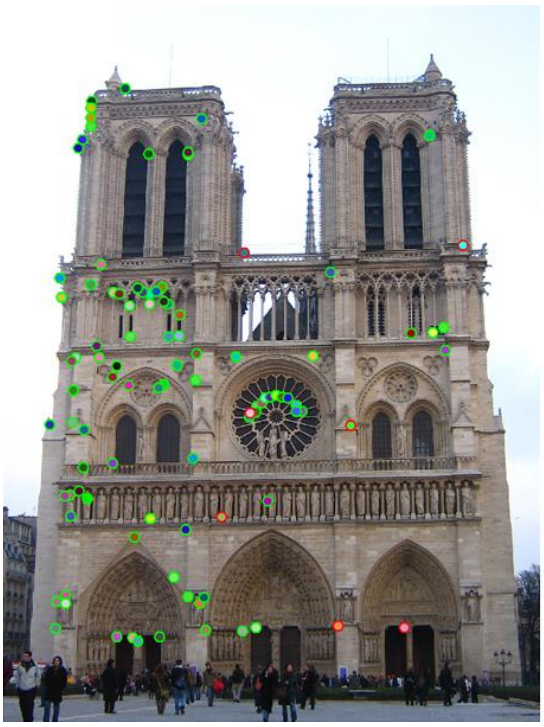
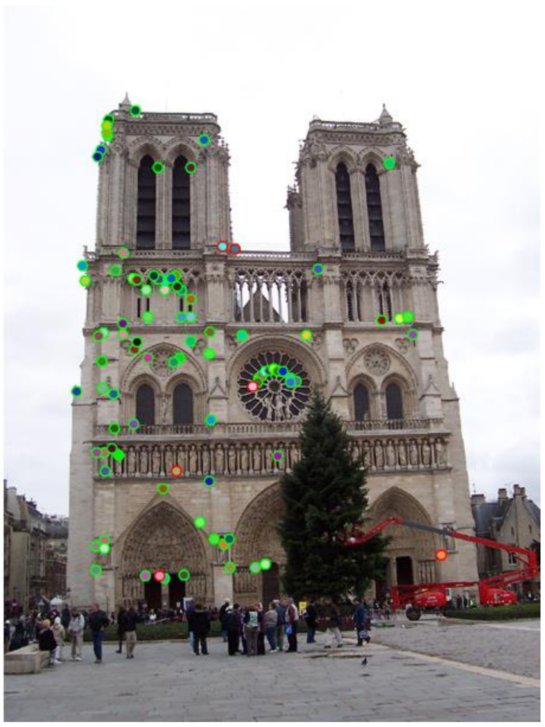

The top 100 most confident local feature matches from a baseline implementation of project 2. In this case, 93 were correct (highlighted in green) and 7 were incorrect (highlighted in red)..

Project 2: Local Feature Matching

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 get_interest_points.m (see Szeliski 4.1.1)
- Local feature description in get_features.m (see Szeliski 4.1.2)
- Feature Matching in match_features.m (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 (get_interest_points.m)

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.
Bells & whistles for interest point detection. Try detecting keypoints at multiple scales or using a scale selection method to pick the best scale. Try estimating the orientation of keypoints to make your local features rotation invariant. You can also try the adaptive non-maximum suppression discussed in the textbook. Finally, you can try an entirely different interest point detection strategy like that of MSER. If you implement an additional interest point detector, you can use it alone or you can take the union of keypoints detected by multiple methods.

Local feature description (get_features.m)

You will implement a SIFT-like local feature as described in the lecture materials and Szeliski 4.1.2. See the placeholder get_features.m 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.
Bells & whistles for local feature description. The simplest thing to do is to experiment with the numerous SIFT parameters: how big should each feature be? How many local cells should it have? How many orientations should each histogram have? Different normalization schemes can have a significant effect, as well. Don't get lost in parameter tuning, though. If your keypoint detector can estimate orientation, your local feature descriptor should be built accordingly so that your pipeline is rotation invariant. Likewise, if you are detecting keypoints at multiple scales, you should build the features at the corresponding scales. You can also try different spatial layouts for your feature (e.g. GLOH) or entirely different features (e.g. local self-similarity).

Feature matching (match_features.m)

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. Simply compute all pairs of distances between all features.
Bells & whistles for local feature description. The "ratio" test is simple and can work surprisingly well, but it will allow some obvious outliers to match. More sophisticated matching schemes involve some form of geometric verification. Such verification can be weak (e.g. is the offset of this match roughly similar to the offset of nearby matches) or stronger (e.g. using RANSAC to estimate explicit transformations for groups of features and rejecting matches that are incompatible with the dominant transformation). Another issue with the baseline matching algorithm is the computational expense of computing distance between all pairs of features. For a reasonable implementation of the base pipeline, this is likely to be the slowest part of the code. There are numerous schemes to try and approximate or accelerate feature matching such as lowering the dimensionality of the features or

using space partitioning data structures to accelerate the nearest neighbor computations. All of these methods are fundamentally lossy, though.

Using the starter code (proj2.m)

The top-level proj2.m script 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. For these two images there is a ground truth evaluation in the starter code, as well. evaluate_correspondence.m will classify each match as correct or incorrect based on similarity to hand-provided matches (run show_ground_truth_corr.m to see the ground truth annotations). The starter code only includes ground truth for this image pair, but you can create additional ground truth matches with collect_ground_truth_corr.m, although it is a tedious process.

As you implement your feature matching pipeline, you should see your performance according to evaluate_correspondence.m increase. Hopefully you find this useful, but don't overfit to these particular (and relatively easy) images. 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 /course/cs143/asgn/proj2/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.

Potentially useful MATLAB functions: *imfilter(), fspecial(), bwconncomp(), colfilt(), sort().*
Forbidden functions you can use for testing, but not in your final code: *extractFeatures(), detectSURFFeatures()*.

More Bells & Whistles (Extra Credit)

In addition to the specific suggestions above for each stage of the pipeline, you can try running a structure from motion algorithm based on the feature matches you find. For example, try configuring the widely used VisualSFM package or Bundler to accept the feature matches you find rather than their default (SIFT-based) matches. If you find many matches between many photos you can achieve a sparse 3d reconstruction of the physical scene. You can then use a stereo algorithm to generate a dense reconstruction. These Bells & Whistles may be quite involved, though, because you'll be dealing with complicated software packages.

Writeup

For this project, and all other projects, you must do a project report . 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 Notre Dame images, 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.jpg instead. A good writeup will assess how important various design decisions were. E.g. by using SIFT-like features instead of normalized patches, I went from 70% good matches to 90% good matches. This is especially important if you did some of the More Bells & Whistles and want extra credit. You should clearly demonstrate how your additions changed the behavior on particular test cases.

.