Project 2: SIFT Local Feature Matching

CS 6476

Fall 2024

Logistics

- Due: Check Canvas for up to date information
- Hand-in: through Gradescope

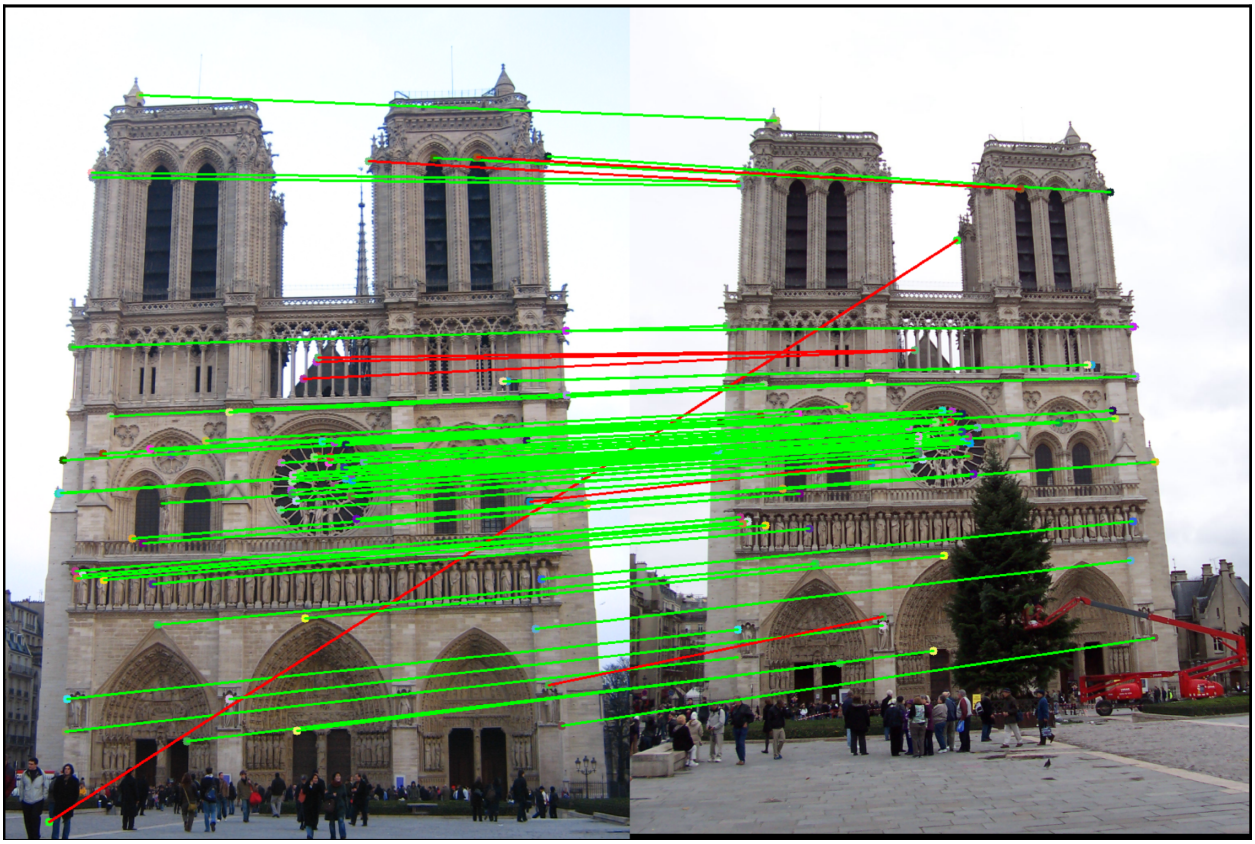

Figure 1: The top 100 most confident local feature matches from a baseline implementation of project 2. In this case, 89 were correct (lines shown in green), and 11 were incorrect (lines shown in red).

Overview

The goal of this assignment is to create a local feature matching algorithm using techniques described in Szeliski chapter 7.1. 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.

For this project, you need to implement the three major steps of a local feature matching algorithm (detecting interest points, creating local feature descriptors, and matching feature vectors). We'll implement two versions of the local feature descriptor, and the code is organized as follows:

- Interest point detection in `part1_harris_corner.py` (see Szeliski 7.1.1)
- Local feature description with a simple normalized patch feature in `part2_patch_descriptor.py` (see Szeliski 7.1.2)
- Feature matching in `part3_feature_matching.py` (see Szeliski 7.1.3)
- Local feature description with the SIFT feature in `part4_sift_descriptor.py` (see Szeliski 7.1.2)

Setup

1. Install Jupyter extension inside VSCode
2. Open `project-2.ipynb` inside VSCode and use "Select Kernel" to re-use the conda environment for Assignment #1.
3. After implementing all functions, ensure that all sanity checks are passing by running `pytest tests` inside the main folder.
4. Generate the zip folder for the code portion of your submission once you've finished the project using `python zip_submission.py --gt_username <your_gt_username>`

1 Interest point detection (`part1_harris_corner.py`)

You will implement the Harris corner detection as described in the lecture materials and Szeliski 7.1.1.

The auto-correlation matrix A can be computed as (Equation 7.8 of Szeliski, p. 424)

$$A = w * \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}, \tag{1}$$

where we have replaced the weighted summations with discrete convolutions with the weighting kernel w.

The Harris corner score R is derived from the auto-correlation matrix A as (Equation 7.9 of Szeliski, p. 425):

$$R = \det(A) - \alpha \cdot \text{trace}(A)^2 \tag{2}$$

with $\alpha = 0.06$.

Algorithm 1: Harris Corner Detector

1. Compute the horizontal and vertical derivatives I_x and I_y of the image by convolving the original image with a Scharr filter;
2. Compute the three images corresponding to the outer products of these gradients. (The matrix A is symmetric, so only three entries are needed.);
3. Convolve each of these images with a larger Gaussian;
4. Compute a scalar interest measure using the formulas (Equation 2) discussed above;
5. Find local maxima above a median threshold;
6. Remove border values and report them as detected feature point locations.

To implement the Harris corner detector, you will have to fill out the following functions in `part1_harris_corner.py` under `src/vision` folder:

- `compute_image_gradients()`: Computes image gradients using the Scharr filter.
- `compute_harris_response_map()`: Gets the raw corner responses over the entire image (the previously implemented methods may be helpful).
- `nms_maxpool_pytorch()`: Performs non-maximum suppression using max-pooling. You can use PyTorch max-pooling operations for this.
- `get_harris_interest_points()`: Gets interests points from the entire image (the previously implemented methods may be helpful).

We have also provided the following helper methods in `part1_harris_corner.py`:

- `get_gaussian_kernel_2D_pytorch()`: Creates a 2D Gaussian kernel.
- `second_moments()`: Computes the second moments of the input image.
- `maxpool_numpy()`: Performs the maxpooling operation using just NumPy. This manual implementation will not be called, but help you understand what's happening in the next step.
- `remove_border_vals()`: Removes values close to the border that we can't create a useful SIFT window around.

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. The original paper by Chris Harris and Mike Stephens describing their corner detector can be found here.

2 Part 2: Local feature descriptors (`part2_patch_descriptor.py`)

To get your matching pipeline working quickly, you will implement a bare-bones feature descriptor in `part2_patch_descriptor.py` using normalized, grayscale image intensity patches as your local feature. See Szeliski 7.1.2 for more details when coding `compute_normalized_patch_descriptors()`

Choose the top-left option of the 4 possible choices for center of a square window, as shown in Figure 2.

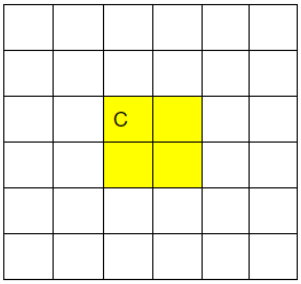

Figure 2: For this example of a 6×6 window, the yellow cells could all be considered the center. Please choose the top left (marked "C") as the center throughout this project.

The expected accuracy using Normalized patches on Notre Dame is around $40 - 45\%$ and Mt Rushmore around $45 - 50\%$.

3 Part 3: Feature matching (`part3_feature_matching.py`)

You will implement the "ratio test" (also known as the "nearest neighbor distance ratio test") method of matching local features as described in the lecture materials and Szeliski 7.1.3 with (Equation 7.18 of Szeliski, p. 444):

$$\text{NNDR} = \frac{d_1}{d_2} = \frac{\|D_A - D_B\|}{\|D_A - D_C\|} \tag{3}$$

The potential matches that pass the ratio test the easiest should have a greater tendency to be correct matches – think about *why* this is. In `part3_feature_matching.py`, you will have to code `compute_feature_distances()` to get pairwise feature distances, and `match_features_ratio_test()` to perform the ratio test to get matches from a pair of feature lists.

4 Part 4: SIFT Descriptor (`part4_sift_descriptor.py`)

You will implement a SIFT-like local feature as described in the lecture materials and Szeliski 7.1.2. We'll use a simple one-line modification ("Square-Root SIFT") from a 2012 CVPR paper (linked here) to get a free boost in performance. See the comments in the file `part4_sift_descriptor.py` for more details.

Regarding Histograms SIFT relies upon histograms. An unweighted 1D histogram with 3 bins could have bin edges of $[0, 2, 4, 6]$. If $x = [0.0, 0.1, 2.5, 5.8, 5.9]$, and the bins are defined over half-open intervals $[e_{\text{left}}, e_{\text{right}})$ with edges e, then the histogram $h = [2, 1, 2]$.

A weighted 1D histogram with the same 3 bins and bin edges has each item weighted by some value. For example, for an array $x = [0.0, 0.1, 2.5, 5.8, 5.9]$, with weights $w = [2, 3, 1, 0, 0]$, and the same bin edges ($[0, 2, 4, 6]$), $h_w = [5, 1, 0]$. In SIFT, the histogram weight at a pixel is the magnitude of the image gradient at that pixel.

In `part4_sift_descriptor.py`, you will have to implement the following:

- `get_orientations_and_magnitudes()`: Retrieves gradient magnitudes and orientations of the image.
- `get_gradient_histogram_vec_from_patch()`: Retrieves a feature consisting of concatenated histograms.
- `get_feat_vec()`: Gets the adjusted feature from a single point.
- `get_SIFT_descriptors()`: Gets all feature vectors corresponding to our interest points from an image.

The accuracy expected on running the SIFT pipeline on the Notre Dame image is at least 80%. Note that the Gaudi image pair will have low accuracy (close to 0) and think about why this could be happening.

5 Part 5: SIFT Descriptor Exploration

Take two images of a building or structure near you. Save them in the `additional_data/` folder of the project and run your SIFT pipeline on them. Analyze the results - why do you think our pipeline may have performed well or poorly for the given image pair? Is there anything about the building that is helpful or detrimental to feature matching?

6 Writeup

For this project (and all other projects), you must do a project report using the template slides provided to you. Do ***not*** change the order of the slides or remove any slides, as this will affect the grading process on Gradescope and you will be deducted points. 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. The template slides provide guidance for what you should include in your report. A good

writeup doesn't just show results–it tries to draw some conclusions from the experiments. You must convert the slide deck into a PDF for your submission, and then assign each PDF page to the relevant question number on Gradescope.

If you choose to do anything extra, add slides *after the slides given in the template deck* to describe your implementation, results, and analysis. You will not receive full credit for your extra credit implementations if they are not described adequately in your writeup.

Using the starter code (`project-2.ipynb`)

The top-level iPython notebook, `project-2.ipynb`, provided in the starter code includes file handling, visualization, and evaluation functions for you, as well as calls to placeholder versions of the functions listed above.

For the Notre Dame image pair there is a ground truth evaluation in the starter code as well. `evaluate_correspondence()` will classify each match as correct or incorrect based on hand-provided matches . 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 `project-2.ipynb`.

As you implement your feature matching pipeline, you should see your performance according to `evaluate_correspondence()` 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 faily well on these Notre Dame images.

Potentially useful NumPy and Pytorch functions

From Numpy: `np.argsort()`, `np.arctan2()`, `np.concatenate()`, `np.fliplr()`, `np.flipud()`, `np.histogram()`, `np.hypot()`, `np.linalg.norm()`, `np.linspace()`, `np.newaxis`, `np.reshape()`, `np.sort()`.

From Pytorch: `torch.argsort()`, `torch.arange()`, `torch.from_numpy()`, `torch.median()`, `torch.nn.functional.conv2d()`, `torch.nn.Conv2d()`, `torch.nn.MaxPool2d()`, `torch.nn.Parameter`, `torch.stack()`.

For the optional, extra-credit vectorized SIFT implementation, you might find `torch.meshgrid`, `torch.norm`, `torch.cos`, `torch.sin`.

We want you to build off of your Project 1 expertise. Please use `torch.nn.Conv2d` or `torch.nn.functional.conv2d` instead of convolution/cross-correlation functions from other libraries (e.g., `cv.filter2D()`, `scipy.signal.convolve()`).

Forbidden functions

(You can use these OpenCV, Sci-kit Image, and SciPy functions for testing, but not in your final code). `cv2.getGaussianKernel()`, `np.gradient()`, `cv2.Sobel()`, `cv2.Scharr()`, `cv2.SIFT()`, `cv2.SURF()`, `cv2.BFMatcher()`, `cv2.BFMatcher().match()`, `cv2.BFMatcher().knnMatch()`, `cv2.FlannBasedMatcher().knnMatch()`, `cv2.HOGDescriptor()`, `cv2.cornerHarris()`, `cv2.FastFeatureDetector()`, `cv2.ORB()`, `skimage.feature`, `skimage.feature.hog()`, `skimage.feature.daisy`, `skimage.feature.corner_harris()`, `skimage.feature.corner_shi_tomasi()`, `skimage.feature.match_descriptors()`, `skimage.feature.ORB()`, `cv.filter2D()`, `scipy.signal.convolve()`.

We haven't enumerated all possible forbidden functions here, but using anyone else's code that performs interest point detection, feature computation, or feature matching for you is forbidden.

Tips, tricks, and common problems

- Make sure you're not swapping x and y coordinates at some point. If your interest points aren't showing up where you expect, or if you're getting out of bound errors, you might be swapping x and y coordinates. Remember, images expressed as NumPy arrays are accessed `image[y, x]`.
- Make sure your features aren't somehow degenerate. you can visualize features with `plt.imshow( image1_features)`, although you may need to normalize them first. If the features are mostly zero or mostly identical, you may have made a mistake.
- If you receive the error message "pickle.UnpicklingError: the STRING opcode argument must be quoted", the .pkl files in the ground_truth folder likely have CRLF line endings from cloning. You can convert these to LF easily in VSCode by clicking "CRLF" in the bottom right, toggling it to "LF", and saving or using a tool like dos2unix.

Bells & whistles (extra points) / Extra Credit

Implementation of bells & whistles can increase your grade on this project by up to 10 points (potentially over 100). The max score for all students is 110.

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.

As detailed in lecture, there are numerous opportunities for extra credit. These include:

- Implement a vectorized version of SIFT. There are provided tests that check if it runs in under 5 seconds, with at least 80% accuracy on the Notre Dame image pair to make it simple to check (this is just one of the options).
- Detecting keypoints at multiple scales/picking the best scale for a given image
- Making keypoints rotation invariant
- Implementing a different interest point detection strategy
- Implementing a different feature descriptor

Rubric

- +15 pts: Implementation of Harris corner detector in `part1_harris_corner.py`
- +10 pts: Implementation of patch descriptor `part2_patch_descriptor.py`
- +10 pts: Implementation of "ratio test" matching in `part3_feature_matching.py`
- +40 pts: Implementation of SIFT-like local features in `part4_sift_descriptor.py`
- +25 pts: Report (remember to include part 5 `sift descriptor exploration`)
- -5*n pts: Lose 5 points for every time you do not follow the instructions for the hand-in format

Submission format

This is very important as you will lose 5 points for every time you do not follow the instructions. You will submit two items to Gradescope:

1. `<your_gt_username>.zip` containing:
    - `src/`: directory containing all your code for this assignment
    - `setup.cfg`: setup file for environment, no need to change this file
    - `additional_data/`: the images you took for Part 5, and/or if you use any data other than the images we provide, please include them here
    - `README.txt`: (optional) if you implement any new functions other than the ones we define in the skeleton code (e.g., any extra credit implementations), please describe what you did and how we can run the code. We will not award any extra credit if we can't run your code and verify the results.
2. `<your_gt_usernamme>_proj2.pdf` - your report

Credits

Assignment developed by James Hays, Cusuh Ham, John Lambert, Vijay Upadhya, and Samarth Brahmbhatt. Slightly modified by Humphrey Shi, Manushree Vasu, Aditya Kane, Chieh-Yun Chen, and Fengzhe Zhou.