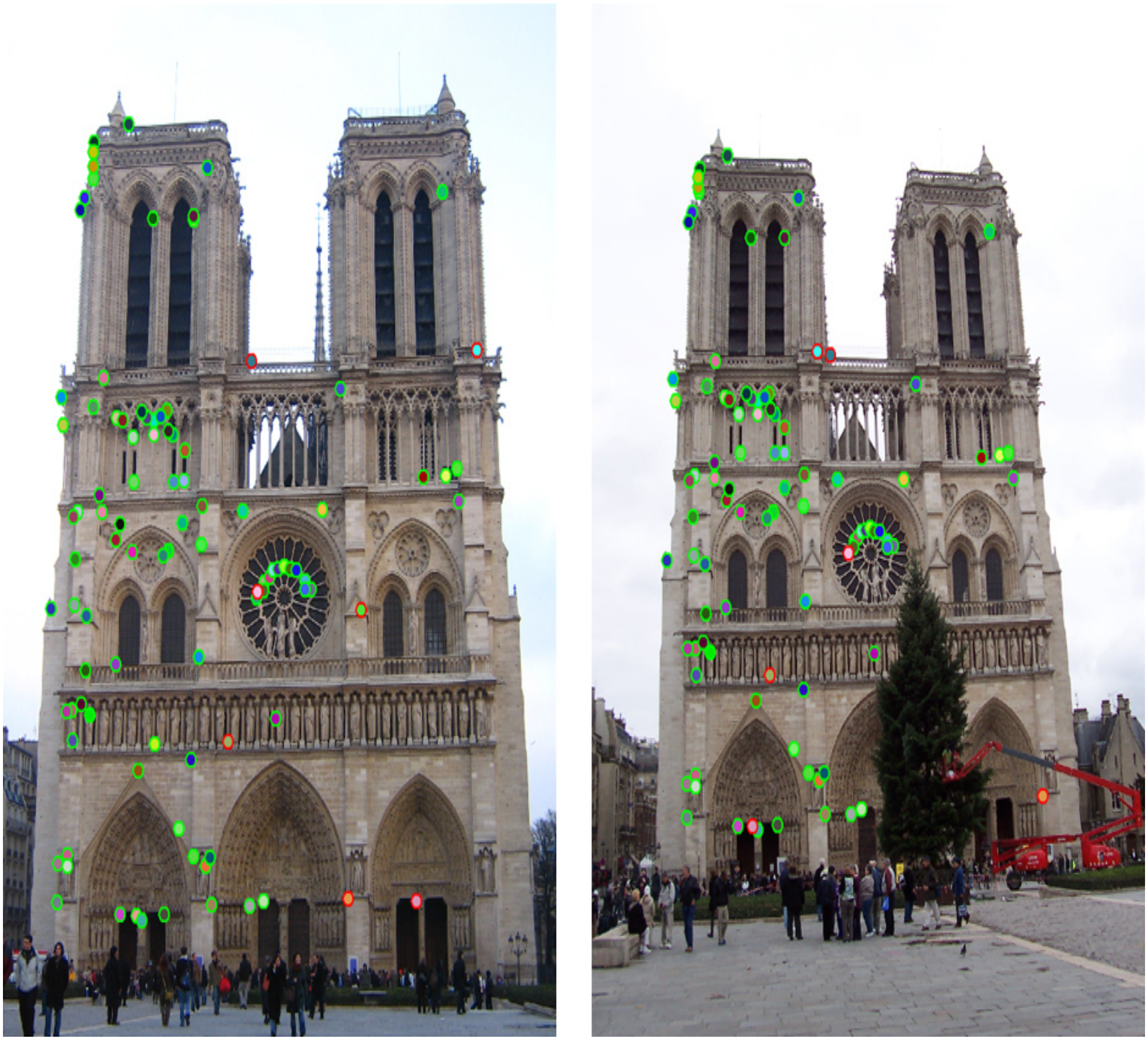
The top 100 most confident local feature matches from a baseline implementation of project 2. In this case, 93 were correct (highlighted in green) and 7 were incorrect (highlighted in red).

Project 2: Local Feature Matching
CSCI 1430: Introduction to Computer Vision (..)

Logistics

- ~~Files: proj2.zip (12 MB) (proj2.zip).~~
- ~~Part 1: Questions~~
    - ~~Questions + template in~~ the zip: questions/
    - Hand-in process: Gradescope (http://www.gradescope.com/) as PDF. Submit anonymous materials please!
    - Due: Friday 08th Feb. 2019, 9pm.
- Part 2: Code
    - Writeup template: In the zip: writeup/
    - Required files: code/, writeup/writeup.pdf

- Hand-in process: Gradescope (http://www.gradescope.com/) as ZIP file created by `createSubmissionZip.py`. Submit anonymous materials please!
- Due: Friday 15th Feb. 2019, 9pm.

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 (http://www.cs.ubc.ca/~lowe/keypoints/); 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: `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. `scipy.spatial.distance()`, which includes functions for measuring distances between vectors. `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. 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.

To define these arguments in Visual Studio Code for debugging, we need to create a Launch Configuration. More information on how to do that is here (https://code.visualstudio.com/docs/editor/debugging). In launch.json, we would add input arguments `args` something like this:

```

{
  "version": "0.2.0",
  "configurations": [
      {
          "name": "Python: Current File (Integrated Terminal)",
          "type": "python",
          "request": "launch",
          "program": "${file}",
          "args": ["-p", "notre_dame"],
          "console": "integratedTerminal"
      },
      ...
    ]
}
```

Requirements / Rubric

If your implementation reaches 70% accuracy on the *most confident* 100 correspondences in 'matches' for the Notre Dame 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 Gradescope autograder will stop executing `proj2_averageAccuracy.py` after 20 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 Gradescope autograder runs on an Amazon EC2 cluster with a modern desktop CPU about as powerful as your laptop. It does not have a GPU. It has 768MB memory, and going over this may terminate your program unexpectedly (!). It uses the same Python virtual environment as the department machines.

- +20 pts: Written questions.
- +25 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()`
- +05 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 (`createSubmissionZip.py` will compile and include it for you).

- 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 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.jpg`.
- *We conduct anonymous TA grading, so please don't include your name or ID in your writeup or code.*

- +10 pts: Extra credit (max 10 pts total).
    - Detection:
        - Up to 5 pts: Detect keypoints at multiple scales or use a scale-selection method to pick the best scale.
        - Up to 5 pts: Use adaptive non-maximum suppression as discussed in the textbook.
        - Up to 10 pts: Use a different interest point detection strategy like MSER. Use it alone, or take the union of multiple methods.
    - Description:
        - Up to 3 pts: Experiment with SIFT parameters: window size, number of local cells, orientation bins, different normalization schemes.
        - Up to 5 pts: Estimate feature orientation.
        - Up to 5 pts: Multi-scale descriptor. If you are detecting keypoints at multiple scales, you should build the features at the corresponding scales, too.
        - Up to 5 pts: Different spatial layouts for your feature (e.g., GLOH).
        - Up to 10 pts: Entirely different features (e.g., local self-similarity).
    - Matching; the baseline matching algorithm is computationally expensive, and will likely be the slowest part of your code. Consider approximating or accelerating feature matching:
        - Up to 10 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 5 pts: Use a space partitioning data structure like a kd-tree or some third party approximate nearest neighbor package to accelerate matching.
- -05*n pts: Where n is the number of times that you do not follow the instructions.

Accuracy Competition

Independent of your grade, we will average your top 100 accuracy across the three image pairs—Notre Dame de Paris, Mount Rushmore, and Gaudi's Episcopal Palace. The top ranked submissions will earn a (probably tasty) prize, and `eternal' recognition below. Furthermore, Gradescope offers an anonymous leaderboard for a little friendly competition among your peers.

- 2019 Spring:
    - *Highest Average Accuracy:* ????—xx.xx%.
    - *Gaudi's Choice Award (Episcopal Palace only):* ????—xx.xx%
- 2017 Fall:
    - *Highest Average Accuracy:* Henry Stone—80.33%.

- *Gaudi's Choice Award (Episcopal Palace only):* Henry Stone—46%
- 2017 Spring:
  - *Highest Average Accuracy:* Katya Schwiegershausen—66%.
  - *Gaudi's Choice Award (Episcopal Palace only):* Spencer Boyum—25%

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. *Our solution* generates around 750–1000 feature points for each image. 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).

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.

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. We provide additional image pairs in extra_data.zip (194 MB) (http://cs.brown.edu/courses/csci1430/proj2/extra_data.zip), 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 Mount Rushmore and Episcopal Gaudi.

Credits

Python port by Anna Sabel and Jamie DeMaria. Assignment originally developed by James Hays.