

Homework Set 2:

Local Feature Detection and Matching

Instructions

- This homework contains two parts, i.e., a set of questions (20 pts, in Section 1) and a programming project (80 pts, in Section 2).
- Upload a zipped file named as `id_yourname.zip` via the [Beihang SPOC platform](#), including
 - (1) an electronic version (saved as `id_name.pdf`, can be printed or handwritten) of your report, it includes the answers of the questions in Section 1; and describes the algorithm process, shows the results and discussions (if required) in Section 2.
 - (2) a folder (named as `id_name`) includes the codes and results. You may use a [Jupyter Notebook](#) file (saved as `id_name.ipynb`) instead.

For more details about python and Jupyter Notebook, as well as some useful packages (such as `numpy` and `matplotlib`), please check the following [link](#).
- The deadline is 23:59 pm, May 15th, 2024.

1 Questions (30 pts)

1.1 (10 pts) Imagine we wished to find points in one image which matched to the same world point in another image – so-called feature point correspondence matching. We are tasked with designing an image feature point algorithm which could match world points in the following three pairs of images:



Figure 1: These three pairs of images are in the directory `data/matching`.

- (a) Please use the included python script `plot_corners.py` to find corners using Harris corner detection for each image above.
- (b) For each pair, discuss the differences in the returned corners (if any), and what aspects of the images may have caused these differences. Then discuss what real world challenges exist in matching features between images.

1.2 (8 pts) Distance metrics for feature matching.

- (a) Explain the differences between the geometric interpretations of the Euclidean distance and the cosine similarity metrics. What does this tell us about when each should be used?
- (b) Given a distance metric, what is a good method for feature descriptor matching and why?

1.3 (12 pts) Given a point $\mathbf{x} = (u, v)$ in the image plane, and a vector $\mathbf{n} = (a, b)$, any point \mathbf{y} following the equation $\mathbf{y} = \lambda\mathbf{n} + \mathbf{x}, \forall \lambda \in \mathbf{R}$, defines a line ℓ . Now given a homography matrix

$$\mathbf{H} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}, \quad (1)$$

- (a) Calculate the new line ℓ' after transforming the line ℓ by the homography \mathbf{H} , *i.e.* how \mathbf{y}' will be after transforming from \mathbf{y} . Note that \mathbf{y}' is also in the image plane.
- (b) Discuss in which case the line ℓ' will converge to a vanishing point, and calculate the vanishing point \mathbf{y}'_v .
- (c) Discuss in which case the line ℓ' will not converge to a vanishing point.

2 Programming Project (70 pts)

2.1 Overview

We will create a local feature matching algorithm and attempt to match multiple views of real-world scenes. We will implement the Harris corner detection and a simplified version of **SIFT**; however, you are encouraged to experiment with more sophisticated algorithms and the later on image alignment!

Task: Implement three major steps of local feature matching:

- **Detection** in the `get_interest_points` function in `student.py`. Please implement the Harris corner detector. You do not need to worry about scale invariance or keypoint orientation for your Harris corner detector.
- **Description** in the `get_features` function, also in `student.py`. Please implement a *SIFT-like* local feature descriptor. *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.

Potentially useful functions: `zip()`, `skimage.measure.regionprops()`, `numpy.arctan2()`, `skimage.feature.peak_local_max()`, any existing filter function.

Potentially useful libraries: `skimage.filters.x()` or `scipy.ndimage.filters.x()`, which provide many pre-written image filtering functions. `numpy.gradient()`, which provides a more sophisticated estimate of derivatives. `numpy.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 many 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. Any functions that compute nearest neighbor ratios for you, such as `sklearn.neighbors.NearestNeighbors()`, `scipy.spatial.distance.cdist()` and any other functions that compute the distance between arrays of vectors (use the guide we've provided to implement your own distance function!). If you are unsure about a function, please ask.

2.2 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.

2.3 Requirements/Rubric

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`. We have included this function in the starter code for you so you can measure your own accuracy.

- (a) **(15 pts)** Implementation of Harris corner detector in `get_interest_points()`.
- (b) **(15 pts)** Implementation of SIFT-like local feature in `get_features()`.
- (c) **(10 pts)** Implementation of “Ratio Test” matching in `match_features()`.
- (d) **(10 pts)** Report:
 - (i) Please describe your process and algorithm, show your results, describe any extra requirement, and tell us any other information you feel is relevant.
 - (ii) Quantitatively compare the impact of the method you've implemented. For example, using SIFT-like descriptors instead of normalized patches increased our performance from 50% good matches to 70% good matches. Please include the performance improvement for any extra component.
 - (iii) 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_matches.jpg`
- (e) **(20 pts)** Extra requirements (max 20 pts in total, you may not all requirements).
 - (i) **(5 pts)** Detection: detect keypoints at multiple scales or use a scale-selection method to pick the best scale.
 - (ii) **(5 pts)** Description (finishing either requirement will achieve full credit).
 - 1) Estimate feature orientation.
 - 2) Multi-scale descriptor. If your are detecting keypoints at multiple scales, you should build the features at the corresponding scales, too.
 - (iii) **(5 pts)** Matching (finishing either requirement will achieve full credit): The baseline matching algorithm is computationally expensive, and will likely be the slowest part of your code. Consider approximating or accelerating feature matching.
 - 1) 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 set of your local descriptors from many images.

- 2) Use a space partitioning data structure like a KD-tree or some third party approximate nearest neighbor package to accelerate matching.
- (iv) **(10 pts)** Remove outlier matches if exist. You may use RANSAC together with homography estimation, and then report the output homography matrix for each test image pair.

Time limit: The time limit of executing your code is 20 minutes, after which you will receive a maximum of 35 pts. You must write efficient code.

- Hint 1: Use ‘steerable’ (oriented) filters to build the descriptor.
- Hint 2: Use matrix multiplication for feature descriptor matching.

2.4 An Implementation Strategy

1. Use `cheat_interest_points()` instead of `get_interest_points()`. At this point, just work with the Notre Dame image pair (the cheat points for Mt. Rushmore aren’t very good and the Episcopal Palace is difficult to feature match). 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. To do this, you’ll need to calculate the distance between all pairs of features across the two images (much like `scipy.spatial.distance.cdist()`). While this could be written using a series of `for` loops, we’re asking you to write a matrix implementation using `numpy`. Make sure to also return proper confidence values for each match. Your most confident matches should be highest values.
3. Change `get_features()` to cut out image patches. Accuracy should increase to $\sim 60 - 70\%$ 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 (especially using the bad interest points from `cheat_interest_points()`). 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. If your accuracy doesn’t increase (or even decreases a little bit), don’t worry because normalized patches are actually a good feature descriptor for the Notre Dame image pair. The difference will become apparent when you implement `get_interest_points()` and can test on the Mt. Rushmore pair.
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 $300 \sim 700$ feature points for each image.

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.

2.5 Notes

1. `main.py` handles files, visualization, and evaluation, and calls placeholders of the three functions to implement. For the most part you will only be working in `student.py`.
2. 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 changing the `-p` argument you pass to `main.py`.
3. 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.
4. You will likely need to do extra requirements to get high accuracy on Episcopal Gaudi.
5. Compute/memory limit: If you are concerned about the memory usage of your program, you can check it by running `python memusecheck` in your code directory.

Feedback? (Optional)

Please help us make the course better. If you have any feedback for this assignment, we'd love to hear it!