CSci 4270 and 6270
Computational Vision,
Spring Semester, 2025
Homework 2
Due: Friday, January 31, at 11:59:59 pm EST

Overview

This homework has some problems that require written solutions (one for undergrads and two for grads) and the rest require programming. Your written solutions should be submitted as a single pdf. Your programming solutions should be submitted in a single Jupyter notebook, similar to HW 1, starting from the one provided. Combine these two files as a single zip file that you upload to Submitty.

Students can do programming problem 2 based on Lecture 4 and all but the last programming problem based on Lecture 5 (Thursday 1/23). The last programming problem depends on material from class on Monday, 1/27.

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 simplest 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. Note that t will be in the denominator.

   (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. (**10 points** Grad Only) Problem 1 includes an important over-simplification: the perspective projection of a line does not extend infinitely in both directions. Instead, the projection of the line terminates at what is referred to as the "vanishing point", which may or may not appear within the bounds of the image. Using the parametric form of a line in three-space and the simple perspective projection model, find the equation of the vanishing point of the line. Then, show why this point is also the intersection of all lines that are parallel to the original line. Under what conditions is this point non-finite? Give a geometric interpretation of these conditions.

Programming Problems

1. **(30 points)** This problem is about constructing a camera matrix and applying it to project points onto an image plane. The data is provided in two Python lists of lists specified in the Jupyter notebook. The first list contains parameters that can be used to form the 3×4 camera matrix. Specifically, the following ten floating point values will be given, with each line formed into a sublist

```
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 05 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 careful about the meaning of each parameter and its units. The posted example 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. (In the provided example output I have also printed **R** and **K**, but *you do not need to 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.

So far, this problem does not address 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 one decimal place)

(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. (**25 points**) Implement the RANSAC algorithm for fitting a line to a set of points. The notebook provides you with the following values to get started

```
fn = 'points.txt'
samples = <int>
tau = <float>
seed = <int>
```

where `points.txt` is a text file containing the x, y coordinates of one points per line, `samples` is a positive integer indicating the number of random pairs of two points to generate, `tau` is the bound on the distance from a point to a line for a point, and `seed` is the seed to the random number generator.

After reading the input, your first call to a NumPy function must be

```
np.random.seed(seed)
```

Doing this will allow us to create consistent output.

For each of `samples` iterations of your outer loop you must make the call

```
sample = np.random.randint(0, N, 2)
```

to generate two random indices into the points. If the two indices are equal, skip the rest of the loop iteration (it still counts as one of the samples though). Otherwise, generate the line and run the rest of the inner loop of RANSAC. Each time you get a new best line estimate according to the RANSAC criteria, print out the following values, one per line, with a blank line afterward:

- the sample number (from 0 up to but not including `samples`)
- the indices into the point array (in the order provided by `randint`),
- the values of a, b, c for the line (ensure $c \leq 0$ and $a^2 + b^2 = 1$) accurate to three decimal places, and
- the number of inliers.

At the end, output a few final statistics on the best fitting line, in particular output the average distances of the inliers and the outliers from the line. Keep all of your output floating point values accurate to two decimal places.

In the interest of saving you some work, I've not asked you to generate any plots for this assignment, but it would not hurt for you to do so just to show yourself that things are working ok. For similar reasons, no least-squares fit is required at the end.

3. (**15 points — Students in 4270 ONLY**) 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 value is closely related to what is referred to as the "energy" of the image. More specifically, this is

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

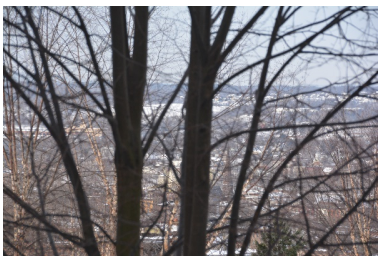
Note that using the **squared** gradient magnitude is important here. In order to ensure consistency across our implementations, use the two OpenCV Sobel kernels to compute the x and y derivatives and then combine into the square gradient magnitude as in the above equation. Specifically, the calls to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

```
im_dx = cv2.Sobel(im, cv2.CV_32F, 1, 0)
im_dy = cv2.Sobel(im, cv2.CV_32F, 0, 1)
```

The only parameter controlling your code is `image_dir`, 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), and all images have the same dimensions. 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 just one decimal place.) Finally output the name of the best-focused image.

4. (**30 points — 6270 ONLY**) You are given a series of images (all the images in one folder) taken of the same scene but with different objects in focus in different images. In some images, the foregound objects are in focus, in others objects in the middle are in focus, and in still others, objects far away are in focus. Here are three examples from one such series:





Your goal in this problem is to use these images to create a single composite image where everything is as well focused as possible.

The key idea is to note that the blurring you see in defocused image regions is similar to Gaussian smoothing, and we know that Gaussian smoothing reduces the magnitude of the intensity gradients. Therefore, if we look at the weighted average of the intensity gradients

in the neighborhood of a pixel, we can get a measure of image energy of the pixel. Higher energy implies better focus.

The equation for this energy is

$$E(I;x,y) = \frac{\sum_{u=x-k}^{x+k}\sum_{v=y-k}^{y+k} w(u-x,v-y)\left[\left(\frac{\partial I}{\partial x}(u,v)\right)^2 + \left(\frac{\partial I}{\partial y}(u,v)\right)^2\right]}{\sum_{u=x-k}^{x+k}\sum_{v=y-k}^{y+k} w(u-x,v-y)},$$

where $w(\cdot,\cdot)$ is a Gaussian weight function, whose standard deviation σ will be a command-line parameter. Use $k = \lfloor 2.5\sigma \rfloor$ to define the bounds on the Gaussian. More specifically, use `cv2.GaussianBlur`, let $h = \lfloor 2.5\sigma \rfloor$ and define

```
ksize = (2*h+1, 2*h+1).
```

See our class examples from the lecture on image processing.

Note that using the **squared** gradient magnitude, as written above, is important here. In order to ensure consistency with our implementation, 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. Specifically, the calls to the Sobel kernel should be

```
im_dx = cv2.Sobel(im, cv2.CV_32F, 1, 0)
im_dy = cv2.Sobel(im, cv2.CV_32F, 0, 1)
```

Then, after computing $E(I_0;x,y),\ldots,E(I_{n-1};x,y)$ across the n images in a sequence, there are many possible choices to combine the images into a final image. Please use

$$I^*(x,y) = \frac{\sum_{i=0}^{n-1} E(I_i;x,y)^p I_i(x,y)}{\sum_{i=0}^{n-1} E(I_i;x,y)^p},$$

where $p > 0$ is a parameter provided in the Jupyter notebook. In other words, $E(I_i;x,y)^p$ becomes the weight for a weighted averaging. (As $p \to \infty$ this goes toward the maximum, and as $p \to 0$ this becomes a simple average, with the energy measure having no effect (something we do not want).) Note that a separate averaging is done at each pixel.

While this seems like a lot, OpenCV and NumPy tools make it relatively straight forward. You will have to be careful of image boundaries (see lecture discussion of boundary conditions). Also, this will have to work on color images, in the sense that the gradients are computed on gray scale images, but the final image still needs to be in color.

In summary, your code will require three variables to be specified in the Jupyter notebook: (1) `image_dir` is the path to the directory that contains the images to test; (2) `sigma` (assume $\sigma > 0$) for the Gaussian weighting; and (3) $p > 0$ is the exponent on the energy function E in the formation of the final image.

Now for the displayed results. For pixels $(M//3, N//3)$, $(M//3, 2N//3)$, $(2M//3, N//)$ and $(2M//3, 2N//3)$ where (M,N) is the shape of the image array, output the following:

- The value of E for each image
- The final value of I^* at that pixel

These should be accurate to 1 decimal place. Example results are in the starting Jupyter notebook. Lastly, display the final image.

Finally, as part of the submitted Jupyter notebook (in a separate markdown section) discuss the results of your program: what works well, what works poorly, and why this might be. Illustrate with examples where you can. This part is worth 8 points toward your grade on this homework.