SLAMfusion: Fusing SLAM Methods for Improved Robustness

Miguel Fernandes* and Luís A. Alexandre†
Instituto de Telecomunicações, Universidade da Beira Interior
Rua Marquês d'Ávila e Bolama, 6201-001, Covilhã, Portugal
*Email: killazfern{at}gmail.com
†Email: luis.alexandre{at}ubi.pt

***Abstract*—There are multiple approaches for SLAM, but we found the the ones implemented in ROS had problems when a robot drove over small obstacles. This paper presents a proposal to make a more robust SLAM by running three SLAM methods in parallel and using their information to produce a better estimate of the robot's surroundings. The proposed method defines its output by making the three methods vote for the value of each pixel in the map. To deal with the increased computational complexity, the method is implemented in the GPU. The performed experiments show that our method shows smaller error than any of the three fused methods alone both when there are ground obstacles that induce map errors and also when no obstacles are present, thus presenting in fact an increase in robustness.**

I. INTRODUCTION

In the area of mobile robotics, one of the first and most fundamental steps is the SLAM (Simultaneous Localization and Mapping), but it is also one of the most problematic areas. A small error in the map may mean the robot can't return to its dock or home or it can't accomplish its mission correctly. There are multiple SLAM methods, but we found that several of the methods implemented in ROS [11] had problems when the robot we used (a Turtlebot 2) drove over cables or other ground objects.

So in this paper we present a new SLAM method to increase the robustness of SLAM by using different maps created from different methods at the same time, fusing the resultant maps into a single one, in an attempt to create a robust map that has less errors than the original maps. Since the computational cost of running three SLAM maps simultaneously is higher than running a single one, we implemented our method in the GPU to take advantage of the parallel computation available in the graphical card.

The paper is organized as follows: the next section discusses some related work, section III presents the proposed method, section IV contains the experiments, the following section contains a discussion of the obtained results and the final section contains the conclusions.

II. RELATED WORK

Several work has been done to improve the robustness of SLAM, such has Ratter[7] presenting 2D information fused with 3D, by combining the information of a rangefinder and an RGB-D camera, using the 3D camera for mapping and localization and the rangefinder as a larger overview to aid the 3D system in localization (rangefinders have a very large field of view compared to the RGBD-Camera). This additional information aids in the loop closure of the created maps. The method runs on a robot rather than on a workstation.

Eudes *et al* [10] proposes to fuse the odometer information to monocular visual information in order to improve the camera position at each input image, with also the 3D point cloud being rebuilt to allow iterative positioning.

Ableles [9] contributed with methods to aid in uneven floors using new stable geometric estimation equations, ideas that aid a robot in localizing itself on a unknown location, only with a map created from blueprints or previous exploration, that may not be extremely accurate since different sensors and different robot sizes may provide different maps. The blueprint will not have ground depressions, bumps or other kind of defects, or even other problems like different furniture placement.

Graham *et al* [8] provides an incremental SLAM. Instead of focusing in the incorrect loop closures, it focuses in the problem of mapping incorrect landmark information and instead of working in an offline way, it works in real time preventing the problems, rather than fixing them later.

In ROS there are two methods that would do some kind of map merging: MapStitch[6] and MapMerger[5].

MapStitch is meant to align a map from a running `/world` node to a previous made 2D saved map, combining them to a single image. The problem with it was a lack of references to work with ROS Indigo, a poor documentation and the ROS component does not allow the use of several SLAM methods to create a single map or even to use two running SLAM methods to create the map, allowing only a previous map and the actual one, which would not help in preventing errors or improving the real-time creation of a robust map.

On the other hand, MapMerger had references to work with ROS Indigo, although this method relied on various robots connected via Ad-Hoc communication, and merged the different maps that the different robots did, into a single global one. Due to the difficulties related with the need to use different masters for a single SLAM method instead of running everything under the same master, we decided to create the SLAMfusion method, that we believed could not only solve these usability issues but also improve the overall system robustness.

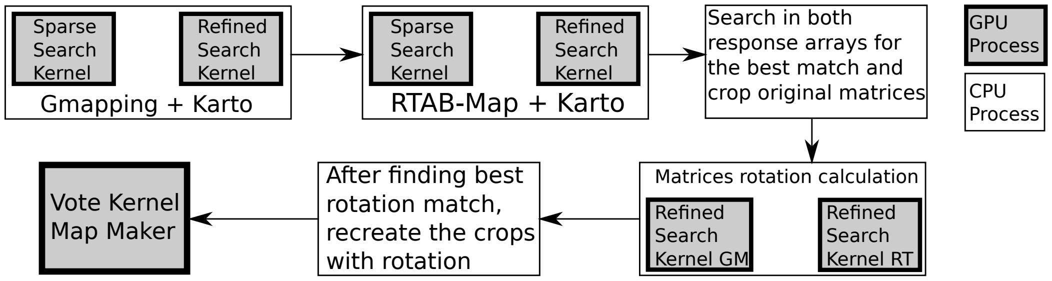


Fig. 1. The workflow of SLAMfusion.

III. Fusing multiple SLAM methods

A. Methods Used

Our proposed method is based on running three SLAM methods in parallel and fusing their decisions. The three SLAM methods that we run are Gmapping, Karto and RTAB-Map.

Gmapping [3] is an OpenSlam ROS wrapper. It is a Rao-Blackwellized particle filter used to create grid maps from laser scanner data. Each particle of the filter carries an individual map of the environment. Normally it requires a high number of particles to get a good map. On our experiments we used 80 particles, since we were running the SLAM method on the workstation instead of the computer controlling the the turtlebot, thus having more room to obtain more accuracy without sacrificing performance.

Karto is SRI International's commercial Graph-based SLAM algorithm available open-source [12] on ROS repositories. There is not much information or documentation available about it on ROS or Karto's website. From [4] we found that each node of the graph keeps pose information along with sensor measurements. Normally, with a large number of landmarks, more memory is required. But since it is a graph-based SLAM that only keeps the pose information per node, it is very Memory-efficient.

RTAB-Map (Real Time Appearance-Based Mapping) is a Graph-Based RGBD-SLAM, used due to its visual SLAM rather than the depth-to-laserscan approach of the two other SLAM methods, functioning as a tie breaker. It can easily detect other obstacles under and above the fake laser scanner, since it is a 3D-SLAM that projects itself as a 2D map. As referred by Labbe [2], it works as a multi-level memory, as it has several memory levels to keep it working in real-time, unloading nodes from working memory to long term memory and moving them back to working memory when they are required for loop closure. It is also a multi-session SLAM Method: it keeps a database with the graph information providing the multi-slam solution, although in our experiments, we deleted the said database file so it wouldn't affect the various results from the several conducted experiments, giving less of a advantage against the other methods that do not have multi-session memory.

These methods were chosen due to various aspects. Gmapping was chosen due to fact that it was already implemented and the tutorials available for our robot use it. Karto was chosen because it had the best accuracy in an evaluation presented in Santos *et al* [4]. RTAB-Map was chosen due to the fact that it was different from the other two SLAM methods already chosen, since it is real time appearance based, meaning that it uses the entire depth cloud to do the map, instead of using only filtered information from depth image to laserscan to create the map.

The methods referred were running simultaneously, publishing information to different ROS nodes instead of the default `/map` node.

B. Fusion

The proposed method is based on the fusion by majority vote among the three existing methods presented in the previous section. Figure 5 presents the several steps of the method. It schematizes in a simple way how our setup works, from the robot sensors, to the different SLAM methods we used. The SLAM methods' output nodes are remapped from `/map` to `/theirname/map`, like `/Gmapping/map`. These nodes contain a `OccupancyGrid` data structure, composed of *.info*, that contains information like height and width of the map and resolution, and `.data` that contains a list of values that range from -1 (unknown), 0 (empty) and 100 (occupied). The algorithm starts by normalizing (achieved by transforming the values 100 to 1) and calculating the best match (translation and rotation) between the various methods by calculating the the error of *Gmapping's map* with *Karto's map* and *RTAB-Map's map* with *Karto's map*. The choice of Karto as a base map is due to the fact that it is the smaller map calculated, as it does not have a unknown margin, situation that happens with the other used methods. After the best match is found and smaller crops are made of each made, removing unneeded information, the algorithm calculates the final map by voting each point that composes the map, this means, if at least two methods on a certain value, the final map will have said value. In case of disagreement between the various methods, it was decided that the value to be used was 1. This decision was made after various tests that shown that the most disagreements were on extremities, meaning, zones that are between empty spaces and unknown spaces.

C. GPU implementation

Due to the fact that the amount of data to process in real time was very high, we decided to implement our method using a GPU to speed up the calculations. We used the Pyopencl library in Python, that acts as an opencl wrapper, giving us access to GPU Computing while maintaining high-level code/scripting given by Python.

By developing two small pieces of code in C, the *kernels*, made to be compiled and ran in the GPU (in our particular case, but it can be ran in any compatible opencl device), we upgraded significantly the performance of our method, from an average of 5 minutes per map created to 25 seconds to create the entire output used in this paper, which is: 18 error calculations from each individual method map to the ground truth, 6 maps from each trio of method maps and 6 errors from the created maps to the ground truth, with a global jumping step of 40.

Since the GPU runs the code, either Python or C will have the same performance potential, with the advantage that the heavy-duty part of allocating memory, controlling buffers, etc. is managed by the machine instead of the programmer.

As referred previously, we developed two kernels, the first receives, through buffers, the first map and the second map as a pointer to float, a pointer to a response float array, and an integer pointer to a two position array that has the size of the arrays. Then, the kernel executes the formula referred in the the following Results section, and by calculating the error for two different maps, we are able to find the best spot where a smaller map matches a larger map. This is accomplished by having two `for` cycles going from 0 to the size of the bigger map minus the smaller map, where the iterations match the coordinates where the larger map splices. At the end of each iteration, the error is saved to a tuple, X coordinate, Y coordinate and the rotation of the map.

Since it is computationally demanding to run this search for every single position of the map, an approach that we took was to jump positions, this means, instead of the `for` cycles processing each possible position in the map, they jump by a given step horizontally and vertically. Then, we re-run said function, with a narrower search area, and a smaller step, improving speed w.r.t. the method that would search every possible position in a single loop.

The second kernel receives the three maps and their sizes, and returns an array composed by the vote for the value of each pixel in the map from each SLAM method. When in doubt, we decide to assign the value corresponding to a obstacle, since in our experiments when the methods cannot agree it usually happens in the border from unknown to empty space, where normally there is an obstacle.

Regarding pre-GPU and pos-GPU processing, there are the array and buffer creation, tuple sorting to calculate where there is a better match (smaller intermap error), and rotation calculation. This is necessary since the SLAM Methods don't always output aligned maps. The rotation is achieved with OpenCV-Python wrapper, that rotates 2D matrices, after getting the closest position between the three maps, rotating the maps and saving the error value, and the rotation values. At the end, we chose the lowest error, recalculate the rotation and provide the three matrices to the second kernel.

IV. Experiments

We run several experiments to evaluate the performance of the SLAMfusion method. The experiments were run, first without obstacles in the robot's path, to obtain baseline values for the errors of the individual SLAM methods and afterwards, using obstacles to evaluate the variations in robustness of the SLAM methods.

A. Our Setup

The workstation used has the following features:

- CPU: Intel(R) Core(TM) i7 @ 3.20Ghz
- GPU: GeForce GTX TITAN X.
- Ram: 24 GB
- OS : Ubuntu 14.04.03 LTS with ROS Indigo

The robot is a Turtlebot2:

- Max speed: 700 mm/sec and 180 deg/sec
- Odometry: Built-in 3-axis gyro-meter, High resolution wheel encoder (11.7 ticks/mm)
- Microsoft XBox 360 Kinect RGBD-Camera/sensor.

The robot is controlled by:

- Portátil Magalhães 1 (Based on the second version of Intel's Classmate PC)
- CPU: Intel Atom @ 1.6Ghz
- RAM: 2 GB
- OS: Xubuntu 14.04 with ROS Indigo

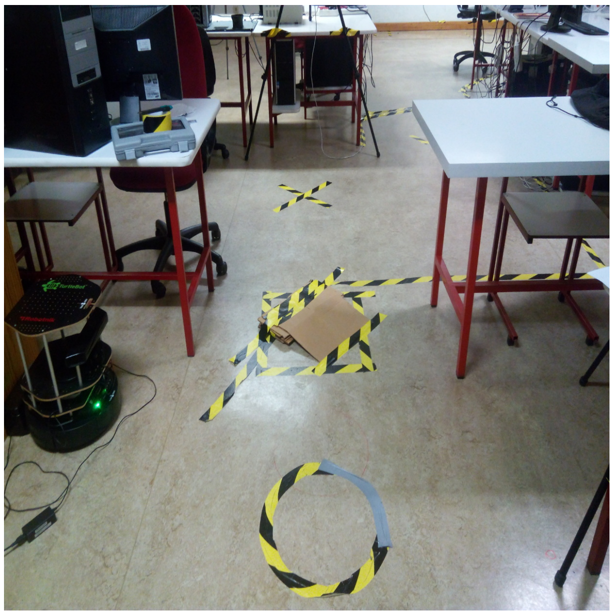


Fig. 2. The turtlebot on its charging dock and the obstacle course. The ramp is low enough so it won't activate the wheel drop sensors that cause the turtlebot to freeze, but high enough to cause slips that confuse the odometry, Also, the entire setup is marked and affixed with duct-tape.

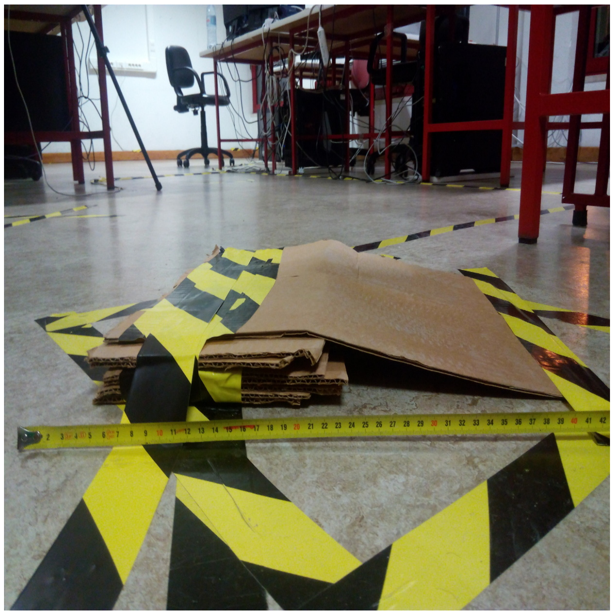


Fig. 3. The obstacle used, made of cardboard with duct-tape binding it all together.

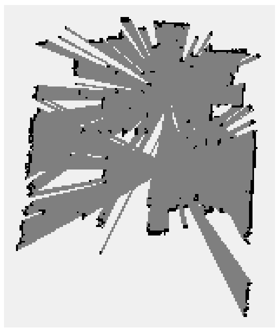


Fig. 4. Our ground truth map, a map made with teleoperation with aid of Rviz ROS program, displaying the map, Laser scan and Depth Cloud, to ensure the best map to be a reference to the experiments.

B. Course setup and ground truth map

The experiment procedure was the following. At the start we manually placed the turtlebot on the duct-tape circle, started its nodes, `Startup` and `3Dsensor`, then we started the three SLAM methods and started our SLAMfusion method. When our method receives at least one frame from each SLAM method, it starts a script that does a 360 degree turn, at 90 deg/s, then it goes forward at 0.5 m/s, in the direction of the duct-tape cross, passing over the obstacle, then it stops and does another 360 degree turn. After confirming that our method receives all broadcast information, we backup all created data, stop the SLAM-methods and our method, delete the RTAB-map database file to ensure a clean state experiment, then relocate the robot to the duct-tape circle and repeat the experiment.

The calculations and consequently the SLAMfusion maps are calculated offline since it is not yet fully able to process all frames in real-time.

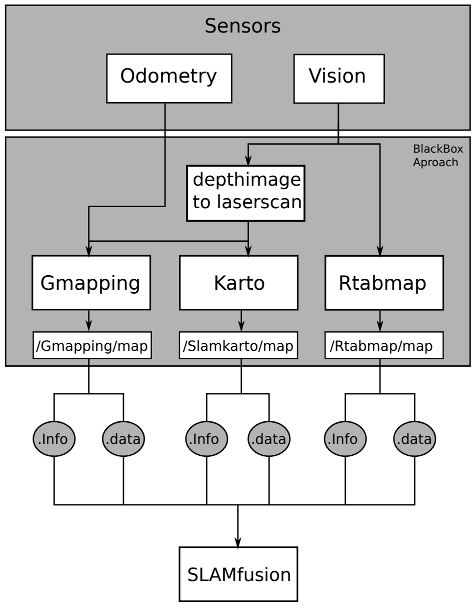


Fig. 5. The overall idea of the proposed method: fusing three SLAM methods using voting (see text for details).

C. Results

The error was found by the square root of the sum of the squared errors between each pixel of the map and the ground truth. If $T(x,y)$ is the ground truth map and $M(x,y)$ is a given map, then the error is obtained using:

$$Error(M(x,y)) = \left(\sum_{\forall (x,y)} (T(x,y) - M(x,y))^2 \right)^{1/2}$$

The results presented in table I show the average errors and standard deviations for three repetitions of the experiment. The average errors are also shown in figure V.

TABLE I
AVERAGE ERROR (AND STANDARD DEVIATION) FOR 3 REPETITIONS OF THE EXPERIMENT FOR BOTH THE CASES WHERE NO OBSTACLES WERE PRESENT AND FOR THE CASE WHERE THE WAS AN OBSTACLE.

Method	No obstacles	With Obstacles
Gmapping	148.63 (2.24)	151.28 (5.30)
Karto	128.36 (2.03)	132.58 (0.75)
RTAB-Map	106.38 (7.74)	110.03 (2.27)
SLAMfusion	97.88 (6.58)	97.45 (4.73)

Figure 6 shows one example of a map produce by each of the four methods without any obstacle and figure 7 shows one example of a map produce by each of the four methods when the obstacle is present.

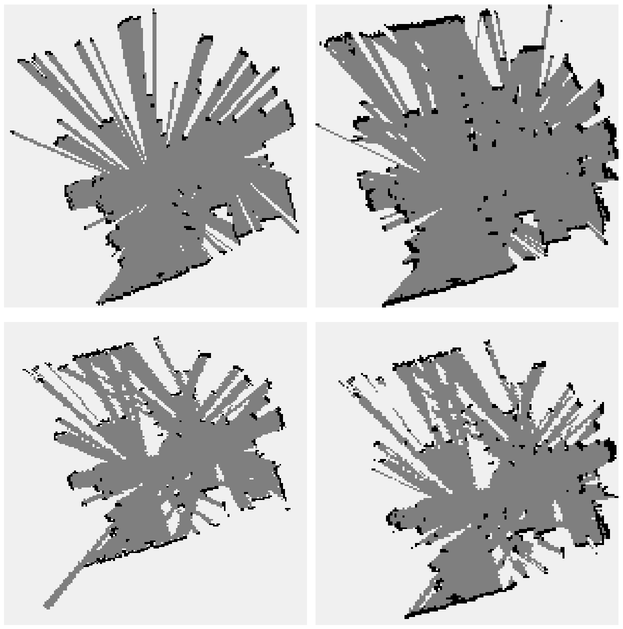


Fig. 6. Examples of maps when no obstacles are used, from top left to bottom right: Gmapping, Karto, Rtabmap and SLAMFusion.

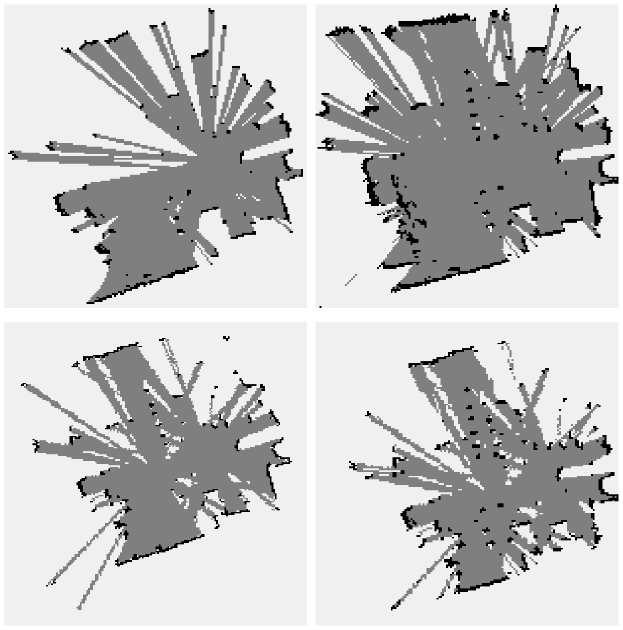


Fig. 7. Examples of maps when the obstacle is used, from top left to bottom right: Gmapping, Karto, Rtabmap and SLAMFusion.

TABLE II

p VALUES FOR A STATISTICAL SIGNIFICANCE TEST OF THE RESULTS PRESENTED IN TABLE I FOR THE EXPERIMENTS WITHOUT OBSTACLE. ALL DIFFERENCES BETWEEN PAIRS OF METHODS ARE SIGNIFICANT WITH THE EXCEPTION OF THE DIFFERENCE BETWEEN RTAB-MAP AND SLAMFUSION. SIGNIFICANCE LEVEL $\alpha = 5\%$.

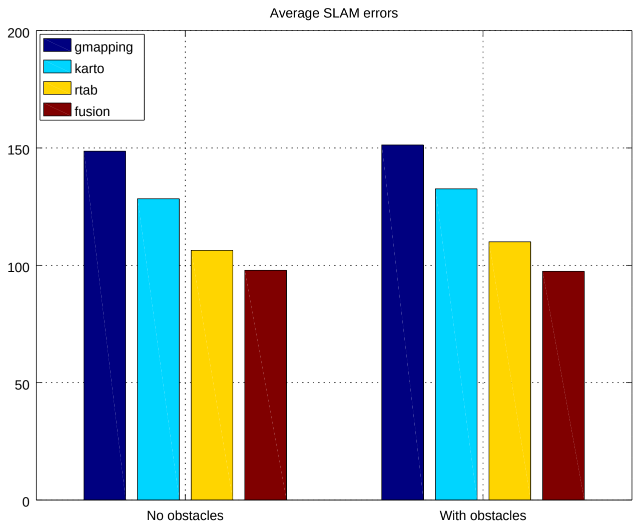
Method	Gmapping	Karto	RTAB-Map	SLAMfusion
Gmapping	-	3.1e-4	8.1e-4	2.3e-4
Karto	3.1e-4	-	8.9e-3	1.6e-3
RTAB-Map	8.1e-4	8.9e-3	-	2.2e-1
SLAMfusion	2.3e-4	1.6e-3	2.2e-1	-

TABLE III

p VALUES FOR A STATISTICAL SIGNIFICANCE TEST OF THE RESULTS PRESENTED IN TABLE I FOR THE EXPERIMENTS WITH OBSTACLE. ALL DIFFERENCES BETWEEN PAIRS OF METHODS ARE SIGNIFICANT. SIGNIFICANCE LEVEL $\alpha = 5\%$.

Method	Gmapping	Karto	RTAB-Map	SLAMfusion
Gmapping	-	3.8e-3	2.4e-4	1.9e-4
Karto	3.8e-3	-	8.2e-5	2.2e-4
RTAB-Map	2.4e-4	8.2e-5	-	1.4e-2
SLAMfusion	1.9e-4	2.2e-4	1.4e-2	-

V. DISCUSSION



From the results we can conclude that all methods had a consistent performance without and with obstacle. Gmapping had the worst result, followed by Karto and RTAB-map. In all cases, SLAMfusion presented the smallest error. The differences in performance between all pairs of methods in each setting (without and with obstacle) are statistically significant with the exception of the difference between RTAB-Map and SLAMfusion for the setting without obstacle (see Tables II and III).

As referred by Santos *et al* [4], Karto has better performance than Gmapping in real world experiments. As for RTAB-Map, the fact that it has a smaller error value than the other SLAM methods may be due to the fact that it is in reality a 3D SLAM method that is flatten to a 2D map.

It is interesting to see that SLAMfusion had better results (marginally) when the conditions were worst (obstacles). This might just be a coincidence and the difference is not significant. More experiments are needed to better confirm these results.

VI. Conclusion

In this paper we presented a proposal to improve SLAM robustness to noise. It takes advantage of three implementations of SLAM present in ROS running in parallel and makes them vote for the class of each map pixel.

To improve the speed of SLAMfusion it was implemented in GPU with some heuristics for faster convergence in terms of map registration between the different SLAM methods.

The experiments show that SLAMfusion is in fact a more robust solution than any of the other three approaches, and is immune to the errors caused by the obstacle tested in the experiments.

As future work we want to run more tests and in different locations to have a more precise idea of the robustness of SLAMfusion and continue improving the speed of the method in various ways, for instance, by using an autonomous search ”jump” calculation while searching for the maps’ alignment for best accuracy/performance in order to be able to achieve real-time processing.

Acknowledgments

This work was supported by National Funding from the FCT - Fundação para a Ciência e a Tecnologia, through the UID/EEA/50008/2013 project.

The Titan X used in this research was donated by the NVIDIA Corporation.

References

[1] Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., Fasih, A. (2012). PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. Parallel Computing, 38(3), 157–174. doi:10.1016/j.parco.2011.09.001

[2] Labbe, M., Michaud, F. (2014). Online Global Loop Closure Detection for Large-Scale Multi-Session Graph-Based SLAM. In International Conference on Intelligent Robots and Systems (pp. 2661 – 2666). doi:10.1109/IROS.2014.6942926

[3] Grisetti, G., Stachniss, C., Burgard, W. (2007). Improved techniques for grid mapping with Rao-Blackwellized particle filters. IEEE Transactions on Robotics, 23(1), 34–46. doi:10.1109/TRO.2006.889486

[4] Santos, M., Rocha, R. P. (2011). An Evaluation of 2D SLAM Techniques Available in Robot Operating System. IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR), 1–6. doi:10.1109/SSRR.2013.6719348

[5] Andre, T., Neuhold, D., & Bettstetter, C. (2014). Coordinated multi-robot exploration: Out of the box packages for ROS. 2014 IEEE Globecom Workshops, GC Wkshps 2014, 1457–1462. http://doi.org/10.1109/GLOCOMW.2014.7063639

[6] https://github.com/tu-darmstadt-ros-pkg/mapstitch

[7] Ratter, A., & Sammut, C. (2015). Fused 2D/3D Position Tracking for Robust SLAM on Mobile Robots. Cse.Unsw.Edu.Au, 1962–1969. http://doi.org/10.1109/IROS.2015.7353635

[8] Graham, M. C., How, J. P., & Gustafson, D. E. (2015). Robust Incremental SLAM with Consistency-Checking. Iros 2015. http://doi.org/10.1109/IROS.2015.7353363

[9] Abeles, P. (2011). Robust local localization for indoor environments with uneven floors and inaccurate maps. IEEE International Conference on Intelligent Robots and Systems, 475–481. http://doi.org/10.1109/IROS.2011.6094498

[10] Eudes, A., Naudet-Collette, S., Lhuillier, M., & Dhome, M. (2010). Weighted Local Bundle Adjustment and Application to Odometry and Visual SLAM Fusion. Procedings of the British Machine Vision Conference 2010, 25.1–25.10. doi:10.5244/C.24.25

[11] Quigley, M., Conley, K., Gerkey, B., FAust, J., Foote, T., Leibs, J., . . . Mg, A. (2009). ROS: an open-source Robot Operating System. Icra, 3(Figure 1), 5. doi:http://www.willowgarage.com/papers/ros-open-source-robot-operating-system

[12] http://www.ros.org/news/2010/04/karto-mapping-now-open-source-and-on-coderosorg.html