**CSci 4270 and 6270**
**Computational Vision**
**Fall Semester, 2019**
**Homework 4**
**Due: Tuesday October 29, 5 pm**

This assignment covers material through Lecture 10. Please refer back to previous homework assignments for guidelines.

**A note on x/y vs. row/col values:** Always remember that row==y and col==x, despite the reversal in the usual ordering. Numpy 2d arrays generally have rows first and columns second, while OpenCV has x first and y second. I will not principly use one or the other, but I will try to be clear. Be sure you can swap back and forth between the two coordinate systems.

1. **(30 points)** How consistent are results of Harris keypoint detection and ORB keypoint detection with each other? One way to check is to extract the keypoints with the highest value of the Harris measure (as implemented in class, but be sure to use $\kappa = 0.004$) and the highest value of the ORB measure (see the `response` member variable of the `KeyPoint` class) and compare them.

   The free parameter is the value of $\sigma$ for the Harris code we wrote in class. (We'll fix the number of keypoints extracted by the ORB measure at 1,000.) For the Harris detector, you should apply non-maximum suppression but skip the thresholding operation. The ORB keypoints are at all different scales (see the `size` member variable of the `KeyPoint` class). For the sake of simplicity in this problem, eliminate any point whose size is above 45. This will help reduce the redundancy in the keypoints.

   The command-line should be simply

   ```
   python p1_compare.py sigma img
   ```

   where `sigma` is the value of $\sigma$ to be used for Gaussian smoothing of the original image in the Harris measure, and `img` is the input image. You may assume $\sigma$ is an integer

   For each of the Harris and ORB keypoints, output the ten keypoints with the strongest responses. For each keypoint, include the x,y position, and the response, all on one line. The response values should be accurate to 4 decimal places, while the positions should be accurate to just 1.

   Now for each of the top 100 keypoints of the Harris measure find the closest keypoint (in *image position*) among the top 200 ORB keypoints. (If a detector returns fewer than these numbers, just use what it returns.) Then calculate:

   - The average and median image distance.
   - The average and median difference in rank positions. (To be clear, for the $i$-th highest Harris keypoint, if the closest ORB keypoint is the $j$-th highest among ORB keypoints, then the difference in rank positions is $\mid i - j \mid$.)

   Repeat the process calculation, but comparing the top 100 ORB keypoints against the top 200 Harris keypoints.

   See the example provided for output details. These outputs should be accurate to one decimal place.

Finally output the top 200 keypoints themselves, drawn on top of the original grayscale (not color) images. Add `_harris` and `_orb` to the file name prefixes to generate the files.

2. **(40 points)** Implement the keypoint gradient direction estimation technique based on the Lecture 09 notes and in particular the detailed discussion in class. This includes weighted voting, peak estimation, and subpeak interpolation. The $\sigma_v$ used in Gaussian smoothing for the orientation voting should be 2 times the $\sigma$ used in Gaussian smoothing prior to computation of the image derivatives. The region over which the orientation calculation is made should be $2w + 1$ pixels wide, where $w = \text{rnd}(2 * \sigma_v)$. In this problem $\sigma$ might be a non-integer.

   Be sure to include the final smoothing step applied to the histogram. In particular, replace each histogram value with the weighted average of it and half the magnitudes of its two closest neighbors. For example, if a histogram entry is 10 and its two neighbors have values of 8 and 7, then the new histogram value will be $(10 + (8 + 7)/2)/2 = 8.75$.

   The command-line for your program should be

   ```
   python p2_orientation.py sigma img points
   ```

   where `sigma` is the value of $\sigma$ for Gaussian smoothing of the image prior to derivative computation, `img` is the input image, and `points` is a file containing integer-valued row/column pixel locations at which to compute orientation.

   The output from your program is a detailed output from the keypoint point locations provided. For each, output

   (a) the histogram before and after smoothing: in each line, output the min and max orientations (in integer degrees) covered by that bin, the histogram value before smoothing and after smoothing (accurate to one decimal).

   (b) all peaks locations (in degrees in the range -180 to 180), and the peak values, ordered by decreasing peak height (interpolated), and

   (c) the number of orientation histogram peaks that are either the maximum (interpolated) value or within 80% of the maximum value (after interpolated).

   (You may want to draw for yourself the histogram as a pyplot figure, but remember to remove this before submitting) Finally, you may assume that the points are sufficiently far from the image border that the orientation computation region is entirely inside the image.

   Finally, this problem will significantly test your skills in using vector programming to build the histogram for any particular location. Once this histogram is built you will need to iterate through it to find the peaks and apply parabolic interpolation.

3. **(40 points)** In class we derived the equation showing the relationship between two images taken of the same scene when the two cameras taking the pictures have the same location (no translation between them) and different orientations and intrinsic parameters. The intrinsic parameter matrices and the rotation matrices of the cameras can be combined to form a single $3 \times 3$ homography matrix $\mathbf{H}$ that determines the mapping of each pixel from one image to the other. Specifically, if $\tilde{\mathbf{u}}_1 = (c_1, r_1, 1)^\top$ and $\tilde{\mathbf{u}}_2 = (c_2, r_2, 1)^\top$ are the homogeneous pixel column and row locations in images 1 and 2, respectively, for the same point in the world, then they are related by

$$\tilde{\mathbf{u}}_2 = \mathbf{K}_2\mathbf{R}_2\mathbf{R}_1^\top\mathbf{K}_1^{-1}\tilde{\mathbf{u}}_1,$$

or more simply

$$\tilde{\mathbf{u}}_2 = \mathbf{H}_{2,1}\tilde{\mathbf{u}}_1,$$

where

$$\mathbf{H}_{2,1} = \mathbf{K}_2\mathbf{R}_2\mathbf{R}_1^\top\mathbf{K}_1^{-1}.$$

With this background in mind, write a program that achieves the following:

(a) Reads the intrinsic parameters and the rotation angles for each camera from a file. These will be in the same format as Problem 1 from HW 3, except there will be no translation vector and the pixel dimension and focal length are combined into a single scale factor, $s$, that will be assigned to $\mathbf{K}(0,0)$ and $\mathbf{K}(1,1)$. In particular, these are

```
rx1 ry1 rz1
s1  ic1 jc1
rx2 ry2 rz2
s2  ic2 jc2
```

Use these values to form $\mathbf{H}_{2,1}$. Normalize $\mathbf{H}_{2,1}$ so that the sum of the squared magnitude of the values in the matrix is 1 (the "Froebnius-norm" is 1) and the last term of the matrix is positive (you may assume it is non-zero). Multiply by 1,000 to scale the values and then print $\mathbf{H}_{2,1}$ accurate to three decimal points.

(b) Assuming (throughout the rest of the entire problem) that each image has 4,000 rows and 6,000 columns, determine the bounding rectangle in image 2's coordinate system on the mapping of image 1. To do this, map the four corners of image 1 — pixel locations $(0,0)$, $(0,6000)$, $(4000,0)$ and $(4000,6000)$ — into image 2 and compute the minimum and maximum row and column values, giving the upper left corner and lower right corner of the box. Output these values accurate to one decimal place.

(c) Determine the amount of overlap between images 1 and 2. This is difficult to do analytically (almost impossible for non-linear transformations, which first appear when we start to consider radial-lens distortion). A more general way that works even for non-linear transformations is to do the following:

- sample a regular grid of pixel locations in image 1,
- map each sampled location into image 2,
- count the number of locations that fall inside the bounds of image 2, and
- determine the fraction by diving this count by the total number of samples.

Do this for both mapping image 1 into image 2 and, using the inverse of $\mathbf{H}_{2,1}$, image 2 into image 1. (In general, these fractions will be different, primarily due to changes in scale between the images.) Use $N$ equally-spaced row samples and $N$ equally-spaced column samples, where $N$ is an integer that you read from the fifth line of the input file (the line after the one containing `f2 d2 ic2 jc2`). To determine the row samples, compute $\Delta_r = 4000/N$ and take the first sample from row $\Delta_r/2$. (Remind yourself what `np.linspace` does.) Do the same to sample $N$ columns. As an example, if $N = 10$, the row samples should be $200, 600, 1000, 1400, 1800, 2200, 2600, 3000, 3400 and 3800$.

(d) Finally, returning to the derivation of $\mathbf{H}$ that we did in class, find the unit direction vector $\hat{d}_2$ of the line that projects onto the center pixel of **image 2** (pixel location $(2000, 3000)$). Avoid sign ambiguity by making sure the last term of $\hat{d}_i$ is positive. Output the vector as three numbers on one line, accurate to 3 decimal points.

The commmand-line of the Python script should just have a single argument giving the name of the input text file.

**Notes:**

- Follow the example output file to match our formatting.
- Be careful of the ordering of the intrinsic parameter matrices and rotation matrices for images 1 and 2. In the formatted notes and as described above the mapping is from image 1 to image 2, but in the hand-written notes I derived from the mapping from image 2 to image 1.
- This can be solved completely using vector programming. Numpy tools you might need include `meshgrid`, `concatenate`, `ravel`, `stack` and `count_nonzero`.