ENGG5104 – Image Processing and Computer Vision – Spring 2021

Assignment 3 — Local Feature Matching

Deadline: April 1, 2021 (Thursday) 23:59

1 Backgound

The goal of this assignment is to implement a local feature-matching algorithm. The suggested pipeline is a simplified version of the famous SIFT. The matching pipeline should work for instance-level matching - multiple views of the same physical scene.

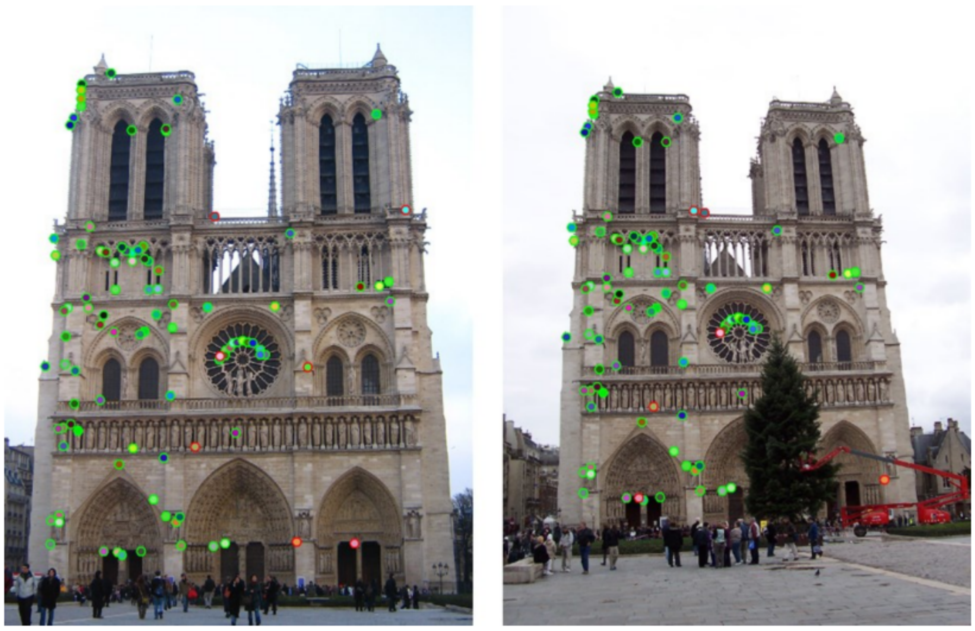


Figure 1: Demonstration of Feature Matching. In this case, 93 are correct (highlighted in **green**) and 7 are incorrect (highlighted in **red**).

2 Algorithms

In this project, you need to finish a local feature matching algorithm. The algorithm takes 2 images of the same scene as inputs and outputs corresponding matched feature points. The pipeline contains 3 major steps:

(1) **Interest point detection**. This step is to detect several special points (feature points) in an image, which are highly discriminative or give important clues about scenes or objects, e.g., Harris corner points in Assignment 1.

(2) **Local feature description**. This step is to extract a feature vector for each detected interest points in the last step. These feature vectors can then be easily compared for similarity measure between feature points.

(3) **Feature matching**. This step is to find the correct correspondences between interest points in these two input images, based on calculated feature vectors in (1) and (2).

2.1 Interest point detection

Function: `get_interest_points()`

You will implement the Harris corner detector as described in **Assignment 1**. The skeleton code gives some additional suggestions. You do not need to worry about scale invariance or key-point orientation estimation for your baseline Harris corner detector.

Hint: For more details and a pseudocode, please refer to **Chapter 4.1.1** and **Algorithm 4.1** of Szeliski's textbook [1].

2.2 Local feature description

Function: `get_features()`

This part is to extract a feature representation for each key-point. You will implement a SIFT-like local feature. (See `get_features()` for more details) The feature is a gradient orientation histogram calculated from a $n \times n$ window around each interest point. n is better to be multiples of 4. Typically, we choose $n = 16$.

A simplified version contains the following steps.

Step 1: We simplify this part by fix Gaussian kernel standard deviation to $\sigma = 1$. Then convolve this Gaussian kernel with the whole image. You can use `cv2.GaussianBlur`.

$$L(x, y) = \text{Gaussian}(x, y, \sigma) * I(x, y) \tag{1}$$

Step2: Calculate the image gradient of the Gaussian blurred image $L(x, y)$. The gradients are denoted as G_x and G_y. You can also use OpenCV or Numpy built-in functions to do this, e.g., `cv2.spatialGradient`)

$$G_x(x, y) = L(x + 1, y) - L(x, y) \tag{2}$$
$$G_y(x, y) = L(x, y + 1) - L(x, y) \tag{3}$$

Step3: Calculate the magnitude and orientation of each pixel.

$$\text{mag} = \sqrt{G_x^2 + G_y^2} \tag{4}$$
$$\text{orient} = \arctan(\frac{G_y}{G_x}) \tag{5}$$

Step 4: For each detected interest point, we consider $n \times n$ window centered at this point. Like the Figure 2 below, we choose $n = 16$, and divide this $n \times n$ window into 4×4 cells, each with size $\frac{n}{4} \times \frac{n}{4}$ (i.e., 4×4 here).

The magnitudes of the $n \times n$ pixels are further weighted by a 2D Gaussian function with kernel size equal to n and standard deviation to be $\frac{n}{2}$ (you can use `cv2.getGaussianKernel`).

Step 5: We then consider each cell of the window. For the above example, $n = 16$, and cell is 4×4 in pixels. We first cast the orientations of each pixel (inside this cell) into 8 bins: $[0°, 45°), [45°, 90°) \cdots [315°, 360°)$ as Figure 3.

Then for each bin X, we accumulate the Gaussian weighted gradient magnitude of those pixels (from **Step 4**), whose orientations belong to bin X.

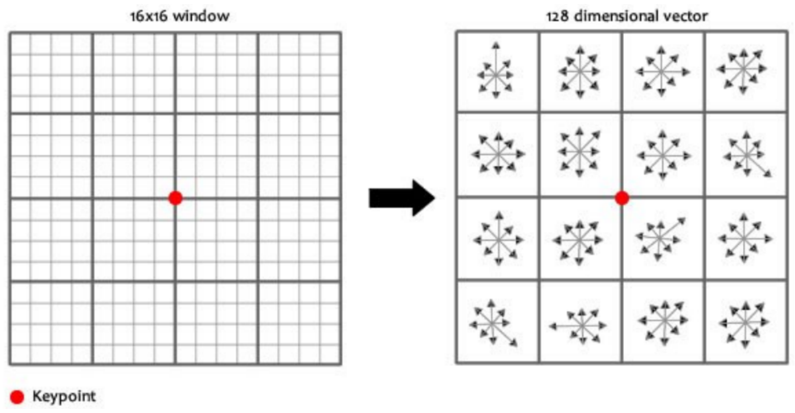


Figure 2: An example with $n = 16$ and 4×4 cells.

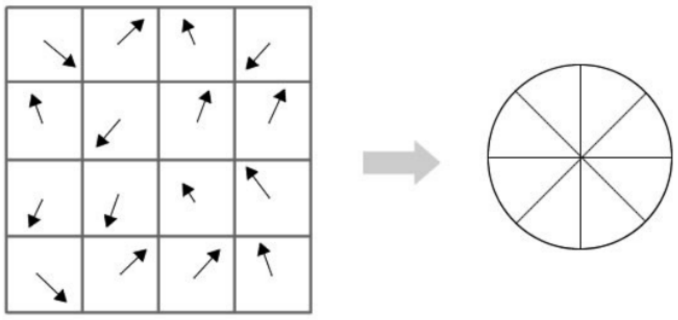


Figure 3: Bins for orientations.

Step 6: After previous steps, we have **8** floating values of accumulated gradient magnitudes for each cell. And for each interest point, we have $\mathbf{4} \times \mathbf{4} = \mathbf{16}$ cells. So totally we have $\mathbf{16} \times \mathbf{8} = \mathbf{128}$ values. We then concatenate these textbf 128 values into one vector and then normalize the vector to make its length (L2-norm) to be 1. The resulting vector is our local feature vector (descriptor).

Hint: For more details, please refer to **Chapter 4.1.2** (Page 223) from Szeliski's textbook [1].

2.3 Feature matching

Function: `match_features()`

You will implement the "**ratio test**" (i.e., "**nearest neighbor distance ratio test**") method of matching local features.In this part, given interest points (with their feature vectors) for two images, we want to match these two sets of points. The matching confidence can be evaluated using the ratio test.

Step 1: Compute the feature distances (e.g., Euclidean distance) between the key-points. Assume we have **m** interest points for image A, and **n** points for image B. Then we can calculate $m \times n$ distances in total.

Step 2: Compute the confidence for each point pairs with the ratio test.

For each point in image A, we compute:

$$\text{ratio} = \frac{\text{score of the best feature match}}{\text{score of the second best feature match}} \tag{6}$$

That is the same as:

$$\text{ratio} = \frac{\text{nearest distance}}{\text{second nearest distance}} \tag{7}$$

It is easy to understand that if the value is smaller, the matching is better.The matching confidence should be the inverse of this ratio:

$$\text{conf} = \frac{1}{\text{ratio}} \tag{8}$$

Step 3: Set a threshold value according to your experiment. Find all matching pairs whose confidence exceeds the threshold. These selected matching pairs are the final results.

Hint: For more details, please refer to **Chapter 4.1.3** from Szeliski's textbook [1] and **Eq. 4.18** in particular.

3 Todo List

We provide you the following files as the skeleton code:

In folder `‘code/’`:

`proj2.py` - the main function of this project

`match_functions.py` - contains the functions you need to implement

`utils.py` - auxiliary functions for visualization and evaluation

We also provide some test images with ground truth correspondences:

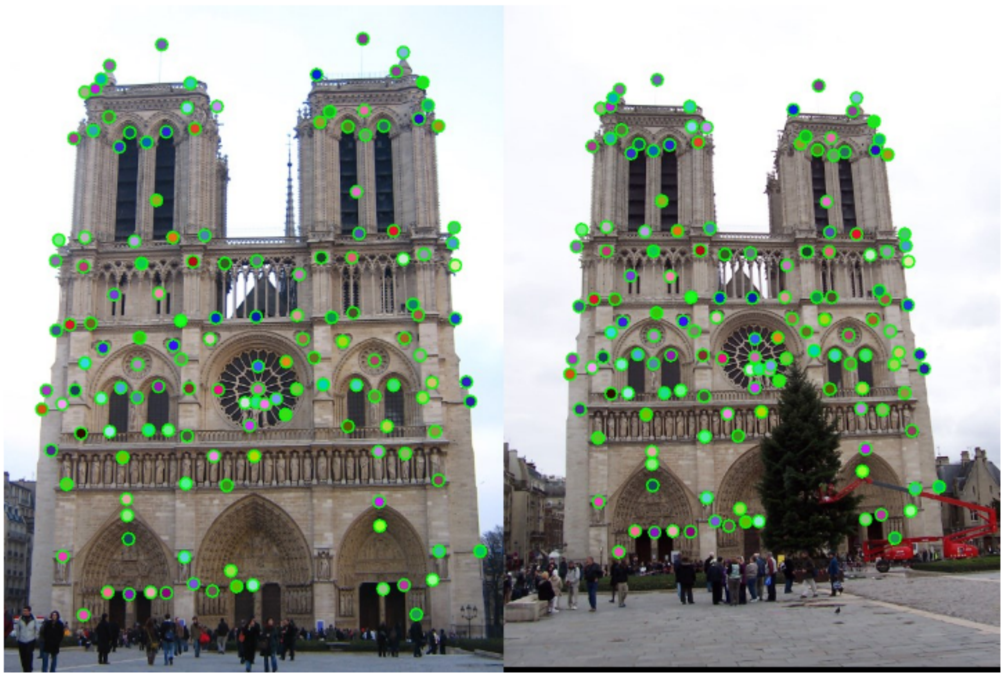
In folder `‘data/’`: `Episcopal Gaudi, Mount Rushmore, Notre Dame`

You need to add your own code to complete the files.

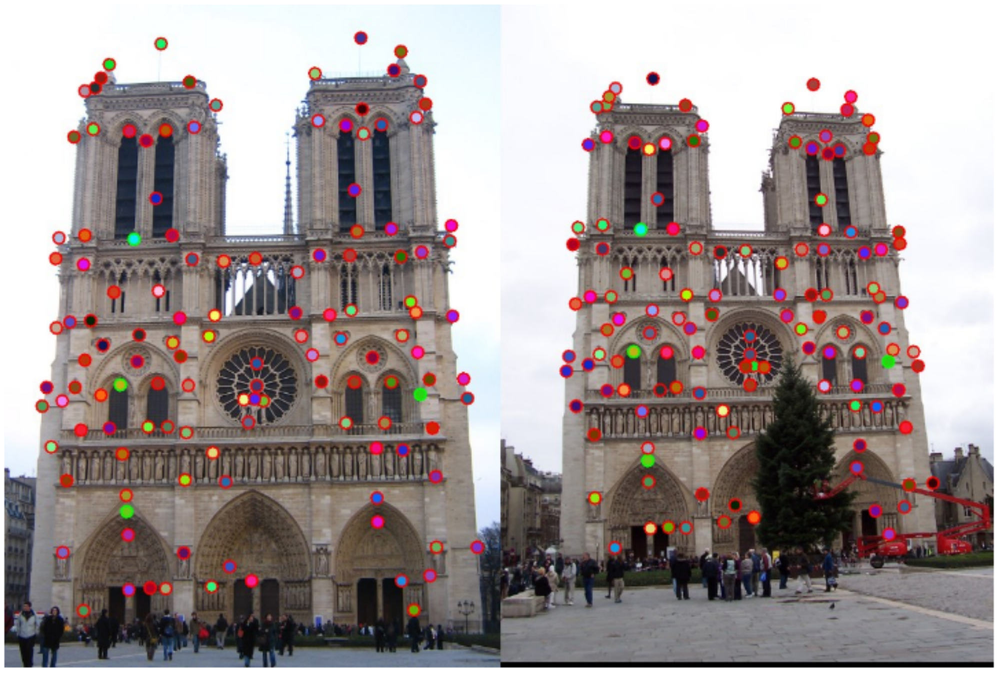
4 Hints

Here are some hints about the implementation details:

- First, use `cheat_interest_points()` instead of `get_interest_point()`. 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. By using the provided visualization tool, you will get:
- Second, implement `match_features()`. Accuracy should still be near zero because the features are random.
- Third, change `get_features()` to cut out image patches. Accuracy should increase to about 40% on the Notre Dame pair if you're using 16×16 (256 dimensional) patches as your feature. Accuracy on the other test cases will be lower (Mount Rushmore 25%, Episcopal Gaudi 7%).



(a) Use `cheat_interest_points()` and ground truth matching. Green circles mean correct. The circles with the same color inside indicate matched pairs.



(b) Use `cheat_interest_points()` and random matching. Green circles mean correct. Red circles mean wrong. Only 4 is correct here, accuracy is near 0%.

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.

- Fourth, finish `get_features()` by implementing a sift-life 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 skeleton 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.

- Fifth, stop cheating 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.

5 General Specification

You are required to complete all functionalities according to the specification with Python. Skeleton codes are given. Do not change the files' name or the functions' name. You are **NOT** allowed to implement your program in another language (e.g. Matlab/C++) and then initiate system calls or external library calls in Python.

The folder's name, filename, and code can not contain any Chinese.

- **Restrictions on using Python**
  Besides the given imported modules in the skeleton code, **no more** other modules could be imported.
- **NO PLAGIARISM!**
  You are free to design your algorithm and code your implementation, but you should not "borrow" codes from your classmates. If you use an algorithm or code snippet that is publicly available or use codes from your classmates or friends, be sure to DECLARE it in the comments of your program. Failure to comply will be considered as plagiarism.

6 Submission Guidelines

The folder you hand in must contain the following:

- **README.txt** - containing anything about the project that you want to tell the TAs, including a brief introduction of the usage of the code.
- **code/** - directory containing all your code for this assignment, which is expected to have three .py files as shown in the skeleton.
- **report.pdf** - describing your algorithm and any decisions you made to write your algorithms. Show and discuss your results in your report. Discuss algorithms' efficiency.
- **data/** - directory containing all the given data as shown in the skeleton.

Rename the folder as <your student ID>-Asgn3, and compress it into <your student ID>-Asgn3.zip, and upload it to the blackboard system. (For example, 1155123456-Asgn3/ and 1155123456-Asgn3.zip, pay attention to the name.)

Please read the guidelines CAREFULLY. If you fail to meet the deadline because of a submission problem on your side, marks will still be deducted.

The late submission policy is as follows:

- 1 day late: -20 marks

- 2 days late: -40 marks

- 3 days late: -100 marks

Pay attention to the format before. 10% deduction for every wrong format (filename, function name, etc.).

7 Score

The maximum score for this assignment is 100:

- 25%: The implementation of `get_interest_points()`.
- 40%: The implementation of `get_features()`.
- 15%: The implementation "Ratio Test" of `match_features()`.
- 20%: PDF report with at least 5 pairs of results (visualization, accuracy, running time, etc.). At least 1 pair of your own photos.

8 Reference

[1] Computer Vision: Algorithms and Applications. Richard Szeliski. Download here.

[2] SIFT. http://www.cs.ubc.ca/ lowe/keypoints/.

[3] MSER. http://www.cs.ubc.ca/ lowe/papers/07forssen.pdf.