Autonomous Robot Patrol System

Final Report

Computer Vision Master Project 2018/2019

Fabian Behrendt, Chris Carstensen,
Christian Reichel, Nicolás Pérez de Olaguer, Caus Danu, Nana Baah

27th of September 2019

Abstract

Recent advancements in artificial intelligence offer the possibility to create autonomous and decentralized systems with a wide area of applications. Surveillance is one such area of interest that has unfortunate connotations with ethical concerns. With this work we try to leverage the recent progress in robotics and computer vision to implement useful and ethical surveillance systems, namely in the interest of guarding public institutions more efficiently against potential intruders and making the jobs of human security guards more comfortable. Our system is intended as an extra layer of security to already existing state of the art methods, such as static cameras and sonar systems.

1 Introduction

Surveillance systems are still heavily relying on human input and control. Nowadays, many static, camera based architectures exist and not so much decentralised, dynamic solutions. With the help of mobile robots and modern computer vision systems, this problem can be mitigated to some extent. The appearance of diverse object-detection algorithms makes control and monitoring much more responsive. Among the uprising deep learning techniques, convolutional neural networks (CNN) have been proven to be especially successful in classifying image-based systems. Moreover, sophisticated object detectors, also based on CNNs have emerged. *Regions with CNN features* (R-CNN) [1] and its successors [2, 3, 4] refine the object detection problem by being less computer intensive and more accurate. The main idea under the object-detector-like networks is to choose regions of interest (RoI) of same size and compute the probability that a certain object appears in the region. Therefore, finding objects with CNNs would help detect possible intruders or anomalies in a specific environment. In this work, we propose a surveillance system composed of a robot capable of moving around an unknown environment using Simultaneous Localisation and Mapping (SLAM) [5] and a deep learning network "You Only Look Once" (also known as YOLO) [4].

2 Task Description

The main task of the robot is to navigate through one of the floors within the University of Hamburg, without prior information of it, and detect when a door is open. This is regarded as an anomaly, since doors are always closed and locked during off-hours. Therefore, the robot guards the floor when there is no one around and reports to a web portal as to what is the status of the floor and whether an intruder might have potentially tempered with the scene. If some signs of intrusion are present, enough to raise suspicion, a human guard is alerted by an online tool and can then inspect more thoroughly the area in question.

3 System Description

Accomplishing the described task requires hardware and software components. This section gives a brief overview of all components of the developed

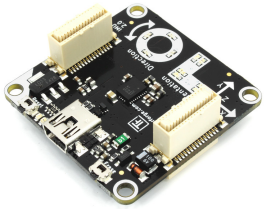
system. First, the used Hardware and thereafter all software components will be described.

3.1 Used Hardware

Our agent is surprisingly human. The plain robot can be viewed as the bare-bones body, obeying the "brain" commands and giving feedback from its encoders to the controller. A laptop is placed on the robot surface, which acts as the brain of the agent, executing all the algorithms and logic. The Kinect camera is suspended on a vertical metallic bar, attached to the robot body and acts as the eyes of the agent, the most important sensor, perceiving color, as well as depth. Finally, the IMU is placed below the camera and acts as the cerebellum, which is very important for spatial orientation.

(a) Pioneer P2 DX Robot

(b) Kinect Camera (Version 2)



(c) TinkerForge IMU Brick 2.0

Figure 1: Main hardware components

3.2 Used Software

A complex system is defined by a clever combination of simpler building blocks. We make use of the following nodes provided by the ROS ecosystem to solve the navigation task:

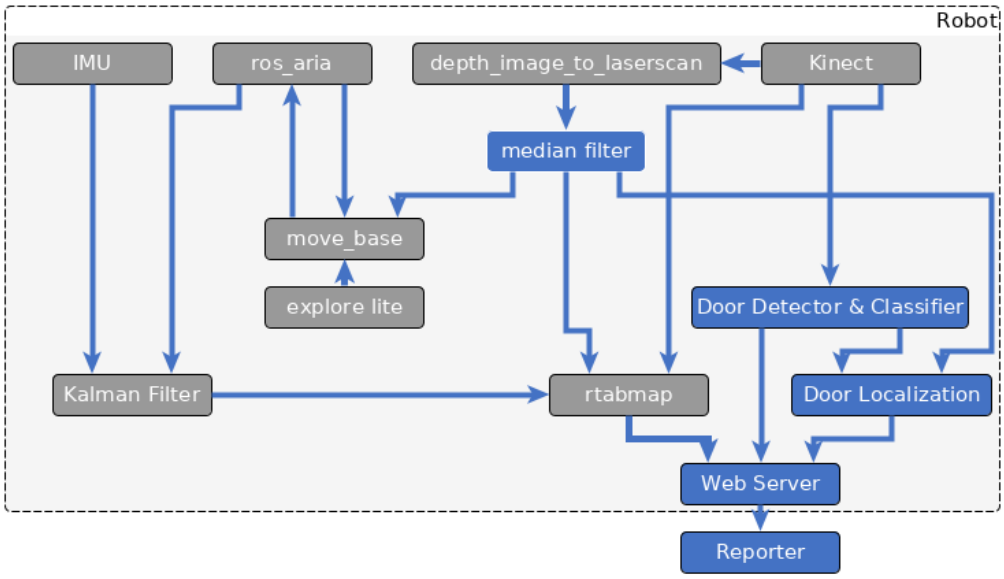


Figure 2: Overview of the software components
The blue components denote all self developed ROS nodes during this project. The gray nodes represents ROS nodes, which we took from the community and configured to our needs.

- **depthimage_to_laserscan** to emulate a laser from depth data [6].
- **movebase** to interact with the navigation stack [7].
- **rtabmap** as our visual SLAM solution [8].
- **robot_localization** to make use of the Extended Kalman Filter [9].
- **explore_lite** to efficiently sample unknown environment [10].
- **mongodb_store** to store necessary information while navigating as part of the Door Localization [11].
- **Kinect Bridge** to access the images provided by the Kinect via the ROS framework [12].

The main task is to correctly connect, configure and fine-tune all these blocks such that they result in the expected outcome in terms of navigation and behaviour of the robot.

4 Simultaneous Localization and Mapping

One crucial component of our system is the Simultaneous Localization and Mapping (SLAM) system. It enables the robot to localize itself in its environment and generate a map while doing so. All of this is done by collecting measurement data from different sensors available. The detected doors can be placed as landmarks on the generated map. This section gives a brief theoretical background about the SLAM system used and some implementation details specific to our project.

4.1 Theoretical Background

Typical SLAM systems use LIDAR (Light Detection and Ranging) and/or RGB(D) cameras to perceive the environment. Optionally, inertial measurement units (a.k.a. IMU) and wheel encoders are used to estimate the movement of the robot itself. With these sensors there are two major approaches to the SLAM problem. The first approach aligns the measurement of the LIDAR to the map. This can be done with an iterative closest point algorithm (ICP) [13]. Such an approach has the advantage that a map is generated directly from the SLAM algorithm, but is quite computationally expensive.

The second approach computes landmarks from measurements. In an RGB image taken by a camera, this could be features like ORB-Features [14], while in a LIDAR scan it can be "L" shapes for example. These features are now considered by the SLAM system as static objects in the environment represented through position, orientation and scale. Once a new measurement observes the same landmarks plus some new landmarks, the overall map of landmarks can be updated and extended. In addition, the robot can calculate its own position, since the viewpoint towards the landmarks is also known. Often, these landmark based SLAM systems are represented internally as graphs, whose nodes store relevant information related to the landmarks that were perceived, like their position for instance, while the edges encode the relative translation and rotation between these nodes.

Since measurements are generally noisy, we cannot align the landmarks from one node with the landmarks from the other node perfectly, and the error accumulates over time. This can be observed, when the robot drives along a lengthy loop and should be ending up where it started, but, according to the generated map - it doesn't. In this situation, another method, namely

"loop closure" can detect that the robot has already visited the place in the past and trigger the necessary corrections. Often, this is done using a Bag of Words (BoW) containing the visual features used from the landmarks [15]. If the loop closure method detects a closure, the total error within the loop can be calculated and the positions of the nodes and landmarks get updated using the least squares method. Examples of this landmark SLAM are [16] or [8].

4.2 Implementation

For this project, we used the **rtabmap** ROS node [8] as our SLAM algorithm. This node provides an occupancy grid map out of the box. As stated in Section 3 - "System Description", we make use of a Kinect camera for capturing the color and depth pointcloud, as well as an IMU and the Pioneer 2 robot, which has wheel encoders available that count the number of rotations each wheel makes. These sensors serve data inputs to our SLAM system, which can be seen in Figure 3.

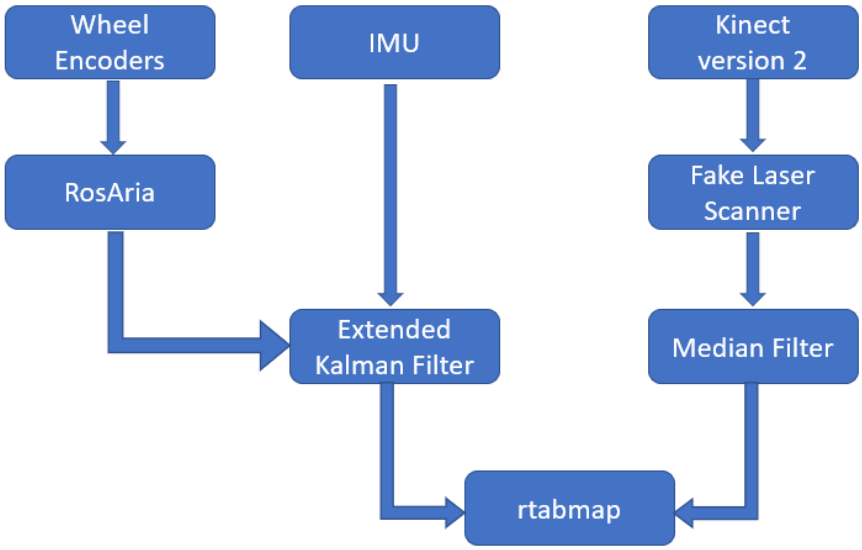
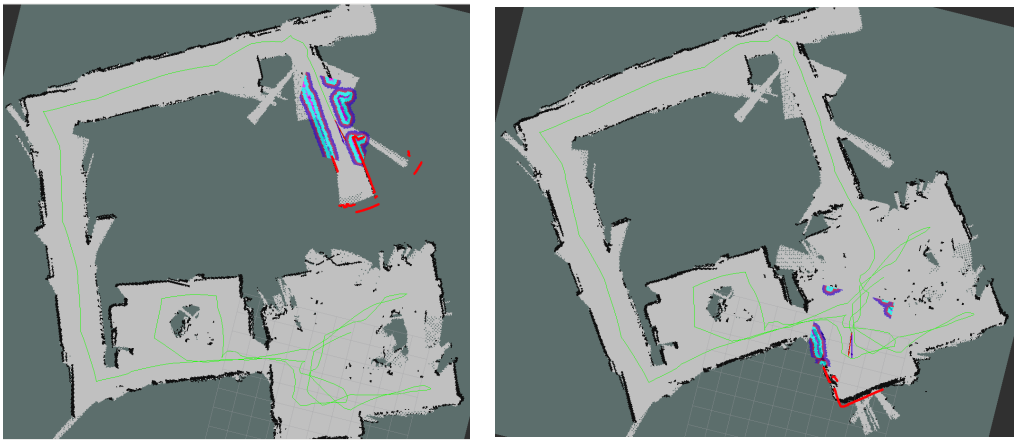


Figure 3: SLAM setup

It is noteworthy that firstly - rtabmap is not capable of processing IMU

data directly and secondly - using the full depth image results in an unacceptably slow update rate of the map. These facts lead to overall bad map quality. To compensate for the update frequency issue, we reduced the RGB-D images from the Kinect to a laser scan using a depth image to laser scan node [6]. Basically, this node computes a fake laser by slicing the point cloud in a 3D pyramid and subsequently projecting its points onto a thin plane sectioning the camera and being parallel to the ground.

By running rtabmap with this laser scan and the odometry provided by the wheel encoders, the robot was capable of mapping and locating itself in a room. However, it was loosing quite often track of its position and was not capable of moving through doors. The latter issue was mainly due to noisy fake depth laser scan, which produced random artifacts and imaginary obstacles in front of the robot. To get rid of most of the noise in the laser scan, we implemented a median filter ourselves, which is well known for removing "salt and pepper noise". With this component plugged in, the robot was capable of moving through doors and then complete a loop spanning multiple, successive rooms and corridors.



(a) Before Loop Closure (b) After Loop Closure

Figure 4: Effect of Loop closure (Both images show the computer vision lab, as well as the corridors)

As Figure 4 shows, the error in the odometry of the wheel encoders has accumulated and causes a significant drift, but the loop closure manages at least to connect the extremes of the route. However, the drift is simply too large to be able to use the map for exploration purposes. One would have to

place the goals/destinations factoring in the potential unknown deviations, which is clearly not desirable.

To cope with the problem at hand, we integrated an IMU chip. As stated above, rtabmap cannot process IMU data directly. Hence we had to use an Extended Kalman Filter to fuse the IMU and the wheel odometry data into a resulting filtered odometry. The enhancement of the overall pose estimation and the effect on the SLAM performance is described in Section 9.2.1.

5 Autonomous Exploration

In order to not have to move the robot around manually to create a map of its environment, we make use of autonomous exploration. It allows the robot to explore surroundings on its own with the support of our SLAM system (which is described in Section 4). This section explains the theory of such autonomous exploration algorithms in a concise manner and provides information on our approach to integrate it into our system.

5.1 Theoretical Background

To autonomously explore an unknown area, the robot needs to build a map of its environment. This map consists of cells, where each one has a specific state denoting whether it is occupied, open or unknown. Occupied means the cell contains an obstacle, so the robot is unable to move there, while open space is defined with cells through which the robot is able to move. Unknown space describes areas which were not perceived by the robot yet.

The key idea is to move to the border between open and unknown space to gather as much new information about the environment as possible. That border is called a frontier. The robot tries to move to a specific point regarding each frontier, which is called the centroid of a frontier. This centroid can be chosen according to different strategies, for example the nearest point at the frontier with respect to the robot, the middle point, or the Cartesian average of the frontier.

Figure 5 shows a simplified example of detecting frontiers based on the perception of a robot at a specific position.

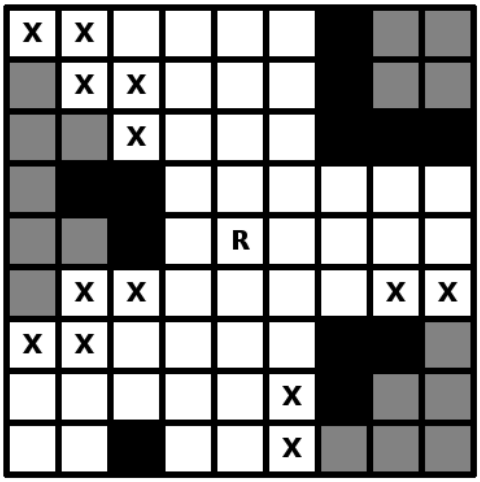


Figure 5: Simplified grid for frontier detection. The robot is represented by the letter R, black cells are occupied, white ones are open and gray ones are unknown cells. Cells marked with an X are frontier cells.

5.2 Implementation

For the implementation of the autonomous exploration procedure we use an already predefined ROS package called **explore_lite** [10], which uses a greedy exploration strategy. It subscribes to a topic providing an occupancy grid map, which is produced either by the **move_base** node [7] or the SLAM system, and computes frontiers from these maps. In our project, we use the occupancy grid map from our SLAM system. Based on these frontiers, it issues movement commands to **move_base** [7] in order to travel to the centroids of the frontiers.

At first, that occupancy grid map was solely built from the output of the Kinect camera. This led to a problem regarding objects that are closer to the camera than its minimum perception range, which is described in Section 9.2.2. To correct that, we incorporate the front sonar of the robot into the generation of the map.

6 Door Detection & Classification

To recognize and report open doors, we developed a door detection algorithm that was subsequently wrapped as a ROS-Node, accepting environment images as input and providing detection results in the form of bounding boxes with confidence scores attached. In the following sections, we will describe the development and the results of the detector in more detail.

6.1 Theoretical Background

To detect doors, we could use the method of Gerard Salton and Michael McGill [17], namely consider vertical edges as potential door candidates and test them against color, texture and other cue differences. This solution is lacking in terms of detecting open doors however. A promising and novel approach is to use the currently successful implementations of trained convolutional neural networks (CNNs) to detect doors in both open and closed states. Lately, CNN based object detectors are able to predict candidates with a speed of 22 images per second and more, all while still being accurate enough for practical purposes. These so called single-shot detectors are using just one CNN to detect, localize and classify multiple objects per image. The Single-Shot MultiBox Detector (SSD) utilizes a set of feature maps on different scales to detect variably sized objects in one image [18]. You Only Look Once (YOLO) on the other hand splits the image into a fixed number of cells and runs the detection and classification of each cell in parallel [4]. Since the first version of YOLO, the model was extended to use clustered default bounding box candidates, and more layers. A lighter custom alternative was also developed (see [19]). The third version adopts different feature map sizes, a larger net, as well as logistic regression that have slowed down the detection to 30 images per second, while improving the detection accuracy (see [20]).

6.2 Implementation

We use an open source Keras implementation of YOLO version 3 by [21]. We have built a fork from this version and merged the pull requests #206, that increases the training result outputs and #262, that uses OpenCV for faster training.

The ROS-Node subscribes to the Kinect image topic and publishes a custom detection message, containing the bounding box information. To enable message synchronization for reporting purposes, it publishes the detection result with the same time stamp as the received input.

6.3 Data Collection, Augmentation & Preprocessing

To make capturing images more convenient, we have implemented a ROS-Node that captured an image every 5 seconds, without any user interaction. The robot was driven using a Bluetooth enabled X-Box joystick around the laboratory during evening time, without any further manipulation of the environment. We captured 376 images. After that we have used the open source project *LabelImg* [22] to mark and classify doors in the images, thus creating ground truth. We have marked the doors including their frames.

We have split the available ground truth into 80% training set, 10% validation set and 10% test set. With this we can perform cross-validation safely, making use of the validation set and never touching the test set, until the final evaluation at the very end.

The data was augmented by resizing images (to satisfy the 608×608 pixel resolution required by YOLO), flipping images and adjusting bounding boxes accordingly, changing the hue, saturation and value five times randomly of the original and flipped versions. In the end we had 11 images created for every single original image, resulting in a total of 4136.

6.4 Training

We have instantiated the network with weights from [20] and trained it further using 608×608 pixel images. Before starting the official training, we have trained the network for 20 epochs with batch sizes of 8, 16 and 32, as well as learning rates of 0.01, 0.001 and 0.0001. With a batch size of 32 and a learning rate of 0.001, we have achieved a stable and fast conversion of the loss and validation errors close to zero. For stable training, we have frozen all but the last two layers of the network and trained them for 50 epochs. Subsequently, all layers were unfrozen and the batch size was reduced to 8 due to memory limitations and learning rate was decreased to 0.0001 to ensure conversion towards minimum loss. Cross-validation was performed using the dedicated validation set.

6.5 Evaluation

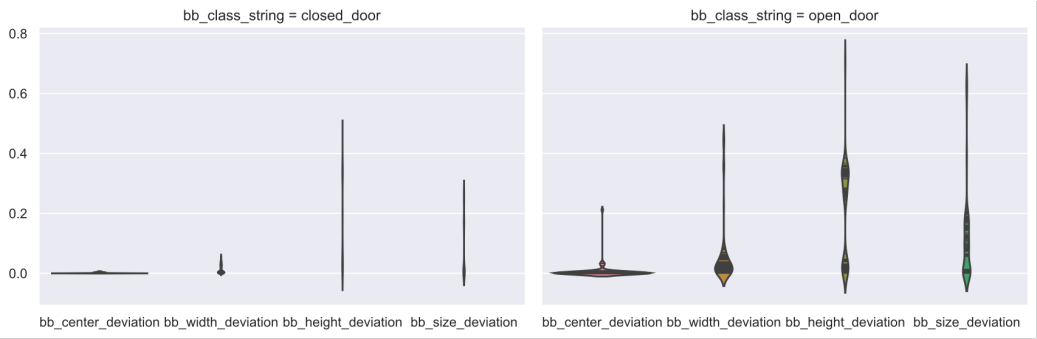


Figure 6: Bounding deviations between detected and ground truth.

For evaluating the door detector, we have written a dedicated test tool that tests the detector. Mean average precision defined in [23] was used as an evaluation metric, as well as the average of confidence score of the classification and percentage of wrongly detected objects. Additionally, we calculated the bounding box deviations between the ground truths and predictions to highlight the main issues of the detector, visible in Figure 7.

To run the tests, we have used the separate, augmented test data set with 396 images (including 554 labels). With them, the deviations and IoU were calculated and the results were categorized into three possible sets: classification result (correct and wrongly classified), detection result (no object shown in the image, missed object) and overall result (true positive, false positive, false negative). In the end, we have calculated the interpolated average precision of each detection according to [24] and summarized the result by calculating the mean over all recall levels. We summarized the detection as well by accumulating the results per classification. They are visible in figures 6 and 7. The test results of the

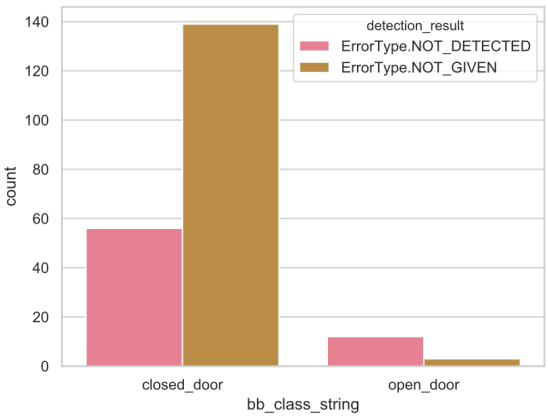


Figure 7: Detection errors per class.

detection tests are listed in Table 1.

6.6 Conclusion and further improvements

We have implemented a novel door detector by using the single-shot object detector "You Only Look Once" version 3, training it with a dedicated set of images relevant to the task at hand. The test results indicate clearly room for improvement, especially in case of detecting closed doors correctly. Also, our detector has significant problems in detecting non-doors as closed doors.

	open door	closed door
% Wrongly detected	4.6	0
Average precision (AP)	0.99	0.81
% Average confidence score	94.94	92.75

Table 1: Test results per class

We ran the system with some samples of the data set by [17] and weren't able to detect the doors correctly. Reasons for this might be the distortions introduced by different camera intrinsic parameters, different resolution and noise patterns, as well as different environments in which door images were taken.

To generalize the detection further, more data sets should be used. Multiple cameras, perspective angles, environments and lighting conditions should be considered, as well as an equally distributed set of open and closed doors.

7 Door Localization

This section highlights the process involved in computing the absolute location of doors, with the purpose of displaying them in the reporter map.

7.1 Pinhole camera model

The location of the door is relative to the Kinect camera. Therefore, in order to get the absolute position, we must transform it from the camera frame to the world frame. Initially, we calculate the relative position of doors with

respect to the camera. For this we make use of the pinhole camera model:

$$WorldCoordinate = Depth * \frac{ImagePlaneCoordinate - CenterDisplacement}{FocalLength}$$

Afterwards, in order to compute the corresponding absolute position of the door, each relative position of a detected door is then transformed to the world frame by applying a 4×4 transformation matrix, encoding the rotation and translation of the camera with respect to the world frame.

7.2 Relative and absolute value storage

Both relative and absolute door positions are stored in *mongodb_store*, an extension of MongoDB (one of the NoSQL database types), that works well with ROS and allows handling data in JSON format.

The pose of the robot is continuously corrected presumably after each loop closure. Hence, the calculated absolute door positions may be inaccurate. Thus, the stored list of absolute door positions are clustered periodically to correct the errors.

8 Reporter

The door detections, as well as the position of the robot on the map should be provided to the user by a user interface. For this, we have developed a reporting engine and a web application which communicate via a RESTful API.

8.1 Implementation

We have implemented a ROS-node that collects the detection output, the generated map and robot odometry. It converts the given map into an image and then draws the robot position, after converting from meters to pixels. This happens every time a new robot location has been received. The web application is implemented two-fold: REST API based back-end using *FLASK* and *CORS* python packages, and JavaScript based web front-end, developed with *NodeJS*, *Vue.js* [25] and *Quasar* [26]. We have wrapped the web application into a ROS-node to deploy it locally on the computer, but it can be deployed on a remote server as well.

8.2 Conclusion and further improvements

The user interface of the web application is visible in figure 8. It is currently read-only. Future improvements would show the door positions as additional map annotations, as well as their closed/open status. The reporter could also be more interactive. The user could set new goals for the robot to visit and inspect for example. Additional controls could be provided as well in a dedicated section, so the user could stop the robot every time it is in danger or stuck.

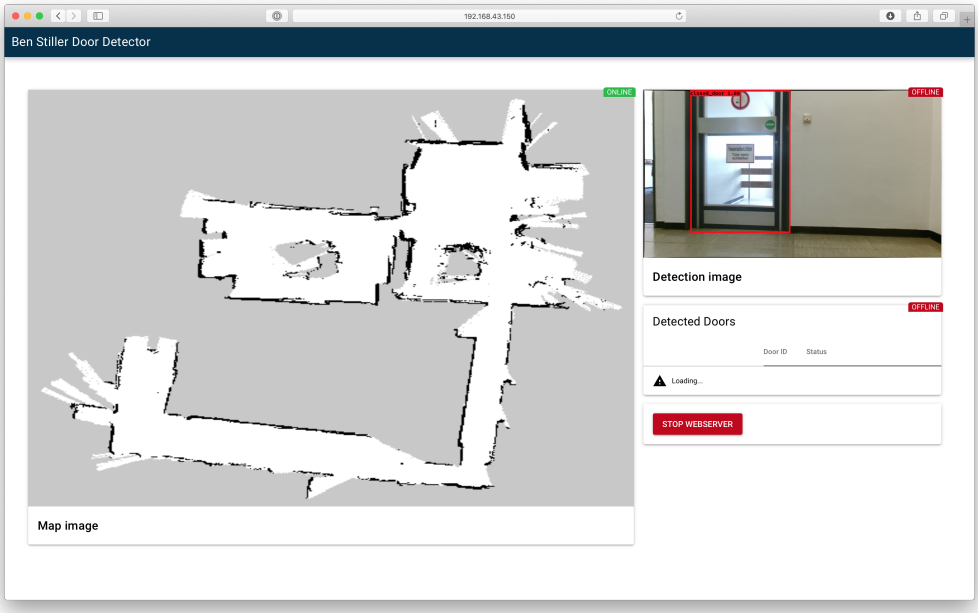


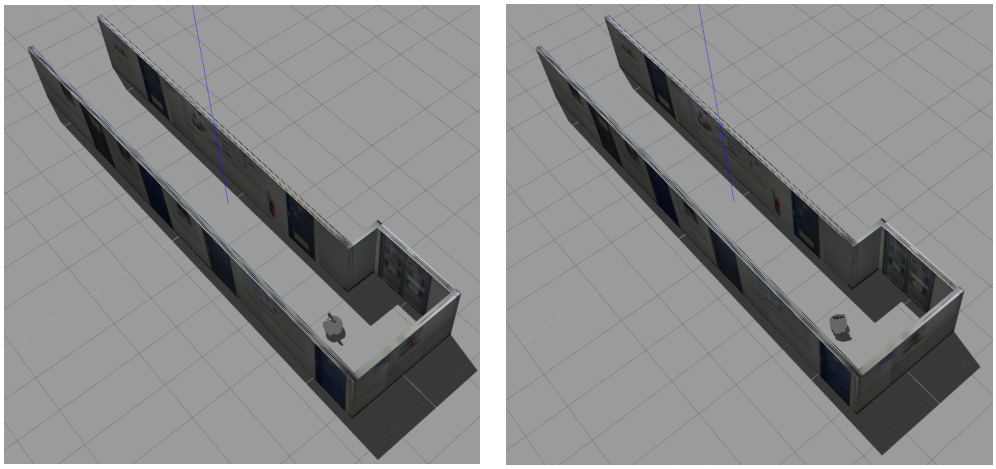
Figure 8: User interface of the web application (intermediate state without the robot odometry annotation)

9 Experimental Results

9.1 Simulation

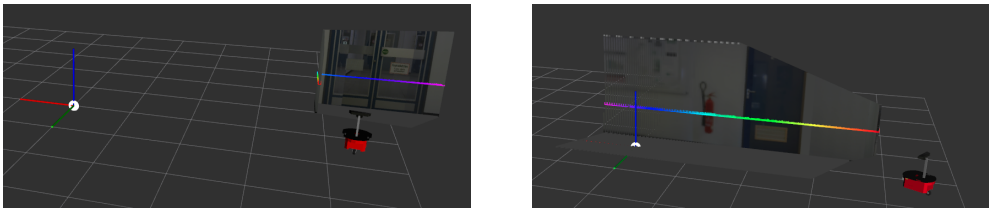
In order to have a proof of concept about the workings of the SLAM algorithm, we have created a simulation environment in Gazebo (see figure 9). The walls of the corridor were photographed using a simple camera and

manual stitching of the resulting pictures was done. In the end, we could set goals in RVIZ via the "2D Nav Goal" functionality and the robot successfully moved towards the desired locations (see figure 10). Early on we started noticing drift and the initial hypothesis was that it occurs because of the lack of depth information, since we just used 2D pictures on the walls, and not 3D pointclouds. We predicted that this drift will happen also in real life experiments because, although we have a laser scan slice of depth data, the corridors in the university are very homogeneous in RGB and not very diverse in terms of "depth texture". This turned out to be true, and we have made use of an IMU to correct the issues. Interestingly enough though, even when we add obstacles in the gazebo simulation such as tables and bookshelves, the drift still persits and moreover, it becomes accentuated (see figure 11)



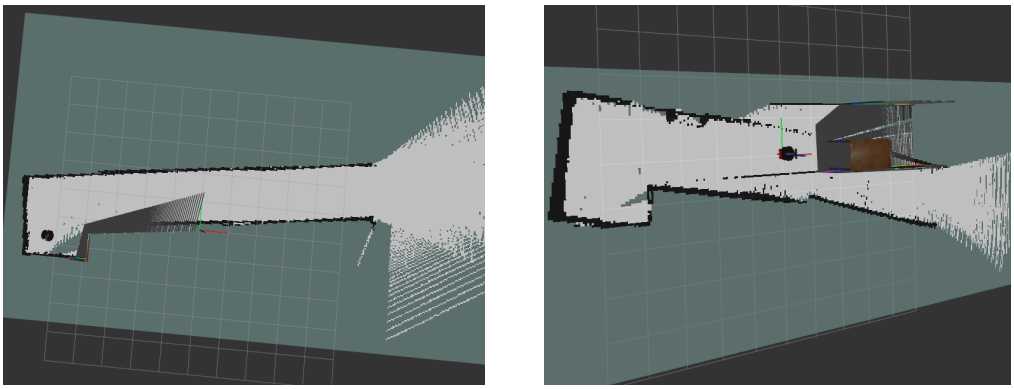
(a) Facing the entrance door (b) Facing the office door

Figure 9: In these GAZEBO Simulations the robot is oriented under different yaw angles. The captured simulated environment can be seen in the corresponding RVIZ images below.



(a) Facing the entrance door (b) Facing the office door

Figure 10: In these RVIZ Simulations the robot captures the image and laser depth data from the GAZEBO scenes presented above.



(a) Drift is present when no obstacles are placed in Gazebo simulation. (b) More drift appears when obstacles are placed in Gazebo simulation.

Figure 11: Drift is present with or without "depth texture".

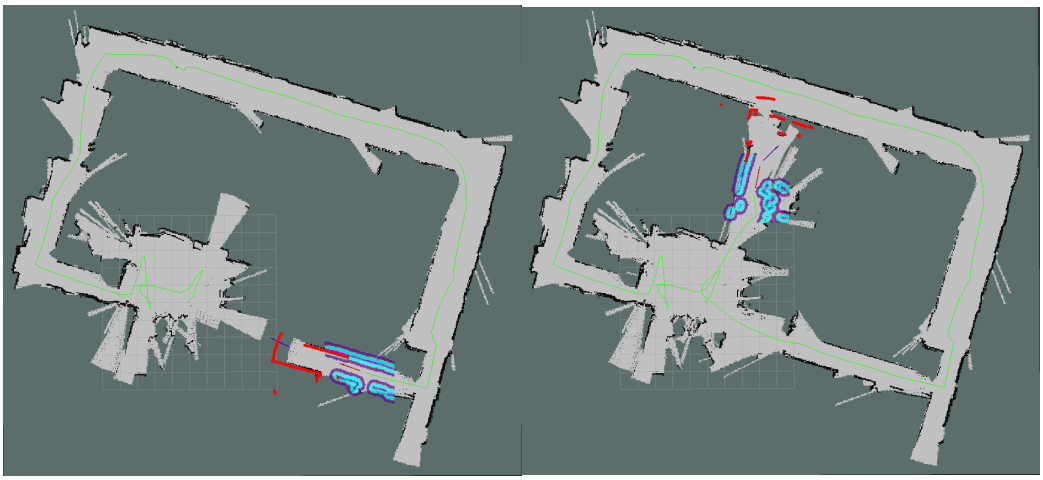
9.2 Real World Experiment

The following sections describe the main insights we have gained during the project by using the robot in real world settings.

9.2.1 SLAM

This section presents the final results of the SLAM system in our project. As mentioned in the corresponding implementation section, the wheel encoder odometry and laser scan SLAM has quite a big drift until a loop closure occurs. Incorporating an Extended Kalman Filter node, we were able to use IMU data and enhance the pose estimation as a result. Figure 12 shows the

map before and after a loop closure. Here the robot drove a larger loop as in Figure 4, but has a significantly lower drift. This drift can be seen in the right image ("After Loop Closure"), where the red laser scan does not match exactly the corridor wall behind the room.



(a) Before Loop Closure (b) After Loop Closure

Figure 12: Effect of the IMU and the Kalman Filter on SLAM performance

One other insight is that the ambient lights in addition to the relatively high amount of reflected rays by the floor (which is the case in all tested corridors) introduces ghost obstacles in the laser scan. This causes the robot to avoid those alleged barriers if space permits, or in the case of a narrow corridor for instance, the robot will not move through. As for the SLAM system, this introduces temporary obstacles in the map, which are cleared afterwards if the robot passes the same location from a different direction.

Yet another observation we've made was that, at the very beginning of building the map, if the robot is turning on the same spot, the map is likely to become corrupt.

Finally, it should be noted that the only part of the SLAM system relying on color data is the loop closure procedure. Therefore, it is the only component within the SLAM system which depends on external lighting and we might address the associated issues with an image enhancement preprocessing step in the future.

9.2.2 Exploration

Regarding exploration, the main thing to note is that we can't rely only on the Kinect to detect obstacles, because it has blind spots in its close vicinity due to the limitation of its minimum perception range. Consequently, we noticed that the exploration node benefits from the sonar system of the robot in order to work properly and avoid bumping into obstacles. The system works like well known toys that move until they hit a wall, at which point they change direction. In our case, we can't really afford such collisions, although we did install a soft buffer to protect especially the IMU chip protruding in front of the robot. Hence, the sonar system is a very useful component in such a scenario, compensating for the blind spots the Kinect is unable to perceive.

Another problem occurs when the robot tries to move through rather small doorways. At the moment of a map drift, it can happen that the opening of the door gets too small for the robot to pass through. This happens because of the overlap of the door frames in the map before and after the drift. Also the robot rejects to go through doors if there is too much noise in the doorway, which is recognized as many small obstacles blocking the doorway.

The preceding problem with not being able to move through doors leads to another issue. When the robot tries to move into another room during the process of autonomous exploration and is not able to pass through the door, because it perceives obstacles that are not present in the real world, it immediately tries to find another way into that room.

10 Conclusion

In conclusion we will mention the differences between the original plan and expectations versus the final outcome of the project, as well as future potential improvements and extensions that can be made.

Firstly, we can say that the overall gist of the project was successfully accomplished. We have a robot that can move autonomously through an environment, detect doors and semantically label them as open or closed. The robot does communicate with a backend system, which packages the data and sends it to a user-friendly interface for a human guard to assess.

The part that is partially implemented and still needs more testing deals with how to place the detected doors in the map shown to the user. We have a

mongodb database in place that stores the absolute distances of the detected doors, but we still need to check if the clustering algorithm is robust enough to avoid storing the same doors as different entities. Future work related to this aspect might be to evaluate different clustering algorithms and tune them appropriately. For instance, if we use KMeans, the question is what would be an appropriate K for our scenario (based on moving speed, average number of doors in a certain area, etc.)

One of the major missing components (much so that we had to rename the project) is the "night vision" node that would enable the robot to successfully navigate in darker environments. At its current stage, the loop closure thread would not be able to work without illumination because it relies on RGB data to match features. Regarding labeling whether a door is open or closed, we still have the system trained exclusively on RGB to perform the task. It would be interesting to actually use depth instead as a potential extension and improvement, such that labeling works in low light as well. For the loop closure to work in low light, we would require probably more accurate wheel encoders and other sensors that do not rely on visual features, and maybe also a different kind of slam node, other than rtabmap, that would accept a pointcloud instead of RGB data. Alternatively, a possible solution to try in the future could be to convert depth data into black and white images and feed those into the system in the hope that enough matching features can be extracted from such data as well.

References

[1] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[2] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[3] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[4] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You

only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[5] Hugh Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: part i. *IEEE robotics & automation magazine*, 13(2):99–110, 2006.

[6] Chad Rockey. depthimage to laserscan ros node. `https://wiki.ros.org/depthimage_to_laserscan`.

[7] David V. Lu. Move base. `http://wiki.ros.org/move_base`.

[8] Mathieu Labbé and François Michaud. Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation. *Journal of Field Robotics*, 36(2):416–446, 2019.

[9] Tom Moore. Robot localization. `http://wiki.ros.org/robot_localization`.

[10] Jiří Hörner. Map-merging for multi-robot system. Bachelor's thesis, Charles University in Prague, Faculty of Mathematics and Physics, Prague, 2016.

[11] Tom Moore. Mongodb store. `http://wiki.ros.org/mongodb_store`.

[12] Thiemo Wiedemeyer. Openni kinect bridge. `https://github.com/code-iai/iai_kinect2`.

[13] S. Rusinkiewicz and M. Levoy. Efficient variants of the ICP algorithm. In *Proceedings Third International Conference on 3-D Digital Imaging and Modeling*, pages 145–152, May 2001.

[14] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. ORB: An efficient alternative to SIFT or SURF. In *2011 International Conference on Computer Vision*, pages 2564–2571, Nov 2011.

[15] Dorian Galvez-Lopez and Juan Tardos. Bags of binary words for fast place recognition in image sequences. *Robotics, IEEE Transactions on*, 28:1188–1197, 10 2012.

[16] R. Mur-Artal and J. D. Tardós. ORB-SLAM2: An open-source SLAM system for monocular, stereo, and RGB-D cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, Oct 2017.

[17] Zhichao Chen and S. T. Birchfield. Visual detection of lintel-occluded doors from a single image. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–8, June 2008.

[18] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single Shot MultiBox Detector. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, pages 21–37, Cham, 2016. Springer International Publishing.

[19] Joseph Redmon and Ali Farhadi. YOLO9000: Better, Faster, Stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[20] Joseph Redmon and Ali Farhadi. YOLOv3: An Incremental Improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[21] qqwweee. a keras implementation of yolov3.

[22] tzutalin. LabelImg. `https://github.com/tzutalin/labelImg`.

[23] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The Pascal Visual Object Classes (VOC) Challenge. *International Journal of Computer Vision*, 88(2):303–338, June 2010.

[24] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[25] Evan You. Vue.js. `https://vuejs.org`.

[26] Razvan Stoenescu. Quasar Framework. `https://quasar.dev`.