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

$$A = b^2 + d^2 - r^2, \quad B = -2ab, \quad C = a^2 + d^2 - r^2,$$
$$D = -2ad, \quad E = -2bd, \quad F = a^2 + b^2 - r^2.$$

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

Feature Descriptors (SIFT)

This section details the front-end implementation of a visual odometry system based on feature matching. The core task is to create stable correspondences between successive image frames by identifying and comparing unique visual landmarks. We chose the *Scale-Invariant Feature Transform (SIFT)* for its strong performance under varying scales and rotations. The code is designed to isolate the feature extraction algorithm from the broader tracking workflow, making it straightforward to swap in different methods later.

This file `sift\_feature\_tracker.cpp` C contains the core implementation of the SIFT algorithm.

The `detectKeypoints` and `describeKeypoints` functions rely on OpenCV's `SIFT::create()`. SIFT finds keypoints by locating peaks in the Difference of Gaussians (DoG) pyramid across different scales, which gives it scale invariance. It then builds a 128-element descriptor vector from the distribution of gradient directions around each keypoint, providing rotation invariance.

In file `feature\_tracker.cpp`, we updated the `trackFeatures` function to manage the processing steps. It calls `detectKeypoints` on both images. Running the command `roslaunch lab\_5 two\_frames\_tracking .launch descriptor:=SIFT` generates a visualization of the matched features.

The figures 1 show the results. Keypoints, drawn with their detected size and orientation, cluster in textured regions like the printed text and box edges. This demonstrates the detector's ability to find distinctive image patterns.

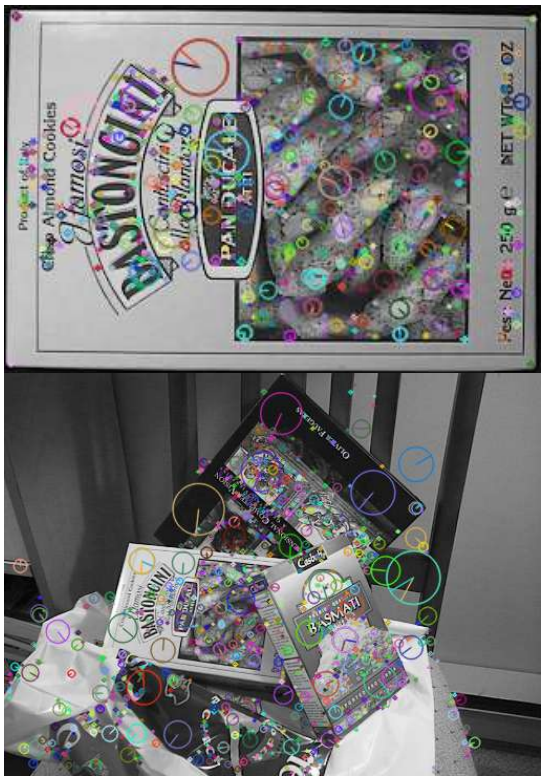


Fig. 1: Local Feature Extraction (SIFT Keypoints)

Feature Matching via Descriptors

This part connects the features found earlier by matching them across images. We built the `matchDescriptors` function in `sift\_feature\_tracker.cpp` to handle this.

Rather than using a slow, exhaustive search, we chose a *FLANN-based matcher* with a K-Nearest Neighbors approach (KNN, $k = 2$. To weed out unreliable matches, we added *Lowe's Ratio Test* as a filter.

The test works by checking:

$$d_1 < \text{ratio} \cdot d_2$$

Here, d_1 is the distance to the best-matching descriptor, and d_2 is the distance to the second-best. We kept the ratio threshold at 0.8. This helps ignore ambiguous matches—like those from the box's repeating patterns—and tightens up the overall match quality.

You can see the results 2. While most matches correctly pair up similar parts of the box (corners with corners, for example), some *outliers* still slip through (shown by crossing lines). These mismatches break the expected smooth motion between frames, showing that matching based on looks alone isn't enough.

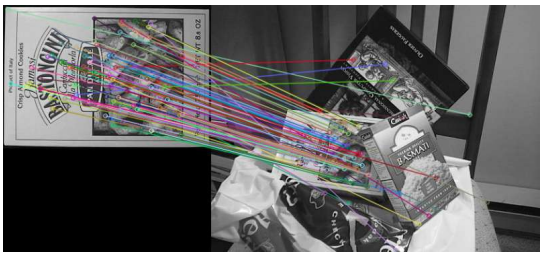


Fig. 2: Feature Matching Results (After Ratio Test Filtering)

Matching Quality Refinement

To clean up the remaining bad matches, we added Geometric Verification in `feature\_tracker.cpp` through the `inlierMaskComputation` function.

The method uses *RANSAC (Random Sample Consensus)* to find the best Fundamental Matrix (F that fits the matched points. The core idea is the *Epipolar Constraint*:

$$p_2^\top F p_1 = 0$$

Here, p_1 and p_2 are matched points in the first and second images (in homogeneous coordinates). Matches that don't follow this rule within 3.0 pixels are thrown out as outliers.

Compared to the results from Deliverable 4, 3 the crossing mismatch lines are now gone. The surviving matches (inliers) show a clean, consistent motion pattern, which better reflects the actual camera movement around the box.

The terminal output looked like:

```
Avg. Keypoints 1 Size: 603
Avg. Keypoints 2 Size: 969
Avg. Number of matches: 603
Avg. Number of good matches: 98
Avg. Number of Inliers: 77
Avg. Inliers ratio: 0.785714
Num. of samples: 1
```

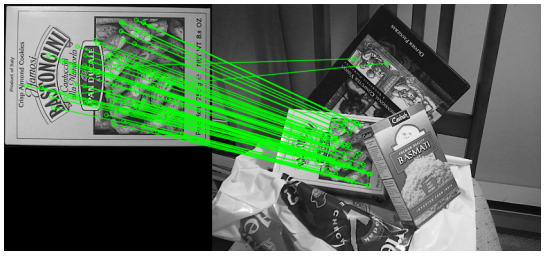


Fig. 3: Inliers after RANSAC verification

Comparing Feature Matching Algorithms on Real Data

This section tests three more feature matching algorithms alongside SIFT: *SURF*, *ORB*, and *FAST+BRIEF*. We want to see how they stack up against our SIFT baseline in different situations.

We wrote C++ classes for each method (`SurfFeatureTracker`, `OrbFeatureTracker`, `FastFeatureTracker`). One important note: matching works differently for floating-point descriptors (SIFT/SURF) and binary ones (ORB/BRIEF). For SIFT/SURF we use Euclidean distance, but for ORB/BRIEF we switch to Hamming distance, which is faster and more appropriate for binary data.

We ran tests in two scenarios: Pair of Frames, where two images with a big viewpoint change (rotation and scale), and Real Datasets (Video), where continuous video to see how they handle small, sequential motions.

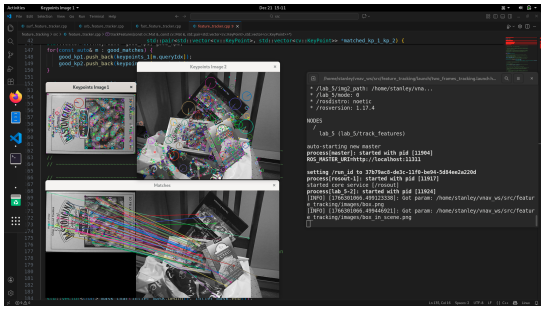


Fig. 4: Demo: processing two frames

Pair of Frames: We ran this command four times, once for each method (SIFT, SURF, ORB, FAST+BRIEF):

```
roslaunch lab_5 two_frames_tracking.launch
↪  descriptor:=<METHOD>
```

The terminal outputs were in the appendix B-D1. From these logs, we analyse and compare the *floating-point* and the *binary* methods:

1) Floating-point methods (SIFT/SURF): Both did well with the big viewpoint change. SIFT had a solid 78.6% inlier ratio. SURF found more keypoints but was a bit less accurate (66.7%).
2) Binary methods (ORB/FAST+BRIEF): These struggled more with the large baseline. *FAST+BRIEF* found tons of keypoints but had a low inlier rate (41.5%). BRIEF isn't built for big rotations, which explains it. At the same time, *ORB* gave very few good matches (only 8), likely because Lowe's Ratio Test was too strict with

Hamming distance on this texture. But the matches that passed were very accurate (87.5% inliers).

Real Datasets: Next, we tested on two video datasets with this command (run for each method):

```
roslaunch lab_5 video_tracking.launch
↪  path_to_dataset:=/home/$USER/Desktop/<DAT
↪  ASET>.bag descriptor:=<METHOD>
```

`30fps_424x240_2018-10-01-18-35-06.bag`

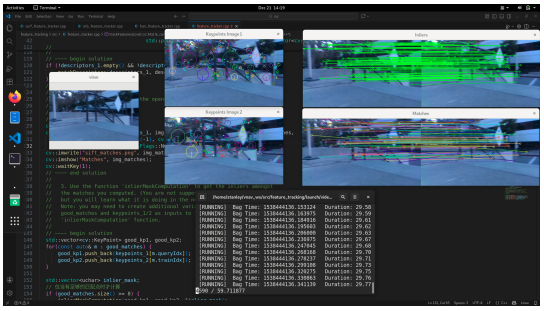


Fig. 5: Processing the 30fps video

And the logs are in appendix B-D2. Through whose data, we found out two results for Video Sequences. Binary methods are more precise here. In continuous video, ORB and FAST+BRIEF had higher inlier ratios (88%) than SIFT/SURF (75-78%). For small, sequential motions, binary descriptors gave cleaner matches. Feature density varies a lot. SURF found the most features (over 1000 per frame) and kept the most inliers (345), giving a dense tracking field. This is good for robustness but slow. While SIFT was sparse (50 inliers), which could be risky if the scene lacks texture, FAST+BRIEF struck a balance: good density (273 inliers) with high accuracy. Speed differences were obvious. ORB and FAST+BRIEF ran fast, suitable for real-time use. SURF felt slow because it processes so many floating-point descriptors per frame.

This came out a conclusion. For tasks with big viewpoint changes (like loop closure), use *SIFT* or *SURF*, while for real-time visual odometry with small motions, *ORB* or *FAST+BRIEF* gives the best speed/accuracy trade-off.

Lucas Kanade Feature Tracking

This final part implements the Lucas-Kanade (LK) tracker, which works differently from the descriptor-based methods (SIFT, SURF, ORB) we tested earlier.

Instead of finding and matching features between frames, LK tracks them using the *Brightness Constancy Assumption*:

$$I(x, y, t) \approx I(x + dx, y + dy, t + dt)$$

It calculates the motion vector (u, v) by minimizing the error in a small window, using image gradients. We used OpenCV's `cv::calcOpticalFlowPyrLK` with an image pyramid to handle larger movements. Tracking starts with *Good Features to Track (GFTT)* in the first frame.

We ran the LK tracker on the same two real-world datasets. The code is in appendix C and the results are in Fig. 6.

Dataset: `30fps_424x240_2018-10-01-18-35-06.bag`

```
[INFO] [1766307648.340722934]: LK Stats:
↪  Matches: 776 Inliers: 756 Ratio: 0.974227
```

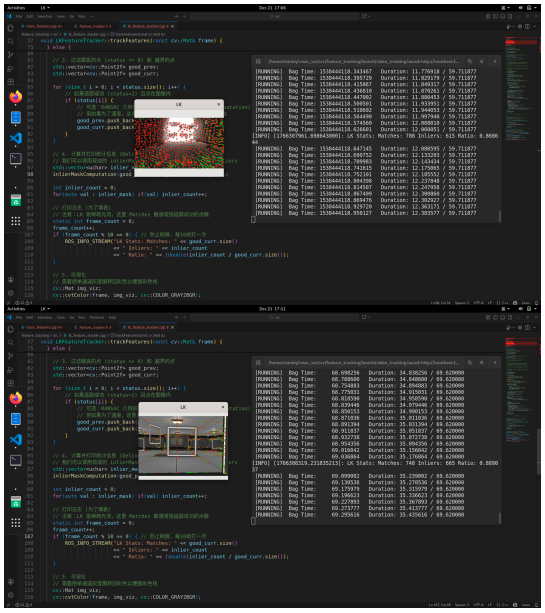


Fig. 6: Lucas Kanade tracking results on real datasets

Dataset: `vnav-lab5-smooth-trajectory.bag`

```
[INFO] [1766308353.398518896]: LK Stats:
↪  Matches: 653 Inliers: 634 Ratio: 0.970904
```

What we observed that the highest inlier ratio is over 97%, LK had the cleanest matches (97.4% inliers). Descriptor matching (like SIFT/ORB) searches globally, so a feature in one corner might match a similar-looking feature in the opposite corner (an outlier). LK only searches locally, assuming features don't move far between frames. This naturally avoids those big mistakes in smooth videos. We didn't log exact timings, but LK is faster because it skips computing descriptors and doing brute-force matching. It just uses image gradients, which is great for real-time needs (like drone control). LK worked so well here because our videos had smooth, small motions (30fps). If the camera moved too fast or frames were skipped, LK would lose track. Descriptor-based methods like SIFT/ORB are better at handling big jumps.

Lucas-Kanade gives the most accurate and efficient tracking for *smooth, continuous video*. But it can't re-lost features or match across big viewpoint changes—for that, you still need descriptor-based methods like ORB or SIFT.

Optical Flow

The Farneback algorithm estimates optical flow for *every pixel* in the image, unlike the sparse methods (SIFT/LK) we tested before. You can see our implementation C. In the figures above, we visualize the flow field in HSV colors: *Hue* indicates direction of motion, and *Value* shows how fast the pixel is moving.

The LK tracker is efficient for following specific points, which works well for tasks like SLAM. Farneback gives a motion vector at each pixel, which paints a fuller picture of how everything in the scene is moving—you can even make out the shape of the box from the flow. The downside is speed: calculating flow everywhere is heavy, and we noticed a clear slowdown compared to the real-time LK tracker.

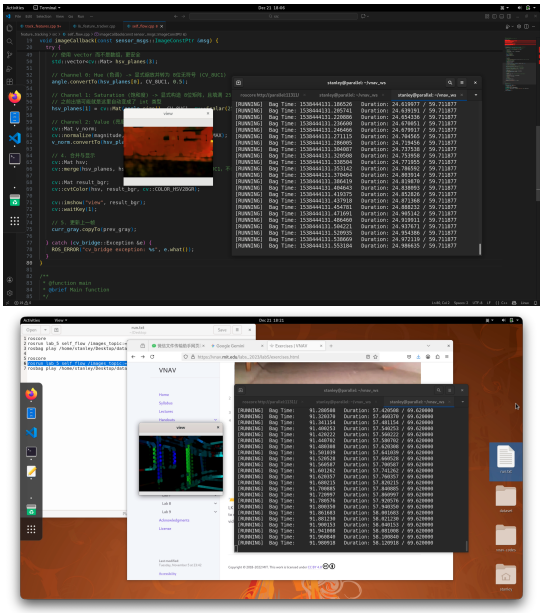


Fig. 7: Dense Optical Flow using Farneback's method

III. Some words

Lab 5 was a special experiment to me. I didn't have to do almost all stuffs on my own Ubuntu Desktop, because my teammate finally could do this on his arm based MacOS. Things became much easier to me like a month ago, and I finally had time to do my own stuff. However, everything is unimportant before "however", I found that I still couldn't handle my time confidently. There will always be endless stuffs, from nowhere, waiting for me to handle. That is to say, I was still busy and have no time to truelly relax.

Then, I relieved. Since I could never finish all these things, just leave them ahand. I need to relax, to have my own time. The deadline is tomorrow? That's fine, I still have a day and a half to do that. There's no need to be hurry. This is a lesson to me, a very important lesson.

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:

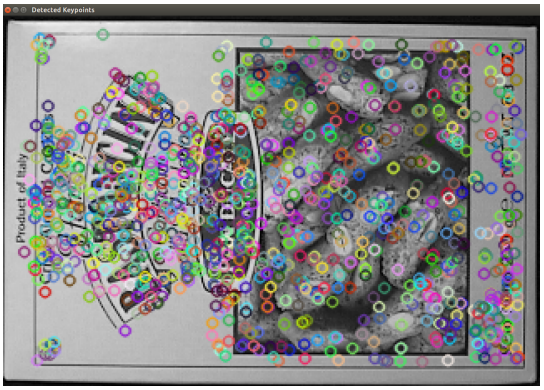
- The stub `SiftFeatureTracker::detectKeypoints()` and `SiftFeatureTracker::describeKeypoints()` in `lab5/feature_tracking/src/sift_feature_tracker.cpp`
- The first part of `FeatureTracker::trackFeatures()` in `lab5/feature_tracking/src/feature_tracker.cpp`

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.
}
```

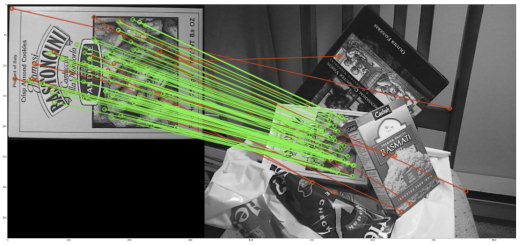
  Likewise, `good_matches` is a pointer so to add to it you will need `good_matches->push_back(...)`.

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 BruteForce 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.

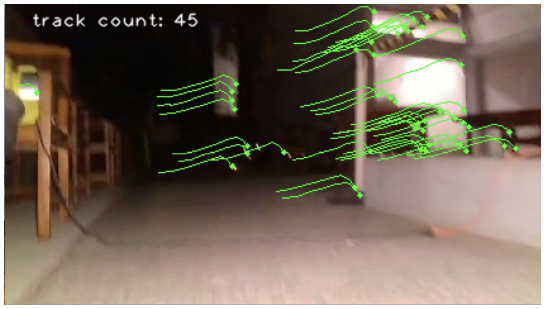
We got the terminal logs to be:

```
Avg. Keypoints 1 Size: 603
Avg. Keypoints 2 Size: 969
Avg. Number of matches: 603
Avg. Number of good matches: 98
Avg. Number of Inliers: 77
Avg. Inliers ratio: 0.785714
Num. of samples: 1
```,
caption: [SIFT output],
kind: "code",
supplement: [Log],

```
Avg. Keypoints 1 Size: 997
Avg. Keypoints 2 Size: 1822
Avg. Number of matches: 997
Avg. Number of good matches: 135
Avg. Number of Inliers: 90
Avg. Inliers ratio: 0.666667
Num. of samples: 1
```

```
Avg. Keypoints 1 Size: 500
Avg. Keypoints 2 Size: 500
Avg. Number of matches: 500
Avg. Number of good matches: 8
Avg. Number of Inliers: 7
Avg. Inliers ratio: 0.875
Num. of samples: 1
```,
caption: [ORB output],
kind: "code",
supplement: [Log],

```
Avg. Keypoints 1 Size: 2266
Avg. Keypoints 2 Size: 3653
Avg. Number of matches: 2266
Avg. Number of good matches: 41
Avg. Number of Inliers: 17
Avg. Inliers ratio: 0.414634
Num. of samples: 1
```,
caption: [FAST+BRIEF output],
kind: "code",
supplement: [Log],

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.

The logs of Real Datasets are:

```
Avg. Keypoints 1 Size: 321.819
Avg. Keypoints 2 Size: 321.602
Avg. Number of matches: 321.819
Avg. Number of good matches: 160.974
Avg. Number of Inliers: 149.751
Avg. Inliers ratio: 0.928412
Num. of samples: 425
```,
caption: [SIFT output],
kind: "code",
supplement: [Log],

```
Avg. Keypoints 1 Size: 389.408
Avg. Keypoints 2 Size: 389.252
Avg. Number of matches: 389.408
Avg. Number of good matches: 234.594
Avg. Number of Inliers: 216.93
Avg. Inliers ratio: 0.926189
Num. of samples: 488
```,
caption: [SURF output],

```
kind: "code",
supplement: [Log],
```

```
Avg. Keypoints 1 Size: 268.751
Avg. Keypoints 2 Size: 268.612
Avg. Number of matches: 268.751
Avg. Number of good matches: 166.031
Avg. Number of Inliers: 160.54
Avg. Inliers ratio: 0.964453
Num. of samples: 541
```,
caption: [ORB output],
kind: "code",
supplement: [Log],

```
Avg. Keypoints 1 Size: 741.136
Avg. Keypoints 2 Size: 741.256
Avg. Number of matches: 741.136
Avg. Number of good matches: 499.004
Avg. Number of Inliers: 465.635
Avg. Inliers ratio: 0.928044
Num. of samples: 477
```,
caption: [FAST+BRIEF output],
kind: "code",
supplement: [Log],

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)

```
Avg. Keypoints 1 Size: 137.468
Avg. Keypoints 2 Size: 136.608
Avg. Number of matches: 137.468
Avg. Number of good matches: 67.9915
Avg. Number of Inliers: 50.8766
Avg. Inliers ratio: 0.75093
Num. of samples: 235
```,
caption: [SIFT output],
kind: "code",
supplement: [Log],

```
Avg. Keypoints 1 Size: 1009.98
Avg. Keypoints 2 Size: 1006.7
Avg. Number of matches: 1009.98
Avg. Number of good matches: 457.648
Avg. Number of Inliers: 345.381
Avg. Inliers ratio: 0.778896
Num. of samples: 310
```,
caption: [SURF output],
kind: "code",
supplement: [Log],

```
Avg. Keypoints 1 Size: 258.294
Avg. Keypoints 2 Size: 257.972
Avg. Number of matches: 258.294
Avg. Number of good matches: 133.933
Avg. Number of Inliers: 116.456
Avg. Inliers ratio: 0.878598
Num. of samples: 465
```,
caption: [ORB output],
kind: "code",
supplement: [Log],

```
Avg. Keypoints 1 Size: 524.664
Avg. Keypoints 2 Size: 523.742
Avg. Number of matches: 524.664
Avg. Number of good matches: 313.428
Avg. Number of Inliers: 273.366
Avg. Inliers ratio: 0.877645
Num. of samples: 465
```,
caption: [FAST+BRIEF output],
kind: "code",
supplement: [Log],

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.

APPENDIX C
SOURCE CODE

sift_feature_tracker.cpp

```cpp
#include "sift_feature_tracker.h"

#include <vector>
#include <glog/logging.h>
#include <opencv2/features2d.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/calib3d.hpp>

using namespace cv;
using namespace cv::xfeatures2d;

/**
   Sift feature tracker Constructor.
*/
SiftFeatureTracker::SiftFeatureTracker()
  : FeatureTracker(),
    detector(SIFT::create()) {}

/** This function detects keypoints in an image.
    @param[in] img Image input where to detect keypoints.
    @param[out] keypoints List of keypoints detected on the given image.
*/
void SiftFeatureTracker::detectKeypoints(
  const cv::Mat& img,
  std::vector<KeyPoint>* keypoints
) const {
  CHECK_NOTNULL(keypoints);

  detector->detect(img, *keypoints);
}

/** This function describes keypoints in an image.
    @param[in] img Image used to detect the keypoints.
    @param[in, out] keypoints List of keypoints detected on the image. Depending
    on the detector used, some keypoints might be added or removed.
    @param[out] descriptors List of descriptors for the given keypoints.
*/
void SiftFeatureTracker::describeKeypoints(
  const cv::Mat& img,
  std::vector<KeyPoint>* keypoints,
  cv::Mat* descriptors
) const {
  CHECK_NOTNULL(keypoints);
  CHECK_NOTNULL(descriptors);

  detector->compute(img, *keypoints, *descriptors);
}

/** This function matches descriptors.
    @param[in] descriptors_1 First list of descriptors.
    @param[in] descriptors_2 Second list of descriptors.
    @param[out] matches List of k best matches between descriptors.
    @param[out] good_matches List of descriptors classified as "good"
*/
void SiftFeatureTracker::matchDescriptors(
  const cv::Mat& descriptors_1,
  const cv::Mat& descriptors_2,
  std::vector<std::vector<DMatch>>* matches,
  std::vector<cv::DMatch>* good_matches
) const {
  CHECK_NOTNULL(matches);
  FlannBasedMatcher flann_matcher;

  flann_matcher.knnMatch(descriptors_1, descriptors_2, *matches, 2);

  const float ratio_thresh = 0.8f;
  for (size_t i = 0; i < matches->size(); i++) {
    if ((*matches)[i].size() >= 2) {
      if((*matches)[i][0].distance < ratio_thresh * (*matches)[i][1].distance) {
        good_matches->push_back((*matches)[i][0]);
      }
    }
```

```
  }
}
```

feature_tracker.cpp

```
#include "feature_tracker.h"

#include <vector>
#include <numeric>

#include <gflags/gflags.h>
#include <glog/logging.h>

#include <opencv2/features2d.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/calib3d.hpp>

#include <ros/ros.h>

using namespace cv;
using namespace cv::xfeatures2d;

/** This is the main tracking function, given two images, it detects,
 * describes and matches features.
 * We will be modifying this function incrementally to plot different figures
 * and compute different statistics.
 @param[in] img_1, img_2 Images where to track features.
 @param[out] matched_kp_1_kp_2 pair of vectors of keypoints with the same size
 so that matched_kp_1_kp_2.first[i] matches with matched_kp_1_kp_2.second[i].
*/
void FeatureTracker::trackFeatures(
  const cv::Mat &img_1,
  const cv::Mat &img_2,
  std::pair<std::vector<cv::KeyPoint>,
  std::vector<cv::KeyPoint>> *matched_kp_1_kp_2
) {

  std::vector<KeyPoint> keypoints_1, keypoints_2;

  detectKeypoints(img_1, &keypoints_1);
  detectKeypoints(img_2, &keypoints_2);

  cv::Mat img_1_kp, img_2_kp;

  cv::drawKeypoints(img_1, keypoints_1, img_1_kp, cv::Scalar::all(-1),
  ↪  cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
  cv::drawKeypoints(img_2, keypoints_2, img_2_kp, cv::Scalar::all(-1),
  ↪  cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

  cv::imwrite("sift_keypoints_1.png", img_1_kp);
  cv::imwrite("sift_keypoints_2.png", img_2_kp);

  cv::imshow("Keypoints Image 1", img_1_kp);
  cv::imshow("Keypoints Image 2", img_2_kp);
  cv::waitKey(50);

  cv::Mat descriptors_1, descriptors_2;
  describeKeypoints(img_1, &keypoints_1, &descriptors_1);
  describeKeypoints(img_2, &keypoints_2, &descriptors_2);

  std::vector<std::vector<DMatch>> matches;
  std::vector<DMatch> good_matches;

  if (!descriptors_1.empty() && !descriptors_2.empty()) {
      matchDescriptors(descriptors_1, descriptors_2, &matches, &good_matches);
  }

  cv::Mat img_matches;
  cv::drawMatches(
    img_1, keypoints_1, img_2, keypoints_2, good_matches, img_matches,
    cv::Scalar::all(-1), cv::Scalar::all(-1), std::vector<char>(),
    cv::DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS
  );
  cv::imwrite("sift_matches.png", img_matches);
  cv::imshow("Matches", img_matches);
```

```
  cv::waitKey(0);

  std::vector<cv::KeyPoint> good_kp1, good_kp2;
  for(const auto& m : good_matches) {
    good_kp1.push_back(keypoints_1[m.queryIdx]);
    good_kp2.push_back(keypoints_2[m.trainIdx]);
  }

  std::vector<uchar> inlier_mask;
  if (good_matches.size() >= 8) {
    inlierMaskComputation(good_kp1, good_kp2, &inlier_mask);
  } else {
    inlier_mask.resize(good_matches.size(), 0);
  }

  unsigned int num_inliers = 0;
  for(auto mask_val : inlier_mask) {
    if(mask_val) num_inliers++;
  }

  cv::Mat img_inliers;

  std::vector<char> mask_char(inlier_mask.begin(), inlier_mask.end());

  cv::drawMatches(
    img_1, keypoints_1, img_2, keypoints_2, good_matches, img_inliers,
    cv::Scalar(0, 255, 0),
    cv::Scalar(0, 0, 255),
    mask_char,
    cv::DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS
  );

  cv::imwrite("sift_inliers.png", img_inliers);
  cv::imshow("Inliers", img_inliers);
  cv::waitKey(50);

  float const new_num_samples = static_cast<float>(num_samples_) + 1.0f;
  float const old_num_samples = static_cast<float>(num_samples_);
  avg_num_keypoints_img1_ = (avg_num_keypoints_img1_ * old_num_samples +
  ↪  static_cast<float>(keypoints_1.size())) / new_num_samples;
  avg_num_keypoints_img2_ = (avg_num_keypoints_img2_ * old_num_samples +
  ↪  static_cast<float>(keypoints_2.size())) / new_num_samples;
  avg_num_matches_ = (avg_num_matches_ * old_num_samples + static_cast<float>(matches.size())) /
  ↪  new_num_samples;
  avg_num_good_matches_ = (avg_num_good_matches_ * old_num_samples +
  ↪  static_cast<float>(good_matches.size())) / new_num_samples;
  avg_num_inliers_ = (avg_num_inliers_ * old_num_samples + static_cast<float>(num_inliers)) /
  ↪  new_num_samples;
  avg_inlier_ratio_ =
      (avg_inlier_ratio_ * old_num_samples + (static_cast<float>(num_inliers) /
      ↪  static_cast<float>(good_matches.size()))) / new_num_samples;
  ++num_samples_;
}

/** Compute Inlier Mask out of the given matched keypoints.
   *  Both keypoints_1 and keypoints_2 input parameters must be ordered by match
   * i.e. keypoints_1[0] has been matched to keypoints_2[0].
   * Therefore, both keypoints vectors must have the same length.
    @param[in] keypoints_1 List of keypoints detected on the first image.
    @param[in] keypoints_2 List of keypoints detected on the second image.
    @param[out] inlier_mask Mask indicating inliers (1) from outliers (0).
  */
void FeatureTracker::inlierMaskComputation(
  const std::vector<KeyPoint> &keypoints_1,
  const std::vector<KeyPoint> &keypoints_2,
  std::vector<uchar> *inlier_mask
) const {
  CHECK_NOTNULL(inlier_mask);
  const size_t size = keypoints_1.size();
  CHECK_EQ(keypoints_2.size(), size) << "Size of keypoint vectors "
                                        "should be the same!";

  std::vector<Point2f> pts1(size);
  std::vector<Point2f> pts2(size);
  for (size_t i = 0; i < keypoints_1.size(); i++) {
    pts1[i] = keypoints_1[i].pt;
    pts2[i] = keypoints_2[i].pt;
```

```
  }

  static constexpr double max_dist_from_epi_line_in_px = 3.0;
  static constexpr double confidence_prob = 0.99;
  try {
    findFundamentalMat(pts1, pts2, CV_FM_RANSAC, max_dist_from_epi_line_in_px, confidence_prob,
    ↪  *inlier_mask);
  } catch (...) {
    ROS_WARN("Inlier Mask could not be computed, this can happen if there"
             "are not enough features tracked.");
  }
}

/** Example of function to draw matches. Feel free to re-use this example or
 *  create your own. You will need to modify it in order to plot the different
 *  figures. You can add more functions to this class if needed.
 */
void FeatureTracker::drawMatches(
  const cv::Mat &img_1,
  const cv::Mat &img_2,
  const std::vector<KeyPoint> &keypoints_1,
  const std::vector<KeyPoint> &keypoints_2,
  const std::vector<std::vector<DMatch>> &matches
) {
  cv::namedWindow("tracked_features", cv::WINDOW_NORMAL);
  cv::Mat img_matches;
  cv::drawMatches(
    img_1,
    keypoints_1,
    img_2,
    keypoints_2,
    matches,
    img_matches,
    Scalar::all(-1),
    Scalar::all(-1),
    std::vector<std::vector<char>>(),
    DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS
  );

  imshow("tracked_features", img_matches);

  while (ros::ok() && waitKey(10) == -1) {}
}

void FeatureTracker::printStats() const {
  std::cout << "Avg. Keypoints 1 Size: " << avg_num_keypoints_img1_ << std::endl;
  std::cout << "Avg. Keypoints 2 Size: " << avg_num_keypoints_img2_ << std::endl;
  std::cout << "Avg. Number of matches: " << avg_num_matches_ << std::endl;
  std::cout << "Avg. Number of good matches: " << avg_num_good_matches_ << std::endl;
  std::cout << "Avg. Number of Inliers: " << avg_num_inliers_ << std::endl;
  std::cout << "Avg. Inliers ratio: " << avg_inlier_ratio_ << std::endl;
  std::cout << "Num. of samples: " << num_samples_ << std::endl;
}
```

surf_feature_tracker.cpp

```
#include "surf_feature_tracker.h"

#include <vector>
#include <glog/logging.h>
#include <opencv2/features2d.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/calib3d.hpp>

using namespace cv;
using namespace cv::xfeatures2d;

/**
   Surf feature tracker Constructor.
*/
SurfFeatureTracker::SurfFeatureTracker()
  : FeatureTracker(),
    detector(SURF::create()) {

}
```

```
/** This function detects keypoints in an image.
    @param[in] img Image input where to detect keypoints.
    @param[out] keypoints List of keypoints detected on the given image.
*/
void SurfFeatureTracker::detectKeypoints(
  const cv::Mat& img,
  std::vector<KeyPoint>* keypoints
) const {
  CHECK_NOTNULL(keypoints);

  detector->detect(img, *keypoints);
}

/** This function describes keypoints in an image.
    @param[in] img Image used to detect the keypoints.
    @param[in, out] keypoints List of keypoints detected on the image. Depending
    on the detector used some keypoints might be added or removed.
    @param[out] descriptors List of descriptors for the given keypoints.
*/
void SurfFeatureTracker::describeKeypoints(
  const cv::Mat& img,
  std::vector<KeyPoint>* keypoints,
  cv::Mat* descriptors
) const {
  CHECK_NOTNULL(keypoints);
  CHECK_NOTNULL(descriptors);

  detector->compute(img, *keypoints, *descriptors)
}

/** This function matches descriptors.
    @param[in] descriptors_1 First list of descriptors.
    @param[in] descriptors_2 Second list of descriptors.
    @param[out] matches List of k best matches between descriptors.
    @param[out] good_matches List of descriptors classified as "good"
*/
void SurfFeatureTracker::matchDescriptors(
  const cv::Mat& descriptors_1,
  const cv::Mat& descriptors_2,
  std::vector<std::vector<DMatch>>* matches,
  std::vector<cv::DMatch>* good_matches
) const {
  CHECK_NOTNULL(matches);

  FlannBasedMatcher flann_matcher;
  // 1. KNN Match (k=2)
  if (!descriptors_1.empty() && !descriptors_2.empty()) {
    flann_matcher.knnMatch(descriptors_1, descriptors_2, *matches, 2);
  }

  // 2. Lowe's Ratio Test
  const float ratio_thresh = 0.8f;
  for (size_t i = 0; i < matches->size(); i++) {
    if ((*matches)[i].size() >= 2) {
      if ((*matches)[i][0].distance < ratio_thresh * (*matches)[i][1].distance) {
        good_matches->push_back((*matches)[i][0]);
      }
    }
  }
}
```

orb_feature_tracker.cpp

```
#include "orb_feature_tracker.h"
#include <opencv2/features2d.hpp>

using namespace cv;
using namespace cv::xfeatures2d;

OrbFeatureTracker::OrbFeatureTracker()
    : detector(ORB::create(500, 1.2f, 1)) {}

void OrbFeatureTracker::detectKeypoints(
  const cv::Mat &img,
  std::vector<cv::KeyPoint> *keypoints
) const {
```

```
  detector->detect(img, *keypoints);
}

void OrbFeatureTracker::describeKeypoints(
  const cv::Mat &img,
  std::vector<cv::KeyPoint> *keypoints,
  cv::Mat *descriptors
) const {
  detector->compute(img, *keypoints, *descriptors);
}

void OrbFeatureTracker::matchDescriptors(
  const cv::Mat &descriptors_1,
  const cv::Mat &descriptors_2,
  std::vector<std::vector<cv::DMatch>> *matches,
  std::vector<cv::DMatch> *good_matches
) const {
  cv::BFMatcher matcher(cv::NORM_HAMMING);

  if (!descriptors_1.empty() && !descriptors_2.empty()) {
    matcher.knnMatch(descriptors_1, descriptors_2, *matches, 2);
  }

  // Ratio Test
  const float ratio_thresh = 0.8f;
  for (size_t i = 0; i < matches->size(); i++) {
    if ((*matches)[i].size() >= 2) {
      if ((*matches)[i][0].distance < ratio_thresh * (*matches)[i][1].distance) {
        good_matches->push_back((*matches)[i][0]);
      }
    }
  }
}
```

fast_feature_tracker.cpp

```
#include "fast_feature_tracker.h"
#include <opencv2/features2d.hpp>

#include <opencv2/xfeatures2d.hpp>

using namespace cv;
using namespace cv::xfeatures2d;

FastFeatureTracker::FastFeatureTracker()
  : detector(FastFeatureDetector::create()) {}

void FastFeatureTracker::detectKeypoints(
  const cv::Mat &img,
  std::vector<cv::KeyPoint> *keypoints
) const {
  detector->detect(img, *keypoints);
}

void FastFeatureTracker::describeKeypoints(
  const cv::Mat &img,
  std::vector<cv::KeyPoint> *keypoints,
  cv::Mat *descriptors
) const {
  Ptr<BriefDescriptorExtractor> extractor = BriefDescriptorExtractor::create();
  extractor->compute(img, *keypoints, *descriptors);
}

void FastFeatureTracker::matchDescriptors(
  const cv::Mat &descriptors_1,
  const cv::Mat &descriptors_2,
  std::vector<std::vector<cv::DMatch>> *matches,
  std::vector<cv::DMatch> *good_matches
) const {
  cv::BFMatcher matcher(cv::NORM_HAMMING);

  if (!descriptors_1.empty() && !descriptors_2.empty()) {
    matcher.knnMatch(descriptors_1, descriptors_2, *matches, 2);
  }

  // Ratio Test
```

```
  const float ratio_thresh = 0.8f;
  for (size_t i = 0; i < matches->size(); i++) {
    if ((*matches)[i].size() >= 2) {
      if ((*matches)[i][0].distance < ratio_thresh * (*matches)[i][1].distance) {
        good_matches->push_back((*matches)[i][0]);
      }
    }
  }
}
```

lk_feature_tracker.cpp

```
#include "lk_feature_tracker.h"

#include <numeric>
#include <vector>

#include <glog/logging.h>

#include <opencv2/calib3d.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/video/tracking.hpp>

#include <ros/ros.h>

using namespace cv;
using namespace cv::xfeatures2d;

/**
   LK feature tracker Constructor.
*/
LKFeatureTracker::LKFeatureTracker() {
  // Feel free to modify if you want!
  cv::namedWindow(window_name_, cv::WINDOW_NORMAL);
}

LKFeatureTracker::~LKFeatureTracker() {
  // Feel free to modify if you want!
  cv::destroyWindow(window_name_);
}

/** This is the main tracking function, given two images, it detects,
 * describes and matches features.
 * We will be modifying this function incrementally to plot different figures
 * and compute different statistics.
 @param[in] frame Current image frame
*/
void LKFeatureTracker::trackFeatures(const cv::Mat& frame) {
  // 1. Initialize the keypoint container for the current frame
  std::vector<cv::Point2f> curr_points;
  std::vector<uchar> status;
  std::vector<float> err;

  // 2. Logical branch: initialization vs tracing
  if (prev_corners_.empty()) {
    cv::goodFeaturesToTrack(frame, prev_corners_, 1000, 0.01, 10);

    frame.copyTo(prev_frame_);
  } else {
    cv::calcOpticalFlowPyrLK(
      prev_frame_, frame, prev_corners_, curr_points,
      tatus, err, cv::Size(21, 21), 3
    );

    // 3. Filter out lost points (status == 0) and out-of-bounds points
    std::vector<cv::Point2f> good_prev;
    std::vector<cv::Point2f> good_curr;

    for (size_t i = 0; i < status.size(); i++) {
      if (status[i]) {
        good_prev.push_back(prev_corners_[i]);
        good_curr.push_back(curr_points[i]);
      }
    }
```

```
    // 4. Calculate and print the statistics
    std::vector<uchar> inlier_mask;
    inlierMaskComputation(good_prev, good_curr, &inlier_mask);

    int inlier_count = 0;
    for(auto val : inlier_mask) if(val) inlier_count++;

    // Log (for form filling)
    static int frame_count = 0;
    frame_count++;
    if (frame_count % 10 == 0) {
      ROS_INFO_STREAM("LK Stats: Matches: " << good_curr.size()
                      << " Inliers: " << inlier_count
                      << " Ratio: " << (double)inlier_count / good_curr.size());
    }

    // 5. Visualization
    cv::Mat img_viz;
    cv::cvtColor(frame, img_viz, cv::COLOR_GRAY2BGR);
    show(img_viz, good_prev, good_curr);

    // 6. Compensate Mechanism
    if (good_curr.size() < 800) {
      std::vector<cv::Point2f> new_points;
      cv::Mat mask = cv::Mat::zeros(frame.size(), CV_8UC1);
      mask.setTo(cv::Scalar(255));
      for(auto& p : good_curr) cv::circle(mask, p, 10, cv::Scalar(0), -1);

      cv::goodFeaturesToTrack(frame, new_points, 1000 - good_curr.size(), 0.01, 10, mask);
      good_curr.insert(good_curr.end(), new_points.begin(), new_points.end());
    }

    // 7. Refresh stats
    prev_corners_ = good_curr;
    frame.copyTo(prev_frame_);
  }
}

/** Display image with tracked features from prev to curr on the image
 * corresponding to 'frame'
 * @param[in] frame The current image frame, to draw the feature track on
 * @param[in] prev The previous set of keypoints
 * @param[in] curr The set of keypoints for the current frame
 */
void LKFeatureTracker::show(
  const cv::Mat& frame,
  std::vector<cv::Point2f>& prev,
  std::vector<cv::Point2f>& curr
) {
  cv::Mat viz_img = frame.clone();

  for (size_t i = 0; i < curr.size(); i++) {
    cv::line(viz_img, prev[i], curr[i], cv::Scalar(0, 255, 0), 2);
    cv::circle(viz_img, curr[i], 3, cv::Scalar(0, 0, 255), -1);
  }

  cv::imshow(window_name_, viz_img);
  cv::waitKey(1);
}

/** Compute Inlier Mask out of the given matched keypoints.
 @param[in] pts1 List of keypoints detected on the first image.
 @param[in] pts2 List of keypoints detected on the second image.
 @param[out] inlier_mask Mask indicating inliers (1) from outliers (0).
*/
void LKFeatureTracker::inlierMaskComputation(
  const std::vector<cv::Point2f>& pts1,
  const std::vector<cv::Point2f>& pts2,
  std::vector<uchar>* inlier_mask
) const {
  CHECK_NOTNULL(inlier_mask);

  static constexpr double max_dist_from_epi_line_in_px = 3.0;
  static constexpr double confidence_prob = 0.99;
  try {
    findFundamentalMat(
      pts1, pts2, CV_FM_RANSAC,
```

```
      max_dist_from_epi_line_in_px, confidence_prob,
      *inlier_mask
    );
  } catch(...) {
    ROS_WARN("Inlier Mask could not be computed, this can happen if there"
             "are not enough features tracked.");
  }
}
```

self_flow.cpp

```
#include <memory>
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>

#include <opencv2/core.hpp>
#include <opencv2/highgui/highgui.hpp>

#include <opencv2/imgproc.hpp>
#include <opencv2/video/tracking.hpp>

using namespace std;

/** imageCallback This function is called when a new image is published. */
void imageCallback(const sensor_msgs::ImageConstPtr &msg) {
  try {
    cv::Mat image = cv_bridge::toCvCopy(msg, "bgr8")->image;

    cv::Mat curr_gray;
    cv::cvtColor(image, curr_gray, cv::COLOR_BGR2GRAY);

    static cv::Mat prev_gray;

    if (prev_gray.empty()) {
      curr_gray.copyTo(prev_gray);
      return;
    }

    cv::Mat flow;
    cv::calcOpticalFlowFarneback(prev_gray, curr_gray, flow, 0.5, 3, 15, 3, 5, 1.2, 0);

    std::vector<cv::Mat> flow_parts;
    cv::split(flow, flow_parts);

    cv::Mat magnitude, angle;
    cv::cartToPolar(flow_parts[0], flow_parts[1], magnitude, angle, true);

    std::vector<cv::Mat> hsv_planes(3);

    angle.convertTo(hsv_planes[0], CV_8UC1, 0.5);

    hsv_planes[1] = cv::Mat(angle.size(), CV_8UC1, cv::Scalar(255));

    cv::Mat v_norm;
    cv::normalize(magnitude, v_norm, 0, 255, cv::NORM_MINMAX);
    v_norm.convertTo(hsv_planes[2], CV_8UC1);

    cv::Mat hsv;
    cv::merge(hsv_planes, hsv);

    cv::Mat result_bgr;
    cv::cvtColor(hsv, result_bgr, cv::COLOR_HSV2BGR);

    cv::imshow("view", result_bgr);
    cv::waitKey(1);

    curr_gray.copyTo(prev_gray);

  } catch (cv_bridge::Exception &e) {
    ROS_ERROR("cv_bridge exception: %s", e.what());
  }
}

/**
 * @function main
```

```
 * @brief Main function
 */
int main(int argc, char **argv) {
  ros::init(argc, argv, "optical_flow");
  ros::NodeHandle local_nh("~");

  cv::namedWindow("view", cv::WINDOW_NORMAL);
  image_transport::ImageTransport it(local_nh);
  image_transport::Subscriber sub = it.subscribe("/images_topic", 100, imageCallback);

  ros::spin();
  cv::destroyWindow("view");
  return EXIT_SUCCESS;
}
```