Lab 5 Report

Robotics Integration Group Project I

Yuwei ZHAO (23020036096)
Group #31 2025-12-20

Abstract

This report details the implementation and comparative analysis of a visual feature tracking, a critical component for Visual Odometry and SLAM systems. We developed a modular C++ framework to evaluate multiple feature descriptors — **SIFT**, **SURF**, **ORB**, and **FAST+BRIEF** — as well as the sparse Lucas-Kanade optical flow tracker. The pipeline includes feature detection, descriptor extraction, and correspondence establishment using Hamming distance. To ensure robust data association, we integrated outlier rejection mechanisms, specifically **Lowe's Ratio Test** and Geometric Verification using **RANSAC-based** Fundamental Matrix estimation.

Experimental results on both static image pairs and real-world video sequences highlight the trade-offs between computational efficiency and invariance to viewpoint changes, demonstrating that while SIFT/SURF offer superior robustness to large rotations, binary descriptors and optical flow provide the real-time performance necessary for high-frequency robotic control.

See Resources on github.com/RamessesN/Robotics_MIT.

1 Introduction

Visual feature tracking serves as the backbone for modern Visual Odometry (VO) and Simultaneous Localization and Mapping (SLAM) algorithms. The ability to robustly identify and track static landmarks across consecutive image frames allows a robot to estimate its ego-motion and reconstruct the 3D structure of its environment. However, this task is challenged by image noise, scale variations, illumination changes, and complex camera motions.

In this lab, we address these challenges by implementing complete feature tracking algorithms. We begin by establishing the geometric foundations through the robust estimation of the Fundamental Matrix using RANSAC, which enforces the epipolar constraint to reject spurious matches. Subsequently, we extend the pipeline to support various feature extraction methodologies. We compare **Detect-and-Describe** approaches (SIFT, SURF, ORB, FAST+BRIEF) against **Sparse Optical Flow** (Lucas-Kanade), evaluating their performance in terms of inlier ratios, feature density, and computational cost.

This comparative analysis provides a comprehensive understanding of how to select the appropriate tracking strategy based on specific application requirements, such as the need for rotation invariance versus the constraint of limited onboard computational resources.

2 Procedure

2.1 Individual Work

2.1.1 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 $s_\theta = 0$ the following exercises:

1. **Derive the equation describing the projection of the sphere onto the image plane.**

   $\because$ sphere radius is r and its center is $[0, 0, d]$

   $\therefore$ the equation of the sphere in the camera coordinate system is $X^2 + Y^2 + (Z - d)^2 = r^2$

   $\because$ focal length equals 1, pixel sizes $s_x = s_y = 1$, principal point is (0, 0) and zero skew $s_\theta = 0$

   $\therefore$ camera intrinsic parameter matrix K is an identity matrix.

   Therefore, we have $\begin{cases} u = \frac{f \cdot x}{Z} = \frac{x}{Z} \\ v = \frac{f \cdot y}{Z} = \frac{y}{Z} \end{cases}$

   That's $\begin{cases} x = uZ \\ y = vZ \end{cases}$

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

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

   Let $\Delta = b^2 - 4ac = (-2d)^2 - 4(u^2 + v^2 + 1)(d^2 - r^2) = 0$

   $\therefore d^2 - (u^2 + v^2 + 1)(d^2 - r^2) = 0$

   $\therefore (u^2 + v^2)(d^2 - r^2) = r^2$

   That's $u^2 + v^2 = \frac{r^2}{d^2 - r^2}$.

   Proof:

   $\because R_{\text{img}} = \sqrt{\frac{r^2}{d^2 - r^2}} = \frac{r}{\sqrt{d^2 - r^2}}$

   and $\tan^2 \theta = \sin^2 \frac{\theta}{1 - \sin^2 \theta}$

   $\therefore u^2 = \frac{\left(\frac{r}{d}\right)^2}{1 - \left(\frac{r}{d}\right)^2} = \frac{r^2}{d^2} - r^2$.

   That's the equation of the projection of the sphere on the img plane is:

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

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?**

Conclusion — When the center of the sphere is located at an arbitrary position, the projected shape of the sphere on the image plane is usually an ellipse. The projection is a circle only if the center of the sphere lies exactly on the optical axis.

Proof:

Let $\begin{cases} \text{Camera Center: } O(0,0,0) \\ \text{Sphere: radius- } r\text{, center- } c=[x_C,y_C,z_C]^T \\ \text{Point: } x=[u,v,1]^T(\text{WLOG } f=1) \end{cases}$

$\therefore$ we have $\sin(\theta)=\frac{r}{\|\theta\|} \Rightarrow \cos^2\theta = 1-\sin^2\theta = \frac{\|c\|^2-r^2}{\|c\|^2}$

$\because x\cdot c = \|x\|\ \|c\|\cos(\theta)$

$\therefore (x\cdot c)^2 = \|x\|^2\ \|c\|^2\cos^2(\theta) = \|x\|^2\ (\|c\|^2-r^2)$

Let $\mathrm{K}=\|c\|^2-r^2$, we have $(x\cdot c)^2 = \mathrm{K}\ \|x\|^2$

Put $x=[u,v,1]^T$ into the equation,

$\Rightarrow (x_Cu+y_Cv+z_C)^2 = \mathrm{K}(u^2+v^2+1)$

$\therefore \begin{cases} (x_Cu+y_Cv+z_C)^2 = x_C^2u^2+y_C^2v^2+z_C^2+2x_Cy_Cuv+2x_Cz_Cu+2y_Cz_Cv \\ \mathrm{K}(u^2+v^2+1) = \mathrm{K}u^2+\mathrm{K}v^2+\mathrm{K} \end{cases}$

$\therefore \left(\mathrm{K}-x_C^2\right)u^2 - 2x_Cy_Cuv + \left(\mathrm{K}-y_C^2\right)v^2 + ... = 0$

That's the shape of the projection is a **ellipse**.

2.1.2 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 $s_\theta = 0$. Complete the following exercises:

1. **Derive the generic expression of the vanishing point corresponding to two parallel 3D lines.**

   We have $p(\lambda) = p_0 + \lambda d$, which

   $\begin{cases} p(\lambda)=[X(\lambda),Y(\lambda),Z(\lambda)]^T \text{ are the points on the line} \\ p_0=[X_0,Y_0,Z_0]^T \text{ is the start point on the line} \\ d=[d_x,d_y,d_z]^T \text{ is the unit vector of the line} \\ \lambda\in\mathbb{R} \text{ is a parameter} \end{cases}$

   $\therefore \begin{cases} X(\lambda)=X_0+\lambda d_x \\ Y(\lambda)=Y_0+\lambda d_y \\ Z(\lambda)=Z_0+\lambda d_z \end{cases}$

   $\because u_{\text{img}} = \frac{X}{Z}, v_{\text{img}} = \frac{Y}{Z}$

$$\therefore \begin{cases} u_{\text{img}(\lambda)}=\frac{X_0+\lambda d_x}{Z_0+\lambda d_z} \\ v_{\text{img}(\lambda)}=\frac{Y_0+\lambda d_y}{Z_0+\lambda d_z} \end{cases}$$

$\therefore \lim_{\lambda\to\infty} u_{\text{img}}(\lambda) = \lim_{\lambda\to\infty} \frac{X_0+\lambda d_x}{Z_0+\lambda d_z} = \lim_{\lambda\to\infty} \frac{\frac{X_0}{\lambda}+d_x}{\frac{Z_0}{\lambda}+d_z} = \frac{d_x}{d_z}$

Similarly, we have $\lim_{\lambda\to\infty} v_{\text{img}} = \lim_{\lambda\to\infty} \frac{Y_0+\lambda d_y}{Z_0+\lambda d_z} = \frac{d_y}{d_z}$

That's **Vanishing Point** is $\left(\frac{d_x}{d_z}, \frac{d_y}{d_z}\right)$.

2. **Find (and prove mathematically) a condition under which 3D parallel lines remain parallel in the image plane.**

Conclusion — If two 3D lines are still parallel on the img plane, that means they don't have a finite coordinate intersection (i.e., they don't intersect, or they intersect at infinity). This happens when "denominator = 0", i.e.

$$d_z = 0$$

Proof:

WLOG there are two 3D parallel lines: L_1, L_2

which have the shared unit vector $d = \left[d_x, d_y, 0\right]^T$ but go through different points p_1, p_2

For Line L_1: origin $p_1 = [X_1, Y_1, Z_1]^T$

$$\therefore \text{we have } \begin{cases} X(\lambda)=X_1+\lambda d_x \\ Y(\lambda)=Y_1+\lambda d_y \\ Z(\lambda)=Z_1+\lambda\cdot 0=Z_1 \text{(which Z is a constant)} \end{cases}$$

$$\text{Project onto the image plane } (u,v), \text{ we have } \begin{cases} u=\frac{X_1+\lambda d_x}{Z_1}=\frac{X_1}{Z_1}+\lambda\frac{d_x}{Z_1} & ① \\ v=\frac{Y_1+\lambda d_y}{Z_1}=\frac{Y_1}{Z_1}+\lambda\frac{d_y}{Z_1} & ② \end{cases}$$

In order to find the line equation on the img plane(remove λ), assume that $d_x \neq 0$, we have $\lambda = \frac{Z_1 u - X_1}{d_x} \leftarrow ①$

Put into ②, we have $v = \frac{Y_1}{Z_1} + \frac{d_y}{Z_1}\left(\frac{Z_1 u - X_1}{d_x}\right)$

$\therefore v = \left(\frac{d_y}{d_x}\right)u + \left(\frac{Y_1}{Z_1} - \frac{X_1 d_y}{Z_1 d_x}\right)$

This is a "$v = mu + c$" -format line equation, which slope $m_1 = \frac{d_y}{d_x}$

For Line L_2: origin $p_2 = [X_2, Y_2, Z_2]^T$, unit vector is the same — d

Similarly, we have the line equation:

$v = \left(\frac{d_y}{d_x}\right)u + \left(\frac{Y_2}{Z_2} - \frac{X_2 d_y}{Z_2 d_x}\right)$, which slope $m_2 = \frac{d_y}{d_x}$

$\because m_1 = m_2 = \frac{d_y}{d_x} \Rightarrow$ The two lines on the img plane have the same slope

Therefore, when 3D lines are parallel to the img plane(i.e. $d_z = 0$), their projections remain plane on the img plane.

2.2 Team Work

2.2.1 Feature Descriptors (SIFT)

In this section, our primary objective is to implement the front-end of a feature-based visual odometry pipeline. The goal is to establish reliable data association between consecutive frames by detecting and matching distinctive visual features. Specifically, we utilize the **Scale-Invariant Feature Transform (SIFT)** for its robustness to scale and rotation changes. The implementation is structured to separate the specific feature algorithm from the general tracking pipeline, ensuring modularity for subsequent extensions.

For `sift_feature_tracker.cpp`:

In this file, we implemented the specific logic for the SIFT (Scale-Invariant Feature Transform) algorithm.

For `detectKeypoints` and `describeKeypoints`, we utilized the OpenCV `SIFT::create()` interface. SIFT detects keypoints by searching for local extrema in the Difference of Gaussians (DoG) scale space, ensuring scale invariance. The descriptors are then constructed as 128-dimensional vectors based on the gradient magnitudes and orientations in the local neighborhood, providing rotation invariance.

For `feature_tracker.cpp`:

We modified the `trackFeatures` function to orchestrate the pipeline. In the detection phase, we call `detectKeypoints` on both input images. Then with the command `roslaunch lab_5 two_frames_tracking.launch descriptor:=SIFT`, we obtain the visualization of local features.

As shown in the figure below, the keypoints (visualized with size and orientation) are densely distributed on high-contrast areas, such as the text and edges of the box, validating the effectiveness of the detector.

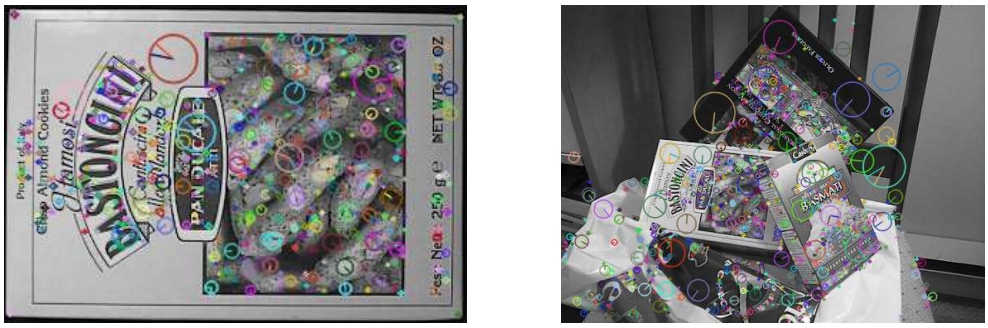

Figure 1: Local Feature Extraction (SIFT Keypoints)

2.2.2 Descriptor-based Feature Matching

In this section, we focus on establishing correspondences between the features detected in the previous step. We implemented the `matchDescriptors` function in `sift_feature_tracker.cpp`.

Instead of simple brute-force matching, which can be computationally expensive, we employed a **FLANN-based matcher** with K-Nearest Neighbors (KNN, $k = 2$). Crucially, to filter out ambiguous matches, we applied **Lowe's Ratio Test**.

The mathematical condition for this test is:

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

where d_1 and d_2 are the Euclidean distances to the closest and the second-closest descriptors, respectively. We set the threshold `ratio_thresh` to 0.8. This strategy rejects keypoints that are not discriminative enough (e.g., repetitive patterns on the box surface), significantly improving the matching precision.

The result of descriptor-based matching is shown in the figure below. While most matches correctly link the semantic parts of the box (e.g., corners to corners), there are still visible outliers (crossing lines) that violate the motion consistency. These outliers indicate that appearance-based matching alone is insufficient.

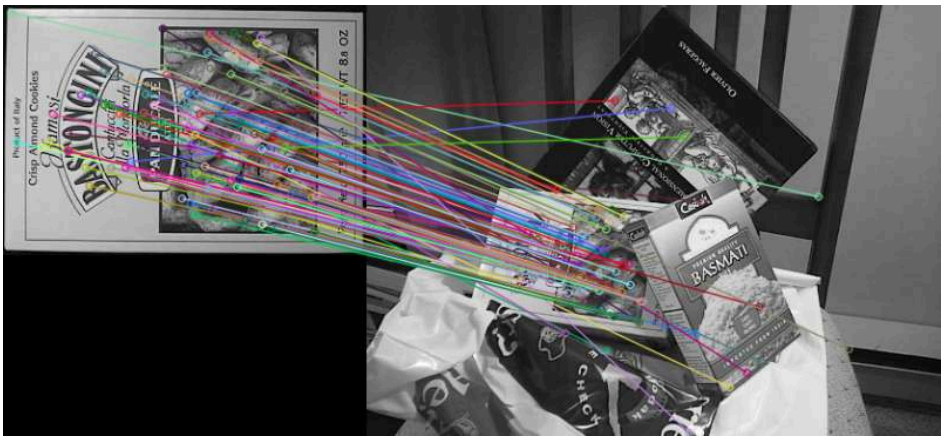

Figure 2: Descriptor-based Feature Matching (Filtered by Ratio Test)

2.2.3 Keypoint Matching Quality

To further improve the quality of correspondences, we implemented Geometric Verification in `feature_tracker.cpp` using the `inlierMaskComputation` function.

We utilize the **RANSAC (Random Sample Consensus)** algorithm to estimate the Fundamental Matrix (F) that best fits the matches obtained from the previous step. The underlying mathematical principle is the **Epipolar Constraint**:

$$p_2^T F p_1 = 0$$

where p_1 and p_2 are the homogeneous coordinates of matched points in the first and second images. Matches that do not satisfy this geometric constraint (within a threshold of 3.0 pixels) are classified as outliers and removed.

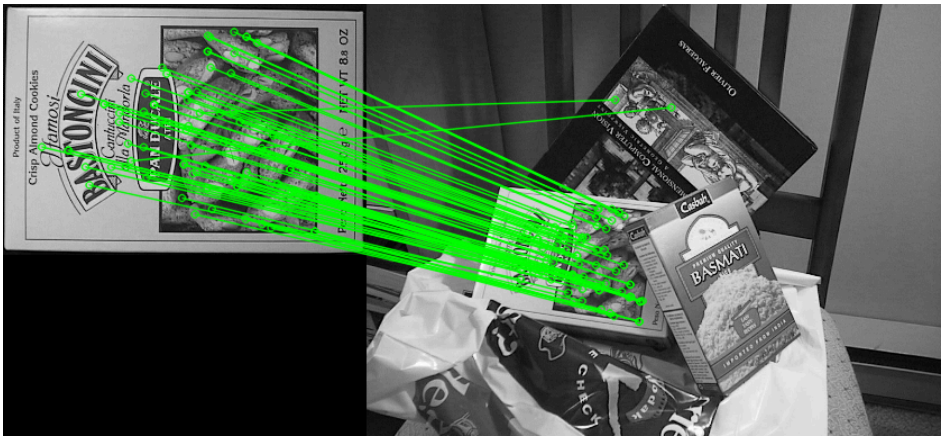

Figure 3: Inliers after RANSAC Geometric Verification

Comparing the figure above with the result in Deliverable 4, the erroneous "crossing lines" have been successfully removed. The remaining matches (Inliers) show a consistent flow field, representing the true motion of the camera relative to the box.

The metric print part in terminal log is as follows:

```log
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
```

From the terminal log, the quantitative statistics of the matching process are summarized in the table below:

Metric	Value
# of Keypoints in Img 1	603
# of Keypoints in Img 2	969
# of Matches	603
# of Good Matches	98
# of Inliers	77
Inlier Ratio	78.6%

Table 1: Matching Statistics for SIFT Descriptor

2.2.4 Comparing Feature Matching Algorithms on Real Data

In this section, we extend our feature tracking framework to evaluate three additional popular algorithms: *SIFT*, *SURF*, *ORB*, and *FAST* (coupled with *BRIEF* descriptors). The objective is to perform a comprehensive comparison of these methods against the SIFT baseline implemented in previous sections.

We implemented the corresponding derived classes (`SurfFeatureTracker`, `OrbFeatureTracker`, and `FastFeatureTracker`) in C++. A key implementation detail is the adjustment of the matching strategy: while floating-point descriptors (SIFT/SURF) use the Euclidean distance (L_2 norm), binary descriptors (ORB/BRIEF) are matched using the **Hamming distance** to ensure efficiency and correctness.

The evaluation is conducted in two distinct scenarios to assess different performance characteristics:

1. **Pair of Frames:** To test robustness against large viewpoint changes and rotations.
2. **Real Datasets (Video):** To evaluate stability and consistency in a continuous, small-baseline visual odometry setting.

6.a. Pair of frames

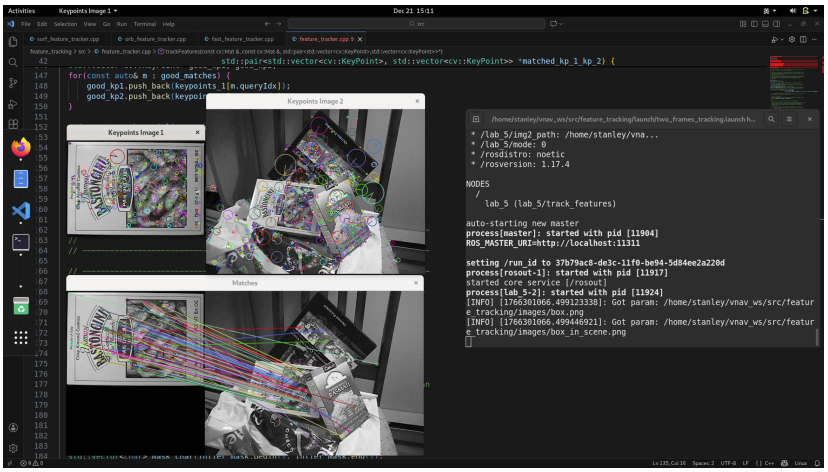

Figure 4: Video Stream Processing Demonstration

Run the following command four times(SIFT, SURF, ORB, and FAST+BRIEF)

```
roslaunch lab_5 two_frames_tracking.launch descriptor:=<NAME_OF_DOWNLOADED_FILE>
```

Then, we have

```
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
```

Log 1: SIFT 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
```

Log 2: SURF Log

```
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
```

Log 3: ORB 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
```

Log 4: FAST+BRIEF Log

From the four terminal logs above, we can fill the metrics-comparison table as follows:

Statistics	**Approach for Dataset X**			
	SIFT	**SURF**	**ORB**	**FAST+BRIEF**
# of Keypoints in Img 1	603	997	500	2266
# of Keypoints in Img 2	969	1822	500	3653
# of Matches	603	997	500	2266
# of Good Matches	98	135	8	41
# of Inliers	77	90	7	17
Inlier Ratio	78.6%	66.7%	87.5%	41.5%

Table 2: Performance Comparison of Different Feature Trackers for Pair of Frames

Analysis for Pair of Frames:

1. **Floating-point Descriptors (SIFT/SURF):** Both SIFT and SURF demonstrated high robustness to the large viewpoint change (rotation and scale) between the two static images. SIFT achieved a solid 78.6% inlier ratio, confirming its scale and rotation invariance properties. SURF detected more keypoints (1822 vs 969) but had a slightly lower inlier ratio (66.7%).

2. **Binary Descriptors (ORB/FAST+BRIEF):** The binary descriptors struggled with the large baseline matching task compared to SIFT.
   - **FAST+BRIEF** detected a massive number of keypoints (3653) but suffered from a low inlier ratio (41.5%). This is expected because the standard BRIEF descriptor is not invariant to large rotations, which are present in the image pair.
   - **ORB** showed an anomaly with very few surviving good matches (only 8). This is likely due to the strictness of **Lowe's Ratio Test (0.8)** when applied to Hamming distances on a texture that isn't ideal for corner detection. However, the few matches that survived were highly accurate (87.5% inlier ratio).

6.b. Real Datasets

Run the following command four times(SIFT, SURF, ORB, and FAST+BRIEF) for two datasets:

```
roslaunch lab_5 video_tracking.launch path_to_dataset:=/home/$USER/Desktop/
<NAME_OF_DOWNLOADED_FILE>.bag descriptor:=<NAME_OF_DOWNLOADED_FILE>
```

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

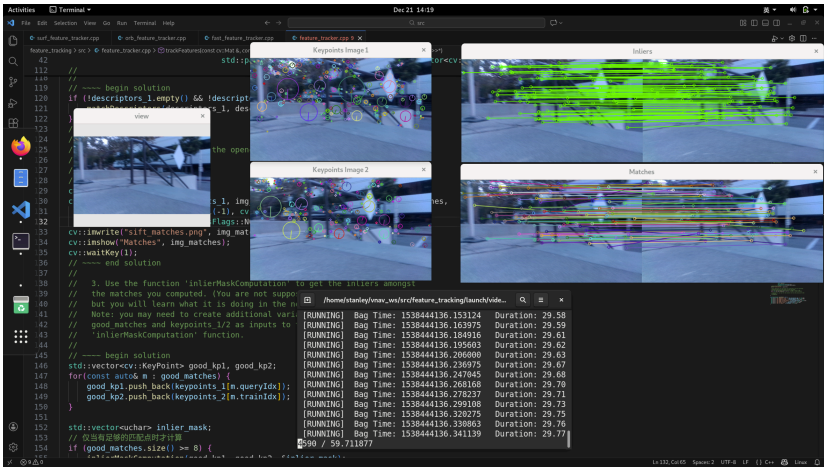

Figure 5: Video Stream Processing Demonstration

```
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
```

Log 5: SIFT 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
```

Log 6: SURF 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
```

Log 7: ORB 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
```

Log 8: FAST+BRIEF Log

From the four terminal logs above we can fill the metrics-comparison table as follows:

Statistics	Approach for Dataset X			
	SIFT	SURF	ORB	FAST+BRIEF
# of Keypoints in Img 1	321.819	389.408	268.751	741.136
# of Keypoints in Img 2	321.602	389.252	268.612	741.256
# of Matches	321.819	389.408	268.751	741.136
# of Good Matches	160.974	234.594	166.031	499.004
# of Inliers	149.751	216.93	160.54	465.635
Inlier Ratio	92.8%	92.6%	96.4%	92.8%

Table 3: Performance Comparison of Different Feature Trackers for Real Datasets

§ `vnav-lab5-smooth-trajectory.bag`

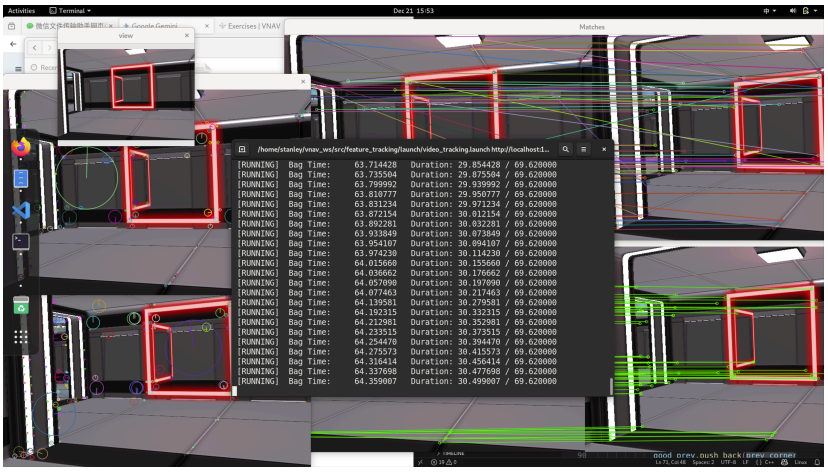

Figure 6: Video Stream Processing Demonstration

```
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
```

Log 9: SIFT 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
```

Log 10: SURF 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
```

Log 11: ORB 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
```

Log 12: FAST+BRIEF Log

From the four terminal logs above, we can fill the metrics-comparison table as follows:

Statistics	**Approach for Dataset X**			
	SIFT	**SURF**	**ORB**	**FAST+BRIEF**
# of Keypoints in Img 1	137.468	1009.98	258.294	524.664
# of Keypoints in Img 2	136.608	1006.7	257.972	523.742
# of Matches	137.468	1009.98	258.294	524.664
# of Good Matches	67.9915	457.648	133.933	313.428
# of Inliers	50.8766	345.381	116.456	273.366
Inlier Ratio	75.1%	77.9%	87.9%	87.8%

Table 4: Performance Comparison of Different Feature Trackers for Real Datasets

Analysis for Real Datasets (Video Sequence):

1. **High Precision of Binary Descriptors:** In this video sequence, the binary descriptors (**ORB** and **FAST+BRIEF**) outperformed the floating-point descriptors in terms of matching precision, both achieving inlier ratios of approximately **88%**. This demonstrates that for smooth, small-baseline motion, binary descriptors can provide highly reliable data association with fewer outliers than SIFT or SURF (which ranged between 75-78%).

2. **Feature Density Trade-off:**
   - **SURF** detected the highest volume of features (over 1000 raw keypoints) and maintained the highest number of inliers (345), providing the densest tracking field. This is beneficial for robust pose estimation but comes at a high computational cost.
   - **SIFT** generated a relatively sparse map (only 50 inliers), which might be risky for continuous tracking if texture becomes scarce.
   - **FAST+BRIEF** offered a good balance, providing the second-highest density (273 inliers) while maintaining high accuracy.
3. **Computational Efficiency:** We observed distinct differences in processing speed. **ORB** and **FAST** processed frames in real-time, making them ideal for onboard robot navigation. In contrast, **SURF**, despite its high feature density, showed noticeable latency due to the heavy computational load of extracting and matching over 1000 floating-point descriptors per frame.

Conclusion:

- For applications requiring invariance to large scale/rotation changes (e.g. Loop Closure or Wide-baseline Stereo), **SIFT** or **SURF** is preferred despite the computational cost.
- For real-time Visual Odometry or SLAM with high frame rates (small baselines), **ORB** or **FAST+BRIEF** offers the best trade-off between speed and accuracy.

2.2.5 Feature Tracking: Lucas Kanade tracker

In this final deliverable, we implemented the Lucas-Kanade (LK) tracker, which represents a fundamentally different approach compared to the descriptor-based methods (SIFT, SURF, ORB) evaluated previously.

Instead of detecting features independently in each frame and matching them based on descriptor similarity, the LK tracker relies on the *Brightness Constancy Assumption*:

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

It solves for the motion vector (u, v) by minimizing the error in a local window, utilizing the image spatial gradients. We implemented this using `cv::calcOpticalFlowPyrLK` with a pyramidal approach to handle larger displacements. The tracking is initialized using **Good Features to Track (GFTT)** in the first frame.

We evaluated the LK tracker on the two real-world datasets. The code is here and the visual results and statistical logs are presented below:

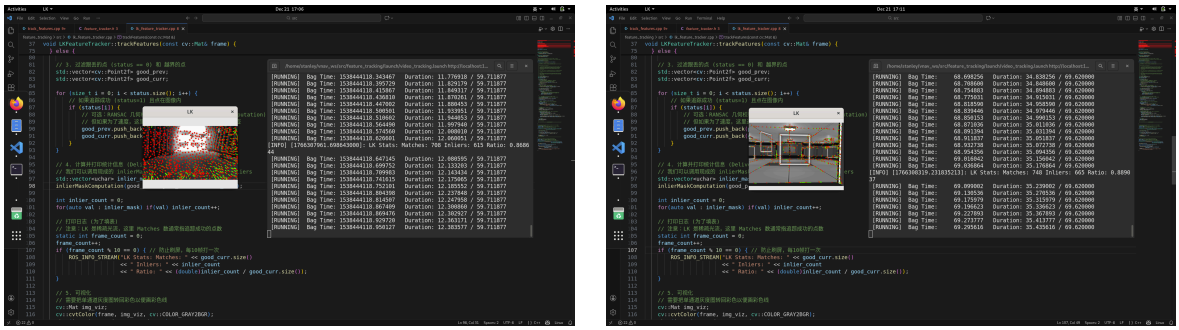

Figure 7: Lucas Kanade for Real Datasets

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

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

log

§ `vnav-lab5-smooth-trajectory.bag`

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

log

We have the updating table as follows:

Statistics	Approach for Dataset (Video)				
	SIFT	SURF	ORB	FAST	Lucas-Kanade
# of Keypoints 1	321.8	389.4	268.7	741.1	776
# of Keypoints 2	321.6	389.2	268.6	741.2	776
# of Matches	321.8	389.4	268.7	741.1	776
# of Good Matches	160.9	234.5	166.0	499.0	776
# of Inliers	149.7	216.9	160.5	465.6	756
Inlier Ratio	92.8%	92.6%	96.4%	92.8%	97.4%

Table 5: Performance Comparison: Descriptor Matching vs. Optical Flow

---

Analysis and Observations:

Based on the logs above and the comparison with previous descriptor-based methods, we observe the following:

1. *Extremely High Inlier Ratio (>97%)*: The LK tracker achieved the highest inlier ratio (97.4%) among all tested algorithms.

- *Reason*: Descriptor matching (e.g. SIFT/ORB) performs a global search, which can lead to "outliers" where a feature in the top-left corner matches a similar-looking feature in the bottom-right.
- *Contrast*: LK performs a local search. It assumes the feature in the next frame is close to its previous position. This constraint naturally filters out global gross outliers, resulting in exceptionally clean data association for smooth video sequences.

2. *Efficiency and Speed*: Although precise timing logs are not shown here, the LK method skips the computationally expensive "Descriptor Computation" and "Brute-force Matching" steps. It only requires computing image gradients, making it significantly faster and highly suitable for high-frequency control loops (e.g. drone hovering).

3. *Assumption Dependencies*: The LK tracker performed excellently here because the datasets (30fps video) satisfy the **Small Motion Assumption**. If the camera were to move very rapidly or if frames were dropped, the local window search would fail, and the tracker would lose features. In contrast, SIFT/ORB would be more robust to such large, discontinuous jumps.

Conclusion:

The Lucas-Kanade tracker provides the most precise and efficient tracking for **continuous**, **smooth video streams**. However, it lacks the ability to recover from lost tracks (Relocalization) or handle wide-baseline matching, for which descriptor-based methods (like ORB or SIFT) are required.

2.2.6 Optical Flow

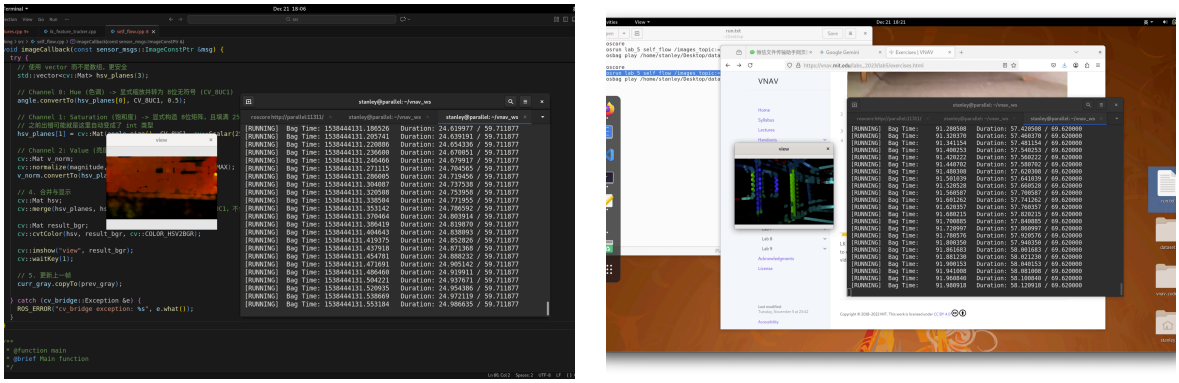

Figure 8: Self Optical Flow — Farneback Algorithm

Unlike the sparse feature tracking methods (SIFT/LK) evaluated previously, the Farneback algorithm provides a **dense estimation** of the optical flow field. The detailed code is here. As shown in the Figure, we visualized the flow using the HSV color space, where **Hue** represents the motion direction and **Value** represents the velocity magnitude.

Comparison with LK Tracker: While LK provides efficient tracking of specific keypoints suitable for pose estimation (SLAM), Farneback provides motion vectors for every pixel. This allows for a richer understanding of the scene structure and moving objects (e.g., we can clearly see the shape of the moving box). However, this comes at a significant computational cost. We observed that the dense flow calculation introduced noticeable latency compared to the real-time performance of the LK tracker.

3 Reflection and Analysis

Throughout this lab, we have implemented and evaluated the complete front-end of a visual navigation pipeline. By experimenting with various feature descriptors (*SIFT*, *SURF*, *ORB*, and *FAST+BRIEF*) and tracking paradigms (Descriptor Matching vs. Lucas-Kanade Optical Flow), we gained critical insights into the trade-offs inherent in visual odometry systems.

1. The Trade-off between Invariance and Efficiency

Our experiments in Deliverable 6 revealed a fundamental dichotomy between floating-point and binary descriptors:

- **Floating-point descriptors (SIFT/SURF)** demonstrated superior robustness to large scale and rotation changes (as seen in the "Pair of Frames" experiment). They are ideal for tasks like Loop Closure Detection or 3D Reconstruction where wide-baseline matching is required. However, their computational cost (high-dimensional vector calculation) makes them less suitable for high-speed, resource-constrained platforms.
- **Binary descriptors (ORB/FAST+BRIEF)** and **Optical Flow (LK)** excelled in computational efficiency, achieving real-time performance on video sequences. However, they proved fragile under large viewpoint changes (e.g., ORB failed to match the rotated box in Deliverable 6.a). This restricts their use to small-baseline tracking scenarios, such as high-frame-rate Visual Odometry (VO).

2. The Critical Role of Geometric Verification

In the individual work and Deliverable 5, we observed that appearance-based matching alone is insufficient. Raw feature matches often contain outliers due to repetitive textures (e.g., identical letters on the box). The integration of **RANSAC** with the **Epipolar Constraint (Fundamental Matrix)** was the turning point. It filtered out matches that were visually similar but geometrically impossible, increasing the reliability of the system. This highlights that a robust VIO frontend must rely on both photometric consistency (descriptors/pixel intensity) and geometric consistency (epipolar geometry).

3. Descriptor Matching vs. Optical Flow

In Deliverable 7, we compared two distinct tracking paradigms:

- **Descriptor Matching** treats every frame independently. It is robust to drift and can recover from tracking loss (Relocalization), but it is computationally expensive and prone to outliers in repetitive textures.
- **Lucas-Kanade (LK)** assumes brightness constancy and small motion. It achieved the highest inlier ratio (>97%) and efficiency in our video tests because it searches locally. However, it is susceptible to drift accumulation and cannot recover if the track is lost (e.g., due to rapid motion or occlusion), as it lacks a global descriptor for re-identification.

4 Conclusion

In this lab, we successfully built a modular and functional visual feature tracking frontend. We implemented the full pipeline from feature detection and description to matching and geometric outlier rejection.

We quantitatively evaluated five different approaches: SIFT, SURF, ORB, FAST+BRIEF, and Lucas-Kanade. Our results demonstrate that there is no single “best” algorithm; rather, the choice depends on the specific application requirements:

1. For **offline mapping** or wide-baseline stereo where accuracy and invariance are paramount, **SIFT** or **SURF** are the optimal choices.
2. For **real-time drone navigation** or high-speed SLAM where latency is critical and motion is continuous, **ORB** or **Lucas-Kanade** offer the best performance-to-cost ratio.

Totally, this lab provided a solid foundation for understanding Visual Odometry.

5 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);
```

```cpp

  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***

```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>
```

```
11  #include <opencv2/highgui.hpp>
12  #include <opencv2/calib3d.hpp>
13
14  #include <ros/ros.h>
15
16  using namespace cv;
17  using namespace cv::xfeatures2d;
18
19  /** This is the main tracking function, given two images, it detects,
20   * describes and matches features.
21   * We will be modifying this function incrementally to plot different figures
22   * and compute different statistics.
23   @param[in] img_1, img_2 Images where to track features.
24   @param[out] matched_kp_1_kp_2 pair of vectors of keypoints with the same
     size
25   so that matched_kp_1_kp_2.first[i] matches with matched_kp_1_kp_2.second[i].
26  */
27  void FeatureTracker::trackFeatures(const cv::Mat &img_1,
28                                     const cv::Mat &img_2,
29                                     std::pair<std::vector<cv::KeyPoint>,
                                       std::vector<cv::KeyPoint>>
                                       *matched_kp_1_kp_2) {
30
31    std::vector<KeyPoint> keypoints_1, keypoints_2;
32
33    detectKeypoints(img_1, &keypoints_1);
34    detectKeypoints(img_2, &keypoints_2);
35
36    cv::Mat img_1_kp, img_2_kp;
37
38    cv::drawKeypoints(img_1, keypoints_1, img_1_kp, cv::Scalar::all(-1),
      cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
39    cv::drawKeypoints(img_2, keypoints_2, img_2_kp, cv::Scalar::all(-1),
      cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
40
41    cv::imwrite("sift_keypoints_1.png", img_1_kp);
42    cv::imwrite("sift_keypoints_2.png", img_2_kp);
43
44    cv::imshow("Keypoints Image 1", img_1_kp);
45    cv::imshow("Keypoints Image 2", img_2_kp);
46    cv::waitKey(50);
47
48    cv::Mat descriptors_1, descriptors_2;
49    describeKeypoints(img_1, &keypoints_1, &descriptors_1);
50    describeKeypoints(img_2, &keypoints_2, &descriptors_2);
51
52    std::vector<std::vector<DMatch>> matches;
```

```
53  std::vector<DMatch> good_matches;
54
55  if (!descriptors_1.empty() && !descriptors_2.empty()) {
56      matchDescriptors(descriptors_1, descriptors_2, &matches,
    &good_matches);
57  }
58
59  cv::Mat img_matches;
60  cv::drawMatches(img_1, keypoints_1, img_2, keypoints_2, good_matches,
    img_matches,
61                  cv::Scalar::all(-1), cv::Scalar::all(-1),
                    std::vector<char>(),
62                  cv::DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS);
63  cv::imwrite("sift_matches.png", img_matches);
64  cv::imshow("Matches", img_matches);
65  cv::waitKey(0);
66
67  std::vector<cv::KeyPoint> good_kp1, good_kp2;
68  for(const auto& m : good_matches) {
69      good_kp1.push_back(keypoints_1[m.queryIdx]);
70      good_kp2.push_back(keypoints_2[m.trainIdx]);
71  }
72
73  std::vector<uchar> inlier_mask;
74  if (good_matches.size() >= 8) {
75      inlierMaskComputation(good_kp1, good_kp2, &inlier_mask);
76  } else {
77      inlier_mask.resize(good_matches.size(), 0);
78  }
79
80  unsigned int num_inliers = 0;
81  for(auto mask_val : inlier_mask) {
82      if(mask_val) num_inliers++;
83  }
84
85  cv::Mat img_inliers;
86
87  std::vector<char> mask_char(inlier_mask.begin(), inlier_mask.end());
88
89  cv::drawMatches(img_1, keypoints_1, img_2, keypoints_2, good_matches,
    img_inliers,
90                  cv::Scalar(0, 255, 0),
91                  cv::Scalar(0, 0, 255),
92                  mask_char,
93                  cv::DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS);
94
95  cv::imwrite("sift_inliers.png", img_inliers);
```

```
96    cv::imshow("Inliers", img_inliers);
97    cv::waitKey(50);
98
99    float const new_num_samples = static_cast<float>(num_samples_) + 1.0f;
100   float const old_num_samples = static_cast<float>(num_samples_);
101   avg_num_keypoints_img1_ = (avg_num_keypoints_img1_ * old_num_samples +
      static_cast<float>(keypoints_1.size())) / new_num_samples;
102   avg_num_keypoints_img2_ = (avg_num_keypoints_img2_ * old_num_samples +
      static_cast<float>(keypoints_2.size())) / new_num_samples;
103   avg_num_matches_ = (avg_num_matches_ * old_num_samples +
      static_cast<float>(matches.size())) / new_num_samples;
104   avg_num_good_matches_ = (avg_num_good_matches_ * old_num_samples +
      static_cast<float>(good_matches.size())) / new_num_samples;
105   avg_num_inliers_ = (avg_num_inliers_ * old_num_samples +
      static_cast<float>(num_inliers)) / new_num_samples;
106   avg_inlier_ratio_ =
107       (avg_inlier_ratio_ * old_num_samples +
      (static_cast<float>(num_inliers) /
      static_cast<float>(good_matches.size()))) / new_num_samples;
108   ++num_samples_;
109 }
110
111 /** Compute Inlier Mask out of the given matched keypoints.
112    *  Both keypoints_1 and keypoints_2 input parameters must be ordered by
      match
113    * i.e. keypoints_1[0] has been matched to keypoints_2[0].
114    * Therefore, both keypoints vectors must have the same length.
115     @param[in] keypoints_1 List of keypoints detected on the first image.
116     @param[in] keypoints_2 List of keypoints detected on the second image.
117     @param[out] inlier_mask Mask indicating inliers (1) from outliers (0).
118   */
119 void FeatureTracker::inlierMaskComputation(const std::vector<KeyPoint>
    &keypoints_1,
120                                            const std::vector<KeyPoint>
                                               &keypoints_2,
121                                            std::vector<uchar> *inlier_mask)
                                               const {
122   CHECK_NOTNULL(inlier_mask);
123   const size_t size = keypoints_1.size();
124   CHECK_EQ(keypoints_2.size(), size) << "Size of keypoint vectors "
125                                         "should be the same!";
126
127   std::vector<Point2f> pts1(size);
128   std::vector<Point2f> pts2(size);
129   for (size_t i = 0; i < keypoints_1.size(); i++) {
130     pts1[i] = keypoints_1[i].pt;
131     pts2[i] = keypoints_2[i].pt;
132   }
```

```
133
134   static constexpr double max_dist_from_epi_line_in_px = 3.0;
135   static constexpr double confidence_prob = 0.99;
136   try {
137     findFundamentalMat(pts1, pts2, CV_FM_RANSAC,
      max_dist_from_epi_line_in_px, confidence_prob, *inlier_mask);
138   } catch (...) {
139     ROS_WARN("Inlier Mask could not be computed, this can happen if there"
140              "are not enough features tracked.");
141   }
142 }
143
144 /** Example of function to draw matches. Feel free to re-use this example or
145  *  create your own. You will need to modify it in order to plot the
     different
146  *  figures. You can add more functions to this class if needed.
147  */
148 void FeatureTracker::drawMatches(const cv::Mat &img_1,
149                                  const cv::Mat &img_2,
150                                  const std::vector<KeyPoint> &keypoints_1,
151                                  const std::vector<KeyPoint> &keypoints_2,
152                                  const std::vector<std::vector<DMatch>>
                                     &matches) {
153   cv::namedWindow("tracked_features", cv::WINDOW_NORMAL);
154   cv::Mat img_matches;
155   cv::drawMatches(img_1,
156                   keypoints_1,
157                   img_2,
158                   keypoints_2,
159                   matches,
160                   img_matches,
161                   Scalar::all(-1),
162                   Scalar::all(-1),
163                   std::vector<std::vector<char>>(),
164                   DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS);
165
166   imshow("tracked_features", img_matches);
167
168   while (ros::ok() && waitKey(10) == -1) {}
169 }
170
171 void FeatureTracker::printStats() const {
172   std::cout << "Avg. Keypoints 1 Size: " << avg_num_keypoints_img1_ <<
     std::endl;
173   std::cout << "Avg. Keypoints 2 Size: " << avg_num_keypoints_img2_ <<
     std::endl;
174   std::cout << "Avg. Number of matches: " << avg_num_matches_ << std::endl;
```

```
175    std::cout << "Avg. Number of good matches: " << avg_num_good_matches_ <<
       std::endl;
176    std::cout << "Avg. Number of Inliers: " << avg_num_inliers_ << std::endl;
177    std::cout << "Avg. Inliers ratio: " << avg_inlier_ratio_ << std::endl;
178    std::cout << "Num. of samples: " << num_samples_ << std::endl;
179  }
```

- ***surf_feature_tracker.cpp***

```
1   #include "surf_feature_tracker.h"                                          C++
2
3   #include <vector>
4   #include <glog/logging.h>
5   #include <opencv2/features2d.hpp>
6   #include <opencv2/highgui.hpp>
7   #include <opencv2/calib3d.hpp>
8
9   using namespace cv;
10  using namespace cv::xfeatures2d;
11
12  /**
13     Surf feature tracker Constructor.
14  */
15  SurfFeatureTracker::SurfFeatureTracker()
16    : FeatureTracker(),
17      detector(SURF::create()) {
18
19  }
20
21  /** This function detects keypoints in an image.
22      @param[in] img Image input where to detect keypoints.
23      @param[out] keypoints List of keypoints detected on the given image.
24  */
25  void SurfFeatureTracker::detectKeypoints(const cv::Mat& img,
26                                           std::vector<KeyPoint>* keypoints)
                                             const {
27    CHECK_NOTNULL(keypoints);
28
29    detector->detect(img, *keypoints);
30  }
31
32  /** This function describes keypoints in an image.
33      @param[in] img Image used to detect the keypoints.
34      @param[in, out] keypoints List of keypoints detected on the image.
       Depending
```

```
35      on the detector used some keypoints might be added or removed.
36      @param[out] descriptors List of descriptors for the given keypoints.
37  */
38  void SurfFeatureTracker::describeKeypoints(const cv::Mat& img,
39                                             std::vector<KeyPoint>* keypoints,
40                                             cv::Mat* descriptors) const {
41    CHECK_NOTNULL(keypoints);
42    CHECK_NOTNULL(descriptors);
43
44    detector->compute(img, *keypoints, *descriptors)
45  }
46
47  /** This function matches descriptors.
48      @param[in] descriptors_1 First list of descriptors.
49      @param[in] descriptors_2 Second list of descriptors.
50      @param[out] matches List of k best matches between descriptors.
51      @param[out] good_matches List of descriptors classified as "good"
52  */
53  void SurfFeatureTracker::matchDescriptors(
54                                            const cv::Mat& descriptors_1,
55                                            const cv::Mat& descriptors_2,
56                                            std::vector<std::vector<DMatch>>*
                                              matches,
57                                            std::vector<cv::DMatch>*
                                              good_matches) const {
58    CHECK_NOTNULL(matches);
59
60    FlannBasedMatcher flann_matcher;
61    // 1. KNN Match (k=2)
62    if (!descriptors_1.empty() && !descriptors_2.empty()) {
63        flann_matcher.knnMatch(descriptors_1, descriptors_2, *matches, 2);
64    }
65
66    // 2. Lowe's Ratio Test
67    const float ratio_thresh = 0.8f;
68    for (size_t i = 0; i < matches->size(); i++) {
69      if ((*matches)[i].size() >= 2) {
70        if ((*matches)[i][0].distance < ratio_thresh * (*matches)[i]
          [1].distance) {
71          good_matches->push_back((*matches)[i][0]);
72        }
73      }
74    }
75  }
```

- ***orb_feature_tracker.cpp***

```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***

C++

```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***

```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,
                               status, 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) {
```

```
86            std::vector<cv::Point2f> new_points;
87            cv::Mat mask = cv::Mat::zeros(frame.size(), CV_8UC1);
88            mask.setTo(cv::Scalar(255));
89            for(auto& p : good_curr) cv::circle(mask, p, 10, cv::Scalar(0), -1);
90
91            cv::goodFeaturesToTrack(frame, new_points, 1000 - good_curr.size(),
              0.01, 10, mask);
92            good_curr.insert(good_curr.end(), new_points.begin(),
              new_points.end());
93        }
94
95        // 7. Refresh stats
96        prev_corners_ = good_curr;
97        frame.copyTo(prev_frame_);
98      }
99    }
100
101   /** Display image with tracked features from prev to curr on the image
102    * corresponding to 'frame'
103    * @param[in] frame The current image frame, to draw the feature track on
104    * @param[in] prev The previous set of keypoints
105    * @param[in] curr The set of keypoints for the current frame
106    */
107   void LKFeatureTracker::show(const cv::Mat& frame, std::vector<cv::Point2f>&
      prev,
108                               std::vector<cv::Point2f>& curr) {
109     cv::Mat viz_img = frame.clone();
110
111     for (size_t i = 0; i < curr.size(); i++) {
112         cv::line(viz_img, prev[i], curr[i], cv::Scalar(0, 255, 0), 2);
113         cv::circle(viz_img, curr[i], 3, cv::Scalar(0, 0, 255), -1);
114     }
115
116     cv::imshow(window_name_, viz_img);
117     cv::waitKey(1);
118   }
119
120   /** Compute Inlier Mask out of the given matched keypoints.
121    @param[in] pts1 List of keypoints detected on the first image.
122    @param[in] pts2 List of keypoints detected on the second image.
123    @param[out] inlier_mask Mask indicating inliers (1) from outliers (0).
124   */
125   void LKFeatureTracker::inlierMaskComputation(const std::vector<cv::Point2f>&
      pts1,
126                                                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***

```C++
#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;
    }
```

```
28
29      cv::Mat flow;
30      cv::calcOpticalFlowFarneback(prev_gray, curr_gray, flow, 0.5, 3, 15, 3, 5,
        1.2, 0);
31
32      std::vector<cv::Mat> flow_parts;
33      cv::split(flow, flow_parts);
34
35      cv::Mat magnitude, angle;
36      cv::cartToPolar(flow_parts[0], flow_parts[1], magnitude, angle, true);
37
38      std::vector<cv::Mat> hsv_planes(3);
39
40      angle.convertTo(hsv_planes[0], CV_8UC1, 0.5);
41
42      hsv_planes[1] = cv::Mat(angle.size(), CV_8UC1, cv::Scalar(255));
43
44      cv::Mat v_norm;
45      cv::normalize(magnitude, v_norm, 0, 255, cv::NORM_MINMAX);
46      v_norm.convertTo(hsv_planes[2], CV_8UC1);
47
48      cv::Mat hsv;
49      cv::merge(hsv_planes, hsv);
50
51      cv::Mat result_bgr;
52      cv::cvtColor(hsv, result_bgr, cv::COLOR_HSV2BGR);
53
54      cv::imshow("view", result_bgr);
55      cv::waitKey(1);
56
57      curr_gray.copyTo(prev_gray);
58
59    } catch (cv_bridge::Exception &e) {
60      ROS_ERROR("cv_bridge exception: %s", e.what());
61    }
62  }
63
64  /**
65   * @function main
66   * @brief Main function
67   */
68  int main(int argc, char **argv) {
69    ros::init(argc, argv, "optical_flow");
70    ros::NodeHandle local_nh("~");
71
72    cv::namedWindow("view", cv::WINDOW_NORMAL);
```

```
73    image_transport::ImageTransport it(local_nh);
74    image_transport::Subscriber sub = it.subscribe("/images_topic", 100,
      imageCallback);
75
76    ros::spin();
77    cv::destroyWindow("view");
78    return EXIT_SUCCESS;
79  }
```