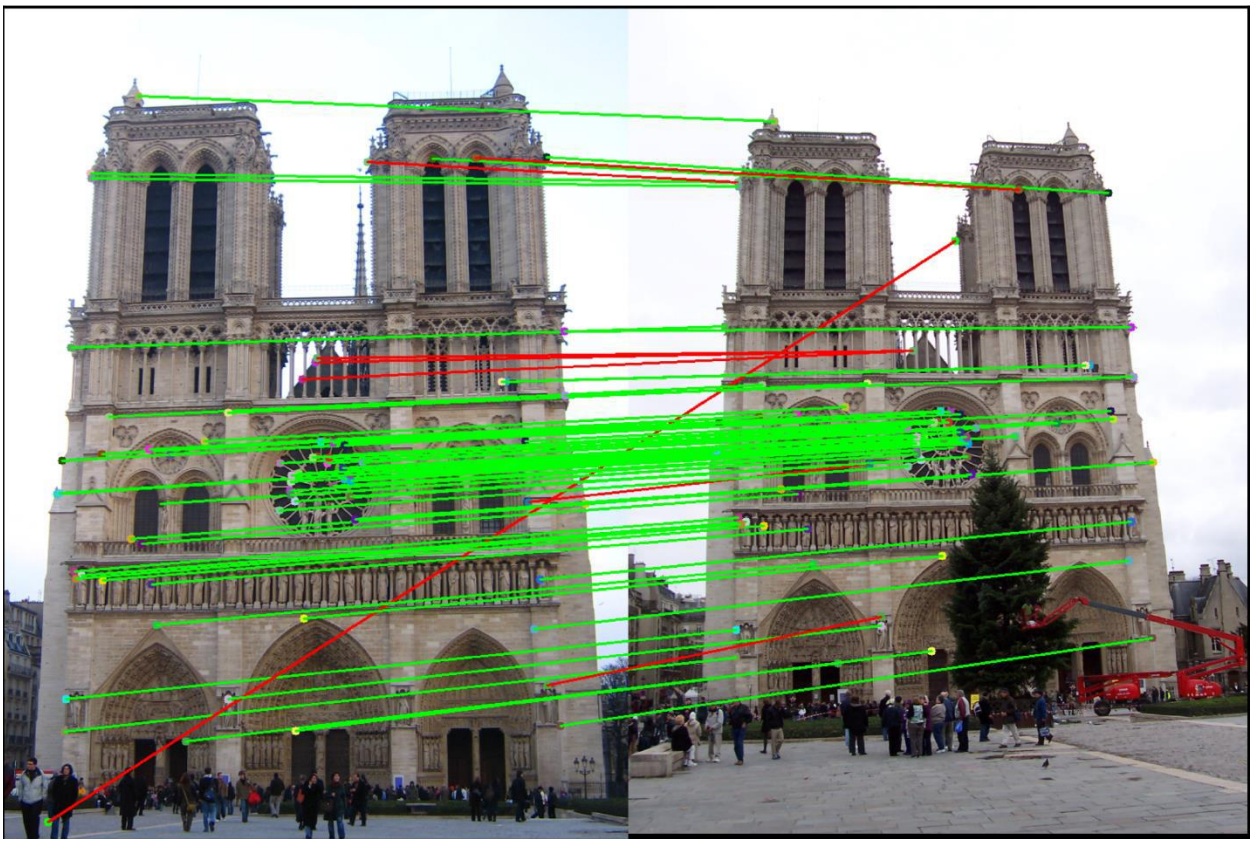
CS 815: Computer Vision
Assignment-2
Due Date: June 16 at 11:59 pm
Submit your solutions as described in the class.



The top 100 most confident local feature matches from a baseline implementation of assignment 2. In this case, 89 were correct (lines shown in green) and 11 were incorrect (lines shown in red).

Overview

The goal of this assignment is to create a local feature matching algorithm using techniques described in Szeliski chapter 4.1. 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.

Details

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

- Interest point detection in `student_harris.py` (see Szeliski 4.1.1)
- Local feature description in `student_sift.py` (see Szeliski 4.1.2)

- Feature Matching in `student_feature_matching.py` (see Szeliski 4.1.3)

There are numerous papers in the computer vision literature addressing each stage. For this project, we will suggest specific, relatively simple algorithms for each stage. You are encouraged to experiment with more sophisticated algorithms!

Interest point detection (`student_harris.py`)

You will implement the Harris corner detector as described in the lecture materials and Szeliski 4.1.1. See Algorithm 4.1 in the textbook for pseudocode. The starter code gives some additional suggestions. You do not need to worry about scale invariance or keypoint orientation estimation for your baseline Harris corner detector. The original paper by Chris Harris and Mike Stephens describing their corner detector can be found here.

You will also implement **adaptive non-maximal suppression**. While most feature detectors simply look for local maxima in the interest function, this can lead to an uneven distribution of feature points across the image, e.g., points will be denser in regions of higher contrast. To mitigate this problem, Brown, Szeliski, and Winder (2005) only detect features that are both local maxima and whose response value is significantly (10%) greater than that of all of its neighbors within a radius r. The goal is to retain only those points that are a maximum in a neighborhood of radius r pixels. One way to do so is to sort all points by the response strength, from large to small response. The first entry in the list is the global maximum, which is not suppressed at any radius. Then, we can iterate through the list and compute the distance to each interest point ahead of it in the list (these are pixels with even greater response strength). The minimum of distances to a keypoint's stronger neighbors (multiplying these neighbors by >=1.1 to add robustness) is the radius within which the current point is a local maximum. We call this the suppression radius of this interest point, and we save these suppression radii. Finally, we sort the suppression radii from large to small, and return the n keypoints associated with the top n suppression radii, in this sorted order. Feel free to experiment with n, we used `n=1500`.

You can read more about ANMS in the textbook, this conference article, or in this paper which describes a fast variant.

Local feature description (`student_sift.py`)

You will implement a SIFT-like local feature as described in the lecture materials and Szeliski 4.1.2. See the placeholder `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.

Feature matching (`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 4.1.3. See equation 4.18 in

particular. The potential matches that pass the ratio test the easiest should have a greater tendency to be correct matches -- think about *why*.

Using the starter code (`proj2.ipynb`)

The top-level `proj2.ipynb` IPython notebook provided in the starter code includes file handling, visualization, and evaluation functions for you as well as calls to placeholder versions of the three functions listed above. Running the starter code without modification will visualize random interest points matched randomly on the particular Notre Dame images shown at the top of this page. 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 in `proj2.ipynb`. You can create additional ground truth matches with the `CorrespondenceAnnotator().collect_ground_truth_corr()` found in `annotate_correspondences/collect_ground_truth_corr.py` (but it's a tedious process).

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 `extra_data2.zip` 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 ground truth correspondences for the test cases. Even with this cheating, your accuracy will initially be near zero because the starter code features are all zeros and the starter code matches are random. `get_interest_points()` returns non-integer values, but you'll have to cut patches out at integer coordinates. You could address this by rounding the coordinates or doing some form of interpolation. Your own `get_features()` can also return non-integer coordinates (many methods do try to localize interest points to sub-pixel coordinates).

- 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()` is randomly assigning matches.

- Third, implement `match_features()`. Accuracy should increase to ~40% 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 (Mount Rushmore 25%, Episcopal Gaudi 7%). If you're sorting your matches by confidence (as the starter code does in `match_features()`) you should notice that your more confident matches (which pass the ratio test more easily) are more likely to be true matches.

- 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 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 90% accuracy. As long as your accuracy on the Notre Dame image pair is 80% for the 100 most confident matches you can receive full credit for the project. When you implement adaptive non-maximal suppression, your accuracy should improve even more.

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(),        cv2.Sobel(),
cv2.filter2D(), scipy.signal.convolve().
```

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.FastFeatureDetector()`, `cv2.ORB()`, `cv2.cornerHarris()`,`skimage.feature`, `skimage.feature.hog()`, `skimage.feature.daisy`, `skimage.feature.corner_shi_tomasi()`, `skimage.feature.corner_harris()`,`skimage.feature.match_descriptors()`, `skimage.feature.ORB()`.

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

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 your 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

In the report you will describe your algorithm and any decisions you made to write your algorithm a particular way. Then you will show and discuss the results of your algorithm.

In the case of this project, show how well your matching method works not just on the Notre Dame image pair, but on additional test cases. For the 3 image pairs with ground truth correspondence, you can show `eval.jpg` which the starter code generates. For other image pairs, there is no ground truth evaluation (you can make it!) so you can show `vis_circles.jpg` or `vis_lines.jpg` instead. A good writeup will assess how important various design decisions were.

Rubric

- 25 pts: Implementation of Harris corner detector in `student_harris.py`
- 10 pts: Implementation of adaptive non-maximal suppression in `student_harris.py`
- 35 pts: Implementation of SIFT-like local feature in `student_sift.py`
- 10 pts: Implementation of "Ratio Test" matching in `student_feature_matching.py`
- 20 pts: Writeup with several examples of local feature matching.

Credits

This assignment was originally developed by James Hays, Cusuh Ham, John Lambert, Vijay Upadhya, and Samarth Brahmbhatt.