Lab 5 Report: EXERCISES

Yilin Zhang [1]. Lab Group: 31.

I. Individual

A. Deliverable - Practice with Perspective Projection

1. Sol.

Consider a sphere of radius r centered at $(0, 0, d)$, where $d > r + 1$. Camera parameters: focal length $f = 1$, principal point $(0, 0)$, pixel sizes $s_x = s_y = 1$, zero skew. Image plane coordinates are denoted as (u, v), corresponding to the projection relations $u = X/Z$, $v = Y/Z$, where (X, Y, Z) is a 3D point in the camera coordinate system.

The sphere equation is:

$$X^2 + Y^2 + (Z - d)^2 = r^2. \tag{1}$$

Substituting $X = uZ$ and $Y = vZ$ gives:

$$(uZ)^2 + (vZ)^2 + (Z - d)^2 = r^2.$$

Expanding and rearranging into a quadratic equation in Z:

$$Z^2(u^2 + v^2 + 1) - 2dZ + (d^2 - r^2) = 0.$$

For a given (u, v), if this equation has a positive real solution Z (i.e., a point on the sphere and in front of the camera), then (u, v) lies within the projection region. The condition for existence of real solutions is that the discriminant is nonnegative:

$$\Delta = (-2d)^2 - 4(u^2 + v^2 + 1)(d^2 - r^2) \geq 0.$$

Simplifying:

$$d^2 - (u^2 + v^2 + 1)(d^2 - r^2) \geq 0.$$

Since $d > r$, we have $d^2 - r^2 > 0$, leading to:

$$u^2 + v^2 + 1 \leq \frac{d^2}{d^2 - r^2}.$$

Thus:

$$u^2 + v^2 \leq \frac{r^2}{d^2 - r^2}.$$

Therefore, the projection region is a disk, whose boundary is the circle:

$$u^2 + v^2 = \frac{r^2}{d^2 - r^2}. \tag{2}$$

The radius of this circle is $R = \dfrac{r}{\sqrt{d^2 - r^2}}$.

2. Sol.

Let the sphere center be at (a, b, d), still satisfying $d > r+1$. The sphere equation becomes:

$$(X - a)^2 + (Y - b)^2 + (Z - d)^2 = r^2. \tag{3}$$

Substituting $X = uZ$, $Y = vZ$ gives:

$$(uZ - a)^2 + (vZ - b)^2 + (Z - d)^2 = r^2.$$

Expanding and rearranging into a quadratic in Z:

$$(u^2 + v^2 + 1)Z^2 - 2(au + bv + d)Z + (a^2 + b^2 + d^2 - r^2) = 0.$$

The nonnegative discriminant condition yields:

$$[2(au + bv + d)]^2 - 4(u^2 + v^2 + 1)(a^2 + b^2 + d^2 - r^2) \geq 0.$$

Define $L^2 = a^2 + b^2 + d^2$. Simplifying gives:

$$(au + bv + d)^2 \geq (u^2 + v^2 + 1)(L^2 - r^2).$$

Expanding and rearranging into a quadratic inequality yields

$$Au^2 + Buv + Cv^2 + Du + Ev + F \leq 0, \tag{4}$$

where

$$\begin{aligned} A &= b^2 + d^2 - r^2, & B &= -2ab, & C &= a^2 + d^2 - r^2, \\ D &= -2ad, & E &= -2bd, & F &= a^2 + b^2 - r^2. \end{aligned}$$

This inequality represents an elliptical disk (including its interior). When $a = b = 0$, it reduces to a circular disk. In general, the projection is an ellipse whose shape and orientation depend on the sphere center position (a, b, d). Since the sphere is convex and entirely in front of the camera ($d - r > 0$), the projection region is convex; thus, the ellipse is closed.

B. Deliverable - Vanishing Points

1. Sol.

Consider two 3D lines parallel to a direction vector $\mathbf{u} = (u_x, u_y, u_z)^\top$. Under the given camera model (intrinsic matrix is identity), the projection of a 3D point (X, Y, Z) onto the image plane is $(x, y) = (X/Z, Y/Z)$.

Parameterize one line as $\mathbf{p}(\lambda) = \mathbf{p}_0 + \lambda\mathbf{u}$, where $\mathbf{p}_0 = (X_0, Y_0, Z_0)$ and $\lambda \in \mathbb{R}$. Its projection is:

$$\mathbf{x}_1(\lambda) = \left(\frac{X_0 + \lambda u_x}{Z_0 + \lambda u_z},\ \frac{Y_0 + \lambda u_y}{Z_0 + \lambda u_z} \right).$$

As $\lambda \to \infty$, the projected point tends to $(u_x/u_z, u_y/u_z)$ provided $u_z \neq 0$. This limit point is independent of $\mathbf{p}_0$ and is

[1] Yilin zhang, OUC id: 23020036094, is with Faculty of Robotics and Computer Science, Ocean University of China and Heriot-Watt University, China Mainland zyl8820@stu.ouc.edu.cn

common to all lines parallel to $\mathbf{u}$. Thus, the projections of any two such lines intersect at this point, known as the vanishing point.

If $u_z = 0$, the denominator remains constant, and the projection becomes:

$$\mathbf{x}_1(\lambda) = \left(\frac{X_0}{Z_0} + \lambda \frac{u_x}{Z_0},\ \frac{Y_0}{Z_0} + \lambda \frac{u_y}{Z_0} \right),$$

which is a line with direction (u_x, u_y) scaled by $1/Z_0$. In this case, the vanishing point is at infinity, and the projected lines remain parallel.

Therefore, the generic expression for the vanishing point is:

$$\mathbf{v} = \left(\frac{u_x}{u_z},\ \frac{u_y}{u_z} \right) \quad \text{for } u_z \neq 0. \tag{5}$$

If $u_z = 0$, the vanishing point is at infinity.

2. Claim: 3D parallel lines remain parallel in the image plane if and only if their direction vector satisfies $u_z = 0$, i.e., the lines are parallel to the image plane.

Proof:

Consider two parallel lines with direction $\mathbf{u} = (a, b, c)$:

$$\mathbf{p}(\lambda) = \mathbf{p}_0 + \lambda \mathbf{u}$$
$$\mathbf{q}(\mu) = \mathbf{q}_0 + \mu \mathbf{u},$$

where $\mathbf{p}_0 = (X_0, Y_0, Z_0)$, $\mathbf{q}_0 = (X_0', Y_0', Z_0')$, and $Z_0, Z_0' > 0$. Their projections are:

$$\mathbf{x}_1(\lambda) = \left(\frac{X_0 + \lambda a}{Z_0 + \lambda c},\ \frac{Y_0 + \lambda b}{Z_0 + \lambda c} \right),$$
$$\mathbf{x}_2(\mu) = \left(\frac{X_0' + \mu a}{Z_0' + \mu c},\ \frac{Y_0' + \mu b}{Z_0' + \mu c} \right).$$

These are straight lines in the image. Their tangent directions (constant along each line) are proportional to:

$$\mathbf{d}_1 = (aZ_0 - cX_0,\ bZ_0 - cY_0),$$
$$\mathbf{d}_2 = (aZ_0' - cX_0',\ bZ_0' - cY_0').$$

The projected lines are parallel if and only if $\mathbf{d}_1$ and $\mathbf{d}_2$ are parallel, i.e., $\mathbf{d}_1 \times \mathbf{d}_2 = 0$.

- If $c = 0$ (i.e., $u_z = 0$), then $\mathbf{d}_1 = (aZ_0, bZ_0)$ and $\mathbf{d}_2 = (aZ_0', bZ_0')$. Both are scalar multiples of (a, b), so the lines are parallel regardless of $\mathbf{p}_0$ and $\mathbf{q}_0$.
- If $c \neq 0$, then $\mathbf{d}_1$ and $\mathbf{d}_2$ generally depend on $\mathbf{p}_0$ and $\mathbf{q}_0$. For arbitrary choices, they are not parallel, and the lines intersect at the vanishing point $(a/c, b/c)$. Hence, the projected lines are not parallel.

Thus, for arbitrary 3D parallel lines (not passing through the camera center) to remain parallel in the image, the necessary and sufficient condition is $u_z = 0$. This means the lines are parallel to the image plane.

That is, 3D parallel lines remain parallel in the image plane if and only if their direction vector satisfies $u_z = 0$.

II. Team

III. Some words

Appendix A
Individual

A. Deliverable - Practice with Perspective Projection

Consider a sphere with radius r centered at $[0\ 0\ d]$ with respect to the camera coordinate frame (centered at the optical center and with axis oriented as discussed in class). Assume $d > r + 1$ and assume that the camera has principal point at $(0, 0)$, focal length equal to 1, pixel sizes $s_x = s_y = 1$ and zero skew $s_\theta = 0$ (see lecture notes for notation) the following exercises:

1) **Derive** the equation describing the projection of the sphere onto the image plane.
   Hint: Think about what shape you expect the projection on the image plane to be, and then derive a characteristic equation for that shape in the image plane coordinates u, v along with r and d.
2) **Discuss** what the projection becomes when the center of the sphere is at an arbitrary location, not necessarily along the optical axis. What is the shape of the projection?

B. Deliverable - Vanishing Points

Consider two 3D lines that are parallel to each other. As we have seen in the lectures, lines that are parallel in 3D may project to intersecting lines on the image plane. The pixel at which two 3D parallel lines intersect in the image plane is called a vanishing point. Assume a camera with principal point at (0,0), focal length equal to 1, pixel sizes $s_x = s_y = 1$ and zero skew $s_\theta = 0$ (see lecture notes for notation). Complete the following exercises:

1) **Derive** the generic expression of the vanishing point corresponding to two parallel 3D lines.
2) **Find (and prove mathematically)** a condition under which 3D parallel lines remain parallel in the image plane.

Hint: For both 1. and 2. you may use two different approaches:
Algebraic approach: a 3D line can be written as a set of points $p(\lambda) = p_0 + \lambda u$ where $p_0 \in \mathbb{R}^3$ is a point on the line, $u \in \mathbb{R}^3$ is a unit vector along the direction of the line, and $\lambda \in \mathbb{R}$.

Geometric approach: the projection of a 3D line can be understood as the intersection beetween two planes.

Appendix B
Team

A. Deliverable - Feature Descriptors (SIFT)

We provide you with skeleton code for the base class FeatureTracker that provides an abstraction layer for all feature trackers. Furthermore, we give you two empty structures for the SIFT and SURF methods that derive from the class FeatureTracker.

Inside the `lab5` folder, we provide you with two images 'box.png' and 'box_in_scene.png' (inside the `images` folder).

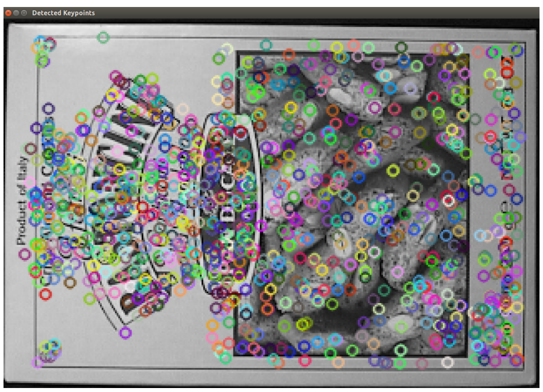
We will first ask you to extract keypoints from both images using SIFT. For that, we refer you to the skeleton code in the `src` folder named `track_features.cpp`. Follow the instructions written in the comments; specifically, you will need to complete:

- The stub `SiftFeatureTracker::detectKeypoints()` and `SiftFeatureTracker::describeKeypoints()` in `lab5/feature_tracking/src/sift_feature_tracker.cpp`
- The first part of `FeatureTracker::trackFeatures()` in `lab5/feature_tracking/src/feature_tracker.cpp`

Once you have implemented SIFT, you can test it by running:

```
roslaunch lab_5 two_frames_tracking.launch
↪ descriptor:=SIFT # note you can change the
↪ descriptor later
```

Your code should be able to plot a figure like the one below (keypoints you detected should not necessarily coincide with the ones in the figure):



Now that we have detected keypoints in each image, we need to find a way to uniquely identify these to subsequently match features from frame to frame. Feature descriptors in the literature are multiple, and while we will not review all of them, they all rely on a similar principle: using the pixel intensities around the detected keypoint to describe the feature.

Descriptors are multidimensional and are typically represented by an array.

Follow the skeleton code in `src` to compute the descriptors for all the extracted keypoints.

Hint: In OpenCV, SIFT and the other descriptors we will look at are children of the "Feature2D" class. We provide you with a SIFT `detector` object, so look at the Feature2D "Public Member Functions" documentation to determine which command you need to detect keypoints and get their descriptors.

B. *Deliverable - Descriptor-based Feature Matching*

With the pairs of keypoint detections and their respective descriptors, we are ready to start matching keypoints between two images. It is possible to match keypoints just by using a brute force approach. Nevertheless, since descriptors can have high-dimensionality, it is advised to use faster techniques.

In this exercise, we will ask you to use the FLANN (Fast Approximate Nearest Neighbor Search Library), which provides instead a fast implementation for finding the nearest neighbor. This will be useful for the rest of the problem set, when we will use the code in video sequences.

1) What is the dimension of the SIFT descriptors you computed?
2) Compute and plot the matches that you found from the *box.png* image to the *box_in_scene.png*.
3) You might notice that naively matching features results in a significant amount of false positives (outliers). There are different techniques to minimize this issue. The one proposed by the authors of SIFT was to calculate the best two matches for a given descriptor and calculate the ratio between their distances: `Match1.distance < 0.8 * Match2.distance`. This ensures that we do not consider descriptors that have been ambiguously matched to multiple descriptors in the target image. Here we used the threshold value that the SIFT authors proposed (0.8).
4) Compute and plot the matches that you found from the *box.png* image to the *box_in_scene.png* after applying the filter that we just described. You should notice a significant reduction of outliers.

Specifically, you will need to complete:

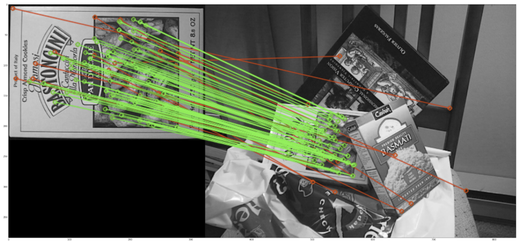
- The stub `SiftFeatureTracker::matchDescriptors()` in `feature_tracking/src/sift_feature_tracker.cpp`
- The second part of `FeatureTracker::trackFeatures()` in `feature_tracking/src/feature_tracker.cpp`
  Hint: Note that the `matches` object is a pointer type, so to use it you will usually have to type `*matches`. Once you've used the FLANN matcher to get the matches, if you want to iterate over all of them you could use a loop like:
  ```
  for (auto& match : *matches) {
  // check match.size(), match[0].distance,
  ↪ match[1].distance, etc.
  }
  ```
  Likewise, `good_matches` is a pointer so to add to it you will need `good_matches->push_back(...)`.

C. *Deliverable - Keypoint Matching Quality*

Excellent! Now, that we have the matches between keypoints in both images, we can apply many cool algorithms that you will see in subsequent lectures.

For now, let us just use a blackbox function which, given the keypoints correspondences from image to image, is capable of deciding whether some matches are considered outliers.

1) Using the function we gave you, compute and plot the inlier and outlier matches, such as in the following figure:
   Hint: Note that `FeatureTracker::inlierMaskComputation` computes an inlier mask of type `std::vector<uchar>`, but for `cv::drawMatches` you will need a `std::vector<char>`. You can go from one to the other by using:
   `std::vector<char> char_inlier_mask{inlier_mask.begin(), inlier_mask.end()};`
   Hint: You will need to call the `cv::drawMatches` function twice, first to plot the everything in red (using `cv::Scalar(0,0,255)` as the color), and then again to plot



inliers in green (using the inlier mask and `cv::Scalar(0,255,0)`). The second time, you will need to use the `DrawMatchesFlags::DRAW_OVER_OUTIMG` flag to draw on top of the first output. To combine flags for the `cv::drawMatches` function, use the bitwise-or operator:

```
DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS |
 DrawMatchesFlags::DRAW_OVER_OUTIMG
```

2) Now, we can calculate useful statistics to compare feature tracking algorithms. First, let's compute statistics for SIFT. The code for computing the statistics is already included in the skeleton file. You just need to use the appropriate functions to print them out. Submit a table similar to the following for SIFT (you might not get the same results, but they should be fairly similar):
TODO: This is a table.

D. Deliverable - Comparing Feature Matching Algorithms on Real Data

The most common algorithms for feature matching use different detection, description, and matching techniques. We'll now try different techniques and see how they compare against one another:

1) Pair of frames: Fill the previous table with results for other feature tracking algorithms. We ask that you complete the table using the following additional algorithms:

1) SURF (a faster version of SIFT)
2) ORB (we will use it for SLAM later!)
3) Optional : FAST (detector) + BRIEF (descriptor)
Hint: The SURF functions can be implemented exactly the same as SIFT, while ORB and FAST can also be implemented exactly the same way except that the FLANN matcher must be initialized with a new parameter like this: `FlannBasedMatcher matcher(new flann::LshIndexParams(20, 10, 2));`. This is not necessarily the best solution for ORB and FAST however, so we encourage you to look into other methods (e.g. the Brute-Force matcher instead of the FLANN matcher) if you have time. Note for BRIEF, you will need to create a BriefDescriptorExtractor to create keypoint descriptors when implementing FAST+BRIEF.

We have provided method stubs in the corresponding header files for you to implement in the CPP files. Please refer to the OpenCV documentation, tutorials, and C++ API when filling them in. You are encouraged to modify the default parameters used by the features extractors and matchers. A complete answer to this deliverable should include a brief discussion of what you tried and what worked the best.

By now, you should have four algorithms, with their pros and cons, capable of tracking features between frames.

Hint: It is normal for some descriptors to perform worse than others, especially on this pair of images – in fact, some may do very poorly, so don't worry if you observe this.

2) Real Datasets: Let us now use an actual video sequence to track features from frame to frame and push these algorithms to their limit!

We have provided you with a set of datasets in rosbag format here. Please download the following datasets, which are the two "easiest" ones:

- `30fps_424x240_2018-10-01-18-35-06.bag`
- `vnav-lab5-smooth-trajectory.bag`

Testing other datasets may help you to identify the relative strengths and weaknesses of the descriptors, but this is not required.

We also provide you with a roslaunch file that executes two ROS nodes:

```
roslaunch lab_5 video_tracking.launch
↪  path_to_dataset:=/home/$USER/Downloads/<NAME_OF
↪  _DOWNLOADED_FILE>.bag
```

- One node plays the rosbag for the dataset
- The other node subscribes to the image stream and is meant to compute the statistics to fill the table below

You will need to first specify in the launch file the path to your downloaded dataset. Once you're done, you should get something like this:

Hint: You may need to change your plotting code in `FeatureTracker::trackFeatures` to call `cv::waitKey(10)` instead of `cv::waitKey(0)` after imshow in order to get the video to play continuously instead of frame-by-frame on every keypress.

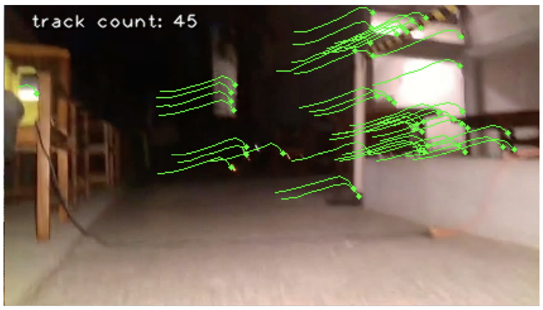
Finally, we ask you to summarize your results in one table for each dataset and asnwer some questions:

- Compute the **average** (over the images in each dataset) of the statistics on the table below for the different datasets and approaches. You are free to use whatever parameters you find result in the largest number of inliers.
TODO: this is a table
- What conclusions can you draw about the capabilities of the different approaches? Please make reference to what you have tested and observed.
Hint: **We don't expect long answers, there aren't specific answers we are looking for, and you don't need to answer every suggested question below!** We are just looking for a few sentences that point out the main differences you noticed and that are supported by your table/plots/observations.
Some example questions to consider:
Which descriptors result in more/fewer keypoints?
How do they the descriptors differ in ratios of good matches and inliers?
Are some feature extractors or matchers faster than others?
What applications are they each best suited for? (e.g. when does speed vs quality matter)

E. Deliverable - Feature Tracking: Lucas Kanade Tracker

So far we have worked with descriptor-based matching approaches. As you have seen, these approaches match features by simply comparing their descriptors. Alternatively, feature tracking methods use the fact that, when recording a video sequence, a feature will not move much from frame to frame. We will now use the most well-known differential feature tracker, also known as Lucas-Kanade (LK) Tracker.

1) Using OpenCV's documentation and the C++ API for the LK tracker, track features for the video sequences we provided you by using the Harris corner detector (like here). Show the feature tracks at a given frame extracted when using the Harris corners, such as this:



2) Add an extra entry to the table used in **Deliverable 6** B-D. using the Harris + LK tracker that you implemented.
3) What assumption about the features does the LK tracker rely on?
4) Comment on the different results you observe between the table in this section and the one you computed in the other sections.

   Hint: You will need to convert the image to grayscale with `cv::cvtColor` and will want to look into the documentation for `cv::goodFeaturesToTrack` and `cv::calcOpticalFlowPyrLK`. The rest of the trackFeatures() function should be mostly familiar feature matching and inlier mask computation similar to the previous sections. Also note that the `status` vector from calcOpticalFlowPyrLK indicates the matches.

   Hint: For the show() method, you will just need to create a copy of the input frame and then make a loop that calls `cv::line` and `cv::circle` with correct arguments before calling `imshow`.

F. [Optional] Deliverable - Optical Flow

LK tracker estimates the optical flow for sparse points in the image. Alternatively, dense approaches try to estimate the optical flow for the whole image. Try to calculate your own optical flow, or the flow of a video of your choice, using Farneback's algorithm.

Hint: Take a look at this tutorial, specifically the section on dense optical flow. Please post on piazza if you run into any issues or get stuck anywhere.