Assignment 2: Local Feature Matching

Overview

We will create a local feature matching algorithm (Szeliski chapter 4.1) 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 steps of local feature matching:

- **Detection** in the `get_interest_points` function in `student.py`. Please implement the Harris corner detector (Szeliski 4.1.1, Algorithm 4.1). You do not need to worry about scale invariance or keypoint orientation estimation for your Harris corner detector.
- **Description** in the `get_features` function, also in `student.py`. Please implement a *SIFT-like* local feature descriptor (Szeliski 4.1.2). *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 4.1.3; equation 4.18 in particular).

Potentially useful functions: Any existing filter function, `zip()`, `skimage.measure.regionprops()`, `skimage.feature.peak_local_max()`, `numpy.arctan2()`.

Potentially useful libraries: `skimage.filters.x()` or `scipy.ndimage.filters.x()`, which provide many pre-written image filtering functions. `np.gradient()`, which provides a more sophisticated estimate of derivatives. `np.digitize()` provides element-wise binning. In general, anything which we've implemented ourselves in a previous project is fair game to use as an existing function.

Forbidden functions: `skimage.feature.daisy()`, `skimage.feature.ORB()`, and any other functions that extract features for you, `skimage.feature.corner_harris()` and any other functions that detect corners for you, any function which *computes histograms*, `sklearn.neighbors.NearestNeighbors()` and any other functions that compute nearest neighbor ratios for you. `scipy.spatial.distance.cdist()` and any other functions that compute the distance between arrays of vectors (use the guide we've provided to implement your own distance function!). If you are unsure about a function, please ask.

Running the Code

`main.py` takes a command-line 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.

Requirements / Rubric

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 75 pts (full code credit). We will evaluate your code on the image pairs at scale_factor=0.5 (`main.py`), so please be aware if you change this value. The evaluation function we will use is `evaluate_correspondence()` in `helpers.py` (our copy, not yours!). We have included this function in the starter code for you so you can measure your own accuracy.

Time limit: The autograder will stop executing your code after 10 minutes. This is your time limit, after which you will receive a maximum of 40 points for the implementation. You must write efficient code—think before you write.

- *Hint 1: Use 'steerable' (oriented) filters to build the descriptor.*
- *Hint 2: Use matrix multiplication for feature descriptor matching.*

Compute/memory limit: The autograder runs on a modern desktop CPU about as powerful as your laptop. It does not have a GPU. It has 4GB memory, and going over this may terminate your program unexpectedly (!). It uses the same Python virtual environment as the department machines. If you are concerned about the memory usage of your program, you can check it by running `python memusecheck` in your code directory.

- +30 pts: Implementation of Harris corner detector in `get_interest_points()`
- +40 pts: Implementation of SIFT-like local feature in `get_features()`
- +10 pts: Implementation of "Ratio Test" matching in `match_features()`
- +20 pts: Writeup.
    - Template in writeup/. Please describe your process and algorithm, show your results, describe any extra credit, and tell us any other information you feel is relevant. We provide you with a LaTeX template. Please compile it into a PDF and submit it along with your code.
    - Quantitatively compare the impact of the method you've implemented. For example, using SIFT-like descriptors instead of normalized patches increased our performance from 50% good matches to 70% good matches. Please includes the performance improvement for any extra credit.
    - 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`.

- -05*n pts: Where n is the number of times that you do not follow the instructions.

An Implementation Strategy

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. To help you with this implementation, we're providing a guide that can be found here. 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 autograder to look at the wrong matches when grading your code.
3. Change `get_features()` to cut out image patches. Accuracy should increase to ~60-70% 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 (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 SIFT-like 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. 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. *Our solution* generates around 300–700 feature points for each image.

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.

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`.

You may submit your code to gradescope as many times as you like. We will by default only consider the last submission. You may find this helpful because the autograder will test each of your functions individually and give you insight as to where problems might lie.

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 changing the `-p` argument you pass to `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. We provide additional image pairs in 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 more difficult.

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

Credits

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