Assignment 2 - Coding Part: Local Feature Matching

You will create a local feature matching algorithm (Szeliski chapter 7.1 in the 2nd edition) and attempt to match multiple views of real-world scenes. There are hundreds of papers in the computer vision literature addressing each stage. We will implement a simplified version of SIFT(`https://www.cs.ubc.ca/~lowe/keypoints/`) ; however, you are encouraged to experiment with more sophisticated algorithms for extra credit!

The main file where you will be writing your code is **student.py**.

Create a separate Jupyter Notebook **Code_Writeup.ipynb** (with all cells already run) for your writeup. You may also opt to provide the writeup as a PDF instead.

Task:

Implement the three major steps of local feature matching:

- **Detection** in the get_interest_points function in student.py. Please implement the Harris corner detector (Szeliski 7.1.1). You do not need to worry about scale invariance or keypoint orientation estimation for your corner detector.
- **Description** in the get_features function, also in student.py. Please implement a SIFT-like local feature descriptor (Szeliski 7.1.2). You do not need to implement full SIFT! Add complexity until you meet the rubric. To quickly test and debug your matching pipeline, start with normalized patches as your descriptor.
- **Matching** in the match_features function of student.py. Please implement the "ratio test" or "nearest neighbor distance ratio test" method of matching local features (Szeliski 7.1.3; equation 7.18 in particular).

Potentially useful functions Any existing filter function, including the one you developed in Assignment 1.

- zip()
- skimage.measure.regionprops()
- skimage.feature.peak_local_max()
- numpy.arctan2()

Potentially useful libraries: These will provide many pre-written image filtering functions.

- skimage.filters.x()
- scipy.ndimage.filters.x()
- np.gradient() provides a more sophisticated estimate of derivatives
- np.digitize() provides element-wise binning.

Forbidden functions ANY feature/corner detector or extractor whatsoever the following are just examples.
ANY function that computes histograms for you are also forbidden.

- skimage.feature.daisy()
- skimage.feature.ORB()
- skimage.feature.corner_harris()
- sklearn.neighbors.NearestNeighbors() or any other function that computes the nearest neighbor ratio for you.
- scipy.spatial.distance.cdist() or any other function that computes the distance between arrays or vectors. You have to implement your own distance measurement.

1 Running the code

main.py takes a commandline argument using 'p' to load a dataset, e.g., 'p notre_dame'. For example, $ python main.py p notre_dame. Please see main.py for more instructions. Other valid names of datasets: mt_rushmore or e_gaudi or notre_dame

2 Requirements

2.1 Accuracy

If your implementation reaches 60% accuracy on the most confident 50 correspondences in 'matches' for the Notre Dame pair, and at least 60% accuracy on the most confident 50 correspondences in 'matches' for the Mt. Rushmore pair, you will receive full code credit. Your code will be evaluated on the image pairs at scale_factor=0.5 (main.py), so please be aware if you change this value. The evaluation function I will use is evaluate_correspondence() in helpers.py (my copy, not yours). The function is included in the starter code for you so you can measure your own accuracy.

2.2 Time limit

I really value our TA's time, so they're instructed to stop executing your code after at most 20 minutes. This is your time limit, after which you will receive a maximum of half the full grade for the implementation. You must write efficient code. Hint 1: (optional) You can use 'steerable' (oriented) filters to build the descriptor. Hint 2: Use matrix multiplication for feature descriptor matching.

2.3 Memory efficiency

You must write memory aware code. If your code's memory requirement is larger than 500 MB, you will receive a maximum of half the full grade for the implementation. The file memusecheck.py is provided to help you evaluate your program's memory usage.

3 Writeup

- Describe your process and algorithm, show your results, describe any of the extra credit steps you did.
- Report on any other information you feel is relevant
- Quantitatively compare the impact of the methods you've implemented. For example, using the SIFTlike descriptors instead of normalized patches increased the performance from 50% good matches to 70% good matches. Please includes the performance improvement induced by any of the extra credit steps.
- Show how well your method works on the Notre Dame, Mount Rushmore, and Episcopal Gaudi image pairs. The visualization code will write image-pair-specific output to your working directory, e.g., notre_dame_matches.jpg.

4 Extra Credit: (max 10 points total)

Detection:

- Up to 5 pts: Detect keypoints at multiple scales or use a scale-selection method to pick the best scale.
- Up to 5 pts: Use adaptive non-maximum suppression as discussed in the textbook.
- Up to 10 pts: Use a different interest point detection strategy like MSER. Use it alone, or take the union of multiple methods.

Description:

- Up to 3 pts: Experiment and repoet on the SIFT parameters: window size, number of local cells, orientation bins, different normalization schemes.
- Up to 5 pts: Estimate feature orientation.
- Up to 5 pts: Multi-scale descriptor. If you are detecting keypoints at multiple scales, you should build the features at the corresponding scales, too.
- Up to 5 pts: Different spatial layouts for your feature (e.g., GLOH).
- Up to 10 pts: Entirely different features (e.g., local self-similarity).

Matching: the baseline matching algorithm is computationally expensive, and will likely be the slowest part of your code. Consider approximating or accelerating feature matching:

- Up to 10 pts: Create a lower dimensional descriptor that is still sufficiently accurate. For example, if the descriptor is 32 dimensions instead of 128 then the distance computation should be about 4 times faster. PCA would be a good way to create a low dimensional descriptor. You would need to compute the PCA basis on a sample of your local descriptors from many images.
- Up to 5 pts: Use a space partitioning data structure like a kd-tree or some third party approximate nearest neighbour package to accelerate matching.

5 Implementation strategy

These feature point and accuracy numbers are only a guide; don't worry if your method doesn't exactly produce these numbers. Feel confident if they are approximately similar, and move on to the next part.

1. Use cheat_interest_points() instead of get_interest_points(). At this point, just work with the Notre Dame image pair (the cheat points for Mt. Rushmore aren't very good and the Episcopal Palace is difficult to feature match). This function cannot be used in your final implementation. It directly loads the 100 to 150 ground truth correspondences for the test cases. Even with this cheating, your accuracy will initially be near zero because the features are random and the matches are random.

2. Implement match_features(). Accuracy should still be near zero because the features are random. To do this, you'll need to calculate the distance between all pairs of features across the two images (much like scipy.spatial.distance.cdist(). While this could be written using a series of for loops, we're asking you to write a matrix implementation using numpy As you learned in project 0, writing things in numpy makes them orders of magnitude faster, and knowing how to do this is an important skill in computer vision. Make sure to also return proper confidence values for each match. Your most confident matches should have the highest values, and doing this wrong could cause the automatic code evaluation to look at the wrong matches when grading your code.

3. Change get_features() to cut out image patches. Accuracy should increase to $\tilde{6}$0-70% on the Notre Dame pair if you're using 16×16 (256 dimensional) patches as your feature. Accuracy on the other test cases will be lower (Especially using the bad interest points from cheat_interest_points). Image patches are not invariant to brightness change, contrast change, or small spatial shifts, but this provides a baseline.

4. Finish get_features() by implementing a SIFTlike feature. Accuracy should increase to 70% on the Notre Dame pair. If your accuracy doesn't increase (or even decreases a little bit), don't worry because normalized patches are actually a good feature descriptor for the Notre Dame image pair. The difference will become apparent when you implement get_interest_points and can test on Mt. Rushmore.

5. Finally Stop using the cheat_interest_points() and implement get_interest_points(). Harris corners aren't as good as ground truth points, 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. Typical solutions generates around 300 to 700 feature points for each image.

6 Notes

- main.py handles files, visualization, and evaluation, and calls placeholders of the three functions to implement. For the most part you will only be working in student.py.
- The Notre Dame, Mount Rushmore, and Episcopal Gaudi image pairs include 'ground truth' evaluation. evaluate_correspondence() will classify each match as correct or incorrect based on handprovided matches. You can test on those images by changing the arguments you pass to main.py. Check the main.py file for further info.
- As you implement your feature matching pipeline, check whether your performance increases using evaluate_correspondence(). Take care not to tweak parameters specifically for the initial Notre Dame image pair. additional Image pairs in optional_extra_data.zip (194 MB), which exhibit more viewpoint, scale, and illumination variation. With careful consideration of the qualities of the images on display, it is possible to match these, but it is much more difficult and hence are only optional.
- You will likely need to do one or more of the extra credit notes to get high accuracy on Episcopal Gaudi!!

7 Credits

Python port by Anna Sabel and Jamie DeMaria. Updated for 2020 by Eliot Laidlaw, Trevor Houchens and Georges Younes. Assignment originally developed by James Hays.