SLAM for smart bikes

Omar Mohamed Anwar Mohamed Elkady
University of Twente
PO Box 217, 7500 AE Enschede
the Netherlands
omarmohamedanwarmohamedelkady@student.utwente.nl

Abstract
In an effort to utilize technological advances in the field of computer science, often state of the art technology finds its way into previously existing activities and tools. Such as the case with driving a normal car, now advancing into autonomous vehicles aided with sensors and cameras to ensure a comfortable and safe trip for the car passengers. Similarly, this study will look into how SLAM technology could lead to advancements in the realm of smart bicycles, specifically in terms of making cycling safer and more comfortable. This will all be done by evaluating different SLAM algorithms, in terms of usability on a bike and the performance on the provided hardware, hence giving a realistic result into the usability of SLAM for the smart bike domain. After that we pursued implementing a variation on SLAM that could potentially allow for lower resource usage on embedded systems.

The possibility of applying SLAM in a dynamic environment has been investigated in prior research articles[2]. Furthermore, the use of SLAM on autonomous bikes was investigated and shown to be feasible[1]. The potential applications of SLAM on smart bikes will be investigated in this study.

Keywords
SLAM, simultaneous localization and mapping, smart bikes, LIDAR, bike safety.

1. Introduction
SLAM (simultaneous localization and mapping) algorithms, deal with navigation in previously unknown areas by creating a map of that given environment and simultaneously localizing the agent (bike, car, robot, etc...). The algorithm makes use of different types of hardware, namely: cameras, LIDAR sensors, and laser range sensors among others, all to better detect the surrounding environment to build the agent a map.

SLAM algorithms are now being implemented on mobile robots as well as different types of autonomous vehicles. It is also being adopted on mass scale projects, which makes it an intriguing approach that is worthy of exploring in the implementation of smart bikes. However, since implementation on a smart bikes have not been done a lot and considering that smart bikes have some limitations in terms of budget and resources, this research aims to aid further implementations and research of SLAM on smart bikes by showing the possible use cases of the algorithm while also showing how the algorithms can run on lower resource systems such as a raspberry pi, and discuss the limitations that may come with such implementations.

The main research question is: How can SLAM algorithm be used to aid the development of smart bikes?

This simple research question will take into consideration a lot of aspects, and so because of that a few more research questions generically spawn such as:

- How can a SLAM algorithm be improved or modified to serve the purposes of smart bikes?
- Can a SLAM algorithm be used on a lower-end embedded system?

To find answers to such research questions some extensive research into existing literature and solutions was done, after that some of these solutions were tried and tested on datasets that show similar footage to what would be expected in a smart bike setting, the features and functionality of such algorithms was noted. Furthermore, an alternate solution to one of these tested implementations was designed to try and make the algorithm run on a lower-end embedded systems, and then it was tested to see if smooth performance on smart bikes can be obtained using such method.

2. Requirements
To try and answer such research questions, planning for the implementation was divided into two stages, each stage having its own requirements.

The first stage is just installing and building the SLAM algorithms that seemed to make sense in a smart bike context, this only requires the algorithms to run on the different datasets so that a frame of reference is obtained for when the improved algorithm is implemented. Some form of visual result must be obtained, such as seeing a map built from the dataset provided.

The second stage concerns enhancing at least one of the SLAM algorithms by implementing a version of SLAM that uses socket connection, making it work on two machines in parallel to save resources on one of the machines. The algorithm is supposed to show higher performance than the regular implementation. What that implies is that the implementation should decrease resources used on the raspberry pi’s end. The algorithm that was chosen for this is ORB-SLAM2, since a lot of previous research has been done on the algorithm and so previous literature provided ideas into how such an algorithm can be enhanced.

3. Existing solutions
After reading some research papers it had been made clear in previous studies that building and installing SLAM on some embedded systems is possible[3]. However, the

results vary from one system to the other, some high-end machines yield quick computation, like the Tegra X1, while some other systems were noticeably slower, such as the Panda-board ES.

Since there are many libraries and approaches that exist for the implementation of SLAM, some filtering process needed to be done, and the criteria was simply to find algorithms that can be easily incorporated with a camera to test in real time. Another criteria was to try to find algorithms that would also allow easy incorporation of datasets. That being said, the two algorithms we chose to implement were ORBSLAM2[9] and VDO-SLAM[6].

3.1. ORBSLAM2

ORBSLAM[8] is an open source feature based implementation of SLAM that allowed for various features such as loop closing and relocalization. Throughout the paper for ORBSLAM, various testing has been done for both indoor and outdoor environments. However, the algorithm lacked in accuracy, had some performance problems and only allowed for the implementation of monocular cameras. This is where ORBSLAM2 came in.

ORBSLAM2[9] is a massive upgrade from ORBSLAM, it allows the implementation of monocular, RGB-D and stereo camera. It also enhances greatly on the problems ORBSLAM faced, such as the performance issues as well as the accuracy issues. By the time ORBSLAM2 was released, it would have been considered state of the art in many aspects of SLAM research. Another advantage ORBSLAM2 provided is that it is not so much outdated as to render it useless, and it is also not so new that no research or community discussion have been done on it. It was perfect for this research, given the time frame, as it is very well documented, and it has an active community reporting back and posting consistently about fixes to errors that may be encountered, hence resolutions to a lot of problem could be easily found. In addition to this, there already exists a lot of research that make use of ORBSLAM and ORBSLAM2. All these aspects made ORBSLAM2 a very advantageous algorithm, in terms of both time and functionality, to use for this research project.

That is not to say that no roadblocks were faced when dealing with ORBSLAM2. However, these will be documented in greater detail in the upcoming sections. But despite such problems, the initial hypothesis was right, the extensive documentation came in handy when faced with the many problems encountered throughout the research and further research provided the basic idea in which the attempted improvement of ORBSLAM2 was conducted.

3.2. VDO-SLAM

VDO-SLAM[6] is another open source implementation that takes another approach for using SLAM, since this implementation focuses more on dynamic environments.

VDO-SLAM is intended to be used in dynamic environments, the reason as to why it was chosen for the project can be plainly clear. The algorithm provides features ORBSLAM does not, such as dynamic object tracking, which was a feature that seemed to offer great value to a smart bike provided it can work with reasonable performance on embedded systems. The paper on VDO-SLAM[6] provides decent documentation on the implementation, as well as an extensive performance results on analysis on different datasets providing a dynamic environment, these environment include indoors as well as outdoor settings.

Further, another reason VDO-SLAM was used, was due to the fact that, like ORBSLAM2, this algorithm is feature based, with relatively high accuracy. However, unlike ORBSLAM2, VDO-SLAM only has implementation for RGB-D camera. This can be considered a bit of a disadvantage but, if the final implementation on a smart bike only uses an RGB-D camera, both implementations will work fine. In addition to that, VDO-SLAM supports real time SLAM in a very easy manner.

Again, there were many roadblocks that were faced while building and installing VDO-SLAM, these will also be discussed in further sections, however, the features VDO-SLAM can greatly enhance the smart bike experience. Specifically, the dynamic object tracking feature, which allows the algorithm to identify moving objects, such as cars, and identify their velocity with good accuracy, which, again, can provide real world benefits to a smart bike.

4. Implementations, proof of concept and SLAM using socket connection

The approach that was taken throughout this research has varied in a lot of ways. Initially, all the SLAM algorithms were supposed to be run and tested in real time, providing more reliable and realistic results. However, the reason for the multiple change of plans that were done spawned from the problem that the hardware that was necessary to pursue this research did not arrive, and by mid-week 4 it had to be cancelled so that this research can carry on with whatever time was left. For about a week more, some reviewing of a bit of literature was done to try and detour from the initial subject to a more time appropriate subject and research question, but ultimately the choice to carry on with this research with the same research questions was taken. However, the approach was slightly varied.

Initially a LIDAR sensor was to be used with a combination of SLAM algorithms such as BreezySLAM[4], further RGB-D cameras would have also been used. This approach has entirely changed and the entire project will be based on working with the same algorithms that would’ve been used with the RGB-D cameras.

However, instead of working on hardware that can provide some real time results, such as the RGB-D camera, this research instead worked solely on datasets to test the algorithms, specifically the KITTI dataset[9]. The choice of the dataset was taken since KITTI provides outdoors videos next to cars and bikes, and it also shows footage traveling in narrow lanes, making it a reasonable dataset when looking into viewing a bike experience. The dataset also has almost 20 gigabytes of material, and thus providing a lot of footage to work with.

Furthermore, since the budget of a smart bike needs to be accounted for, the research is extended to enhance ORBSLAM2, which is an implementation of SLAM, in such a way that it can run better on lower end hardware such as a raspberry pi. This is done by implementing a version of SLAM using online socket connection with the ORBSLAM2 source.

As was mentioned before in the requirements section, the first part of the project concerns installing and building the libraries, while also getting familiar with the classes of each algorithm and the features they provide. Furthermore, for each algorithm, the difficulties faced while installing them will be mentioned, as well as how each algorithm was tested using datasets to make sure each implementation works.

In the second part of this section, the improvements made to ORBSLAM2 will be mentioned, as well as the difficulties faced while attempting to implement such improvements. Regarding the improvement of ORBSLAM2, two approaches were taken, each one will be explained in extensive detail and why each was implemented.

Most of the testing is done on virtual machines, where two virtual machines are used, one to simulate a higher-end machine, which will be referred to as server VM, and the other is to simulate a raspberry pi, which will be referred to as pi VM. This made testing the alterations to the algorithm a quicker process.

Various problems were faced throughout these processes, some are due to personal inexperience with building and installing libraries and programs on Linux, and some are due to various factors that have to do with the libraries, these issues are all detailed below. All in all, it took four trails on five different virtual machines to get both libraries to work on two different virtual machines, these two will come in handy when improving ORBSLAM2.

4.1. Building ORBSLAM2

As previously mentioned, the process of building ORBSLAM2 had many difficulties throughout. The main problem has to do with the library being a bit outdated, and that caused some trouble due to two reasons, one being the outdated installing instructions, and the other is that the install instructions do not mention the versions of some of the dependencies used.

For example, the install instructions only mention installing the Pangolin library, but it is only through trial and error that we uncovered that Pangolin version 0.5 is the version needed. Similar problems are faced with other dependencies such as Eigen3, as referencing Eigen3 in C++ headers has changed slightly since ORBSLAM2 was last updated.

In addition to that some problems were encountered with installing the different versions of OpenCV, these were due to multiple wrong assumptions that were made. One such assumption is that ORBSLAM2 can work fine with the newer OpenCV 4.0+ versions, which was not the case, this wasted about a day trying to understand why ORBSLAM2 does not build.

Whilst building ORBSLAM2 an issue was encountered in which the virtual machine would freeze until it was forced to shut down, that problem also caused some delay, between 1 to 1.5 days. The problem being that, by default, ORBSLAM2 builds by doing 8 processes at a time, this caused the machine to overload, the solution was just changing the build settings. Although the solution was simple, it was not obvious at the time.

4.2. Building VDO-SLAM

VDO-SLAM was faster to build because most of the dependencies had been installed already, additionally, some experience had built up while resolving errors in similar situations for ORBSLAM2. And so installing and building VDO-SLAM only took about half a day worth of work.

The only problem faced was that the documentation does not make it clear that OpenCV must be built with the extra modules. This took some time to figure out but not too long.

4.3. Socket connection ORBSLAM2

Socket connection SLAM refers to the implementation that was done during this research. The concept of the implementation is rather simple, one lower-end system is running on the smart bike gathering the image data from the camera, it then sends the data over to a higher-end computer over socket connection and then that computer runs the SLAM algorithm and returns the results for viewing back to the lower-end system. In the previous section the problems with resources on the virtual machine was discussed, as well as taking a look into the processing time of each frame, this online socket SLAM solution is aimed at either decreasing the processing time for the raspberry pi, or decreasing the resources used on the machine, making more space in the hardware for other processes to run.

To do such implementations a python wrapper[10] was needed to be installed. The reason for that was, again, to save as much time as possible since we were more familiar with python sockets and serialization unlike the alternatives in C++. The python wrapper uses Boost to make C++ functions work in python. However, the developers only provided python alternatives for the example classes in ORBSLAM2 and not the src classes.

Two variations of the “online socket SLAM” algorithm have been done during this research. One is more of a naïve approach that does not fulfill the complete vision of the algorithm. The second one did not fully work. The two approaches will be explained in detail in the following subsections.

4.3.1. Approach one: Naïve approach

The first approach was easy to implement, it is referred to as the naïve approach because it does not fully give the working result that was pursued. However, after seeing the results, it does show somewhat of a proof of concept. This allowed me to pursue a better approach.

The approach is done in two steps, the first one is sending the data gathered, in this case the data was from the dataset, on the raspberry pi VM and sending it over by using a socket connection to the server VM. The server VM carries on with the ORBSLAM2 algorithm and the results are shown.

Although this serves as a good proof of concept, this implementation is still flawed. The reason for that is that the results are not sent back to the raspberry pi VM, and so all the SLAM visualization happens on the server VM side. This can be a problem since it does not help the smart bike rider much. The reason the results are not just sent back to the raspberry pi VM is due to the way that ORBSLAM2 classes are structured. The python file uses one of the C++ source files called System.cc which in turn calls functions from Tracking.cc. As the tracking, in the Tracking.cc file, is updated the viewing of the results is done and thus happens simultaneously. And so, the problem remains that getting the results back to the raspberry pi VM will require a bit more work than what was expected, which is where the second approach comes in.

Even if this approach did not provide the demanded results, after seeing the results it encouraged more work into the second approach, and so this naïve approach worked more as a proof of concept. The reasoning for this as well as the results for the naïve approach are mentioned in the results section.

4.3.2. Approach two

The second approach requires more hacking into the source code of ORBSLAM2. This is where the python wrapper becomes a slight disadvantage, since whenever changes are made to the ORBSLAM2 source code, the python wrapper needs to be updated by making new functions that make use of the changes done, then ORBSLAM2 needs to be rebuilt and then finally the python wrapper needs to be rebuilt. This can get time consuming, so preferably this process is done the minimum number of times.

Since this second approach did not work, the explanation is all theoretical backed up by the results shown for the first naïve approach. The second approach works by sending the data from the raspberry pi VM to the server VM as done in the previous approach, then the System.cc file returns the Frame datatype that is updated in the variable mCurrentFrame, which is found in the function GrabImageMonocular belonging to the Tracking.cc file, to the python code. This Frame stuct is then serialized and sent back to the raspberry pi VM through a socket connection. The raspberry pi VM then should update the map and carry on with the tracking function.

This approach is a more realistic algorithm, since it would provide useful results to the person riding the smart bike. However, it will of course have more processing time than the first (naïve) approach.

5. Results

This section discusses the results of building and running the algorithms and their use cases, while also showing the results of the improvements made using both approaches.

5.1. Build results of ORBSLAM2 and VDO-SLAM

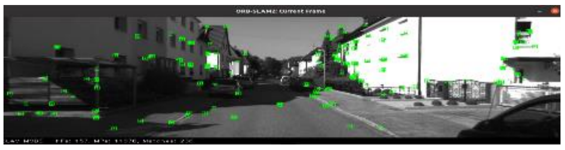

Figure 1 shows ORBSLAM2 operating on the KITTI dataset for monocular camera

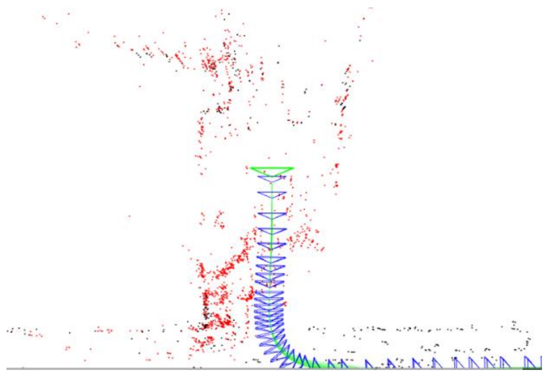

Figure 2 shows part of the map built by the ORBSLAM2 algorithm

Even though, some problems occurred during building, however, after ORBSLAM2 was built, the results were as expected on the KITTI dataset as shown in figure 1. ORBSLAM2 was simple to use and test the datasets on. ORBSLAM2 constructs a map of the road, as seen in figure 2, which can be of great help to smart bike users.

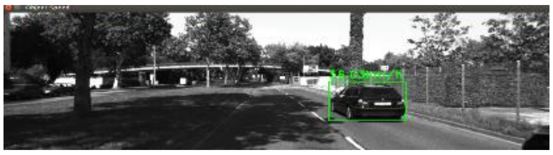

Figure 3 shows VDO-SLAM operating on the KITTI dataset for an RGB-D camera

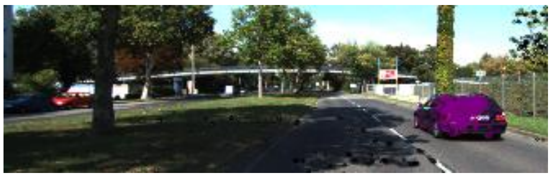

Figure 4 shows VDO-SLAM operating on the KITTI dataset for an RGB-D camera

The same results apply for the build of VDO-SLAM. The results were in line with what was expected as shown in figure 2, and it was also easy to test datasets on. As can be seen in figure 3, VDO-SLAM can calculate the velocity of a vehicle on the street, which is a unique feature that can help users.

5.2. Results of both algorithms on VMs

Average time of processing one frame (s)		
	Average time using Monocular (s)	Average time using RGB-D (s)
ORBSLAM2	0.002114	-
VDO-SLAM	-	1.258050

Table 1 shows the average time taken to process a frame for the different algorithms on virtual machines

For ORBSLAM2 the average is calculated based on using the processing time for the first 200 frames. VDO-SLAM does not have an implementation for monocular cameras which is why the cell is left empty. Table 1 shows the average time to process one frame for both algorithms using both a monocular camera and an RGB-D camera, these results are obtained on a virtual machine that has the same hardware resources as a raspberry pi, virtually simulating it.

The results obtained in using ORBSLAM2 are good in terms of processing time, however the problem that was encountered concerns the resources used. Operating ORBSLAM2 is demanding in terms of computer resources, and the more time it keeps running, the more resources it needs. Figure 3 shows the CPU and RAM usage on the virtual machine had almost 1.8 GB of the 2 GB ram being used as seen in figure 4. This system is only running ORBSLAM2, which will not be reasonable for the implementation of a smart bike since more algorithms will be needed to run. The problem only gets worse the more time the algorithm runs, taking up more RAM.

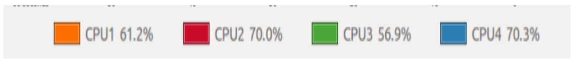
🟧 CPU1 61.2% 🟥 CPU2 70.0% 🟩 CPU3 56.9% 🟦 CPU4 70.3%

Figure 5 shows CPU usage on virtual machine

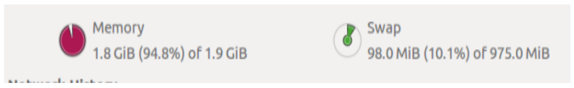
Memory
1.8 GiB (94.8%) of 1.9 GiB
Swap
98.0 MiB (10.1%) of 975.0 MiB

Figure 6 shows RAM usage on virtual machine

Additionally, the algorithm was also run on a virtual machine with more resources put into it. The average time for processing one frame in ORBSLAM2 was 0.001850 seconds. Surprisingly, the time difference between the two virtual machines was not that great, just about 0.000264 seconds.

5.3. Results of both algorithms on a raspberry pi 4

	Average time of processing one frame (s)	
Trials	ORBSLAM2	VDO-SLAM
1	0.007077	8.110560
2	0.007272	4.967875
3	0.007146	5.326230
4	0.007117	4.780151
5	0.007088	7.179776
6	0.007132	6.699401
7	0.007072	4.531626
8	0.007072	4.415732
9	0.007006	3.730072
10	0.007161	4.626327
11	0.007184	4.850431
12	0.007171	7.498999
13	0.007255	6.637877
14	0.007130	5.288502
15	0.006928	7.014448

Table 2 shows the average time taken to process a frame for the different algorithms on a raspberry pi

15 trails were carried out for each algorithm on the raspberry pi. ORBSLAM2 showed consistent results, with an average of 0.00712057 seconds, however that was not the case for VDO-SLAM which showed very inconsistent results, ranging from 3.730072 seconds at its lowest to 8.11056 seconds at its highest. 1.2 GB of the 1.88 GB on the raspberry pi where used, with almost 800 GB from the memory swap also being used. These figures further illustrated the problem cause by running SLAM on lower-end embedded systems. These surprising results also show the large difference between running the two algorithms on a virtual machine versus running both on the raspberry pi. As can be seen, the difference is almost 3.37 times more on the raspberry pi for ORBSLAM2 and the difference can range from almost 3 times to 6.4 times on a raspberry pi for VDO-SLAM.

One reason for the large increase in running time might be the fact that the raspberry pi is running on Ubuntu 18.04 which was not made to be run on a raspberry pi. This version of Ubuntu was needed to mimic the environment that can support both OpenCV 3.2 and OpenCV 3.4, the environment was also needed to be able to build ORBSLAM2, which is a bit outdated. To make that happen, Ubuntu 18.04 server for raspberry pi was installed on the device and then we installed the desktop version from the terminal. Further to counter the performance problems as much as possible, we tried to increase the memory swap to almost 5 GB, which was the best that could be done given the SD card which was only 32 GB.

5.4. Socket connection SLAM: approach one (Naïve approach)

I mentioned previously that the Naïve approach is considered a proof of concept that encouraged working on and developing a better solution. Given that, the most important part of this section is to reveal why such consideration was done and what it can reveal about the eventual application of the second approach.

	Average time using Monocular (s)
Raspberry pi VM	4.642626e-06
Server VM	0.002629

Table 3 shows the average processing time for both virtual machines.

First, the most noticeable difference is the processing time of the Raspberry pi VM which is significantly decreased. However, this result is not greatly impressive or important because using this implementation, the only thing the system functionally does is send the data to the Server VM. Absolutely no SLAM processing or result viewing is done on that side, so the results of the second approach will not be expected to have such low processing times, on the contrary. Given that the second approach works, the time for it to show the results will be the time it already takes to send the files, which is virtually next to nothing, plus the time for the Server VM to process that frame, plus the time for the Server VM to send back the frame.

The second difference is clearly the rather mysterious increase in the processing time on the Server VM side. We frankly could not find a valid reason for this increase in processing time for the SLAM algorithm, but it is consistently present.

With that being said, the enhancement that could be expected from the second approach would be the decrease in resources used on the Raspberry pi VM, rather than a better processing time overall. This could still be helpful to the system, since the increase in time would still not be excessive to the point where the system is rendered useless, however, the decrease in resource use can make way for multiple functionalities to be implemented for the smart bike.

5.5. Socket connection SLAM: approach two

The second approach to the socket connection ORBSLAM2 did not work out as planned, in the end it didn't function properly to be able to obtain final results on the resource usage of the algorithm, in this section we will explain why that happened and what the alternatives may be. The problem simply has to do with the python wrapper that was used, as that converter does not convert all the data types and functionality of ORBSLAM2. That causes the conflict, since in order to send the results to the server VM we must first return the struct variable to the python file with the Frame dataset, which was not converted to python, so that causes the error. Trying to change the datatype to its python equivalent would not be the best way to go, a better alternative is to not use the python wrapper all together and instead do the socket connection and serialization on C++, however this could not be done during the time of this research due to time constraints.

6. Conclusion

Throughout this research, different SLAM algorithms were run and tested, where each algorithm can bring benefit to the implementation of a smart bike. Further, a method was developed to try to improve the algorithm for the smart bike use case. The use cases for SLAM on smart bikes was also shown, as it can be used to map out places the user has not visited before, which can be seen when using ORBSLAM2, and it can also be used to show the user the velocity and trajectories of the surrounding vehicles, as shown using VDO-SLAM, which can be an additional safety measure.

Further, some adjustments to the ORBSLAM2 algorithm using socket connections to allow for lower resource use on the embedded system working on the smart bike. The first approach, that worked as a proof of concept, showed that the system can be helpful for the smart bike provided some additional changes are made, which is where the second approach comes in, however, it is still not fully functioning. Regardless, this paper shows the SLAM can be used on some lower-end embedded systems such as the raspberry pi, which may encounter some problems due to hardware limitations, but the online (socket) slam implementation may eliminate such problems.

6.1. Future work

For further research, the implementation of the socket ORBSLAM2 could be resumed, with testing to show how useful the approach may be. Also, testing on live feed using a monocular camera and ROS could be done, to ensure that the algorithm fully works in real world scenarios.

Although the research showed some promising results, some further research into more use cases can be beneficial, specifically use cases of LiDAR sensors and SLAM.

7. References

[1] Sotirios Stasinopoulos, Mingguo Zhao, and Yisheng Zhong. 2017. Simultaneous localization and mapping for autonomous bicycles. International Journal of Advanced Robotic Systems 14, 3 (2017), 172988141770717. DOI:http://dx.doi.org/10.1177/1729881417707170

[2] Oliver Roesler and Vignesh Padubidri Ravindranath. 2019. Evaluation of slam algorithms for highly dynamic environments. Advances in Intelligent Systems and Computing (2019), 28–36. DOI:http://dx.doi.org/10.1007/978-3-030-36150-1_3

[3] Mohamed Abouzahir, Abdelhafid Elouardi, Rachid Latif, Samir Bouaziz, and Abdelouahed Tajer. 2018. Embedding slam algorithms: Has it come of age? Robotics and Autonomous Systems 100 (2018), 14–26. DOI:http://dx.doi.org/10.1016/j.robot.2017.10.019

[4] Joachim Clemens, Thomas Reineking, and Tobias Kluth. 2016. An evidential approach to slam, path planning, and active exploration. International Journal of Approximate Reasoning 73 (2016), 1–26. DOI:http://dx.doi.org/10.1016/j.ijar.2016.02.003

[5] Suraj Bajracharya. 2014. BreezySLAM: A Simple, efficient, cross-platform Python package for Simultaneous Localization and Mapping (thesis).

[6] Jun Zhang, Mina Henein, Robert Mahony, Viorela Ila. 2020. VDO-SLAM: A visual dynamic object-aware SLAM

system.DOI:https://doi.org/10.48550/arXiv.2005.11052

[7] Andreas Geiger, Philip Lenz, and Raquel Urtasun. 2012. Are we ready for autonomous driving? The KITTI vision benchmark suite. *2012 IEEE Conference on Computer Vision and Pattern Recognition* (2012). DOI:https://doi.org/10.1109/cvpr.2012.6248074

[8] Raul Mur-Artal, J. M. M. Montiel, and Juan D. Tardos. 2015. ORB-SLAM: A Versatile and Accurate Monocular SLAM System. IEEE Transactions on Robotics 31, 5 (2015), 1147-1163. DOI:https://doi.org/10.1109/tro.2015.2463671

[9] Raul Mur-Artal and Juan D. Tardos. 2017. ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras. *IEEE Transactions on Robotics* 33, 5 (2017), 1255-1262. DOI:https://doi.org/10.1109/tro.2017.2705103

[10] John Skinner and Dmytro Mishkin. ORB_SLAM2-PythonBindings. GitHub. Retrieved June 26, 2022 from https://github.com/jskinn/ORB_SLAM2-PythonBindings