Department of Information Engineering and Computer Science

SIV Project

A Monocular Visual Odometry system

STUDENTS

Lorenzo Orsingher

Ilaria Rocchi

Academic year 2023/2024

Contents

Summary

The goal of the project is to build a Monocular Visual Odometry system from scratch, without the assistance of machine learning or neural networks, relying only on traditional computer vision techniques. The main objective is to construct a robust system that can be used on different kinds of cameras by providing an easy-to-use pipeline that goes from calibration to exploration.

The system is divided into three main parts:

- camera calibration
- extraction of features
- motion estimation

1 Camera Calibration

To extract 3D information from a 2D image, it is necessary to know the intrinsic parameters of the camera. These parameters are not only used in the extrapolations of the 3D points but also in the extraction of the features, in the estimation of the motion, and in the rectification of the frames. The calibration of the camera is a crucial step in the pipeline of the Monocular Visual Odometry system.

A widely used and well known model for the camera is the pinhole camera model.

1.1 The Pinhole Camera Model

The pinhole camera model is a simple model that describes the way light enters the camera and forms an image. It is based on the principle that light travels in straight lines and that the camera is a dark box with a small hole on one side. The light enters the camera through the hole and forms an inverted image on the opposite side of the box.

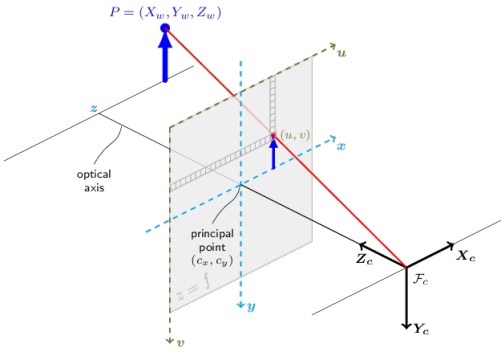


Figure 1.1: The pinhole camera model

The model describes the mathematical relationship between the coordinates of a point in three-dimensional space and its projection onto the image plane of an ideal pinhole camera, where the camera aperture is described as a point and no lenses are used to focus light. The distortion-free projective transformation given by a pinhole camera model is shown below.

$$s\, p = A \left[R|t\right] P_w,$$

Where P_w is a 3D point expressed with respect to the world coordinate system, p is a 2D pixel in the image plane, A is the camera intrinsic matrix, R and t are the rotation and translation that describe the change of coordinates from world to camera coordinate systems (or camera frame) and s is the projective transformation's arbitrary scaling and not part of the camera model.

The camera intrinsic matrix A (Figure 1.2) projects 3D points given in the camera coordinate system to 2D pixel coordinates, it's composed of the focal lengths fx and fy, which are expressed in pixel units, and the principal point (cx,cy), that is usually close to the image center

Figure 1.2: Intrinsic camera matrix

$$A = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

The intrinsic camera matrix is not the only set of parameters estimated during the calibration process. Real lenses introduce, to some degree, barrel distortion and tangential distortion; to compensate for this, a vector of distortion coefficients is estimated along the intrinsic camera matrix and subsequently when the rectification of the image is needed.

1.2 Calibration Process

The estimation of the camera parameters consists of finding the values that minimize the error between the observed and the predicted image points. The calibration process is usually performed using a set of images of a known pattern, such as a chessboard, that is placed in different positions and orientations with respect to the camera. The images are then used to estimate the camera parameters.

In our case, we decided to pick a ChArUco board, a chessboard with ArUco markers on the corners. Compared to a traditional chessboard, the ChArUco board has the advantage of being more robust to occlusions and to the camera's position and orientation.

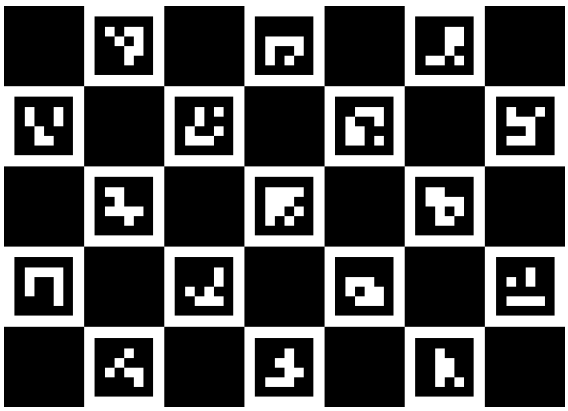


Figure 1.3: The ChArUco board used for calibration

The calibration process is divided into two main steps: the detection of the ChArUco board in the images and the estimation of the camera parameters.

First things first, the *camera_calibration_charuco.py* is fed a video clip of the ChArUco board, where the camera slowly pans and tilts capturing the board from every angle in all the areas of the camera frame.

At that point, corner detection is run on each and every frame and the coordinates (when enough corners are detected and the frame is deemed valid) are stored in a list. From that list, an arbitrary number of frames between 50 and 100 are selected and used to estimate the camera parameters using OpenCV's *calibrateCameraCharucoExtended*.

Once finished, the intrinsic and extrinsic parameters plus the distortion coefficients are saved in a JSON file and stored for later.

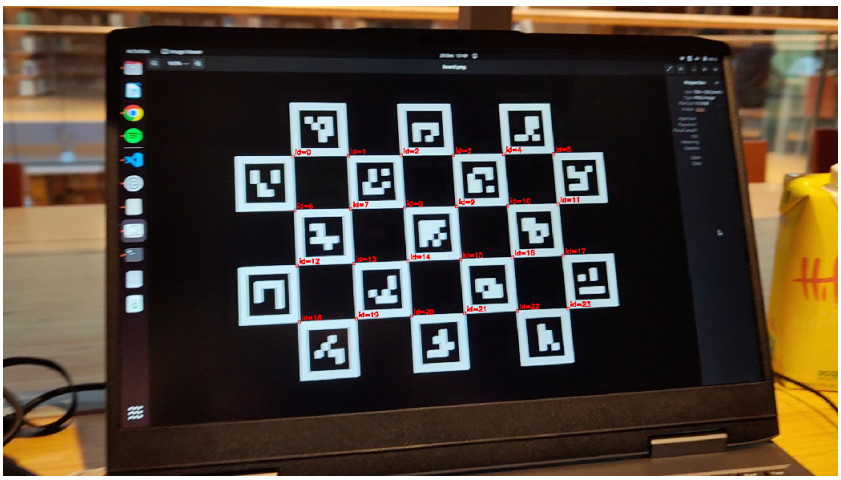


Figure 1.4: Running the calibration process from a cellphone clip, note in red the detected corners.

While it is true that this process is a one-time-only step, it is important to remember that changes in focus or zoom completely invalidate the calibration. This happens in particular when using smartphone cameras, but it is valid for any devices that provide autofocus; it is therefore crucial to disable or lock it between the calibration and the actual use of the camera.

2 Feature Extraction and Matching

Feature Extraction is the process of obtaining the most relevant information from the images. In our context, we are interested in the extraction of the keypoints and the descriptors. The keypoints are the points of interest in the image, while the descriptors are the vectors that describe the keypoints. Matching is the process of finding the correspondences between the keypoints of two images or, in our case, two subsequent frames in order to then triangulate and track the movement of the camera.

To fine-tune the process we decided to try many different combinations of feature detectors, matchers, and preprocessing techniques, and to compare the performance of each combination.

2.1 Preprocessing

In order to improve the robustness and accuracy of the created visual odometry system, preprocessing of the provided raw image data has been performed. This section describes the image processing steps we decided to apply.

2.1.1 Gaussian blur

We decided to use the Gaussian blur as a denoising technique. The adoption of the Gaussian low-pass filter results in cleaner smoother images, which might lead to feature detection and tracking being more reliable.

2.1.2 Undistortion

Due to cameras' lens imperfections, lens distortions may arise, causing deviations from the ideal pinhole camera model. During the camera calibration process, distortion coefficients are obtained and later used to correct such distortions. Therefore, in order to ensure the accuracy of motion estimation, if the provided images are not rectified yet, undistortion is applied.

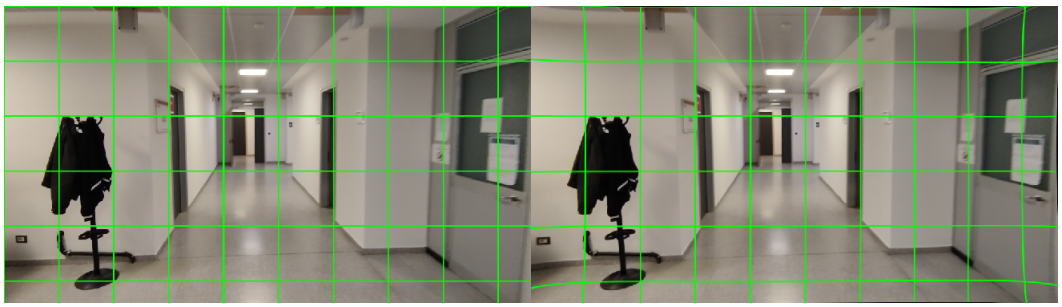


Figure 2.1: A frame before and after undistortion, the distortion is clearly visible only in the corners of the image due to the fact that the smartphone camera software heavily crops the frame when shooting a video.

2.2 Methods

When it comes to feature extraction and matching, various methods can be adopted. In this project, we focused on the SIFT (Scale-Invariant Feature Transform) algorithm and the ORB (Oriented FAST and Rotated BRIEF) algorithm for feature detection. Each detection algorithm has been paired up with various feature matching techniques, which have the purpose of finding correspondences between the detected keypoints in consecutive frames.

2.2.1 SIFT

SIFT [2] is a feature detection algorithm that is invariant to scale and rotation and robust to changes in illumination. It is based on the detection of keypoints and the extraction of descriptors. The keypoints are detected at different scales and locations in the image, and the descriptors (used to match the keypoints in different images) are then computed for each keypoint.

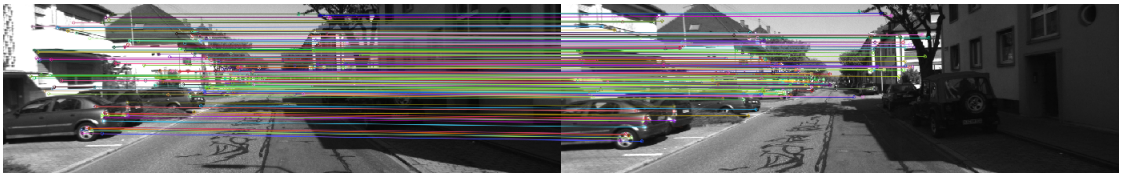


Figure 2.2: Example of SIFT feature matching between consecutive frames.

2.2.2 ORB

ORB [4] is a feature detection algorithm that is based on the FAST [6] (Features from Accelerated Segment Test) keypoint detector and the BRIEF [1] (Binary Robust Independent Elementary Features) descriptor. Unlike SIFT, BRIEF is a binary descriptor, which means that the found keypoints are converted into binary feature vectors, which only contain 1 and 0.

2.2.3 Feature matching

The feature matching process is the process of finding the correspondences between the keypoints of two images. In our case, we are interested in finding the correspondences between the keypoints of two subsequent frames. The feature matching process is crucial for the estimation of the camera motion and the triangulation of the 3D points. We decided to try different feature matching techniques and compare their performances.

We focused on two main feature matching techniques: the Brute-Force matcher and the FLANN [5] (Fast Library for Approximate Nearest Neighbors) matcher. The Brute-Force matcher is a simple matcher that compares each feature of the first image with each feature of the second image. The FLANN matcher is a more complex matcher that uses a variety of optimized algorithms; for our experiments we paired SIFT with the kd-tree search to find the nearest neighbors of each feature, meanwhile for ORB (due to the different nature of the descriptor) we picked multi-probe LSH.

Moreover, we decided to compare the performances of the feature matching techniques with and without the use of Lowe's ratio test, a technique that is used to filter out false matches. The ratio

test is based on the comparison of the distance of the closest match with the distance of the second closest match: if the ratio is below a certain threshold, the match is considered valid.

3 Motion Estimation

While for feature extraction, matching, and tracking we decided to opt for a combination of different methods and techniques, for the motion estimation we decided to focus on a single algorithm, based on the concept of epipolar geometry.

3.1 Epipolar Geometry

Epipolar geometry [7] is commonly used in the context of stereo vision, by seeing the same scene from different points there are a number of geometric relations between the 2D points and the 3D world. These constraints can be used, paired with some approximation of the camera's intrinsic parameters, to estimate the relative position of said points and the camera as shown in Figure 3.1.

In the same way, by using the epipolar geometry on two frames shot by the same camera in different positions, it is possible to estimate the relative motion of the camera in the 3D space.

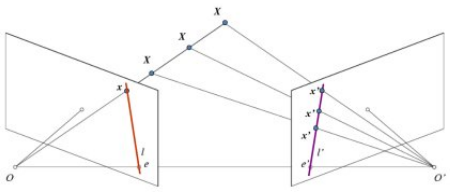


Figure 3.1: Intuition behind the epipolar geometry. [3]

Figure 3.2 shows the system tracking during a test. In purple is the ground truth meanwhile in green it's the estimated trajectory of the camera, the pink arrow shows the current heading of the camera extracted from the rotation matrix, and in the top left corner is again the heading in degrees.

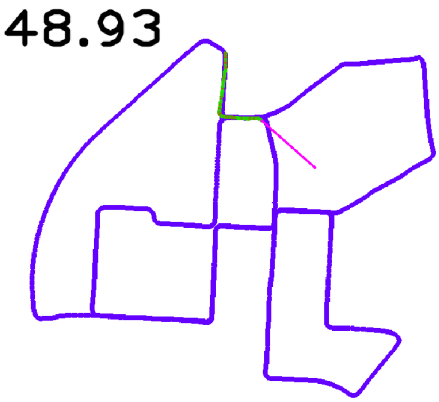


Figure 3.2: The system tracking live.

3.1.1 Known Issues

The estimation of the camera motion using epipolar geometry is not a trivial task, and there are a number of issues that need to be addressed.

- First of all, the *decomposition of the essential matrix*, that is the matrix that describes the epipolar geometry, is an ill-posed problem, meaning that there are multiple solutions that satisfy the constraints. We have 4 possible combinations of rotation and translation that satisfy the

constraints, and only one of them is the correct one, what we did to find the correct combination was to reproject the 3D points for each pair of R and t and then pick the combination that minimizes the number of points that end up reprojected behind the camera.

- It is also known that epipolar geometry does not fare well with complanar points, meaning that if the camera moves in a way that the points are all on the same plane or too far away, the estimation of the motion becomes very difficult.

3.2 Model Limitations

A big problem is due to the fact that we are working with a *monocular camera*, meaning that we don't have the depth information; this means that we can't really distinguish between a close point that moves a lot and a far point that moves a little, and this becomes evident when we try to estimate the translation, which is only defined up to a scale factor. We really can only accurately estimate the direction of movement, not the magnitude. While there are more sophisticated methods to estimate the scale factor that take into account the position of the camera, as well as the size of known objects in the environment, it was outside of the scope of the project and we decided to opt for a simpler approach:

- During normal estimation we assume that the camera moves at constant speed and we try to approximate the scale factor by measuring the difference in distance between the position of couples of estimated 3D points in consecutive frames.
- During benchmarking we extrapolate the absolute scale factor directly from the ground truth data by computing the distance between the position of the camera in consecutive frames.

This is a known issue and it's a common problem in the field of monocular visual odometry, and it's usually addressed by using additional sensors, such as an IMU or a GPS, or by using a stereo camera.

4 Evaluation

In order to evaluate the performance of the system, a set of metrics and tests have been defined. The tests are designed to evaluate the performance of the system in different scenarios, such as different camera motions, and different environments. We decided to focus on two parameters: robustness of the system to drift and execution time.

4.1 Error Function

To accurately estimate the drift of the system it's not enough to measure the distance of the agent at a current time from the ground truth position at the same instant. This is because the drift is a cumulative error that grows over time. For this very reason, the error function we developed does not take into account the absolute position of the agent but instead the relative movement, both in rotation and translation, between two consecutive frames. The error function is defined as the difference between the ground truth motion and the estimated motion.

4.1.1 Rotation Error

The rotation delta is extracted from consecutive frames as the product between the rotation matrix of the current frame and the transpose of the rotation matrix of the previous frame. The same process is then applied between the ground truth rotation delta and the estimated rotation delta. The error is then calculated as the angle of the rotation matrix obtained from the difference between the two rotation deltas and converted into degrees.

$$\Delta\mathbf{R} = \mathbf{R}_{t'} \times (\mathbf{R}_t)^T$$

$$\mathbf{R}_{err} = \left| deg(\Delta\mathbf{R}_{est} \times (\Delta\mathbf{R}_{gt})^T) \right|$$

4.1.2 Traslation Error

The translation delta is extracted from consecutive frames as the difference between the translation vector of the current frame and the translation vector of the previous frame. Since there might be an accumulated error in the rotation matrix, it is not enough to evaluate the distance between the two vectors. Before that, we realign the movement vectors using the absolute rotation delta between the pose of the ground truth and the pose of the estimated motion. The error is then calculated as the distance between the two vectors.

$$\Delta\mathbf{R}_{abs} = \mathbf{R}_{gt} \times (\mathbf{R}_{est})^T$$
$$\Delta\mathbf{t} = \mathbf{t}_t - \mathbf{t}_{t'}$$
$$\Delta\mathbf{t}_{est'} = \Delta\mathbf{R}_{abs} \times \Delta\mathbf{t}_{est}$$
$$\mathbf{t}_{err} = ||\Delta\mathbf{t}_{gt} - \Delta\mathbf{t}_{est'}||$$

4.1.3 Squishing

Once the rotation and translation errors are calculated, the final error is calculated as the sum of the two errors. The last step is to plug the total error into a function that will return a value between 0 and 1. Since the range of error is always nonnegative, the squishing function is defined (as shown in Figure 4.1) as:

$$err = \frac{\mathbf{t}_{err} + \mathbf{R}_{err}}{\frac{1}{2} + (\mathbf{t}_{err} + \mathbf{R}_{err})}$$

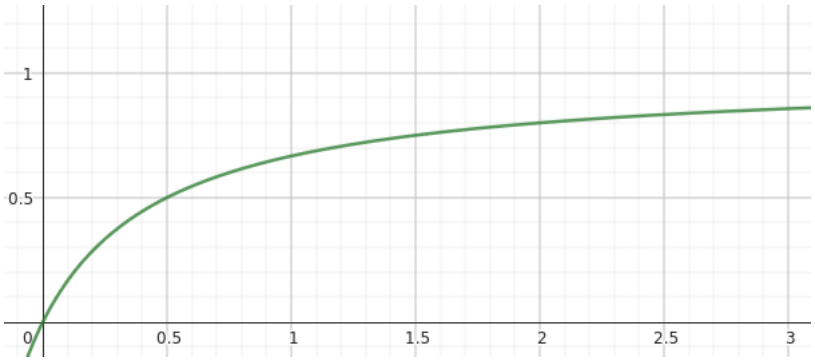


Figure 4.1: Shape of the error function.

4.2 Tests

To find the best set of parameters for the system, a series of tests have been defined. Such tests were conducted using various sequences in the KITTI dataset, for this we built a custom dataset loader *kitti_loader.py* that allows us to easily switch between different sequences as well as quickly perform some utility operations on the dataset.

The first set of tests is designed to evaluate the performance of the system both in terms of accuracy and execution time, focusing mainly on the different combinations of feature extraction methods and feature matching techniques. The second set of tests focused more on finetuning additional the parameters for denoising and scaling.

Each of the evaluated methods presents advantages as well as drawbacks. The aim of this analysis is to explore such characteristics in order to choose the best approach given a certain scenario.

Each test was run in different instances to prevent the occurrence of any kind of bias due to memory leaks or other issues.

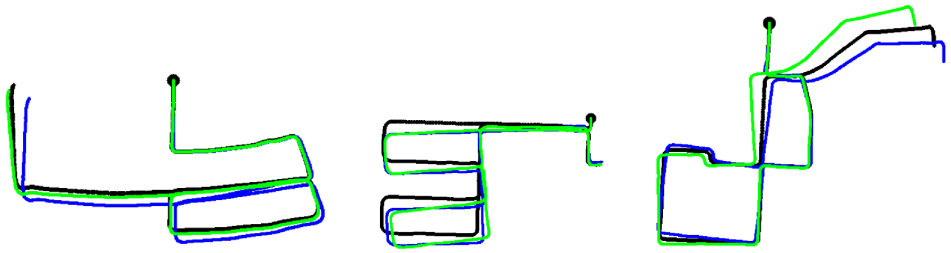


Figure 4.2: Examples of multiple runs on different sequences.

4.2.1 Tuning Feature Matchers

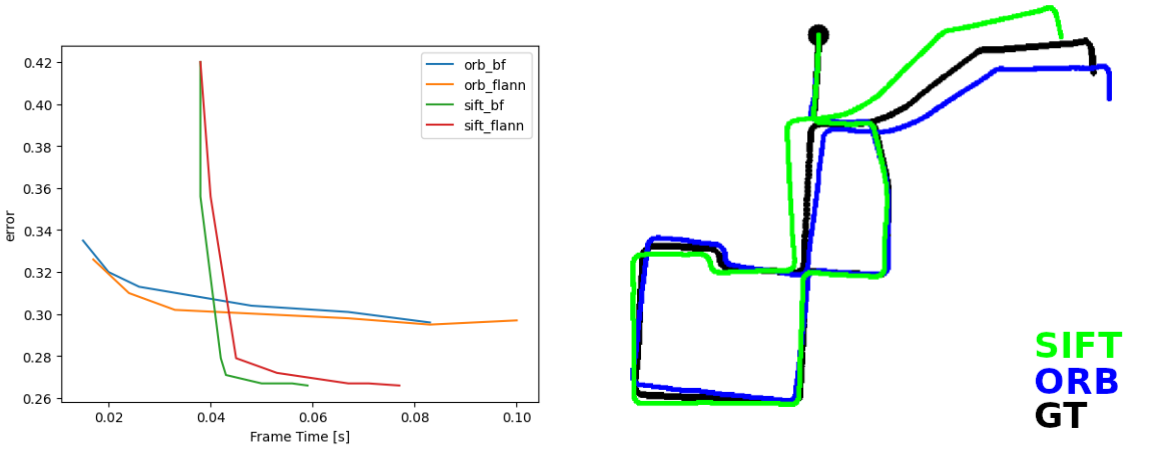
To compare the performance of different methods, we ran an initial test on SIFT (paired up with the brute force matcher and the FLANN matcher), and on ORB (coupled with the same matchers). For each run, we set a cap for the number of features extracted. Specifically, we used 50, 100, 500, 1000, 2000, 3000, and 6000 features for SIFT and 1000, 2000, 3000, 6000, 8000 and 10000 features for ORB. This allowed us to evaluate the correlation between the measured error and the speed of computation, which is expressed in FPS and frame time. The results of this evaluation showed that the method leading to the lowest average error is SIFT paired with FLANN, as shown in Table 4.1, and that generally speaking a lower frame time leads to a higher average error due to diminished precision.

Method	Matcher	Features Cap	err_avg	err_max	FPS	frame_time
SIFT	FLANN	6000	0.266	0.754	13	0.077
SIFT	BF	6000	0.266	0.769	17	0.059
SIFT	BF	3000	0.267	0.769	18	0.056
SIFT	FLANN	2000	0.267	0.772	15	0.067
SIFT	BF	2000	0.267	0.762	20	0.05
SIFT	FLANN	3000	0.267	0.776	14	0.071
SIFT	BF	1000	0.271	0.782	23	0.043
SIFT	FLANN	1000	0.272	0.784	19	0.053
SIFT	FLANN	500	0.279	0.858	22	0.045
SIFT	BF	500	0.279	0.883	24	0.042
ORB	FLANN	8000	0.295	0.864	12	0.083
ORB	BF	10000	0.296	0.921	12	0.083
ORB	FLANN	10000	0.297	0.914	10	0.1
ORB	FLANN	6000	0.298	0.903	15	0.067
ORB	BF	8000	0.301	0.906	15	0.067
ORB	FLANN	3000	0.302	0.91	30	0.033
ORB	BF	6000	0.304	0.906	21	0.048
ORB	FLANN	2000	0.31	0.901	42	0.024
ORB	BF	3000	0.313	0.93	39	0.026
ORB	BF	2000	0.32	0.905	51	0.02
ORB	FLANN	1000	0.326	0.999	58	0.017
ORB	BF	1000	0.335	0.911	68	0.015
SIFT	FLANN	100	0.356	0.938	25	0.04
SIFT	BF	100	0.356	0.938	26	0.038
SIFT	FLANN	50	0.42	0.994	26	0.038
SIFT	BF	50	0.42	0.994	26	0.038

Table 4.1: Results of first test.

Observing Table 4.1, it is clear that SIFT reaches a better precision compared to ORB. It must be noted that SIFT's precision plateaued after around 500 features are extracted; this is probably due to the fact that SIFT rarely gets more than that amount of actual features from each frame (although it is strictly dependent on the provided data), but when it does that results in a marginal improvement in precision at a heavy cost in terms of speed.

By looking at the last column it's clear how SIFT struggles to reach the same speed of computation as ORB. When the feature extracted per frame exceed 500, a sharp drop in frame rate can be observed. In contrast, ORB is able to maintain a fairly high frame rate even when the number of features extracted is increased; as a result, we can get much higher speeds without compromising too much on precision.



(a) Correlation between error and frame time (lower is better).

(b) Comparison between different methods of visual odometry (green and blue lines) and ground truth (black line).

Figure 4.3: Estimated tracks.

This behavior is also confirmed by the graph in Figure 4.3, where it's clear how SIFT error skyrockets as soon as the time frame diminishes (due to a lower number of features being extracted). On the other hand, ORB's error is higher when it comes to a higher frame time, but it's also more stable and doesn't grow as much when the frame rate increases, staying stable up to around 40 FPS.

It's interesting how the two methods present a trade-off between speed and precision and the lines representing the two methods cross at around 30FPS, meaning that for any application requiring a frame rate higher than that, ORB would be the best choice, while if lower frame rates were acceptable SIFT would offer better precision.

The method used for feature extraction and macthing within a system is strictly dependent on the needs of the system itself. If the speed of computation is not considered to be crucial, then SIFT could be thought of as the preferred feature detection algorithm. On the other hand, ORB resulted in being fairly faster in terms of computational speed, making it the best candidate when it comes to time-sensitive tasks.

4.2.2 Scale factor

Further tests were run to gain a better understanding of the impact different parameters have on the global final performance. The first of said parameters is the scale factor. We ran tests setting a scale factor of 0.25, 0.5, and 1 on both ORB and SIFT with the FLANN matcher. The results (Table 4.2) showed that, as expected, a decrease in the scale factor led to an increased average error and a substantial increase in the speed of computation.

Method	Scale	err_avg	err_max	FPS	frame_time
SIFT	0.25	0.39	0.905	100	0.01
SIFT	0.5	0.272	0.692	40	0.025
SIFT	1	0.236	0.622	18	0.056
ORB	0.25	0.629	0.997	149	0.007
ORB	0.5	0.332	0.888	31	0.032
ORB	1	0.271	0.837	11	0.091

Table 4.2: Results of test on scale factor.

4.2.3 Denoising

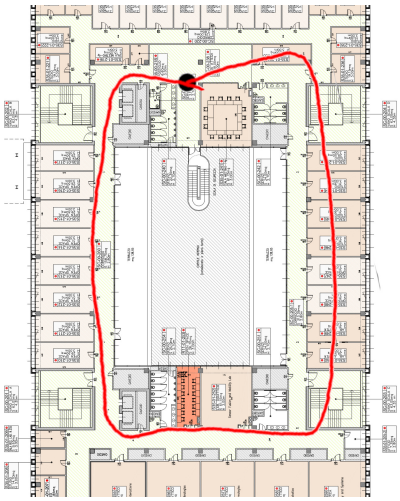
Finally, tests were run to determine the impact of varying the dimension of the kernel used for denoising. The results can be observed in Table 4.3. For both SIFT and ORB, the denoising factor is not particularly relevant in terms of the committed error. Increasing the size of the kernel used for the Gaussian blur does not appear to cause a relevant increase in the average error, although the speed of computation presents a slight improvement, maybe due to fewer features tied to noise being detected by the feature extractors. This means that an increase in terms of speed of computation can be obtained with denoising without the error being affected in a relevant way. It is however worth noting that said improvement is not particularly substantial.

Method	Denoise	err_avg	err_max	steps_sec	frame_time
SIFT	0	0.236	0.622	19	0.053
SIFT	3	0.234	0.634	19	0.053
SIFT	9	0.237	0.731	20	0.05
SIFT	11	0.245	0.632	20	0.05
ORB	0	0.271	0.837	10	0.1
ORB	3	0.274	0.891	11	0.091
ORB	9	0.271	0.78	23	0.043
ORB	11	0.277	0.878	27	0.037

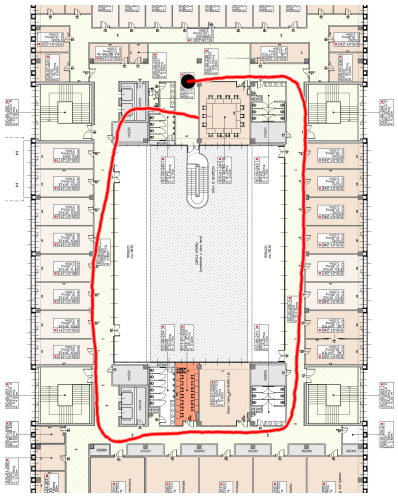
Table 4.3: Results of test on denoising.

4.2.4 Testing in real-world scenarios

Although to evaluate the performance of the system we mainly focused on the KITTI dataset, the system is built to run with any kind of video, provided that the camera is calibrated. The following test was conducted using a video taken on the second floor of our department (*Polo Tecnologico F. Ferrari (Povo 1)*) with an optically stabilized fisheye camera, walking at a constant pace.



(a) Using SIFT @ 0.5 scale with a 500 features cap.



(b) Using SIFT @ 1 scale with a 8000 features cap.

Figure 4.4: Tracking done in Povo 1.

We can observe that even though no loop closure system is in place and even the relative scale of motion is not known, the system is able to track the movement of the camera with good precision. The cumulative drift is not particularly high, and the system is able to keep up with the movement of the agent.

5 Conclusions

The aim of the project was to understand challenging aspects of Monocular Visual Odometry by developing and analyzing such system. In particular, a monocular system was implemented in all of its components, from camera calibration to motion estimation. The entire project was built with scalability and modularity in mind, in order to allow for easy integration of new features and methods; this turned out to be crucial when particular issues were encountered and a more in-depth analysis was needed.

A big role was played by the testing framework. During and after development, the system underwent extensive testing: we ran more than a thousand tests to compare the performances of the different methods and techniques and to evaluate the impact of different parameters on the system's performance. The tests were crucial to understand the tradeoffs between accuracy and speed of computation and to fine-tune the system.

It turns out that there isn't a one-size-fits-all solution, the best method or technique depends on the specific use case and on the requirements of the system. For example, if the system needs to be fast and accuracy is not a priority, then the ORB algorithm is to be considered the best choice, whereas the SIFT algorithm is best suited when prioritizing accuracy over speed.

Due to the modularity of the system, it is possible to easily upgrade the framework with new parts. The first step for a second iteration of the project would be the implementation of a more sophisticated motion estimation algorithm that takes into account the scale factor and the position of the camera. After that, the integration of loop closure would render the system more robust to drift, and the addition of a bundle adjustment algorithm would improve the accuracy of the 3D points.

In conclusion, the project was a success: the system was able to estimate the motion of the camera with a good degree of accuracy, and the tests showed that the system is robust to different conditions. The system is also very flexible and can be easily upgraded with new features and methods, making it a good starting point for future research in the field of Monocular Visual Odometry.

Bibliography

[1] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. Brief: Binary robust independent elementary features. In Kostas Daniilidis, Petros Maragos, and Nikos Paragios, editors, *Computer Vision – ECCV 2010*, pages 778–792, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[2] David G Lowe. Object recognition from local scale-invariant features. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 1150–1157. Ieee, 1999.

[3] OpenCV. Epipolar geometry.

[4] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *2011 International Conference on Computer Vision*, pages 2564–2571, 2011.

[5] D Arul Suju and Hancy Jose. Flann: Fast approximate nearest neighbour search algorithm for elucidating human-wildlife conflicts in forest areas. In *2017 Fourth International Conference on Signal Processing, Communication and Networking (ICSCN)*, pages 1–6, 2017.

[6] Deepa Viswanathan. Features from accelerated segment test (fast). 2011.

[7] Wikipedia contributors. Epipolar geometry.