Project 2: SIFT Local Feature Matching and Camera Calibration

CS 6476

Fall 2025

Brief

- Due: Check Canvas for up to date information
- Project materials including report template: zip file on Canvas
- Hand-in: through Gradescope
- Required files: `<your_gt_username>.zip`, `<your_gt_username>_proj2.pdf`

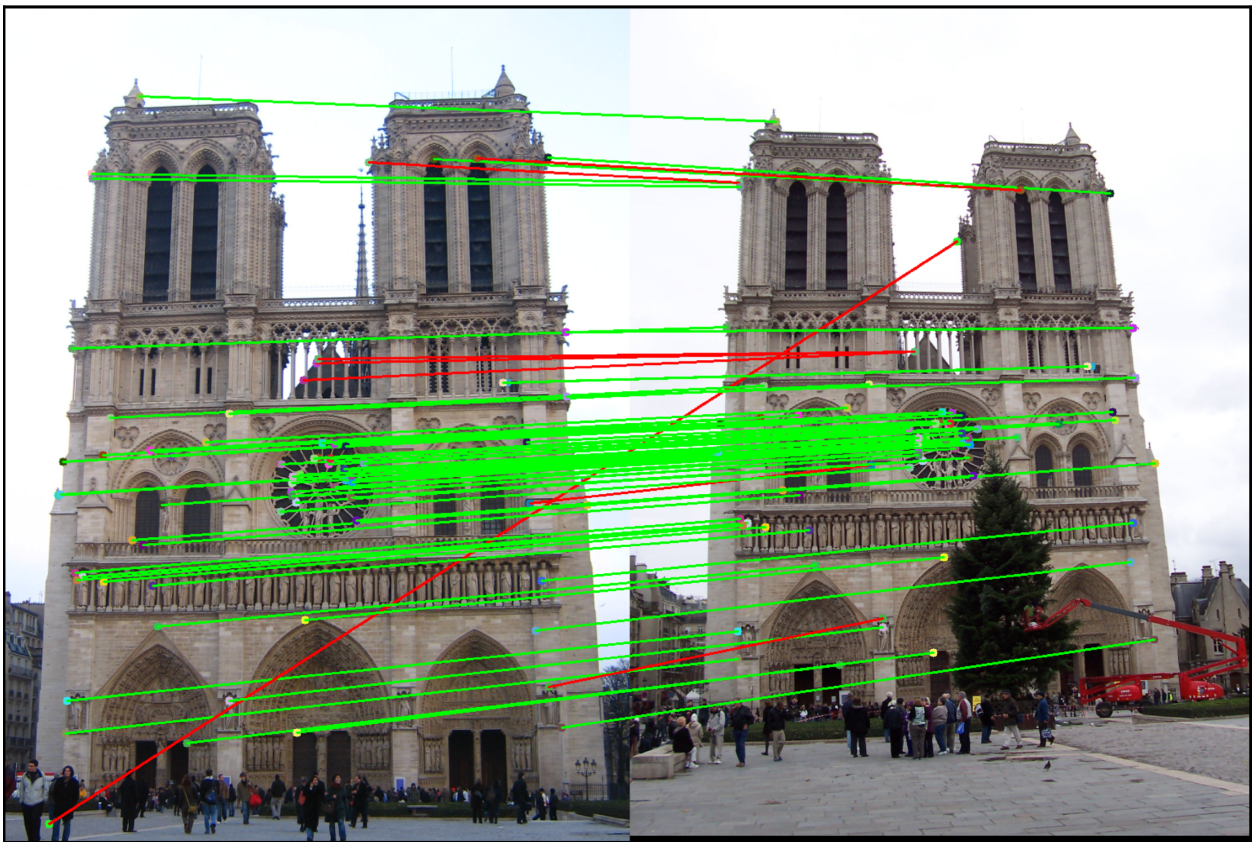

Figure 1: The top 100 most confident local feature matches from a baseline implementation of project 2. In this case, 89 were correct (lines shown in green), and 11 were incorrect (lines shown in red).

Overview

The goal of this assignment is to create a local feature matching algorithm using techniques described in Szeliski chapter 7.1, combined with epipolar constraints for feature matching in a stereo setup. 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

1. Check the README for environment installation.
2. Run the notebook using `jupyter notebook ./project-2.ipynb`
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>`

Details

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)
- Feature matching in `part2_feature_matching.py` (see Szeliski 7.1.3)
- Local feature description with the SIFT feature in `part3_sift_descriptor.py` (see Szeliski 7.1.2)
- 2D to 3D Projection matrices in `part4_projection_matrix.py`
- Fundamental matrix estimation in `part5_fundamental_matrix.py`

Part 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 book, p. 424)

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

where we have replaced the weighted summations with discrete convolutions with the weighting kernel w (Equation 7.9, p. 425).

The Harris corner score R is derived from the auto-correlation matrix A as:

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

with $\alpha = 0.06$.

Algorithm 1: Harris Corner Detector

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

To implement the Harris corner detector, you will have to fill out the following methods in `part1_harris_corner.py`:

- `compute_image_gradients()`: Computes image gradients using the Sobel filter.
- `compute_harris_response_map()`: Gets the raw corner responses over the entire image (the previously implemented methods may be helpful).
- `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`:

- `nms_maxpool_pytorch()`: Performs non-maximum suppression using max-pooling. You can use PyTorch max-pooling operations for this.
- `get_gaussian_kernel_2D_pytorch()`: Creates a 2D Gaussian kernel (this is essentially the same as your Gaussian kernel method from project 1).
- `second_moments()`: Computes the second moments of the input image. This makes use of your `get_gaussian_kernel_2D_pytorch()` method.
- `maxpool_numpy()`: Performs the maxpooling operation using just NumPy. This manual implementation will 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.

Part 2: Feature matching (`part2_feature_matching.py`)

To get your feature matching pipeline working quickly, we have provided you with an implementation for using normalized, grayscale image intensity patches as your local feature in `compute_normalized_patch_descriptors()` in `utils.py`. See Szeliski 7.1.2 for more details.

Using this pre-built feature descriptor, you will implement the "ratio test" (also known as the "nearest neighbor distance ratio test") method of matching local features. The *nearest neighbor distance ratio* is defined as

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

where d_1 and d_2 are the nearest and second nearest neighbor distances between the target descriptor D_A and the closest two neighbor descriptors D_B and D_C, respectively. For more details, please look at the lecture materials and Szeliski 7.1.3 (page 444). See equation 7.18 and figure 7.23 in particular. 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`, we have provided you with `compute_feature_distances()`, which calculates pairwise feature distances. You will have to code `match_features_ratio_test()` to perform the ratio test to get matches from a pair of feature lists. The expected accuracy using Normalized patches on Notre Dame is around $40 - 45\%$ and Mt Rushmore around $45 - 50\%$.

Part 3: SIFT Descriptor (`part3_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 `part3_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 $[e_{left}, e_{right})$ 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 `part3_sift_descriptor.py`, you will have to implement the following:

- `get_magnitudes_and_orientations()`: 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.

Part 4: Camera projection matrix (Extra Credit)

Introduction

The goal is to compute the projection matrix that goes from world 3D coordinates to 2D image coordinates. Recall that using homogeneous coordinates the equation for moving from 3D world to 2D camera coordinates is:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \cong \begin{pmatrix} u * s \\ v * s \\ s \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{3}$$

Another way of writing this equation is:

$$u = \frac{m_{11}X + m_{12}Y + m_{13}Z + m_{14}}{m_{31}X + m_{32}Y + m_{33}Z + m_{34}} \tag{4}$$

$$\rightarrow (m_{31}X + m_{32}Y + m_{33}Z + m_{34})u = m_{11}X + m_{12}Y + m_{13}Z + m_{14}$$

$$\rightarrow 0 = m_{11}X + m_{12}Y + m_{13}Z + m_{14} - m_{31}uX - m_{32}uY - m_{33}uZ - m_{34}u$$

$$v = \frac{m_{21}X + m_{22}Y + m_{23}Z + m_{24}}{m_{31}X + m_{32}Y + m_{33}Z + m_{34}} \tag{5}$$

$$\rightarrow (m_{31}X + m_{32}Y + m_{33}Z + m_{34})v = m_{21}X + m_{22}Y + m_{23}Z + m_{24}$$

$$\rightarrow 0 = m_{21}X + m_{22}Y + m_{23}Z + m_{24} - m_{31}vX - m_{32}vY - m_{33}vZ - m_{34}v$$

At this point, you're almost able to set up your linear regression to find the elements of the matrix M. There's only one problem–the matrix M is only defined up to a scale. Therefore, these equations have many different possible solutions (in particular M = all zeros is a solution, which is not very helpful in our context). The way around this is to first fix a scale, and then do the regression. There are several options for doing this: 1) You can fix the last element, $m_{34} = 1$, and then find the remaining coefficients, or 2) you can use the singular value decomposition to directly solve the constrained optimization problem:

$$\begin{aligned} \arg\min_{x} \quad & \|Ax\| \\ \text{s.t.} \quad & \|x\| = 1 \end{aligned} \tag{6}$$

To make sure that your code is correct, we are going to give you a set of "normalized points" in the files `pts2d-norm-pic_a.txt` and `pts3d-norm.txt`. If you solve for M using all the points, you should get a matrix that is a scaled equivalent of the following:

$$M_{\text{norm}A} = \begin{pmatrix} -0.4583 & 0.2947 & 0.0139 & -0.0040 \\ 0.0509 & 0.0546 & 0.5410 & 0.0524 \\ -0.1090 & -0.1784 & 0.0443 & -0.5968 \end{pmatrix}$$

For example, this matrix will take the last normalized 3D point $(1.2323, 1.4421, 0.4506, 1.0)$ and project it to (u, v) of $(0.1419, -0.4518)$, converting the homogeneous 2D point (us, vs, s) to its inhomogeneous version (the transformed pixel coordinate in the image) by dividing by s.

First, you will need to implement the least squares regression to solve for M given the corresponding normalized points. The starter code will load 20 corresponding normalized 2D and 3D points. You have to write the code to set up the linear system of equations, solve for the unknown entries of M, and reshape it into the estimated projection matrix. To validate that you've found a reasonable projection matrix, we've provided evaluation code which computes the total "residual" between the projected 2D location of each 3D point and the actual location of that point in the 2D image. The residual is just the distance (square root of the sum of squared differences in u and v). This should be very small.

Once you have an accurate projection matrix M, it is possible to tease it apart into the more familiar and more useful matrix K of intrinsic parameters and matrix $[R|T]$ of extrinsic parameters. For this project we will only ask you to estimate one particular extrinsic parameter: the camera center in world coordinates. Let us define M as being composed of a 3×3 matrix, Q, and a 4th column, m_4:

$$M = \begin{bmatrix} Q \mid m_4 \end{bmatrix} \tag{7}$$

From class we said that the center of the camera C could be found by:

$$C = -Q^{-1}m_4 \tag{8}$$

To debug your code, if you use you the normalized 3D points to get the M given above, you would get a camera center of:

$$C_{\text{norm}A} =< -1.5125, -2.3515, 0.2826 >$$

We've also provided a visualization which will show the estimated 3D location of the camera with respect to the normalized 3D point coordinates.

In `part4_projection_matrix.py`, you will implement the following:

- `projection()`: Projects homogeneous world coordinates $[X, Y, Z, 1]$ to non-homogeneous image coordinates (u, v). Given projection matrix M, the equations that accomplish this are (4) and (5).
- `calculate_projection_matrix()`: Solves for the camera projection matrix using a system of equations set up from corresponding 2D and 3D points.
- `calculate_camera_center()`: Computes the camera center location in world coordinates.

Part 5: Fundamental matrix (Extra Credit)

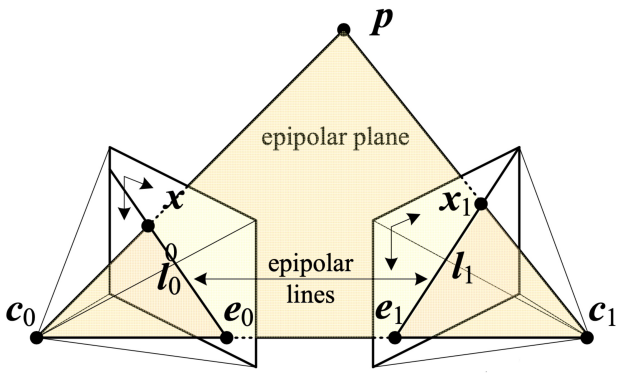

Figure 2: Two-camera setup. Reference: Szeliski, p. 682.

The next part of this project is estimating the mapping of points in one image to lines in another by means of the fundamental matrix. This will require you to use similar methods to those in part 4. We will make use of the corresponding point locations listed in `pts2d-pic_a.txt` and `pts2d-pic_b.txt`. Recall that the definition of the fundamental matrix is:

$$\begin{pmatrix} u' & v' & 1 \end{pmatrix} \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{pmatrix} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = 0 \tag{9}$$

for a point $(u, v, 1)$ in image A, and a point $(u', v', 1)$ in image B. See Appendix A for the full derivation. Note: the fundamental matrix is sometimes defined as the transpose of the above matrix with the left and right image points swapped. Both are valid fundamental matrices, but the visualization functions in the starter code assume you use the above form.

Another way of writing this matrix equations is:

$$\begin{pmatrix} u' & v' & 1 \end{pmatrix} \begin{pmatrix} f_{11}u + f_{12}v + f_{13} \\ f_{21}u + f_{22}v + f_{23} \\ f_{31}u + f_{32}v + f_{33} \end{pmatrix} = 0 \tag{10}$$

Which is the same as:

$$\left(f_{11}uu' + f_{12}vu' + f_{13}u' + f_{21}uv' + f_{22}vv' + f_{23}v' + f_{31}u + f_{32}v + f_{33}\right) = 0 \tag{11}$$

Starting to see the regression equations? Given corresponding points you get one equation per point pair. With 8 or more points you can solve this (why 8?). Similar to part 4, there's an issue here where the matrix is only defined up to scale and the degenerate zero solution solves these equations. So you need to solve using the same method you used in part 4 of first fixing the scale and then solving the regression.

The least squares estimate of F is full rank; however, a proper fundamental matrix is a rank 2. As such we must reduce its rank. In order to do this, we can decompose F using singular value decomposition into the

matrices $U\Sigma V' = F$. We can then construct a rank 2 matrix by setting the smallest singular value in Σ to zero thus generating Σ_2 . The fundamental matrix is then easily calculated as $F = U\Sigma_2 V'$. You can check your fundamental matrix estimation by plotting the epipolar lines using the plotting function provided in the starter code.

Coordinate normalization

As discussed in lecture, your estimate of the fundamental matrix can be improved by normalizing the coordinates before computing the fundamental matrix (see [Hartley(1997)]). It is suggested for this project you perform the normalization through linear transformations as described below to make the mean of the points zero and the average magnitude 1.0.

$$\begin{pmatrix} u' \\ v' \\ 1 \end{pmatrix} = \begin{pmatrix} s_u & 0 & 0 \\ 0 & s_v & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -c_u \\ 0 & 1 & -c_v \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \tag{12}$$

The transform matrix T is the product of the scale and offset matrices. c_u and c_v are the mean coordinates. To compute a scale s you could estimate the standard deviation after subtracting the means. Then the scale factor s would be the reciprocal of whatever estimate of the scale you are using. You could use one scale matrix based on the statistics of the coordinates from both images or you could do it per image.

In `part5_fundamental_matrix.py` you will need to use the scaling matrices to adjust your fundamental matrix so that it can operate on the original pixel coordinates. This is performed as follows:

$$F_{orig} = T_b^T * F_{norm} * T_a \tag{13}$$

In `part5_fundamental_matrix.py`, you will implement the following:

- `normalize_points()`: Normalizes the 2D coordinates.
- `unnormalize_F()`: Adjusts the fundamental matrix to account for the normalized coordinates. See Appendix B.
- `estimate_fundamental_matrix()`: Calculates the fundamental matrix.

Part 6: Homography with RANSAC

For two photographs of a scene, it's unlikely that you'd have perfect point correspondence with which to solve for the homography matrix. So, next you are going to compute the homography with point correspondences computed using SIFT. As discussed in class, a direct least-squares solution alone is not appropriate in this scenario due to the presence of multiple outliers. In order to estimate the homography from this noisy data, you'll need to use RANSAC. This technique is especially powerful for aligning images of planar surfaces, like building facades or the ground plane.

You'll use these putative point correspondences and RANSAC to find the "best" homography matrix. You will iteratively choose a minimal set of point correspondences (4 for a homography), solve for the homography matrix, and then count the number of inliers. Inliers in this context will be point correspondences that "agree" with the estimated homography. In order to count how many inliers a homography has, you'll need a distance metric. For a homography, this is typically the reprojection error: the geometric distance between a point in the second image (x') and the location predicted by mapping its corresponding point from the first image (Hx). You'll need to pick a threshold between inliers and outliers; your results are very sensitive to this threshold, so explore a range of values. You don't want to be too permissive about what you consider an inlier, nor do you want to be too conservative. Return the homography with the most inliers.

Recall from lecture the expected number of iterations of RANSAC to find the "right" solution in the presence of outliers. For example, if half of your input correspondences are wrong, then you have a $(0.5)^4 = 6.25\%$

chance to randomly pick 4 correct correspondences when estimating the homography. To be confident that you find the correct model, you should probably do hundreds or thousands of iterations of RANSAC.

For many real-world scenes like building facades, posters, or ground surfaces, the homography is the mathematically correct model describing the transformation between two camera views. The estimated homography can be "wrong" if the scene is not actually planar and contains significant depth, or if the camera has severe lens distortions that are not accounted for. Still, a robustly estimated homography does an excellent job at removing incorrect SIFT matches and is the foundation for tasks like image stitching and perspective correction.

For this part, you will implement the following methods in `part6_ransac.py`:

- `calculate_num_ransac_iterations()`: Calculates the number of RANSAC iterations needed for a given guarantee of success.
- `ransac_homography()`: Uses RANSAC to find the best homography matrix.

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 three 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.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

- +50 pts: Code (detailed point breakdown is on the Gradescope autograder)
- +5 pts: Code (Extra Credit)
- +50 pts: Report
- +5 pts: Report (Extra Credit)
- -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/`: (optional) the images you took for Part 6, 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.

References

[Hartley(1997)] Richard I Hartley. In defense of the eight-point algorithm. *IEEE Transactions on pattern analysis and machine intelligence*, 19(6):580–593, 1997.

Fundamental matrix derivation

Recall that the definition of the fundamental matrix is:

$$\begin{bmatrix}\mathbf{x}_2\\1\end{bmatrix} K^{-T}\left({}^2E_1\right)K^{-1}\begin{bmatrix}\mathbf{x}_1\\1\end{bmatrix} = 0 \tag{14}$$

Where does this equation come from? Szeliski shows that a 3D point $\mathbf{p}$ being viewed from two cameras (see Figure 2) can be modeled as:

$$d_1\hat{\mathbf{x}}_1 = \mathbf{p}_1 = {}^1\mathbf{R}_0\mathbf{p}_0 + {}^1\mathbf{t}_0 = {}^1\mathbf{R}_0(d_0\hat{\mathbf{x}}_0) + {}^1\mathbf{t}_0 \tag{15}$$

where $\hat{\mathbf{x}}_j = \mathbf{K}_j^{-1}\mathbf{x}_j$ are the (local) ray direction vectors. Note that ${}^1\mathbf{R}_0$ and ${}^1\mathbf{t}_0$ define an SE(3) '1_T_0' object that transforms $\mathbf{p}_0$ from camera 0's frame to camera 1's frame. We'll refer to these just as $\mathbf{R}$ and $\mathbf{t}$ for brevity in the following derivation.

We can eliminate the $+\mathbf{t}$ term by a cross-product. This can be achieved by multiplying with a skew-symmetric matrix as $[\mathbf{t}]_\times\mathbf{t} = 0$. Then:

$$d_1[\mathbf{t}]_\times\hat{\mathbf{x}}_1 = d_0[\mathbf{t}]_\times\mathbf{R}\hat{\mathbf{x}}_0. \tag{16}$$

Swapping sides and taking the dot product of both sides with $\hat{\mathbf{x}}_1$ yields

$$d_0\hat{\mathbf{x}}_1^T([\mathbf{t}]_\times\mathbf{R})\hat{\mathbf{x}}_0 = d_1\hat{\mathbf{x}}_1^T[\mathbf{t}]_\times\hat{\mathbf{x}}_1 = 0, \tag{17}$$

Since the cross product $[\mathbf{t}]_x$ returns 0 when pre- and post-multiplied by the same vector, we arrive at the familiar epipolar constraint, where $\mathbf{E} = [\mathbf{t}]_\times\mathbf{R}$:

$$\hat{\mathbf{x}}_1^T\mathbf{E}\hat{\mathbf{x}}_0 = 0 \tag{18}$$

The fundamental matrix is defined as $F = \mathbf{K}_1^{-T}\mathbf{E}\mathbf{K}_0^{-1}$. Thus,

$$\begin{pmatrix} u' & v' & 1 \end{pmatrix} \mathbf{K}_1^{-T}\mathbf{E}\mathbf{K}_0^{-1} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = 0 \tag{19}$$

We can write this as:

$$\begin{pmatrix} u' & v' & 1 \end{pmatrix} \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{pmatrix} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = 0 \tag{20}$$

Unnormalizing normalized coordinates

The main idea of coordinate normalization is to replace coordinates $\mathbf{u}_a$ in image a with $\hat{\mathbf{u}}_a = T_a\mathbf{u}_a$, and coordinates $\mathbf{u}_b$ in image b with $\hat{\mathbf{u}}_b = T_b\mathbf{u}_b$. If T is chosen to be invertible, then we can recover the original coordinates from the transformed ones, as

$$\begin{aligned} \hat{\mathbf{u}}_a &= T_a\mathbf{u}_a \\ T_a^{-1}\hat{\mathbf{u}}_a &= T_a^{-1}T_a\mathbf{u}_a \\ T_a^{-1}\hat{\mathbf{u}}_a &= \mathbf{u}_a \end{aligned} \tag{21}$$

Substituting in the equation $\mathbf{u}_b^T F\mathbf{u}_a = 0$, we derive the equation

$$\begin{aligned} \mathbf{u}_b^T F\mathbf{u}_a &= 0 \\ (T_b^{-1}\hat{\mathbf{u}}_b)^T F T_a^{-1}\hat{\mathbf{u}}_a &= 0 \\ \hat{\mathbf{u}}_b^T T_b^{-T} F T_a^{-1}\hat{\mathbf{u}}_a &= 0 \end{aligned} \tag{22}$$

If we use the normalized points $\mathbf{u}_a, \mathbf{u}_b^T$ when fitting the fundamental matrix, then we will end up estimating $\hat{F} = T_b^{-T}FT_a^{-1}$. In other words, $\mathbf{u}_b^T F\mathbf{u}_a = \hat{\mathbf{u}}_b^T\hat{F}\hat{\mathbf{u}}_a$. If we want to find out the original F that corresponded to raw (unnormalized) point coordinates, than we need to transform backwards:

$$
\begin{aligned}
\hat{F} &= T_b^{-T} F T_a^{-1} \\
T_b^T \hat{F} &= T_b^T T_b^{-T} F T_a^{-1} \\
T_b^T \hat{F} T_a &= F T_a^{-1} T_a \\
T_b^T \hat{F} T_a &= F
\end{aligned}
\tag{23}
$$