ENGG 5104 Image Processing and Computer Vision

Assignment 2 – Local Feature Matching

Due Date: 11:59pm on Tuesday, February 21st, 2017

I. Background

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

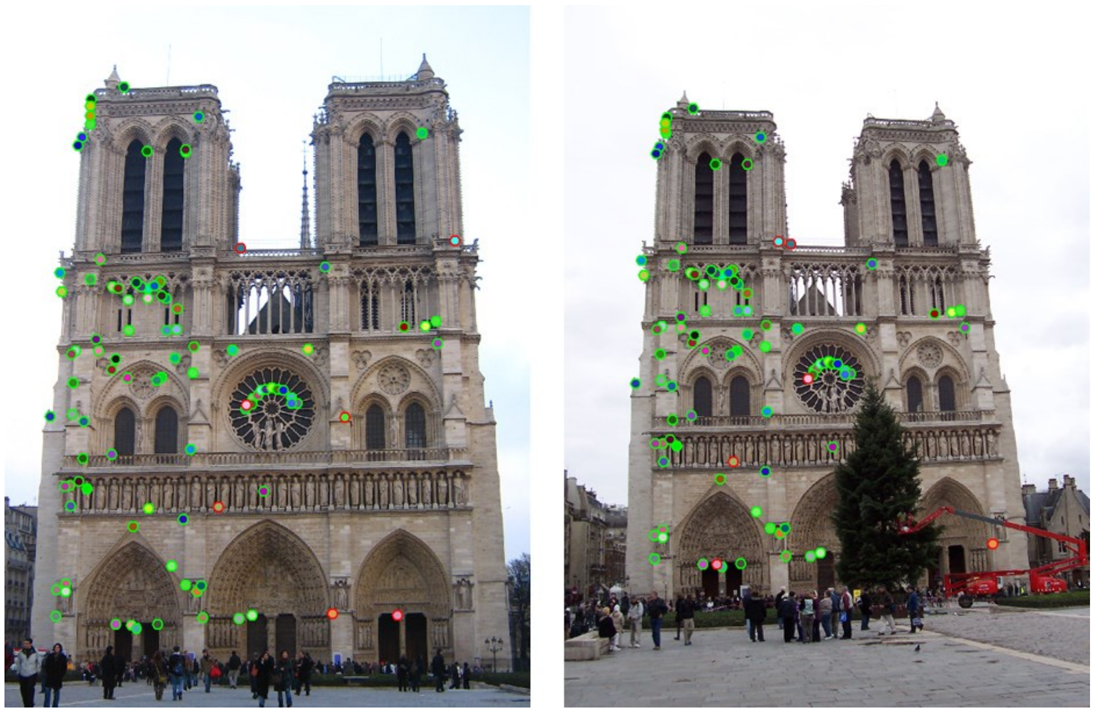


Figure 1. Demonstration of Feature Matching. In this case, 93 are correct (highlighted in **green**) and 7 are incorrect (highlighted in **red**).

II. Details

In this project, you need to implement 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 the image, which are highly discriminative or give important clues about scene or objects, eg, Harris corner points in Assignment 1.

(2) **Local feature description** This step is to extract a feature vector for each detected interest points in Step (1). 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 two input images, based on calculated feature vectors in Step (1)-(2).

Algorithms

Interest point detection - `get_interest_points()`

You will implement the Harris corner detector as described in **Assignment 1**. The starter 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].

Local feature description - `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`

$$L(x, y) = Gaussian(x, y, \sigma) * I(x, y)$$

Step 2: 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`)

$$G_x(x, y) = L(x + 1, y) - L(x, y)$$

$$G_y(x, y) = L(x, y + 1) - L(x, y)$$

Step 3: Calculate the magnitude and orientation of each pixel.

$$\text{mag} = \sqrt{G_x^2 + G_y^2}$$

$$\text{orient} = \arctan(\frac{G_y}{G_x})$$

Step 4: For each detected interest point, we consider $n \times n$ window centered at this point. Like the figure 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`).

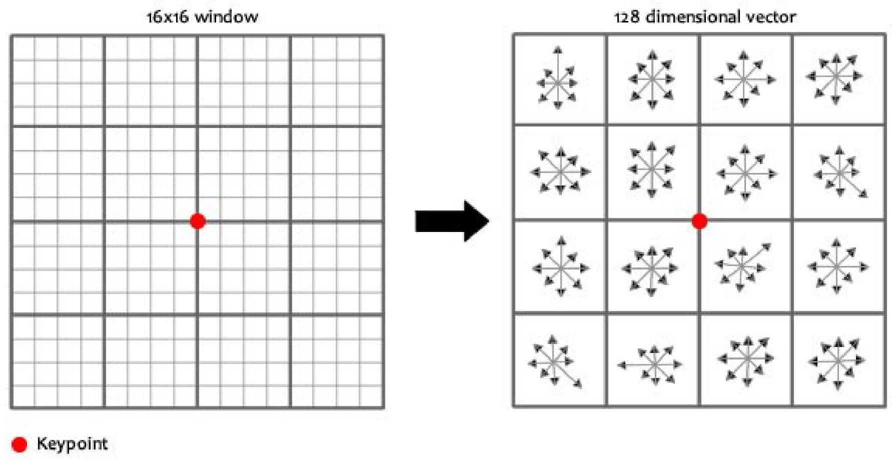
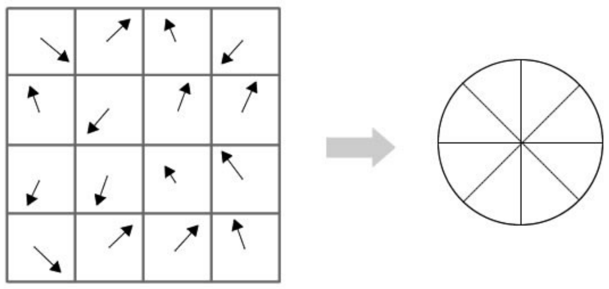


Figure. An example with n=16 and 4 × 4 cells.

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°) … [315°, 360°).

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



Step 6: After previous steps, we have **8** floating values of accumulated gradient magnitudes for each cell. And for each interest point, we have $\mathbf{4 \times 4 = 16}$ cells. So totally we have $\mathbf{16 \times 8 = 128}$ values. We then concatenate these **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].

Feature matching - `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 (for example, 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 **ratio test**.

For each point in image A, we compute:

ratio = (score of the best feature match) / (score of the second best feature match)

That is the same as: ratio = (nearest distance) / (second nearest distance).

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:

$$\text{conf} = 1/\text{ratio}$$

Step 3: Set a threshold value according to your experiment. Find all matching pairs whose confidence exceed 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 **Equ. 4.18** in particular.

TODOs

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 correspondances:

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

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

Useful Hints

Here are some hints about the implementation detail (please read this part carefully):

1) 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 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:

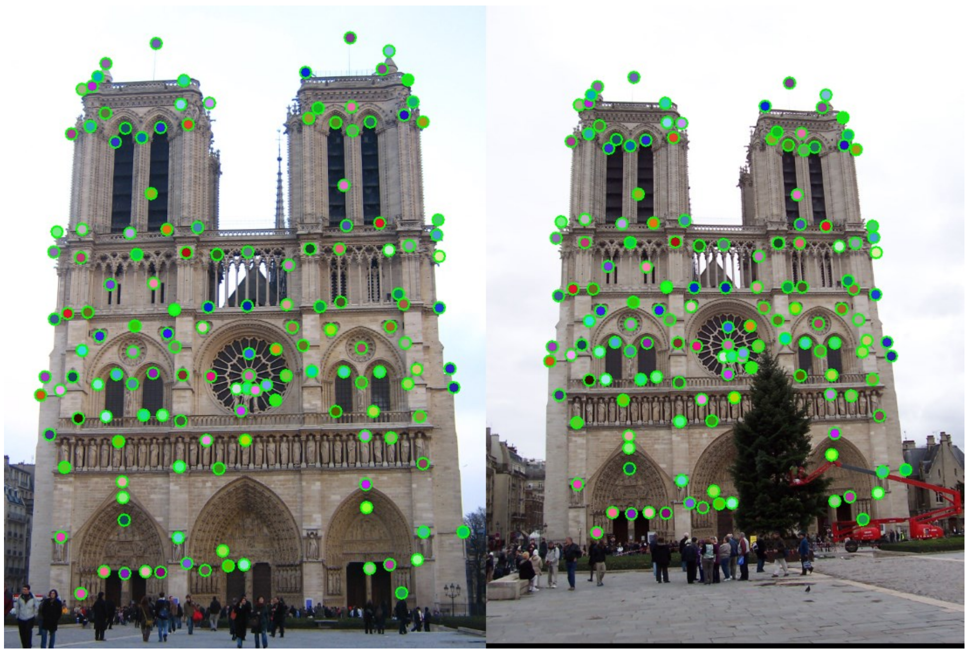


Figure. Use `cheat_interest_points()` and <u>ground truth</u> matching. Green circles mean correct. The circles with the same color inside indicate matched pairs.

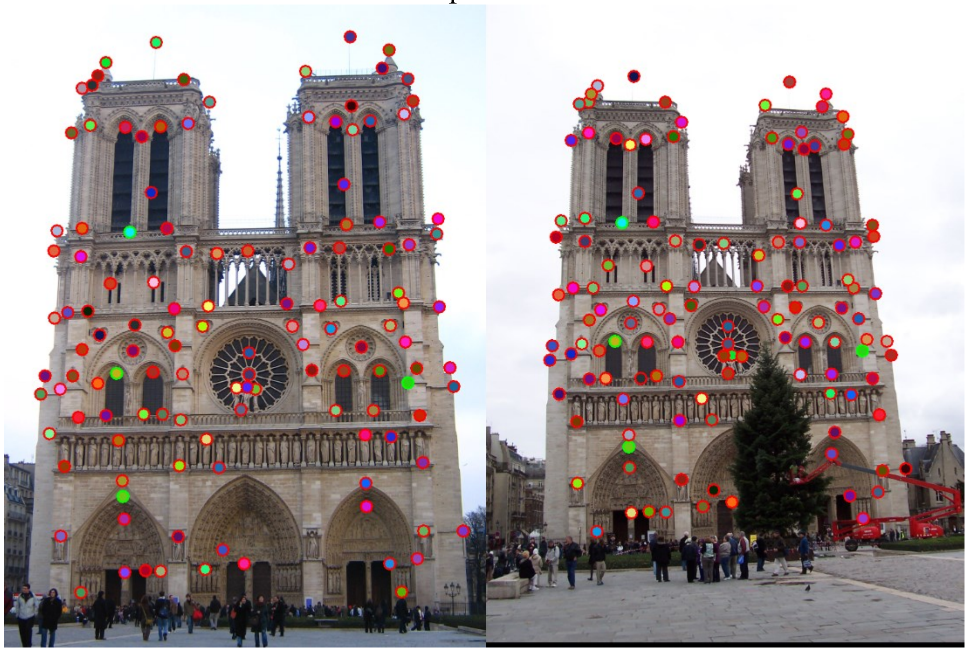


Figure. Use `cheat_interest_points()` and <u>random</u> matching. Green circles mean correct. Red circles mean wrong. Only 4 is correct here, accuracy is near 0%.

2) Second, implement `match_features()`. Accuracy should still be near zero because the features are random.

3) Third, 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 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.

4) 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 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.

5) 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.

III. Submission

a. Upload package

1) Create a folder with name:

   `<your student ID>-Asgn2 (e.g. 11550xxxxx-Asgn2\)`

2) Readme file containing anything about the project that you want to tell the TAs, including brief introduction of the usage of the codes

   `<your student ID>-Asgn2\README.txt`

3) Place all your code in subfolder `code\`

   `<your student ID>-Asgn2\code\`

4) A report in **HTML**. In the report, describe your algorithm and any decisions you made to write your algorithms. Show and discuss your results in your report. Discuss algorithms' efficiency and highlight all extra credits you did.

   Place all your html report in subfolder `html\`. The home page should be `index.html`

   `<your student ID>-Asgn2\html\index.html`

5) Compress the folder into <*your student ID*>**-Asgn2.zip**, and upload to e-learn system.

b. Hand-in via CUHK e-learn system

1) Go to http://elearning.cuhk.edu.hk/
2) Go to: 2016R2-ENGG5104 : Image Processing and Computer Vision
3) Go to: **Assignments** tab
4) Click **Assignment 2**→**Browse My Computer** →Select your package → Click **Submit.**

IV. Scoring & Extra Bonus

The maximum score for this assignment is 100.

a. Rubric

1) 20%: The implementation of `get_interest_points()`.
2) 35%: The implementation of `get_features()`.
3) 10%: The implementation "Ratio Test" of `match_features()`.
4) 15%: HTML report with at least 5 pairs of results (visualization, accuracy, running time, etc.). At least 1 pair of your own photos.

b. Extra Credit

Highlight your bonus part clearly in your report. You can consider the following ideas:

Bonus 1: 5%: Try detecting key-points at multiple scales or using a scale selection method to pick the best scale. Please refer to original SIFT paper for details [2].

Bonus 2: 5%: Try estimating the orientation of key-points to make your local features rotation invariant. Please refer to original SIFT paper for details [2].

Bonus 3: 10%: Try an entirely different interest point detection strategy like that of MSER [3].

If you implement your own algorithms, you can get full mark. You can also test on OpenCV built-in functions for interest point detection, only 3% marks will be given.

Bonus 4: 5%: Experiment with at least 3 different SIFT parameters: how big should each window be (n)? How many local cells should it have in each window? How many orientation bins should each histogram have? Different normalization schemes can have a significant effect, as well. Compare and show different results.

Bonus 5: 10%: 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. You can use 3rd party libraries for PCA. Compare the accuracy and running time with this dimension change.

Bonus 5: 5%: Use a space partitioning data structure like a KD-tree or some approximate nearest neighbor algorithm to accelerate matching. You can use 3rd party libraries for these data structures.

V. Other Remarks

a. 20% off from each extra late day.

b. The assignment is to be done **INDIVIDUALLY** on your own.

c. You are encouraged to use methods in related academic papers.

d. Absolutely **NO sharing or copying** code! **NO sharing or copying** reports! Offender will be given a failure grade and the case will be reported to the faculty.

VI. 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.