RGBD SLAM with ROS and Visual Odometry

EECE 6356 Final Report

Arjun Keerthi
School of Engineering
Vanderbilt University
arjun.b.keerthi@vanderbilt.edu

***Abstract*—This report demonstrates running RGBD SLAM with ROS on a Raspberry Pi 4 Model B mounted on a remote-controlled vehicle, using an Xbox Kinect v1 as the only sensor. Additionally, a basic visual odometry (VO) algorithm is implemented in Python and run using various feature detectors. The underlying theory behind SLAM and visual odometry is outlined as well. Utilizing the KITTI Odometry Benchmark dataset, it was found that compared to the FAST and AGAST feature detectors, the GFTT detector provides odometry closest to the ground truth. Also, it was found that a Raspberry Pi was capable of running ROS's RGBD SLAM implementation in real time but required driving the vehicle slowly to prevent loss of odometry quality.**

I. Introduction

A fundamental requirement for autonomous robotics is localization within an environment. One component for satisfying this is capturing accurate odometry for the motion of the robot. For autonomous vehicles, the simple solution is dead reckoning. Unfortunately, this approach fails to account for motion not measurable by the sensors used for dead reckoning (e.g. wheel slippage) [1]. Visual Odometry (VO), on the other hand, better avoids such issues by estimating vehicle motion from camera images. In addition to requiring cheaper sensors than IMU, GPS, or LASER scanners, Visual Odometry has been shown to give more accurate results than dead reckoning, so much so that VO algorithms were developed for and deployed on the Mars Rover to provide robust odometry on Mars's rough, uneven terrain [2].

Another component that is important for autonomous robotics is mapping of their environment. Common map representations include feature maps and occupancy grids. The former utilizes geometric shapes like points and lines to represent the environment while the latter uses a grid of cells, each containing the probability that it is occupied by something in the environment. [3]. Simultaneous Localization and Mapping (SLAM) attempts to construct a map of the environment while also localizing the robot within the map, making it very important for real-time robots that need to navigate novel environments.

II. Background

A. Visual Odometry

VO dates back to the work of Moravec in 1980 in which he designed a sliding-stereo system on a planetary rover to determine motion, providing the basis for future work in stereo VO systems [4]. Monocular VO is also a useful technique, both because it requires cheaper hardware and is required for environments where features are so far away that the stereo case devolves into the monocular case [5]. Nister et al developed the first real-time monocular VO system, using their novel 5-point RANSAC algorithm with 3D-to-2D motion estimation [6].

Scaramuzza's and Fraundorfer's two-part tutorial on VO is referenced extensively here to outline the theory behind VO [5], [7]. The general algorithm for VO is presented in Fig. 1.

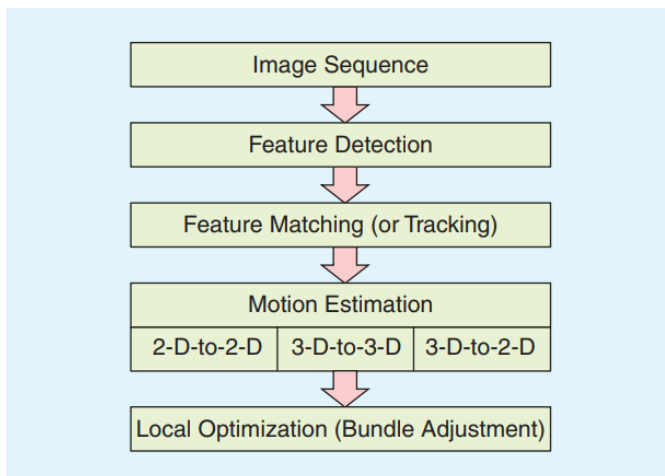

Fig. 1. General VO steps [5]

The monocular VO problem (which is the focus of this report) consists of a robot capturing camera images as it moves through an environment, building a sequence of images $I_{0:n} = \{I_0, I_1, ..., I_n\}$. The goal is to estimate the transformation $T_{k,k-1} \in \mathbb{R}^4$ between subsequent images I_{k-1} and I_k:

$$T_{k,k-1} = \begin{bmatrix} R_{k,k-1} & t_{k,k-1} \\ 0 & 1 \end{bmatrix},$$

with $R_{k,k-1}$ as the 3x3 rotation matrix and $t_{k,k-1}$ as the 3x1 translation vector [5].

After obtaining an image sequence from a rigidly mounted camera on the robot, features are extracted from each image. A multitude of feature detectors exist, three of which are compared in this report. Next, features are either matched or tracked between subsequent image frames. Feature matching involves detecting features in both image frames and then comparing all features in the first image to all features in the second image to find matches. Sum of Squared Differences

(SSD) and Normalized Cross-Correlation (NCC) are common methods for achieving this. It can also be optimized by only searching for matches along epipolar lines [7].

Feature tracking searches the second image for areas of similar appearance to features detected in the first image. The Kanade-Lucas-Tomasi (KLT) feature tracker is used in this report. It assumes constancy of brightness for the same features across consecutive images so that the following equation holds:

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

where dt is the difference in time between consecutive images. Then, by taking the 1st order Taylor series expansion of the right side and rearranging terms, the following is obtained:

$$I_x u + I_y v = 0,$$

where

$$u = \frac{dx}{dt}, v = \frac{dy}{dt}.$$

KLT uses a 3x3 region, providing a system of 9 equations and 2 unknowns (u and v) that be can solved to find the pixel location of the feature in successive images [8].

Once features are tracked between consecutive images, motion can be estimated. Scaramuzza outlines 3 variants: 2D to 2D, 3D to 2D, and 3D to 3D. This report makes use of the first method: estimating $R_{k-1,k}$ and $t_{k,k-1}$ from features specified as 2D image coordinates that are extracted from I_{k-1} and I_k. The essential matrix $E_{k,k-1}$ is a 3x3 matrix that specifies the motion parameters between two camera views (up to an unknown scalar):

$$E_{k,k-1} \backsimeq \hat{t}_{k,k-1} R_{k,k-1},$$

where $\hat{t}_{,k-1k}$ is the skew-symmetric matrix associated with $t_{k,k-1} = \begin{bmatrix} t_x & t_y & t_z \end{bmatrix}^T$:

$$t_{k,k-1} = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix}.$$

The essential matrix satisfies the following epipolar constraint, where p and p' are corresponding features in the two consecutive images and must lie on the same epipolar lines:

$$p'^T E p = 0.$$

The feature points are normalized with the form $p = \begin{bmatrix} u & v & 1 \end{bmatrix}^T$. With a set of features tracked across consecutive images, a system of equations from the above epipolar constraint can be formed and solved to determine the essential matrix. The singular value decomposition (SVD) is a good approach for solving this. Random sample and consensus (RANSAC) is utilized to efficiently determine a good choice for the essential matrix by selecting a set of tracked features, solving the corresponding system to find a model hypothesis for E_k, and then testing the remaining feature points to determine outliers (i.e. points that do not conform to the model hypothesis). This procedure is repeated multiple times until a robust model is found [7].

Nister et al devised an efficient 5-point RANSAC procedure for estimating the essential matrix and is utilized in this report. This method rewrites the epipolar constraint:

$$\tilde{p}^T \tilde{E} = 0,$$

where

$$\tilde{p} = \begin{bmatrix} p'_1 p_1 & p'_1 p_2 & p'_1 p_3 & p'_2 p_1 & p'_2 p_2 & p'_2 p_3 & p'_3 p_1 & p'_3 p_2 & p'_3 p_3 \end{bmatrix},$$

$$\tilde{E} = \begin{bmatrix} E_{11} & E_{12} & E_{13} & E_{21} & E_{22} & E_{23} & E_{31} & E_{32} & E_{33} \end{bmatrix}^T.$$

Then, a 5x9 matrix is constructed by stacking each $\tilde{p}^T$ for 5 feature point pairs. With this reformulation, the paper shows how to find four 9x1 vectors that span the null space of this 5x9 matrix. These four vectors directly correspond to four 3x3 matrices, of which E is a linear combination [6].

Next, $R_{k,k-1}$ and $\hat{t}_{k,k-1}$ can be extracted from E_k via the following ($E = USV^T$ via SVD):

$$R = U(\pm W^T)V^T,$$

$$\hat{t} = U(\pm W)SU^T.$$

As the above implies four possibilities for $R_{k,k-1}$ and $\hat{t}_{k,k-1}$, a cheirality check is imposed to ensure the feature points lie in front of the camera [9], [6].

Finally, now that the transformation $T_{k,k-1}$ is fully estimated, the new camera pose can be computed:

$$C_k = C_{k-1} T_{k,k-1},$$

where C_{k-1} is the previous camera pose (corresponding to image I_{k-1}) and C_k is the current camera pose. Here, each C_i is the 3x4 projection matrix $[R_i|t_i]$ [5]. This relation is applied for every incoming image to continuously estimate the position of the camera and thus provide odometry for the robot. Fig. 2 summarizes the steps for monocular VO.

Algorithm 1. VO from 2-D-to-2-D correspondences.

1) Capture new frame I_k
2) Extract and match features between I_{k-1} and I_k
3) Compute essential matrix for image pair I_{k-1}, I_k
4) Decompose essential matrix into R_k and t_k, and form T_k
5) Compute relative scale and rescale t_k accordingly
6) Concatenate transformation by computing $C_k = C_{k-1} T_k$
7) Repeat from 1).

Fig. 2. 2D VO steps [5]

B. Simultaneous Localization and Mapping

While VO addresses local consistency for the robot's trajectory, SLAM attempts to build a global estimate of the trajectory, necessitating a globally consistent map as well. VO can be a component of SLAM, but because SLAM consists of

numerous components, this section will be less technical than the VO section.

Fig. 3 provides a general overview for the steps in SLAM. Odometry data captures movement made by the robot and is used to construct a prediction of the state of the system (robot pose and map features). Then, measurements made by the robot's sensors are used to apply an update to the state. Motion models for the robot are applied to determine how odometry will correspond to changes in state, while various probabilistic and optimization techniques are employed to update this state based on the information attained by sensors [3].

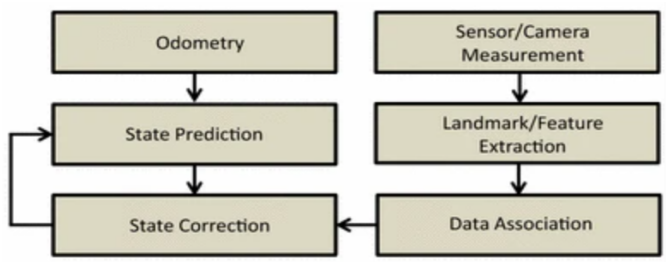

Fig. 3. Outline of SLAM [3]

There are multiple approaches to SLAM. EKF-SLAM is one common form of online-slam (maintains only a local representation of the environment). It builds on top of the Extended Kalman Filter (EKF). The state vector y_t at time t is represented as follows:

$$y_t = \left[\, x \;\; y \;\; \theta \;\; m_{1,x} \;\; m_{1,y} \;\; s_1 \;\; m_{2,x} \;\; m_{s,y} \;\; s_2 \;\; \ldots \;\; m_{N,x} \;\; m_{N,y} \;\; s_N \,\right],$$

where x, y, θ specify the robot's coordinates and $m_{i,x}, m_{i,y}$ specify the coordinates of the ith landmark/feature (s_i is the corresponding signature). EKF-SLAM aims to determine the posterior distribution $p(y_t|z_{1:t}, u_{1:t})$. This reflects this method being "online" as it only estimates the distribution for the current state (versus "full" SLAM that aims to estimate the full posterior $p(y_{1:t}|z_{1:t}, u_{1:t})$). Like the standard EKF, this distribution is represented with a multivariate Gaussian distribution, parameterized wholly by the mean μ_t and covariance Σ_t [10].

Using odometry data (e.g. from VO), an initial belief of the posterior is determined by using the robot's motion model to find $\bar{\mu}_t$ and $\bar{\Sigma}_t$. Unlike the EKF however, the state vector can change in size as the number of landmarks N changes as the robot discovers new areas. As a result, before applying the measurement update, all observed features must be compared to a matrix stored throughout the robot's run containing landmark position information, updating this matrix if new landmarks are found or if existing landmark positions can be improved [10], [3].

Unfortunately, EKF-SLAM has disadvantages, including quadratic runtime complexity in updating landmark information, which in turn makes data association difficult (i.e. comparing current landmarks to previously seen ones). One popular approach that addresses some of EKF-SLAM's issues is FastSLAM, a full SLAM approach that utilizes a particle filter. This system maintains a set of particles, each containing a sample robot path as well as a map. Each iteration of the algorithm is similar to that of a particle filter, sampling a new pose and updating the features. The benefit of approaching the problem this way stems from the ability to factor the full posterior distribution as follows:

$$p(y_{1:t}|z_{1:t}, u_{1:t}, c_{1:t}) = p(x_{1:t}|z_{1:t}, u_{1:t}, c_{1:t}) \prod_{n=1}^{N} p(m_n|x_{1:t}, z_{1:t}, c_{1:t}).$$

This formulations factors out the distribution for the path of the robot (term outside of product). This implies that the posteriors for each landmark is conditionally independent given the path and so (since the path was factored out) each landmark can be updated independently. As a result, the distribution for each feature can be estimated with a separate EKF versus using a single EKF to estimate a Gaussian for all features simultaneously. This in turn makes the update step much more efficient than in EKF-SLAM as only low-dimensional EKFs are required for each feature. Additionally, the above factorization implies that each particle will have a copy of the estimator for each feature, which means that data association can be done on a per-particle basis rather than the more intensive single data association across the entire filter required by EKF-SLAM [10].

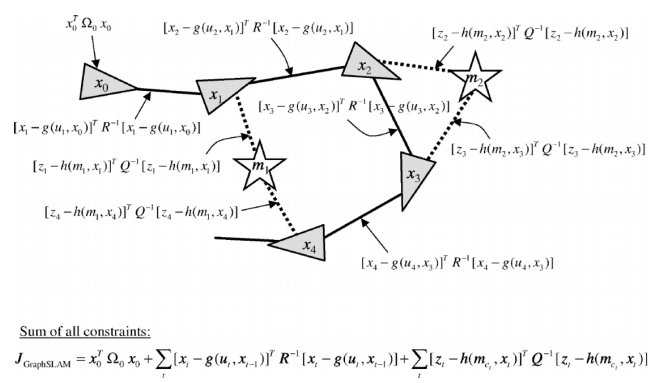

Fig. 4. Diagram of Graph SLAM [10]

Another very popular approach is GraphSLAM. Here, the aim is to construct a sparse graph that represents constraints modeled by the data acquired. Fig. 4 provides an example graph for a simple scenario. The nodes in this graph either represent the robot's pose (x_i) or the location of a feature (m_i). Edges (solid) between robot poses represent a constraint on the motion model $g(u_{i+1}, x_i)$ while edges (dashed) between robot poses and features represent constraints on the measurement model $h(m_i, x_i)$. Each of these constraints are nonlinear and are added to the graph as the robot moves.

This process of updating the graph as new poses/features are acquired makes GraphSLAM very efficient in growing maps. Unlike EKF-SLAM, where every iteration required updating μ, Σ and performing data association, GraphSLAM adds soft constraints as it accumulates information, allowing its maps to be much larger than for EKF-SLAM.

When constraints are added to the graph, the target function $J_{GraphSLAM}$ (sum of all constraints, shown in Fig. 4) is minimized to yield the most likely graph to represent the robot and its environment. It is a function over the pose variables $x_{1:t}$ and feature locations m_i and requires solving a nonlinear least squares problem. Thrun outlines a procedure of recovering the map and robot pose by linearizing the constraints into an information matrix Ω and information vector ξ, reducing the matrix to contain just the robot poses, and calculating the full posterior distribution with $\mu = \Omega^{-1}\xi$. However, he mentions that more efficient approaches like Levenberg-Marquardt and conjugate gradient methods are likely more practical [10].

This report utilizes ROS's RTAB-Map (Real Time Appearance Based Mapping) SLAM implementation. It is a GraphSLAM approach, dividing computation into a front-end and back-end, where the front-end is responsible for updating the graph with new constraints and the back-end performs the optimization procedure to recover the map and robot path. Specifically, this report tests SLAM with an RGBD sensor (Xbox Kinect v1), which provides depth images and associated RGB images. 3D coordinates for features extracted from RGB images are consequently immediately available (bypasses scale issues that monocular VO must solve) and SLAM proceeds using the ideas introduced for GraphSLAM [11].

III. Methods

This report accomplished the following: 1) Implemented a basic VO program in Python, and 2) Ran ROS's RTAB-Map SLAM package on a Raspberry Pi 4 Model B mounted on a remote-controlled vehicle with an Xbox Kinect v1 sensor.

A. VO Program in Python

The algorithm used for VO stems from the theory discussed in section II.A but will be summarized here. Singh has provided many details towards implementing this algorithm, which are utilized in this report [12]. OpenCV is used extensively for its many functions that implement the ideas required for VO. Also, the implementation provided with this report reads images from the Kitti Odometry dataset [13]. Please refer to the Appendix for the code.

First, the initial two images are loaded and features are extracted from the first image. OpenCV provides the `Feature2D` abstract class, which includes a `detect()` function that can be easily called on each image. Features from Accelerated Segment Test (FAST) [14], Good Features to Track (GFTT) [15], and Adaptive and Generic Accelerated Segment Test (AGAST) [16] feature detectors are evaluated in this report.

Next, features are tracked from the first image to the second image using OpenCV's `calcOpticalFlowPyrLK()` function, which uses the KLT tracker. Points tracked to the second image that either are not good or are located outside the image are eliminated from both the first and second images' point lists.

Then, the essential matrix is computed between the two images via the `findEssentialMat()` function (employs 5-point RANSAC). The rotation matrix R and translation vector t are recovered with `recoverPose()`. In order to use these two functions, the camera matrix must be provided. The Kitti odometry data set supplies this in a separate calibration file in a 12x1 vector format. This vector is a flattened 3x4 projection matrix $P = [K|t]$, where K is the 3x3 camera matrix and t is the 3x1 translation vector from the origin of the camera-array system (Kitti data set was acquired via a stereo setup) to the current camera's origin. The code provided for this report utilizes the images from the left rgb camera, so this vector is zero [17].

Then, the remaining images are looped over, repeating the above procedure between the previous and current images. To compute the global translation vector from the camera's starting pose to the pose in the current image, the following equations are used:

$$t_{global,k} = t_{global,k-1} + scale * R_{global,k} t_{current},$$

$$R_{global,k} = R_{global,k-1} R_{current},$$

where $R_{global,i}$ is the rotation matrix and $t_{global,i}$ the translation vector from the starting pose to the pose in image I_i. This is equivalent to the equation below discussed in section II.A

$$C_k = C_{k-1} T_{k,k-1},$$

$$T_{k,k-1} = \begin{bmatrix} R_{k,k-1} & t_{k,k-1} \\ 0 & 1 \end{bmatrix}$$

because the matrix multiplication works out as follows:

$$\begin{aligned} C_k &= C_{k-1} T_{k,k-1} \\ &= [R_{global,k-1} | t_{global,k-1}] \begin{bmatrix} R_{k,k-1} & t_{k,k-1} \\ 0 & 1 \end{bmatrix} \\ &= [R_{global,k-1} R_{k,k-1} | R_{global,k-1} t_{k,k-1} + t_{global,k-1}] \\ &= [R_{global,k} | t_{global,k}] \,. \end{aligned}$$

One issue here is determining the scale for translations. For this report, the scale was extracted from the Kitti dataset's ground truth poses. Since `recoverPose()` returns translation vectors with unit length, the Euclidean distance between the current and previous ground truth poses is directly the scale value. When applying this algorithm to a webcam or situations where ground truth is not known, 3D triangulation of points between image frames is necessary to determine scale [5].

Finally, to plot the vehicle's trajectory, $t_{global,i}$ is used to color the corresponding pixel in a Numpy matrix. The ground truth poses are also formatted as 12x1 vectors, so the terms corresponding to the translation vector in the flattened 3x4 projection matrix are extracted to plot these poses as well.

B. RTAB-Map with Kinect on Raspberry Pi

The main sensor used was the Xbox Kinect v1 sensor, paired with the Raspberry Pi 4 Model B as the sole microcontroller. The remote-controlled vehicle chosen was the Traxxas Slash 2WD for its size and durability. The Raspberry Pi was powered via a battery pack. The Kinect, however, without resorting to

Fig. 5. RC Vehicle - Side views

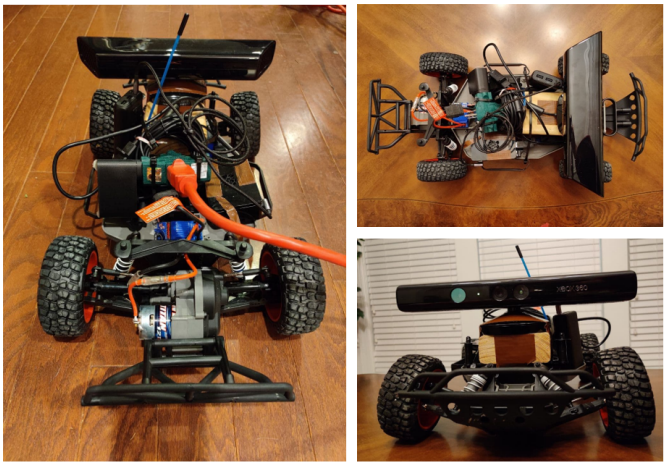

Fig. 6. RC Vehicle - Other views

invasive mods, can only be powered via an AC adapter. As a result, the vehicle was driven while connected to a wall outlet via a very long extension cord. See Fig. 5, 6 for images of the vehicle with the components mounted.

Dmitri Maslov's tutorial on running RTAB-Map on the Raspberry Pi 4 Model B was referenced extensively for this task, although the operating system and software versions were chosen slightly differently [18]. Ubiquity Robotics's Ubuntu 16.04 image for the Raspberry Pi was chosen as the operating system as it already had ROS Kinetic pre-installed (and has been used by the author previously). To interface with the Kinect sensor, the libfreenect Kinect drivers were installed, along with multiple other dependencies (see Maslov's tutorial for these - due to using ROS Kinetic versus Melodic as in the tutorial, the git branches for many of these packages had to be switched to Kinetic or Indigo).

One issue that arose after installing libfreenect was that ROS nodes subscribing to two or more topics published to by libfreenect would cause the Raspberry Pi to hang and become unresponsive. After further testing, the author found that the system would only function the very first time the subscriber was run (subsequent runs would crash immediately). After installing RTab-Map, the same pattern emerged: the SLAM system could only run on the first attempt; the Pi had to be restarted to run the system again.

Another issue was loss of odometry quality as the remote-controlled vehicle drove around the author's kitchen. Driving too quickly caused the quality to drop to 0, making the SLAM system lose track of its position in the map and halt until returned to a location it recognized. To avoid this and actually complete a loop, the vehicle had to be driven slowly.

Finally, to view the point cloud, occupancy grid, and graph, the `rtabmap-databaseViewer` command provided by RTAB-Map was used. Each slam run wrote the map to a database file, which was parsed by this command (see Experiments section for the views generated here) [19].

IV. Experiments

A. VO with Different Feature Detectors

The plots in Fig. 7, 8, 9, and 10 display the trajectories determined by the VO algorithm outlined in section III.A for different sequences in the Kitti dataset. Each plot has the path for the FAST detector in yellow, GFTT in red, AGAST in blue, and ground truth in green.

The errors were calculating by summing the Euclidean distance between the camera pose determined by VO and the ground truth pose for every camera image. The units for the trajectory plots are in meters. For frames-per-second (FPS), the number of frames was divided by the total time taken by the VO program to interpret the images. Sequence 00 and 02 were longer at 4000 images while 05 and 09 were shorter at 2500 and 1500 images, respectively. All runs were completed on a Lenovo X1 Carbon with an i7-7600U.

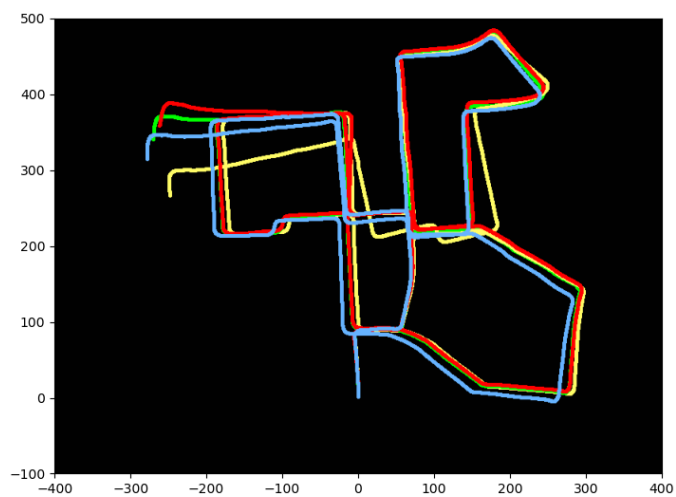

Fig. 7. Trajectories for Sequence 00

TABLE I
Sequence 00

Detector	Error	FPS
FAST	68289.42	8.51
GFTT	32109.28	8.00
AGAST	45537.18	7.63

The GFTT feature detector was the fastest in all sequences except sequence 00 and had the least error for all sequences except sequence 05. The AGAST feature detector had a slightly

lower error (except for its poor performance in sequence 09) but was slightly slower than the FAST detector. The plots reflect the differences in error as we generally see the GFTT trajectory (red) more closely mirroring the ground truth (green) compared to the other detectors' paths.

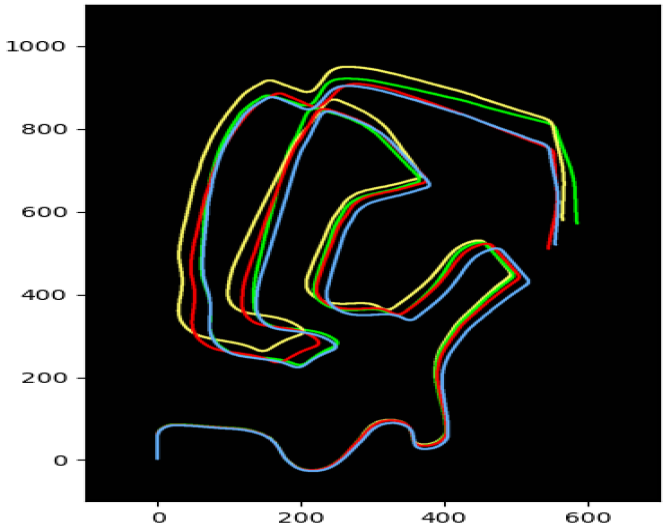

Fig. 8. Trajectories for Sequence 02

TABLE II
SEQUENCE 02

Detector	Error	FPS
FAST	119526.91	6.49
GFTT	90155.22	7.52
AGAST	113057.09	6.45

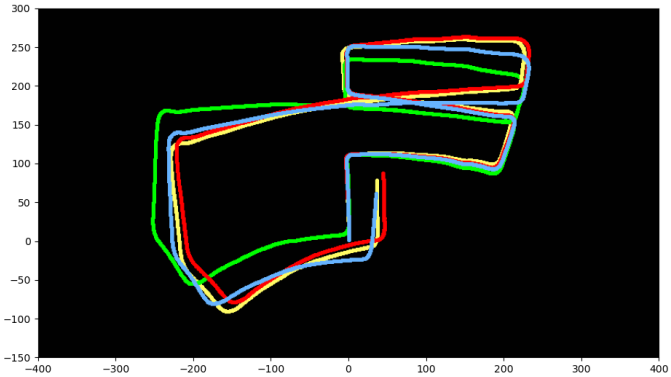

Fig. 9. Trajectories for Sequence 05

TABLE III
SEQUENCE 05

Detector	Error	FPS
FAST	71769.66	9.09
GFTT	82786.21	10.42
AGAST	55664.58	8.15

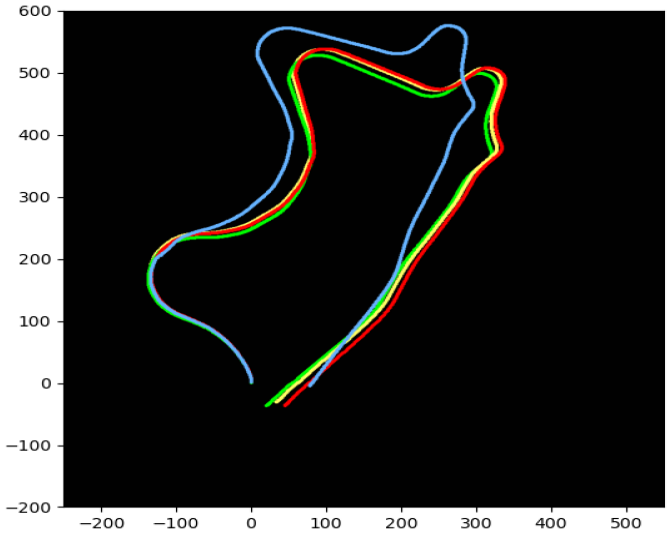

Fig. 10. Trajectories for Sequence 09

TABLE IV
SEQUENCE 09

Detector	Error	FPS
FAST	22045.31	8.38
GFTT	19264.75	9.12
AGAST	77018.87	6.83

B. RTAB-Map RGBD SLAM on Raspberry Pi in Kitchen

As explained in section III.B, RTab-Map was run on the Raspberry Pi using an Xbox Kinect sensor for RGBD SLAM. Fig. 11, 12, 13, 14, 15 display the data visualized from a run of this SLAM system while driving the remote-controlled vehicle around the author's kitchen. Fig. 11 demonstrates the SLAM system extracting and tracking features from consecutive images in order to estimate motion. Additionally, the depth image is also displayed (likely from the Kinect's published depth topic).

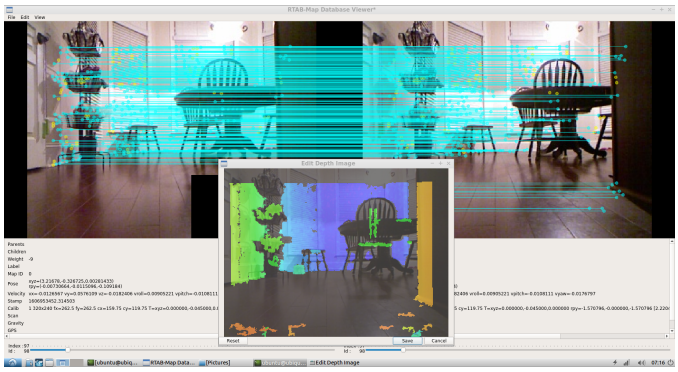

Fig. 11. Feature correspondence and depth perception during SLAM run

Fig. 12 shows the occupancy grid that was constructed. The grid pattern that RTAB-Map uses to segment the environment and the ability of this map representation to adequately represent the surroundings are both apparent. Certain features are

Fig. 12. Occupancy grid

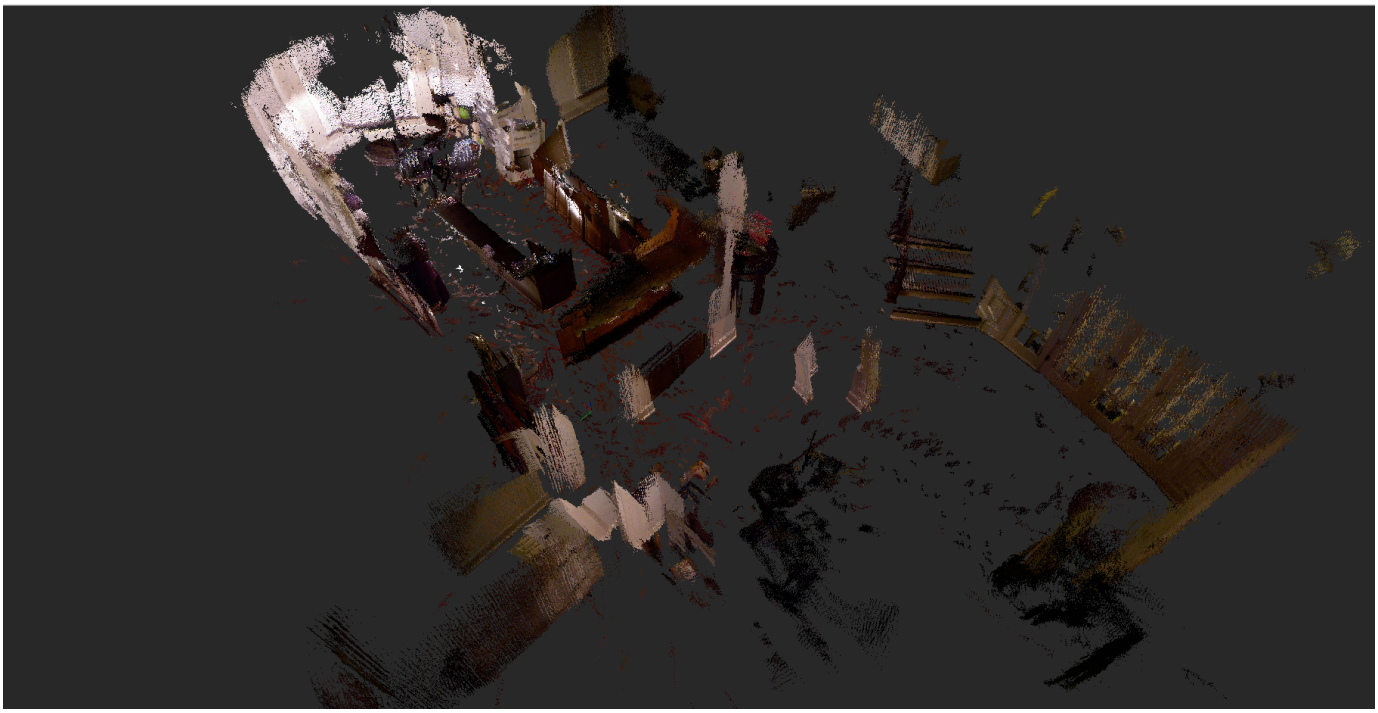

Fig. 13. Point cloud map

visible, such as the staircase at the middle-top and the kitchen island near the top-left.

Fig. 13 shows the point cloud 3D image for the environment in which the robot drove. It appears similar to the occupancy grid from Fig. 12, but the precision is much higher (as expected since this map representation utilizes geometric shapes, mostly points). For example, the window panes on the doors at the bottom-right are much more defined in the point cloud image. Additionally, the cabinets in the kitchen at the top-left are far clearer and fine details there are distinguishable. It is even possible to discern a bit of the dining table at the very top-left (past the island and cabinets).

Finally, Fig. 14 and 15 display the graph being updated throughout the GraphSLAM procedure in RTAB-Map. The zoomed in picture in Fig. 15 shows each node corresponding to the robot's pose at different times and that each blue line is an edge linking consecutive poses. RTAB-Map also colors certain edges yellow and red, likely indicating weak/misassociated

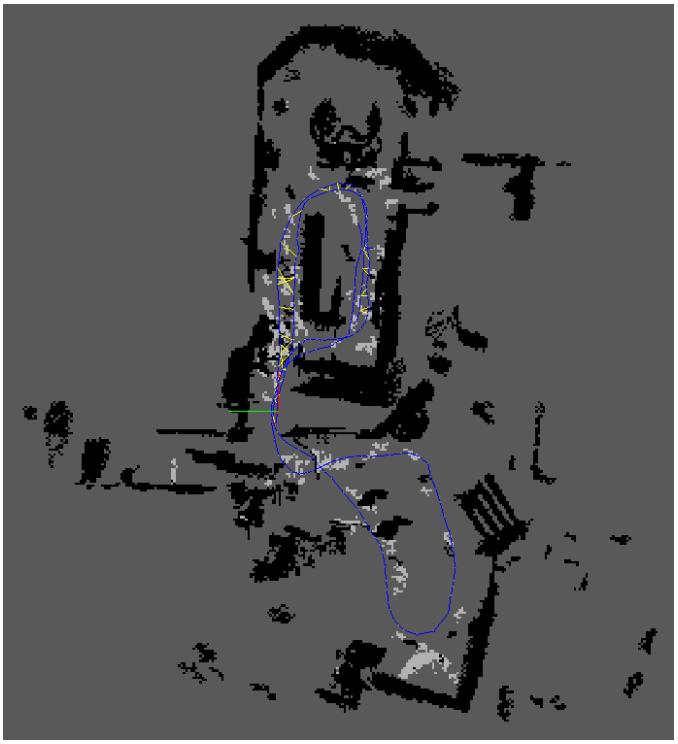

Fig. 14. Graph from RTAB-Map

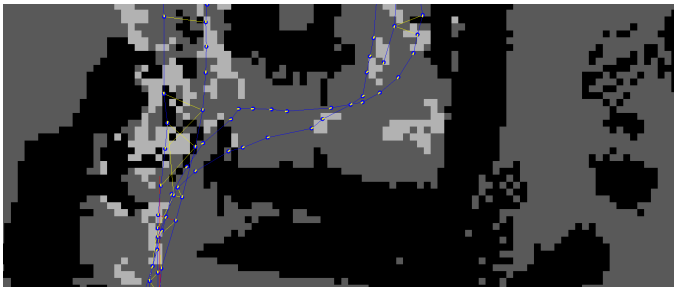

Fig. 15. Zoomed in view of SLAM graph

constraints between poses.

V. Conclusion

This report found that for the basic VO algorithm implemented, the Good Features to Track feature detector performed with the best speed and accuracy compared to the FAST and AGAST detectors. Additionally, the RTAB-Map SLAM package provided by ROS is capable of running in real-time on a Raspberry Pi 4 Model B with an Xbox Kinect v1 as the only sensor. The only drawbacks here are that the remote-controlled vehicle with this hardware had to be driven slowly in order to ensure the SLAM program did not lose track of its pose and that the Pi had to be restarted between every run.

Regarding future avenues to pursue, Scaramuzza et. al details a simplification of the essential matrix and a 1-point RANSAC procedure that takes advantage of nonholonomic constraints of Ackerman-style steering for vehicles that could result in a VO algorithm more performant than the one implemented in this paper [20]. Furthermore, the trajectory plots shown in section IV.A display significant drift as the sequences progressed. This could be addressed by incorporating bundle adjustment over a sliding window of images in the sequence [5]. Additionally, other SLAM implementations could be tested, such as DVO-SLAM [21] and ORB_SLAM2 [22].

References

[1] G. Roston and E. Krotkov, "Dead reckoning navigation for walking robots," *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*.

[2] D. Helmick, Y. Cheng, and S. Roumeliotis, "Path following using visual odometry for a mars rover in high-slip environments," *2004 IEEE Aerospace Conference Proceedings (IEEE Cat. No.04TH8720)*.

[3] K. Yousif, A. Bab-Hadiashar, and R. Hoseinnezhad, "An overview to visual odometry and visual slam: Applications to mobile robotics," *Intelligent Industrial Systems*, p. 289–311, Nov 2015.

[4] H. P. Moravec, "Obstacle avoidance and navigation in the real world by a seeing robot rover," *Robotics Institute of Carnegie Mellon*, Sep 1980.

[5] D. Scaramuzza and F. Fraundorfer, "Visual odometry: Part i: The first 30 years and fundamentals," *IEEE Robotics & Automation Magazine*, p. 80–92, Dec 2011.

[6] D. Nister, "An efficient solution to the five-point relative pose problem," *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.*

[7] F. Fraundorfer and D. Scaramuzza, "Visual odometry : Part ii: Matching, robustness, optimization, and applications," *IEEE Robotics & Automation Magazine*, vol. 19, no. 2, p. 78–90, 2012.

[8] "Optical flow." [Online]. Available: https://docs.opencv.org/3.4/d4/dee/tutorial_optical_flow.html

[9] "Camera calibration and 3d reconstruction." [Online]. Available: https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html\#gadb7d2dfcc184c1d2f496d8639f4371c0

[10] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. The MIT Press, 2005.

[11] S. Das, "Simultaneous localization and mapping (slam) using rtab-map," Sep 2018.

[12] A. Singh, "Monocular visual odometry using opencv," Jun 2015. [Online]. Available: https://avisingh599.github.io/vision/monocular-vo/

[13] [Online]. Available: http://www.cvlibs.net/datasets/kitti/eval_odometry.php

[14] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection," *Computer Vision – ECCV 2006 Lecture Notes in Computer Science*, p. 430–443, 2006.

[15] J. Shi and Tomasi, "Good features to track," *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition CVPR-94*, 1994.

[16] E. Mair, G. D. Hager, D. Burschka, M. Suppa, and G. Hirzinger, "Adaptive and generic corner detection based on the accelerated segment test," *Computer Vision – ECCV 2010 Lecture Notes in Computer Science*, p. 183–196, 2010.

[17] Ash, "Format of parameters in kitti's calibration file." [Online]. Available: https://stackoverflow.com/a/50211379

[18] D. Maslov, "Rgb-d slam with kinect on raspberry pi 4 ros melodic," May 2020. [Online]. Available: https://www.hackster.io/dmitrywat/rgb-d-slam-with-kinect-on-raspberry-pi-4-ros-melodic-ace795

[19] "Rgb-d handheld mapping." [Online]. Available: http://wiki.ros.org/rtabmap_ros/Tutorials/HandHeldMapping

[20] D. Scaramuzza, F. Fraundorfer, and R. Siegwart, "Real-time monocular visual odometry for on-road vehicles with 1-point ransac," *2009 IEEE International Conference on Robotics and Automation*, Jun 2009.

[21] C. Kerl, J. Sturm, and D. Cremers, "Dense visual slam for rgb-d cameras," *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013.

[22] R. Mur-Artal and J. D. Tardos, "Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras," *IEEE Transactions on Robotics*, vol. 33, no. 5, p. 1255–1262, 2017.

VI. Appendix

The following Python code was used to run the VO algorithm. Certain code blocks were commented out on runs in order to draw multiple trajectories over the same plot. The code can also be found here: https://github.com/arjunkeerthi/monocular_VO.git.

```
"""
Credit goes to Avi Singh: https://github.com/avisingh599/mono-vo.
The code below The code below builds of his C++ code, with modifications..

The MIT License

Copyright (c) 2015 Avi Singh

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
"""

import numpy as np
import cv2
import math
import time
from matplotlib import pyplot as plt

cap = cv2.VideoCapture(0)
orb = cv2.ORB_create(nfeatures=3000)
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
enable_nms = True

# Change sequence value to run on different image sets.
# Report used 0, 2, 5, and 9.
sequence = 0

# Size of trajectory plot for the sequences tested (trial and error
# to figure out the right size so the trajectory doesn't go outside the
# plot).
trajectory_size_dict = {
    0: (600, 800, 3),
    2: (1200, 800, 3),
    5: (450, 800, 3),
    9: (800, 800, 3)
}
```

```python
# Offset from bottom left corner of trajectory plot to call (0,0).
# Each is provided as (x offset, y offset).
trajectory_start_dict = {
    0: (400, 100),
    2: (100, 100),
    5: (400, 150),
    9: (250, 200)
}

# Number of images to read from each sequence.
sequence_num_images_dict = {
    0: 4000,
    2: 4000,
    5: 2500,
    9: 1500
}

# Path to directory that contains the data_odometry_poses and
# data_odometry_color folders from the Kitti Benchmark data set.
# <add the correct path here for the computer being used>
path = ""

# Feature detectors to use.
fast = cv2.FastFeatureDetector_create(threshold=20, nonmaxSuppression=True)
gftt = cv2.GFTTDetector_create(maxCorners=3000)
agast = cv2.AgastFeatureDetector_create()

# Color for each detector's path on trajectory plot.
detector_color_dict = {
    "FAST": (102, 252, 255),
    "GFTT": (0, 0, 255),
    "AGAST": (255, 179, 102)
}


# Read lines from ground truth pose text file and return as array.
# Format for each line of file is the 3x4 projection matrix flattened into
# a list of 12 numbers (projection matrix is [R | t], where R is 3x3 rotation
# matrix and t is 3x1 translation vector. After testing this out, it appears
# that each line gives these matrices with respect to the starting pose (no need
# to chain every prior together to get current position). So, first 3 numbers is
# first row of R, 4th number is 1st element of t. 5th-7th give second row of R,
# 8th gives second element of t. And finally 9th-11th give third row of R, 12th
# gives third element of t.
def getPoses():
    def parse_line(line):
        parts = line.split(' ')
        return list(map(lambda p: float(p), parts))

    with open(f"{path}\\data_odometry_poses\\dataset\\poses\\"
              f"{sequence:02}.txt", 'r') as f:
        lines = f.readlines()
        return list(map(parse_line, lines))
```

```python
# Compute distance between two poses to use as scale (as translation from
# recoverPose() should be of unit length).
def getAbsoluteScale(pose1, pose2):
    x1 = pose1[3]
    y1 = pose1[7]
    z1 = pose1[11]
    x2 = pose2[3]
    y2 = pose2[7]
    z2 = pose2[11]
    return math.sqrt((x1-x2)**2 + (y1-y2)**2 + (z1-z2)**2)


# Extract rotation and translation matrices from ground truth pose.
def getTransformation(pose):
    rotation = np.array([pose[:3], pose[4:7], pose[8:11]])
    translation = np.array([[pose[3]], [pose[7]], [pose[11]]])
    return rotation, translation


# Extract FAST features from image and convert to a numpy array of points.
# Returned array has shape Nx2, where N is the number of features. x is first col,
# y is second col.
def featureDetection(img, type="FAST"):
    keypoints = None
    if type == "FAST":
        keypoints = fast.detect(img)
    elif type == "GFTT":
        keypoints = gftt.detect(img)
    elif type == "AGAST":
        keypoints = agast.detect(img)

    return cv2.KeyPoint_convert(keypoints)


# Track features provided in points2 from img1 to img2, removing the features
# from points1 and points2 that are not good.
def featureTracking(img1, img2, points1):
    lk_params = dict(winSize  = (21,21),
                     maxLevel = 3,
                     criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_COUNT,
                                 30, 0.001))

    points2, status, err = cv2.calcOpticalFlowPyrLK(img1, img2, points1, None,
                                                    **lk_params)

    # Mask to retain features with status of 1 and have nonnegative x and y
    # position.
    status_mask = np.squeeze(status == 1)
    points1_mask = (points1[:, 0] >= 0) & (points1[:, 1] >= 0)
    points2_mask = (points2[:, 0] >= 0) & (points2[:, 1] >= 0)
    keep_mask = status_mask & points1_mask & points2_mask

    points1 = points1[keep_mask, :]
    points2 = points2[keep_mask, :]

    return points1, points2
```

```python
# Get image from Kitti odometry dataset with provided index and from left rgb camera.
# Return both colored and grayscale images.
def getImage(index):
    img_path = f"{path}\\data_odometry_color\\dataset\\sequences\\" \
        f"{sequence:02}\\image_2\\{index:06}.png"
    img = cv2.imread(img_path)
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    return img, img_gray


# Run main odometry on Kitti images.
def main():
    # Get all the ground truth poses
    poses = getPoses()

    # Detector type.
    # Change type here for each run.
    # Available types: "FAST", "GFTT", "AGAST".
    type = "FAST"

    # Get first two images
    index = 0
    img1_c, img1 = getImage(index)
    index += 1
    img2_c, img2 = getImage(index)
    index += 1

    # Detect features in img1 and track to img2
    points1 = featureDetection(img1, type)
    points1, points2 = featureTracking(img1, img2, points1)

    # Camera matrix extracted from calib.txt for sequence 00.
    # Format of this file found from this answer:
    # https://stackoverflow.com/a/50211379.
    # To summarize: 5 lines, first four define projection matrix for each of
    # the 4 cameras: 1st - left grayscale, 2nd - right grayscale, 3rd - left rgb,
    # 4th - right rgb. I am using the left camera images, so I'm using the 3rd
    # camera. Each line 12 numbers, defining 3x4 projection matrix flattened:
    # [C | t], where C is the camera matrix. So I just copied those numbers over
    # to here.
    camera_matrix = np.array([
        [7.18856e+02, 0.0, 6.071928e+02],
        [0.0, 7.18856e+02, 1.852157e+02],
        [0.0, 0.0, 1.0]
    ])

    # Compute essential matrix between first two images and then recover rotation
    # and translation matrices between the two images.
    E, mask = cv2.findEssentialMat(points2, points1, camera_matrix, cv2.RANSAC,
                                   0.999)
    retval, R, t, mask = cv2.recoverPose(E, points2, points1, camera_matrix)

    prev_img = img2
    prev_features = points2
```

```
R_f = R
t_f = t

# Will store plot of vehicle's trajectory.
# Use first line when running on a sequence for the first time to initialize
# trajectory image.
# Use second line to load trajectory image and draw current feature's trajectory
# over it.
trajectory_size = trajectory_size_dict[sequence]
start_offset = trajectory_start_dict[sequence]
trajectory = np.ones(trajectory_size, dtype=np.uint8)  # Equivalent to cv2.CV_8UC3
# trajectory = cv2.imread(f"trajectory{sequence:02}.png")

cv2.namedWindow("Road facing camera")
cv2.namedWindow("Trajectory")

total_error = 0
start = time.time()
num_images = sequence_num_images_dict[sequence]
for i in range(2, num_images+2):
    # Get ground truth rotation and translation.
    R_gt, t_gt = getTransformation(poses[index])

    # Get current image.
    curr_img_c, curr_img = getImage(index)
    index += 1

    # Track features from prev to curr image.
    prev_features, curr_features = featureTracking(prev_img, curr_img,
                                                   prev_features)

    # Compute essential matrix and compute rotation and translation from prev
    # to curr.
    E, mask = cv2.findEssentialMat(curr_features, prev_features, camera_matrix,
                                   cv2.RANSAC, 0.999)
    retval, R, t, mask = cv2.recoverPose(E, curr_features, prev_features,
                                         camera_matrix)

    # Calculate scale and compute the new translation and rotation.
    scale = getAbsoluteScale(poses[index-1], poses[index])
    if scale > 0.1 and t[2] > t[0] and t[2] > t[1]:
        t_f = t_f + scale * (R_f @ t)
        R_f = R_f @ R

    # If the number of features being tracked drops below 2000, re-detect
    # features.
    if len(prev_features) < 2000:
        prev_features = featureDetection(prev_img, type)
        prev_features, curr_features = featureTracking(prev_img, curr_img,
                                                       prev_features)

    prev_img = curr_img
    prev_features = curr_features

    total_error += np.linalg.norm(t_f - t_gt)
```

```
        # Compute current location.
        # Admittedly this part confuses me a little. It looks like t_f, R_f are
        # the matrices from starting point to current point. t_f is the translation
        # from start to current, so I think that's why we index this directly to find
        # (x,y) coordinate. Orientation doesn't matter for plot, so we don't consider
        # R here (if we needed a bearing for the car, then we probably would have to
        # use R as well).
        x = int(t_f[0][0]) + start_offset[0]
        y = trajectory_size[0] - (int(t_f[2][0]) + start_offset[1])
        cv2.circle(trajectory, (x,y), 1, detector_color_dict[type], 2)

        # Ground truth location.
        # Uncomment this line after the first run on a sequence to avoid re-drawing
        # the ground truth poses every time.
        x_gt = int(t_gt[0][0]) + start_offset[0]
        y_gt = trajectory_size[0] - (int(t_gt[2][0]) + start_offset[1])
        cv2.circle(trajectory, (x_gt,y_gt), 1, (0,255,0), 2)

        # Display text for current position.
        # Uncomment to see position for debugging purposes.
        # cv2.rectangle(trajectory, (10,30), (550,50), (0,0,0), cv2.FILLED)
        # text = f"Coordinates: x = {t_f[0][0]:.4f}m, y = {t_f[1][0]:.4f}m, " \
        #     f"z = {t_f[2][0]:.4f}m"
        # cv2.putText(trajectory, text, (10,50), cv2.FONT_HERSHEY_PLAIN, 1,
        #             (255,255,255), 1, cv2.LINE_8)

        cv2.imshow("Road facing camera", curr_img_c)
        cv2.imshow("Trajectory", trajectory)
        cv2.waitKey(1)

    end = time.time()
    cv2.imwrite(f"trajectory{sequence:02}.png", trajectory)
    cv2.destroyAllWindows()
    print(f"Total error for {type}: {total_error:.4f}")
    print(f"Total time: {end-start} seconds, # of frames: {num_images}")
    print(f"Average fps: {num_images/(end-start):.4f}")

    fig, ax = plt.subplots()
    extent = (-start_offset[0], trajectory_size[1]-start_offset[0],
              -start_offset[1], trajectory_size[0]-start_offset[1])
    rgb_trajectory = cv2.cvtColor(trajectory, cv2.COLOR_BGR2RGB)
    im = ax.imshow(rgb_trajectory, origin='upper', extent=extent)
    plt.show()


if __name__ == "__main__":
    main()
```