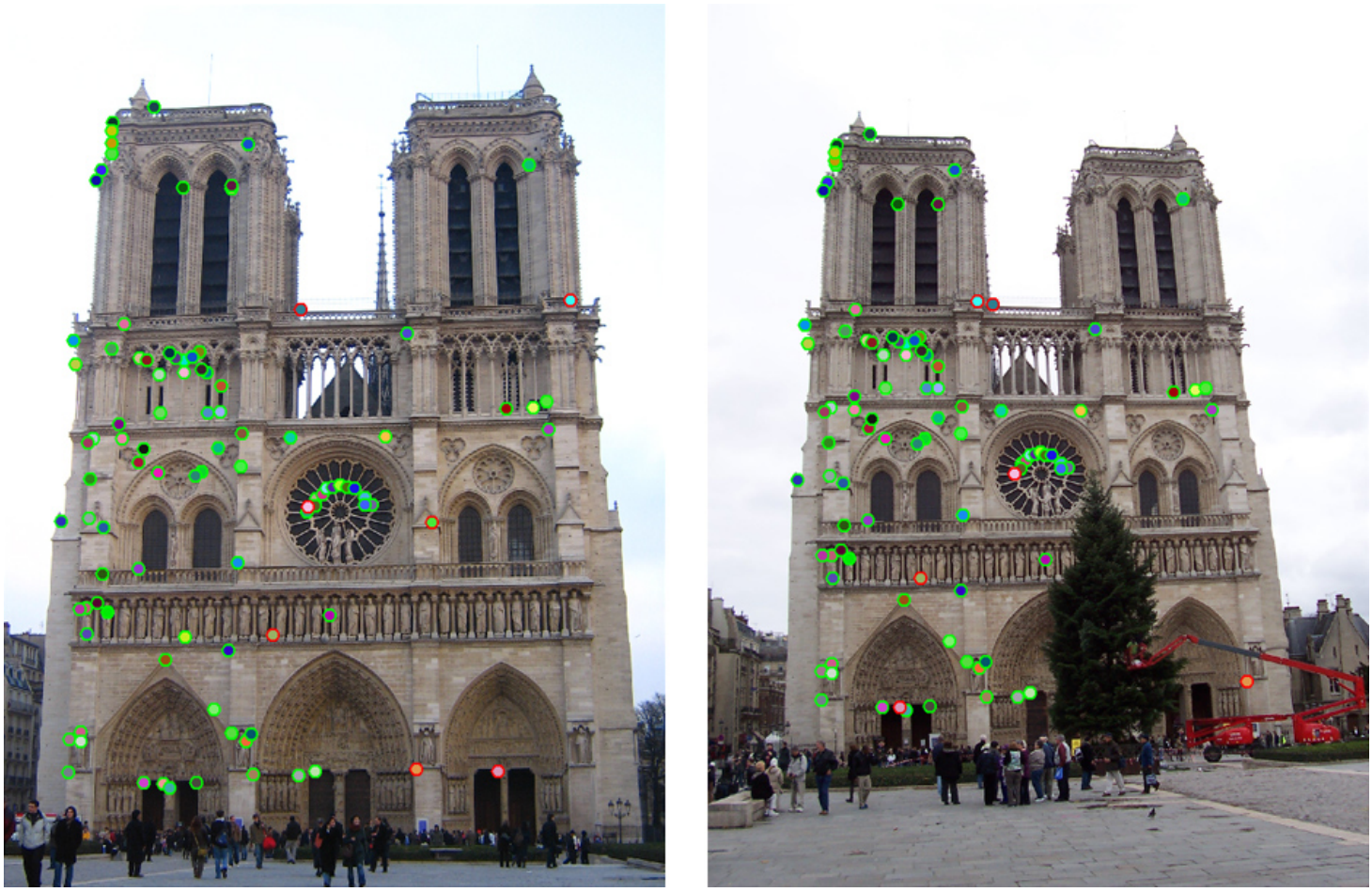
Assignment 3: Local Feature Matching



The top 100 most confident local feature matches from a baseline implementation of Assignment 3. In this case, 93 were correct (highlighted in green) and 7 were incorrect (highlighted in red).

Brief

- Due: 11:55pm on Friday, November 20th, 2015
- Project materials including writeup template **assign3.zip (6.9 MB)**.
- Additional scenes to test on **extra_data.zip (194 MB)**.
- Handin: through **LMS**
- Required files: README, code/, html/, html/index.html

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.

THIS ASSIGNMENT HAS BEEN TAKEN FROM THE JAMES HAYS COMPUTER VISION COURSE. Its excelent resource that allows you understand what limitations for matching are. Please feel free to email me and your TA if you don't understand any part of the assignment.

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.

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.

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 Szeliski 4.1.3. See equation 4.18 in particular. Simply compute all pairs of distances between all features.

Using the starter code (`assign3.m`)

The top-level `assign3.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. The correspondence will be visualized with `show_correspondence.m` and `show_correspondence2.m` (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.m` will classify each match as correct or incorrect based on hand-provided matches (see `show_ground_truth_corr.m` 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 `assign3.m`.You can create additional ground truth matches with `collect_ground_truth_corr.m` (but it's 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 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_data.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.

Potentially useful MATLAB functions: `imfilter(), fspecial(), bwconncomp(), colfilt(), sort(), atan2`.
Forbidden functions you can use for testing, but not in your final code: `extractFeatures(), detectSURFFeatures()`.

Writeup

For this project you must do a project report in HTML. We provide you with a placeholder .html document which you can edit. 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_dots.jpg` or `vis_arrows.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 (these are James Hays's words, but I agree with him that SIFT like features will give you much better matching) 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.

Extra Credit

NOTE: YOU CAN DO ANY OF THE FOLLOWING TASKS, UPTO 10 POINTS.

For all extra credit, be sure to include quantitative analysis showing the impact of the particular method you've implemented. Each item is "up to" some amount of points because trivial implementations may not be worthy of full extra credit

Interest point detection extra credit:

- up to 5 pts: Try detecting keypoints at multiple scales or using a scale selection method to pick the best scale.
- up to 5 pts: Try estimating the orientation of keypoints to make your local features rotation invariant.
- up to 5 pts: Try the adaptive non-maximum suppression discussed in the textbook.
- up to 10 pts: 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 extra credit:

- up to 3 pts: 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.
- up to 5 pts: If your keypoint detector can estimate orientation, your local feature descriptor should be built accordingly so that your pipeline is rotation invariant.
- up to 5 pts: Likewise, if you are detecting keypoints at multiple scales, you should build the features at the corresponding scales.
- up to 5 pts: Try different spatial layouts for your feature (e.g. GLOH).
- up to 10 pts: Try entirely different features (e.g. local self-similarity).

Local feature matching extra credit:

An 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:

- up to 10 pts: Create a lower dimensional descriptor that is still accurate enough. 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 neighbor package to accelerate matching.

Rubric

- +20 pts: Implementation of Harris corner detector in `get_interest_points.m`
- +40 pts: Implementation of SIFT-like local feature in `get_features.m`
- +10 pts: Implementation of "Ratio Test" matching in `match_features.m`
- +20 pts: Writeup with several examples of local feature matching.
- +10 pts: Extra credit (up to ten points) You are welcome to implement any bells & whistles.
- -5*n pts: Lose 5 points for every time (after the first) you do not follow the instructions for the hand in format

Web-Publishing Results

All the results for each project will be put on the course website so that the students can see each other's results. The professor and TA will select "winning" projects that impress us and there will be in class presentations for these projects. If you do not want your results published to the web, you can choose to opt out. If you want to opt out, email henryhu[at]gatech.edu saying so.

Handing in

This is very important as you will lose points if you do not follow instructions. Every time you do not follow instructions, you will lose 5 points. The folder you hand in must contain the following:

- README - text file containing anything about the project that you want to tell the TAs
- code/ - directory containing all your code for this assignment
- html/ - directory containing all your html report for this assignment (including images). Only this folder will be published to the course web page,

so your webpage cannot contain pointers to images in other folders of your handin.

- html/index.html - home page for your results

Hand in your project as a zip file through **lms.itu.edu**.

Credits

THIS ASSIGNMENT IS SLIGHTLY MODIFIED VERSION OF ASSIGNMENT DEVELOPED BY JAMES HAYS