Project 2: Local Feature Matching

(Assignment developed by James Hays.)

Overview

We will create a local feature matching algorithm 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; however, you are encouraged to experiment with more sophisticated algorithms for extra credit!

Task: Implement the three major functions of local feature matching in “student.py”:

- **Detection** in `get_interest_points()`. Please implement the Harris corner detector. You do not need to worry about scale invariance or keypoint orientation estimation for your Harris corner detector.
- **Description** in `get_features()`. Please implement a *SIFT-like* local feature descriptor. To quickly test and debug your matching pipeline, start with normalized patches as your descriptor.
- **Matching** in `match_features()`. Please implement the "ratio test" or "nearest neighbor distance ratio test" method of matching local features.

Any ready-made functions are forbidden.

Task:

- Quantitatively compare the impact of the method you've implemented. For example, using SIFT-like descriptors instead of normalized patches increased our performance from 70% good matches to 90% good matches. Please include the performance improvement for any extra credit.
- Show how well your method works on the Notre Dame, Mount Rushmore, and Episcopal Gaudi image pairs (or make new ground truth!).

Rubric

If your implementation reaches 80% accuracy on the *first* 100 correspondences returned in 'matches' for the Notre Dame pair, you will receive 8 pts (full code credit).

We will evaluate your code on the image pairs at scale_factor=0.5, so please be aware if you change this value. The evaluation function we will use is scale_factor=0.5 (our copy, not yours!). We have included this file in the starter code for you.

Time limit: Your time limit is 20 minutes, after which you will receive a maximum of 60% for the implementation. You must write efficient code—think before you write.

- +2 pts: Implementation of Harris corner detector in `get_interest_points()`

- +4 pts: Implementation of SIFT-like local feature in `get_features()`
- +2 pts: Implementation of "Ratio Test" matching in `match_features()`
- +2 pts: Writeup.
- +2 pts: Extra credit (max 2 pts).
- -0.5*n pts: Where n is the number of times that you do not follow the instructions.

Extra Credit (max 2 pts)

Detection:

- up to 0.5 pts: Detect keypoints at multiple scales or using a scale selection method to pick the best scale.
- up to 1 pts: Use a different interest point detection strategy like MSER. Use it alone, or take the union of multiple methods.

Description:

- up to 0.5 pts: Experiment with SIFT parameters: window size, number of local cells, orientation bins, different normalization schemes.
- up to 0.5 pts: Estimate feature orientation.
- up to 1 pts: Multi-scale descriptor. If you are detecting keypoints at multiple scales, you should build the features at the corresponding scales, too.

Matching:
The baseline matching algorithm is computationally expensive, and will likely to be the slowest part of your code. Let's try to approximate or accelerate feature matching:

- up to 1 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 1 pts: Use a space partitioning data structure like a kd-tree or some third party approximate nearest neighbor package to accelerate matching.

Competition

A section of the grade will be based on an efficiency competition. The efficiency is calculated based on matching accuracy (taking into account the number of matches) and the memory used. The most accurate, lowest memory using code will get the highest score and the reverse will get the lowest score. This only applies to a small percentage of the grade not the whole grade.

Notes

`main.py` handles files, visualization, and evaluation, and calls placeholders of the three functions to implement. Running the starter code without modification will visualize random interest points matched randomly on the Notre Dame images. The correspondence will be visualized with `show_correspondence()`.
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 hand-provided matches. You can test on those images by uncommenting the appropriate lines in `main.py`.
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.
You should **only** edit in the '*student.py*' file.

An Implementation Strategy

1. 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 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.
3. Change `get_features()` to cut out image patches. Accuracy should increase to ~40% on the Notre Dame pair if you're using 16x16 (256 dimensional) patches as your feature. Accuracy on the other test cases will be lower (Mount Rushmore 25%, Episcopal Gaudi 7%). 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 SIFT-like feature. Accuracy should increase to 70% on the Notre Dame pair, 40% on Mount Rushmore, and 15% on Episcopal Gaudi. These accuracies still aren't great because the human-selected correspondences might look quite different at the local feature level. If you're sorting your matches by confidence (as the starter code does in `match_features()`) you should notice that your more confident matches are more likely to be true matches—these pass the ratio test more easily.

5. Stop cheating (!) and implement `get_interest_points()`. Harris corners aren't as good as ground-truth points (which we know to 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 on the Notre Dame pair, you should be able to achieve 90% accuracy (see the `num_pts_to_evaluate` parameter).

You will likely need to do extra credit to get high accuracy on Mount Rushmore and Episcopal Gaudi.

Deliverables

Upload your files following the code convention to Classroom. The file should include your code, data, ReadMe file, and report.