Intro to Computer Vision **Instructor:** *Meera Hahn*
Problem Set 2
Due: Sep 29th 11:59pm

1 Overview

The goal of this assignment is to create a local feature matching algorithm using techniques described in Szeliski Chapter 7. The pipeline we suggest is a simplified version of the famous SIFT pipeline. The matching pipeline is intended to work for instance-level matching – multiple views of the same physical scene.

2 Setup

1. Install Miniconda. This list of installers is for the latest release of Python: 3.11.3. please install from here.
2. Create a conda environment, using the appropriate command. On Windows, open the installed "Conda prompt" to run this command. On MacOS and Linux, you can just use a terminal window to run the command. Modify the command based on your OS ('linux', 'mac', or 'win'): `conda env create -f proj2_env_<OS>.yml`
3. This should create an environment named `proj3`. Activate it using the following Windows command: `activate proj3` or the following MacOS / Linux command: `source activate proj3`
4. Install the project package, by running `pip install -e .` inside the repo folder.
5. Run the notebook using: `jupyter notebook ./proj3_code/proj3.ipynb`
6. Generate the submission once you've finished the project using: `python zip_submission.py` with the argument `emory_username`

3 Details

For this project, you need to implement the three major steps of a local feature matching algorithm: see Szeliski 7

- Interest point detection in `student_harris.py`
- Local feature description in `student_sift.py`
- Feature Matching in `student_feature_matching.py`

4 Interest point detection

Fill out **`student_harris.py`**
You will implement the Harris corner detector as described in the lecture materials and Szeliski 7. See Algorithm below for pseudocode. The starter code gives some additional suggestions. The main function will be `get_interest_points()` while implementing `my_filter2D()`, `get_gradients()`, `get_gaussian_kernel()`, `second_moments()`, `corner_response()`, `non_max_suppression()` will be helper functions that will have tests to check progress as you work through Harris corner detection. You do not need to worry about scale invariance or keypoint orientation estimation for your baseline Harris corner detector.

1. Compute the horizontal and vertical derivatives of the image I_x and I_y by convolving the original image with derivatives of Gaussians (Section 3.2.3).
2. Compute the three images corresponding to the outer products of these gradients. (The matrix $\boldsymbol{A}$ is symmetric, so only three entries are needed.)
3. Convolve each of these images with a larger Gaussian.
4. Compute a scalar interest measure using one of the formulas discussed above.
5. Find local maxima above a certain threshold and report them as detected feature point locations.

5 Local feature description

Fill out **student_sift.py**

You will implement a SIFT-like local feature as described in the lecture materials and Szeliski 7.1.2. See `get_features()` for more details. If you want to get your matching pipeline working quickly (and maybe to help debug the other algorithm stages), you might want to start with normalized patches as your local feature. There are 2 helper functions in `student_sift.py` that you will need to code to get your SIFT pipeline working. `get_magnitudes_and_orientations()` will return the magnitudes and orientations of the image gradients at every pixel. `get_feat_vec()` will get the feature vector associated with a specific interest point. Once these are done, move on to coding `get_features()`, which will combine these to get feature vectors for all interest points. More info about each function can be found in the function headers.

6 Feature matching

Fill out **student_feature_matching.py**

You will implement the "ratio test" or "nearest neighbor distance ratio test" method of matching local features as described in the lecture materials and Szeliski 7.1.3. See equation 7.18 in particular. The potential matches that pass the ratio test the easiest should have a greater tendency to be correct matches – think about why.

7 Using the starter code

The top-level `proj3.ipynb` IPython notebook provided in the starter code includes file handling, visualization, and evaluation functions. The correspondence will be visualized with `show_correspondence_circles()` and `show_correspondence_lines()` (you can comment one or both out if you prefer). For the Notre Dame image pair there is a ground truth evaluation in the starter code as well. `evaluate_correspondence()` will classify each match as correct or incorrect based on hand-provided matches (see `show_ground_truth_corr()` for details). The starter code also contains ground truth correspondences for two other image pairs (Mount Rushmore and Episcopal Gaudi). You can test on those images by uncommenting the appropriate lines at the top of `proj3.ipynb`. As you implement your feature matching pipeline, you should see your performance according to `evaluate_correspondence()` increase. Hopefully you find this useful, but don't overfit to the initial Notre Dame image pair which is relatively easy. The baseline algorithm suggested here and in the starter code will give you full credit and work fairly well on these Notre Dame images, but additional image pairs provided in the folder data are more difficult. They might exhibit more viewpoint, scale, and illumination variation. If you add enough Bells & Whistles you should be able to match more difficult image pairs.

Suggested implementation strategy

It is **highly suggested** that you implement the functions in this order:

- First, use `cheat_interest_points()` instead of `get_interest_points()`. This function will only work for the 3 image pairs with ground truth correspondence. This function cannot be used in your final implementation. It directly loads interest points from the the ground truth correspondences for the test cases. Even with this cheating, your accuracy will initially be near zero because the starter code features are empty and the starter code matches don't exist. `cheat_interest_points()` might return non-integer values, but you'll have to cut patches out at integer coordinates. You should address this by truncating the numbers.
- Second, change `get_features()` to return a simple feature. Start with, for instance, 16x16 patches centered on each interest point. Image patches aren't a great feature (they're not invariant to brightness change, contrast change, or small spatial shifts) but this is simple to implement and provides a baseline. You won't see your accuracy increase yet because the placeholder code in `match_features()` isn't assigning matches.
- Third, implement `match_features()`. Accuracy should increase on the Notre Dame pair if you're using 16x16 (256 dimensional) patches as your feature and if you only evaluate your 100 most confident matches. Accuracy on the other test cases will be lower
- Fourth, finish `get_features()` by implementing a sift-like feature. Accuracy should increase to 70% on the Notre Dame pair, 40% on Mount Rushmore, and 15% on Episcopal Gaudi if you only evaluate your 100 most confident matches(these are just estimates). These accuracies still aren't great because the human selected keypoints from `cheat_interest_points()` might not match particularly well according to your feature.
- Fifth, stop using `cheat_interest_points()` and implement `get_interest_points()`. Harris corners aren't as good as ground-truth points which we know correspond, so accuracy may drop. On the other hand, you can get hundreds or even a few thousand interest points so you have more opportunities to find confident matches. If you only evaluate the most confident 100 matches (see the `num_pts_to_evaluate` parameter) on the Notre Dame pair, you should be able to achieve >80% accuracy. You will likely need to do extra credit to get high accuracy on Mount Rushmore and Episcopal Gaudi.

Potentially useful NumPy (Python library), OpenCV, and SciPy functions: `np.arctan2()`, `np.sort()`, `np.reshape()`, `np.newaxis`, `np.argsort()`, `np.gradient()`, `np.histogram()`, `np.hypot()`, `np.fliplr()`, `np.flipud()`, `cv2.getGaussianKernel()`

Forbidden functions (you can use for testing, but not in your final code): `cv2.SIFT()`, `cv2.SURF()`, `cv2.BFMatcher()`, `cv2.BFMatcher().match()`, `cv2.FlannBasedMatcher().knnMatch()`, `cv2.BFMatcher().knnMatch()`, `cv2.HOGDescriptor()`, `cv2.cornerHarris()`, `cv2.FastFeatureDetector()`, `cv2.ORB()`, `skimage.feature`, `skimage.feature.hog()`, `skimage.feature.daisy`, `skimage.feature.corner_harris()`, `skimage.feature.corner_shi_tomasi()`, `skimage.feature.match_descriptors()`, `skimage.feature.ORB()`, `scipy.signal.convolve()`, `cv2.filter2D()`, `cv2.Sobel()`, .

We haven't enumerated all possible forbidden functions here but using anyone else's code that performs filtering, interest point detection, feature computation, or feature matching for you is forbidden.

Unit Tests

To test your code, go the main directory and run the command `pytest`.

Tips, Tricks, and Common Problems

- Make sure you're not swapping x and y coordinates at some point. If your interest points aren't showing up where you expect or if you're getting out of bound errors you might be swapping x and y coordinates. Remember, images expressed as NumPy arrays are accessed `image[y,x]`.
- Make sure you're features aren't somehow degenerate. You can visualize your features with `plt.imshow(image1_features)`, although you may need to normalize them first. If the features are mostly zero or mostly identical you may have made a mistake.

Writeup

For the writeup, the specific visualizations and questions to answer are in the powerpoint template.

Acknowledgment to Mitchell Donley, Judy Hoffman and James Hays for original creation of this assignment