CS 4476: Computer Vision, Spring 2023
PS2

Instructor: Judy Hoffman

Due: Monday, February 13th, 11:58 pm ET

Instructions

1. The assignment must be done in Python3. No other programming languages are allowed.
2. Fill your answers in the answer sheet PPT provided and submit the file under the name: FirstName_LastName_PS2.pdf on Gradescope. Please do not modify the layout of the boxes provided in the answer sheet and fit your answers within the space provided.
3. Please enter your code in the designated areas of the template Python files. Please do not add additional functions/imports to the files. Points will be deducted for any changes to code/file names, use of static paths and anything else that needs manual intervention to fix.
4. Please submit your code and output files in a zipped format, using the helper script `zip_submission.py` with your GT username as a command line argument (using `--gt_username`), to Gradescope. Please do not create subdirectories within the main directory. The `.zip_dir_list.yml` file contains the required deliverable files, and `zip_submission.py` will fail if all the deliverables are not present in the root directory. Feel free to comment and uncomment them as you complete your solutions.
5. For the implementation questions, make sure your code is bug-free and works out of the box. Please be sure to submit all main and helper functions. Be sure to not include absolute paths. Points will be deducted if your code does not run out of the box.
6. If plots are required, you must include them in your Gradescope report and your code must display them when run. Points will be deducted for not following this protocol.
7. Ensure that you follow the instructions very carefully.

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

Setup

Note that we will be using a new conda environment for this project! If you run into import module errors, try `pip install -e .` again, and if that still doesn't work, you may have to create a fresh environment.

1. Install Miniconda. It doesn't matter whether you use Python 2 or 3 because we will create our own environment that uses 3 anyways.
2. Open the terminal
   (a) On Windows: open the installed **Conda prompt** to run the command.

   (b) On MacOS: open a terminal window to run the command
   (c) On Linux: open a terminal window to run the command

3. Navigate to the folder where you have the project
4. Create the conda environment for this project
   (a) On Windows: `conda env create -f ps2_env_win.yml`
   (b) On MacOS: `conda env create -f ps2_env_mac.yml`
   (c) On Linux: `conda env create -f ps2_env_linux.yml`
5. Activate the newly created environment
   (a) On Windows: use the command `conda activate ps2`
   (b) On MacOS: use the command `source activate ps2`
   (c) On Linux: use the command `source activate ps2`
6. Install the project files as a module in this conda environment using `pip install -e .` (Do not forget the `.`).

Run the notebook using `jupyter notebook ./ps2_code/ps2.ipynb`.

At this point, you should see the jupyter notebook in your web browser. Follow all the instructions in the notebook for both the code + report portions of this project. Please go through the instructions given below (including Using the starter code).

1 Local Feature Matching using SIFT

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:

1. Interest point detection in `student_harris.py` (see Szeliski 7.1.1)
2. Local feature description in `student_sift.py` (see Szeliski 7.1.2)
3. Feature Matching in `student_feature_matching.py` (see Szeliski 7.1.2)

There are numerous papers in the computer vision literature addressing each stage. For this project, we will suggest specific, relatively simple algorithms for each stage. You are encouraged to experiment with more sophisticated algorithms!

1.1 Interest point detection (`student_harris.py`) [30 points]

You will implement the Harris corner detector as described in the lecture materials and Szeliski 7.1.1. See Algorithm 4.1 given below (taken from Szeliski Edition 1) for pseudocode. The starter code gives some additional suggestions. The main function will be `get_interest_points()`, while `my_filter2D()`, `get_gradients()`, `get_gaussian_kernel()`, `second_moments()`, `corner_response()`, `non_max_suppression()` will be helper functions that will have tests to check progress as you work through Harris corner detection. 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.

You will implement the Harris corner detector as described in the lecture materials and Szeliski 7.1.1. See the following for pseudocode (taken from Algorithm 4.1 in Szeliski Edition 1):

1. Compute the horizontal and vertical derivatives of the image I x and I y by convolving the original image with derivatives of Gaussians (Section 3.2.3).
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 one of the formulas discussed above.
5. Find local maxima above a certain threshold and report them as detected feature point locations.

1.2 Local feature description (`student_sift.py`) [35 points]

You will implement a SIFT-like local feature as described in the lecture materials and Szeliski 7.1.2. See `get_features()` for more details. If you want to get your matching pipeline working quickly (and maybe to help debug the other algorithm stages), you might want to start with normalized patches as your local feature. There are 2 helper functions in `student_sift.py` that you will need to code to get your SIFT pipeline working. `get_magnitudes_and_orientations()` will return the magnitudes and orientations of the image gradients at every pixel. `get_feat_vec()` will get the feature vector associated with a specific interest point. Once these are done, move on to coding `get_features()`, which will combine these to get feature vectors for all interest points. More info about each function can be found in the function headers.

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 $[e_{left}, e_{right})$ with edges e, the the histogram $h = [2, 1, 2]$.

A weighted 1D histogram with the same 3 and bin edges had 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.

1.3 Feature matching (`student_feature_matching.py`) [10 points]

You will implement the "ratio test" or "nearest neighbor distance ratio test" method of matching local features as described in the lecture materials and Szeliski 7.1.3. See equation 7.18 in particular. The potential matches that pass the ratio test the easiest should have a greater tendency to be correct matches – think about *why*.

1.4 Bells and Whistles [Extra credit: 10 points]

Implementation of bells and whistles can increase your grade 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.

- **up to 2 pts**: (On Mount Rushmore image) Try varying some of the hand selected variables and see how the accuracy of the feature matching changes. Specifically vary the standard deviation of the second moment filter with these options [3, 6, 10, 30] and report the images of the ground truth correspondence and their corresponding accuracies in the report. On top of this vary the feature width in SIFT by these options [8, 16, 24, 32] and report the ground truth correspondence images and the corresponding accuracies for each different value. For the experiment regarding SIFT feature width, you may need to adjust the window size in the function `remove_border_vals` to ensure all SIFT patches will remain entirely in the image. When changing these variables ensure all others are set to default.

  - **Report Question**: When changing the values for large sigma (>20), why are the accuracies generally the same?
  - **Report Question**: What is the significance of changing feature width in SIFT?

Local feature matching bells and whistles: An issue with the baseline matching algorithm is the computational expense of computing distance between all pairs of features. For a reasonable implementation of the base pipeline, this is likely to be the slowest part of the code. There are numerous schemes to try and approximate or accelerate feature matching:

  - **up to 5 pts**: 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. However, calling `sklearn.decomposition.PCA()` is not enough to get credit (you need to implement PCA yourself), but you are allowed to use `numpy.linalg.svd()`. The function for PCA can be found in `feature_matching.py`.
    * PCA is an algorithm that is based around Singular Value Decomposition where you take the SVD of the covariance matrix of your data and reduce the number of usable eigen vectors. Then we project the data onto the remaining usable eigenvectors. Here is a link to a Georgia Tech Machine Learning video that explains it in more depth (there are 3 parts).
  - **up to 3 pts**: Use a space partitioning data structure like a kd-tree or some third party approximate nearest neighbor package to accelerate matching. You must achieve a runtime of less than a second(just matching) for full credit (with 1500 interest points). Your accuracy must also remain > 80% for Notre Dame. Code your implementation in `accelerated_matching()` in `student_feature_matching.py()`.
    * **Report Question**: What did you try and what was the speedup? Why is it faster?

Using the starter code

The `ps2_code/ps2.ipynb` IPython notebook provided in the starter code includes file handling, visualization, and evaluation functions. The correspondence will be visualized with `show_correspondence_circles()` and `show_correspondence_lines()` (you can comment one or both out if you prefer).
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 (see `show_ground_truth_corr()` for details). 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 at the top of `ps2.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 fairly well on these Notre Dame images, but additional image pairs provided in the folder `data/` are more difficult. They might exhibit more viewpoint, scale, and illumination variation. If you add enough bells and whistles you should be able to match more difficult image pairs.

It is **strongly recommended** that you implement the functions in this order:

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 interest points from the the ground truth correspondences for the test cases. Even with this cheating, your accuracy will initially be near zero because the starter

code features are empty and the starter code matches don't exist. `cheat_interest_points()` might return non-integer values, but you'll have to cut patches out at integer coordinates. You should address this by truncating the numbers.

2. Second, change `get_features()` to return a simple feature. Start with, for instance, 16x16 patches centered on each interest point. 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. You won't see your accuracy increase yet because the placeholder code in `match_features()` isn't assigning matches.

3. Third, implement `match_features()`. Accuracy should increase on the Notre Dame pair if you're using 16x16 (256 dimensional) patches as your feature and if you only evaluate your 100 most confident matches. Accuracy on the other test cases will be lower.

4. Fourth, 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 if you only evaluate your 100 most confident matches (these are just estimates). These accuracies still aren't great because the human selected keypoints from `cheat_interest_points()` might not match particularly well according to your feature.

5. Fifth, stop using `cheat_interest_points()` 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 >80% accuracy. You will likely need to do extra credit to get high accuracy on Mount Rushmore and Episcopal Gaudi.

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 your 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.
- Make sure the size of the image resulting from convolution with the filter in `my_filter2D()` is the same as the size of the input image.
- Make sure you are generating bins for your histograms correctly. Check the arguments for the `np.histogram()` function carefully. It is recommended to use the `range` argument of the function to generate accurate bins.
- Check the documentation for the NumPy functions you are using, and make sure your understanding of the arguments and outputs is correct.

Rubric

Code: The score for each part is provided below. Please refer to the submission results on Gradescope for a detailed breakdown.

Part 1: Interest point detection	30
Part 2: Local Feature Description	35
Part 3: Feature matching	10
Part 4: Report	25
Extra Credit: Bells & Whistles	10
Total	*100 (+10)*

Submission Instructions and Deliverables

Unit Tests: Check that you pass all local unit tests by entering the `ps2_unit_tests` directory and running the command `pytest ./`. This command will run all the unit tests once more, and you need to add a screenshot to the report. Ensure that the conda environment `ps2` is being used.
Code zip: The following code deliverables will be uploaded as a zip file on Gradescope.

1. `ps2_code/student_harris.py`
   (a) `get_interest_points()`
   (b) `my_filter2D()`
   (c) `get_gradients()`
   (d) `get_gaussian_kernel()`
   (e) `second_moments()`
   (f) `corner_response()`
   (g) `non_max_suppression()`
2. `ps2_code/student_sift.py`
   (a) `get_magnitudes_and_orientations()`
   (b) `get_feat_vec()`
   (c) `get_features()`
3. `ps2_code/student_feature_matching.py`
   (a) `compute_feature_distances()`
   (b) `match_features()`
4. `ps2_code/ps2.ipynb`
5. `ps2_code/utils.py`
6. `ps2_code/collect_ground_truth_corr.py`

Do not create this zip manually! You are supposed to use the command `python zip_submission.py --gt_username <username>` for this.
Report: The final thing to upload is the PDF export of the report on gradescope.
To summarize, the deliverables are as follow:

- Submit the code as zip on Gradescope at PS2 - Code.
- Submit the report as PDF on Gradescope at PS2 - Report. The report is worth 25 points. Please refer to the pptx template where we have detailed the points associated with each question.

There is no submission to be done on Canvas. Good luck!

This iteration of the assignment is developed by Viraj Prabhu and Judy Hoffman. This assignment was originally developed by James Hays, Samarth Brahmbhatt, and John Lambert, and updated by Judy Hoffman, Mitch Donley, and Vijay Upadhya.