

CSci 4270 and 6270
Computational Vision,
Fall Semester, 2019
Homework 2
Due: Friday, September 27, 5 pm

Please refer back to HW 1 for guidelines. Solve each of these problems as an individual.

There are You will need to submit a pdf file for problems for the written problems

For Problems #1 and #2, please submit a separate pdf file giving your answer. You must typeset (not hand-print) the answer. I strongly recommend LaTeX to create nice, clear mathematics. Use the websites OverLeaf or ShareLaTeX to do this. There will be instructions on how to submit the final pdf on the Submittity page.



Problems 3 and 4 both require you to process a set of points represented as a 2d NumPy array. You should be able to do this without any for loops that explicitly iterate over the points, and will only earn full credit if you do. However, the penalty if you find yourself needing to write complete loops will be about 5-10% of the grade on that problem.

Problem 5 is to be completed only by students in the undergraduate version of the course, 4270, while problem 6 is to be completed by students in the graduate version, 6270.

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)** Let \mathbf{A} be an $m \times n$ matrix of real values, with $m \geq n$. What is the relationship between the SVD \mathbf{A} and the eigen decomposition of $\mathbf{A}^\top \mathbf{A}$? Justify your answer. You will need to know that the eigenvalues of a matrix are unique up to a reordering of eigenvalues and vectors, so you may assume they are provided in any order you wish. By construction, the singular values are non-increasing. (The eigenvectors / singular vectors are unique up to a reordering only if the eigenvalues / singular values are unique.)
3. **(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) Given a set of point coordinates in \mathbb{R}^2 , compute and output the following:
 - (a) The minimum and maximum x and y values.
 - (b) The center of mass (average) x and y values. 
 - (c) The axis along which the data varies the least and the standard deviation (call it s_{\min}) of this variation. **Note that the standard deviation is the square root of the eigenvalue.** **Resolve ambiguity in the direction by ensuring that the x component is positive. If this component is 0, make sure the y component is positive.** 
 - (d) The axis along which the data varies the most and the standard deviation (call it s_{\max}) of that variation. Resolve the ambiguity here in the same way as the previous problem.
 - (e) The closest point form of the best fitting line (through the original data). Ambiguity in the representation should be resolved such that ρ is positive. (You need not check the case of the center of mass being exactly (0,0), which would introduce a further ambiguity.) You should just output the values of ρ and θ , with θ in radians.
 - (f) The *implicit form* of the line. Just output a , b and c . Ensure that $a^2 + b^2 = 1$. Resolve sign ambiguity by ensuring that $c = -\rho$.
 - (g) A decision about the shape that best describes the data: a line or an ellipse. This is a line if $s_{\min} < \tau s_{\max}$ and an ellipse otherwise. **(Note that τ is an input parameter.)**

All output floating point values must be accurate to three decimal places.

Finally, output a Matplotlib plot **saved as an image** containing a scatter plot of the points and of the center of mass. Add to this the best fitting line. Use the functions `scatter`, `plot`, `axes` and `savefig`. Make sure the units both vertically and horizontally are at least roughly the same. I am providing less guidance here on the result than I did on the text output, so make it look good as you see fit. We will not be terribly picky.

The command line will be

```
python p1_shape.py points.txt tau outfig
```

where `points.txt` is a text file containing the points, `tau` is the value of τ described above and `outfig` in an image file name where you are to write the output figure.

Here is an example using data we will provide:

```
python p1_shape.py p1_test1_in.txt 0.5 p1_test1_plot.pdf
```

gives output:

```
min: (0.520,1.201)
max: (48.233,54.608)
com: (30.279,22.690)
min axis: (0.591,0.807), sd 5.202
max axis: (0.807,-0.591), sd 11.698
closest point: rho 36.195, theta 0.939
implicit: a 0.591, b 0.807, c -36.195
best as linee
```

2. (25 points) Implement the RANSAC algorithm for fitting a line to a set of points. We will start our discussion with the command line.

```
python p2_ransac.py points.txt samples tau [seed]
```

where `points.txt` is a text file containing the points in exactly the same form as the previous problem, `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 an optional parameter giving the seed to the random number generator.

After reading the input, if the seed is provided, your first call to a NumPy function must be

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

otherwise, do not call the `seed` function. 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 $a^2 + b^2 = 1$), 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 three 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 — no need to repeat the previous problem.

Here is an example command-line

```
python p2_ransac.py test0_in.txt 25 2.5 999 > p4_test1_out.txt
```

and output

```
Sample 0:
indices (0,28)
line (-0.983,0.184,-26.286)
inliers 13
```

```
Sample 3:  
indices (27,25)  
line (0.426,0.905,-4.913)  
inliers 19
```

```
Sample 10:  
indices (23,4)  
line (0.545,0.838,-0.944)  
inliers 21
```

```
avg inlier dist 0.739  
avg outlier dist 8.920
```

3. **(15 points — 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 is value is closely related to what is referred to as the “energy” of the image. More specifically, this is

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

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.

The command-line of your program will be

```
python p3_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.

Here is an example:

```
python p3_best_focus.py evergreen
```

produces the output

```
DSC_1696.JPG: 283.89  
DSC_1697.JPG: 312.74  
DSC_1698.JPG: 602.41  
DSC_1699.JPG: 2137.22
```

DSC_1700.JPG: 10224.80
DSC_1701.JPG: 18987.11
Image DSC_1701.JPG is best focused.

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 foreground 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 region 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 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.

Then, after computing $E(I_0; x, y), \dots, E(I_{n-1}; x, y)$ across the n images in a sequence, there are a number of 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 another command-line parameter. In other words, $E(I_i; x, y)^p$ becomes the weight for a weighted averaging. (As $p \rightarrow \infty$ this goes toward the maximum, and as $p \rightarrow 0$ 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.

The command-line of your program will be

```
python p4_composite image_dir out_img sigma p
```

where `image_dir` is the path to the directory that contains the images to test, `out_img` is the output image name, `sigma` is the value of σ (assume $\sigma > 0$) for the weighting, and $p > 0$ is the exponent on the energy function E in the formation of the final image.

Now for the output. The most important, of course, is the final image. Make sure you write this to the folder where the program is run (i.e. if you switch folders to get the images, switch back before output). In addition, for pixels $(M//4, N//4)$, $(M//4, 3N//4)$, $(3M//4, N//4)$ and $(3M//4, 3N//4)$ 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 posted with output from the command-line

```
python p4_sharp_focus.py branches test02_p4_branches_combined.jpg 5.0 2
```

Finally, submit a separate discussion of 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 10 points toward your grade on this homework.