Visual SLAM using ORB-SLAM2 with Path Finding*

Runlin Guo[1], Zhengfeng Lai[1] and Wenda Xu[1]

***Abstract*—This paper presents a Visual SLAM approach, attempting to integrate ORB-SLAM2 [1] system with path finding algorithm to achieve real-time autonomous navigation on a self-driving car. To diminish the disturbance of moving objects to camera pose estimation, we introduced a ORB feature refinement step using Mask R-CNN in the ORB-SLAM2 pipeline to remove the ORB features that belong to moving objects. For this purpose, we employed a Python-to-C++ code wrapper for ORB-SLAM2. We evaluated the Python wrapper and our feature refinement approach using the 11 sequences of KITTI dataset on a standard desktop computer. Our approach achieves mostly better performance compared to the original ORB-SLAM2 system in pose estimation accuracy. We also attempted to integrate ORB-SLAM2 system onto a radio controlled car using two RGB cameras and one Nvidia Jetson TX2 module. We performed an evaluation using KITTI dataset and compared the results with desktop performance.**

***Index Terms*—Visual odometry, Simultaneous localization and mapping, Real-time systems, Deep learning, Path planning**

I. Introduction

The Simultaneous Localization and Mapping (SLAM) problem is a computational problem of estimating the robot's poses and constructing the map of an unknown environment at the same time. It is the basis for most autonomous navigation systems. Visual SLAM is a type of SLAM where the observation measurements are obtained mainly using camera sensors. Its advantages include

- Cheap and easy to use camera sensors, contrast to LiDAR sensors.
- Not affected by wheel slip in uneven terrain, contract to classical wheel odometry.

Visual SLAM methods can be categorized into *feature-based* or *direct* based on tracking method and *monocular*, *stereo*, or *RGB-D* based on type of camera sensors used. *Feature-based* visual SLAM methods add an additional step to extract and match feature points before tracking while *direct* methods use the full image to track. A comparison of these two methods are summarized in Table I.

To solve the problem, we used the state-of-the-art ORB-SLAM2 system. It has the following advantages:

- Complete *feature-based* visual SLAM system.
- Works with monocular, stereo and RGB-D camera sensors.
- Operates in real time on standard CPUs, in small and large, indoor and outdoor environments.
- Allows wide baseline loop closing and relocalization.

*This work was supported by Nvidia, XMotors.ai and UC Davis ECE department.

[1]The authors are all with the Electrical and Computer Engineering Department at University of California - Davis, 1 Shields Ave, Davis, CA, USA. kolguo@ucdavis.edu, lzhengfeng@ucdavis.edu, wedxu@ucdavis.edu

TABLE I
Comparison of Feature-Based Method and Direct Method [2]

Feature-Based	Direct
Can only use & reconstruct corners	Can use & reconstruct whole image
Generally faster	Generally slower (but good for parallelism)
Flexible: outliers can be removed retroactively	Inflexible: difficult to remove outliers retroactively
Robust to inconsistencies in the model/system (rolling shutter)	Not robust to inconsistencies in the model/system (rolling shutter)
Decisions (KP detection) based on less complete information	Decision (linearization point) based on more complete information
No need for good initialization	Needs good initialization
∼20+ years of intensive research	∼4 years of research

- Includes full automatic initialization.
- Has excellent robustness and generates a compact and trackable map that grows if the scene content changes, allowing life-long operation.

Therefore, ORB-SLAM achieves unprecedented performance with respect to other state-of-the-art visual SLAM approaches. This is the main motivation on investigating this specific algorithm.

However, when generating the ORB features, the ORB-SLAM2 system does not differentiate between moving objects (such as cars and pedestrian) and stationary objects (such as roads and landmarks). Incorporating features belonging to moving objects into tracking misleads the system about camera translations and rotations. This reduces the accuracy of camera pose estimation.

Our goal is to implement a visual SLAM system based on ORB-SLAM2 and incorporate an additional feature refinement step using Mask R-CNN. Also, we want to incorporate path finding algorithm to achieve auto navigation. In the end, we want to integrate ORB-SLAM2 system onto RC car using two cameras and a Nvidia Jetson TX2 module to test in real-world environment.

Our feature refinement method is described in Section III and the path finding approach is described in Section IV. Our evaluation methods and results are described in Section V.

II. Related Work

A. Monocular SLAM

Monocular SLAM was based on filter initially: every frame of video was processed by the filter to calculate the map feature locations as well as camera pose. There exists a significant drawback in this methods: the computation over processing consecutive frames wastes a wealth of sources but gains little new information over frames while also resulting

in the huge accumulation of linearization errors. Later on, keyframe-based approaches use only selected frames. The famous system, PTAM by Klein and Murray [3], successfully splits camera tracking and mapping in parallel threads for real time in small environments for the first time. But the points are only useful for tracking rather than for place recognition, making it hard operate in detecting large loops and yielding a low invariance to viewpoint.

Strasdat et. al [4] presented a large scale monocular SLAM system with a front-end based on optical flow implemented on a GPU, followed by FAST feature matching and motion-only BA, and a back-end based on sliding-window bundle adjustment (BA).

Pirker et. al [5] proposed CD-SLAM, a very complete system including loop closing, relocalization, large scale operation and efforts to work on dynamic environments.

The visual odometry of Song et. al [6] comes up ORB features for tracing and a temporal sliding window BA back-end. So it needs to reuse the map for global relocalization and loop detection, resulting delay for real-time detection.

The semi-direct visual odometry SVO of Forster et. al [7] is a halfway between direct and feature based methods. Without requiring to extract features in every frame they are able to operate at high frame-rates obtaining impressive results in quadcopters.

The recent work known as LSD-SLAM using monocular camera by Engel et. al [8] can set up large-scale semi-dense maps based on direct methods rather than bundle adjustment over features, This system is able to work out in real time and construct a semi-dense map but without GPU acceleration. However, the accuracy of camera localization seems relatively low and it still requires features for loop detection. Later, a version of LSD-SLAM using stereo cameras [9] also appeared.

B. Stereo and RGB-D SLAM

ORB-SLAM by Mur-Artal et. al [10] performs well in several aspects. It utilizes the same features for tracking, mapping, relocalization and loop closing simultaneously. So this system is highly efficient and reliable. And it uses ORB features, which allow real-time operation without GPUs, for real time operation in large environments, such as loop closing based on the optimization of a pose graph, camera relocalization and mapping points. Later, the ORB-SLAM2 using monocular, stereo and RGB-D cameras [1] also appeared.

The Computer Vision Group of the Technical University of Munich also develops a Visual-SLAM algorithm named Direct Sparse Odometry (DSO) [11]. DSO is a novel direct and sparse formulation for Visual Odometry and does not depend on keypoint detectors or descriptors. It combines a fully direct probabilistic model (minimizing a photometric error) with consistent, joint optimization of all model parameters, including geometry - represented as inverse depth in a reference frame - and camera motion. This is achieved in real time by omitting the smoothness prior used in other direct methods and instead sampling pixels evenly throughout the images. Since then, more variants of the DSO-SLAM method have been published: Stereo DSO [12] in which stereo cameras are used, Visual-Inertial DSO [13] in which an inertial measurement unit (IMU) is used, DSO with Loop Closure (LSDO) [14] and Deep Virtual Stereo Odometry (DVSO) [15] in which a deep monocular depth prediction using a novel deep neural network is performed to overcome limitations of geometry-based monocular visual odometry.

III. Mask R-CNN Feature Refinement

Zhengfeng Lai, Runlin Guo

In this section, we are going to apply Mask R-CNN to the original ORB-SLAM2 system for solving the "relative motion" challenge. Since the open-sourced ORB-SLAM2 system is in C++11, to apply Mask R-CNN in Python, we need a Python wrapper for ORB-SLAM2 system. The Python wrapper is described in Section III-A and the actual feature refinement approach is described in Section III-B.

A. Python Wrapper Overview

We integrated the Python wrapper for ORB-SLAM2 system by John Skinner [16] into our system environment. The Python wrapper uses *Boost.Python* library which enables seamless interoperability between C++ and Python and supports

- Automatic cross-module type conversions
- Efficient function overloading
- Manipulating Python objects in C++
- Exporting C++ iterators as Python iterators

The execution flow of our system with the Python wrapper is as follows:

- Python wrapper loads stereo images using `opencv2.imread()`.
- The wrapper passes the stereo images to C++11.
- Tracking timestamps are measured and recorded in the Python wrapper.
- The wrapper gets camera poses trajectory from C++11 for evaluation.

The evaluation of this Python wrapper is provided in Section V-B.

B. Feature Refinement

Imaging the car is waiting on an intersection but there are several moving objects around the car, ORB-SLAM2 system will simulate this case and generate the results that the car is moving while obviously the car is waiting there, which is called "relative motion". In order to diminish the disturbance of moving objects to camera pose estimation, we designed an ORB feature refinement step using Mask R-CNN in the ORB-SLAM2 pipeline to abandon the ORB features that belong to moving objects.

In the tracking thread, the ORB-SLAM2 system is going to extract the feature points among every frame. However, considering the effect of "relative motion", the feature points extracted from moving contents definitely have negative effect on the tracking thread because these points may lead the system to make wrong estimations on camera estimation and

lower the accuracy. Hence our group decided to remove these points, which means we are saving useful feature points for tracking rather than those negative-effect points.

In order to get these points, we applied Mask R-CNN to segment every moving content of each frame of the video. After this step, we can get the area which has moving contents, as Fig. 1. After using Mask R-CNN, we can obtain the segmentation of all objects in the frame. The following step is to find the areas of moving contents. Obviously during driving, the moving contents around the car can only be pedestrians, cars and bicycles. So we set the label that if the objects detected by Mask R-CNN are in the above categories, the objects are moving contents. And we will abandon all feature points in these moving areas, which indicates that once the extracted feature points by tracking thread are in these areas, we will remove them out for the following threads.

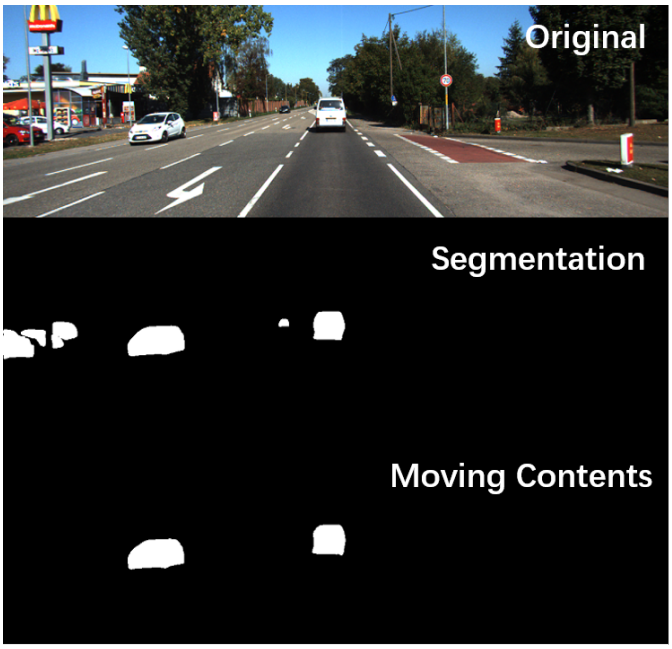

Fig. 1. Remove Moving Contents

The evaluation of our feature refinement approach is provided in Section V-C.

IV. Path Finding

Wenda Xu

In this section, we did researches on several popular path finding algorithms and tried to find an optimal cost function specifically applied to Visual SLAM.

A. Main Algorithm comparisons:

Dijkstra, greedy first search and A* algorithm are three main techniques, which are commonly used for path finding. Dijkstra's algorithm works by repeatedly examines the closet not-yet-examined vertex, adding its vertices to the set of vertices to be examined. Based on the nature of this algorithm, it is guaranteed to find the shortest path with all the positive edges. However, its speed is not optimized. The Greedy Best-First-Search algorithm on the contrary has the heuristic of how far from the goal any vertex is. Instead of selecting the vertex closet to the starting point, it selects the vertex closet to the goal. Greedy-Best-First search is not guaranteed to find a shortest path, but it is much quicker compared to Dijkstra's Algorithm because of its heuristic function. In the end, A* Algorithm uses both the exact cost of the path from the starting point to any vertex n, and heuristic estimated cost from vertex n to the goal to represent cost function. Therefore, based on the nature and better performance of A* algorithm, we use A* algorithm as our baseline model to further investigate on optimizing its cost function corresponding to Visual SLAM application.

B. Goal and Algorithm intuitions:

Our goal is trying to minimize the reconstruction uncertainty of observed geometry and the distance traveled by the sensor between image locations. However, specific visual SLAM challenge is lack of ground truth data. we need to evaluate the statistical uncertainty in order to estimate reconstruction quality [17] (Gaussian distribution of the data points). Based on referencing paper "*Optimal View Path Planning for Visual SLAM*", we have the intuition (Greedy approach: Solve a local minimal problem to global minimal) [17] and specific algorithm (Levenberg-Marquardt method) to develop cost function. General idea is to evaluate cost function before any observations are made and to predict the camera view at a particular location based on the current data distribution.

C. Specific algorithm works as follows:

The following descriptions of the algorithm is obtained from the paper "*Optimal View Path Planning for Visual SLAM*" [17].

Step1: Given initial data points distribution, calculate centroid and assign this to be the interest point of camera. Select a target location for the camera, i.e, select the end point of the path.

Step2: Use linear interpolation to generate a initial path between the first and last camera locations. Use the image sampling rate and speed of the robot camera to verify results.

Step3: Find a minimum of the cost function using LM (Levenberg-Marquardt) method.

Step4: Move the camera to the next location along the path and make an actual observation. Update the camera interest point and path end point.

Repeat Step 3 and Step 4, each time with one less camera location, update the initial guess using the previous path estimation. In the end, a single path will be converged and the algorithm will obtain the minimum distance between optimized image locations.

D. Proposed cost function:

$$C(P,X) = \frac{1}{N} \times tr(\sum P,X) + \frac{\alpha}{(M-1)^{1-q}} \sum_{j=1}^{M-1} ||P_{pos}^{j+1} - P_{pos}^{j}||^{q} + \beta H(n)^{*} \quad (1)$$

The first two terms came from the proposed cost function [17] from paper "*Optimal View Path Planning for Visual SLAM*", the later term H(n) is the heuristic function came from the A* algorithm, which measures the cost from vertex n to the goal. Beta and alpha are parameters required to tune for both functions. The first term measures the camera position path along the path within the estimated data points distributions. Original paper proposed to use functions of the eigenvalues of co-variance matrix to solve condensing probability distribution problem. Since eigenvalues have a direct geometric interpretation [17], the eigenvalues of each block can correspond to the variance of the feature location, calculating the trace of co-variance of matrix to minimize the sum of the eigenvalues. In another words, it minimizes the total reconstructed uncertainty of converging to the optimal path.

Difficulties and Future Notice

1) Key difficulty is to tune the hyper parameters Alpha and Beta [17]. Alpha is the weight which controls camera path between data points. If Alpha is setting too high, the path will be less easy to modified, optimal path will be away from the data points. The first term, will increase as uncertainty of each data point increases. If alpha value was set too low, path will tend to get closer to the data points. In that case, uncertainty of each data points can get dramatically decreased. However, the overall path is deviated a lot. Therefore, Alpha is important in determine path shape and minimizing total costs. At the same time, Beta is also an important factor. If Beta was set too high, then algorithm will not guaranteed to find a shortest path but speed is fast. On the contrary, if Beta is too small, algorithm is guaranteed to find the shortest path. However, more nodes will be expanded and make the running time much slower.
2) The other difficulty is to cop-orate with Visual-SLAM based map points. Using data points alone from the Visual SLAM can't cover the entire data point that we needed, such as some data points of the obstacles. We need to develop a more comprehensive methods to collect all the data points that needed in path finding.

V. Evaluation

Runlin Guo, Zhengfeng Lai

We have evaluated the original ORB-SLAM2 system, the Python wrapper, our feature refinement approach and the ORB-SLAM2 system on Jetson TX2 using the 11 sequences of KITTI dataset [18] with ground truth poses. There are three evaluating metrics:

- $tran$ - Average relative camera pose translation (movement) error [%]
- rot - Average relative camera pose rotation error [degree/100m]
- $time$ - Average tracking time [second]

The camera pose translation error and rotation error will determine the accuracy of the SLAM system while the tracking time will determine whether the system can achieve real-time tracking performance.

Our evaluation is performed on a desktop computer with an Intel Core i7-7820X, an NVIDIA GeForce GTX 1080 Ti and 32 GB of system RAM.

A. Original ORB-SLAM2 C++11 System

The evaluation results below are the average results obtained by running 10 iterations of the 11 sequences of KITTI dataset.

- Translation Error:
  A boxplot of the translation error is shown in Fig. 2. As we can see, sequence 01 has relatively significant translation error. This is because it is the only highway sequence among the 11 sequences. Therefore, there are few close trackable points, making it harder to estimate camera translation. A frame with colored trackable points from sequence 01 is shown in Fig. 3.

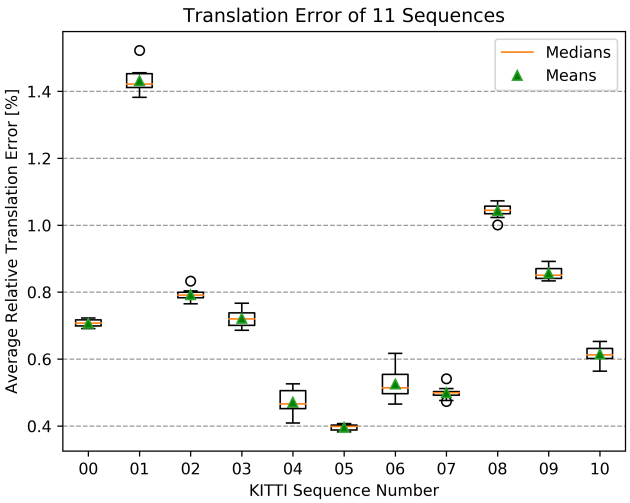

Fig. 2. C++11 Translation Error

- Rotation Error:
  A boxplot of the rotation error is shown in Fig. 4. As we can see, sequence 08 has relatively significant rotation error. This is because it has repeated paths which do not form loops, plenty of sharp turns and few distant trackable points, making it harder to estimate camera rotation. The trajectory of this sequence is shown in Fig. 5.

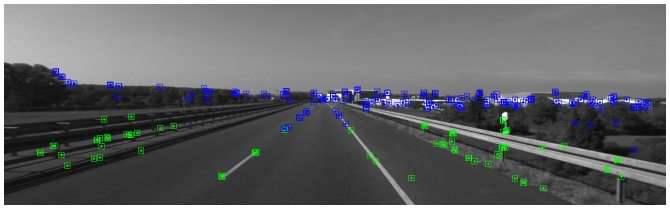

Fig. 3. Tracked Points in KITTI Sequence 01. Green points are close trackable points while blue points are far trackable points.

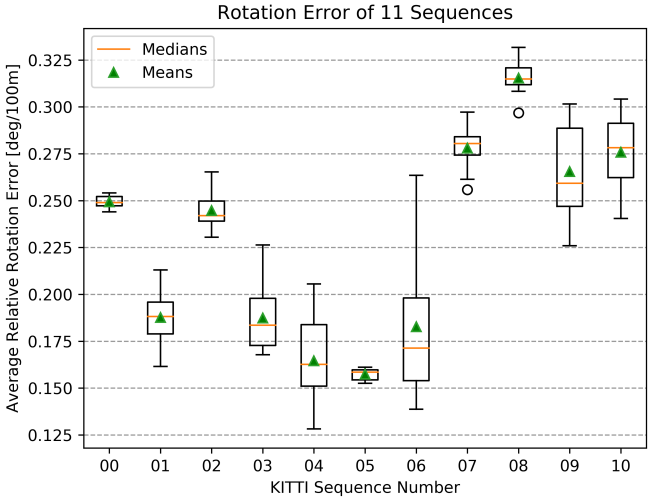

Fig. 4. C++11 Rotation Error

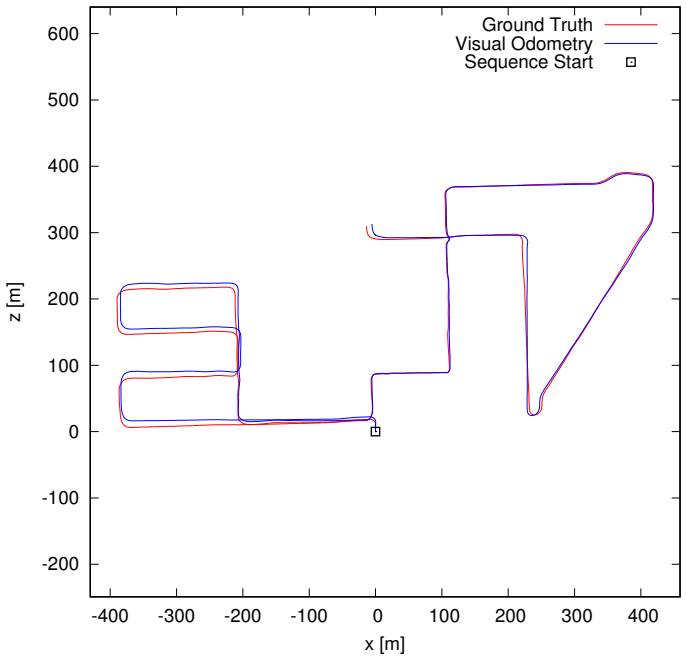

Fig. 5. Trajectory Path of KITTI Sequence 08. Red path is ground truth trajectory while blue path is the estimated trajectory.

- Mean Tracking Time:
  A boxplot of the mean tracking time is shown in Fig. 6. As we can see, sequence 01 has relatively significant mean tracking time. This is because in this highway sequence, the car is moving at high speed and the camera captures at low frame rate (only 10 fps). Therefore, it is important to insert keyframes often enough so that the amount of close points allows for accurate translation estimation and no tracking lost occurs. The increased frequency of keyframe insertion requires more computing power, resulting in longer tracking time.
  Also, since the camera frame rate is 10 Hz, the frame time is $\frac{1}{10} = 0.1$ second. Because all 11 sequences result in less mean tracking time than 0.1 second, the system is operating in real-time.

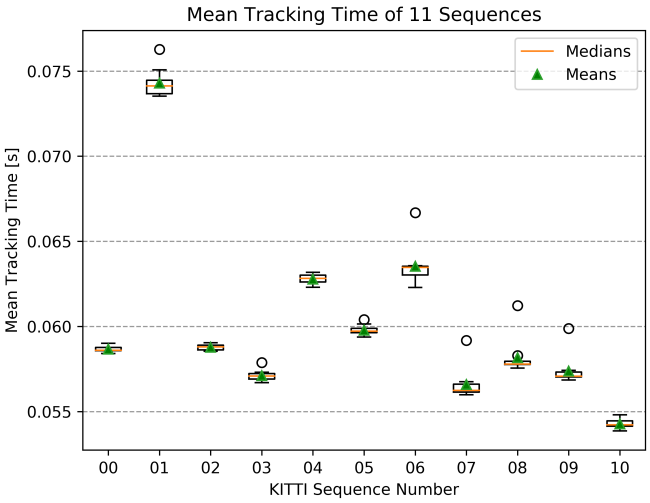

Fig. 6. C++11 Mean Tracking Time

In summary, we concluded that the ORB-SLAM2 system

- Generates **larger translation error** under **high speed and low frame rate**. This can be further illustrated by Fig. 7 which is a plot of translation and rotation error versus speed. As we can see, the translation error increases rapidly when the speed exceeds 60 km/h.
- Generates **larger translation error** with **few close** trackable points.
- Generates **larger rotation error** with **few distant** trackable points.

B. Python Wrapper

The evaluation results below are the average results obtained by running 10 iterations of the 11 sequences of KITTI dataset. The results are summarized in Table II.

- Translation Error:
  As we can see from the *trans* column, the C++11 and Python wrapper have comparable translation error. If we look at the average result, the Python wrapper

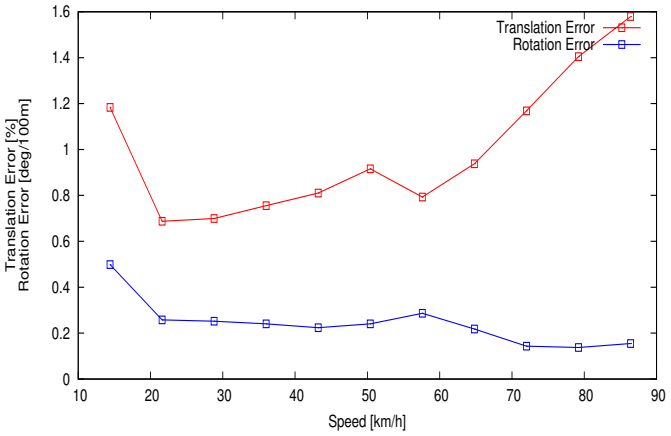

Fig. 7. Average Translation and Rotation Error vs. Speed

even performs slightly better: smaller mean and standard deviation.

- Rotation Error:
  As we can see from the rot column, the C++11 and Python wrapper have comparable rotation error. If we look at the average result, the results of Python wrapper have smaller standard deviation.
  Particularly, if we look at the highway sequence 01, the Python wrapper has much worse mean error: 0.226 compared to only 0.188 of C++11 system. Thus, the Python wrapper seems to suffer more under high speed and low frame rate situation.
- Mean Tracking Time:
  As we can see from the $time$ column, the Python wrapper has generally longer tracking time. This is probably due to the Python library overhead.

In summary, compared with the C++11 system, we concluded that the Python wrapper has

- Almost the same translation error.
- Generally smaller variances but comparable means in rotation error.
  - Under high speed and low frame rate, Python wrapper performs worse.
- Generally longer tracking time.

Overall, there is not much influence on tracking accuracy by integrating the Python wrapper. Therefore, it can be used to apply Mask R-CNN for feature refinement.

C. Mask R-CNN Feature Refinement

Firstly, we ran our model on sequence 09 of KITTI dataset, and the result is shown as Fig. 8. The green line denotes the ORB-SLAM2 system with Mask R-CNN while the red line denotes the original ORB-SLAM2 without Mask R-CNN. The

TABLE II
PYTHON WRAPPER VS. C++11

Sequence #	$trans$ [%]		rot [$deg/100m$]		$time$ [sec]	
	Mean	***Std dev***	***Mean***	***Std dev***	***Mean***	***Std dev***
00 C++	0.707	0.011	0.249	0.003	0.059	0.0002
00 Py	0.708	0.010	0.254	0.007	0.060	0.0005
01 C++	1.432	0.038	0.188	0.017	0.074	0.0008
01 Py	1.409	0.027	0.226	0.018	0.075	0.0011
02 C++	0.792	0.018	0.245	0.009	0.059	0.0002
02 Py	0.773	0.019	0.241	0.007	0.060	0.0005
03 C++	0.721	0.024	0.187	0.018	0.057	0.0003
03 Py	0.719	0.027	0.174	0.012	0.058	0.0004
04 C++	0.471	0.037	0.165	0.024	0.063	0.0003
04 Py	0.462	0.030	0.159	0.022	0.064	0.0006
05 C++	0.396	0.009	0.157	0.003	0.060	0.0003
05 Py	0.409	0.018	0.165	0.008	0.061	0.0005
06 C++	0.525	0.043	0.183	0.037	0.064	0.0011
06 Py	0.533	0.018	0.173	0.018	0.064	0.0006
07 C++	0.500	0.018	0.278	0.011	0.057	0.0009
07 Py	0.518	0.059	0.283	0.023	0.057	0.0005
08 C++	1.043	0.020	0.315	0.009	0.058	0.0010
08 Py	1.031	0.018	0.307	0.005	0.059	0.0005
09 C++	0.857	0.018	0.266	0.024	0.057	0.0009
09 Py	0.842	0.027	0.240	0.013	0.058	0.0005
10 C++	0.615	0.024	0.276	0.019	0.054	0.0003
10 Py	0.617	0.027	0.282	0.026	0.055	0.0005
Avg C++	0.733	0.287	0.228	0.055	0.060	0.0052
Avg Py	0.729	0.277	0.228	0.053	0.061	0.0051

x-axis indicates the length that car has covered while the y-axis indicates the translation error. According to Fig. 8, we can find that generally the system with Mask R-CNN has lower translation error, denoting it has better achievements especially when the car has run over 300 meters. After 300 meters, the translation error is considerably reduced.

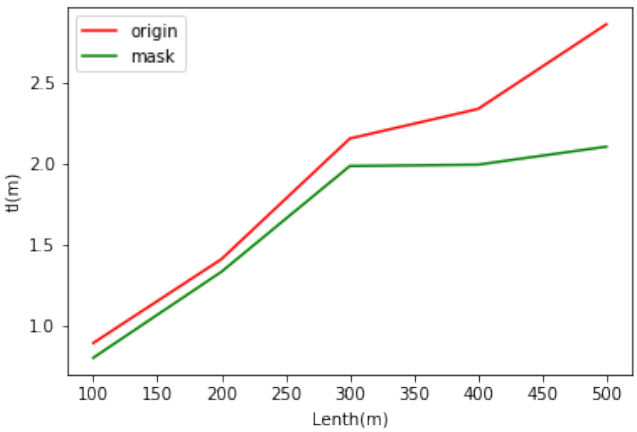

Fig. 8. With VS. Without Mask R-CNN

Later on, we also tested the 11 sequences of KITTI dataset. The performance of the original ORB-SLAM2 and Mask R-CNN combined model is shown in Fig. 9. We can obtain the fact that ORB-SLAM2 with Mask R-CNN in most sequences achieves lower error in both rotation and movement error.

Sequence	Average Rotation Error(°/m)		Average Movement Error(%)	
	Without MS-R	With MS-R	Without MS-R	With MS-R
00	0.002843	0.003015	0.842988	0.893913
01	0.002335	0.002141	1.405338	1.395963
02	0.002815	0.002708	0.81265	0.8079
03	0.001593	0.001547	0.69516	0.69158
04	0.001299	0.002005	0.448333	0.355467
05	0.002206	0.002399	0.566313	0.609663
06	0.002428	0.002399	0.844875	0.823425
07	0.004651	0.003915	0.970867	0.812217
08	0.002994	0.002972	1.01205	1.010413
09	0.002543	0.002421	0.877275	0.83995
10	0.002414	0.002127	0.580788	0.541013
Percentage	27.3%	72.7%	18.2%	81.8%

Fig. 9. Performance on KITTI Sequence

After calculating the exact performance, we can get the results as Fig. 10. It is obvious that with Mask R-CNN, the system could achieve lower error and higher accuracy in both movement and rotation error.

Sequence	Average Rotation Error(°/m)		Average Movement Error(%)	
	Without MS-R	With MS-R	Without MS-R	With MS-R
Average	0.002556	0.002231	0.823331	0.788319
Improvement	13.1%		5.2%	

Fig. 10. Improvement of ORB SLAM2 with Mask R-CNN

Fig. 11. Mapping without Mask R-CNN

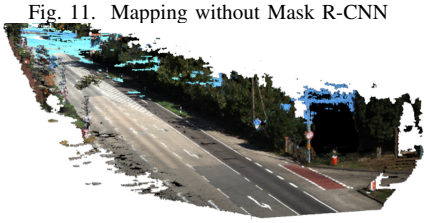

Fig. 12. Mapping with Mask R-CNN

For the mapping thread, due to the limited time, we only tested one sequence of KIITI dataset. And Fig. 11 and Fig. 12 visually and clearly show the differnt performance of ORB SLAM2 with and without Mask R-CNN. If we only use ORB-SLAM2 and all feature points extracted from moving contents are used for tracking thread, we can see that there are many shadows of moving cars in Fig. 11. And obviously the generated map is not ideal and not that clear. However, after using Mask R-CNN to move out the feature points from moving contents, the generated map is much clearer with higher quality.

D. ORB-SLAM2 on Jetson TX2

The NVIDIA Jetson TX2 module has 6 ARM64v8 CPU cores, 8 GB of system RAM, 32 GB of flash storage and a NVIDIA Pascal architecture GPU. After we installed the environment onto Jetson TX2 and solved several library conflicts, we are able to get the original ORB-SLAM2 system working. However, two of the KITTI dataset sequences failed to run on the Jetson due to RAM shortage. Table III shows some details about the 11 KITTI dataset sequences. The evaluation results in Table IV are the average results obtained by running 10 iterations of the 11 sequences of KITTI dataset.

TABLE III
DETAILS OF KITTI DATASET SEQUENCES*

Sequence #	# of Frames	# of Loops
00	4541	4
01	1101	0
02	4661	2
03	801	0
04	271	0
05	2761	3
06	1101	1
07	1101	1
08	4071	0
09	1591	0
10	1201	0

*Red: Failed to run, Green: Succeeded to run

In summary, compared with the desktop results, we concluded that the Jetson TX2 generates

- Almost the same translation error.
- Slightly worse rotation error.
- Generally much longer tracking time. Since the frame time of KITTI dataset is 0.1 second, all nine sequences cannot reach real-time operation on Jetson TX2 module.

Difficulties and Future Notice

When we were trying to integrate the ORB-SLAM2 system onto a radio controlled car using two RGB cameras and the Nvidia Jetson TX2 module, we encountered some environment setup issues and library incompatibilities. After reflashing the Jetson TX2 with the latest Jetpack 4.2 and NVIDIA Tegra Linux Driver Package (L4T) R32.1 and building the ORB-SLAM2 environment natively instead of using Docker, we are still unable to access the USB camera using GStreamer.
However, after further investigation, we found out that L4T Multimedia API sample application *"12_camera_v4l2_cuda"* is able to access the USB camera, record and save a video locally in *YUYV* raw format. Thus, for future notice, try to use the L4T Multimedia API to gain access to USB cameras. The documentation link is https://docs.nvidia.com/jetson/l4t-multimedia.

TABLE IV
PYTHON ORB-SLAM2 JETSON TX2 VS. DESKTOP

Sequence #	$trans\ [\%]$		$rot\ [deg/100m]$		$time\ [sec]$	
	Mean	***Std dev***	***Mean***	***Std dev***	***Mean***	***Std dev***
00 PC	0.708	0.010	0.254	0.007	0.060	0.0005
00 TX2	✗	✗	✗	✗	✗	✗
01 PC	1.409	0.027	0.226	0.018	0.075	0.0011
01 TX2	1.416	0.058	0.242	0.023	0.207	0.0021
02 PC	0.773	0.019	0.241	0.007	0.060	0.0005
02 TX2	✗	✗	✗	✗	✗	✗
03 PC	0.719	0.027	0.174	0.012	0.058	0.0004
03 TX2	0.747	0.028	0.201	0.017	0.176	0.0008
04 PC	0.462	0.030	0.159	0.022	0.064	0.0006
04 TX2	0.479	0.038	0.215	0.036	0.194	0.0005
05 PC	0.409	0.018	0.165	0.008	0.061	0.0005
05 TX2	0.395	0.013	0.158	0.005	0.191	0.0006
06 PC	0.533	0.018	0.173	0.018	0.064	0.0006
06 TX2	0.491	0.040	0.175	0.032	0.194	0.0013
07 PC	0.518	0.059	0.283	0.023	0.057	0.0005
07 TX2	0.504	0.028	0.283	0.015	0.183	0.0005
08 PC	1.031	0.018	0.307	0.005	0.059	0.0005
08 TX2	1.035	0.022	0.314	0.008	0.190	0.0009
09 PC	0.842	0.027	0.240	0.013	0.058	0.0005
09 TX2	0.840	0.026	0.248	0.011	0.186	0.0008
10 PC	0.617	0.027	0.282	0.026	0.055	0.0005
10 TX2	0.591	0.016	0.275	0.018	0.178	0.0007
Avg PC	0.729	0.277	0.228	0.053	0.061	0.0051
Avg TX2	0.722	0.314	0.235	0.053	0.189	0.0090

VI. CONCLUSIONS AND DISCUSSION

A. Conclusions

In this paper we have presented a visual SLAM approach based on ORB-SLAM2 with Mask R-CNN feature refinement. The evaluation results in Section V-C show that this approach will improve the tracking accuracy compared to the original ORB-SLAM2 system. However, our approach will not be able to operate in real-time on standard CPUs contrast to the original system. This is not a concern for modern self-driving cars which almost all are equipped with one or more GPUs.

For the NVIDIA Jetson TX2, the ORB-SLAM2 system does not perform well in terms of tracking time since it is mainly using the CPUs for computation and it only has 8 GB of system RAM. But if we use the GPUs to help with the computations and add more system RAM to it, the ORB-SLAM2 system will definitely perform better and reach the desktop performance.

B. Future Work

For future improvements, we can work on the following aspects:

- Finish integrating ORB-SLAM2 onto a RC car using USB cameras and Jetson TX2.
  The remaining steps are
  - gaining access to USB cameras using L4T Multimedia API,
  - calibrating the cameras using checkerboard images (the code is already written in Python),
  - modifying the current pipeline to accept images for tracking in run-time instead of reading local datasets,
  - setting up *yaml* files for frame timestamps using system run time.
- Instead of using Mask R-CNN neural network to remove ORB features that belong to moving objects, we can use classical digital image processing and computer vision approach without deep learning, such as background subtraction, frame differencing, temporal differencing, and optical flow. This will save us computational power by a lot.
- Since ORB-SLAM2 could not operate at high frame rate such as 60 fps in real-time with default settings of ORB extractor and will lose tracking under sharp and quick turns. GCNv2-SLAM [19] solves this using a deep learning-based network. It can run at around 80 Hz with mobile version of NVIDIA GTX 1070 and generates better distributed features compared with ORB. Since GCNv2-SLAM generates keypoints and binary descriptors similar to ORB features, it can be easily integrated into ORB-SLAM2 to provide more robust operation.
- Most of the current RGB-Depth cameras use active sensing ToF sensors which cannot work or perform poorly under sunlight. Also, they are pretty expensive compared to regular cameras. CNN-SLAM [20] solves this by using a monocular camera setup with depth prediction based on a deep learning network. CNN-SLAM can operate in real-time on standard GPUs and the CNN-predicted depth map can be fed into the RGB-D pipeline of ORB-SLAM2 to integrate into the ORB-SLAM2 system.
- Develop an integrated algorithm to combine SLAM-based map points to supposed algorithm (generate data points including obstacles). Find appropriate alpha and beta value for supposed cost function, benchmark with A* baseline model.

REFERENCES

[1] R. Mur-Artal and J. D. Tardós, "ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras," *CoRR*, vol. abs/1610.06475, 2016. [Online]. Available: http://arxiv.org/abs/1610.06475

[2] J. Engel. Semi-dense direct slam. Technical University Munich. [Online]. Available: http://wp.doc.ic.ac.uk/thefutureofslam/wp-content/uploads/sites/93/2015/12/ICCV-SLAM-Workshop_JakobEngel.pdf

[3] G. Klein and D. Murray, "Parallel tracking and mapping for small AR workspaces," *ISMAR. IEEE*, pp. 225–234, January 2007. [Online]. Available: http://www.robots.ox.ac.uk/~gk/publications/KleinMurray2007ISMAR.pdf

[4] H. Strasdat, J. M. M. Montiel, and A. J. Davison, "Scale drift-aware large scale monocular slam," *Robotics: Science and Systems (RSS)*, June 2010. [Online]. Available: http://roboticsproceedings.org/rss06/p10.pdf

[5] K. Pirker, M. Rther, and H. Bischof, "Cd slam - continuous localization and mapping in a dynamic world," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sep. 2011, pp. 3990–3997.

[6] S. Song, M. Chandraker, and C. C. Guest, "Parallel, real-time monocular visual odometry," in *2013 IEEE International Conference on Robotics and Automation*, May 2013, pp. 4698–4705.

[7] C. Forster, M. Pizzoli, and D. Scaramuzza, "Svo: Fast semi-direct monocular visual odometry," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, May 2014, pp. 15–22.

[8] J. Engel, T. Schöps, and D. Cremers, "LSD-SLAM: Large-scale direct monocular SLAM," in *European Conference on Computer Vision (ECCV)*, September 2014.

[9] J. Engel, J. Stueckler, and D. Cremers, "Large-scale direct slam with stereo cameras," in *International Conference on Intelligent Robots and Systems (IROS)*, September 2015.

[10] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardós, "ORB-SLAM: a versatile and accurate monocular SLAM system," *CoRR*, vol. abs/1502.00956, 2015. [Online]. Available: http://arxiv.org/abs/1502.00956

[11] J. Engel, V. Koltun, and D. Cremers, "Direct sparse odometry," in *arXiv:1607.02565*, July 2016.

[12] R. Wang, M. Schwörer, and D. Cremers, "Stereo dso: Large-scale direct sparse visual odometry with stereo cameras," in *International Conference on Computer Vision (ICCV)*, Venice, Italy, October 2017.

[13] L. von Stumberg, V. Usenko, and D. Cremers, "Direct sparse visual-inertial odometry using dynamic marginalization," in *International Conference on Robotics and Automation (ICRA)*, May 2018.

[14] X. Gao, R. Wang, N. Demmel, and D. Cremers, "Ldso: Direct sparse odometry with loop closure," in *iros*, October 2018.

[15] N. Yang, R. Wang, J. Stueckler, and D. Cremers, "Deep virtual stereo odometry: Leveraging deep depth prediction for monocular direct sparse odometry," in *eccv*, September 2018.

[16] J. Skinner, "ORB-SLAM2 PythonBindings," https://github.com/jskinn/ORB_SLAM2-PythonBindings, 2018.

[17] S. Haner and A. Heyden, "Optimal view path planning for visual slam," *Image Analysis Lecture Notes in Computer Science*, p. 370380, 2011.

[18] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[19] J. Tang, L. Ericson, J. Folkesson, and P. Jensfelt, "GCNv2: Efficient Correspondence Prediction for Real-Time SLAM," *CoRR*, vol. abs/1902.11046, 2019. [Online]. Available: http://arxiv.org/abs/1902.11046

[20] K. Tateno, F. Tombari, I. Laina, and N. Navab, "CNN-SLAM: real-time dense monocular SLAM with learned depth prediction," *CoRR*, vol. abs/1704.03489, 2017. [Online]. Available: http://arxiv.org/abs/1704.03489