. Examples below assume that the image is named `foo.png`. Here are the three images:

- The gradient directions of all pixels in the image encoded to the following five colors: red $(255, 0, 0)$ for pixels whose gradient direction is primarily east/west; green $(0, 255, 0)$ for pixels whose gradient direction is primarily northwest/southeast; blue $(0, 0, 255)$ for pixels whose gradient direction is primarily north-south; white $(255, 255, 255)$ for pixels whose gradient direction is primarily northeast/southwest; and black $(0, 0, 0)$ for any pixel on the image border (first or last row or column) and for any pixel, regardless of gradient direction, whose gradient magnitudent is below 1.0. The file name should be `foo_dir.png`.
- The gradient magnitude before non-maximum suppression and before thresholding, with the maximum gradient mapping to the intensity 255. The file name should be `foo_grd.png`.

- The gradient magnitude after non-maximum suppression **and** after thresholding, with the maximum gradient mapping to the intensity 255. The file name should be `foo_thr.png`.

Notes:

- Be sure that your image is of type float32 (or float64) **before** Gaussian smoothing.
- At first it will seem a bit challenging — or at least tedious — to convert the initial gradient direction, which is measured in radians in the range $[-\pi, \pi]$, into a decision as to whether the gradient magnitude is primarily west/east, northwest/southest, north/south, or northeast/southwest. For example, the ranges from $[-\pi, -7\pi/8]$, $[-\pi/8, \pi/8]$, and $[7\pi/8, \pi]$ are all east-west. You could write an extended conditional to assign these directions, or you write **one or two expressions**, using Numpy's capabilty for floating-point modular arithmetic, to simultaneously assign 0 to locations that are west/east, 1 to locations that are northwest/southeast, etc. Think about it!
- This problem is a bit harder than previous problems to solve without writing Python for loops that range over the pixels, but good solutions do exist. Full credit will be given for a solution that does not require for loops, while up to 27 of 30 for students in 4270 will be given for a solution that requires for loops. (For students in 6270 this will be 24 out of 30.) In other words, we've provided mild incentive for you to figure out how to work solely within Python (and numpy) without for loops. Examples that have been given in class and even worked through on homework can help. You'll have to consider each direction (somewhat) separately.
- A word of wisdom from the TA: Build and test each component thoroughly before using it in a larger system - I know it's hard to force yourself to move this slowly, but I promise it will make this (and future) problems easier!

Written Problem

1. (**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. What causes poor results? Please explain.