IMPLEMENTATION OF PATH PLANNING ALGORITHMS ON A MOBILE ROBOT IN DYNAMIC INDOOR ENVIRONMENTS

by

Aishwarya A. Panchpor

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical Engineering

Charlotte

2018

Approved by:

_____
Dr. James M. Conrad

_____
Dr. Andrew R. Willis

_____
Dr. Ronald Sass

ABSTRACT

AISHWARYA A. PANCHPOR. Implementation of path planning algorithms on a mobile robot in dynamic indoor environments. (Under the direction of DR. JAMES M. CONRAD)

The objective of this thesis was to review and analyze existing algorithms for robot localization and mapping in dynamic indoor environments. Some of the existing algorithms include occupancy grid approach, artificial intelligence approach, dense scene flow approach for localization and mapping in dynamic environments. The occupancy grid approach maintains static and dynamic occupancy grids in parallel. The artificial intelligence approach uses efficient path planning algorithms like Reinforcement Learning along with Simultaneous Localization and Mapping (SLAM) to find a path and map the unknown dynamic environment. The dense scene flow approach detects moving objects to improve the visual SLAM process.

The review phase of this work included identifying different approaches and classifying them. The classification of the different approaches was based on sensors used (in the data acquisition process), localization method used, and map management techniques used. These approaches were categorized into fixed sets. The review of these algorithms lead to a comparison of the already obtained results of these algorithms.

The analysis phase of this work included implementing path planning algorithms on TurtleBot2 with real-time obstacle detection and avoidance. The results obtained were evaluated in terms of a set of fixed parameters. These parameters were accuracy of the algorithm, total time for execution, and errors in planning the final path for the robot.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. James Conrad for being a constant source of support and encouragement right from the first semester of my Master's program. Dr. Conrad's Embedded Systems coursework inclined me towards research. In the course of my thesis, he has helped me realize my potential and always motivated me to aim higher and achieve better results. Dr. Conrad introduced me to the TurtleBot which is the primary component of my thesis work. I am very grateful to Dr. Conrad for guiding me towards achieving my Master's degree.

I would also like to thank my thesis advising committee members Dr. Andrew Willis and Dr. Ronald Sass. Dr. Willis has always helped me in overcoming the different challenges I was facing during the implementation phase of my thesis. His coursework on Mobile Robot Sensing, Mapping and Exploration piqued my interest in this domain. Dr. Sass' coursework on Advanced Embedded Systems helped me in understanding the very basics of any embedded system. His lectures on this topic were very beneficial for my academics as well as my thesis work.

I also want to thank Dr. Sam Shue for his help and guidance all along this thesis. Right from selection of topic for my thesis till the actual implementation of my work, he has helped me every step of the way. I am thankful for all his ideas and suggestions to implement and test my thesis work.

My parents and my sister have always supported me and without their encouragement, this work would not have been possible. I would also like to thank my friends and roommates, Bhagyashree Desai and Karishma Keshav for all their help and support during the course of this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AMCL Adaptive Monte Carlo localization

AOI Area Of Interest

APF Artificial Potential Field

DBN Dynamic Bayesian Network

DPG Dynamic Pose Graph

DWA Dynamic Window Approach

ECE Electrical and Computer Engineering.

EKF-SLAM Extended Kalman Filter Simultaneous Localization and Mapping

HMM Hidden Markov Model

IAICP Intensity Assisted Iterative Closest Point

ICGM Incremental Center of Gravity Matching

LRF Laser Range Finder

RBPF Rao-Blackwellised Particle Filter

RL Reinforcement Learning

ROS Robot Operating System

SLAM Simultaneous Localization and Mapping

UNCC University of North Carolina at Charlotte

CHAPTER 1: INTRODUCTION

1.1 Motivation

Commercial robots have been in existence for more than 50 years. The earlier robots were preprogrammed to carry out specific tasks in a fixed environment. Over the years, due to the use of powerful sensors, the robots were able to detect obstacles. The robots that were developed in this period were able to take actions by processing the input from the sensors. Ultrasonic sensors, infrared sensors, and laser range finders are the most popular distance sensors. Cameras are powerful sensors used in a wide variety of applications. The images from a camera can be processed to detect natural landmarks like the presence of walls, doors, windows or artificial landmarks strategically placed for localizing the robot in unknown environments. To process the data received from these sensors, image and signal processing techniques are used. The processed data is then used to send out control signals to the robot. Control theory is the most widely used technique to issue commands to the robot or to decide action to be taken by the robot.

The use of robots to reduce human effort and simultaneously increase process efficiency has expanded to a great number of fields. Robots are being used in industry for tasks which require heavy lifting. For example, in the transportation industry, a robot lifting heavy equipment reduces the human effort. The car manufacturing industry uses high-precision robotic arms to streamline the production line. The medical robotics field is continuously growing. Use of surgical robots to perform minimally invasive and highly precise procedures, use of rehabilitation robots to improve strength and coordination of people with disabilities, use of disinfection robots to sterilize operating rooms for sensitive medical procedures are some of the applica-

tions of robotics in the medical field. Robots are also used for surveillance purposes, to gather information in space exploration missions and in the study of underwater environments. These are few of the large scale applications of robots.

Use of robots is no longer limited to large scale industries. Robots are now being used in our everyday life. The robotic vacuum cleaner Roomba has made home floor cleaning a trivial task. Laundroid, a completely autonomous laundry folding robot for washing, drying, folding and sorting clothes is another example of day to day use of robots. Another category of robots is social robots. These are mainly used as caretaker and companion robots.

All these different uses of robots need to sense their environment and build a comprehensive map for moving around in the environment. This has led to the development of various algorithms for effective localization and mapping in the presence of moving obstacles. A mobile robot needs to simultaneously solve many problems like sensing, mapping, localization, path planning, obstacle detection and avoidance for a completely autonomous system [1]. When a robot needs to go from one point to another, it needs a definite path. Path planning algorithms play a key role in automating the movements of a robot. In real world scenarios, the environment in which a mobile robot has to operate is dynamic due to moving obstacles. Unlike humans, a robot cannot adapt to the constant changes in the environment. In the pursuit of building intelligent robots, various path planning algorithms have been developed to aid the movements of a robot when the environment is changing continuously. These intelligent robots have the ability to sense the obstacles in the field of vision of the robot and re-plan its path so that it is able to avoid the obstacle and continue towards its desired goal. Real-time obstacle detection and avoidance has been a challenging problem to solve. Because of the varied nature of the obstacles in the surroundings, it is difficult to model the obstacles. If the nature of the obstacles is known in advance then it is easier to categorize the obstacles. For example, if the moving obstacles

are boxes of fixed length, width and height then it is relatively easier to avoid these obstacles. Another method to detect and avoid obstacles is to assign them a motion model and re-plan the path of the robot according to their movement. These scenarios only consider obstacles that fit in a set of specifications. If the nature of the obstacle is not known prior to their detection, then it is difficult to re-plan the path of the robot. The motivation for this thesis is the need to solve the problems arising due to the dynamic nature of the environment as the robot moves from the start point to its goal point.

1.2 Completed Thesis Work

Over the years, various path planning algorithms have been developed to guide a robot from the start point to the goal point. These algorithms work on the premise that the robot has a map of the environment and the position of the robot in this map is known. Combining powerful path planning algorithms with real-time obstacle detection and obstacle avoidance is the main purpose of this thesis.

The work presented in this thesis aims at comparing the implementation of various path planning algorithms in the presence of moving obstacles. The algorithms that have been studied and implemented are Dijikstra's algorithm and A* algorithm. The algorithm developed as part of this thesis is a path planner which takes the minimum number of turns with real-time obstacle detection and avoidance. The Reinforcement Learning algorithm has also been studied as part of the background study and implemented on an arbitrary grid map. The moving obstacles are assumed to have a height equal to or greater than the height of the sensor installed on the robot. The robot used for the purposes of implementation is the TurtleBot2. The contribution of this work is to compare the performance of the path planner developed in the implementation phase to the path found from implementation of Reinforcement Learning on the same grid in the presence of moving obstacles. The work presented in this thesis shows how the built-in global planner fails in the presence of moving obstacles

and the implementation of the path planner developed in this work. Also, the system uses only the Kinect sensor installed on the TurtleBot2 to detect and avoid obstacles which makes it a very low cost option. The results are compared based on the robot reaching the goal point while detecting and avoiding obstacles and the time required to plan the path using various path planning algorithms.

1.3 Organization of the Report

This section gives an overview of the organization of the report. The remainder of the report is organized into following sections: Literature Survey, Implementation, Results, Conclusions and Future Scope, References, and Appendix.

Chapter 2 presents a survey of various algorithms that have been developed to detect and avoid obstacles while localizing the robot and mapping the environment. Some of the algorithms that are surveyed work under the assumption that the map is already given. Some of them assume that the localization of the robot is given or the robot has perfect odometry. These cases are further discussed in Chapter 2.

Chapter 3 discusses the different path planning algorithms, their pseudo code, advantages and disadvantages, and their implementation. The implementation of the proposed optimised path planner algorithm with real-time obstacle detection is further explained in Chapter 3. This chapter also explains the configuration and specifications of the system.

Results obtained from the implementation of the different algorithms discussed in Chapter 3 are presented in Chapter 4. It includes images of the map used and images of the robot as it follows the path. It also includes images of the obstacles used. This chapter also discusses the time required for the algorithms.

Chapter 5 concludes the work done in this thesis and interprets the results that have been presented in Chapter 4. It also discusses the future scope for this thesis.

CHAPTER 2: LITERATURE SURVEY

2.1 Survey of Existing Methods

Mobile robot localization and mapping is an essential part of building an autonomous system. Simultaneous Localization and Mapping (SLAM) algorithms are used to build a map of an unknown environment, while keeping track of the position of the robot. Probabilistic formulations are based on Extended Kalman Filters, particle filters, and maximum likelihood estimation for approximating the pose of the robot and map have been reviewed by Durrant-Whtye and Bailey in their two survey papers [2, 3]. Most of the SLAM methods are developed with the assumption that the environment is static. The work presented in [4] uses the wireless signal strength to localize the robot in indoor environments. When the environment consists of moving objects, their pose changes affect the localization and mapping of the mobile robots.

In real life scenarios, there are moving obstacles like humans in the path of the robot. Apart from humans, there are objects which may change their positions such as tables and chairs. In an industrial setting, there may be shelves or cabinets which are relocated from time to time. In such dynamic environments, the mobile robot must perform SLAM and account for these possible changes within the environment to avoid obstacles to reach the desired goal point. To cope with the dynamic changes, previously explored regions of the map must be updated to account for these dynamic aspects of the environment.

Static and dynamic maps are created for a complete description of the environment over time [5]. The method described in this work assumes that the localization of the robot is already established. The occupancy grid techniques [6] are used for creating a static map and a dynamic map. Occupancy probability is defined for each cell

in both maps. The confidence that the cell is occupied or not is indicated by the occupancy probability. Dynamic objects are represented as free space in the static map and static parts of the environment are represented as free space in the dynamic map. In both the static and the dynamic map, the inverse observation model is used to define if the occupancy probability will be high or low based on the occupancy states in the previous static or dynamic map respectively and the current sensor information. Object detection is carried out by setting a high threshold (in this case 0.85) to the occupancy probability of a region to detect a moving object. Objects are differentiated and classified based on their size. Whenever there is a new sensor reading, to determine if the new cell is part of an existing moving object or if it is a new moving object, the system checks the neighboring cells for objects in the dynamic map. If there exists a moving object in the cells close to the newly observed cell, then that cell is incorporated into that object. Otherwise, it is considered as a new moving object.

Various SLAM methods like Extended Kalman Filter Simultaneous Localization and Mapping (EKF-SLAM) [7], FastSLAM [8], GraphSLAM [9] have been developed for uninhabited and static environments. For dynamic environments, obstacle avoidance plays an important role.

The following sections illustrate different methods developed for localization and mapping of mobile robots along with obstacle avoidance in dynamic indoor environments.

2.2 Artificial Intelligence-based Approach

The artificial intelligence based approach of Reinforcement Learning (RL) [10] is used to find a path and create a map of the unknown environment. The following two methods combine RL and SLAM for dynamic indoor environments.

2.2.1 Reinforcement Learning as Control System

The work by Arana-Daniel et al. [11] focuses on integrating reinforcement learning for navigation in unknown and dynamic environments and utilizes a SLAM algorithm for localization and mapping. EKF-SLAM and fast-SLAM have been used in this implementation. Reinforcement Learning (RL) algorithm known as Q-Learning, a type of RL algorithm, has been used for this application. It consists of state-action pairs, a function of immediate reward for action taken, and a Q value function to determine quality of the action taken. The Q function is defined as follows,

$$Q(s_t, a_t) \rightarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

Where α is the learning rate, r is the reward, t is the time taken, s denotes the state, and γ is the forgetting factor. In this implementation α=0.9 and γ is eliminated. An expected map is given to obtain the optimal route and the same RL algorithm is used for obstacle avoidance in dynamic environments. To find alternative routes to avoid obstacles, the long-term memory obtained from the first run of the Q-learning algorithm is used. Reinforcement learning is used as the control input for the SLAM algorithm. A desired exploration policy is used as a priori to guide the robot. The scanning process creates two maps. The first map describes the environment in a feature based way generated by the SLAM algorithm. The second map contains the discrete representation of the environment and provides relevant information to implement RL.

2.2.2 Inverse Optimal Control

Another method by Guevara-Reyes et al. [12] uses Reinforcement Learning and SLAM to create a map and find a path in an unknown and dynamic environment. The algorithm starts with an initial state and a goal. The initial desired route is obtained. After the first observation, until the goal point is reached, the system continuously updates the map with the current state of the robot and the new observations. The

route is updated based on the new observations and detection of obstacles. RL algorithm is used for finding a new path in the presence of obstacles. The control of the robot is executed based on the high order extension of Hopfield model (RHONN). The neural network is trained with EKF wherein the weights in the neural network are the state to be estimated. EKF is defined as,

$$K_{i(k)} = P_{i(k)} H_{i(k)} M_{i(k)}^{-1}$$

$$\omega_{i(k+1)} = \omega_{i(k)} + \eta_i K_{i(k)} e_{i(k)}$$

$$P_{i(k+1)} = P_{i(k)} - K_{(i,k)} H_{i(k)}^T P_{i(k)} + Q_{i(k)}$$

$$M_{i(k)} = [R_{i(k)} + H_{i(k)}^T P_{i(k)} H_{i(k)}]^{-1}$$

$$e_{i(k)} = x_{i(k)} - \hat{x}_{i(k)}$$

Where K_i is the Kalman gain matrix, ωi is the weight (state) vector, P_i is the prediction error covariance matrix, R_i is the measurement noise covariance matrix, and each H_{ij} entry of H_i is the derivative of one of the neural network outputs $\hat{x}_i$ with respect to one neural weight ω_{ij}. The Lyapunov Control Function is designed as an inverse optimal controller which satisfies the passivity condition. The purpose of the inverse optimal controller and the RHONN model is to force the angular velocities of the left and right wheels of the robot to follow the desired reference signal. After the control action is obtained, the prediction step and the measurement step of SLAM is performed.

2.3 Particle Filter-based Approach

2.3.1 Distance Filter and Scan Matching

Sun et al. [13] presents an approach for localizing in dynamic environments with the use of a particle filter, distance filter and a scan matching method and mapping with the use of an extended sensor model. In scan matching, a multi-layer searching technique is used to find the local maximum of the robot's position given observations

of its environment through the use of a laser range finder. A matching score is obtained which is used to prevent updates based on wrong estimates. For each measurement, the distance filter computes the probability that the distance is shorter than expected from the map as follows,

$$p_{short}(d_i) = \Sigma_l p_{short}(d_i|x_l) \times p(x_l)$$

Where d_i is the measurement, x_l is the position based on current belief and p(x_l) is given by the particle filter.

A full posterior distribution is unnecessary because only the latest configuration of the map is of importance. Hence, the marginal distribution is recursively computed for the current map using a Bayesian filter. Dynamic occupancy grid maps are used to integrate the dynamic changes. The sensor model implemented is independent of shape and type of the dynamic object. A standard SLAM approach is applied to generate the initial map. Maximal translation error and maximal angle error are the two parameters used to evaluate the quality of localization.

The system is also evaluated with the distance filter and scan matching disabled. In this case, the map is updated in every iteration step which leads to divergence of the map. This problem can be overcome if each particle updates and maintains its own map which increases the load and usage of CPU and memory. The mapping results for the system were evaluated by comparing with the ground truth map. Table 2.1 compares the error obtained by implementing Adaptive Monte Carlo localization (AMCL) (from ROS package), $AMCL_DF$ (AMCL with Distance Filter), DL which (method described in this paper), and DL_{simple} (DL with distance filter and scan matching disabled). The results in Table 2.1 are for a production hall (104cm x 52cm). The first two methods do not update the map while the next two update the map at run time.

Table 2.1: Comparison of Static and Dynamic Methods from [13]

Method	Error (m)		Error (degree)		Success
	Mean	Variance	Mean	Variance	
AMCL	1.791	3.868	10.212	21.245	0/10
$AMCL_{DF}$	0.138	0.051	0.638	0.523	10/10
DL_{simple}	0.415	0.756	1.184	1.570	7/10
DL	0.126	0.059	0.687	0.629	10/10

2.3.2 Server-Agent Architecture

The work presented by Dorr et al. [14] uses sensor information from the mobile robots and other stationary sensors to maintain up to date dynamic occupancy grid maps of the changing environment. The transition probabilities are defined as,

$$p_c^{o|f} = p(c_t = occ|c_{t-1} = free)\ p_c^{f|o} = p(c_t = free|c_{t-1} = occ)$$

And the transition matrix is as follows,

$$A_c = 1 - p_c^{f|o} p_c^{f|o}$$
$$p_c^{o|f} 1 - p_c^{o|f}$$

Where $c_t = occ$ and $c_t = free$ are two finite states of the hidden Markov model (HMM). In this implementation, there are three possible outcomes for HMM cells z ϵhit, miss, no-obs and the observation parameters p(z|c=free) and p(z|c=occ) only depend on sensor characteristics.

The Rao-Blackwellized particle filter [15] is used along with a server-agent architecture. In the proposed architecture, the changes in the environment are sent by an agent to the server and the server updates the global map. The mobile robot or different stationary sensors are the agents and the server contains the dynamic occupancy grid map. Each agent has an agent handler which integrates the changes received from

an agent into the global map and provides the agent with the required data from the updated map. To detect dynamic changes in the environment and simultaneously localize, a long-term SLAM algorithm is deployed using Rao-Blackwellised Particle Filter (RBPF) on the agent side. The localization accuracy, total path lengths and travel times of the robot are evaluated.

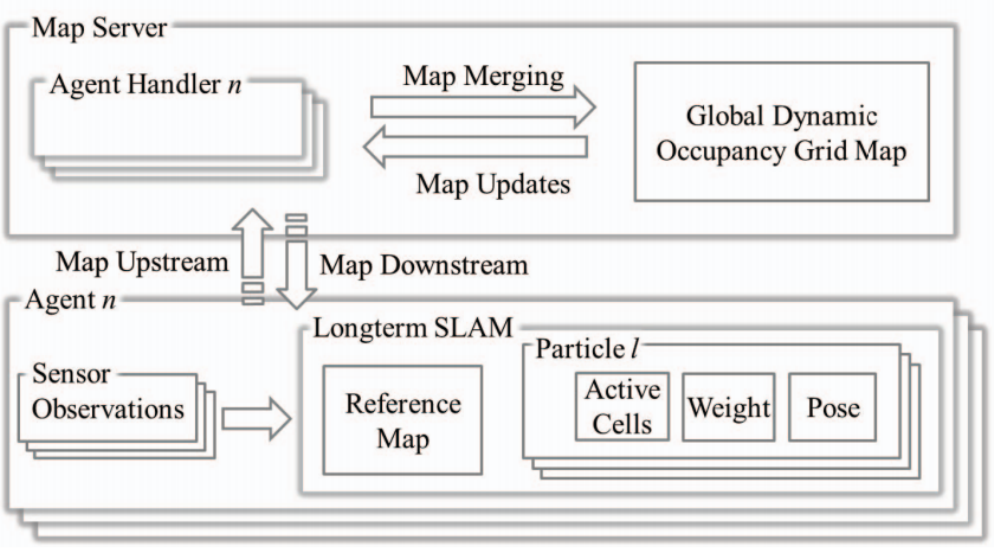

Figure 2.1: Proposed architecture of cooperative SLAM system based on [14]

2.3.3 RFID Tags and Dynamic Bayesian Network

Robot localization and mapping in semi-dynamic environments is carried out with the help of RFID tags with unique identification code attached to semi-dynamic objects [16]. This work also proposes SLAM-SD which consists of a framework of (DBN) and Rao-Blackwellised Particle Filter. In the prediction of a particle, the transition probability from time t-1 to time t is as follows:

$$\tilde{D}_t^{(i)} \leftarrow P(\tilde{D}_t^{(i)} | D_{t-1}^{(i)})$$

$$\tilde{X}_t^{(i)} \leftarrow P(\tilde{X}_t^{(i)} | X_{t-1}^{(i)})$$

$\tilde{D}_t^{(i)}$ and $\tilde{X}_t^{(i)}$ are the predicted particle state given by the transition probability $P(\tilde{D}_t^{(i)} | D_{t-1}^{(i)})$ and $P(\tilde{X}_t^{(i)} | X_{t-1}^{(i)})$. Xt and Dt are the robot pose and the semi-dynamic

object pose at time t.

The particle is evaluated based on weight as follows:

$$\omega_t^{(i)} \alpha P(Z_t, U_t | D_t^{(i)}, X_t^{(i)})$$

$$= P(I_t | D_t^{(i)}) \times P(L_t | D_t^{(i)}) \times P(L_t |, X_t^{(i)})$$

Where U_t indicates the odometry data, Z_t is the sensor data, I_t indicates existence of RFID tag, L_t indicates the range scan data.

The particle is resampled in the update step and the effectiveness of the particle is evaluated based on the following,

$$Neff = 1/\Sigma_{i=1}^{N} (\omega^{(i)})^2$$

The map $M_t^{(1)}$ is generated using $P(M_t^{(1)} | X_t(i), Z_{1:t})$ probability for each particle and time is incremented for the next iteration.

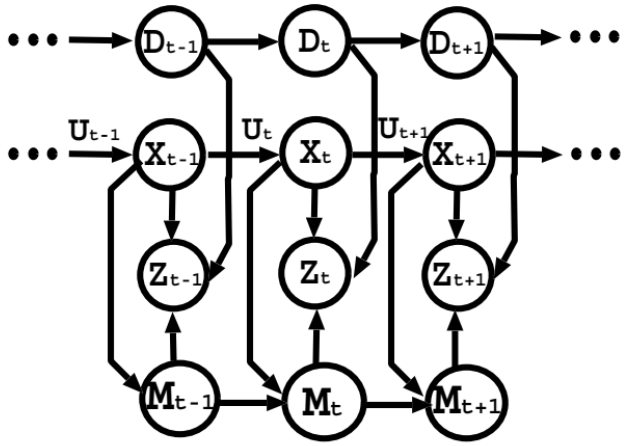

Figure 2.2: Dynamic Bayesian Network (DBN) model of SLAM-SD from [16]

The RFID tags convey the 3D geometrical information and relative coordinate between the object and the RFID tag of the semi-dynamic object. The experiments are conducted in a room environment where the location of the cabinets is changed, and in a long corridor environment where the doors and dust boxes change states.

2.3.4 Modified Particle Filter

A modified particle filter is proposed by Vidal et al. [17] for simultaneous localization and mapping in dynamic environments with the use of only static landmarks. The work proposed in the paper uses FastSLAM [8] implementation with the removal of mobile and/or similar landmarks. The SURF algorithm is used to extract feature points to compute the descriptors consisting of the orientation and neighborhood of that pixel. The outliers (moving landmarks) are identified based on the difference between the correlation of the position of the landmarks. The matrix D stores these Euclidean distances as defined below,

$$D^{i,j} = \begin{cases} 0, & \text{if pos(m,i) = pos(m,j) } \forall m, 0 \leq i \leq N, 0 \leq j \leq N \\ C_i^M \text{ - } C_j^M, & \text{if } \exists m | pos(m,i) \neq pos(m,j) \end{cases}$$

Where pos is the position of the landmark m, N is the number of landmarks and C_t^M is the landmarks positions correlation matrix at time t.

The outlier filter ignores an observation if it belongs to the set of mobile landmarks. The modified particle filter considers the set of static landmarks to estimate the pose of the robot. The Trajectory Planner is used to build a 2D map of the environment and detect obstacles with the help of laser sensor and odometry. The Ant Colony Optimization algorithm is used to plan the trajectory of the robot. The modified particle filter resamples a new set of particles at each estimate of the pose. The weighted average of the positions by their respective beliefs of the particles decide the estimate of the pose of the robot.

2.4 GraphSLAM-based Approach

2.4.1 Use of Static Point Weighting

The work presented in [18] proposes a static weighting method which downweighs the dynamic edge points from the incoming frame. Static point weighting is used to

indicate the likelihood of foreground edge points (extracted from the incoming frame) being part of the static environment. The depth edge points are matched between consecutive frames by geometric and intensity distances. Static weights are computed for every keyframe (Nth incoming frame) and intensity assisted iterative closest point (IAICP) is used to transform from the keyframe to the current frame.

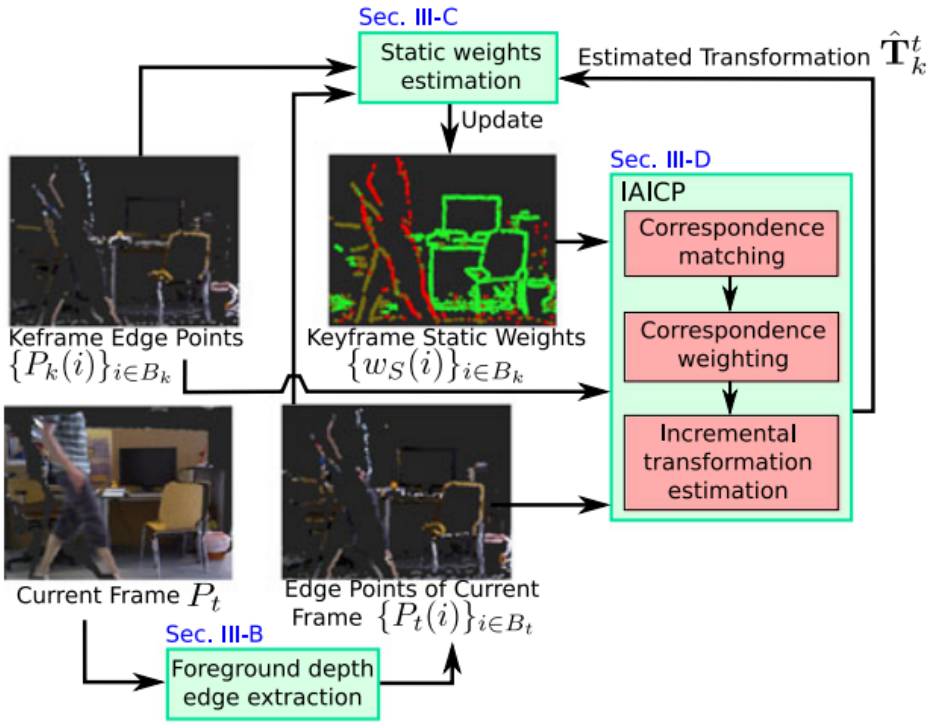

Figure 2.3: Visual odometry system implemented in [18]

The effect of dynamic objects is reduced by combining the static weights during transformation and the static weights are then computed based on the estimated motion. Pose graph SLAM was used to address the problem of drifting of the robot pose. For loop closure, the proposed method checks for three conditions. First condition is to check geometric proximity of the new keyframe with ten randomly selected previous keyframes (Pr). Second condition is that there should be 30% or more common visible part in the two frames, and third condition is forward backward consistency check by registering the two frames twice (Pr as source frame and then as target

frame) using IAICP. The proposed system is evaluated on TUM RGB-D dataset.

2.4.2 Long-Term Mapping

The use of Dynamic Pose Graph (DPG) SLAM [19] for localization and mapping is proposed in the work of Walcott-Bryant et al. This work aims at maintaining a precise map of the gradually changing (low-dynamic) environment over a long period of time. Changes in the environment are detected by comparing laser scans from various time instances. Table 2.2 compares the ground truth error for DPG-SLAM (reduces DPG size in every iteration) and DPG-SLAM-NR (does not reduce the DPG size in every iteration) over 20 passes through the room and traveling 1km.

Table 2.2: Comparison of Error for CSAIL Reading Room [19]

Algorithm	Error (cm)
DPG-SLAM-NR	3.9-13.4
DPG-SLAM	3.4-13.3

2.4.3 Use of Moving Landmarks

Due to moving landmarks, the localization and mapping of the mobile robot results in inaccurate localization of the robot and inaccurate maps. To minimize the effect of moving landmarks, the work of Xiang et al. [20] proposes a model based on Expectation Maximization which uses the mobility of landmarks in the Graph SLAM process. Some moving landmarks are treated as outliers if the mobility is not within the confines of a measurement function described as an augmented Gaussian distribution as follows,

$$P(z_k|x_{ik}, l_{jk}, w_{jk})\alpha exp(-w_{jk}\tilde{\mu}_k^T\Sigma_k^{-1}\tilde{\mu}_k),$$

$$\tilde{\mu}_k = v_k(h_k(x_{ik}, l_{jk}) - z_k)$$

Where w_{jk} is the likelihood of the landmark l_{jk} being static, z_k is the measurement

at time k, v_k is the scaling factor of each landmark representing whether the landmark is an inlier.

2.5 Visual SLAM-based Approach

2.5.1 Visual SLAM and Dense Scene Flow Approach

The work of Alcantarilla et al. [21] focuses on improving the visual SLAM [22] algorithm by means of dense scene flow representation of the environment to detect moving objects for visually impaired users. The visual SLAM system uses a stereo camera for visual odometry [23] and utilizes a hierarchical structure and motion refinement with Bundle Adjustment [24]. Harris corner detector [25] is used to detect features between consecutive frames. The Mahalanobis distance of the 3D motion vector is used to identify moving points. With the help of the residual motion likelihoods for every pixel in the current image, a moving object mask is created. The proposed method was tested with visually impaired users in crowded environments with pedestrians and cars (indoor and outdoor environments).

Table 2.3: Comparison of the Distance Covered [21]

Method		Experiment 1 Error	Experiment 2 Error
Ground Truth (m)		647.00	447.00
Visual SLAM Estimated trajectory (m)	moving object detection	646.07 (0.93)	449.79 (2.79)
	without moving object detection	641.37 (5.63)	451.54 (4.54)

Table 2.3 compares the ground truth of the total path for two experiments with

the estimated trajectory length obtained from Visual SLAM. The error for estimated length of visual SLAM with detection of moving objects is about 1 m for the experiment 1 conducted at Atocha railway station and it is almost 3 m for the second experiment 2 conducted at Alcala de Henares.

2.5.2 Visual SLAM and Exclusion of Dynamic Features

The proposed method called ICGM2.5 [26] excludes dynamic features to create a map of indoor and outdoor environments. The external sensor used is a hand-held monocular camera. The method proposed in this paper is an improvement on an approach proposed by Kayanuma et al. called ICGM2.0 (Incremental Center of Gravity Matching) [27] which is based on an approach proposed by Hua et al. called Incremental Center of Gravity Matching (ICGM) [28]. In the method proposed in this paper, the centroid is calculated after reducing the dynamic features. Also, the center of gravity is calculated based on the features which have a shorter matching distance. The system is evaluated based on the error rate obtained by dividing the difference between the start and end points by the length of the whole visual odometry for ICGM2.5[26], ICGM2.0 [27], ICGM [28], PIRF [29], and Libviso2 [30] as shown in Table 2.4.

Table 2.4: Indoor Experimental Results based on [26]

Method	Error rate (%)	Time (s)
ICGM2.5[26]	0.46	1.22
ICGM2.0[27]	2.13	1.19
ICGM[28]	2.75	1.20
PIRF[29]	2.43	1.25
Libviso[30]	5.76	1.16

2.6 Other Methods

2.6.1 Use of Path Planning Algorithms

The absolute localization of the mobile robot is carried out with the help of 2D-codes on the ceiling and a USB camera [31]. The ROS package hector-slam was used to build 2D occupancy grid map. Adaptive Monte Carlo localization(AMCL) algorithm was used for tracking the pose of a robot against a known map. The robot generates an optimal path from the starting point to the destination based on the established global map. The system uses the A* algorithm and DWA algorithm to generate a new path to avoid obstacles.

Another method proposed by Maurovic et al. [32] for localization and path planning in dynamic environments (with moving obstacles) is based on the D* algorithm with negative edge weights.

2.6.2 Use of Segmentation

The work of Runz and Agapito [33] introduces a new technique to cope with moving objects in the environment. The proposed method segments the scene into background and foreground objects using either motion or semantic cues. The grouping strategies are motion segmentation and object instance segmentation. These identify static objects and objects due to their motion. The system maintains active models with objects that are currently visible in the live frame and inactive models with objects that were visible in the previous frames. For each frame the 6DOF rigid pose of an active model is tracked and the entire frame is segmented based on motion segmentation and multi class image segmentation. Motion segmentation associates a pixel with a rigid motion model. If the connected region has adequate amount of outliers, a new model is added to the list. Multi class image segmentation is based on classifying each pixel based on deep learning approach. The fusion step involves using the 6DOF pose to update the active model.

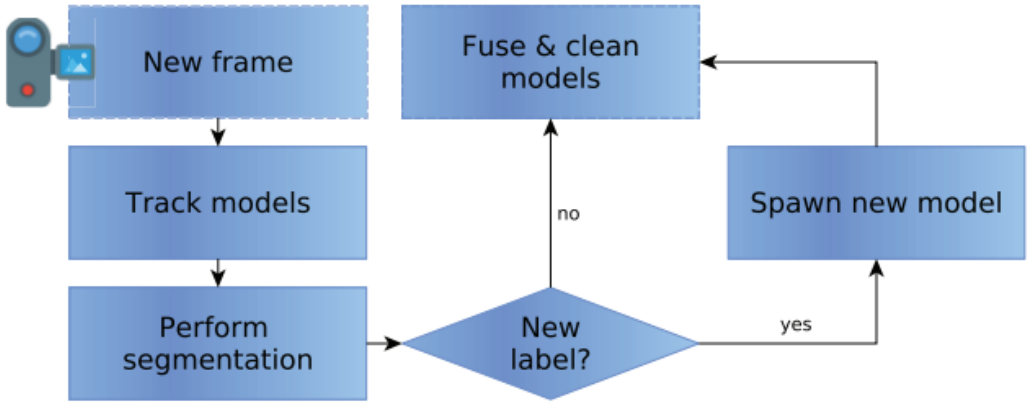

Figure 2.4: Overview of the method proposed in [33]

2.6.3 Real-time 3D Data-based Obstacle Avoidance

The work of Dirk Holz et al. [34] presents a method to scan the environment to obtain continuous real-time 3D data, implement obstacle avoidance and perform online SLAM. This method uses 6DOF pose representation. The implementation in this paper uses KURT3D robot platform and IAIS 3D laser scanner. The Area of Interest (AOI) is defined with the help of theta (pitch). The 3D information is maintained in 2D with the help of 2D obstacle maps (for obstacle avoidance) and 2D structure maps (for robot self-localization). The mapping procedure involves transformation to keep the map in alignment with odometry, removing obsolete points so that the dynamic changes can be maintained in the map, and replacing the previously saved points by the current (latest) laser scan data. The obstacle avoidance is implemented with the help of three commands: steer (robot directed towards maximally free space), brake (stop the robot when an obstacle is detected), turn (turns the robot and steers it out of dead ends).

2.6.4 Use of Navigation Function

The proposed method by Iizuka et al. [35] uses Extended Kalman Filter Simultaneous Localization and Mapping along with Laser Range Finder (LRF) for mapping

and navigation in a dynamic environment. A Navigation function which is a type of Artificial Potential Field (APF) [36] is used to generate orbits to avoid obstacles. The proposed method was tested in a simulated environment. The Navigation function ensures whether a safe distance was maintained between the robot and the obstacle in a dynamic environment. The results in the paper indicate that in the simulated environment, the point mass robot reached the goal point without colliding with the moving obstacles.

2.6.5 FPGA-based Approach

A hardware efficient design implemented on FPGA (Spartan 6) for multiple robots to reach the goal point in the presence of static and dynamic obstacles in indoor environments is proposed by Chinnaaiah et al. [37]. This work presents an algorithm for path planning and behavioral control between two robots. The robots communicate with the help of RF transceivers. An IR beacon sensor acts as the goal point and the leader robot plans the path to be traversed by both the robots. The leader robot senses the obstacles with the help of an ultrasonic sensor and conveys this information to the follower. The obstacle avoidance module evaluates the object and dynamic obstacles are avoided by a 32-bit counter. The proposed algorithm computes the shortest path for multi robot system and uses an FPGA as a control module. Both of these factors lead to low power consumption.

2.7 Inference from the Survey

The above sections describe various methods for mobile robot localization and mapping with obstacle detection and avoidance in dynamic indoor environments. This chapter gives an overview of a wide range of methods but the scope of this topic being so vast, there are numerous methods which can achieve similar outcomes. The different methods are evaluated in terms of a wide variety of parameters and a concrete comparison can be presented if they are implemented under the same set of

conditions. The continuous improvement of the existing methods and development of new ways to deal with the issues arising in dynamic indoor environments has led to better results. Since many problems in mobile robot localization and mapping in dynamic indoor environments do not have a robust solution, there is a great deal of scope for further research and development in this field.

CHAPTER 3: IMPLEMENTATION

3.1 Introduction to Path Planning

Path planning has played a key role in the movement and navigation of robots. Path planning algorithms require a map of the environment. Localization of the robot is not taken into consideration by the path planning algorithms. The primary purpose of path planning algorithms is to find the optimal path from the start point to the goal point. This optimal or best path is decided on various factors. These path planning algorithms compute the shortest path between the start point and the goal point. Most of times there are multiple ways to reach the goal point. These algorithms evaluate the cost or steps to reach the goal point. Based on the cost to move from one position to the next, the particular movement is selected. The use of cost for selecting the step to be taken is explained in the next few sections. Some specialized path planning algorithms focus on developing an optimal path in accordance to specific conditions. For example, some paths need to be optimal around turns. So the best path would contain the least number of turns. If this means taking a longer route to reach the goal point, that would be considered valid since the number of turns is optimal. So, in this case, the number of turns acts as the deciding parameter for selecting the optimal path.

3.2 Map Representations

To implement any path planning algorithm, there has to be a map representation of the environment. Maps can be expressed in metric or topological terms. The metric maps have fixed coordinates for objects in the map and the distances are calculated based on the distance between the coordinates of the two points. In a topological

map, the places in the map are stored with respect to the distance between them. This creates a graph of the connected nodes and the distances between the nodes.

Another type of map representation is with the help of occupancy grids. Occupancy grid maps consist of discretized squares (cells) of fixed resolution. These squares are classified as either occupied or free. Occupied regions denote the presence of obstacles in the occupancy grid. Probabilistic occupancy grids consist of a probability for each cell that they contain an obstacle. Requirement of large memory for storage of these occupancy grid maps is a drawback. As the size of the map increases, so does the computational time to traverse the map and find the shortest path in the map.

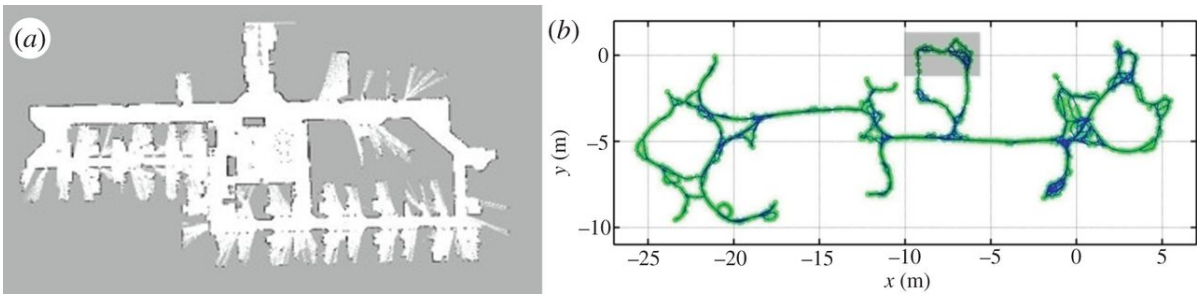

Figure 3.1: Figure (a) is an example of an occupancy grid map and figure (b) is a topological map [38, 39]

3.3 Types of Path Planning Algorithms

Path planning algorithms have been widely used in different domains. In the networking domain, they are used to find the optimal route for a data packet. In robotic applications they are used to find the optimal route to the goal point.

3.3.1 Dijkstra's Algorithm

Dijkstra's algorithm is one of the simplest and earliest algorithms to be developed for path planning. In this algorithm, the shortest path is computed by connecting nodes which have a minimum distance from the given set of nodes. Each connected node **v** or **u** has a weight **w(u,v)** associated with it. Each graph consists of vertices or nodes and weighted edges connecting two nodes. An array of the distance of each

node from the initial vertex **s** is maintained. This array **dist(s)** is initialized to infinity indicating that none of the nodes have been visited and all of them are at infinite distance from the initial vertex. The self-distance is initialized to zero. All the nodes in the graph are store in a queue **Q**. An empty set **S** is initialized to store the visited nodes. The algorithm works as follows:

- While the queue Q is not empty, a node **v** is popped from **Q** which is not already present in the set of visited nodes **S** and which has the smallest distance **dist(v)**. In the first run, the initial vertex **s** will be chosen since the distance of this node has been initialized to zero.
- Add node **v** to the set **S** to indicate that this node has been visited.
- Update the distance values of the nodes adjacent to the current node **v**. For each adjacent node **u** do the following:
  - If **dist(v)** + **w(u,v)** $<$ **dist(u)**, there is a new minimum distance for node **u** and this new minimum distance is updated.
  - If the condition in the previous statement is not true then there is no update for node **u**.

These steps are repeated until **Q** is empty. Now all the nodes have been visited and the array **dist** consists of the shortest path from the initial vertex **v**.

An example of the Dijkstra's algorithm can be seen from Figure 3.2, 3.3, 3.4 and 3.5. The starting node is A and the goal node is B. In the beginning, the distance to all the nodes is initialized to infinity as seen in Figure 3.2. In the next step, the distances to the neighbors of A are computed and these nodes are then added to the set of visited nodes. This is shown in Figure 3.3. The node with the least distance is selected in the next step and neighbors of that node are visited as shown in Figure 3.4. Figure 3.5 shows the final graph where all the nodes have been visited and the

final path from node A to B is selected based on the minimum distances to each node.

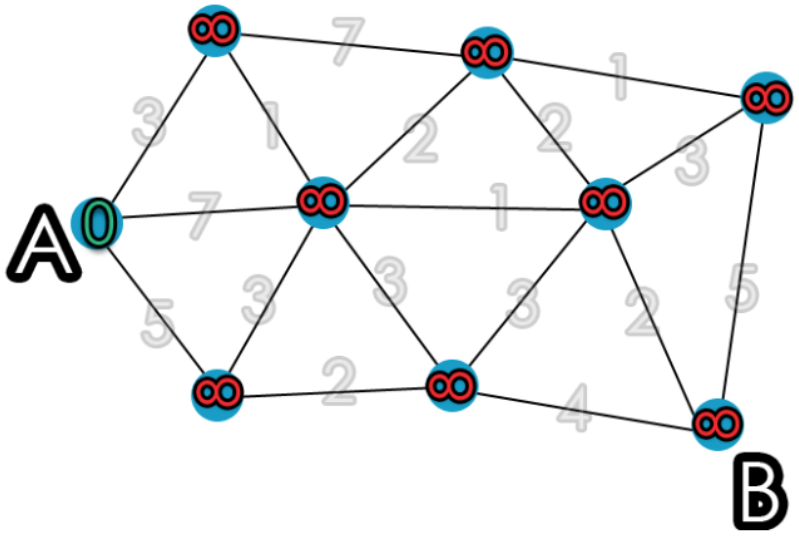

Figure 3.2: Step 1 of Dijkstra's Algorithm [40]

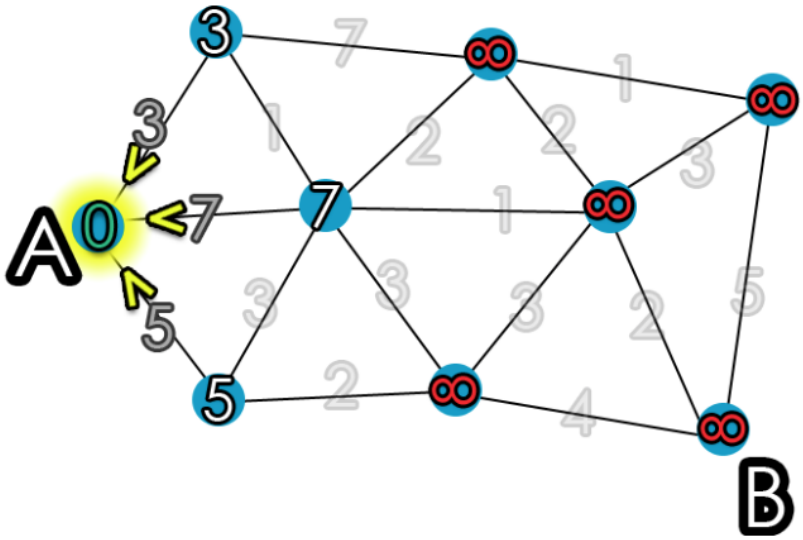

Figure 3.3: Step 2 of Dijkstra's Algorithm [40]

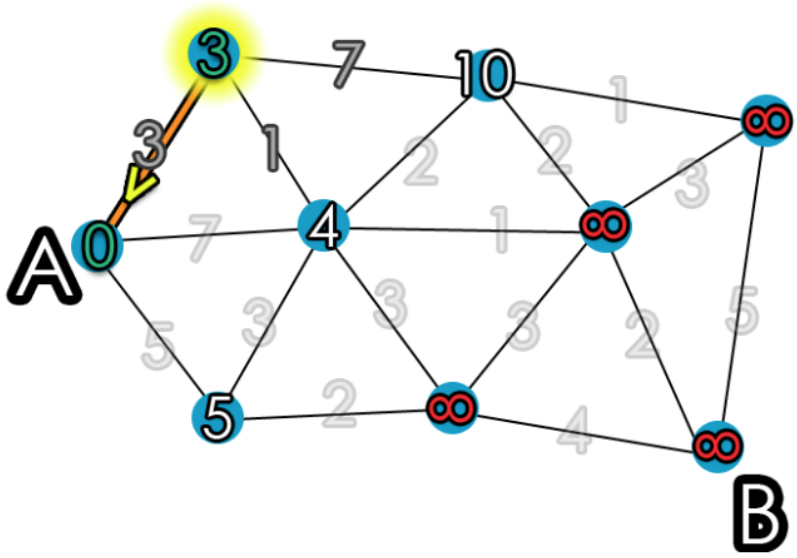

Figure 3.4: Step 3 of Dijkstra's Algorithm [40]

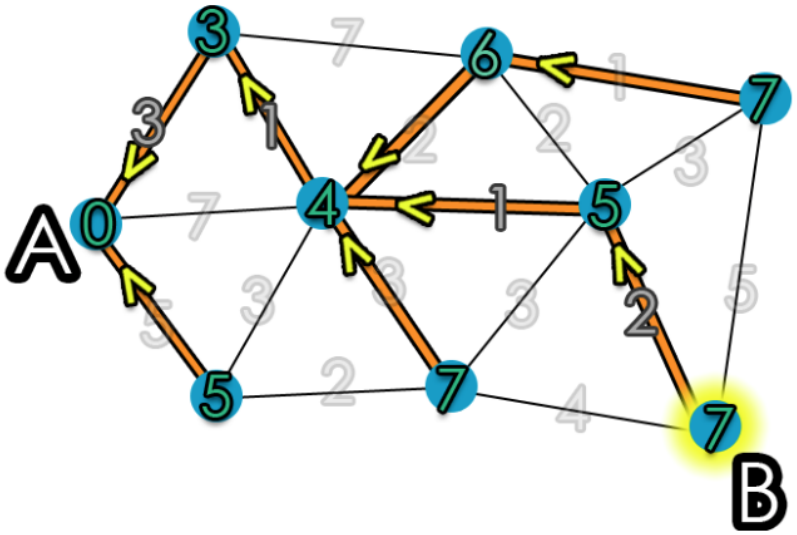

Figure 3.5: Final graph of Dijkstra's Algorithm [40]

One major disadvantage of this algorithm is that it does a blind search resulting in high computation time. It explores every adjacent node to find the minimum distance and hence for large graphs, the time taken to traverse the entire graph is also very high. Another disadvantage is that this algorithm cannot handle negative weights.

3.3.2 A* Algorithm

A* algorithm is an extension of Dijkstra's algorithm. It uses heuristic search to find the shortest path in less number of computations (as compared to Dijkstra's algorithm). This algorithm is tailored to explore paths only in the direction of the goal point. This is achieved by adopting a heuristic search. Breadth-First Search and Dijkstra's algorithm expand or explore in all directions. This means that paths to all locations from the start point will be available by using Breadth-First Search and Dijkstra's algorithm. But in the case of A* algorithm, only the paths that will lead to the goal point are explored.

A heuristic search approach for A* calculates the estimated distance to the goal and uses this parameter to expand in the directions which have the minimum distances with respect to the goal point. Higher priority is given to the nodes that have a lower estimated distance to the goal. So, the actual distance to the node is calculated (same as Dijkstra's algorithm) and in addition to that, the estimated distance to the goal is calculated. The estimated distance to the goal can be the Euclidean distance or the Manhattan distance between the current point and the goal point. This estimated distance to the goal point eliminates the nodes which do not expand in the direction of the goal point.

The A* algorithm works as follows:

- Initialize two lists **OPEN** and **CLOSE**. The **start** node is put in the **OPEN** list.

- While the **OPEN** list is not empty, calculate the total cost of movement denoted by $\mathbf{f(n)}$ where $\mathbf{n}$ is the current node. This is computed as a sum of the actual distance to the current node $\mathbf{g(n)}$ and the estimated distance to the goal node from the current node $\mathbf{h(n)}$. It is given by the following formula:

$$\mathbf{f(n) = g(n) + h(n)}$$

- Find the node **q** with the least **f(n)** value.
- Pop the current node **q** from the **OPEN** list.
- Generate the neighbors of the node **q** and set their parent node as **q**.
- For each successor of the selected node **q**,
  - check if the successor node **s** is the **goal** node. If it is the **goal** node then stop the search.
  - Compute the **g(s)**, **h(s)**, and **f(s)** values for the successor node **s**

    g(s) = **g(q)** + distance between successor node **s** and current node **q**

    h(s) = **h(q)** + distance between the **goal** node to the successor node **s**

    $$\mathbf{f(s)} = \mathbf{g(s)} + \mathbf{h(s)}$$

  - Check if there is a node with the same position as the successor node **s** in the **OPEN** list which has a lower value of **f(n)** than the successor node. If this condition is true then skip this successor node.
  - Check if there is a node with the same position as the successor node **s** in the **CLOSED** list which has a lower value of **f(n)** than the successor node. If this condition is true, skip this successor node. If this condition is not true then add the node to the **OPEN** list.
- Push the current node **q** onto the **CLOSED** list.

The value of **h(n)** (heuristic function) can be calculated in two ways:

- Exact heuristics: This can be done by pre-calculating the distance of each pair of cells. If there are obstacles in-between a pair of cells, then the distance will be different from the Euclidean distance. If there are no obstacles, then the Euclidean distance would be the distance between those two cells. This is a time consuming process. Also, the distance would be stored and the memory requirement would increase.

- Approximate Heuristics: This can generally be done in three different ways:

  1. Manhattan Distance: This the the sum of the absolute values of the difference between the x and y co-ordinates of goal node and the current node respectively. It can be expressed as follows:

$$h = abs(current\_node(x) - goal\_node(x)) + abs(current\_node(y) - goal\_node(y))$$

     Manhattan distance is used when the robot can move in only four directions (up, down, left, right).

  2. Diagonal Distance: This is the maximum of the absolute values of the difference between the x and y co-ordinates of goal node and the current node respectively. It can be expressed as follows:

$$h = max(abs(current\_node(x) - goal\_node(x)), abs(current\_node(y) - goal\_node(y)))$$

     Diagonal distance is used when the robot can move in 8 directions (diagonal, vertical and horizontal).

  3. Euclidean Distance: This is the distance between the current node and the goal node using the distance formula. It can be expressed as follows:

$$h = sqrt((current\_node(x) - goal\_node(x)^2) + (current\_node(y) - goal\_node(y))^2)$$

     Euclidean distance is used when the robot can move in any direction.

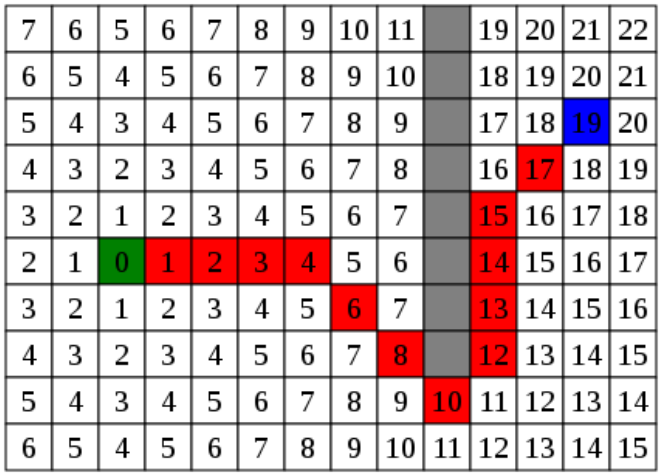

Figure 3.6: A* Algorithm on a grid map [41]

The A* algorithm can be explained with the help of Figure 3.6. The node in green is the start node, node in blue is the goal node and the nodes in red give the shortest path. The costs to all the nodes can be seen in the grid map. The cell with least cost to move and the shortest estimated distance to the goal is selected. The diagonal distance is used to estimate the distance to the goal node from the current node. Even though two or more cells have the same cost of movement, the next node to be selected is the one which has the least estimated distance to the goal node.

3.4 Introduction to Robot Operating System

This section gives a brief introduction to the Robot Operating System (ROS). It is a framework consisting of various libraries and in-built tools developed to aid and support the development of complex algorithms for robots.[42] ROS offers support to various robotic platforms. ROS has different versions like hydro, indigo, kinetic and many more. There are tutorials on how to create and build a ROS package. In ROS, an executable which is connected to the ROS network is called a node. There are different topics running on the ROS network. These topics publish their data in the form of ROS messages and receive data in the form of ROS messages. ROS nodes can publish messages to these topics or subscribe to receive messages from these

topics. There is a ROS Master which monitors the flow of messages form the different topics on the ROS network. There are tutorials on how to write ROS publishers and subscribers. These tutorials are generalized and would work on the robotic platforms supported by ROS.

There are specialized ROS tutorials for the TurtleBot. The ROS TurtleBot tutorials for different versions of ROS explain how to install and use the ROS packages on the TurtleBot. The indigo TurtleBot supports Trusty version of Ubuntu. The network is configured so that the remote PC can ssh to the TurtleBot PC. Rest of the setup consists of setting up the environment variables correctly on the remote PC as well as the TurtleBot PC. The IP address of the master node and the host node is specified in the .bashrc file for both the systems. The turtlebot_3D_SENSOR environment variable is set to kinect in the .bashrc file. Both the systems are configured to use the same ROS master node. There are various packages which are developed for map building and navigation of the TurtleBot. It also has a rviz_launchers package which is used to visualize the robot on the PC screen. The Kobuki base and 3D sensor camera (Kinect) are connected to the laptop.

ROS Gazebo simulator consists of a world which is a collection of different models, light and global parameters.[43] Each model is made up of links and joints. The hierarchy in Gazebo can be explained as follows:

- World
  - Scene
  - Physics
  - Model
    - Link
      - Collision
      - Visual

        - Sensor
        - Plugin
    - Plugin
- Plugin
- Light

Gazebo enables rapid prototyping and testing algorithms for robotics applications. Indoor as well as outdoor scenes can be created for testing purposes.

3.5 System Configuration

3.5.1 TurtleBot2 Specifications

The analysis phase of this thesis included the implementation of three path planning algorithms. These algorithms were implemented on TurtleBot2 (Figure 3.7). It is a low-cost robot kit with open-source software [44].

Figure 3.7: TurtleBot2 [44]

The TurtleBot2 hardware includes:

- Kobuki Base
- Asus Xion Pro Live

- Netbook (ROS Compatible)
- Kinect Mounting Hardware
- TurtleBot Structure
- TurtleBot Module Plate with 1 inch Spacing Hole Pattern

The software development environment includes:

- An SDK for the TurtleBot
- A development environment for the desktop
- Libraries for visualization, planning, and perception, control and error handling.
- Demo applications

The TurtleBot PC has Linux Trusty and ROS indigo running on it.

The Kinect sensor [45] on the TurtleBot (Figure 3.8).

Figure 3.8: Kinect Camera [45]

3.5.2 Assumptions

Following assumptions have been made for the implementation part of this thesis:

- The algorithm assumes that the robot maintains perfect odometry and does not take into consideration the co-ordinates of the robot.
- The map of the environment was built prior to the implementation.
- The starting point and the goal point of the robot are estimated by looking at the map.
- The length of any moving obstacle in the environment is not known in advance. This length of the object is assumed to be 1m. The width of the obstacle is also unknown. This width is set according to the sensor input. This is further discussed in Section 3.6.3.
- The height of the obstacle is assumed to be greater than or equal to the height of the sensor.
- The minimum distance that the robot has to maintain from obstacles (moving or static) is referred as the threshold and is set to 0.8 meters. This distance is selected by taking into consideration the minimum distance that the robot can percieve which is 0.44999 for the sensor installed on the TurtleBot and the distance it should maintain from obstacles.

3.6 Implementation

This section explains in detail the algorithms that have been implemented on the TurtleBot as well as ROS Gazebo.

3.6.1 Gmapping and AMCL

In the ROS TurtleBot tutorials, map creation of the environment and autonomous navigation in that map is explained. To start the master node, '**roscore**' command

was executed. To start the functionalities of ROS, the bringup file was launched by the following command:

roslaunch turtlebot_bringup minimal.launch

This command initialized the sensors and started the Kobuki base.

To create the map, the gmapping package was used. This is a laser based SLAM method. Following command was run on the ROS master node:

roslaunch turtlebot_navigation gmapping_demo.launch

The TurtleBot was then moved around with the help of a keyboard using the following command:

roslaunch turtlebot_teleop keyboard_teleop.launch

To visualize the movement of the robot and the simultaneous map creation, the RVIZ package was launched using the following command:

roslaunch turtlebot_rviz_launchers view_navigation.launch

After the TurtleBot had mapped the entire area of the required map, the map as seen in the RVIZ GUI was saved using the following command:

rosrun map_server map_saver -f path_to_the_folder/name_of_the_file

The map was stored in the form of an image with the file format as PGM. Another file containing the information regarding the map was also created by the map_server node. This file (YAML format) consisted of the path to the image file of the map (.pgm file), resolution of the map in meters per pixel, origin of the map, and occupied and free threshold in the map.

After the map was created, the TurtleBot was able to navigate to any point in the map by running the navigation stack and localization with AMCL. The navigation and AMCL localization is included in the turtlebot_navigation ROS package. Following command was used to export the map that was created using the slam_gmapping package:

export turtlebot_MAP_FILE=path_to_the_folder/name_of_the_file.yaml

The navigation demo launch file was run to localize the robot using the following command:

roslaunch turtlebot_navigation amcl_demo.launch

The RVIZ GUI was launched to visualize the robot and the map. The start point and the goal point was decided by looking at this visualization in RVIZ. Following command launched the RVIZ GUI:

roslaunch turtlebot_rviz_launchers view_navigation.launch –screen

The same procedure was used in ROS Gazebo using the following commands:

roslaunch turtlebot_gazebo turtlebot_world.launch world_file:=path_to_the_world_file.world

This command launched the specified world in Gazebo.

roslaunch turtlebot_gazebo gmapping_demo.launch

This command started the map building process.

The RVIZ GUI was launched using the same command as described for the TurtleBot. The map was saved using the map_server as described earlier.

The navigation in this map is same as described for the TurtleBot and launched using the following command:

roslaunch turtlebot_gazebo amcl_demo.launch map_file:=path_to_the_folder/name_of_the_file.yaml

The initial position and orientation of the robot was estimated by looking at the map and clicking the 2D Pose Estimate button and pointing the arrow in the direction of the orientation of the TurtleBot. This localized the TurtleBot in the map. The goal point was selected by clicking on the 2D Nav Goal and selecting a point on the map with the arrow pointing in the direction of the desired orientation of the TurtleBot. The robot was able to reach the goal point. But it failed to do so if the path or the goal was blocked.

The built-in path planner for the navigation stack in ROS consists of a base local

planner and a global planner. The base local planner consists of the Trajectory Rollout and Dynamic Window Approach(DWA). It consists of a cost map and a controller to issue velocity commands to the mobile base. A grid map is created around the robot and the velocities of traversing to the different locations in the grid are calculated based on a value function of the grid map. Trajectory Rollout samples from the set of achievable velocities over the entire forward simulation period given the acceleration limits of the robot, while DWA samples from the set of achievable velocities for just one simulation step given the acceleration limits of the robot. The global planner package is responsible for creating a path to the goal point. It consists of Dijkstra's algorithm, A* algorithm and NavFn algorithm implementations. But the planned path does not consider sudden obstacles. The move base package is used to combine the data from the base local planner and global planner to move towards the goal point. The rotate recovery package tries to rotate the robot by 360°and clear space. But this rotate recovery behavior is not guaranteed and the robot may hit the obstacle. This is further seen in Chapter 4.

To overcome the problem of dynamic obstacles, the robot must be able to sense the presence of obstacles as it is moved. After sensing the obstacle, the robot has to re-plan the path. The next section explains how this is achieved with the help of the optimised path planner algorithm.

3.6.2 A* algorithm

As discussed in the Section 3.3.2, A* algorithm is the most widely used path planning algorithm. The implementation of A* follows the tutorial given in [46].

Input to A* was a grid (map of the environment), start point and goal point. The start point and the goal point were decided by looking at the map and estimating the position and orientation of the TurtleBot. The map was built prior to running the A* algorithm. The current map gets published on the /map ROS topic when the gmapping node runs. Hence the map was obtained as a 1D array of data from

the /map ROS topic. This data from the ROS topic was first stored in a bag file. The bag file data was difficult to parse so it was then converted to a csv file. The /map ROS topic also publishes the height and width of the map, the origin of the map, resolution of the map in meters/pixels and the data array containing the actual occupancy grid map.

The next step was to parse the csv file to read the map and convert it into a 2D array and an image. This 2D array is the grid map for the A* algorithm. The map comprises of three values: 0, 100, and -1 indicate free space, occupied, and unknown respectively. In the image, white pixels are free space, black pixels are occupied and gray pixels are unknown. The known (static) obstacles in the map are considered as walls in the grid that is given as input to the A* function. The A* algorithm plans a path which avoids these obstacles or walls. For the ease of avoiding these obstacles, all the obstacles were expanded by 2 pixels on each side. This created a buffer or avoidance region around the obstacles.

The image of the map created was used in selecting the start point and the goal point. The start point and goal point were selected by clicking on the image of the map consecutively. First mouse click was the start point and the second mouse click was the goal point. These were the inputs to the A* function. In the A* function, the shortest path was computed as explained in Section 3.3.2.

For implementation on the TurtleBot, there should be directions associated with the co-ordinates. The robot can move in four directions: up, down, left, right. The directions that were assigned to each point in the final path were based on the destination. That means, if the goal point had co-ordinates (4,3) and the point just before the goal point was (3,3), then the direction associated with the point (3,3) was 'down' because the robot had to move down to reach the goal point of (4,3). In the same way, all the points in the final path were assigned a direction based on the point in the final path which succeeded the current point. The initial direction was

an input from the user. A priority queue was implemented for storing the visited nodes. This priority queue had the location from the grid and a priority associated with the location.

The algorithm that was implemented does not consider the robot co-ordinate system and assumes perfect odometry as mentioned in Section 3.5. To consider the actual co-ordinates of the TurtleBot, the real-world co-ordinates would need to be transformed into the robot co-ordinate system and then subsequently transform all the points of the map into the robot co-ordinate system. This is the future scope of the thesis and will be discussed in Chapter 5.

For the TurtleBot to move, velocity messages need to be published. Velocity commands are of two types: linear velocity and angular velocity. These velocity messages are published on the 'cmd_vel_mux/input/navi' Twist topic. The linear and angular velocities can be in x, y or z direction. In this implementation, linear velocity in x direction (forward velocity) and angular velocity in z direction (turn velocity) has been used. When the robot has to move forward, linear velocity is set to 0.2 m/s and and angular velocity is made zero. While turning, the angular velocity is made positive or negative as required by the angle for the turn (clockwise or anti-clockwise). The TurtleBot can turn by 90 degrees in this implementation because the TurtleBot can only move in four directions (up, down, left, right).

This algorithm has the ability to avoid static obstacles (like walls) which are already present in the map. The A* algorithm avoids the locations (cells) in the map that have been marked as occupied. But it fails in the presence of dynamic obstacles. If the obstacle is not present in the map, the path will be planned irrespective of the presence of the obstacle. For example, if a cardboard box of height equal to or greater than the height of the kinect sensor on the TurtleBot is placed in the path of the TurtleBot, it would collide into the box. To overcome this, the optimized path planner algorithm was developed and implemented on the TurtleBot. This path

planner algorithm is further explained in Section 3.6.3.

3.6.3 Optimized Path Planner Algorithm

The A* algorithm that was implemented on the TurtleBot did not have the ability to detect and avoid obstacles. The A* algorithm is not optimized for turns either. The path planner implemented in this work is optimized for turns. It takes the least number of turns. Turning a robot introduces noise and the turning is not always accurate. Hence the path planner turns a minimum number of times. The obstacles considered for the purposes of this thesis must have minimum height equal to the height of the kinect sensor installed on the TurtleBot. The other feature of these obstacles is that they are moving obstacles. The obstacles may or may not possess a velocity of their own. Most of the algorithms that were reviewed in the literature survey dealt with a specific set of obstacles. But the nature of obstacles in indoor environments is not fixed. The work in this thesis addresses this problem and tries to overcome this drawback. Obstacles can be in the form of chairs, tables, cardboard boxes which can be moved from one place to another. Obstacles can also be humans moving around in that space.

In order to deal with a wide variety of obstacles, the robot needs to continuously sense the environment and based on the feedback from the sensed data, re-plan its path. For example, if a box is suddenly placed in the path of the robot (assuming the box satisfies the height criteria mentioned in Section 3.5), the robot has to re-plan its path around the obstacle. To do so, it needs to alter the already existing map and indicate that there is an obstacle at the location of the box. Now if the same box is removed and the sensor is still pointed in the direction of the box (and in the range of the previously placed box), the map should again be updated indicating that there is free space at that location. Now the robot does not need to take an alternative path and hence the path needs to be re-planned again. The re-planning of the path has to be based on the sensor input data and needs to be a continuously improving

process. If the robot re-plans the path once around the obstacle, it should be able to sense if the obstacle is removed from its path and that it does not need to take a detour around the obstacle. Hence, the process of re-planning the path depends on the following two factors:

- When an obstacle is detected in front of the robot at a distance less than or equal to a fixed threshold.
- When a previously detected obstacle is no longer present in the path of the robot.

The modified path planner algorithm plans the path in the same way as explained in Section 3.6.2. As the robot starts moving towards the goal point, the messages published on the /scan 'LaserScan' topic are monitored. A subscriber node is created which subscribes to these messages. The threshold for an object to be classified as an obstacle is set at 0.8 meters. This means that if the readings for the scan ranges from the sensor are 0.8 meters or less than that then there is an obstacle present in front of the robot and the robot needs to take a different path to reach the goal point, that is, path needs to be re-planned. If the robot is going to turn in the next step, this obstacle does not matter as it may not be in the path. So the path re-planning takes place if the previous condition of the minimum threshold is met and if the robot's next step is in the direction of the sensed obstacle.

As explained earlier, when an obstacle is detected, the map is altered to indicate that there is an obstacle in front of the robot which is 0.8 meters away. This part of the map is made 100 as it denotes occupied space. Since the dimensions of the obstacle are unknown, it sets the width of the obstacle as the cells from the starting point of the obstacle to the end point of the obstacle to 100. If the width of the obstacle is greater than the angular range of the sensor, then all the cells in the angular range at a distance of 0.8 meters will be set to 100. If the width of the

obstacle is lesser that the angular range of the sensor, then only those cells are set to 100 which lie in the region between the start point of the obstacle to the end point of the obstacle. There is no way of knowing length of the obstacle either. So for this implementation the length is set to 1 meters. This is an assumption of the length of the obstacle as mentioned in Section 3.5. If the obstacle has a length greater than 1 meters, it would again re-plan the path as it would detect the same object as an obstacle again. So it would be an iterative process. These assumptions have to be made because there should be a minimum distance equal to the threshold that has to be maintained between the obstacle and the robot.

After the new map with the obstacle is created, the path planner algorithm is used again to re-plan the path. But for this call to the path planner algorithm function, the grid input is only from the current position of the TurtleBot to the goal point of the TurtleBot with a minimum threshold distance on either sides of the TurtleBot. So the grid that the path planner receives is relatively smaller as compared to the first run. This reduces the time for computing the new path. If the shortest path is found, then the TurtleBot starts moving towards the goal point as prescribed by the new path which avoids the obstacle in front of it. If no path is found, the grid is expanded by one cell in all directions and the path planner algorithm function is called again. This process repeats itself until the grid expands to the maximum size, that is, the original size of the map. If a path is still not found, then the grid is set to the original static grid and the path is re-planned based on the original grid.

To take care of obstacles which appear or disappear in the surroundings (in the range of the sensor), the message received from the '/scan' topic is compared to the data in the map. If the sensor input detects an obstacle 3 meters away and at an angle of 10 degrees, the current grid map is checked at that cell to see if the map indicates free or occupied. If the map indicates that it is free, this is updated to occupied. So the map is continuously is updated as the TurtleBot moves towards the goal point.

The map is updated only in the range of the sensor. If there is any movement which is not in the angular or the distance range of the sensor, it is not detected and it is not updated in the map.

This optimized path planner algorithm is able to detect obstacles and avoid the obstacles to reach the goal point. In the case where the goal point is blocked or the algorithm is unable to reach the goal point, it would reach the nearest point to the goal point and keep on polling the data from the sensor to find a path to the goal point.

CHAPTER 4: RESULTS

In this chapter, the results obtained from implementing the algorithms explained in Section 3.6 have been evaluated. The algorithms have been implemented on ROS Gazebo simulator and TurtleBot.

4.1 Navigation on ROS Gazebo using Gmapping and AMCL

4.1.1 Output from the Built-in Path Planner

A world was built in Gazebo using the Building Editor. The static world is shown in Figure 4.1. It consists of one dumpster, two cabinets and three bookshelves. The TurtleBot starts by default at (0,0,0). This world is built in comparison to the ECE lab at UNCC.

The map of this static world was built using the gmapping node as explained in Section 3.6.1 which can be seen in Figure 4.2. This map was used for navigation using the AMCL localization and navigation stack. The world was modified by adding objects to the world, this can be seen in Figure 4.3 where new objects have been added to the world. These objects include solid cube, solid cylinder, cardboard boxes, and a TurtleBot. These serve as obstacles in the map. Since these obstacles are not present in the previous map, they are not considered while planning the path. For the in-built path planner in the navigation stack, if the new obstacle is in the range and the vision of the sensor, then the path planner is able to navigate around that object.

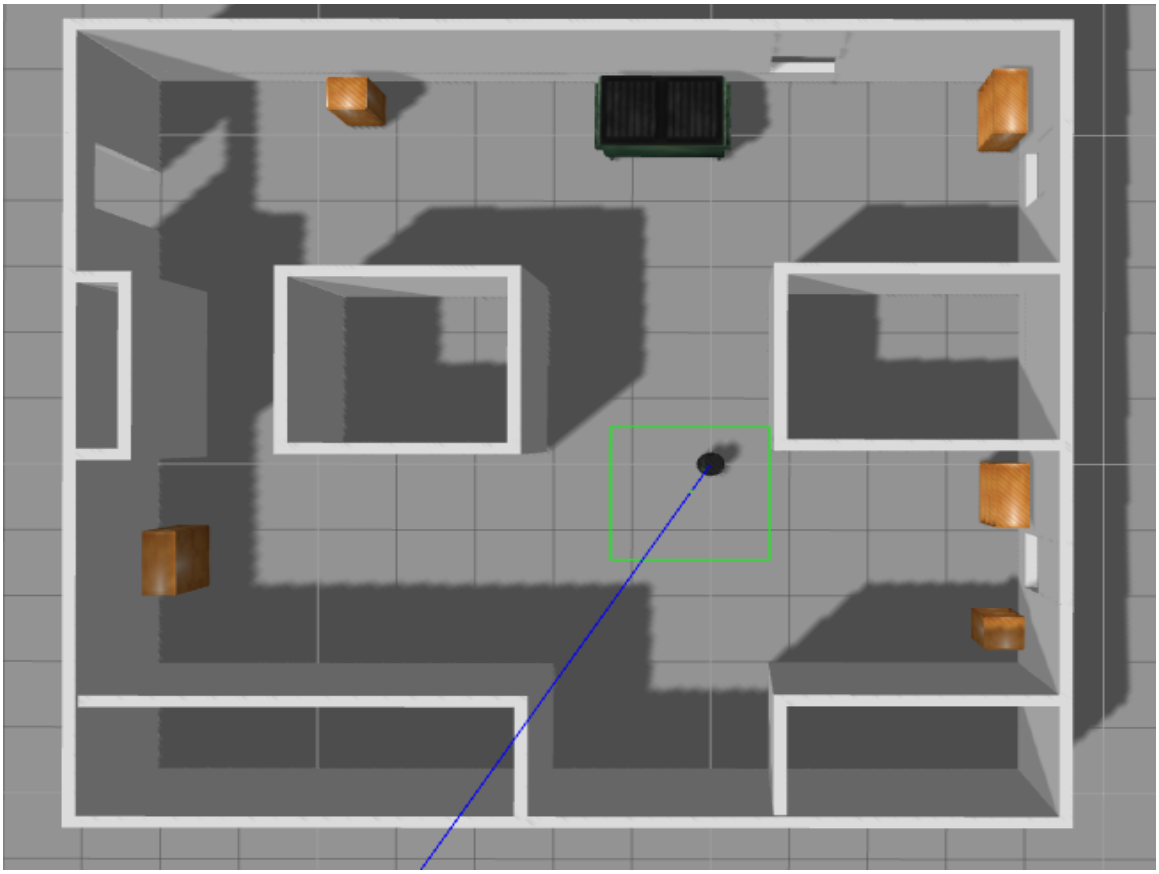

Figure 4.1: Static World

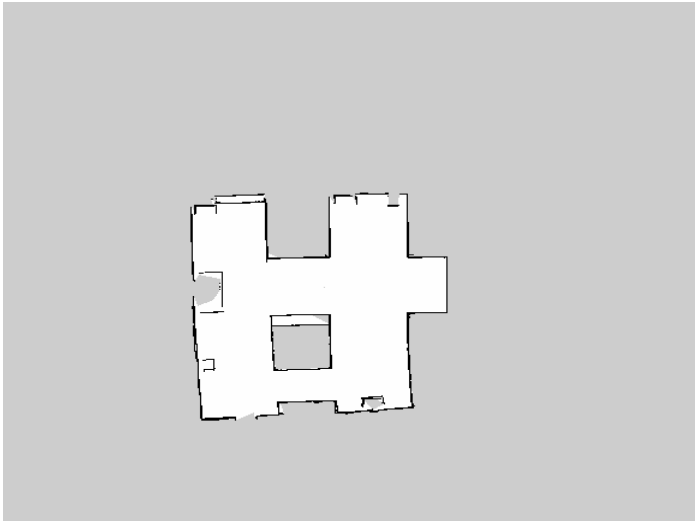

Figure 4.2: Map from Gmapping

The map and the world look as shown in Figure 4.4 before the navigation begins. The first step for navigation is to select the start point and the goal point. The start

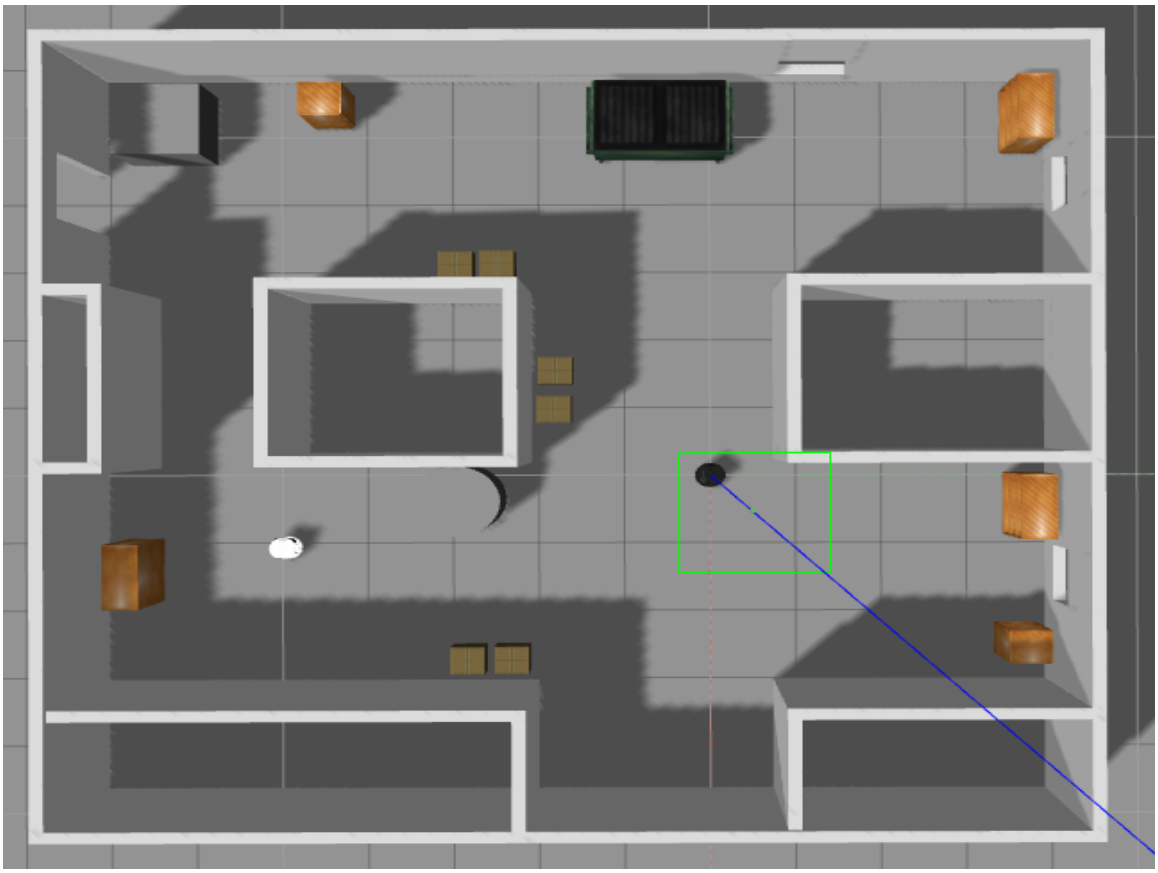

Figure 4.3: New Static World

point is selected by the 2D pose estimate button. This indicates the current position and orientation of the TurtleBot in the map. The goal point is selected by the 2D Nav Goal button. The orientations of both the points are indicated by the direction of the arrow. The selection of start point can be seen in Figure 4.5 and selection of goal point can be seen from Figure 4.6.

The path that is planned can be seen in Figure 4.7.

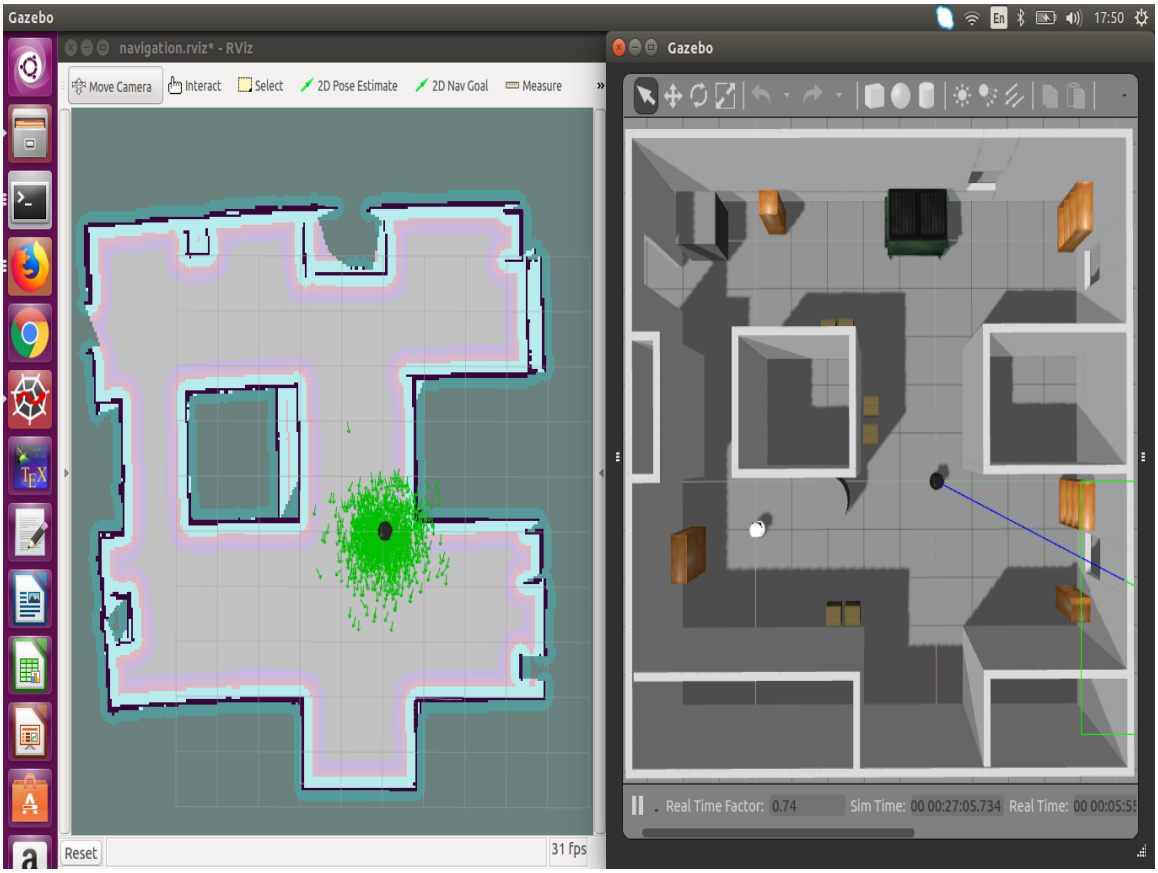

Figure 4.4: Map and World Prior to Navigation

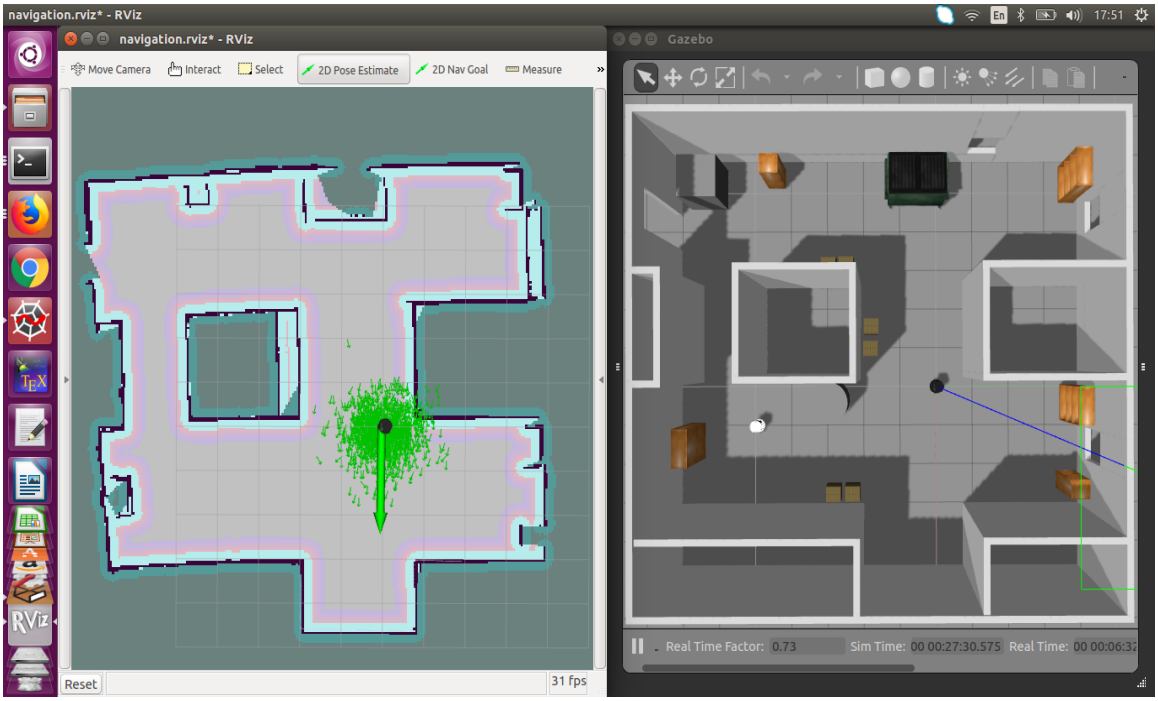

Figure 4.5: Start point in the map

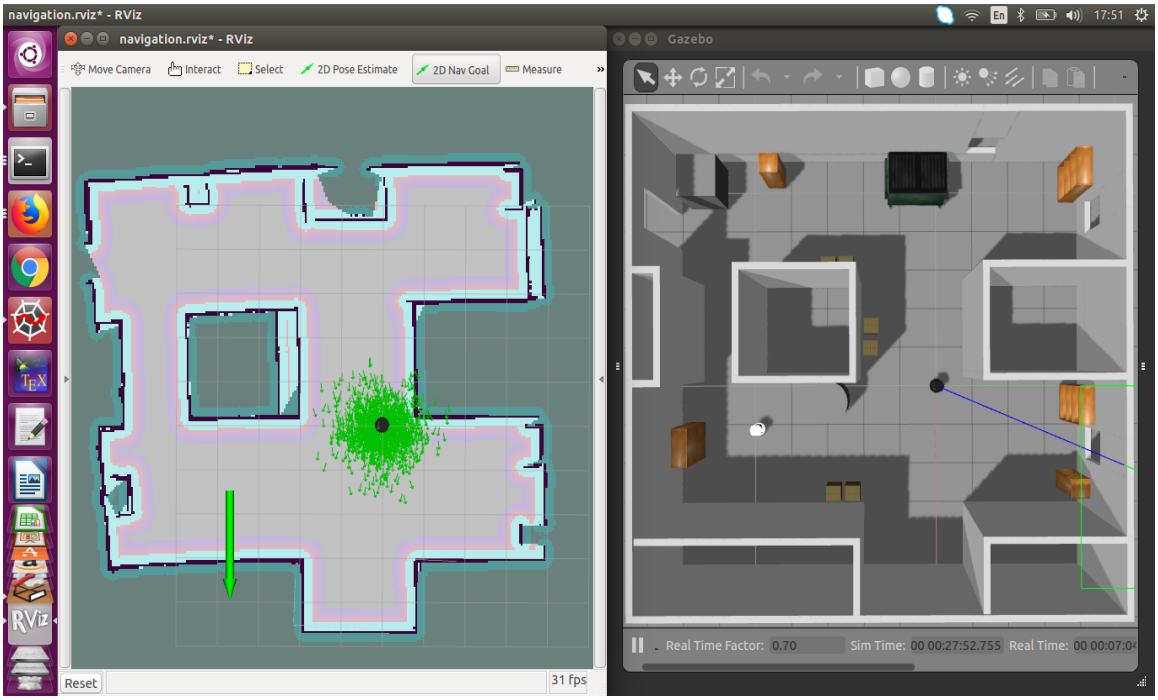

Figure 4.6: Goal point in the map

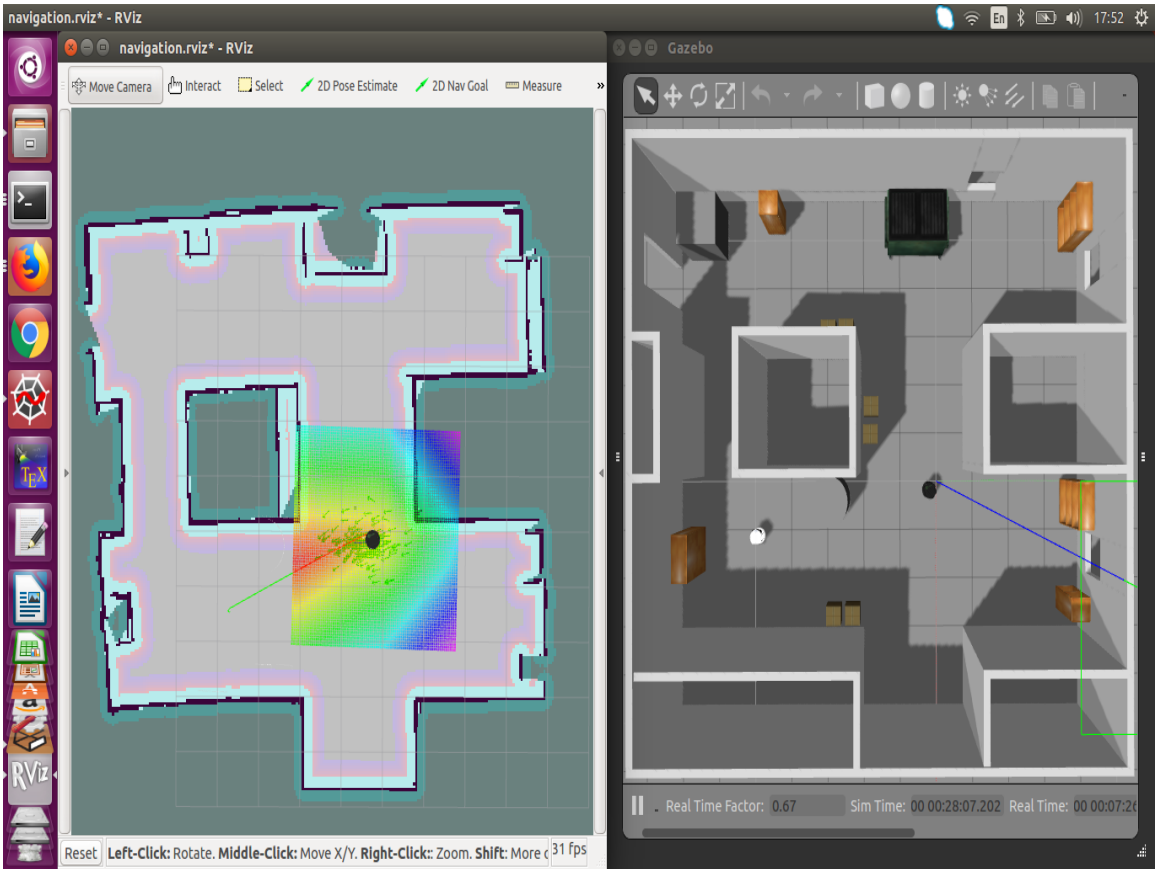

Figure 4.7: Planned trajectory for the TurtleBot

Figure 4.8 shows the trajectory of the TurtleBot as it goes from the start point to the goal point. TurtleBot reaches the goal point as shown in Figure 4.9 and 4.10.

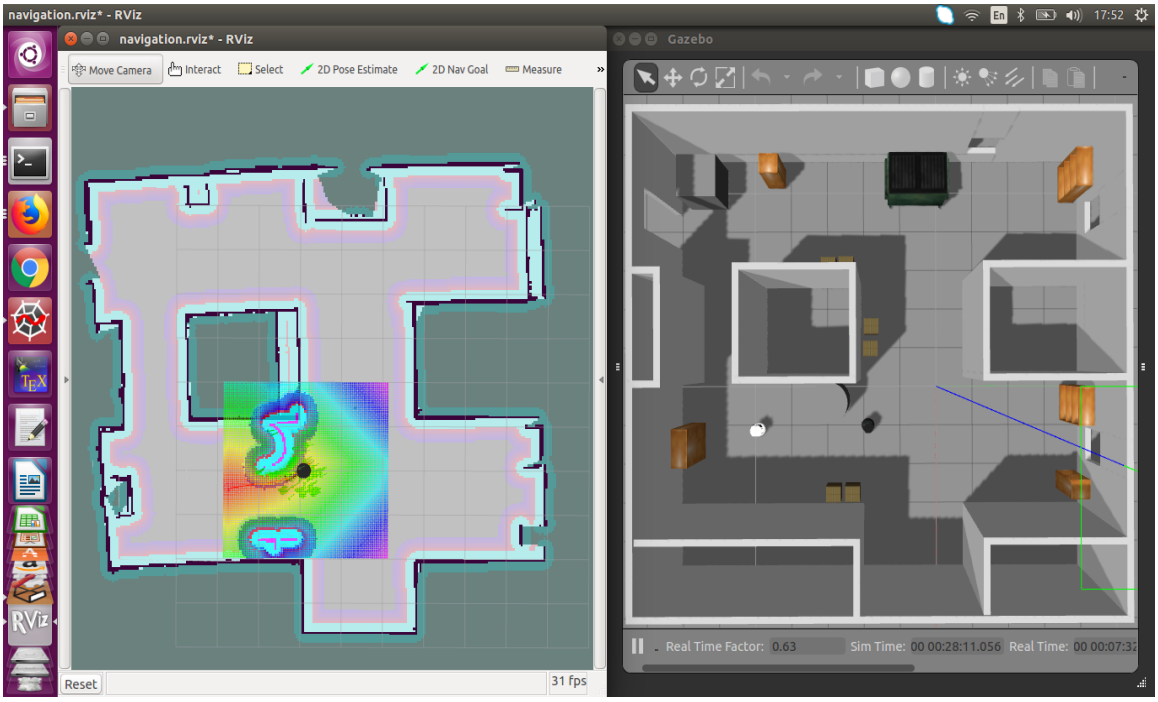

Figure 4.8: Mid-path trajectory for the TurtleBot

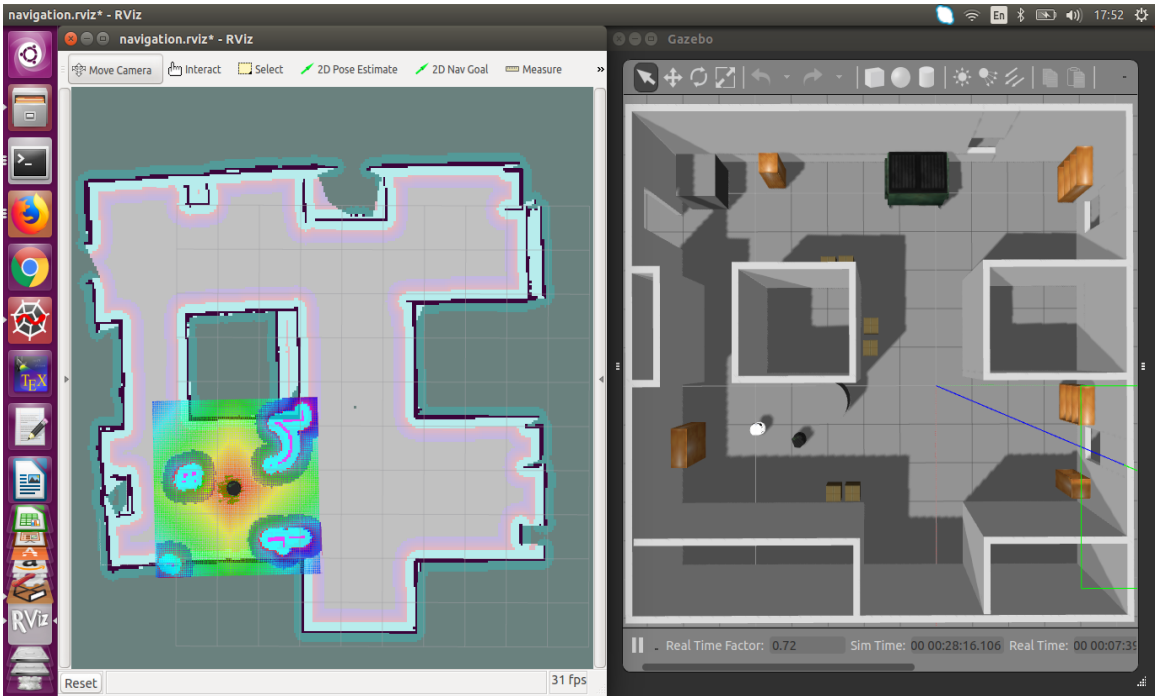

Figure 4.9: TurtleBot reaches the goal point

```
[ INFO] [1523915134.119520163, 1423.712000000]: odom received!
[ INFO] [1523915521.785442149, 1685.856000000]: Got new plan
[ INFO] [1523915523.330225438, 1686.856000000]: Got new plan
[ INFO] [1523915524.763071816, 1687.660000000]: Got new plan
[ INFO] [1523915526.868857919, 1688.856000000]: Got new plan
[ INFO] [1523915528.355186085, 1689.857000000]: Got new plan
[ INFO] [1523915529.820853344, 1690.856000000]: Got new plan
[ INFO] [1523915531.525620088, 1691.856000000]: Got new plan
[ INFO] [1523915533.054703278, 1692.856000000]: Got new plan
[ INFO] [1523915534.411574797, 1693.857000000]: Got new plan
[ INFO] [1523915535.833146516, 1694.856000000]: Got new plan
[ INFO] [1523915537.237440683, 1695.856000000]: Got new plan
[ INFO] [1523915538.893848410, 1696.856000000]: Got new plan
[ INFO] [1523915539.922920928, 1697.458000000]: Goal reached
```

Figure 4.10: Messages on the terminal

4.1.2 Limitations of Built-in Path Planner

Section 4.1.1 shows the implementation when the navigation is a success. As mentioned in Section 3.6.1, the in-built planner fails when a sudden obstacle is placed in front of the robot. This means that the sensor on the robot must see the obstacle before it plans the path. If the robot sees the obstacle before path planning, it is able to avoid the obstacle. But this is not true for all the obstacles in the map. The sensor is not able to perceive objects or obstacles (previously not present in the map) if they are outside the range or the vision of the sensor.

Figure 4.11 and 4.12 show the presence of an obstacle in the planned path of the TurtleBot. Figure 4.13 show the output on the terminal which indicates that the path is being re-planned around the obstacle. The TurtleBot collides with the obstacle and then tries to re-plan the path. Figure 4.14 and 4.15 show that a path cannot be produced and the re-planning process is aborted.

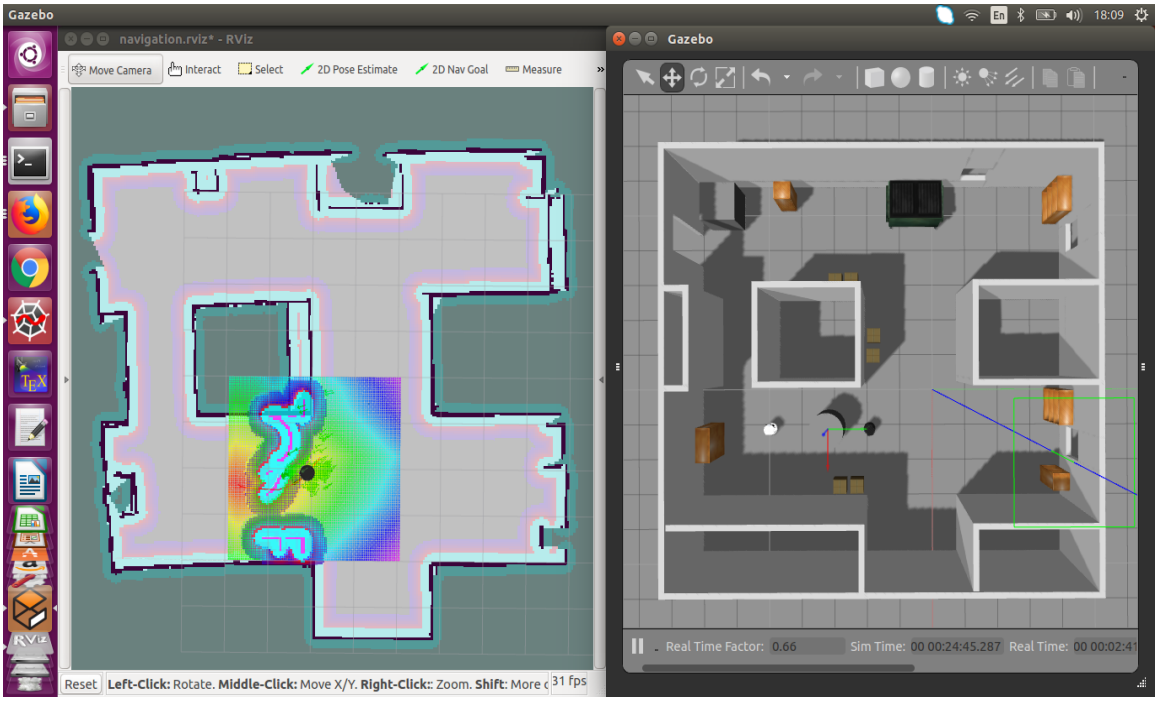

Figure 4.11: Obstacle In Map

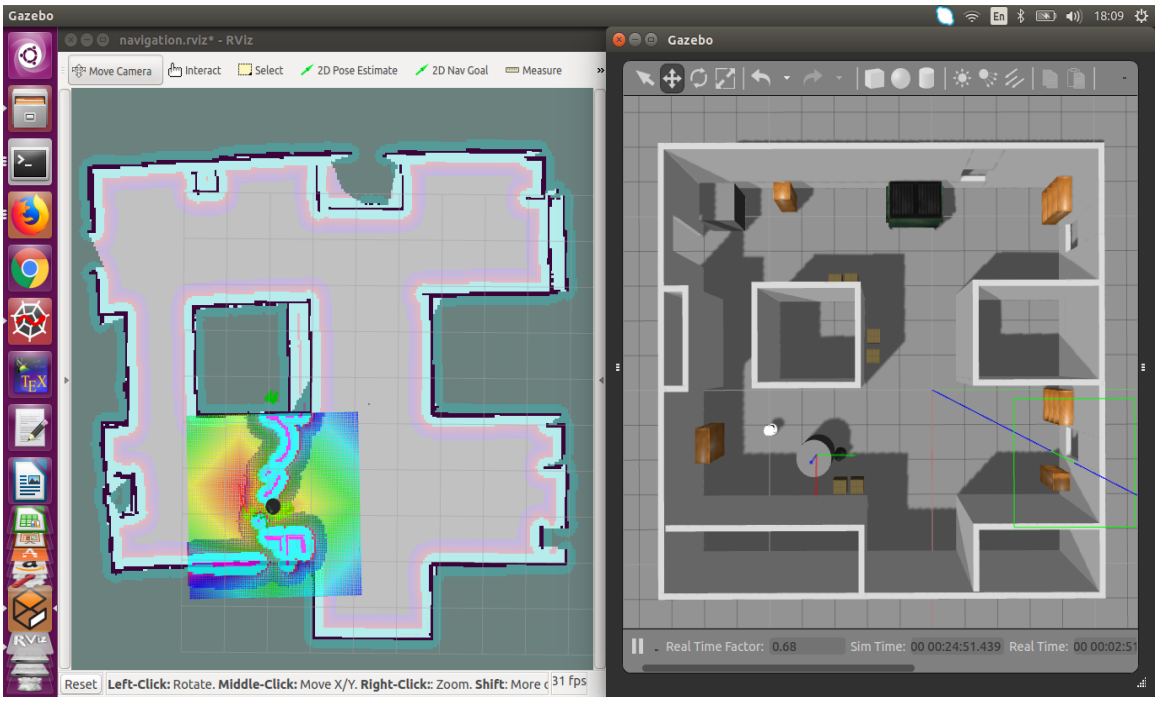

Figure 4.12: Obstacle In Map

```
INFO] [1523916561.107493526, 1490.759000000]: Got new plan
INFO] [1523916562.671261010, 1491.759000000]: Got new plan
INFO] [1523916564.402400817, 1492.759000000]: Got new plan
INFO] [1523916566.094798676, 1493.759000000]: Got new plan
INFO] [1523916567.636619013, 1494.759000000]: Got new plan
INFO] [1523916569.220257865, 1495.759000000]: Got new plan
INFO] [1523916570.865436632, 1496.759000000]: Got new plan
INFO] [1523916572.432999820, 1497.760000000]: Got new plan
WARN] [1523916573.360040363, 1498.360000000]: Clearing costmap to unstuck robo
(3.000000m).
INFO] [1523916574.018754094, 1498.759000000]: Got new plan
INFO] [1523916575.509623527, 1499.759000000]: Got new plan
INFO] [1523916576.704290381, 1500.559000000]: Got new plan
INFO] [1523916578.597067607, 1501.759000000]: Got new plan
INFO] [1523916579.946925738, 1502.559000000]: Got new plan
INFO] [1523916581.921266804, 1503.759000000]: Got new plan
INFO] [1523916583.535771406, 1504.759000000]: Got new plan
INFO] [1523916585.296942628, 1505.759000000]: Got new plan
INFO] [1523916587.210160003, 1506.759000000]: Got new plan
INFO] [1523916588.762972963, 1507.759000000]: Got new plan
INFO] [1523916590.420741854, 1508.759000000]: Got new plan
WARN] [1523916590.728890230, 1508.959000000]: Rotate recovery behavior started
```

Figure 4.13: Output on the terminal when the TurtleBot collides with an object and tries to re-plan the path

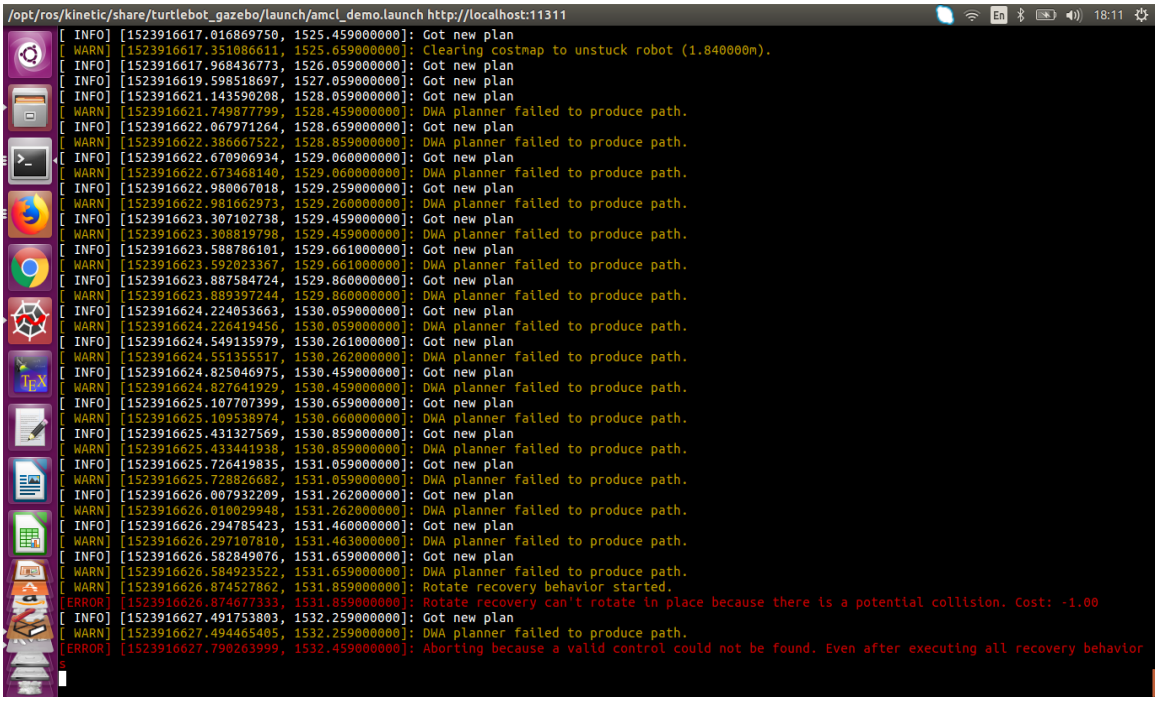

Figure 4.14: Output on the terminal after the built-in planner tries to re-plan the path and fails to reach the goal point

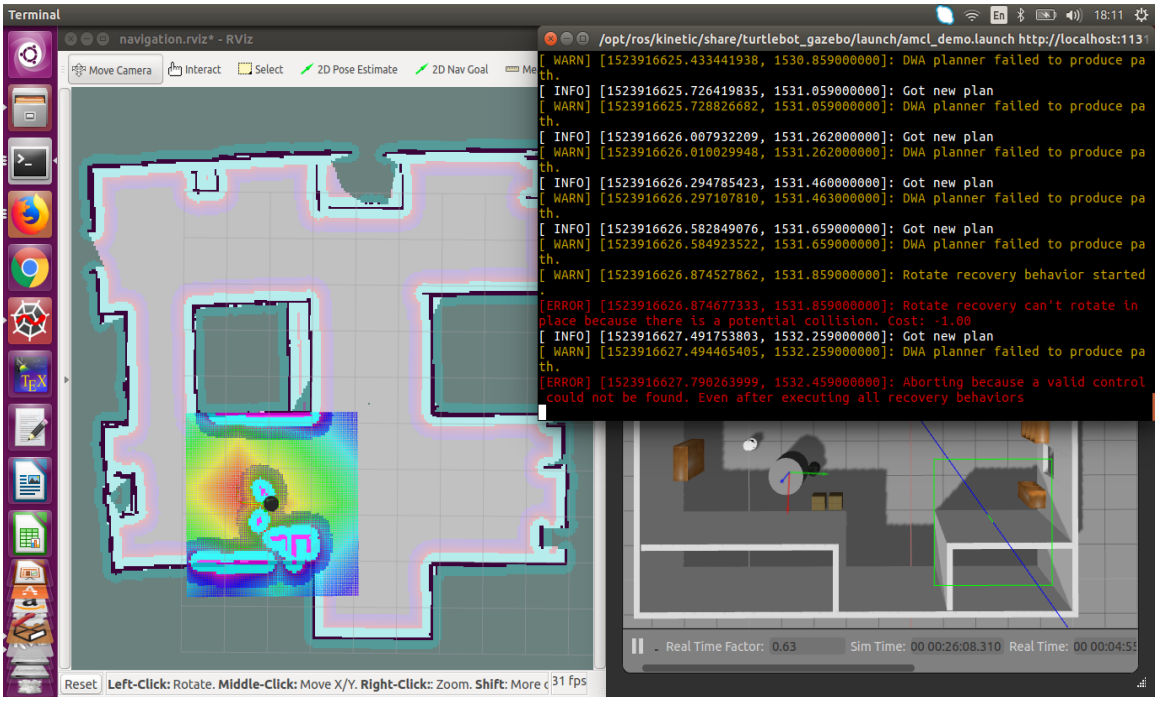

Figure 4.15: Output on the terminal indicating that there is collision with an object in the world. Map and world displaying the collision of the TurtleBot with the cylindrical object

4.2 Navigation on ROS Gazebo using A* Algorithm

The path planner algorithm runs when there are no additional obstacles in the map. It runs the basic A* path planner as described in Section 3.6.2. The robot is able to reach the goal point as long as the path to the goal point is not blocked. This algorithm does not take any readings from the sensor into account before planning the path. The difference between the built-in path planner and the A* algorithm is that the built-in planner considers the obstacles in the range and vision of the robot before planning the path. Whereas the A* algorithm considers only the obstacles which are in the static map to plan the path.

4.3 Navigation on ROS Gazebo using the Optimized Path Planner Algorithm

4.3.1 Output from the Optimized Path Planner Algorithm

In the optimised path planner algorithm the input from the sensor is continuously monitored to detect obstacles in the path of the TurtleBot. Obstacles are detected if they are at a minimum threshold distance (0.8 meters) from the TurtleBot as explained in Section 3.6.3. AMCL is used for localizing the TurtleBot. The implementation of this algorithm can be seen from Figure 4.16, 4.17, 4.18, 4.19, and 4.20. The robot is able to reach the goal point by avoiding the moving obstacle.

In Figure 4.16, the world and the map can be seen before the path is planned. In Figure 4.17, the robot has already started moving along the planned path. The solid cylindrical object is moved. This object is not in the exact path of the robot but it is in the range and vision of the robot. Hence the TurtleBot has to re-plan the path because the object is in the range and vision of the TurtleBot and the TurtleBot will not be able to avoid the obstacle if it continues on the current path. The path is updated and the robot starts moving along the new (updated) path. The solid cylidrical object is again detected as an obstacle as shown in Figure 4.18. The path is re-planned again to avoid this obstacle (Figure 4.19). The TurtleBot is able to reach the goal point while maintaining a minimum threshold distance between the robot and the obstacle as shown in Figure 4.20.

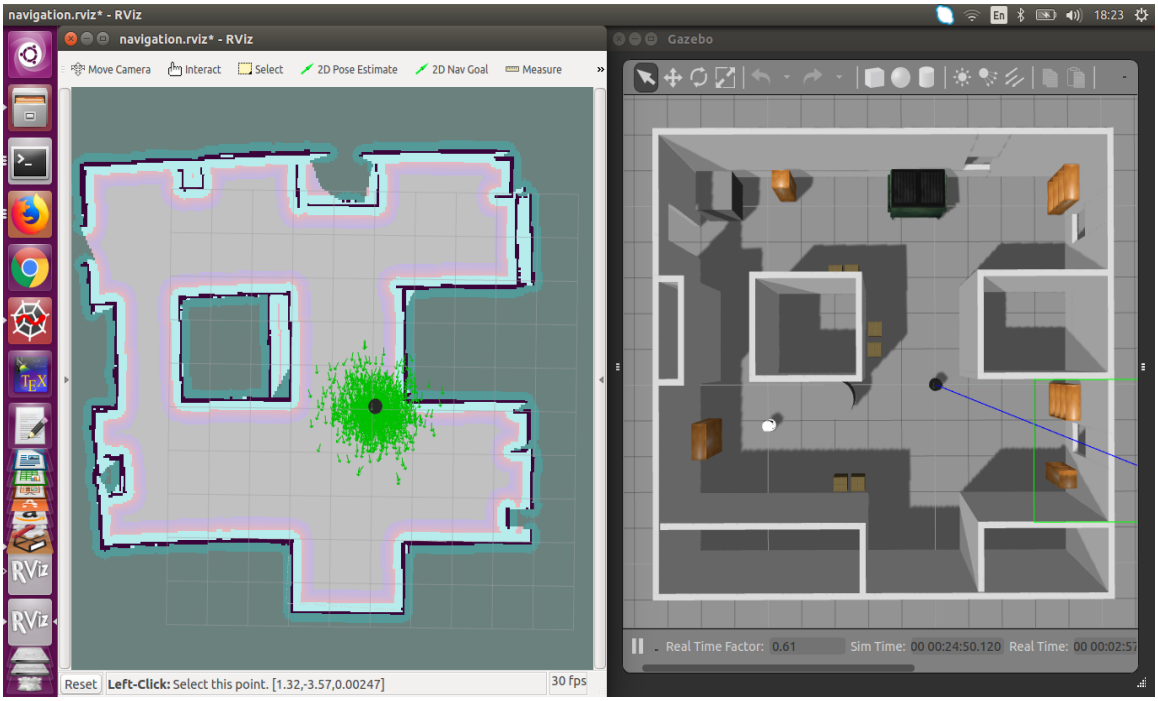

Figure 4.16: Map and World Prior to Navigation

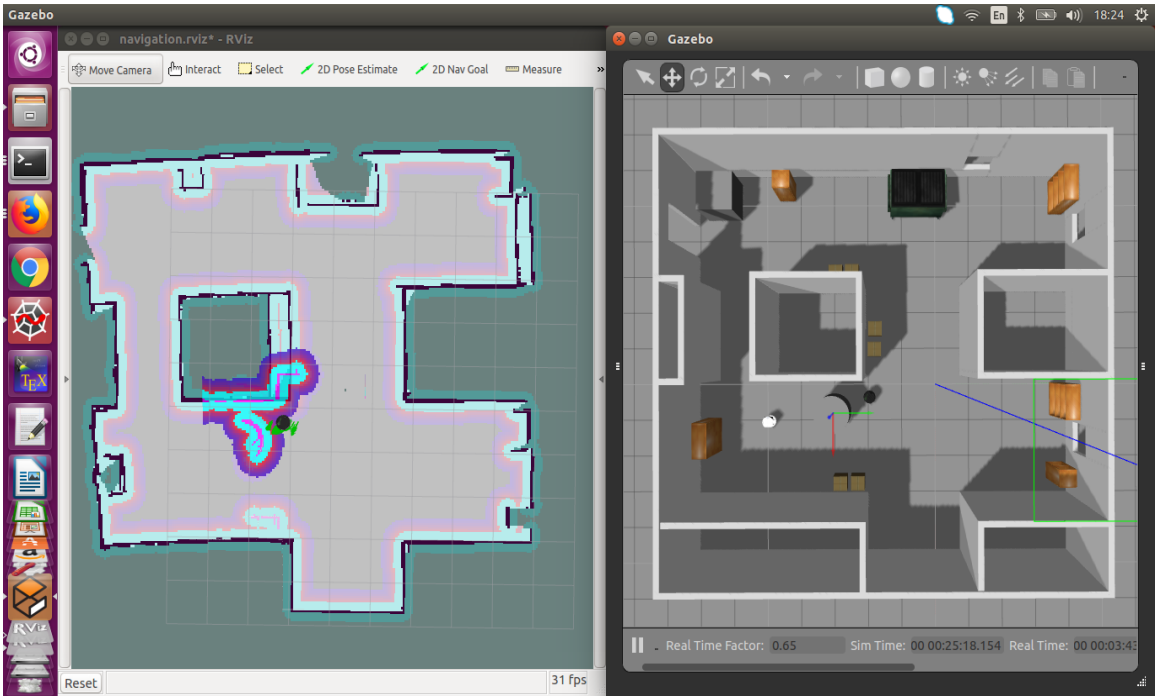

Figure 4.17: Solid cylinder is moved from its location which is detected as an obstacle in the path of the TurtleBot

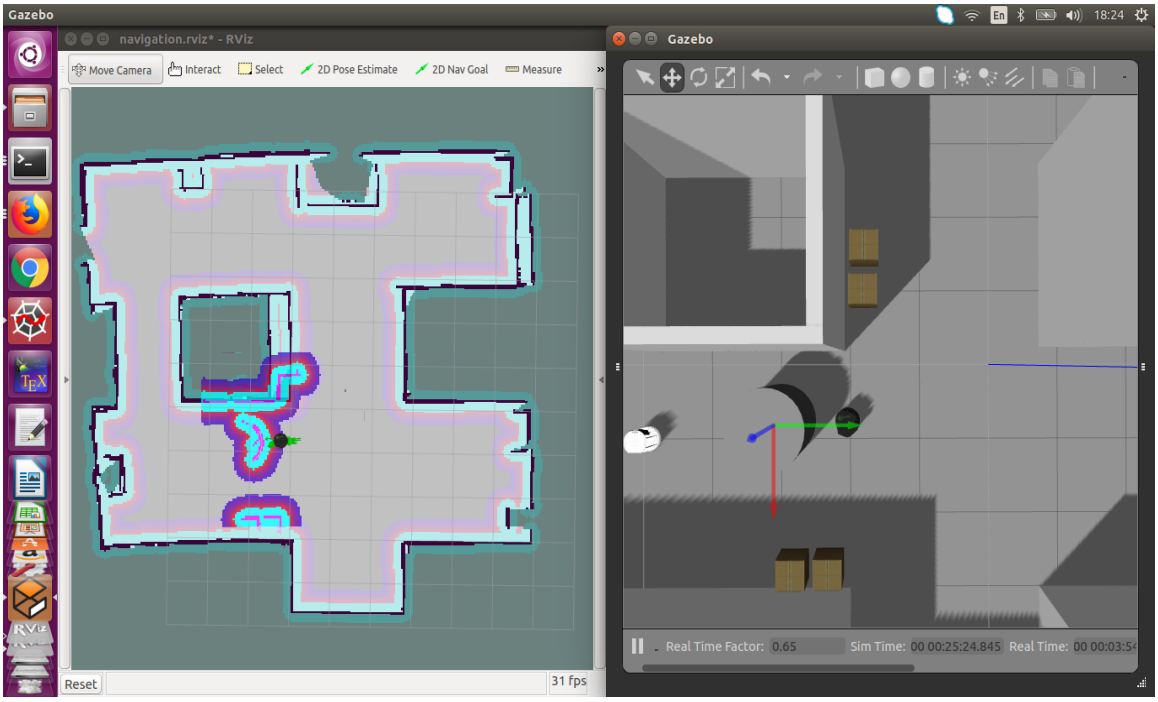

Figure 4.18: Solid cylinder moved again and detected again as an obstacle in the path of the TurtleBot

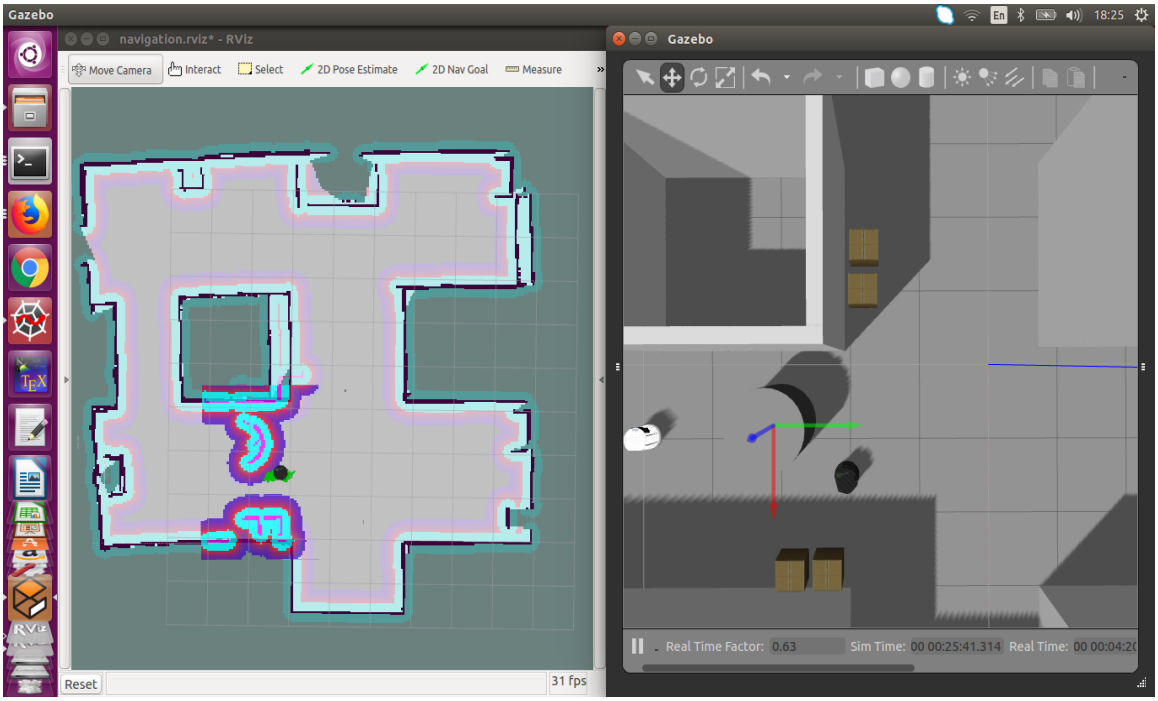

Figure 4.19: TurtleBot is able to find a path around the obstacle

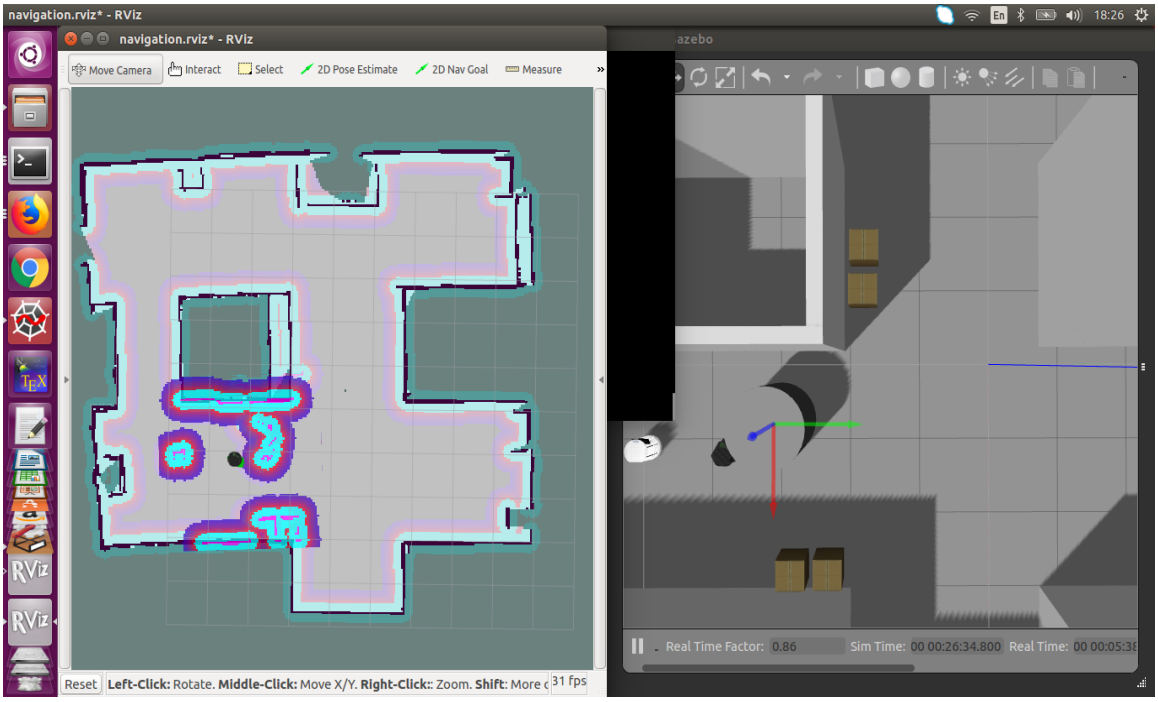

Figure 4.20: Goal point reached after re-planning the path

4.3.2 Limitations of the Optimized Path Planner Algorithm

As mentioned in Section 3.5.2, the obstacles have to be in the range and vision (distance and angular range) of the sensor installed on the TurtleBot. If the obstacle is not in the range and vision of the sensor on the TurtleBot, the algorithm fails to re-plan the path. This is demonstrated by placing a small table in the world as shown in Figure 4.21. This table has one leg in the center of the table and the height of the table top from the ground level is equal to the height of the TurtleBot. The sensor is unable to perceive the table top but it can detect the leg of the table as an obstacle. This results in the TurtleBot hitting the table top and unable to move forward because the sensor readings do not indicate an obstacle. The edge of the wall is detected as an obstacle (Figure 4.22) but the table top is not detected as an obstacle. This can be seen from Figure 4.23, 4.24, 4.25 and 4.26.

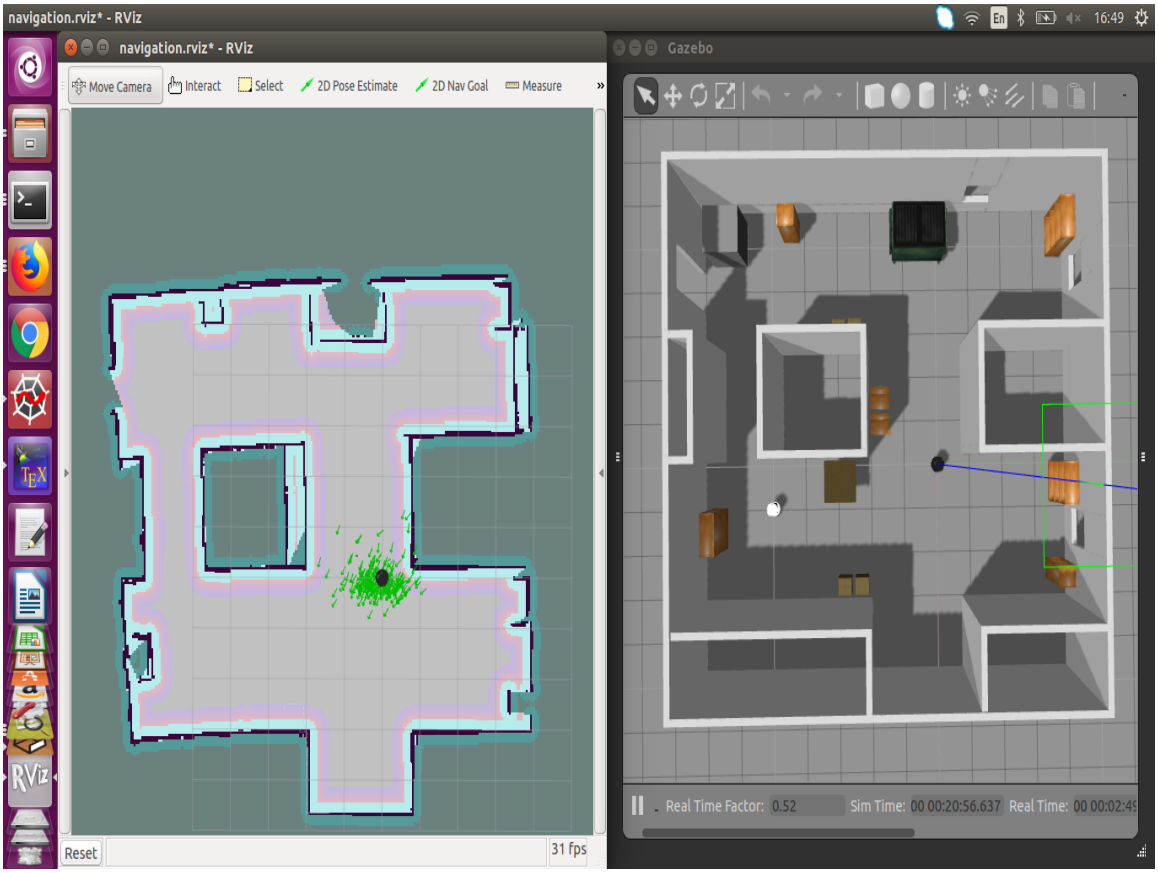

Figure 4.21: Map and World Prior to Navigation

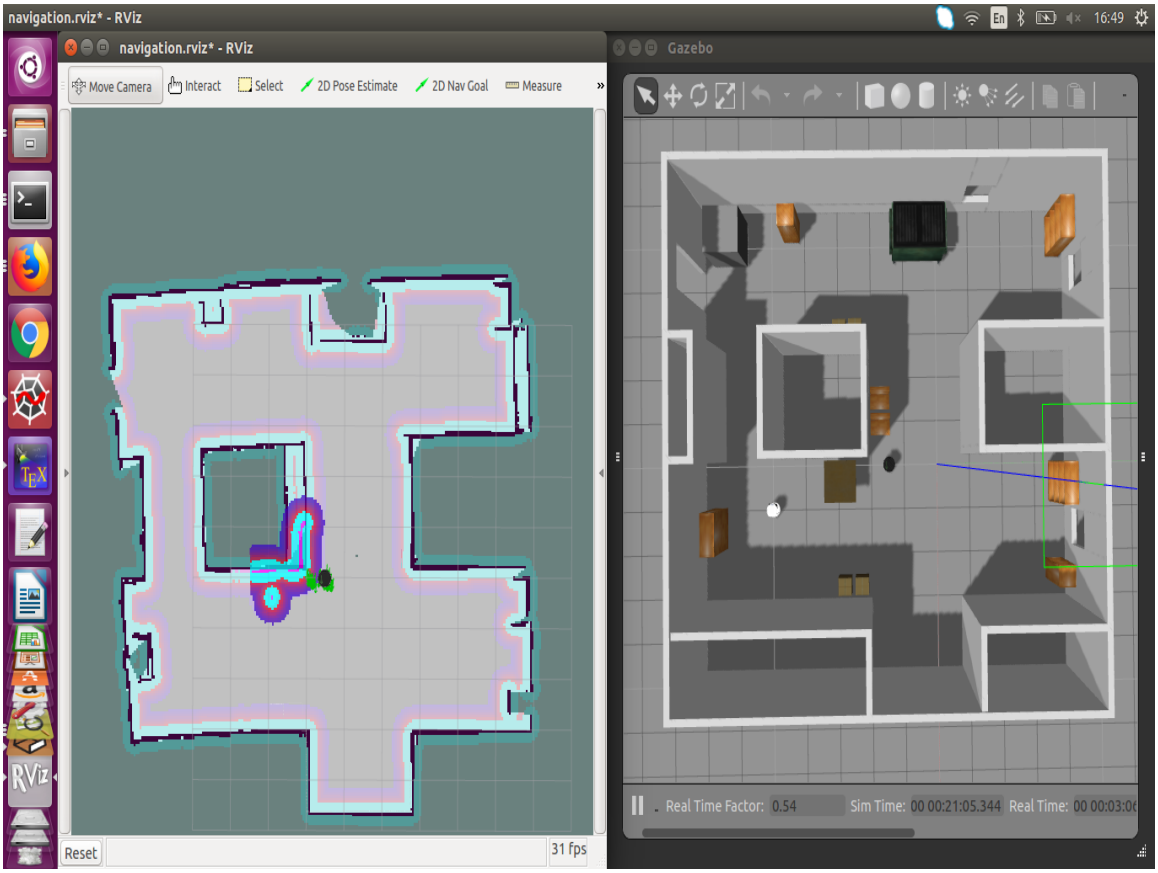

Figure 4.22: Edge of the wall detected as an obstacle by the sensor

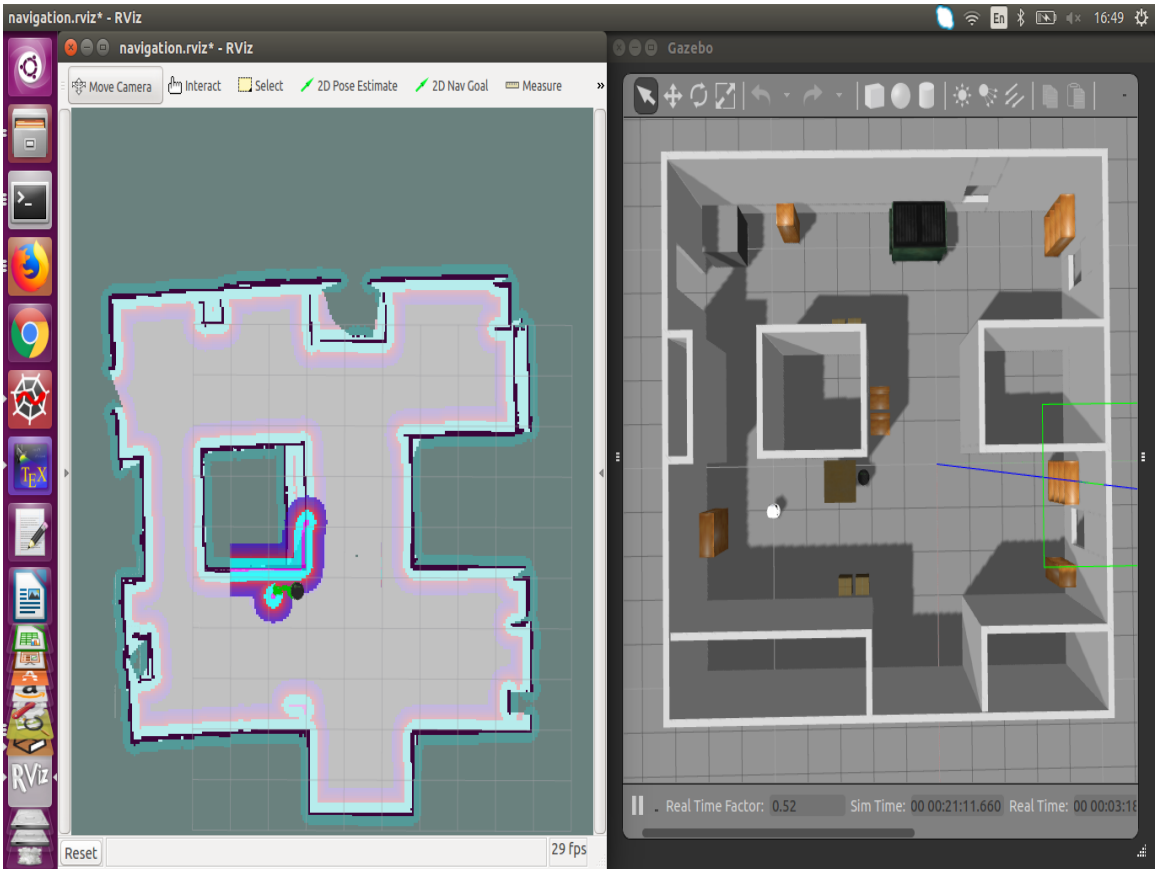

Figure 4.23: Leg of the table detected as an obstacle by the sensor

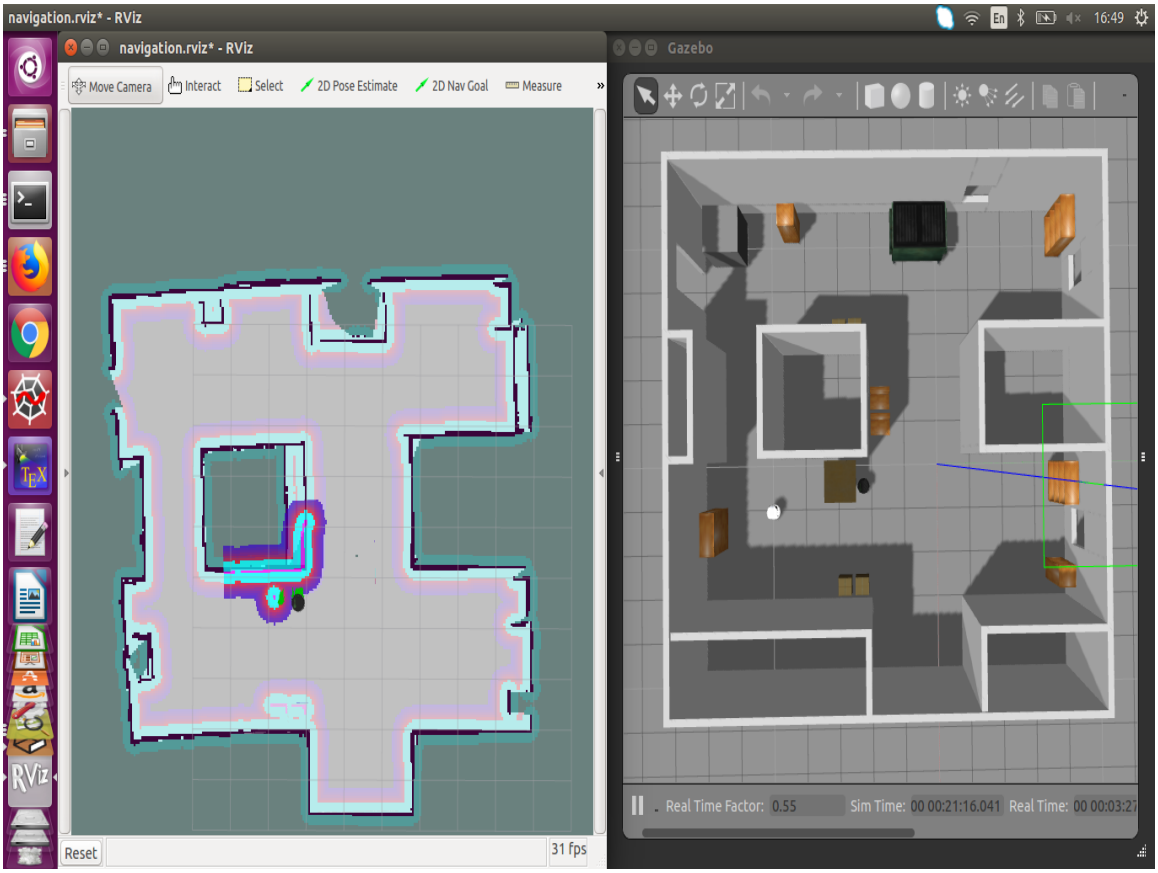

Figure 4.24: Unable to detect table top as an obstacle

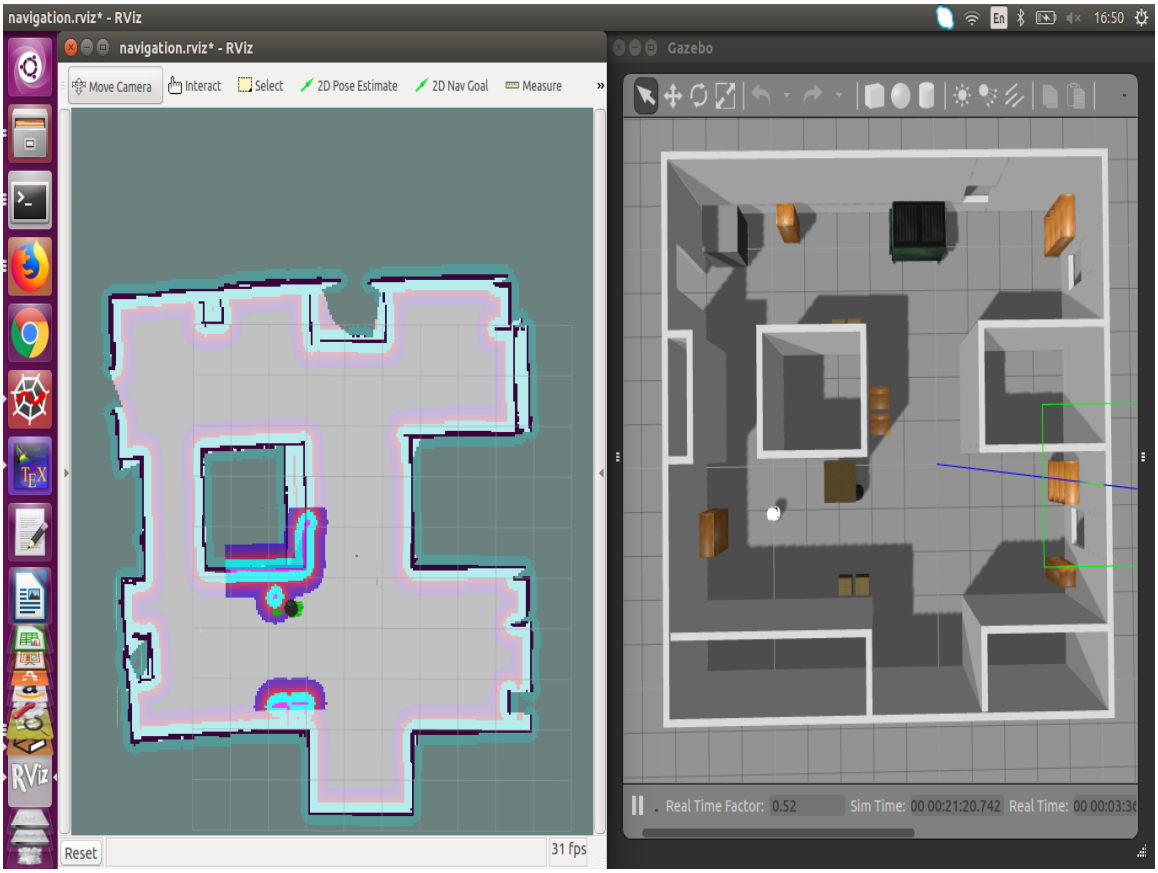

Figure 4.25: Unable to re-plan the path as the sensor does not detect the table top as an obstacle

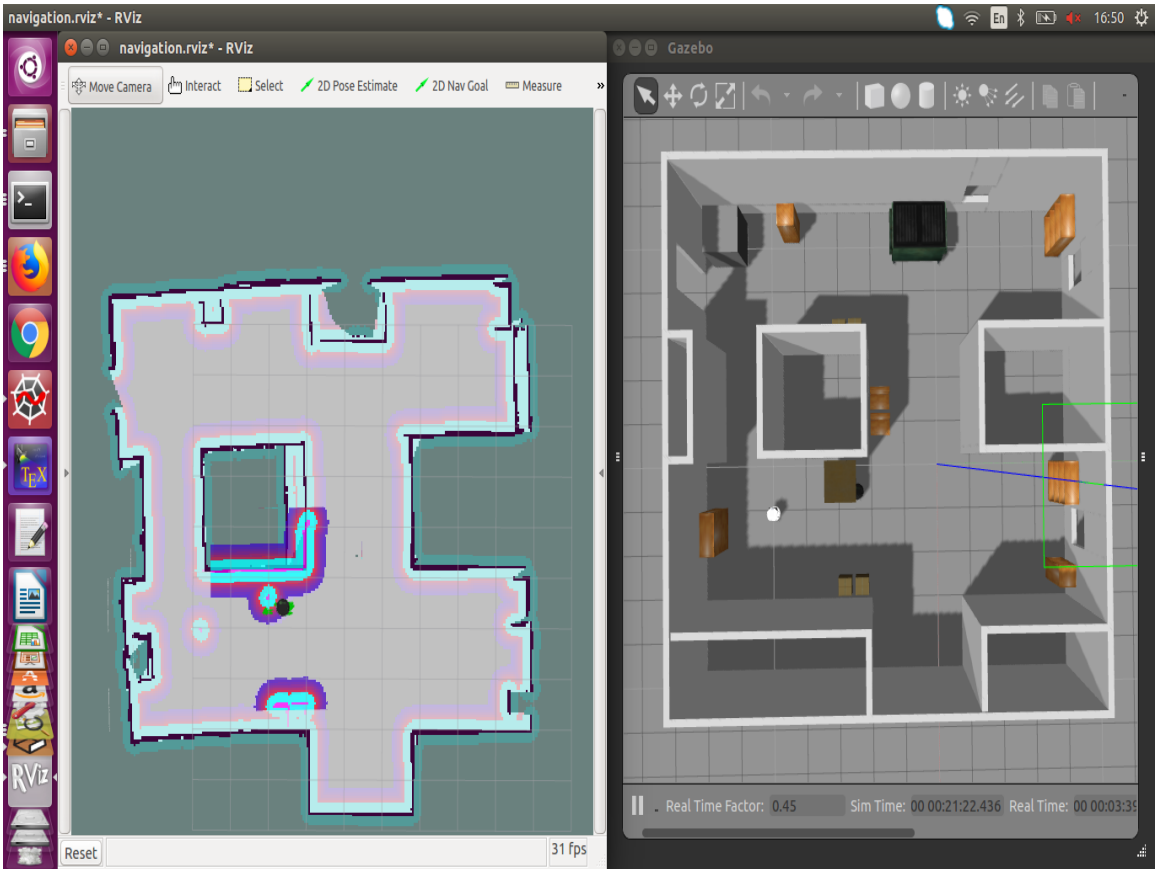

Figure 4.26: Unable to reach the goal point on account of the obstacle which cannot be perceived by the sensor

4.4 Navigation on TurtleBot using the Optimized Path Planner Algorithm

The path planning with obstacle avoidance on the TurtleBot works similar to the Gazebo simulations. The map that has been used is the UNCC ECE lab. AMCL is used for localizing the TurtleBot. This map is seen in Figure 4.27. This map is built using gmapping. The start position on the map is selected by the 2D Pose Estimate button and clicking on the map. This is shown in Figure 4.28. The starting position of the TurtleBot is shown in Figure 4.29. After giving the start point and the end point, the path is planned and then the TurtleBot starts moving according to the planned path. The TurtleBot turns 90°and starts moving forward as shown in Figure 4.30. When the obstacle is below the threshold value, it is detected as an obstacle. Then the path is updated and the TurtleBot turns to move away from the obstacle (Figure 4.31). The obstacle is detected again because it is still in the range and vision of the sensor and the path is updated again (Figure 4.32). It turns again to avoid the obstacle as shown in Figure 4.33. After this, there are no other obstacles in the path and hence the TurtleBot is able to reach the goal point (Figure 4.34).

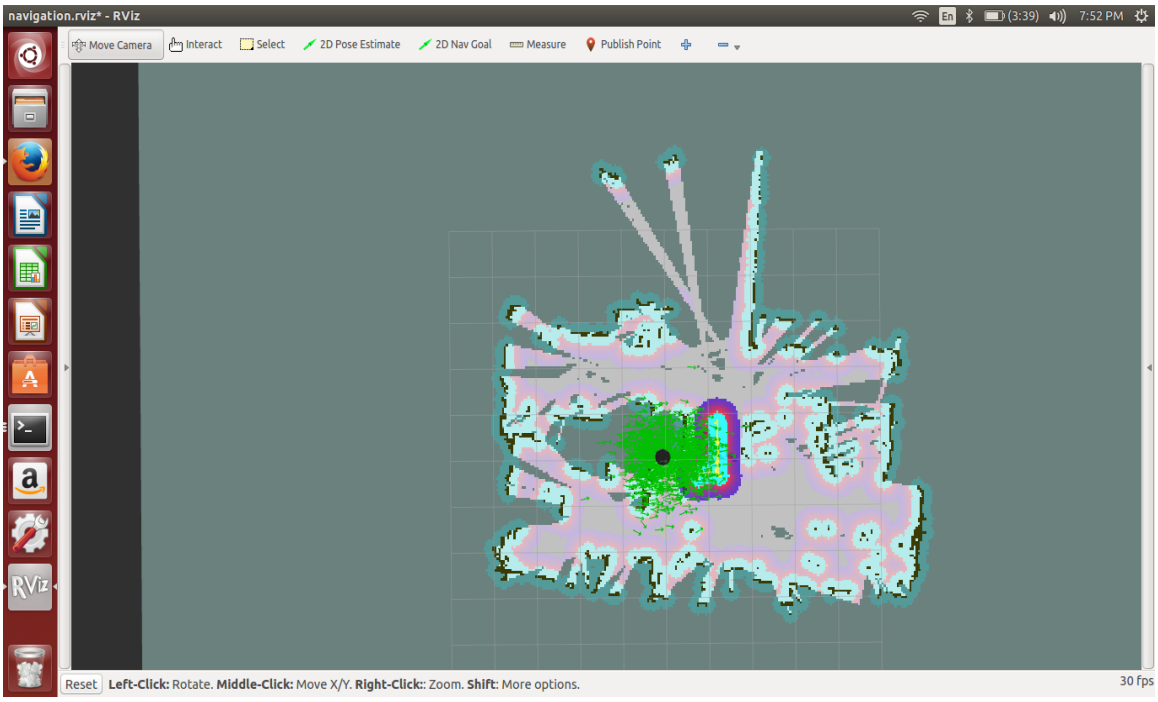

Figure 4.27: Map of the UNCC ECE lab

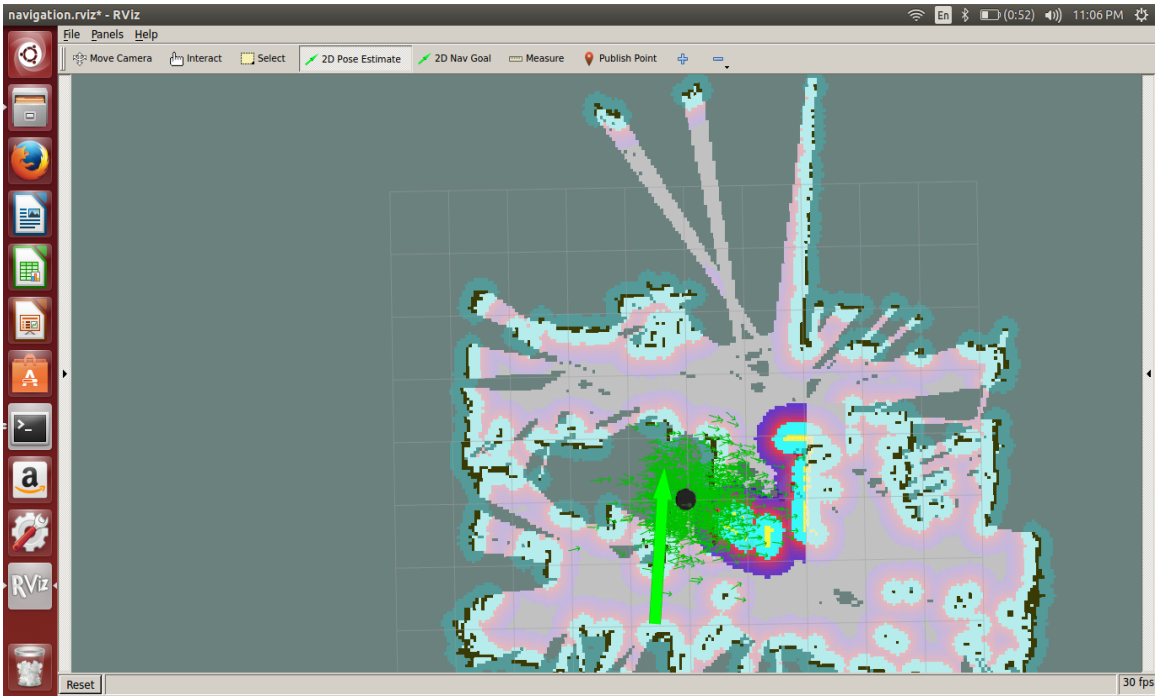

Figure 4.28: Start point selection

Figure 4.29: Start position for the TurtleBot

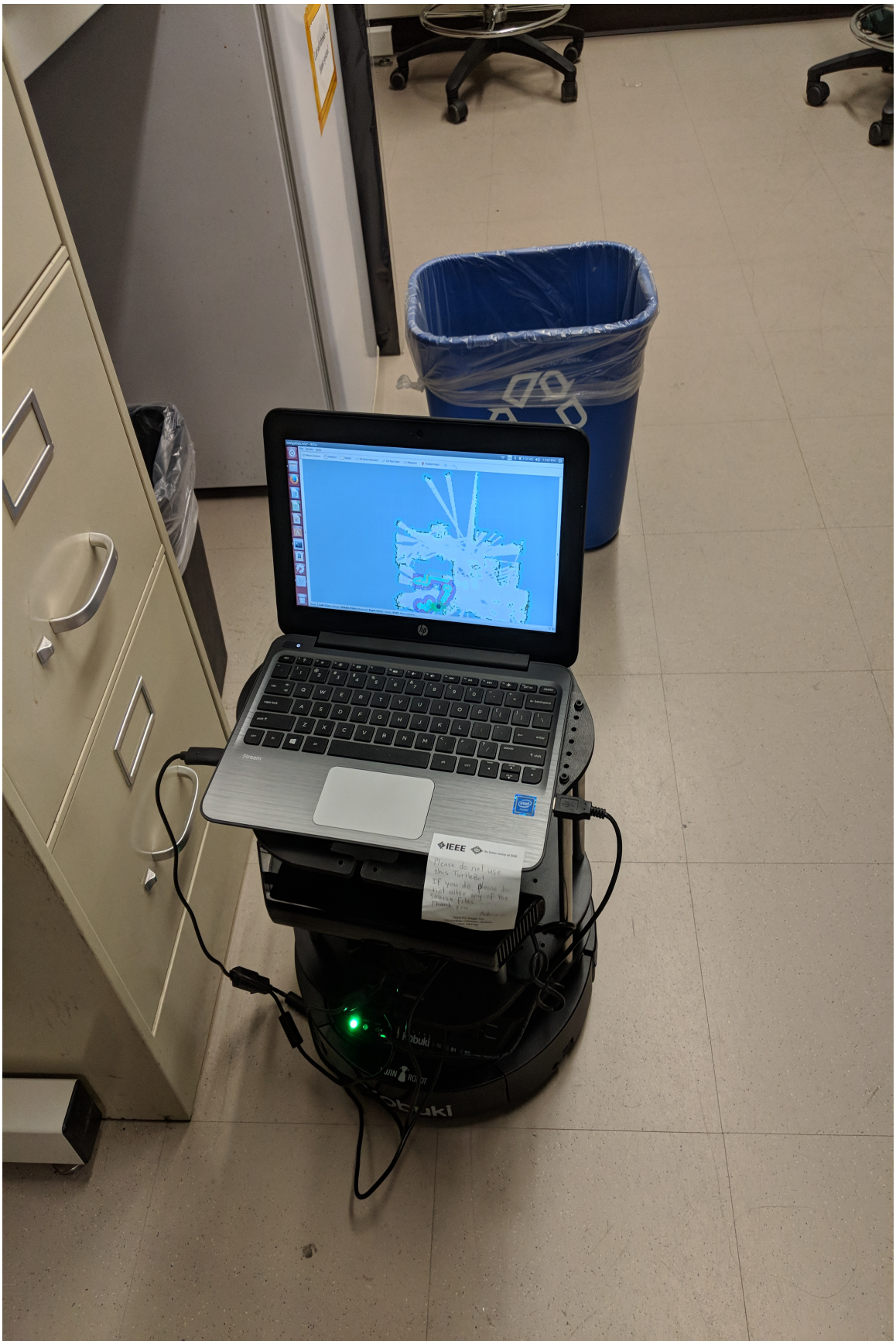

Figure 4.30: TurtleBot starts moving on the planned path

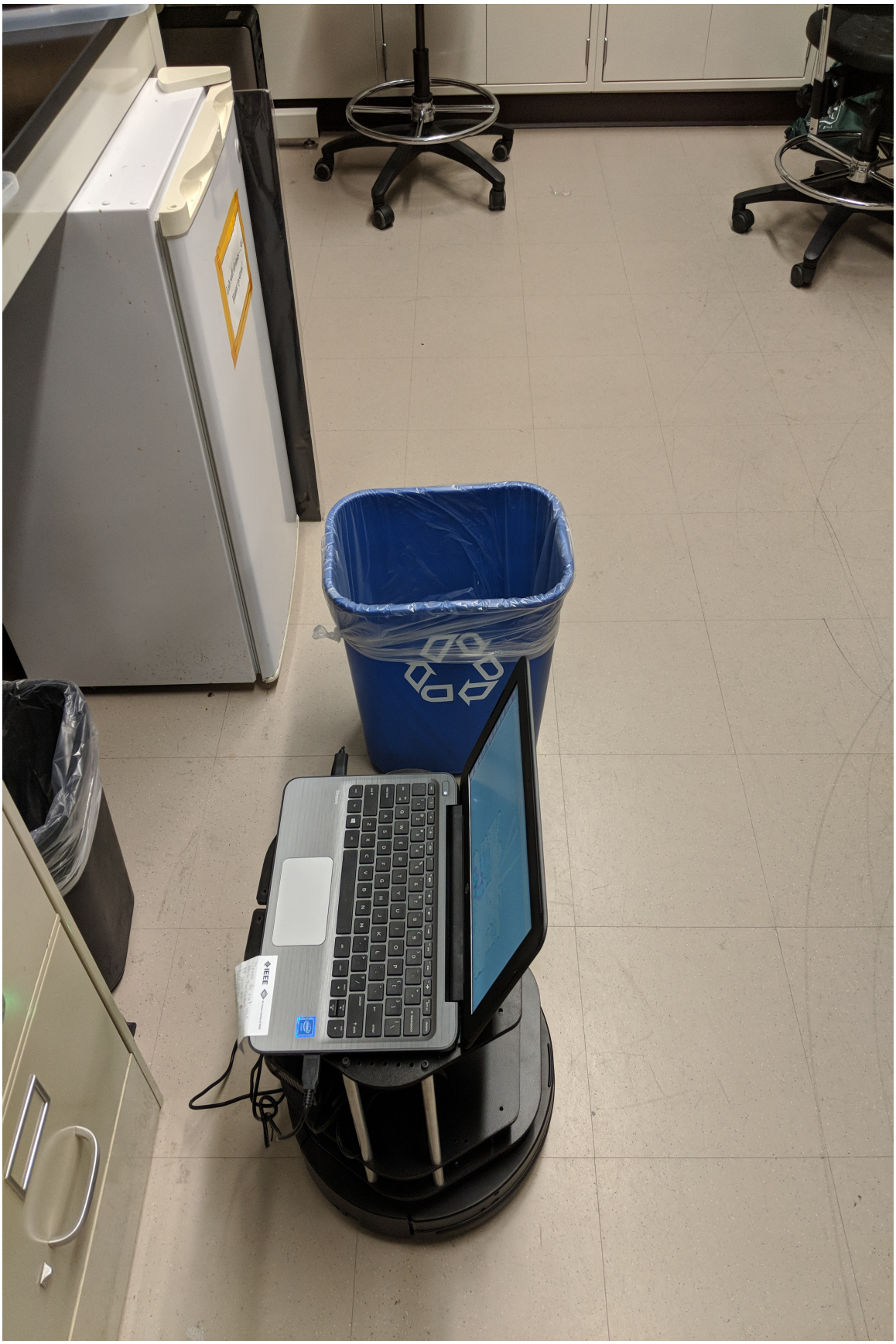

Figure 4.31: Obstacle is detected in the path of the TurtleBot and it turns to avoid the obstacle

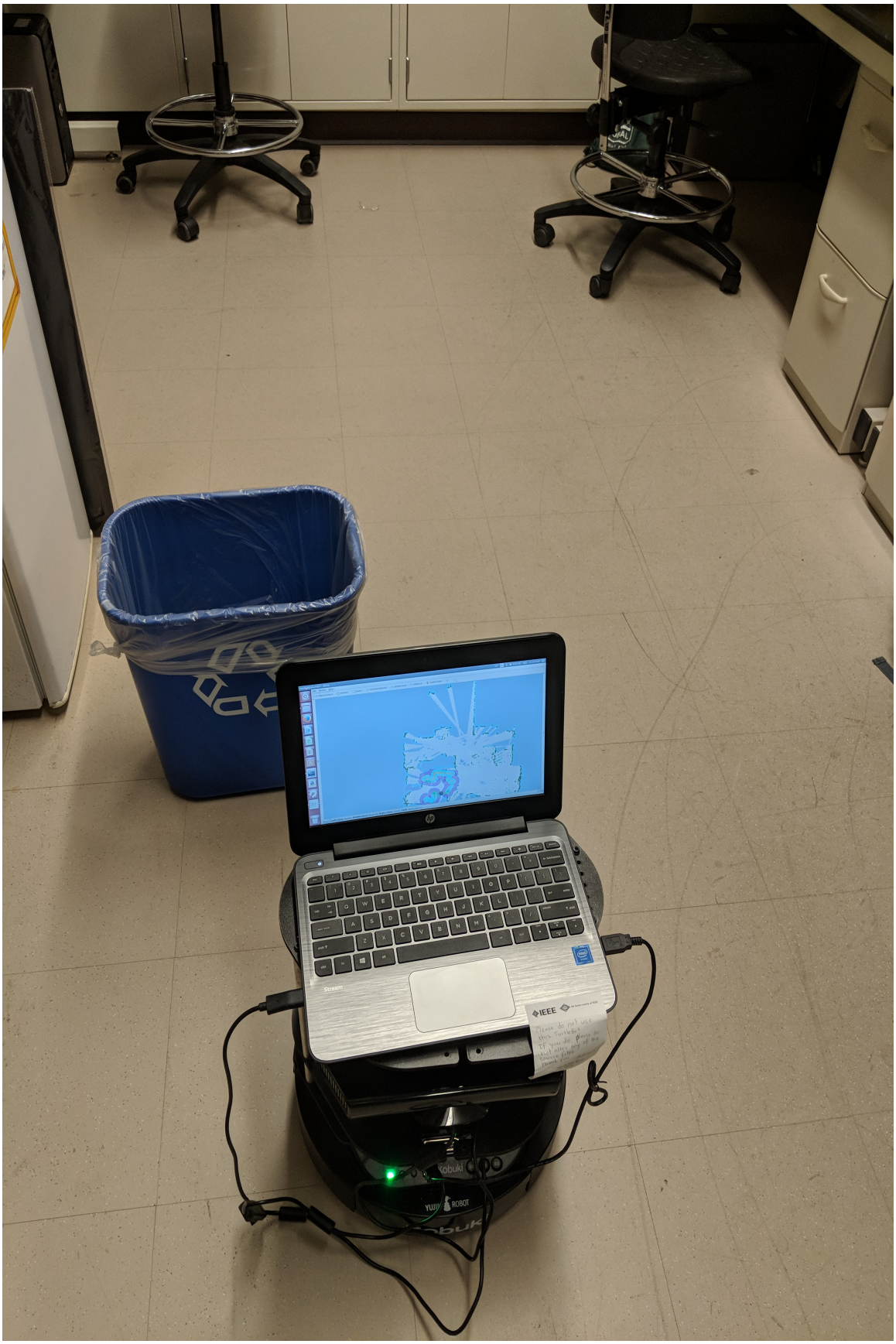

Figure 4.32: TurtleBot starts moving on the new path and detects the obstacle again

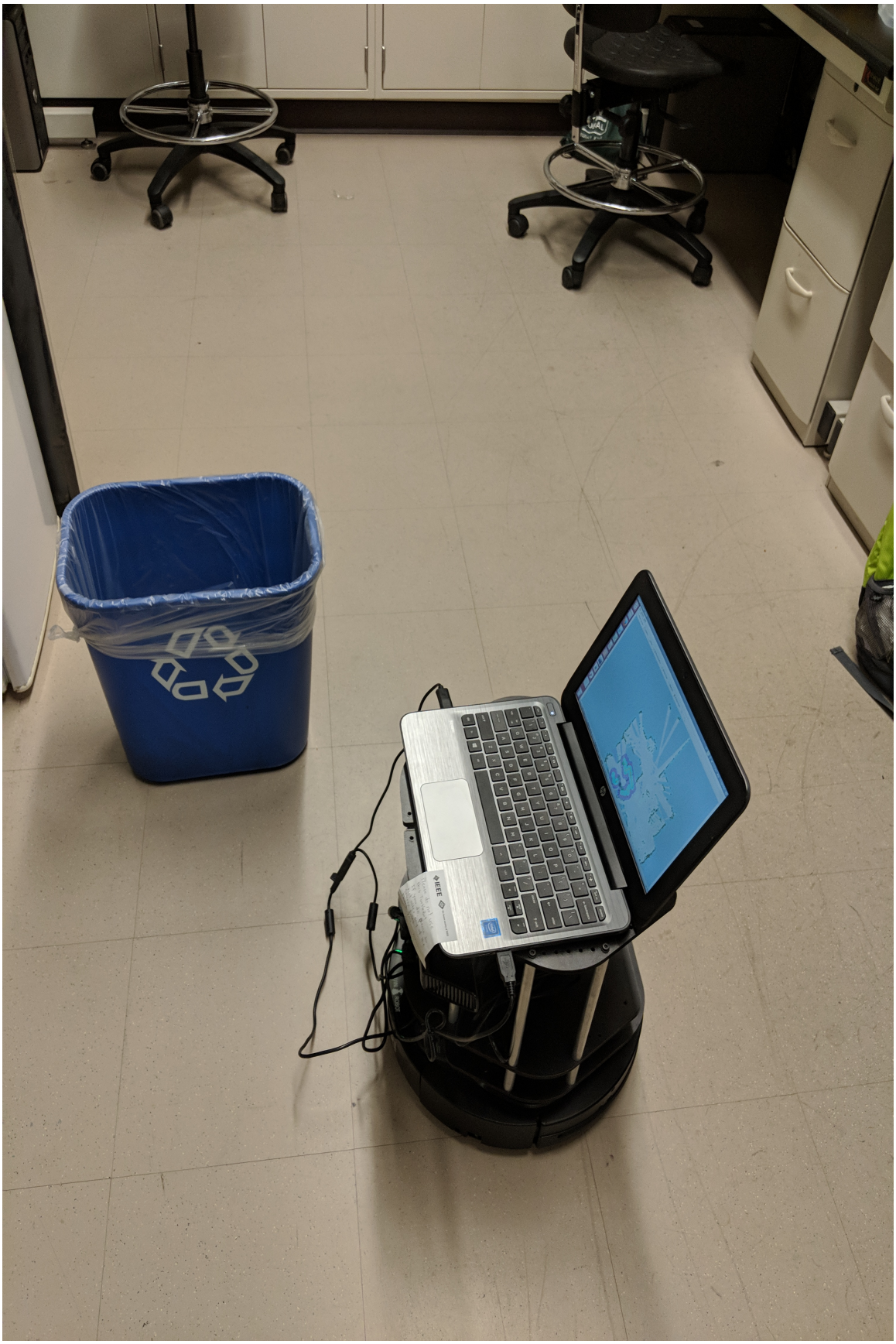

Figure 4.33: TurtleBot turns again to avoid the obstacle

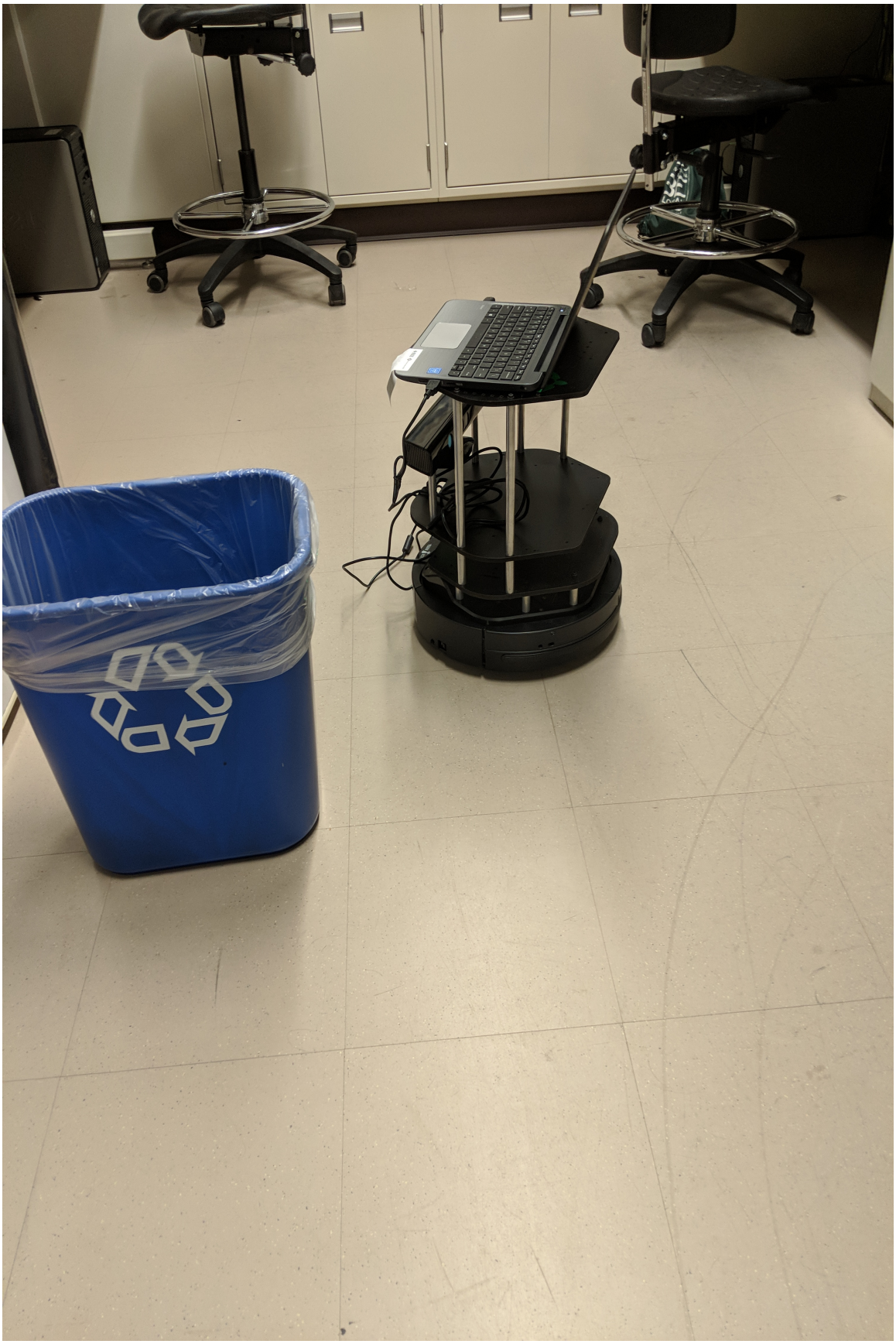

Figure 4.34: TurtleBot reaches the goal position

4.5 Comparison of Different Path Planner Algorithms

The time required to re-plan the path is compared for different methods. For comparison purposes, the Reinforcement Learning (RL) algorithm is also implemented and the time taken to re-plan the path in the same grid using optimized path planner and RL algorithm is compared. RL is a self learning algorithm and hence the time taken to find the shortest path is more as compared to the time taken by Astar algorithm or the optimized path planner algorithm. In one instance, RL takes 105.29 seconds and the optimized path planner takes 0.178 seconds to re-plan the path in the presence of obstacles. The path found from the RL algorithm is the shortest path and it is different that the path found by the optimized path planner. The path found by the RL algorithm will be similar to the path found by the Astar algorithm while optimized path planner will find a path which would take the minimum number of turns.

CHAPTER 5: CONCLUSIONS AND FUTURE SCOPE

This section summarizes the work done as part of this thesis. The review phase of this thesis involved the study of different methods for localization and mapping in dynamic indoor environments. After reviewing this work, the drawbacks of the existing methods were evaluated. Some of the methods had objects which were known prior to the system. These included the use of RFID tags and use of special markers to identify and classify the objects in the environment. This meant that once the RFID tag was read or the marker was identified, the location of the object would be updated based on the new observation, so the objects in the dynamic environment were known to the system. If any unknown objects were placed in the environment, they would not be detected by the system. This was one of the drawbacks observed from the already existing algorithms. Another drawback observed from the existing method was that the objects had movement in fixed directions or that the environment was semi dynamic. In this case the high dynamic objects would not be detected and hence the robot would not be able to reach the goal point efficiently. In one of the methods only the landmarks were moving and the localization was affected. But this method did not consider other moving objects and focused on localization of the robot in the presence of moving obstacles. Another method used distance filter and scan matching technique and focused on the map building in dynamic environments. Even though there were some limitations, all the methods that were surveyed were able to localize the robot and map the surroundings in the presence of moving obstacles.

There was need for combining these methods with a path planner which would detect the obstacles in the trajectory and would efficiently re-plan the path. In order to do so, various path planning algorithms, like Dijkstra's algorithm and A*

algorithm, were studied and implemented. Reinforcement learning was also studied and implemented on a small (14 by 14 cells) grid. RL was later implemented on for re-planning the path in the presence of obstacles. The optimized path planner was implemented and the results obtained from the implementation were as shown in Chapter 4. The implementation phase was a process of continuous improvement over the prior results. The first step of implementation only dealt with the path planning in the map. The second step was detection of obstacles. The third step was to interpret the sensor readings and use them for obstacle avoidance. Using this path planner along with SLAM would make the system very close to an autonomous system. The results obtained by implementing various algorithms were compared in Section 4.5.

Due to the wide variety of obstacles that a robot can encounter, there is scope for further research and improvement in this area. One of the improvements can be the use of multiple sensors and integration of the data from all these sensors to improve the path planning process. For example, if there is another sensor whose range and vision is from the ground plane to the height of the existing sensor, objects which are less than the height of the sensors would be easily detected. This would significantly improve the performance since one of the constraints of this implementation is that the objects must have a minimum height equal to or greater than the height of the sensor installed on the robot. As shown in Section 4.3.2, this problem can be overcome by installing a sensor which would scan the surroundings from the ground plane to a threshold height above the robot. Another area of improvement can be the integration of the velocity controller used in the ROS base local planner package. The velocity would then depend on the distance to the nearest detected obstacle. The velocity would be modulated based on the obstacle. The current algorithm only considers movement in four directions which can be improved to consider movement in any direction or angle. Since the use of robots for indoor applications is ever increasing,

there is great deal of scope for research and development in this area and future scope for improving the work carried out in this thesis.

REFERENCES

[1] A. A. Panchpor, S. Shue, and J. M. Conrad, "A survey of methods for mobile robot localization and mapping in dynamic indoor environments," in *Signal Processing And Communication Engineering Systems (SPACES), 2018 Conference on*, pp. 138–144, IEEE, 2018.

[2] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: Part I," *IEEE robotics & automation magazine*, vol. 13, no. 2, pp. 99–110, 2006.

[3] T. Bailey and H. Durrant-Whyte, "Simultaneous localization and mapping (SLAM): Part II," *IEEE Robotics & Automation Magazine*, vol. 13, no. 3, pp. 108–117, 2006.

[4] S. L. Shue, *Utilization of wireless signal strength for mobile robot localization in indoor environments*. PhD thesis, The University of North Carolina at Charlotte, 2017.

[5] D. F. Wolf and G. S. Sukhatme, "Towards mapping dynamic environments," in *In Proceedings of the International Conference on Advanced Robotics (ICAR)*, pp. 594–600, 2003.

[6] A. Elfes, "Sonar-based real-world mapping and navigation," *IEEE Journal on Robotics and Automation*, vol. 3, no. 3, pp. 249–265, 1987.

[7] M. G. Dissanayake, P. Newman, S. Clark, H. F. Durrant-Whyte, and M. Csorba, "A solution to the simultaneous localization and map building (SLAM) problem," *IEEE Transactions on robotics and automation*, vol. 17, no. 3, pp. 229–241, 2001.

[8] M. Montemerlo, S. Thrun, D. Koller, B. Wegbreit, *et al.*, "FastSLAM: A factored solution to the simultaneous localization and mapping problem," *AAAI/IAAI*, vol. 593598, 2002.

[9] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. MIT press, 2005.

[10] W. D. Smart and L. P. Kaelbling, "Effective reinforcement learning for mobile robots," in *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, vol. 4, pp. 3404–3410, IEEE, 2002.

[11] N. Arana-Daniel, R. Rosales-Ochoa, and C. López-Franco, "Reinforced-SLAM for path planing and mapping in dynamic environments," in *Electrical Engineering Computing Science and Automatic Control (CCE), 2011 8th International Conference on*, pp. 1–6, IEEE, 2011.

[12] E. Guevara-Reyes, A. Y. Alanis, N. Arana-Daniel, and C. Lopez-Franco, "Integration of an inverse optimal control system with reinforced-SLAM for path planning and mapping in dynamic environments," in *Power, Electronics and Computing (ROPEC), 2013 IEEE International Autumn Meeting on*, pp. 1–6, IEEE, 2013.

[13] D. Sun, F. Geißer, and B. Nebel, "Towards effective localization in dynamic environments," in *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*, pp. 4517–4523, IEEE, 2016.

[14] S. Dörr, P. Barsch, M. Gruhler, and F. G. Lopez, "Cooperative longterm SLAM for navigating mobile robots in industrial applications," in *Multisensor Fusion and Integration for Intelligent Systems (MFI), 2016 IEEE International Conference on*, pp. 297–303, IEEE, 2016.

[15] A. Doucet, N. De Freitas, K. Murphy, and S. Russell, "Rao-blackwellised particle filtering for dynamic bayesian networks," in *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, pp. 176–183, Morgan Kaufmann Publishers Inc., 2000.

[16] H. Zhou and S. Sakane, "Localizing objects during robot SLAM in semi-dynamic environments," in *Advanced Intelligent Mechatronics, 2008. AIM 2008. IEEE/ASME International Conference on*, pp. 595–601, IEEE, 2008.

[17] F. S. Vidal, A. d. O. P. Barcelos, and P. F. F. Rosa, "SLAM solution based on particle filter with outliers filtering in dynamic environments," in *Industrial Electronics (ISIE), 2015 IEEE 24th International Symposium on*, pp. 644–649, IEEE, 2015.

[18] S. Li and D. Lee, "RGB-D SLAM in dynamic environments using static point weighting," *IEEE Robotics and Automation Letters*, vol. 2, no. 4, pp. 2263–2270, 2017.

[19] A. Walcott-Bryant, M. Kaess, H. Johannsson, and J. J. Leonard, "Dynamic pose graph SLAM: Long-term mapping in low dynamic environments," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pp. 1871–1878, IEEE, 2012.

[20] L. Xiang, Z. Ren, M. Ni, and O. C. Jenkins, "Robust graph SLAM in dynamic environments with moving landmarks," in *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pp. 2543–2549, IEEE, 2015.

[21] P. F. Alcantarilla, J. J. Yebes, J. Almazán, and L. M. Bergasa, "On combining visual SLAM and dense scene flow to increase the robustness of localization and mapping in dynamic environments," in *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pp. 1290–1297, IEEE, 2012.

[22] N. Karlsson, E. Di Bernardo, J. Ostrowski, L. Goncalves, P. Pirjanian, and M. E. Munich, "The vSLAM algorithm for robust localization and mapping," in *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pp. 24–29, IEEE, 2005.

[23] M. Kaess, K. Ni, and F. Dellaert, "Flow separation for fast and robust stereo odometry," in *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pp. 3539–3544, IEEE, 2009.

[24] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon, "Bundle adjustment-a modern synthesis," in *International workshop on vision algorithms*, pp. 298–372, Springer, 1999.

[25] C. Harris and M. Stephens, "A combined corner and edge detector.," in *Alvey vision conference*, vol. 15, pp. 10–5244, Citeseer, 1988.

[26] T. Terashima and O. Hasegawa, "A visual-SLAM for first person vision and mobile robots," in *Machine Vision Applications (MVA), 2017 Fifteenth IAPR International Conference on*, pp. 73–76, IEEE, 2017.

[27] T. Katsunuma, O. Hasegawa, *et al.*, "Simultaneous localization and mapping by hand-held monocular camera in a crowd," *Research Report Computer Vision and Image Media (CVIM)*, vol. 2016, no. 9, pp. 1–8, 2016.

[28] G. Hua and O. Hasegawa, "A robust visual-feature-extraction method for simultaneous localization and mapping in public outdoor environment," *Journal of Advanced Computational Intelligence and Intelligent Informatics*, vol. 19, no. 1, pp. 11–22, 2015.

[29] A. Kawewong, S. Tangruamsub, and O. Hasegawa, "Position-invariant robust features for long-term recognition of dynamic outdoor scenes," *IEICE TRANSACTIONS on Information and Systems*, vol. 93, no. 9, pp. 2587–2601, 2010.

[30] A. Geiger, J. Ziegler, and C. Stiller, "Stereoscan: Dense 3D reconstruction in real-time," in *Intelligent Vehicles Symposium (IV), 2011 IEEE*, pp. 963–968, Ieee, 2011.

[31] J. Xiong, Y. Liu, X. Ye, L. Han, H. Qian, and Y. Xu, "A hybrid lidar-based indoor navigation system enhanced by ceiling visual codes for mobile robots," in *Robotics and Biomimetics (ROBIO), 2016 IEEE International Conference on*, pp. 1715–1720, IEEE, 2016.

[32] I. Maurović, M. Seder, K. Lenac, and I. Petrović, "Path planning for active SLAM based on the D* algorithm with negative edge weights," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 2017.

[33] M. Rünz and L. Agapito, "Co-fusion: Real-time segmentation, tracking and fusion of multiple objects," in *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pp. 4471–4478, IEEE, 2017.

[34] D. Holz, C. Lorken, and H. Surmann, "Continuous 3d sensing for navigation and SLAM in cluttered and dynamic environments," in *Information Fusion, 2008 11th International Conference on*, pp. 1–7, IEEE, 2008.

[35] S. Iizuka, T. Nakamura, and S. Suzuki, "Robot navigation in dynamic environment using navigation function APF with SLAM," in *Mecatronics (MECATRONICS), 2014 10th France-Japan/8th Europe-Asia Congress on*, pp. 89–92, IEEE, 2014.

[36] L.-F. Lee, *Decentralized motion planning within an artificial potential framework (APF) for cooperative payload transport by multi-robot collectives*. PhD thesis, State University of New York at Buffalo, 2005.

[37] M. Chinnaaiah, S. Ambati, M. Yaddla, J. Sravanthi, and D. Sanjay, "FPGA based robots hardware efficient scheme for real time indoor environment with behavioural control," in *Innovations in Information, Embedded and Communication Systems (ICIIECS), 2015 International Conference on*, pp. 1–5, IEEE, 2015.

[38] L. Murphy, T. Morris, U. Fabrizi, M. Warren, M. Milford, B. Upcroft, M. Bosse, and P. Corke, "Experimental comparison of odometry approaches," in *Experimental Robotics*, pp. 877–890, Springer, 2013.

[39] M. Milford and G. Wyeth, "Hybrid robot control and SLAM for persistent navigation and mapping," *Robotics and Autonomous Systems*, vol. 58, no. 9, pp. 1096–1104, 2010.

[40] T. Abiy, H. Pang, and J. Khim, "Dijkstra's shortest path algorithm." `https://brilliant.org/wiki/dijkstras-short-path-finder/`. Apr 28, 2016.

[41] dbenzhuser, "Pathfinding a star." `https://upload.wikimedia.org/wikipedia/commons/f/f4/Pathfinding\_A\_Star.svg`. Jan 4, 2017.

[42] M. Quigley, K. Conley, B. 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*, vol. 3, p. 5, Kobe, Japan, 2009.

[43] Willow Garage, "Gazebo tutorials." `http://gazebosim.org/assets/gazebo_tutorial_willow_garage_oct19_2012-5b3ad9beba88508592d271a76c034a6a.pdf`. October 19, 2012.

[44] Willow Garage, "Turtlebot," *http://turtlebot.com*, pp. 11–25, 2011.

[45] G. EDU, "Setting up 3D sensor for the TurtleBot." `http://edu.gaitech.hk/\_images/kinect\_camera.jpg`. Aug 26, 2016.

[46] A. Patel, "Introduction to A*." `https://www.redblobgames.com/pathfinding/a-star/introduction.html`. May 26, 2014.