CSci 4270 and 6270
Computational Vision,
Fall Semester, 2019
Homework 5
Due: Tuesday, November 12, 5 pm

WARNING: This may be the longest assignment of the semester. We strongly urge you to start early, and make use of office hours.

This homework, worth **140 points** toward your overall homework grade, is to be done in two parts. The first is a multistage problem of recognizing whether or not images are taken of the same scene and building a montage of images taken not only of the same scene, but from the same viewpoint (or take of a planar surface). This problem will require an extensive write-up describing your design decisions and results.

The second part of this assignment involves two short problems on GrabCut and K-means segementation.

Each part will be submitted separately through Submitty, but neither will involve any Submitty auto-grading.

Part 1 — 100 Points

Here is the basic problem statement, which is elaborated on below: Given a folder of N images as input,

- determine which pairs of images show the same scene,
- among those that do show the same scene, determine which images overlap sufficiently that they can be combined into a mosaic (either because the surface is flat or because the images were taken by a camera that was approximately rotated in place), and
- for each pair of images that can be combined into a mosaic, generate and output a mosaic of the two images.

It is possible that more than two images in a set of input images do show the same scene **and** may be combined into a mosaic. This is where the undergraduate and graduate versions of this assignment differ. In order to earn full credit, graduate students **must** produce a multi-image mosaic (in addition to the image pair mosaics). Undergraduates can earn a small amount of extra credit for multi-image mosaics. For graduate students this is the last 10 points on the assignment. For undergraduates, this is 5 points of extra credit. More on this below.

Returning to the overall assignment, each set images you are given includes $k \geq 2$ images, $I_0, \ldots, I_{k-1}$. For each pair of images, I_i and I_j, with $0 \leq i < j < k$, your code must do the following:

1. Extract and match keypoints from each image. You may use any method you wish, including ORB, SIFT or SURF. Output I_i and I_j side-by-side with lines drawn between matched keypoints (see `cv2.drawMatches`). The final decision about where or not two keypoints match may be made using the ratio test for descriptors like SIFT, or using the symmetric matching criteria for descriptors like ORB. At this point there should be errors in your keypoint matches.

2. As long as there is least a minimum number of matches or percentage of keypoints that match (you decide on thresholds!), use RANSAC to estimate the fundamental matrix $\mathbf{F}_{j,i}$ that maps pixel locations from I_i onto lines in I_j. (Please review the significance of the fundamental matrix in your class notes!) You may use `cv2.findFundamentalMat`. Do this with the `method` setting `cv2.FM_RANSAC`; you will have to explore the other parameter settings. Show I_i and I_j side-by-side with lines drawn between corresponding keypoints (a keypoint match) that was a RANSAC inlier during fundamental matrix estimation.

   In more detail, if $\tilde{\mathbf{u}}_i$ (from image I_i) and $\tilde{\mathbf{u}}_j$ (from image I_j) are matching keypoint locations, written in homogeneous coordinates, then let $\mathbf{a}_{j,i} = \mathbf{F}_{j,i}\tilde{\mathbf{u}}_i$ be the coordinates describing the line in image j along which $\tilde{\mathbf{u}}_j$ must lie if it is a correct match. (This is the "epipolar line".) Make sure you've normalized $\mathbf{a}_{j,i}$ properly so that you can measure distances correctly. The keypoint match survives — is kept as an "inlier" — if the distance from $\tilde{\mathbf{u}}_j$ to the line is less than a threshold that you specify.

3. Based on the results of fundamental matrix estimation, decide if the images do have sufficient matches to be recognized as originating from the same scene. In a clean and easily understood form, output information about the results of matching that led to the decision. This might include some combination of the number of matches, the percentage of keypoints that matched, the percentage of initial matches that remain as inliers, and a measure of the error in the matches.

4. If your code decides that I_i and I_j show the same scene, estimate the homography matrix $\mathbf{H}$ mapping I_i onto I_j using RANSAC. You may use `cv2.findHomography`. Show I_i and I_j side-by-side with lines drawn between corresponding keypoints as decided by the homography matrix estimation.

5. Based on a combination of the homography estimation and fundamental matrix estimation, decide if the two images can be aligned accurately enough to form a nice mosaic (you must judge!). It will take some work to figure out a good decision criteria. Output information about the results of matching that led to the decision. Hint: think about what it means if $\mathbf{H}$ and $\mathbf{F}$ produce (almost) the same number of matches vs. if they produce a significantly different number of matches.

6. If the decision after the previous step is "yes" then build and output the mosaic. Try to come up with a relatively simple blending method that yields nice results instead of looking like one image is mapped and pasted on top of the other.

Here is a bit about forming multi-image mosaics, a problem you should leave until everything else is done. First, you need to remember which pairs of images can be aligned using a homography. Think of the images as nodes in a graph and the image pairs as edges. The images that will form the mosaic are the connected components. (If for some reason there are more than one connected component, pick the largest.) Second, you will need to pick an "anchor" image that will remain fixed while the other images are mapped onto it. This should in some sense be the "center" of the set of images in the connected component. Third, you need to compute the transformations that map the images onto this anchor image. This can get tricky quite quickly, so please do something very easy using only the results of matching pairs of images. In particular, if I_0 is the anchor and I_i aligns with I_0, then use the transformation homography computed between them. If I_i does not have a homography with I_0, but there is another image I_j that does, then "compose" the transformations: I_i onto I_j onto I_0. This is not as hard as it sounds. In particular, if $\mathbf{H}_{j,i}$ is

the estimated transformation matrix from I_i onto I_j and if $\mathbf{H}_{0,j}$ is the estimated transformation matrix from I_j onto I_0, then $\mathbf{H}_{0,i} = \mathbf{H}_{0,j}\mathbf{H}_{j,i}$ is a good estimation of the transformation from I_i onto I_0. In the data I provide there **will not** be any cases where you need to compose more than two transformations if you choose the anchor correctly. Note that commercial software that builds multi-image mosaics uses much more sophisticated methods to estimate $\mathbf{H}_{0,i}$.

Command Line and Output

Your program should run with the following very simple command line:

```
python hw4_align.py in_dir out_dir
```

where `in_dir` is the path to the directory containing input images. Write all images to `out_dir`, which should probably be a different directory from `in`dir+ to avoid clutter across multiple runs. Your code will need to output (via print statements) a significant amount of text as described above. For each mosaic you create, make the file name be the composition of the names of the input file prefixes, in sorted order. For example, if the images are `foo.jpg`, `bar.jpg` and `cheese.jpg`, then the mosaic of the first two will be `bar_foo.jpg` and the mosaic of all three will be `bar_cheese_foo.jpg`. Use the extension from the first image (all images will be jpg).

Write Up and Code

Please generate a write-up describing your algorithms, your decision criteria, your blending algorithms, and your overall results. Evaluate **both strengths and weaknesses**, using images — perhaps including some we did not provide — to illustrate. The actual text should be no more than a page or so, single-spaced, but the document should be longer because of the images.

Finally, make sure your code is clean, reasonably efficient, documented, and well-structured.

Complete Submission

Your final submission for Part 1 will be a single zip find that includes the following:

1. Your .py file
2. The output files — both text and images — from running your code on **each** of the image sets provided.
3. Your final write-up.

The zip file will be limited to 60MB.

Part 2

Combine your write ups from each of these two Part 2 questions into a single pdf file.

1. (**20 points**) Explore the GrabCut function in OpenCV. See

   `https://docs.opencv.org/4.1.2/d8/d83/tutorial_py_grabcut.html`

In particular, demonstrate an example image of your own choosing where GrabCut works well and an example image where it works poorly. (These images must be ones that you find or take and not images that someone else has worked on with GrabCuts.) A big part of your effort is determining the rectangle bounding the foreground and, within this rectangle, which pixels are definitely foreground and which are definitely background. To do this, please examine the image interactively to find the bounds on one or more rectangular regions you'd like to be part of the foreground and one or more rectangular regions you'd like to be part of the background. Provide these to the GrabCuts function as part of the mask. Be sure to discuss why you think GrabCuts succeeded or failed in each case. Also, be sure to resize your image to a reasonable working dimension — say 500 to 600 pixels as the max dimension — to make this practical.

Your Python code should take the original input image as an argument and a file containing pixel coordinates defining rectangles in the image. Each line should contain four pixels defining a rectangle. The first rectangle — the "outer rectangle" — should be the bounding rectangular mask on the object. The remaining rectangles — the "inner rectangles" — should all be enclosed in this rectangle and should bound image locations that should definitely be inside or definitely outside the segmented object. (You will need some way to distinguish.) I suggest, for simplicitly, that you find the coordinates of these rectangles by hand. I realize that this is a bit clunky, but I don't want you to spend your time writing any significant user interaction code.

Your write up should show the original image, the image with the rectangles draw overtop — outer rectangle in one color, inner rectangles to be included in another, and inner rectangles to be excluded in a third. Also show the resulting segmented image. Be sure to explain your result.

2. (**20 points**) Apply k-means clustering to attempt segment the grass in the images provided on Submitty. The "data" vectors input to K-means should be at least be a concatenation of the pixel location values and the RGB values at each pixel. You might include other measures such as the standard deviation of the R, G and B values over a pixel's neighborhood so that you capture some notion of the variation in intensity (e.g. solid green regions aren't very grass-like). You might have to scale your position, color or other measurements to give them more or less influence during k-means. Vary the value of k to see the effect — perhaps several clusters together cover the grass.

   Note that I don't expect perfect segmentations. More sophisticated algorithms are needed for this. Instead, I want you to develop an understanding of the use of k-means and of the difficulty of the segmentation problem.

   In addition to your code, please include in your write-up a small selection of images from the ones I provided demonstrating good and bad results. Include each original image, and separately show the clusters drawn on the images. Include example images that indicate the effect of varying k and, perhaps, that demonstrate the effect of including measures other than color and position.