1 Introduction

In robotics and autonomous navigation, SLAM (Simultaneous Localization and Mapping) is a crucial field. It is the process by which a robot walks around an unknown environment and uses sensor and odometry data to create a map of the environment while estimating its position within it. SLAM is required for robotic systems to operate autonomously in both simple indoor applications such as automatic vacuum cleaners and significantly more complex outside applications like as self-driving cars. SLAM is currently widely regarded as a solved problem, at least in most non-challenging interior contexts [1], because to the use of advanced 3D sensors such as LiDAR and Kinect. However, these sensors have limits, such as LiDAR's expensive cost and Kinect's limited range, making the former unsuitable for low-cost systems and the latter unsuitable for outdoor applications. As a result, visual SLAM, which uses only one or two 2D cameras as sensors, remains a popular study topic. It's also a difficult one, because the lack of direct 3D information necessitates deducing the scene's 3D structure by comparing elements in several photos obtained from various perspectives. However, as rich visual information from cameras is required for successful loop closure detection even in the absence of 3D sensors, using the same information for map construction and localization can be extremely beneficial. As a result, the potential benefits of putting such a system in place are substantial.

ORB-SLAM [2, 3] is a cutting-edge visual SLAM system that uses only a monocular camera to create a point cloud-based map of even the most difficult outside situations. Though this point cloud is valuable for acquiring the environment's 3D structure, it is less effective for path planning and navigation utilising algorithms that require a 2D occupancy grid map [4]. ORB SLAM generates a sparse point cloud, making it challenging to construct an occupancy map that includes the majority of the obstacles while simultaneously giving enough contiguity in the known free areas for path planning to work reliably. This is the challenge that this project attempts to answer by devising a method for creating an occupancy grid map in real time utilising ORB SLAM's 3D data. The grid map should be good enough for ROS's standard navigation stack [5] to use it to generate navigation commands that allow a robot (real or simulated) to follow the ORB SLAM-generated camera trajectory.

2 Literature Review

The production of occupancy grid maps and obstacle identification from a point cloud is a well-studied issue in the literature. Goeddel et al. [6] recently published a method for generating a 2D map from 3D LiDAR data in order to accomplish localization. It operates by putting a verticality constraint on each point based on the slope of the plane in which the point is supposed to be located. It employs two separate thresholds to differentiate between obstacles at a finer level: a smaller one (15 degrees) for recognising navigation hazards and a larger one (80 degrees) for detecting slammable structures. To further limit false detections, obstacles are subjected to two additional constraints. To begin, only points whose z height from the ground falls within a certain range are evaluated. Second, the number of occupied voxels in the vertical line enclosing the point must be more than a certain number. Huesman [7] used the same basic approach of applying slope thresholding to determine barriers to transform a point cloud to a 2D occupancy map. This is one of the ways we recognise misleading hurdles in our own approach.

Beutel et al [8] employed a somewhat different approach based on natural neighbour interpolation to build both 2D and 3D grid maps, the latter of which includes surface elevation information in addition to occupancy probability. Although we were unable to find a way to use the ROS grid mapping library [9] to produce occupancy maps from 3D point clouds, it does contain functions to generate different sorts of 2D grid maps from a number of input sources. The learning-based strategy proposed by Thrun [10], which employs expectation maximisation to produce occupancy maps directly from sensor data, is also inapplicable in our situation.

Unlike the previous methods, which solely employ 3D data, Santana et al. [11] additionally use picture data to separate the scene into floor and non-floor parts by performing color-based visual segmentation. The floor regions of the image are then mapped to free cells in the occupancy map using the homograhy matrix created by SLAM. We considered utilising this strategy to improve our map, but we didn't have enough time. On mobile computing technology, doing image segmentation while also running ORB SLAM and our base grid map creation technique in real time would be extremely difficult. Another option for solving our problem is to transform the point cloud into simulated LiDAR scans using the pointcloud to laserscan ROS module [12] and then feed the generated data into a mapping technique like Gmapping [13] that can generate occupancy maps from LiDAR scans. However, because the mapping algorithm always assumes that the LiDAR data is noisy, and in our

instance, the point cloud created by ORB SLAM is known to contain only reliable points, this method may introduce further uncertainty into the map. As a result, this strategy was not considered.

3 Methodology

3.1 Common Components

This includes the project components and consists of the following tasks:

3.1.1 Review of literature on using vision to build a grid map

This part has been detailed in Sec. 2.

3.1.2 Installation of ORB-SLAM

This stage involved following the instructions on the Raulmur page [14] to install ORB-SLAM2. On ROS-Noetic, the current version of ORB-SLAM2 has some issues with Eigen 3.3.3 and OpenCV 3. As a result, we used Eigen 3.2.10 instead of OpenCV 3 and used the OpenCV 2.4 included with ROS-Noetic.

3.1.3 Calibration of camera

To calibrate the ROS web cam that we utilised to generate our testing sequence, we followed the procedures described in the ROS camera calibration tutorial [15]. (Sec. 3.1.7). Table 1 shows the camera parameters that were derived in this way.

Table 1: Camera Calibration Parameters

Camera Matrix	**fx**	**fy**	**cx**	**cy**	
	642.994934	647.678101	315.938509	235.397152	
Distortion Coefficients	**k1**	**k2**	**p1**	**p2**	**k3**
	-0.084841	0.041748	-0.007377	0.004092	0.000000

3.1.4 Reproduction of results on KITTI and TUM

We began by downloading sequences 00 and 05 from the KITTI dataset [16] and sequence fr3 walking half sphere from the TUM dataset [17] for this section. To replicate the results, we performed ORB-SLAM on all three sequences, following the directions on the Raulmur page [14]. Figure 1 shows screen pictures of the created point clouds. For ORB SLAM on the three sequences, the following commands were used:

```
./Examples/Monocular/mono_kitti Vocabulary/ORBvoc.txt Examples/Monocular/KITTI00-02.yaml ./KITTI/00
./Examples/Monocular/mono_kitti Vocabulary/ORBvoc.txt Examples/Monocular/KITTI04-12.yaml ./KITTI/05
./Examples/Monocular/mono_tum Vocabulary/ORBvoc.txt Examples/Monocular/TUM3.yaml
        ./TUM-RGBD/rgbd_dataset_freiburg3_walking_halfsphere
```

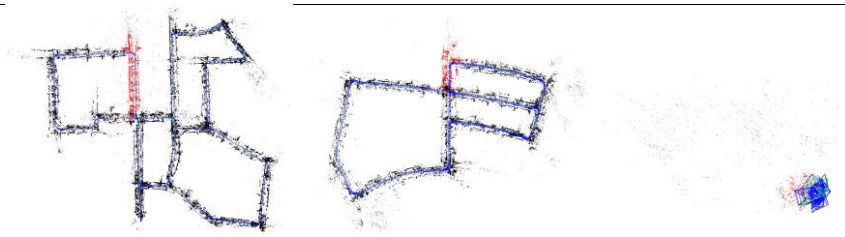

Figure 1: Screenshots of point clouds on datasets: from left to right: KITTI 00, KITTI 05 and TUM fr3 walking half-sphere

3.1.5 2D grid map generation using ORB map points

For this phase, we wrote a Python script that processed all of the keyframes and map points produced by ORB SLAM after it finished analysing all of the frames in the sequence to generate a 2D occupancy grid map off line. This script is included with this report as pointCloudToGridMap2D.py.

We changed the ORB SLAM monocular application Examples/Monocular/mono tum to output the 3D poses of all keyframes as well as all map points visible in each keyframe to a text file in order to provide the input data required by this script. After projecting them to the XZ plane, the script parses this file and records all keyframe/camera positions and related map points in a dictionary structure. The y coordinate is simply removed in this projection. Because the ORB SLAM coordinate locations are in metres, a finer grid resolution is achieved by multiplying all positions by a scaling factor that is selected as the inverse of the required m/cell resolution. If a resolution of 10 cm/cell or 0.1 m/cell is required, for example, the scaling factor becomes 10. The keyframes are then processed one by one, with the following procedures applied to all visible map points in each keyframe:

1. Using the Bressenham's line drawing algorithm [18], cast a ray from the camera position to all visible spots.

2. For each point along the ray, increment a visit counter, and for the end point that corresponds to the location of the map point, increment an occupied counter.

After scaling, the visit and occupied counters are kept as integral arrays of the same size as the camera and map points' range of x and z positions. Because ORB SLAM assumes the XZ plane to be the horizontal plane, a point's y coordinate is considered its height. The occupancy probability for each cell of the grid map is estimated after all keyframes have been processed as follows:

$$p_{free}(i,j) = 1 - \frac{occupied(i,j)}{visit(i,j)} \quad (1)$$

The equivalent entries in the occupied and visit counters are occupied(i, j) and visit(i, j), respectively, and pfree is the probability that this cell is not occupied. This probability map is converted to a ternary cost map by employing two thresholds, free thresh and occupied thresh, such that a cell is deemed free if its pfree is larger than free thresh, occupied if it is less than occupied thresh, and unknown if its pfree is less than occupied thresh.

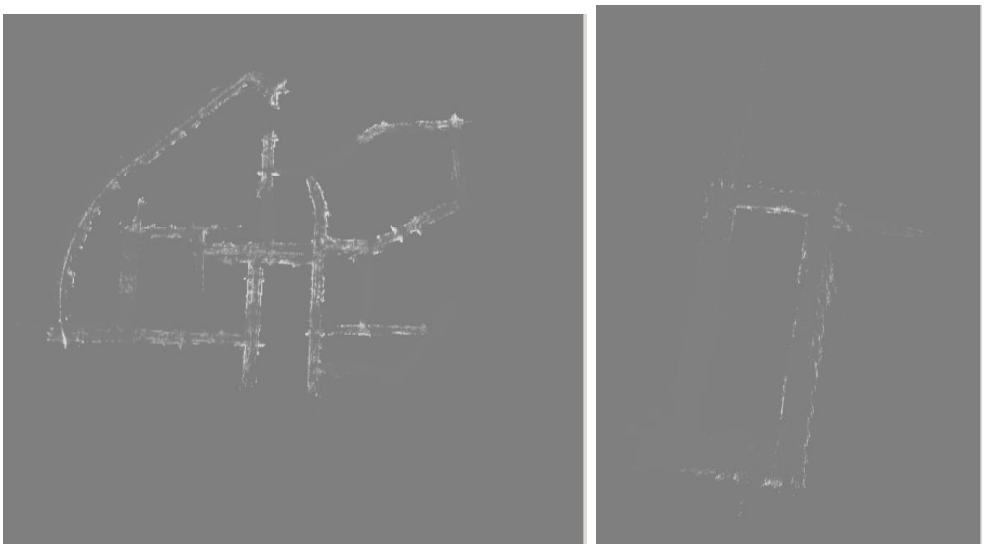

Figure 2: Probability maps before thresholding for to KITTI 00 sequence (left) and CSC first floor sequence (right) (Sec. 3.1.7)

3.1.6 Map visualization and robot navigation in Rviz

We use simulation instead of a genuine Turtlebot robot because we don't have access to one. First, we use the Gazebo [19] simulator to build a virtual Turtlebot in an empty world, and then we employ adaptive Monte Carlo localization (AMCL) [20, 21] for navigation and Rviz [22] for visualisation. These nodes are run using the following commands:

```
roslaunch turtlebot_gazebo turtlebot_world.launch
        world_file:=/opt/ros/indigo/share/turtlebot_gazebo/worlds/empty.world
roslaunch turtlebot_gazebo amcl_demo.launch map_file:=./grid_map.yaml roslaunch
turtlebot_rviz_launchers view_navigation.launch
```

Fig. 3 shows screen shots of the map generated by the Python script and the navigation path produced by AMCL as visualized in Rviz.

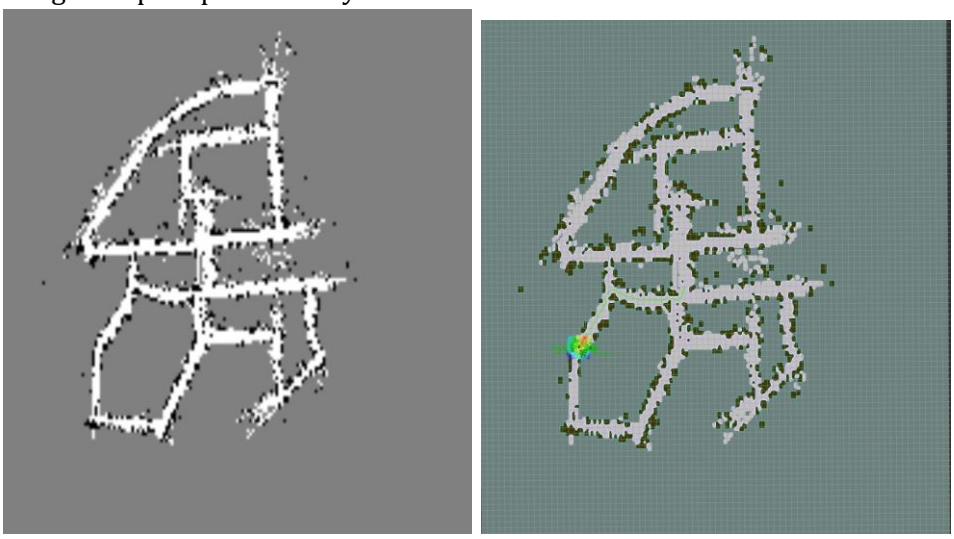

Figure 3: Map visualization and robot navigation

3.1.7 Evaluation and comparison of the result with that found in the literature

We couldn't discover any previous work in the literature that used only 3D points generated by a visual SLAM approach like ORB SLAM to create a 2D grid map. As a result, we needed LiDAR data for our datasets in order to compare them to previous results, which we also lacked. As a result, we used the calibrated camera on the first floor of the CSC building to record our own sequence in order to evaluate our work. Figure 4 depicts a few frames from this sequence together with the matching ORB SLAM point cloud.

Next, we used the floor plan to generate the ground truth map for this sequence, and we evaluated it using two metrics: an accuracy measure and a completeness score. The accuracy score indicates how well our map matches the ground truth map, and the completeness score indicates how much of the true map our algorithm was able to generate, regardless of whether it was correct. Before the two maps can be compared, they must first be aligned,

which can be accomplished by calculating their homography. The evaluation metrics are then calculated using the following formulas:

$$\text{Completeness} = \frac{\text{number of known cells in the map}}{\text{total number of cells in the map}} \quad (2)$$

$$\text{Accuracy} = \frac{\text{number of correct known cells in the map}}{\text{total number of known cells in the map}} \quad (3)$$

3.2 Novel Components

We used the data from the previous section to produce a grid map in real time and utilise it for navigation to follow the camera trajectory in this section. The following steps were taken as part of this process:

3.2.1 Online variant

Because it is not possible to process all keyframes and map points obtained up to any given point in the sequence in real time, the Python code was rewritten to C++ and customised to work in an incremental way. This required changing the monocular ROS node Examples/ROS/ORB SLAM2/Mono to create two ROS nodes. The first node, Monopub, publishes the posture of each keyframe, as well as all map points visible in that keyframe, once it is uploaded to the map. It also recognises when a loop is closed and subsequently publishes all keyframes as well as all map points that have been added so far. In addition to reading photos from disc, Monopub was enhanced to take input images from live cameras and ROS topics. To run this node on the KITTI 00 sequence, TUM frg3 walking halfsphere sequence, live camera, and a ROS node publishing photos to /usb cam/image raw topic, use the following four commands:

```
rosrun ORB_SLAM2 Monopub Vocabulary/ORBvoc.txt Examples/Monocular/KITTI00-02.yaml ./KITTI/00 0 rosrun ORB_SLAM2 Monopub
Vocabulary/ORBvoc.txt Examples/Monocular/TUM3.yaml
        ./TUM-RGBD/rgbd_dataset_freiburg3_walking_halfsphere rosrun ORB_SLAM2 Monopub Vocabulary/ORBvoc.txt
Examples/Monocular/mono.yaml 0 rosrun ORB_SLAM2 Monopub Vocabulary/ORBvoc.txt
Examples/Monocular/demo_cam.yaml -1 /usb_cam/image_raw
```

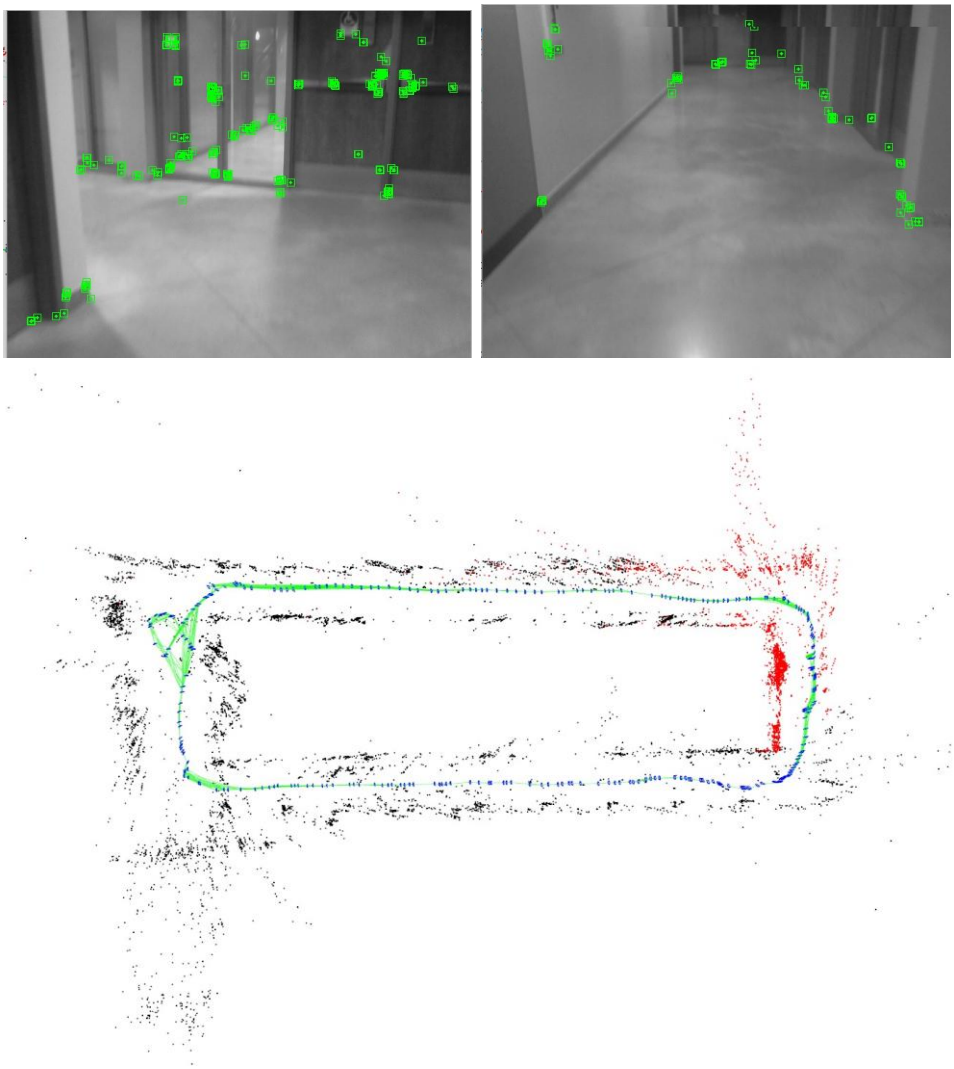

Figure 4: Two frames from the CSC sequence showing the ORB feature points and the point cloud data produced on the sequence

Monosub is the second node, which subscribes to the pose data published by Monopub. When it receives a single keyframe, it processes it in the same way as in Sec. 3.1.5, but with a few more tricks that are explained in the following sections. To create the map progressively, the visit and occupied counters belonging to each such keyframe are combined together. The map thus produced is only an approximation to the genuine map because the counters are updated independently for each keyframe and co-visibility across distinct keyframes is not taken into account. However, in practise, it proved to be quite precise and more than adequate for navigation.

When all keyframes have been received following a loop closure, the counters are reset and recalculated using all keyframes and map points received from the publisher at the same time. This enables us to deal with minor inaccuracies that have built up over time. For instances where loop closures are too few and far between, Monopub also offers the option of publishing all keyframes and points on a regular basis even if no loop closure is detected. Map resetting is a time-consuming process, but it is done infrequently enough that it isn't a problem.

Monosub makes the created map and associated meta data, as well as the camera trajectories, available for use in AMCL and Rviz as navigation goals. Monosub accepts a huge number of command line inputs, which are given below, due to the various settings that can be modified

```
rosrun ORB_SLAM2 Monosub <scale_factor> <resize_factor> <cloud_max_x> <cloud_min_x>
        <cloud_max_z> <cloud_min_z> <free_thresh> <occupied_thresh> <use_local_counters>
        <visit_thresh> <use_gaussian_counters> <use_boundary_detection>
                        <use_height_thresholding> <normal_thresh_deg> <canny_thresh> <enable_goal_publishing>
        <show_camera_location> <gauss_kernel_size>
```

For example, following commands can be used to run the two nodes on the KITTI 00 and floor sequences respectively with different parameters fine tuned for each:

```
rosrun ORB_SLAM2 Monosub 10 1 29 -25 48 -12 0.55 0.50 1 5 1 0 1 75 350 rosrun ORB_SLAM2 Monosub 30 1 10 -10
22 -12 0.45 0.40 1 10 1 1 1 30 400
```

The first command's resolution is 1/10 m/cell, whereas the second command's resolution is 1/30 m/cell. This node also enables for parameter adjustments during runtime via key strokes, which are detailed in the Monopub source file's showGridMap function. This report includes the code for both nodes as well as the modified version of ORB SLAM required to run them. The publisher and subscriber's source files are referred to

- Examples/ROS/ORB SLAM2/src/ros monopub.cc and
- Examples/ROS/ORB SLAM2/src/ros monosub.cc respectively.

3.2.2 Local and global counters

The concept of local and global counters stems from a problem with the initial stage in ray casting, which is a basic 2D projection of 3D points into the XZ plane. Consider a case in which many points' projections in 2D are co-linear in a single keyframe (e.g. Fig. 5]). If we simply utilise one global counter to generate the 2D map in this scenario, we are misrepresenting the available data. The central point in Fig. 5 is occupied, but we're decreasing the occupied ratio for all the places along this line by incrementing their visit counter. This

means that if we solely use global counts, we'll wind up with some occupied cells being replaced with empty ones. With our CSC sequence, Fig. 6 shows the difference between utilising and not using local counters. If local counters are not employed, a lot of truly occupied spots are replaced with free space, as we can see.

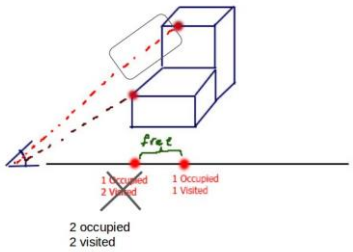

Figure 5: Issue of corrupting counters when points are co-linear in 2D

3.2.3 Visit thresholding

We've utilised a cell's total visit count as a measure of confidence, and we only make a grid cell occupied or vacant if its total visit count exceeds a threshold. This helps us to reduce outliers in ORB SLAM caused by incorrect triangulations, especially those that are far away from the camera. Figure 7 shows a comparison of maps created with various visit criteria. For our experiments, we employed a default visit threshold of 5.

3.2.4 Height thresholding

Only points that exist within a defined range of y-height above the XZ plane are allowed as barriers, similar to the technique employed by Goeddel et al [6] (Sec. 2). As a result, we reverted all points to the camera coordinate frame and set a height threshold for them. Points with y coordinates less than this threshold are presumed to be on the floor/road, and rather than deleting them or counting the related cells as occupied, we treat them as a vacant area by just raising their visit counter. The maps obtained with and without height thresholding are shown in Fig. 8. As can be seen, employing height thresholding, several of the occupied cells on the floor have been converted to free space.

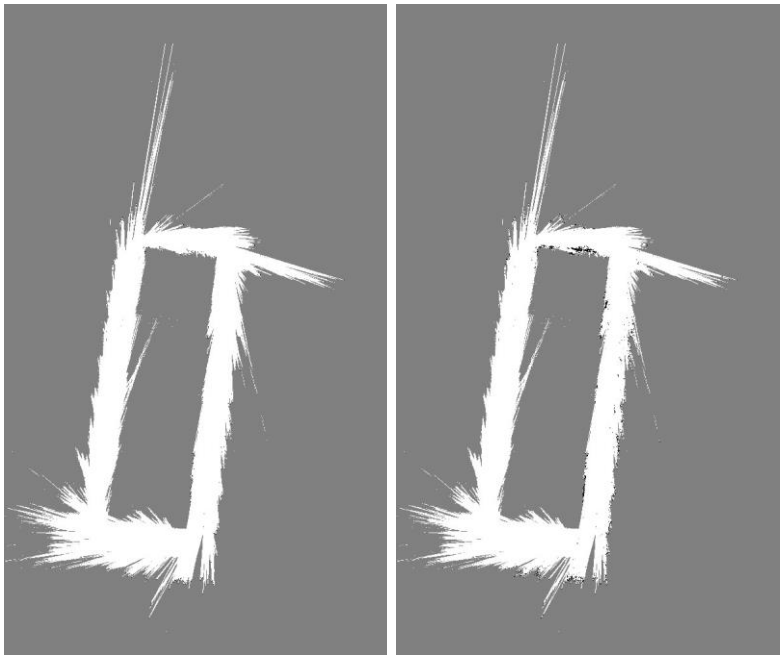

Figure 6: Effect of local counters on map quality. The map on the left was generated without local counters while the one on the right was generated with local counters.

3.2.5 Gaussian smoothing of counters

Because space is not discrete, when two cells separated by a small distance are occupied, the cells between them are very likely to be occupied as well. These, on the other hand, may be labelled as unknown, resulting in "holes" between normally contiguous free or occupied cell sections. To address this issue, we apply Gaussian smoothing to the local counters (both visit and occupied counters) in the manner of the probability field model. This causes the value in any particular cell to influence its neighbouring cells as well, resulting in any empty cells in both counters taking on values identical to their close non-empty cells. As a result, transitions within the probability map become smoother, and occupied and free cells have an impact on their surroundings. The effect of varying widths of the Gaussian kernel used for smoothing is seen in Fig. 9. We chose a kernel size of 3 as the default for our studies based on these findings.

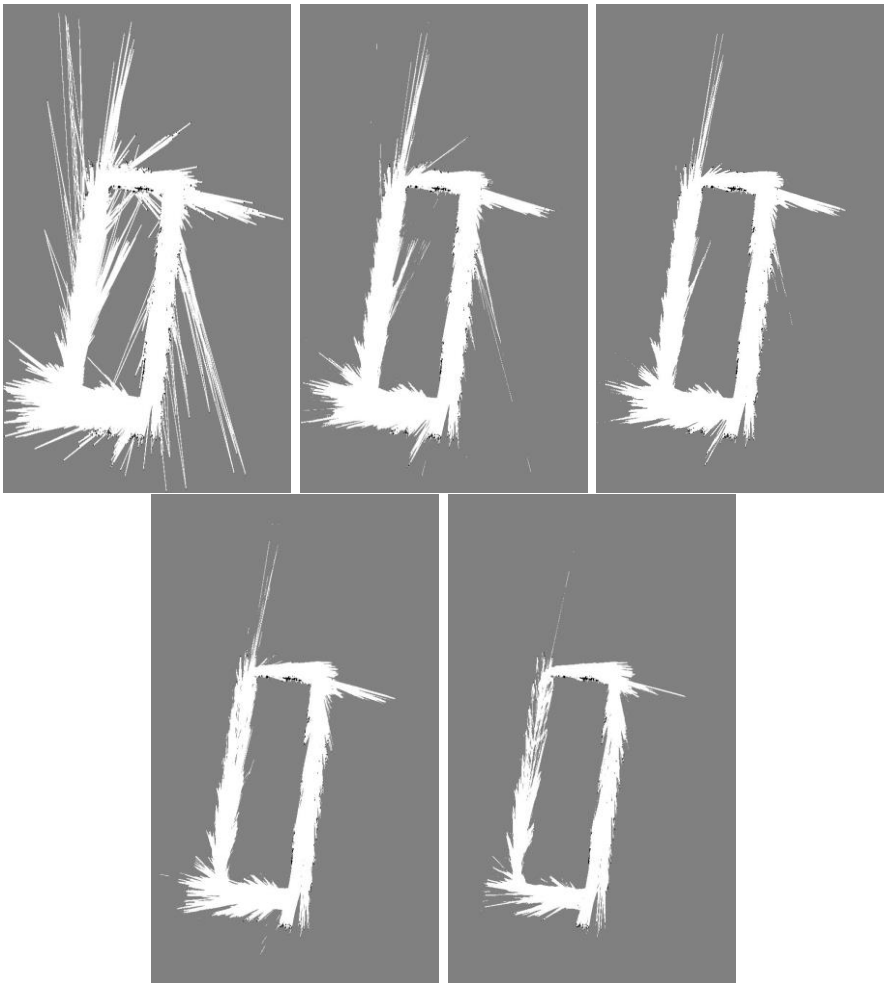

Figure 7: Comparison of different visit thresholds: from left to right and top to bottom - 0, 3, 5, 10, 20, 40

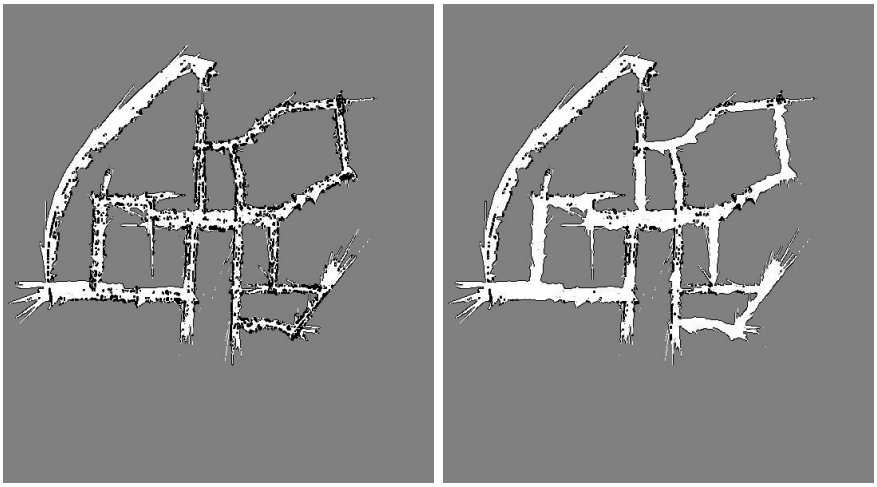

Figure 8: Effect of height thresholding on KITTI 00 sequence. In the left image, height thresholding has been disabled while the right one shows the result in the presence of the height thresholding

3.2.6 Canny boundary detection

Because the majority of the tactics we've used are intended to remove bogus hurdles, one issue that arises is that too many actual impediments are also removed. We noticed that the majority of the removed impediments were on the edge of the free and unknown zones. As a result, marking the boundary pixels between the two areas as barriers is an easy approach to bring them back. To identify these pixels as occupied, we used Canny edge detection to find them and then set all pixels corresponding to the observed edges to 0 in the probability map. The raw output of canny contour is shown in Figure 10 along with a comparison of maps created using different thresholds. As can be seen, this rather basic strategy not only results in a more accurate map, but also in better navigation paths that avoid getting too close to unknown locations.

With the upper threshold set to twice the lower one, Fig. 11 shows the influence of the lower Canny threshold on the quality of the created map. For our tests, we employed a default Canny threshold of 350, which was found to remove the most erroneous edges in the interior of the free areas without losing any of the necessary outer limits.

3.2.7 Slope thresholding

This strategy is comparable to Goeddel et al. [6] (Sec. 2), who do not consider points that are judged to lie on nearly horizontal planes to be impediments.

Finding the two points closest to a point and computing the best fit plane including these three points in the least squares sense is how a point's plane is assessed.

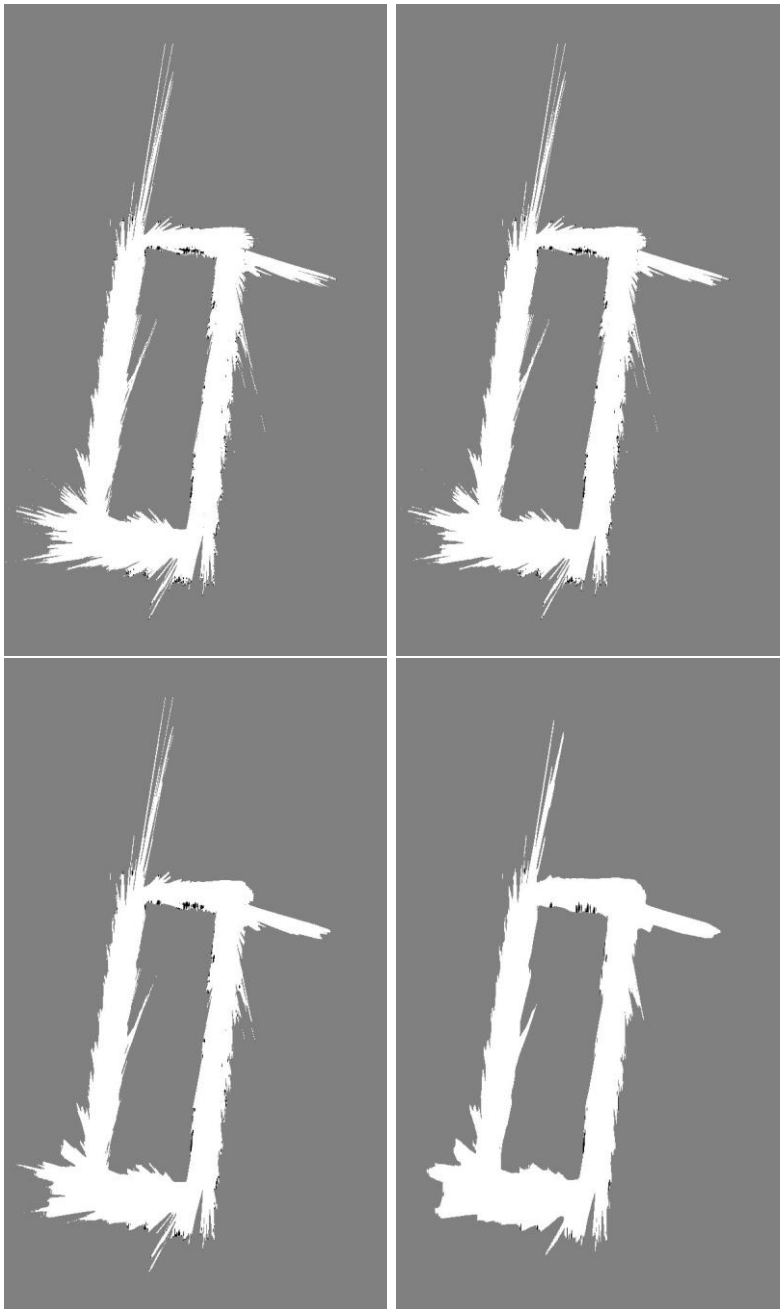

Figure 9: Maps generated using Gaussian smoothing of counters with different kernel sizes: left to right and top to bottom - 3, 5, 10 and 30

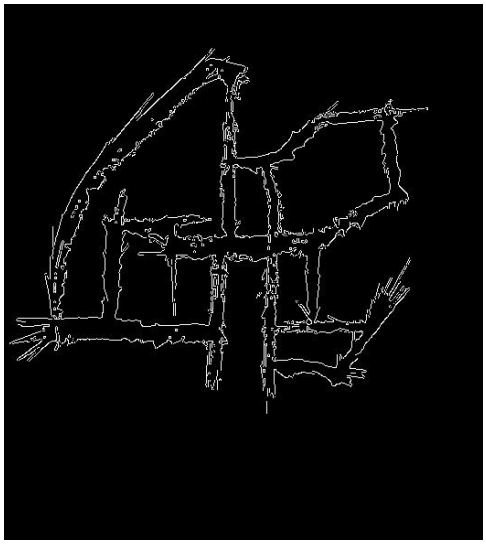

Figure 10: An example of finding contours

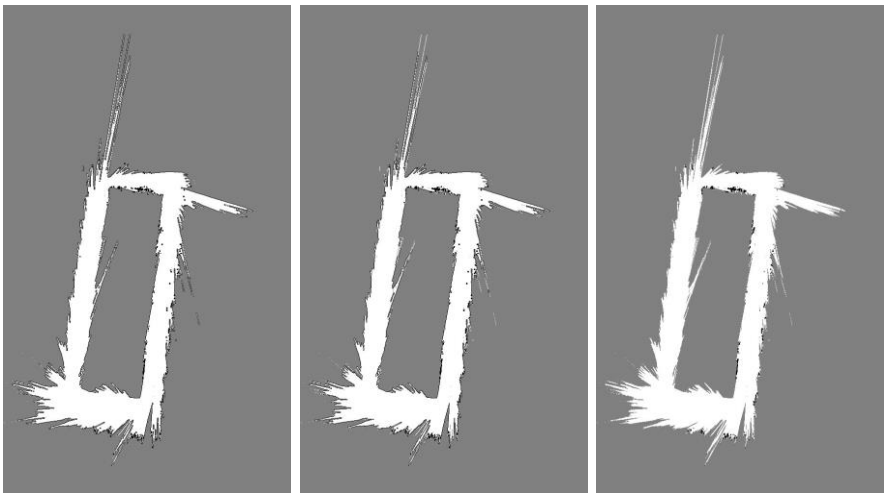

Figure 11: Comparison of maps generated by using different thresholds for Canny boundary detection: from left to right - 200, 350, 700

This plane is deemed horizontal and the point corresponding to a free cell if the angle formed by the normal to this plane with the XZ plane exceeds a threshold. In our tests, we discovered that a threshold of 75 degrees produced

good results. To quickly discover the closest points, the FLANN library [23] was utilised, and SVD was used to estimate the best fit plane.

4 Results

Figure 12 shows a few screen pictures from that demo, including the grid map for the KITTI 00 sequence and the Rviz navigation result. The CSC sequence's results are shown in Fig. 13.

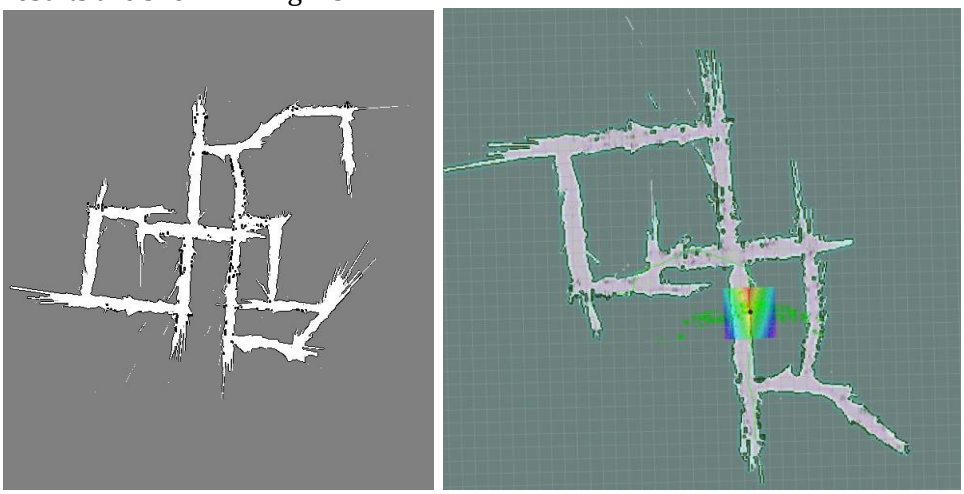

Figure 12: Online occupancy grid map and navigation on KITTI OO

To get the launch files used in Sec. 3.1.6 to operate with Monosub's online map and goals, we had to make some changes. The changed launch files, amcl demo.launch and view navigation.launch, are included in the ORB SLAM root. To start the autonomous navigation, use the following commands:

```
roslaunch turtlebot_gazebo turtlebot_world.launch world_file:=/opt/ros/indigo/share/turtlebot_gazebo/worlds/empty.world
roslaunch amcl_demo.launch roslaunch
view_navigation.launch
<run Monopub>
<run Monosub with enable_goal_publishing set to 1 for automatic goal setting and 0 for manual goal selection in Rviz>
```

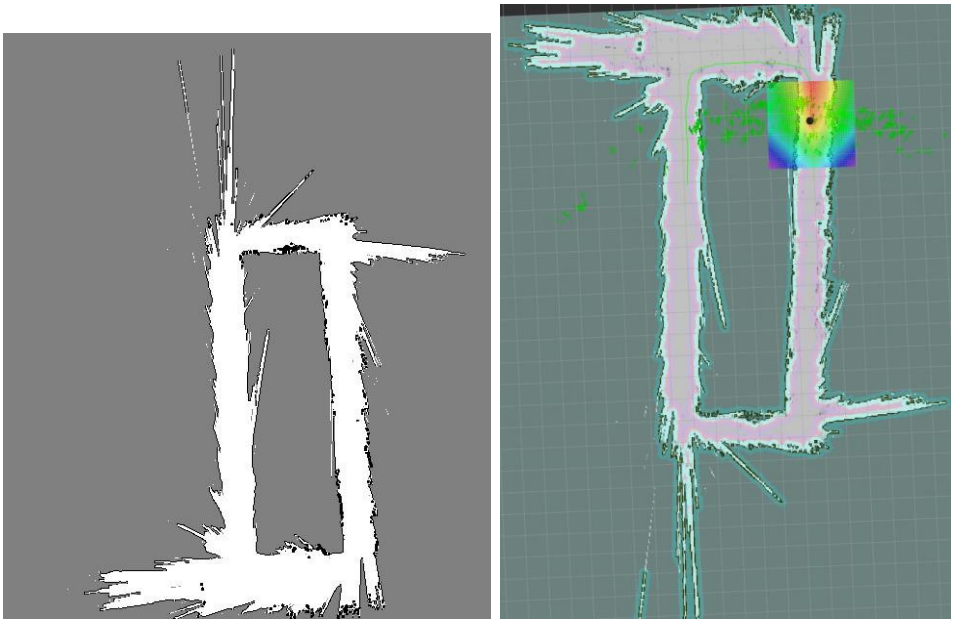

Figure 13: Online occupancy grid map and navigation on the CSC sequence

5 Conclusion

We studied various methods for creating a 2D occupancy grid map from the sparse 3D map provided by ORB-SLAM in this study. To discover the best map, we used a variety of techniques such as local counters, visit thresholding, Gaussian smoothing of counters, Canny border detection, height thresholding, and slope thresholding. We used the resulting map to navigate a simulated Turtlebot and found it to be extremely effective.

References

[1] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. J. Leonard, "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age," *Trans. Rob.*, vol. 32, no. 6, pp. 1309–1332, Dec. 2016. [Online]. Available: https://doi.org/10.1109/TRO.2016.2624754

[2] M. J. M. M. Mur-Artal, Rau´l and J. D. Tard´os, "ORB-SLAM: a versatile and accurate monocular SLAM system," *IEEE Transactions on Robotics*, vol. 31, no. 5, pp. 1147–1163, 2015.

[3] R. Mur-Artal and J. D. Tard´os, “ORB-SLAM2: An open-source SLAM system for monocular, stereo and RGB-D cameras,” *arXiv preprint arXiv:1610.06475*, 2016.

[4] A. Elfes, “Occupancy grids: A probabilistic framework for robot perception and navigation,” Ph.D. dissertation, Pittsburgh, PA, USA, 1989, aAI9006205.

[5] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “ROS: an open-source Robot Operating System,” in *ICRA Workshop on Open Source Software*, 2009.

[6] R. Goeddel, C. Kershaw, J. Serafin, and E. Olson, “Flat2d: Fast localization from approximate transformation into 2d,” in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct 2016, pp. 1932–1939.

[7] J. Huesman, “Converting 3D Point Cloud Data into 2D Occupancy Grids suitable for Robot Applications,” Online: https://library.ndsu.edu/repository/handle/10365/25535.

[8] A. Beutel, T. Mølhave, and P. K. Agarwal, “Natural neighbor interpolation based grid dem construction using a gpu,” in *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ser. GIS ’10. New York, NY, USA: ACM, 2010, pp. 172–181. [Online]. Available: http://doi.acm.org/10.1145/1869790.1869817

[9] P. Fankhauser and M. Hutter, *A Universal Grid Map Library: Implementation and Use Case for Rough Terrain Navigation*. Cham: Springer International Publishing, 2016, pp. 99–120. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-26054-9 5

[10] S. Thrun, “Learning occupancy grid maps with forward sensor models,” *Autonomous Robots*, vol. 15, no. 2, pp. 111–127, 2003.

[11] A. M. Santana, K. R. Aires, R. M. Veras, and A. A. Medeiros, “An approach for 2d visual occupancy grid map using monocular vision,” *Electronic Notes in Theoretical Computer Science*, vol. 281, pp. 175 – 191, 2011.

[12] P. Bovbel and T. Foote, “pointcloud to laserscan,” Online: http://wiki.ros.org/pointcloud to laserscan.

[13] G. Grisetti, C. Stachniss, and W. Burgard, "Improved techniques for grid mapping with rao-blackwellized particle filters," *IEEE Transactions on Robotics*, vol. 23, no. 1, pp. 34–46, Feb 2007.

[14] J. M. M. M. Raul Mur-Artal, Juan D. Tardos and D. Galvez-Lopez, "Orb-slam2," Online: https://github.com/raulmur/ORB SLAM2.

[15] "How to calibrate a monocular camera," Online: http://wiki.ros.org/camera calibration/Tutorials/MonocularCalibration.

[16] 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.

[17] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers, "A benchmark for the evaluation of rgb-d slam systems," in *Proc. of the International Conference on Intelligent Robot Systems (IROS)*, Oct. 2012.

[18] "The bresenham line-drawing algorithm," Online: https://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html.

[19] "gazebo," Online: http://wiki.ros.org/gazebo.

[20] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[21] "amcl," Online: http://wiki.ros.org/amcl.

[22] "rviz," Online: http://wiki.ros.org/rviz.

[23] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *International Conference on Computer Vision Theory and Application VISSAPP'09)*. INSTICC Press, 2009, pp. 331–340.