7-23-2017

Obstacle Avoidance and Path Planning for Smart Indoor Agents

Rasika M. Kangutkar
rmk3541@rit.edu

Obstacle Avoidance and Path Planning for Smart Indoor Agents

By

Rasika M. Kangutkar

July 23, 2017

A Thesis Submitted in Partial Fulfillment of the Requirements for the
Degree of Master of Science
in Computer Engineering

Approved by:

Dr. Raymond Ptucha, Assistant Professor — Date
Thesis Advisor, Department of Computer Engineering

Dr. Amlan Ganguly, Associate Professor — Date
Committee Member, Department of Computer Engineering

Dr. Clark Hochgraf, Associate Professor — Date
Committee Member, Department of Electrical, Computer & Telecom Engr. Technology

R·I·T | Kate Gleason *College of* ENGINEERING

Department of Computer Engineering

Acknowledgments

I am very grateful towards Dr. Raymond Ptucha for guiding and encouraging me throughout this study. I acknowledge and hope the best for my teammates: Jacob Lauzon, Alexander Synesael and Nicholas Jenis. I am thankful for Dr. Ganguly and Dr. Hochgraf for being on my thesis committee. I thank Suvam Bag, Samuel Echefu and Amar Bhatt for getting me involved with Milpet. I acknowledge the support my parents and friends gave me for completing this study.

I dedicate this work to the future lab members working on Milpet, all who supported me and Milpet...

Abstract

Although joysticks on motorized wheelchairs have improved the lives of so many, patients with Parkinson's, stroke, limb injury, or vision problems need alternate solutions. Further, navigating wheelchairs through cluttered environments without colliding into objects or people can be a challenging task. Due to these reasons, many patients are reliant on a caretaker for daily tasks. To aid persons with disabilities, the Machine Intelligence Laboratory Personal Electronic Transport (Milpet), provides a solution. Milpet is an effective access wheelchair with speech recognition capabilities. Commands such as "Milpet, take me to room 237 or "Milpet, move forward can be given. As Milpet executes the patients commands, it will calculate the optimal route, avoid obstacles, and recalculate a path if necessary.

This thesis describes the development of modular obstacle avoidance and path planning algorithms for indoor agents. Due to the modularity of the system, the navigation system is expandable for different robots. The obstacle avoidance system is configurable to exhibit various behaviors. According to need, the agent can be influenced by a path or the environment, exhibit wall following or hallway centering, or just wander in free space while avoiding obstacles. This navigation system has been tested under various conditions to demonstrate the robustness of the obstacle and path planning modules. A measurement of obstacle proximity and destination proximity have been introduced for showing the practicality of the navigation system. The capabilities introduced to Milpet are a big step in giving the independence and privacy back to so many who are reliant on care givers or loved ones.

Contents

List of Figures

List of Tables

Acronyms

COALAS

COgnitive Assisted Living Ambient System

DWA

Dynamic Window Approach

GPS

Global Positioning System

IoU

Intersection over Union

mAP

mean Average Precision

MIL

Machine Intelligence Lab

Milpet

Machine Intelligence Lab Personal Electric Transport

MIT

Massachusetts Institute of Technology

MOI

Moment of Inertia

PASCAL

Pattern Analysis, Statistical Modeling and Computational Learning

PC

Personal Computer

POD

Polar Obstacle Density

ROS

Robot Operating System

RPN

Region Proposal Network

RRT

Rapidly Exploring Random Tree

sdf

Simulation Description Format

urdf

Universal Robotic Description Format

VFH

Vector Field Histogram

VOC

Visual Object Classes

xacro

XML Macros

YOLO

You-Only-Look-Once

YOLOv2

You-Only-Look-Once version2

Chapter 1

Introduction

With the boom in autonomous robots, navigation to a destination while avoiding obstacles has become important. This thesis focuses on making a modular obstacle avoidance system linked up with path planning. The obstacle avoidance system can be easily configured to suit different differential drive robots to exhibit different behaviors. The robot can be influenced by a path, environment, exhibit wall following or hallway centering, or display a wandering nature.

Part of the motivation behind this study is for making the Machine Intelligence Lab Personal Electric Transport, Milpet, autonomous. Milpet is equipped with a single channel Lidar, an omni-vision camera, sonar sensors, encoders and a front facing camera. Milpet is an effective access wheelchair that is aimed at helping patients with vision problems, Parkinson's or any patient, who cannot make use of the joystick for efficient navigation. Avoiding obstacles or maneuvering in cluttered environments can be a challenge for such patients. Milpet is speech activated and commands like "Milpet, take me to Room 237", or "Milpet, take me to the end of the hallway", can be given. These commands invoke the path planning system, which then guides the obstacle avoidance system in which direction to go. Milpet thereby helps patients in navigating their household and regaining their privacy and independence.

Since the target deployment areas are assisted living facilities such as hospitals, homes or even malls, access to 2D maps of these areas is already available. Instead of

building a separate map of the environment through Simultaneous Localization And Mapping (SLAM), we can use static floor plans and convert them to occupancy grids.

The global path of the robot comes from the global path planning module which is based off of D* Lite [14]. Here the path planning module acts at a global level providing waypoints and directions to the obstacle avoidance system. While navigating from location A to location B, the obstacle avoidance system ensures the patient's and other's safety. The You-Only-Look-Once framework version 2 [11], YOLOv2, is a real-time object detector and recognizer that can make Milpet more interactive and smarter with people. This vision information can be integrated into the system as well, and has been tested off the robot

Milpet is a research platform in the Machine Intelligence Lab and is continuously being improved. Making the navigation system modular makes it easier for testing different concepts and behaviors, and expanding it to use various sensors. The Robot Operating System (ROS) is made use of for building the code base. All code is written in Python, which is a scripting language that is easy to understand and modify.

The main challenge of this work was to make use of the onboard sensors to create an optimized navigation system. Milpet has been tested in an academic building in varying scenarios. The obstacle avoidance system has been tested with and without paths to demonstrate its effectiveness. An evaluation measurement of obstacle proximity and destination proximity have been introduced for showing the practicality of the obstacle avoidance and navigation systems.

The main contributions of this work include:

- Modular obstacle avoidance algorithm
- Path planning and guiding algorithm
- Gazebo test platform for Milpet
- Navigation system in ROS

- Interaction interface setup between user and Milpet via speech
- Evaluation metric: obstacle proximity
- Evaluation metric: destination proximity

This thesis is organized as follows: The background chapter talks about the basics of obstacle avoidance, path planning, localization techniques, object recognition frameworks and an overview on other smart wheelchairs. The system overview chapter briefly goes over Milpet's hardware and software used. The following chapters describe the interaction capabilities of the robot, the obstacle avoidance algorithm development and the path planning system. This study is then rounded up with the results, conclusions and areas of future work.

Chapter 2

Background

2.1 Autonomous Robots

For a robot to be termed as truly autonomous, the robot must be able to solve the following questions: "Where am I?","Where to go?' and "How to get there?" [15]. The foremost question, is concerned with determining where a robot is on a given map; this is termed as localization. "Where to go?" represents the destination where the robot wants to go. This can be answered by input from different interfaces; for example: "Go to the kitchen", is a speech command, where the word *kitchen* should be deduced as the destination, and mapped to a position on the map. The final question, "How to get there?", embodies the robots capability of planning a path to the destination and avoiding obstacles to get there. In the following sections, localization, path planning, obstacle avoidance, etc. are discussed; giving an overall base on what algorithms can be used, to make a robot autonomous.

2.2 Localization

Localization answers the question: "Where am I?". The term localizatoin refers to figuring out where a robot is on a given map. A robot's position, is characterized by x, y, z, ϕ in space, where $x, y, z \in$ Cartesian Space and ϕ is the orientation of the robot, which can be described as roll, pitch, yaw, r, p, y, respectively.

There are various methods for carrying out localization. Global Positioning System (GPS) is a popular outdoor localization technique. However, GPS has an accuracy of 2-8 meters [16]; thus, it is not suitable for indoor robots.

2.2.1 Dead Reckoning

Dead reckoning calculates the displacement and orientation of a robot along X, Y and Z axes from the initial position. Using geometry and a robot's configuration and dimensions, the position of the robot can be determined. This method depends heavily on its sensors, and is susceptible to noise, systematic and non systematic errors. Encoders that measure the number of revolutions of each wheel, is one kind of sensor that is used for dead reckoning. One challenge in dead reckoning is determining the initial position.

2.2.2 Environment Augmentation

Many localization techniques add sensors or markers to the environment, to facilitate in determining where the robot is. Making changes to the environment, by adding markers, sensors or tracks comes under this category.

2.2.2.1 Artificial Landmarks

These are markers or labels that are positioned at particular known locations. When the robot is able to sense one of these markers, it can calculate the distance from the marker and deduce where it is. This is because, it already knows the real size of the marker and through comparison between the real and perceived dimensions, it can determine the angle and how far it is from the marker. Figure 2.1 shows a flowchart for calculating the position in a stereo-camera system.

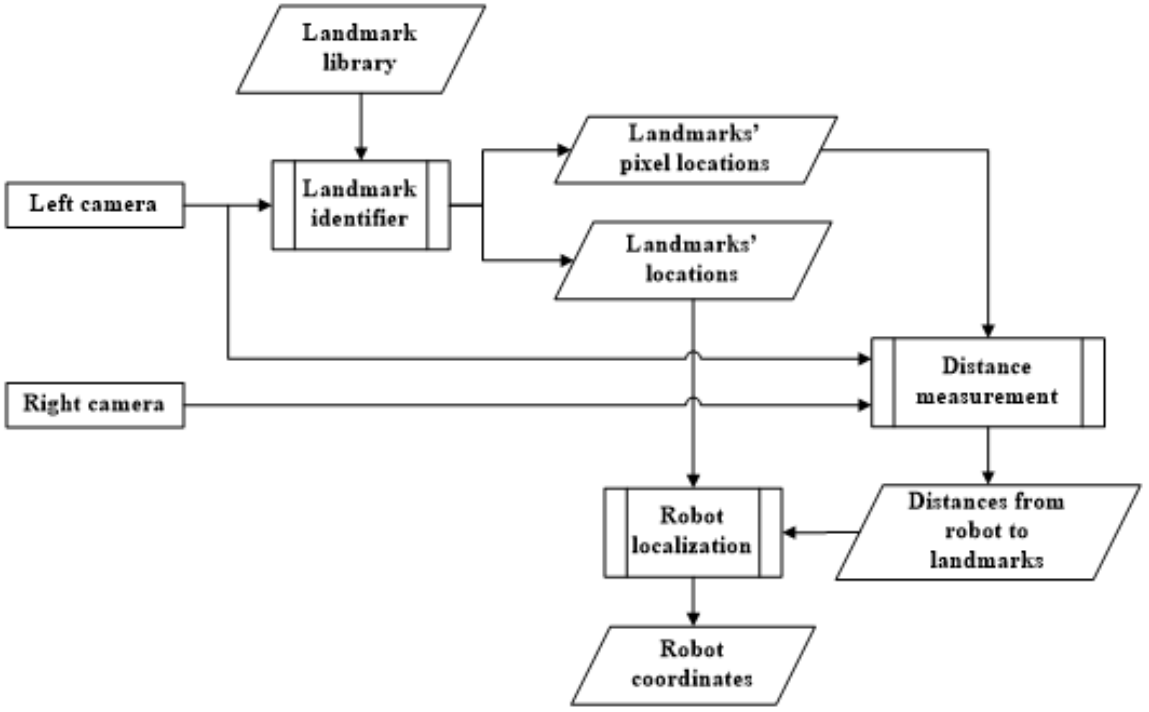


Figure 2.1: Localization using Landmarks in a Stereo Camera System [1].

2.2.2.2 Triangulation

Triangulation makes use of readings, R, from three or more sensors. An outdoor example of triangulation is GPS; it makes use of three satellites to pinpoint a position. For indoor environments, instead of using satellites, sensors like: beacons or even Wireless Fidelity (WiFi), are used. The strength of signals from three sensors is used to determine where it is on the map. Beacons emit continuous signals in a radial manner. The strength of the signal is inversely proportional to the distance from the beacon. Thus, the position of the robot can be determined from the strength of signals w.r.t. the known positions of each beacon. Figure 2.2 shows how triangulation using sensors work. R is the readings returned to the system, which helps in determining the locality.

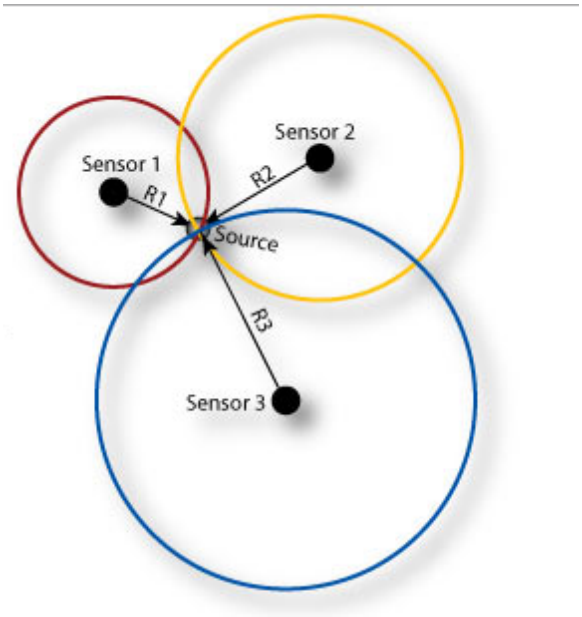


Figure 2.2: Triangulation [2].

2.2.2.3 Following and Tracking

One of the simplest kinds of localization is to have the robot follow a line; the line can be painted on, or even be magnetic. By following the line, the robot's position can always be known.

Another method for localizing a robot, is to track the robot in its environment. One way of doing this is to install an overhead camera in an environment. The camera can then determine the robot's position w.r.t. the environment. This technique is made use of in Automated Guided Vehicles (AGV) and in swarm robotics. Figure 2.3 (left) shows an image captured by an overhead camera; Figure 2.3 (right) shows the detected positions and orientations of the robots, the white line depicts the heading of the robot.

2.2.3 Innate On-board Localization

All the above methods, albeit for dead reckoning, rely on changing the environment for localizing the robot. Scenarios like: a missing marker, or a beacon's battery

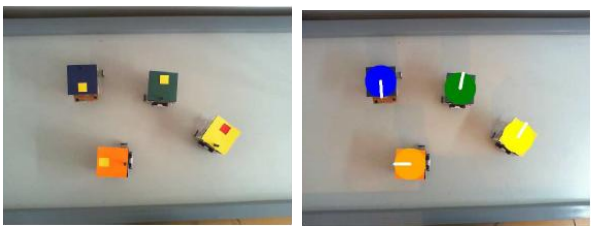


Figure 2.3: Robot Tracking with an Overhead Camera [3].

running out, affect the entire localization system. So, if any of these methods are made use of, ways to check and maintain these augmentations must be carried out.

Instead of relying on these external paraphernalia, the robot can be trained with machine learning techniques to help in localization; this would require no need of adding accessories to the environment.

2.2.3.1 Machine Learning

Localization using machine learning depends on having data with unique signatures/features at different positions. Data from the robot's sensors is collected at different positions, with the x, y positions as ground truths and fed to a classifier. The classifier is then trained and tested on a held out data-set that the robot has never seen. The accuracy of the system can then be calculated.

The classifier used can be trained using different techniques; Support Vector Machines, Decision Trees, Random forest, Convolutional Neural Networks are a few types of classifiers. Some machine learning techniques have a prerequisite step to training: feature extraction, where techniques like Histogram of Gradients (HOG) [17], Scale Invariant Features Technique (SIFT) [18] and data dimension reduction techniques are applied.

2.3 Path Planning

Searching for a goal location is an integral part of robotics. The initial position of the robot is termed as the start / source, while the end position is termed as the destination / goal. The terrain the robot is to navigate is generally represented as a grid.

Planning a path from the start to the destination is dependent on the map representation. The map can be represented in a topological form or metric form. In the metric form, objects, walls, etc. are placed in a two-dimensional space, where each position on the map represents a real area to scale. This is the most common type of map. Topological maps represent specific locations as nodes and the distances between them as edges connecting the nodes, making them graphs.

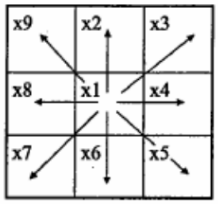


Figure 2.4: Possible moves between grid cells [4].

Figure 2.4 shows the relation between cells in the map. Each cell has at most eight neighbors. Since the terrain is known, informed search algorithms like Djikstra, A*, D* are used. These algorithms make use of an evaluation criteria, to find the optimal path. In this section various path planning techniques are described.

2.3.1 Djikstra

In 1956, the Djikstra path planning technique was formulated by Edsger W. Djikstra [19]. In the Djikstra algorithm, a graph G is considered with n nodes via edges e. Edge e_{n_1,n_2} has the cost from node $n1$ to node $n2$. Here, two lists are maintained:

visited list V and unvisited list U. s is the start node, g is the goal node. U is a min-priority queue.

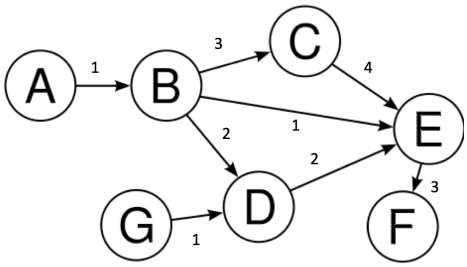


Figure 2.5: Graph G with costs on edges.

First, s is placed in U with $e_{s,s} = 0$; no cost is incurred. The other nodes are added to U with $e_{s,n_{1\ldots k}} = \infty$, where k is the number of nodes in the graph. From U, the node which has the minimum cost is chosen. In this case, the start node is picked, and costs from the start s to its neighbors x are updated. The rule for updating the cost for x is shown in (2.1), where $e_{current}$ refers to x cost in U and $x-1$ is the current node being considered. After updating the costs to each neighbor, the selected node (start node) is removed from U and placed in V. The same steps are repeated until g is added to V or if the minimum cost in U is ∞, i.e. no path to g found.

$$e_{s,x_{updated}} = min(e_{s,x_{current}}, e_{s,x-1} + e_{x-1,x}) \tag{2.1}$$

2.3.2 A*

The A* search algorithm is an extension of Djikstras algorithm and was proposed in 1968. It makes use of an evaluation function with a heuristic , (2.2), to get better performance. A* is a very popular algorithm in robotics and path planning.

$$f(n) = g(n) + h(n) \tag{2.2}$$

$f(n)$ is the evaluation function, where $h(n)$ is the estimated cost / heuristic cost

to the goal node g. $g(n)$ and $g_{s,n}$, both represent the cost from the start node s to n. Figure 2.6 shows how the estimate is determined. Since A* never overestimates the cost to the goal g, the shortest path can always be found. Referring to the grid cell structure in Figure 2.4, diagonal edges have a cost c of 1.4, while horizontal and vertical edges have a cost c of 1. For example, $c_{x_1,x_9} = 1.4$ and $c_{x_1,x_4} = 1$. If x_2 and x_7 are obstacles, then $c_{x_1,x_2} = \infty$ and $c_{x_1,x_7} = \infty$. The notation $b(x_1) = x_2$ shows that cell x_2 is the back-pointer b of x_1.

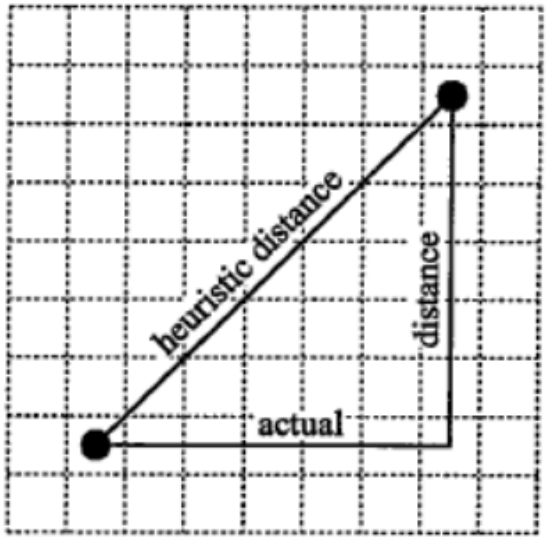


Figure 2.6: A* heuristic $h(n)$ [4].

Two lists are maintained here as well: Open list queue O and Closed C list. Algorithm 1 describes the A* algorithm. Line 1 shows that initially, s is added to O. Lines [2-4] says that if O is empty, then the algorithm is stopped, else node with the least $f(n)$, $f(n_{best})$ is popped from the queue and n_{best} is added to C. Lines [8-12] show that if n_{best} is equal to g then the path is found, else the neighbors $x \notin C$ are added to O. If x is already present in C and $g_{s,n_{best}} + c_{n_{best}} < g_x$ then update back-pointer $b(x)$ to n_{best}.

Algorithm 1: A* Psuedocode.

```
1  O ← s;
2  while n_best ≠ g OR O not empty do
3      f(n_best)= min(f(n_i)),i ∈ O ;
4      C ← n_best;
5      if n_best == g then
6          exit();
7      else
8          for each x neighbor of n_best do
9              if x ∉ C then
10                 if x ∈ O then
11                     g_{s,x}=min(g_{s,x},g_{s,n_best}+g_{n_best,x}) ;
12                 O ← x;
```

2.3.3 D*

A* and Djikstra algorithms assume that the entire environment is known between planning, however, this is seldom the case. Paths can get blocked off, new and/or moving obstacles may be present. To encompass this, D* was proposed in [20]. The term D* comes from Dynamic A*, as it aims at efficient path planning for unknown and dynamic environments. If a robot is following a path provided by A*, and a change in the surroundings is encountered by the robot, A* can be called again; with the robots current location as the start point s. However, this recalculation is expensive and doesnt make use of the former calculations made, while finding the path. Further, by the time the robot recalculates this new path, the robot may have gone along a wrong path. For this reason, a faster and more efficient method for recalculating paths arises, and D* aims at addressing it.

Unlike A*, D* calculates the path starting from the goal g to the start s. In D*, each grid-cell has $h(x)$, $p(x)$ and $b(x)$, which represent the estimated cost from goal g to x, the previous calculated $h(x)$ and a back-pointer. The back-pointer represents the cell reached before/ predecessor of x. Each grid cell can be in either $OPEN$,

$CLOSED$, NEW states t, depending on whether it is, hasnt been or was added to the open list O, respectively.

$$h(Y) = h(X) + c(X, Y) \tag{2.3}$$

$$k(X) = min(h(X), p(X)) \tag{2.4}$$

For implementing D*, initially all cells have state t(deg) set to NEW, the goal is g is placed on the open list O and the cost $h(g)$ is set to zero. $PROCESS\_STATE$ and $MODIFY\_COST$ are the two main functions here. O is a min-priority queue, where the cell with the smallest key k_{min} is removed first. k for each cell is calculated using (2.4). $c(x, y)$ is the cost of moving from x to y cells. k_{old} is the k of the most recent removal of top element x, from O. Figure 2.7 and Algorithm 2 describe the flow chart of D* and $PROCESS\_STATE$ pseudo-code, respectively. The following are the conditions checked for each neighbor Y of current cell X:

1. if neighbor Y, with $t(Y) = CLOSED \bigwedge h(Y) < k_{old} \bigwedge h(X) > h(Y) + c(Y, X)$
2. if neighbor Y has $t(Y) = NEW$
3. if neighbor Y has $b(Y) = X \bigwedge h(Y) < p(Y)$
4. if neighbor Y has $b(Y) \neq X \bigwedge h(Y) > h(X) + c(X, Y)$
5. if $p(X) >= h(X)$
6. if neighbor Y has $b(Y) \neq X \bigwedge h(X) > h(Y) + c(Y, X) \bigwedge h(Y) > k_{old} \bigwedge h(Y) = p(Y)$

If Condition 1 is true, $h(x)$ is set to $h(Y) + c(Y, X)$. For Condition 2, the new cell Y is added to O by determining its $h(Y)$, $p(Y) and b(Y) = X$. h(Y) is calculated

using (2.3). For Condition 3, if X is either $OPEN$ or $CLOSED$, $p(Y)$, $h(Y)$ are updated and Y is added to O. When Condition 4 is satisfied, Condition 5 is checked. If it is true, the back-pointer $b(Y) = X$ is set, and $p(X)$ is updated accordingly. Y is then added to the list O. Y is inserted into O, if Condition 6 is satisfied.

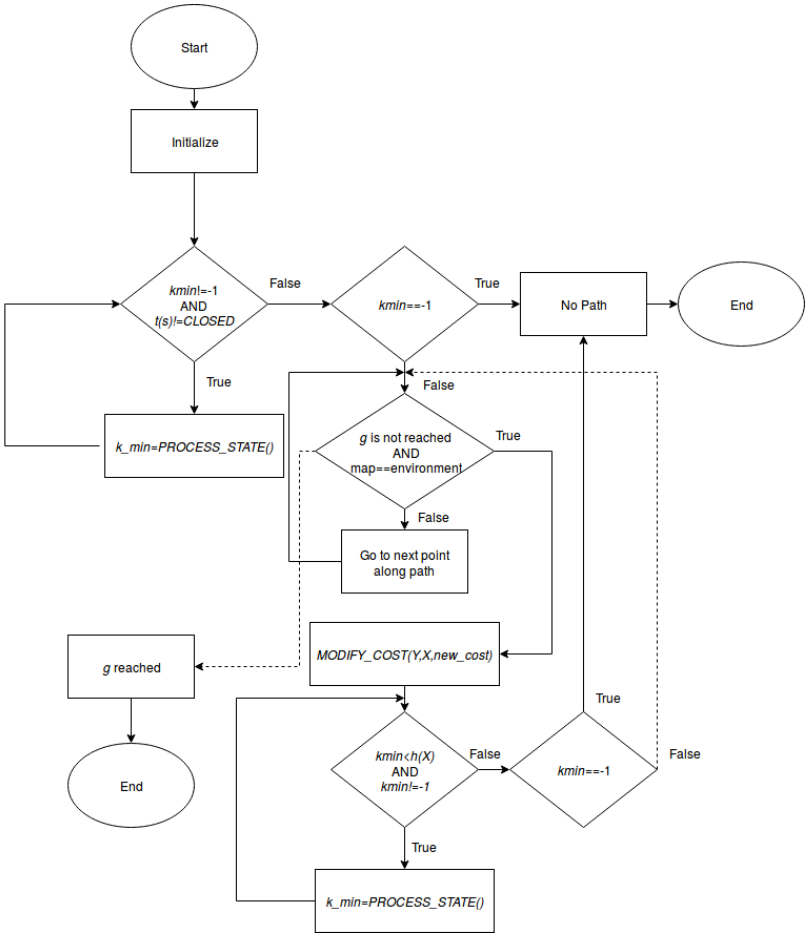


Figure 2.7: D* Algorithm.

D* is thus, capable of repairing and replanning, to get to the goal g.

2.3.4 Focused D*

The number of steps in D* are many. Focused D* [21] reduces the number of steps by adding a heuristic $\phi(X, s)$ to the cost function. This 'focuses' the D* in the direction

Algorithm 2: D* $PROCESS\_STATE$ Psuedocode.

```
1  X = dequeue(O);
2  if X = NULL then
3      return -1;
4  k_old = get_k_min();
5  for each neighbor Y of X do
6      if Condition1 then
7          b(X) = Y;
8          h(X) = h(Y) + c(Y, X);
9  for each neighbor Y of X do
10     if Condition2 then
11         enqueue(Y);
12         h(Y) = h(X) + c(X, Y);
13         b(Y) = X;
14     else
15         if Condition 3 then
16             if t(X) = OPEN then
17                 if h(Y) < p(Y) then
18                     p(Y) = h(Y);
19                     h(Y) = h(X) + c(X, Y);
20                 h(Y) = p(Y) = h(X) + c(X, Y);
21                 enqueue(Y);
22             else
23         else
24             if Condition4 then
25                 if Condition5 then
26                     b(Y) = h(X);
27                     h(Y) = h(X) + c(X, Y);
28                     if t(Y) = CLOSED then
29                         p(Y) = h(Y);
30                     enqueue(Y);
31                 else
32                     p(X) = h(X);
33                     enqueue(X);
34             else
35                 if Condition6 then
36                     p(Y) = h(Y);
37                     enqueue(Y);
```

of the goal. Equation 2.5 describes the updated cost function for $h(Y)$.

$$h(Y) = h(X) + c(X, Y) + \phi(X, s) \tag{2.5}$$

2.3.5 D*Lite

D* Lite [14] has the same output as D*, but is done is lesser steps. D* lite has no back-pointers. Each cell on the grid, x has a $g(u)$ and $rhs(u)$ value. If $g(u) = rhs(u)$ then, u is said to be consistent, else it is inconsistent. Inconsistent cells are added to a priority queue U. A cell is termed as over-consistent if $g(u) > rhs(u)$ and under-consistent if $g(u) < rhs(u)$. The priority, k, is calculated using (2.6). When adding u to U, the function $UPDATE\_VERTEX()$ is called. Here, if $u \in U$, it is removed from U and if the neighbor is inconsistent, it is added to the queue along with its key k.

$$k = min(g(u), rhs(u) + h(u)) \tag{2.6}$$

$$rhs(u) = min_{n \in Neighbors}(g(n) + c(u, n)) \tag{2.7}$$

To start the algorithm, all $g(\text{deg})$ and $rhs(\text{deg})$ are initialized to ∞. Only $rhs(goal)$ is set to zero and added to U. After this $COMPUTE\_SHORTEST\_PATH()$ is called. In the function $COMPUTE\_SHORTEST\_PATH()$, while k_{top} of U is less than the k_{start} or $rhs(start) != g(start)$ the function runs. The top most element, u is dequeued. If u is over-consistent, $g(u) = rhs(u)$ and the $UPDATE\_VERTEX()$ is called for all neighbors. If u is consistent, then $g(u)$ is set to ∞ and $UPDATE\_VERTEX()$ is called on u, itself, as well as its neighbors.

Once the path is calculated, the robot moves along the path. If any discrepancy occurs, and it is not possible to travel to cell v, then the edge cost $c(u, v)$ is set to ∞

and $UPDATE\_VERTEX()$ is called on u. Also, for every member in U, the cost is updated with (2.7). Following this, $COMPUTE\_SHORTEST\_PATH()$ is called again.

2.3.6 Rapidly-exploring Random Trees

A Rapidly-exploring Random Tree (RRT) is a tree structure constructed from randomly sampled waypoints for path planning. With states X, a tree is made, from the start position x_{root} to x_{goal}, where $X \in X_{free}$. States can be either $free$ or have an *obstacle*. The algorithm continues until x_goal is reached, or until the maximum number of iterations,max, are completed.

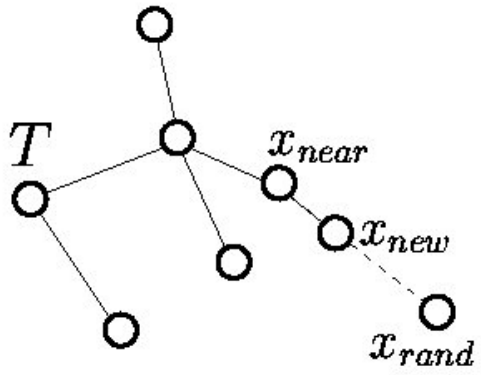


Figure 2.8: Sample RRT tree T [5].

Algorithm 3 describes the steps of building an RRT, T. First, the root node, x_{root}, is added to T, and a point $x_{rand}inX$ is randomly sampled. The closest point x_{new} to x_{root} in the direction of p is considered, with maximum distance $step_size$ away from x_{root}. If $x_{new} \in X_{free}$, then x_new is added to the tree T; if not, x_{new} is discarded and another point is sampled.

Algorithm 3: RRT Psuedocode.

```
1 T ← x_start;
2 while x_goal ∉ T OR i ≠ max do
3     Sample point x_rand;
4     x_near ← closest_point(T, x_rand);
5     x_new = get_new_point(x_near, stepsize);
6     if x_new ∈ X_free then
7         T ← x_new;
8     else
9         discard x_new;
```

2.4 Obstacle Avoidance

Avoiding obstacles while navigating is an important component of autonomous systems. Robots must be able to navigate their environment safely. Obstacle avoidance is more challenging than path planning. Path planning requires one to go in the direction closest to the goal, and generally the map of the area is already known. On the other hand, obstacle avoidance involves choosing the best direction amongst multiple non-obstructed directions, in real time. The following sections briefly discuss common obstacle avoidance algorithms.

2.4.1 Vector Field Histogram Algorithm

The vector field histogram method [6] is a popular local obstacle avoidance technique for mobile robots. The main concepts included in the VFH algorithm are the certainty/histogram grid, polar histogram and candidate valley.

The area surrounding the robot, is represented as a grid. When an obstacle is detected by the sensor at a Cartesian position (x,y), the values in the grid are incremented. Here the grid acts like a histogram and is called a Cartesian histogram. Thus, for sensors prone to noise, this histogram grid aids in determining exact readings from them. Figure 2.9 shows a robot with its certainty grid. As seen from the figure,

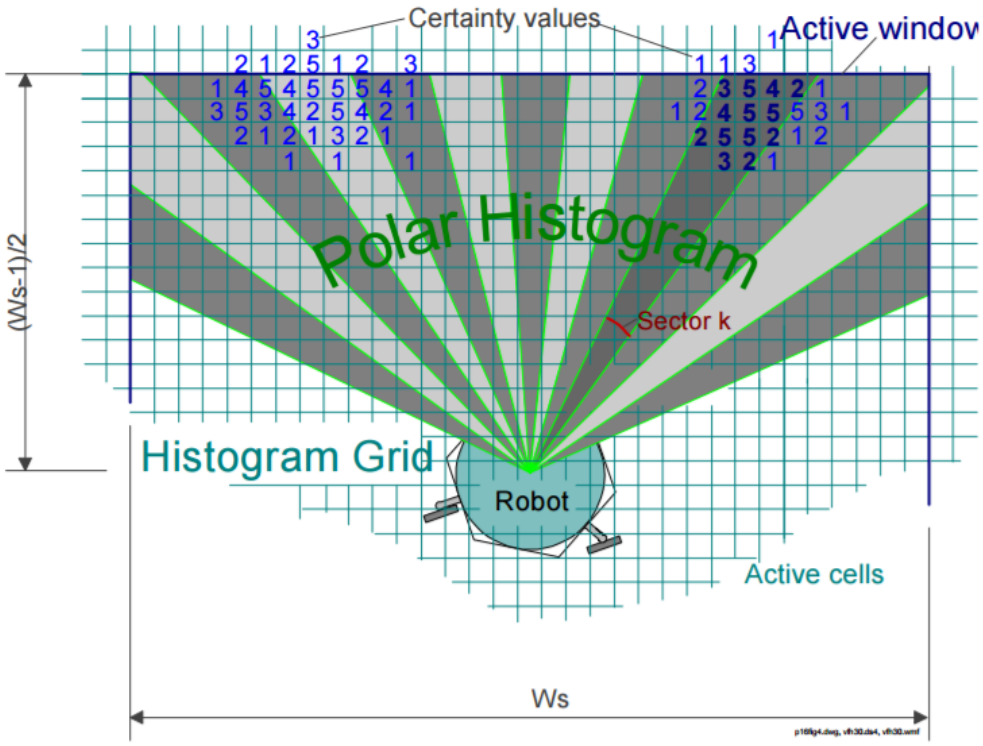


Figure 2.9: Certainty grid of VFH [6].

cells with numerals, show where an obstacle is present. The higher the number, the more probable it is that an obstacle is there.

Once this grid is evaluated, it is collapsed into a polar histogram, which is one dimensional vector representation centered around the robot. It represents the polar obstacle density (POD). Figure 2.10 shows the top view of an environment, with a pole and walls. Figure 2.11 shows the certainty histogram with the polar histogram overlaid on it and Figure 2.12 c. shows the same polar histogram with degrees along the X axis and POD along the Y axis.

Depending on the density level of obstacles, free/safe areas, termed as candidate valleys, are determined and then the robot selects the optimal one. The candidate valleys are determined as areas with POD lesser than the threshold. Figure 2.13 depicts the candidate valleys.

Finally, the algorithm filters out candidate valleys where the robot wont fit. After

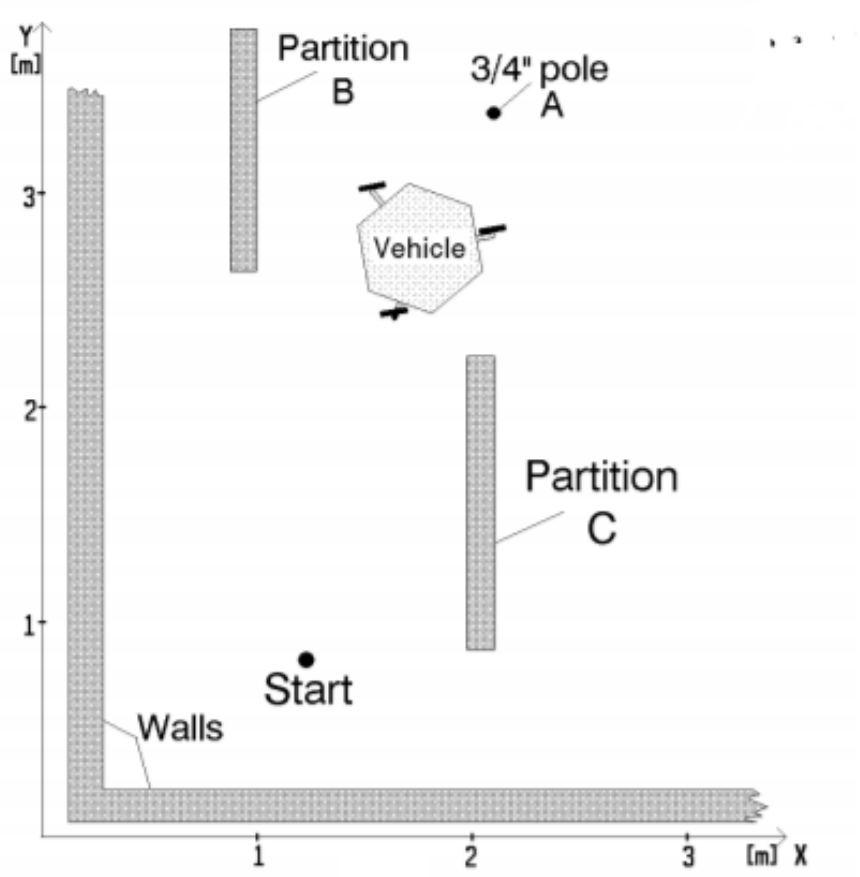


Figure 2.10: Environment with a robot [6].

this, if there is more than one candidate valley, the one which contains the target direction is chosen and the robot steers in this direction. To avoid getting close to obstacles, VFH chooses the center of the candidate valley as the direction to steer in.

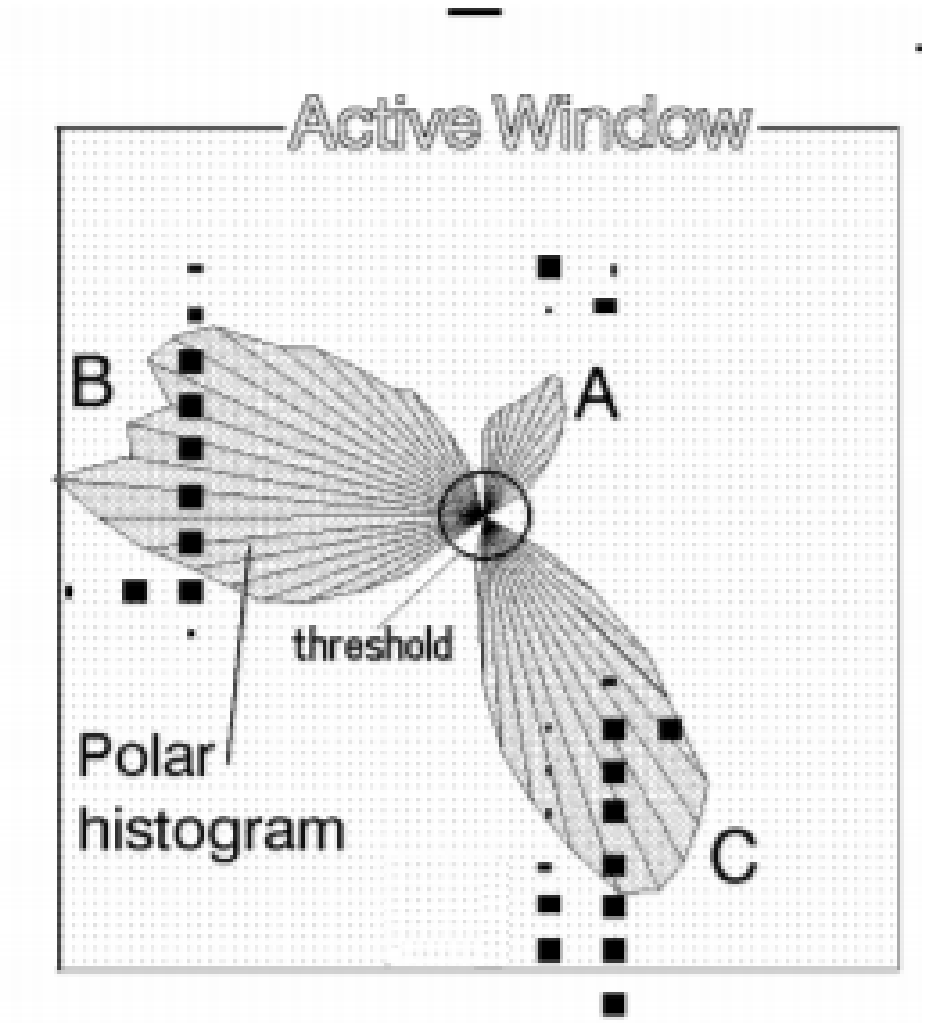


Figure 2.11: Histogram grid (in black) with polar histogram overlaid on the robot [6].

2.4.2 VFH+

The essence of VFH is described in the previous paragraphs. After the original VFH algorithm was proposed, it was revised and further improved to formulate VFH+ and VFH*.

In VFH+ [7], the concepts of considering the dynamics of the robot, enlargement angle, masked and binary polar histograms were added. Every obstacle is dilated by gamma to avoid collision with it. Equation (2.8) shows how gamma is calculated. $r_r + s = r_r + d_s$ where r_r is the robot radius and d_s is the minimum distance to an obstacle; in other words, d_s acts like a safety bubble around the robot. d is the

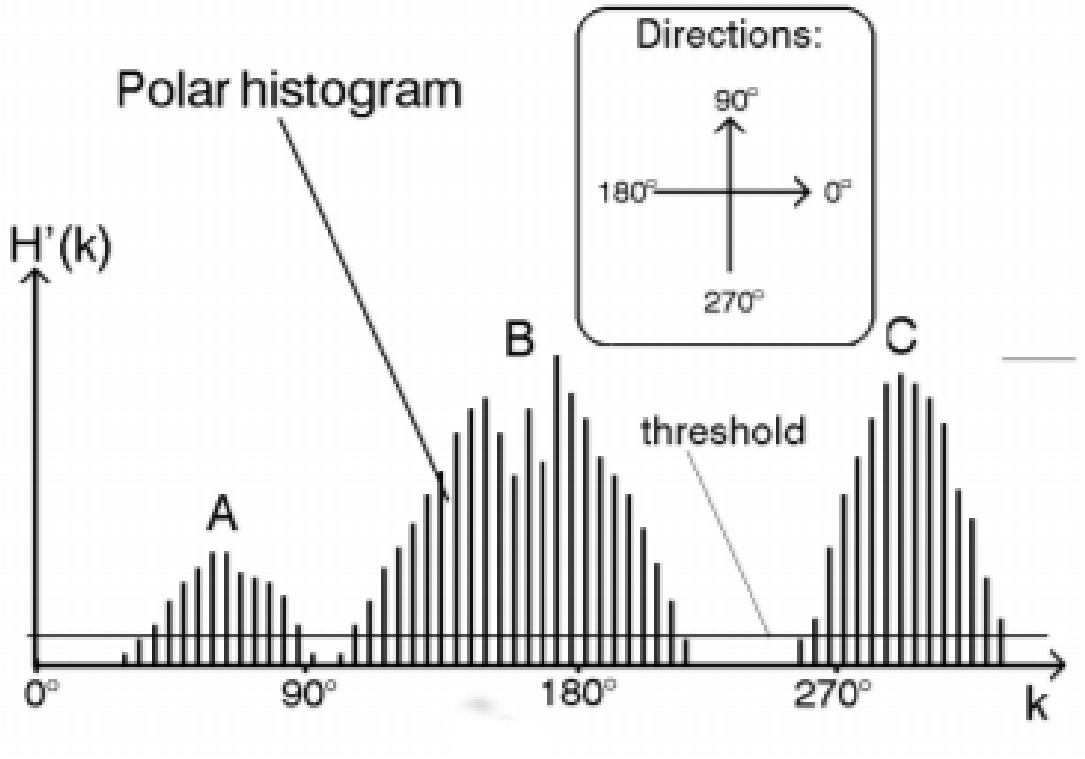


Figure 2.12: Polar histogram with Polar Obstacle Densities and threshold [6].

distance to the obstacle. Figure 2.14 depicts the relevancy of the enlargement angle.

$$\gamma = arcsin(r_{r+s}/d) \tag{2.8}$$

In the original VFH algorithm, the robot is assumed to be able to change its direction immediately. VFH+ considers the dynamics and kinematics of the robot and the minimum left and right turning radii (symbols) are calculated. Knowing r_l and r_r, the possibility of collision with obstacles can be determined before attempting to turn. If a collision is foretold, the area is termed as blocked, else free. This creates a new masked polar histogram. Figure 2.15 shows the original polar histogram described in VFH, the binary polar histogram that VFH uses and the improved masked histogram in VFH+.

After this candidate valleys can be determined using the masked histogram like

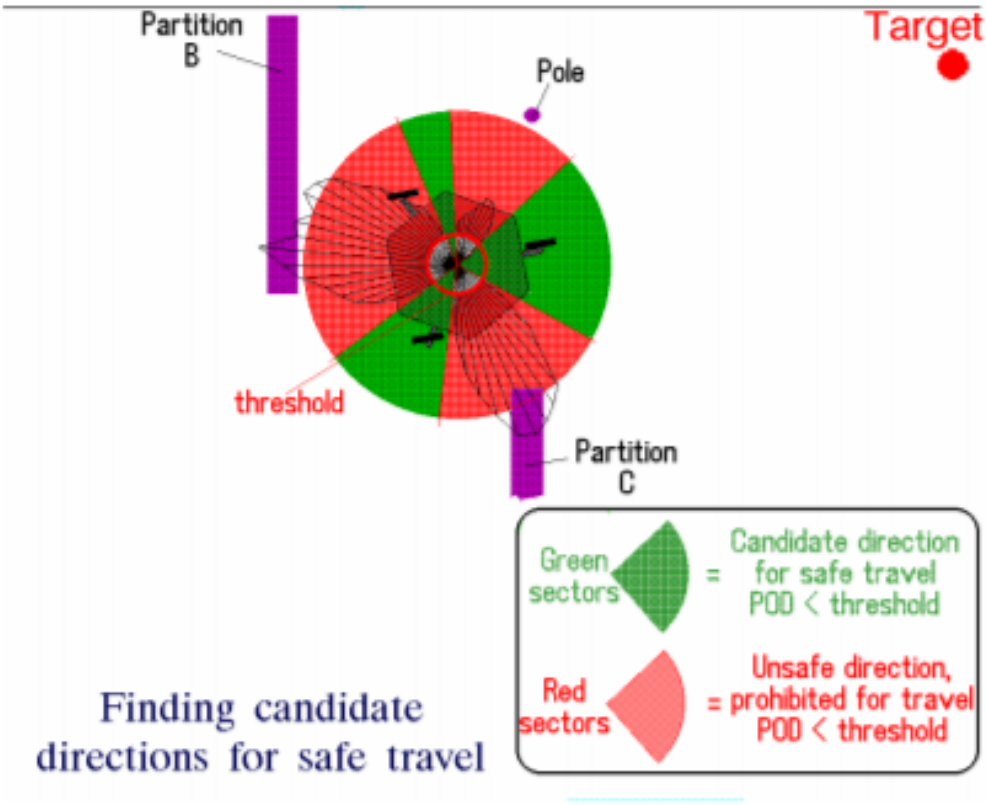


Figure 2.13: Candidate Valleys [6].

the VFH algorithm. In VFH, the center of the candidate valley is chosen as the direction to steer in. Now, a margin of s is added, where s represents the size of sector required for the robot to fit. Candidate valleys are further divided into two categories: narrow and wide. Candidate valleys greater in width than parameter s_max are termed as wide, else termed as narrow. If the selected candidate valley is narrow, 2.9 describes how the candidate direction c_s is chosen.

$$c_s = (k_r + k_l)/2 \tag{2.9}$$

For wide candidate valleys, (2.10), (2.11) describe how the margin limits of the candidate valley are calculated. Equation (2.12) describes how the selected candidate

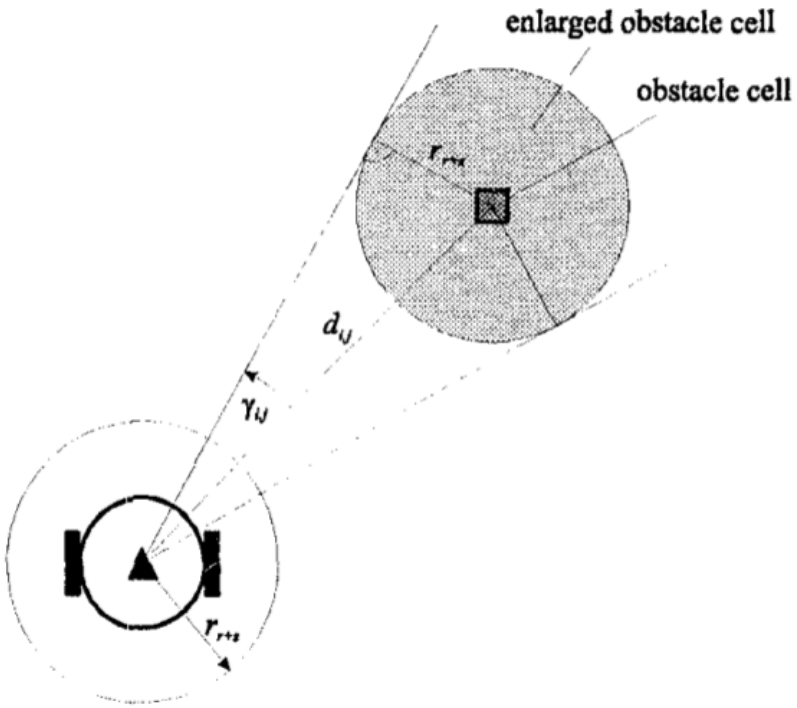


Figure 2.14: VFH+: Enlargement angle [7].

direction c_s is chosen.

$$c_l = (k_l) - s_{max}/2 \tag{2.10}$$

$$c_r = (k_r) + s_{max}/2 \tag{2.11}$$

$$c_s = (c_r + c_l)/2 \tag{2.12}$$

Here k_l and k_r correspond to left and right limits of the candidate valley.

2.4.3 VFH*

VFH+ considers the best local direction to go in. Due to this, it is possible for VFH+ to get stuck in local minima. VFH* [8] applies a look-ahead technique to make sure

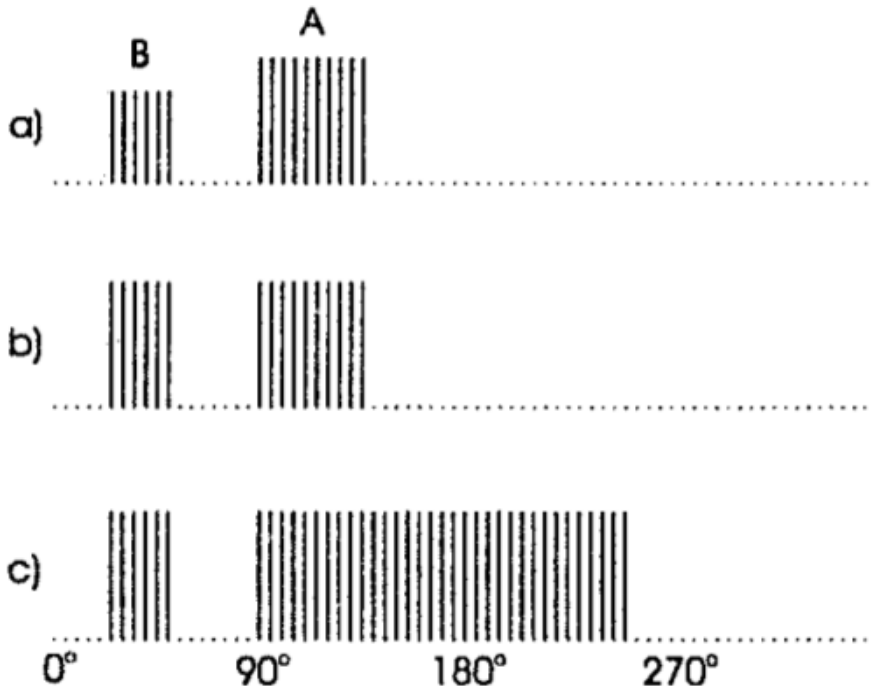


Figure 2.15: a. POD Polar Histogram; b. Binary Polar Histogram; c. Masked Polar Histogram [7].

this doesnt happen, while navigating towards a goal. The name VFH* comes from the fact that VFH+ is combined with A* 2. Figure 2.16 shows a confusing situation for VFH+. Black areas are obstacles, gray areas are the enlarged obstacles and white denotes free space. The figure depicts two directions the VFH+ algorithm may take, with a fifty-fifty chance. If path A is taken, the robot will get stuck. To avoid this, the *projected* VFH+ directions are evaluated in VFH*.

VFH+ provides *primary* candidate directions to go in from the polar histogram. VFH* uses these and envisages the robot's position and orientation after a step d_s. Following this, VFH+ is applied to find the subsequent projected candidate directions for these new positions. This process is done n_g times recursively, in a breadth first search fashion. A node in the graph represents the position and orientation of the robot, while the edges represent the direction the robot goes in. The primary

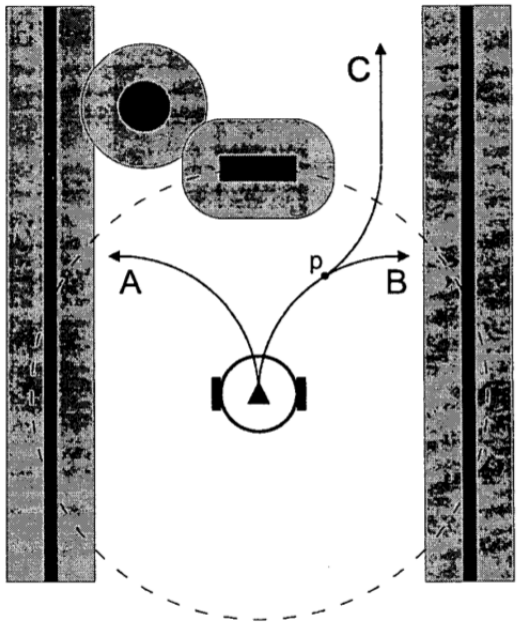


Figure 2.16: A confusing situation for VFH+: Paths A and B can be chosen with equal probability. Path A is not optimal, while Path B is. VFH* overcomes this situation [8].

candidate direction that leads to the goal node with the least cost is selected. Cost is assigned to each node in the graph according A* heuristics. Figure 2.17 shows the chosen branch for varying values of n_g.

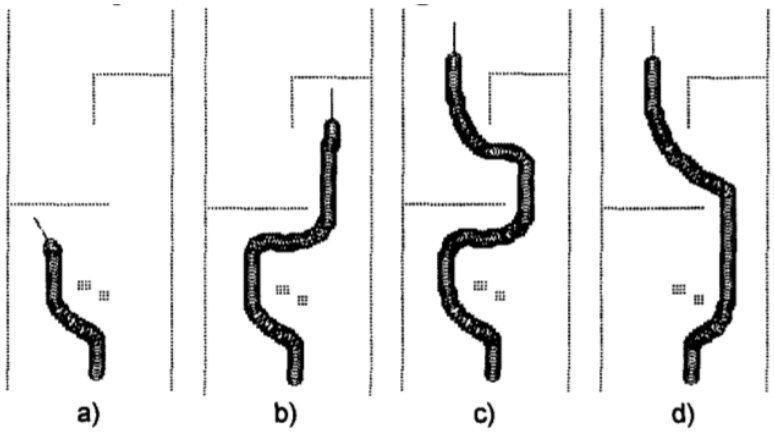


Figure 2.17: VFH* Trajectories for: a. $n_g = 1$; b. $n_g = 2$; c. $n_g = 5$; d. n_g =10 [8].

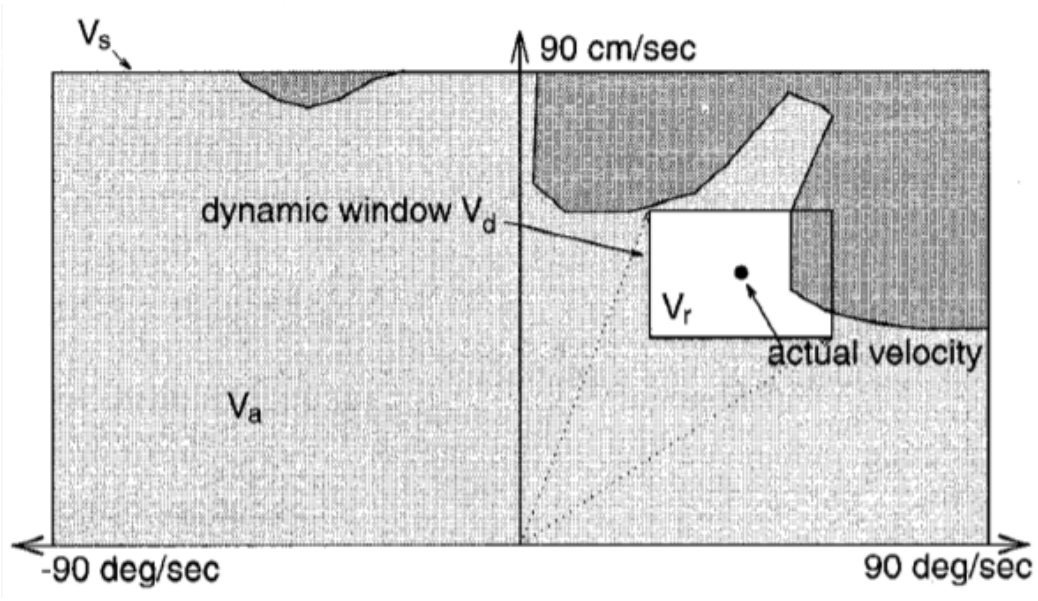


Figure 2.18: Velocity Space: V_r is the white region. [9]

2.4.4 Dynamic Window Approach

The DWA algorithm [9] determines the required linear velocity, v, and angular velocity, ω for the robots dynamics. The velocity pair, $< v, \omega ¿$, is determined by finding the intersection between V_s, V_a and V_d. V_s are admissible velocity pairs that the robot can reach. V_a are velocity pairs where the robot can satisfy (2.13) before hitting any obstacle. Finally, the dynamic window is considered for time t. Velocity pairs that the robot can reach considering the linear acceleration α_v and angular acceleration α_ω give V_d. Figure 2.18 shows the admissible velocity pairs V_r in white.

$$V_r = V_s \cap V_d \cap V_a \tag{2.13}$$

$$G(v, \omega) = \sigma(\alpha * heading(v, \omega) + \beta * dist(v, \omega)\gamma * velocity(v, \omega)) \tag{2.14}$$

With the defined V_r space, the velocity pair that maximizes Eq. () is set as the velocity pair for the robot. α, β,γ are hyper parameters that adjust the importance each of the objective. *Heading* describes the alignment between the goal and current

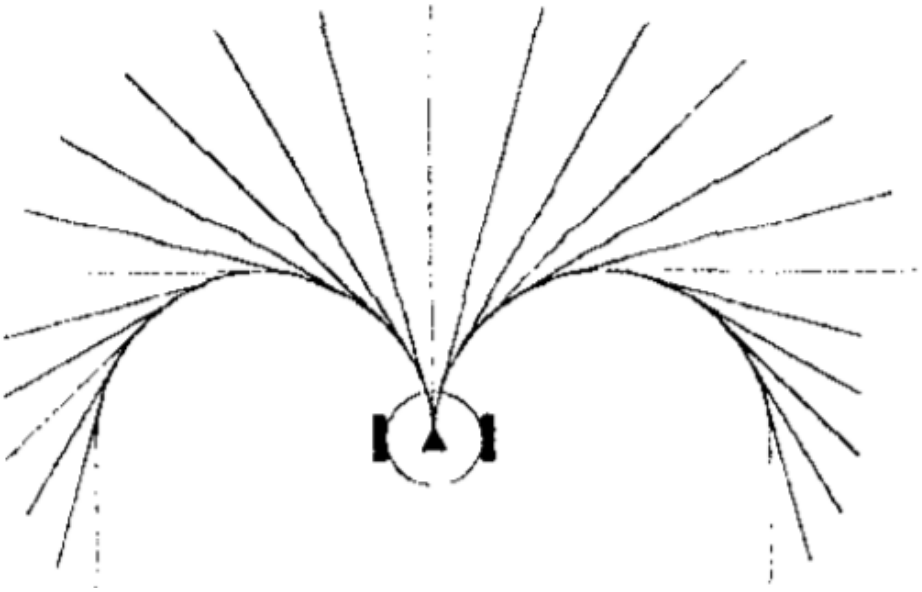


Figure 2.19: Possible curvatures the robot can take [7].

orientation. *dist* gives the distance to the closest obstacle. The closer the obstacle is, the smaller this term becomes. The *velocity* term is a measure of the progress along the curvature. The objective functions for $V_a \cap V_s$ and $V_a \cap V_s \cap V_d$ are shown in Figure 2.20 and Figure 2.21 respectively. The vertical line denotes the max v, ω pair.

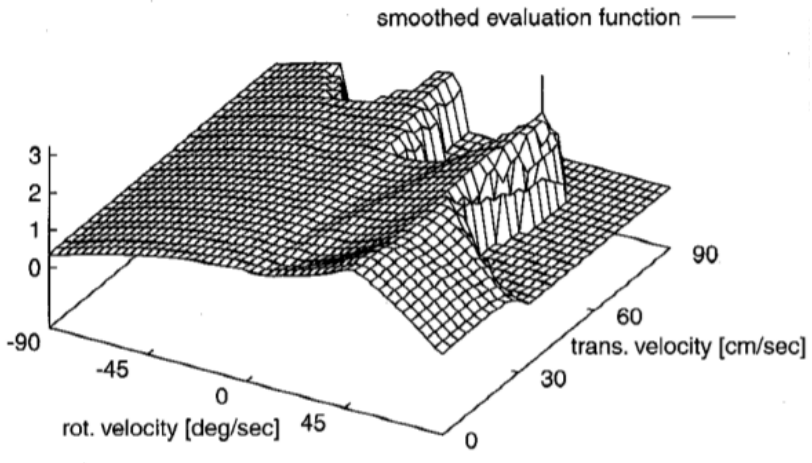


Figure 2.20: Objective function for $V_a \cap V_s$ [9].

Applying the DWA technique allows robots to pass through door ways at ap-

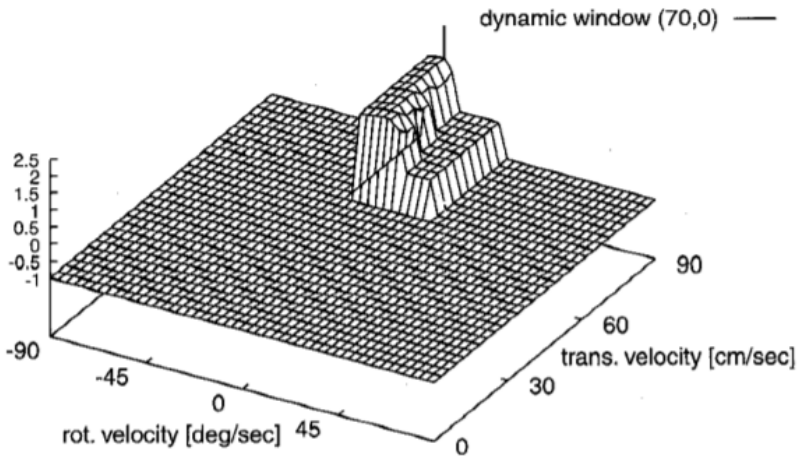


Figure 2.21: Objective function for $V_a \cap V_s \cap V_d$ [9].

propriate velocities. Figure 2.22 shows the behavior of a robot at different velocity pairs.

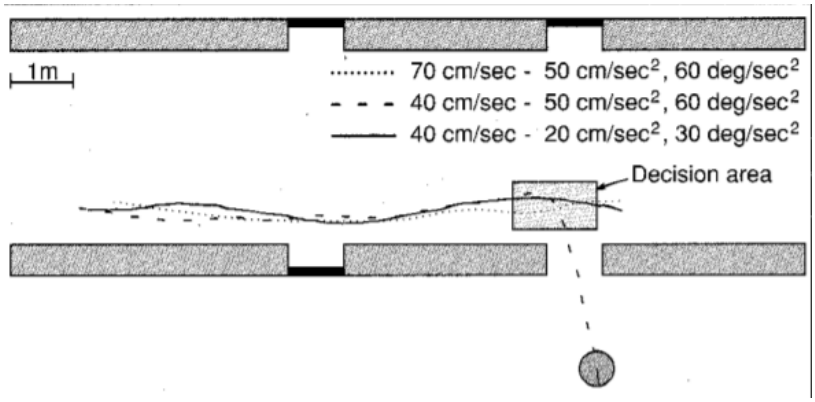


Figure 2.22: Paths taken robot for given velocity pairs [9].

2.5 Object Detectors

2.5.1 YOLO

The You-Only-Look-Once (YOLO) framework [10] is a real-time object detector and recognizer. It runs at 45fps. YOLO is trained to classify 20 different classes, belonging to PASCAL VOC 2007 [22]. To evaluate the performance of the classifier, the mean of the accuracies of each class is calculated. This evaluation measure is termed as

mean Average Precision (mAP). YOLO's performance is 64.4 mAP for the twenty classes.

Other object detectors like R-CNN [23], break down object detection and recognition into parts. The first step gets candidate regions using methods like selective search. This is termed as a region proposal network (RPN) and outputs bounding boxes where objects are likely to be present. These region proposals are then passed through a classifier. Post-processing involves merging overlapping bounding boxes, removing duplicate boxes and re-scoring. Since each of these systems, RPN, classification and post-processing are separate, it is harder and slower to train. Also, unlike humans, these detectors take multiple passes at the image before giving its final output, allowing them to run at 5-18fps, which is too slow for real time applications.

With a single pass, a human can distinguish a variety of objects. YOLO takes inspiration from this, and aims at making its net not deeper, but the concept easier to learn. Instead of breaking down the task into object detection followed by object recognition, a single network is used. Images are divided into a grid of 7×7. Each cell predicts two bounding boxes. Each bounding box predicts how confident it is that an object is present within it, $P(Object)$. Also, the bounding box itself, has a confidence level. This makes the network learn to predict better bounding boxes per cell. The image is also spatially divided, where each cell predicts which class is present, given an object is present, $P(class|Object)$. Using this, along with the bounding boxes above the threshold, a label is given to the bounding box.

The classifier learns to predict better bounding boxes and generalizes to which objects occur together. YOLO makes use of DarkNet [24] for training.

Having a real-time object detector and recognizer can be used in many different application areas. Self - driving cars and robots can make use of these to improve their understanding of the world and react immediately. [25], illustrates the relation between the fps of the detector and the distance traveled by a car at an average speed

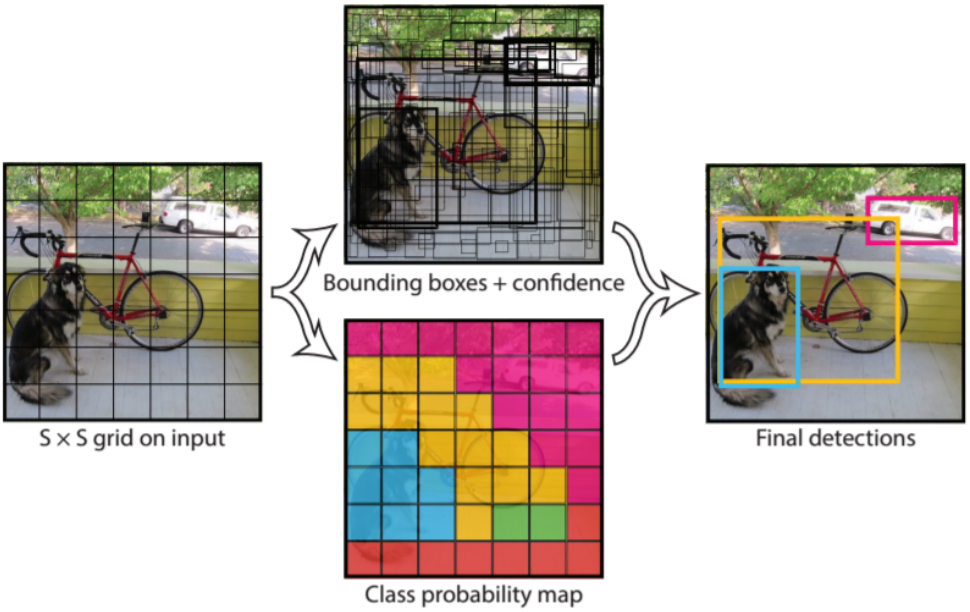


Figure 2.23: YOLOv1 Example: Image divided into 7× 7 grid. Each cell predicts two bounding boxes (top). Each cell predicts a class given an object is present (bottom) [10].

of 60mph. Table 2.1 shows the relation between speed of the classifier (fps) and the speed of the vehicle, with the distance traveled by the car.

Table 2.1: Relation between speed of model and distance travelled by vehicle.

Model	Speed	Distance (in ft)	mAP
R-CNN	0.05fps	1742.4	66
Fast R-CNN	0.5fps	176	20
Faster R-CNN	7fps n	12	73.2
YOLO	45fps	3	63.4

2.5.2 YOLOv2 and YOLO9000

The next iteration of YOLO makes several improvements to make it better, faster and stronger [11]. Making use of batch normalization, the mAP went up by 2%. Batch normalization helps to regularize the model, making it generalize better and prevent overfitting. It often removes the need of implementing dropout. By finetuning on

higher resolution images, such as 448 × 448 pixels, better filters were learned. Taking inspiration from RPN, instead of predicting co-ordinates of the bounding box in the final layer, offsets from the cell are determined. Each bounding box now predicts class and object predictions. Object prediction is the intersection over union (IOU) of the ground truth and bounding box. IOU is the ratio of the overlapping area between the predicted and ground truth bounding boxes over the union of the area of the two boxes. The class prediction is the probability of a class given that an object is present. Providing priors for bounding boxes can make the detector better. To get these priors, k-means clustering was run on the training data. Figure 2.24 shows the priors calculated. k=3 was chosen for YOLO9000 and k=5 for YOLOv2.

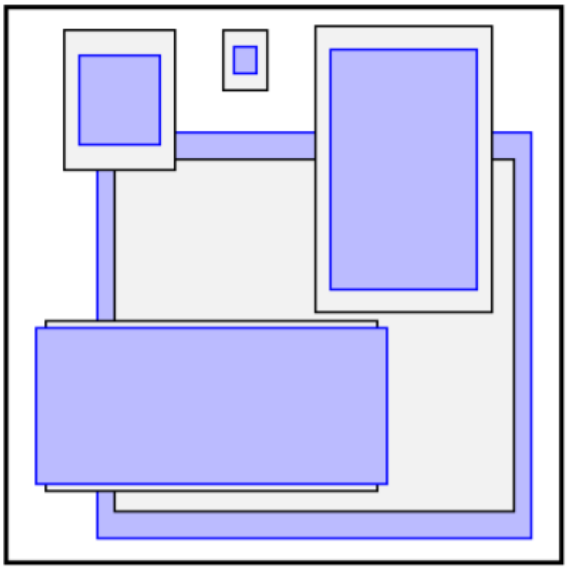


Figure 2.24: Image shows bounding box priors for k=5. Light blue boxes correspond to the COCO dataset while the white boxes correspond to VOC [11].

YOLO has two losses, one related to the bounding box prediction (object detection) and one related to the class (object recognition). Ground truths for the object detection part are limited however that for object recognition is generally more. For example, there are many images with bounding boxes provided for dog but few that would give the bounding box for a Dalmatian. However, there are many examples of Dalmatian in a class dataset, like ImageNet. Making use of hierarchical relationship

between labels, the object detection pipeline can be provided with more data and the model can learn to predict new classes. YOLOv2 performance improves to 76.8mAP at 40fps for 544 $\times$ 544 pixel images.

Considering YOLOv2s accuracy and speed, for this work, YOLOv2, trained on the COCO dataset with 80 classes has been used.

2.6 Smart Wheelchairs

A smart wheelchair is an electric wheelchair, to which changes have been made to control the motors via software, or an autonomous robot upon which a chair has been mounted. Research and innovation in smart wheelchairs has been ongoing since 1986 [26]. Smart wheelchairs aim at supporting a specific niche of wheelchair users: users with vision issues, tremors, cerebral palsy, etc. Smart wheelchairs can exhibit one or more of the following capabilities [26]:

- Travel to a destination while avoiding obstacles
- Avoid obstacles while user controls main directions to go in
- Follow a wall at a fixed distance
- Pass through door ways
- Follow a path already traveled by the robot
- Rewind movements taken by robot up to starting spot
- Follow or go up to a moving or stationary object.
- Plan and follow a line through environment with intersections.
- Turn on spot or make a K-turn

The following sections go over a few smart wheelchairs and the technology they use briefly to be semi-autonomous or autonomous.

2.6.1 COALAS

The COgnitive Assisted Living Ambient System [12] is a smart wheelchair equipped with an omni-directional camera, twelve infrared sensors, eleven sonars, Arduino boards, a microphone and encoders. COALAS makes use of the Robot Operating System for system integration. The wheelchair avoids obstacles by fusing input from

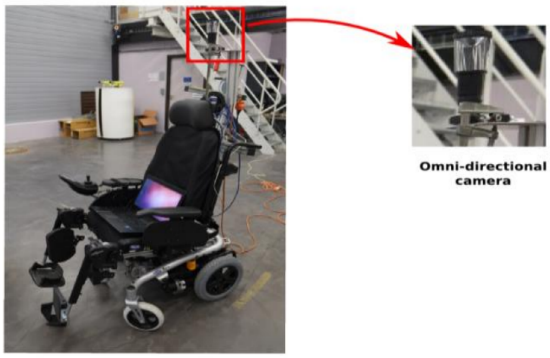


Figure 2.25: Smart Wheelchair: COALAS [12].

its sonar sensors and joystick. An artificial potential field is created, making it avoid obstacles and go in the general direction given by the user via the joystick.

COALAS makes use of geo-localization; it is driven along all the possible paths it can take, during which, the omni-directional camera takes photos and stores them. Feature matching between the perceived images and stored photos, using SIFT, is carried out when traveling along a previously driven path.

2.6.2 MIT's Smart Chair

Research at MIT for smart chairs delves into making the robot recognize different locations on its own. The smart chair at MIT aims at acting as a tour guide [27]. Initially, it follows a human guide through MIT; the person narrates and describes different areas within the building [13]. The robot then interprets the spoken utterances, and labels parts of the map with the help of scene categorization. Figure

2.26 shows an example case of locations on a map. The pie charts denote the class probability of a location.

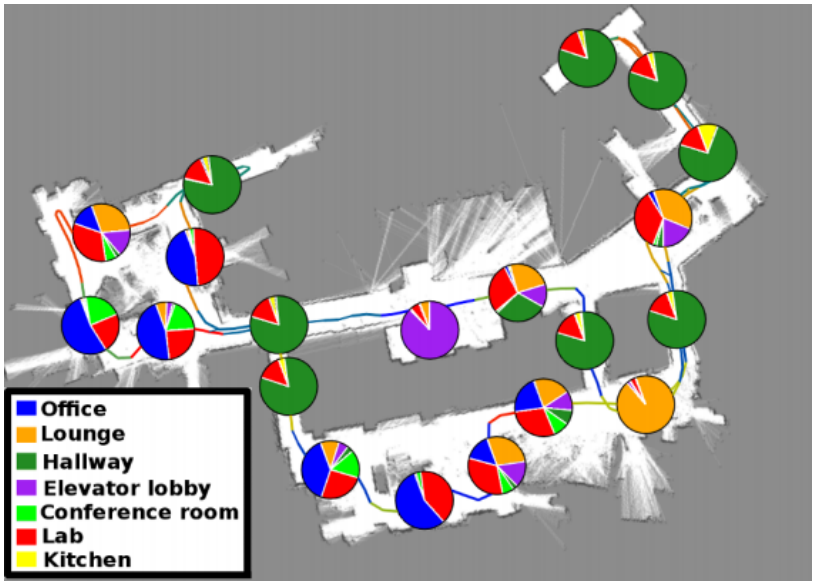


Figure 2.26: Location Map with Class probabilities [13].

Chapter 3

System Overview

3.0.1 Milpet

Figure 3.1: Milpet.

For testing the obstacle avoidance and path planning in practice, a robot called Machine Intelligence Laboratory's Personal Electronic Transport or Milpet, for short, has been used. Milpet is an electric wheelchair aimed at giving independence and privacy back to its disabled user. Figure 3.1 shows Milpet. Milpet can be controlled with speech commands like "Milpet, take me to room 237". Milpet then, navigates and avoids collisions with obstacles, to reach its goal. To do this, Milpet is equipped with special hardware and software.

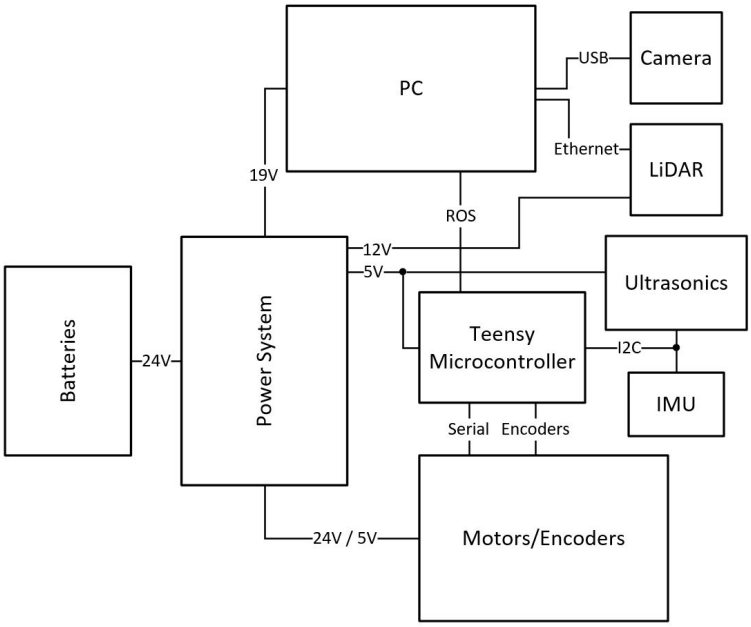


Figure 3.2: Hardware Block Diagram.

3.0.2 Hardware

Milpet is equipped with an onboard computer, the Intel NUC6i7KYK Mini PC as shown in Figure 3.5. The mini PC runs the high level navigation algorithms, and is the main center of software control. There is a microcontroller, Teensy 3.2, for running the low level commands. Figure 3.3 shows the pinout of the Teensy board. Motor control, encoder comprehension, lights control, etc. are handled by the Teensy. Figure 3.2 shows the hardware block diagram and Figure 3.4 shows the hardware on

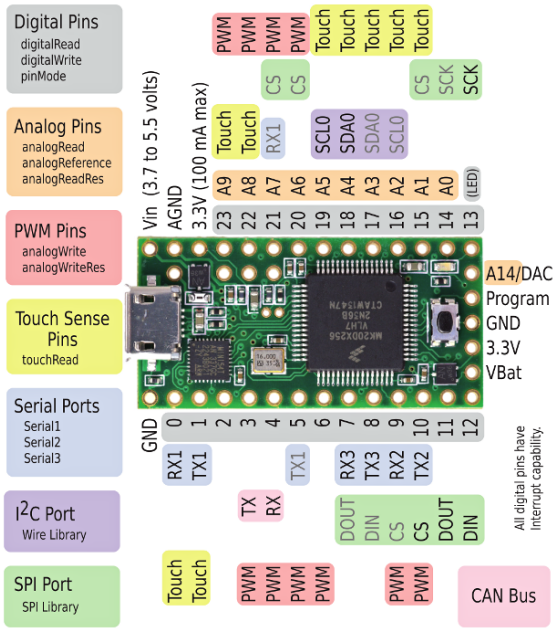


Figure 3.3: Teensy microcontroller's Pin Out.

the chair.

Milpet has encoders, sonar sensors, a Hokuyo USTLX-10 Lidar, an omnivision camera and a front facing camera. Ubuntu 16.04 and the Robot Operating System (ROS) Kinetic Kame are installed on the NUC. The next sections, introduce the software components. Figure 3.6 shows a picture of the Lidar.

3.0.3 Software

The on-board computer runs Ubuntu 16.04 and ROS Kinetic Kame. The following sections describes ROS and Gazebo for running simulations.

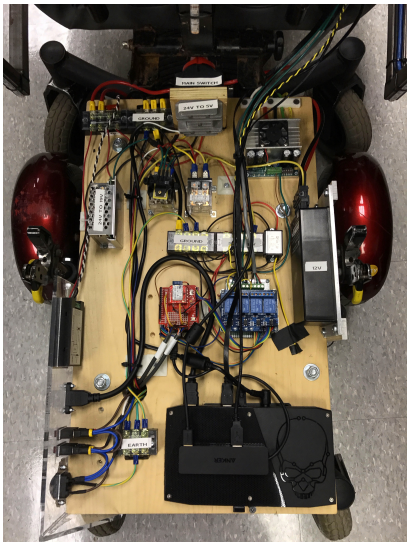


Figure 3.4: Milpet Hardware.

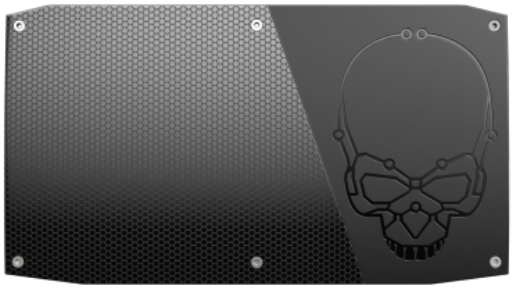


Figure 3.5: Intel NUC Mini PC.

Figure 3.6: Lidar Sensor: Hokuyo UST 10LX.

3.1 Robot Operating System

The Robot Operating System (ROS) [28] offers various packages and tools for robotic applications. ROS has packages for sensor interfacing, obstacle avoidance, mapping, path planning and tools for sensor output visualization, etc. are available. ROS allows message passage and therefore system co-ordination, between programs running on the same or different systems. It is an open-source framework.

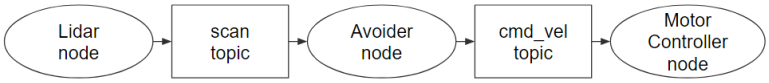
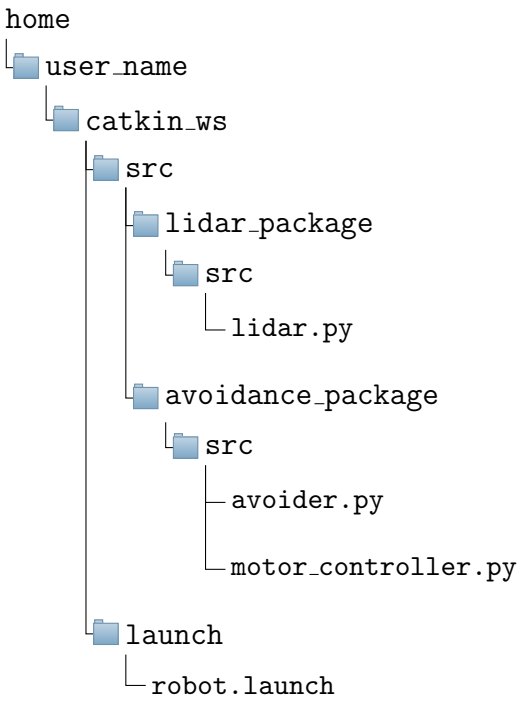


Figure 3.7: ROS nodes and topics related to described scenario.

The programs that use the ROS framework, are termed as 'ROS nodes'. Nodes having similar or related functionality are added in the same 'ROS package'. The collection of various 'ROS packages', are added in a 'workspace'; conventionally the ROS workspace is named 'catkin_ws'. Multiple ROS nodes can be started up, by running a single 'ROS launch' file.

ROS nodes can be written in C++, Python, Lua, Lisp and Java. The ROS framework is supported for Ubuntu and Debian platforms. For OS X, Gentoo and OpenEmbedded/Yocto plaftforms, ROS is in the experimental stage.

A general ROS file hierarchy is shown below:



Multiple ROS nodes running in parallel can run independently. To make them communicate with each other, message passage is required. Messages in ROS are sent to 'ROS topics'; the node that sends the message out is a 'ROS publisher'. Any node running within ROS can listen to messages being published to a topic, without knowing the publisher, this node is called a 'ROS subscriber'.

For example using the above file heierarchy, an autonomous agent equipped with a Lidar may have the following nodes: The 'lidar.py' node is interfaced with the Lidar and publishes the scans it reads in, to 'scans' topic. The 'avoider.py' node subscribes to the 'scans' topic and uses it to decide what linear and angular velocity combination (Twist message) the robot needs for moving. This message is published to the 'cmd_vel' topic. Finally, the 'motorController.py' node subscribes to the 'cmd_vel' topic to control the motors.

Figure 3.7 describes the example scenario, boxes are around the topics, nodes are in an oval and the arrows show direction the messages are passed in.

There are different pre-defined message types in ROS to help in robotic applications; a few of them are:

- Twist: Message for linear and angular velocity
- Pose2d: Message describing the position of a robot in X, Y co-ordinate space, with pose, as x, y, ϕ.
- Scan: Message with the ranges and intensities of a Lidar scan
- String: Message of data type String

3.1.1 Useful ROS commands

Some ROS commands are described in Table 3.1. Figure 3.8 shows an overview of

Table 3.1: Useful ROS commands.

Command	Description	Example
rosrun package_name file_name	Used to run a ROS node	rosrun lidar_package lidar.py
roslaunch package_name launch_file_name	Used to launch multiple nodes, defined in the launch file, with a single command	roslaunch launch robot.launch
rqt_graph	Used to generate a figure containing all the nodes and topics running in ROS	rqt_graph
rostopic list	Lists the present ROS topics	rostopic list
rostopic echo topic_name	Display messages being received by the ROS topic	rostopic echo /cmd_vel
rosnode list	Lists the running ROS nodes	rosnode list
catkin_create_pkg package_name	Creates a ROS package when run in the main /src folder	catkin_create_pkg lidar_package
rospack find package_name	To check if a package is present or not; gives a path to that package	rospack find avoider_package
roscore	Sets up main ROS thread so other nodes can run; not required if roslaunch is used	roscore

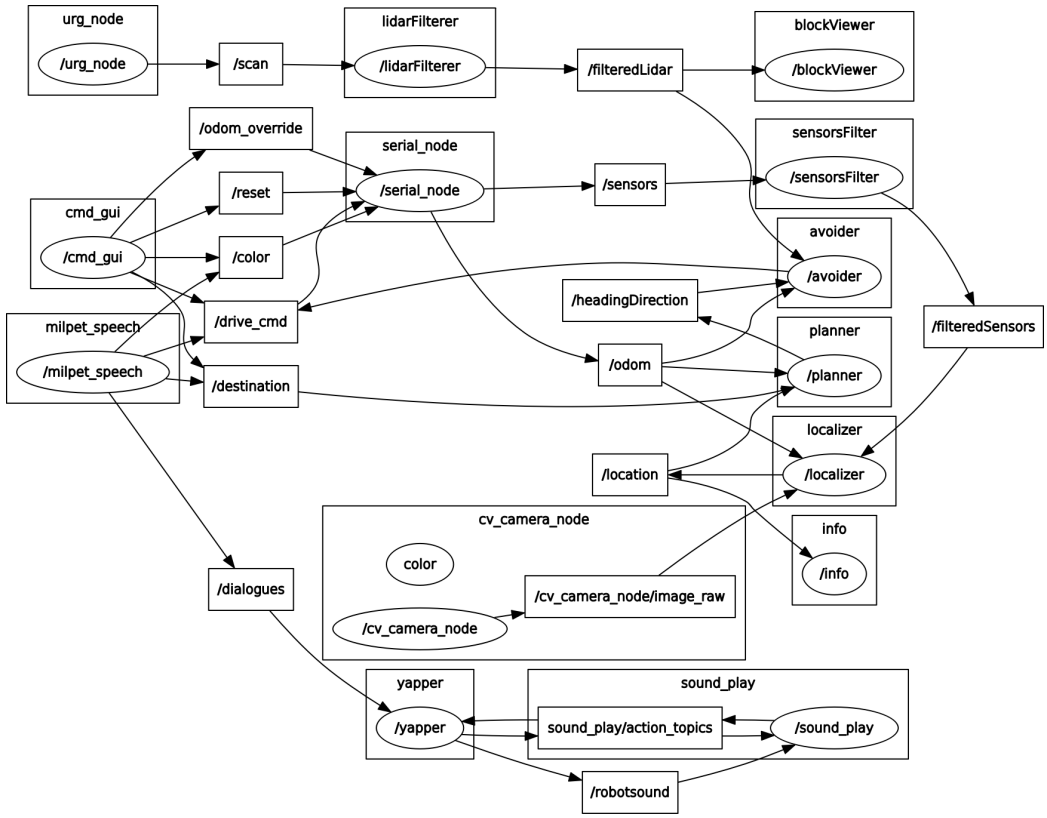


Figure 3.8: System architecture generated by ROS.

the ROS nodes and topics running on Milpet This diagram was genereated by ROS with the command 'rqt_graph'. All the nodes, excluding the serial node, which runs on the Teensy, run on the computer. The Speech, CommandGUI and Talker nodes make up the interaction nodes for Milpet. The Avoider, Path Planner and Localizer make up the navigation system. The Urg and CV_Camera nodes interface between the Lidar and omnivision camera. The rectangles with the ovals inside them represent nodes, while the rectangles alone represent topics. Arrows between the nodes show which messages another node subscribes to.

Table 3.2 shows the publishing rate of different topics on Milpet. The 'scan' topic publishes at 40 hertz (Hz), this reflects the operating frequency of the Lidar sensor. The 'drive_cmd' topic receives commands at 10 Hz, which are passed on to the motor driver. A higher frequency causes the robot to jitter as the motor cannot react fast enough. Lidar filtering is performed over three scans resulting in a frequency one

third of that of the 'scan' topic.

Table 3.2: ROS Topic Rates.

Topic Name	Frequency (in Hz)
/scan	40
/filteredLidar	13
/odom	20
/drive_cmd	10

A tutorial for installing and getting familiarized to ROS, is in the next section.

3.2 Gazebo

Gazebo is a 3-D simulation tool where robots can be designed and tested. Gazebo provides a physics engine, graphics capabilities and allows rapid testing. Gazebo provides packages that link to ROS. With these packages, code can be first tested in simulation and then the same code can be run on physical robots, without making changes to the code.

Gazebo allows one to design their own robot model or use one of the included models: Turtlebot, Pioneer 2-DX, Baxter, etc. Sensors like laser range finders and cameras, with optional noise, can be added to a custom robot. Figure 3.9 shows a screenshot Gazebo's interface. The left side pane gives the properties of objects in the world. The world is displayed in the large right pane. Robots can be dragged and dropped into the environment. In Figure 3.9, a Pioneer 2-DX was dragged and dropped after clicking on the 'Insert' tab.

The following sections will go over the basic concepts needed to build a Gazebo model, followed by a Gazebo and ROS tutorial.

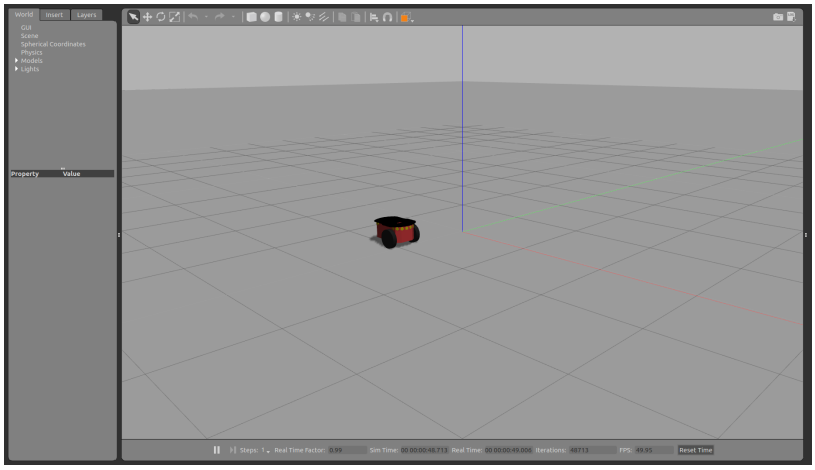


Figure 3.9: Screenshot of Gazebo Environment.

3.2.1 Links and Joints

In Gazebo, links and joints make up a robot. Solid parts of the robot are termed as links and points where the links are fastened are termed as joints.

Links can be of different shapes: box, sphere or cylinder. The dimensions of these are defined in a Simulation Description Format (.sdf), Universal Robotic Description Format (.urdf) or a XML Macros (.xacro) file (Further details in next subsection). Links are defined with collision, visual and inertial tags. The visual tag makes the link visible in Gazebo, while the collision tag makes the link a rigid body; this entails collision, with other rigid bodies. The size dimensions of the links, in the collision and visual tags, are the same. Finally, the inertia tag gives the moment of inertia (MOI) of the rigid body, with mass m. MOI tensors are filled within the inertia tag. Equations (3.1), (3.2), (3.3) [29] give the MOI tensors of a cuboid, with dimensions $w \times b \times h$, sphere, with radius r, and cylinder, with radius r and height h.

$$MOI_{cuboid} = \begin{pmatrix} \frac{1}{12}m(h^2+d^2) & 0 & 0 \\ 0 & \frac{1}{12}m(w^2+d^2) & 0.0 \\ 0 & 0 & \frac{1}{12}m(w^2+h^2) \end{pmatrix} \tag{3.1}$$

$$MOI_{sphere} = \begin{pmatrix} \frac{2}{5}mr^2 & 0 & 0 \\ 0 & \frac{2}{5}mr^2 & 0.0 \\ 0 & 0 & \frac{2}{5}mr^2 \end{pmatrix} \tag{3.2}$$

$$MOI_{cylinder} = \begin{pmatrix} \frac{1}{12}m(3r^2+h^2) & 0 & 0 \\ 0 & \frac{1}{12}m(3r^2+h^2) & 0.0 \\ 0 & 0 & \frac{1}{2}m(r^2+h^2) \end{pmatrix} \tag{3.3}$$

The types of joints are revolute, continuous, prismatic, fixed, floating and planar[30]. Table 3.3 describes each type of joint. Figure 3.10 shows a revolute joint (left) and prismatic joint (right).

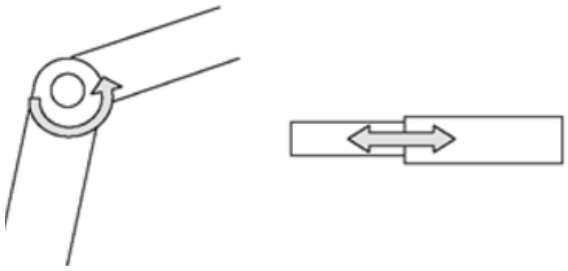


Figure 3.10: Revolute joint (left); Prismatic joint (right).

Table 3.3: Different types of joints.

Joint Type	Movement Description
Fixed	Does not move
Continuous	Rotates about specified axis
Revolute	Rotates about specified axis for a limited range
Prismatic	Moves linear i.e. slides along the specified axis for a limited range
Floating	Moves about six degrees of freedom
Planar	Movement in a plane perpendicular to the specified axis

While building a robot, each joint is defined with a type, parent and child links, an origin and axis of movement. The origin refers to how far the joint is away from the parent link, with its frame of rotation. For example, if the $< roll, pitch, yaw >$, $< r, p, y >$, for a joint 1 is $< 0.78, 0, 0 >$, it means that the frame of reference of the

joint is rotated about X axis by 45° w.r.t. its parent link. Figure 3.11 and Table 3.4 show the relation between the origin of joints for links 1,2,3,4 [gazebo joint ref]. To find the rotated frames, with the right hand, place the thumb along the Z axis, and the index and middle fingers along the X and Y axes, respectively. Turning the hand towards the inner palm, gives a positive change in angle, and outwards rotation gives negative change in angle.

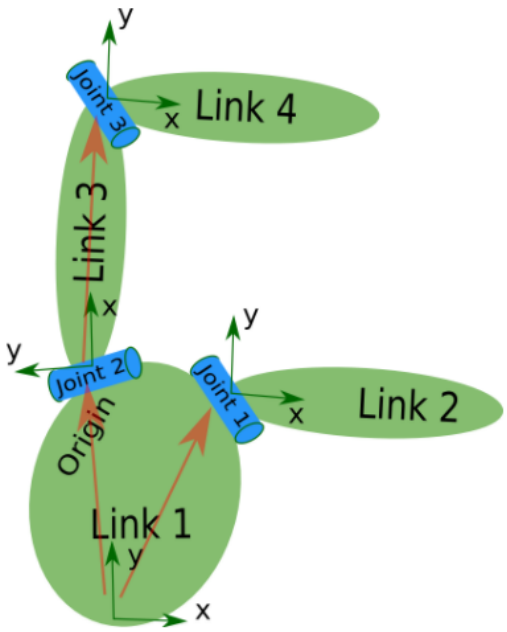


Figure 3.11: Relation between links and joints of a robot model, with reference frames of each joint ref[gazebo].

Table 3.4: Relation between links and joints of a robot model.

Joint	Parent Link	Child Link	Origin	$< r, p, y >$	Reference Frame
1	1	2	5,3,0	$< 0, 0, 0 >$	No rotation of frame
2	1	3	-2,5,0	$< 0, 0, 1.57 >$	Rotated about Z axis by 90°
3	3	4	5,0,0	$< 0, 0, -1.57 >$	Rotated about Z axis by -90°

3.2.2 ROS and Gazebo Tutorial: Creating a Robot in Gazebo, controlled via ROS

A Linux environment for ROS and Gazebo is encouraged; this tutorial was completed on Ubuntu 16.04, with ROS Kinetic Kame and Gazebo7.

1. Install ROS

   Run the following commands to install ROS Kinetic Kame or follow instructions from the official ROS documentation at [refer ros]:

   ```
   sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main"
   > /etc/apt/sources.list.d/ros-latest.list'

   sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key
   421C365BD9FF1F717815A3895523BAEEB01FA116
   sudo apt-get update
   sudo apt-get install ros-kinetic-desktop-full
   sudo rosdep init
   rosdep update
   echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
   source ~/.bashrc
   ```

2. Install Gazebo: Install Gazebo 7 by running the following or follow the step-by-step instructions at [gazebo]:

   ```
   sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu-stable `lsb_release
   -cs` main" > /etc/apt/sources.list.d/gazebo-stable.list'
   wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -
   sudo apt-get update
   sudo apt-get install gazebo7
   sudo apt-get install libgazebo7-dev
   ```

   To check if the installation was successful, run:

```
gazebo
```

A window similar to Figure 3.9 should pop up.

To go over more tutorials about Gazebo, visit: http://gazebosim.org/tutorials.

3. To allow Gazebo to work with ROS, install Gazebo-ROS Packages by running:

```
sudo apt-get install ros-kinetic-gazebo-ros-pkgs ros-kinetic-gazebo-ros-control
```

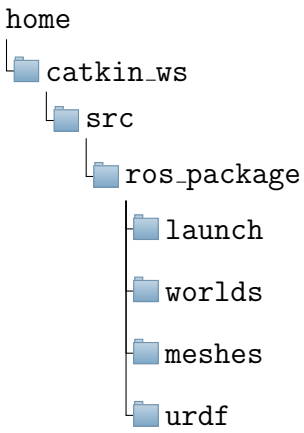
4. Creating a Catkin Workspace:

   The catkin workspace is where all the ROS code resides. It contains the packages, messages and robot models needed to run and simulate robots. To make Catkin workspace run the following on the terminal:

```
mkdir catkin_ws
cd catkin_ws
catkin_make
```

5. Making the file structure:

   The file structure for modeling a Gazebo robot within ROS is as follows:



   Make the /src directory within catkin_ws.

```
cd catkin_ws
mkdir src
```

Create the ROS package *robot*, where the robot will run from:

```
catkin_create_pkg robot
```

Make the following directories within the robot package using the mkdir command: /launch, /worlds, /meshes, /urdf.

6. Writing the world file:

   The world file contains the initial layout of the simulated scene. Go into the directory /catkin_ws/src/robot/worlds and create a file named 'robot.world'. Paste the following text within it:

```
<?xml version="1.0" ?>
<sdf version="1.4">
  <!-- We use a custom world for the robot -->

  <world name="default">
    <include>
      <uri>model://ground_plane</uri>
    </include>

    <!-- Global light source -->
    <include>
      <uri>model://sun</uri>
    </include>

  </world>
</sdf>
```

   Here, an empty plane and a light source are added to the world.

7. Writing launch file:

With a launch file, whatever needs to be launched in the simulator is started. For instance, the initial scene setting (world file) and the robot (xacro file) should be launched when starting the simulator. Inside /catkin_ws/src/robot/launch, create a file named 'robot.launch'. Paste the following text:

```
<launch>

  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find robot)/worlds/robot.world"/>
  </include>

</launch>
```

'empty_world.launch' is a blank scene already provided, after initializing with it, the world file, 'robot.world' is added. Run the following command, from the terminal, to see how a Gazebo simulation is opened via ROS:

```
roslaunch robot robot.launch
```

A screen similar to Figure 3.12 should pop up.

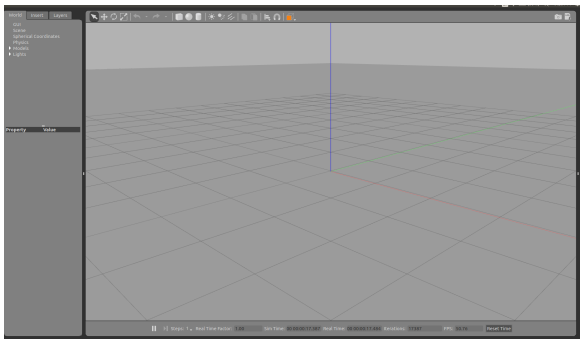


Figure 3.12: Gazebo simulation launched via ROS.

There are multiple pre-made worlds provided by Gazebo. Try launching these too; replace the line :

'$< argname = "world\_name" value = "\$(findrobot)/worlds/robot.world" / >$'

with any world file found in /usr/share/gazebo/worlds/ (this path may be different, depending on the installation). Running the above command,, with 'willowgarage.world', gives Figure 3.13.

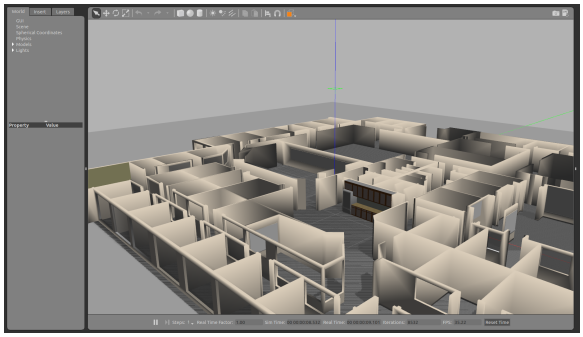


Figure 3.13: Gazebo simulation launched via ROS with Willow Garage.

8. Writing the Xacro file

   Now, in /catkin_ws/src/robot/urdf, create a file named 'robot.xacro'. In this file, the robot model will be made. A small simple robot will be made with one castor wheel and two wheels on either side, with a Hokuyo Lidar on the top. Each joint will have its parent link above it and child link below it, where-ever possible. Paste the following snippet at the top of 'robot.xacro':

```
<?xml version="1.0"?>
<!-- Mini robot -->
<robot name="robot" xmlns:xacro="http://www.ros.org/wiki/xacro">
```

   - Robot Links: The links of this robot consist of a castor wheel (spherical link), two wheels (cylindrical link), a chassis and Lidar (box-shaped links). Table 3.5 describes the links and their origins. Using (3.1), (3.2), (3.3), the MOI tensors are calculated. Each link has a visual, collider and inertial tag, as previously explained.
   - Robot Joints: For this robot, there are three continuous joints for each of the wheels and a fixed joint for attaching the Lidar. Table 3.6 has the

Table 3.5: Robot Link Details .

Link Name	Geometry and Dimensions	Origin	$< r, p, y >$	Mass
Chassis	Box (0.4,0.2,0.1)	0,0,0.5	$< 0, 0, 0 >$	1
Castor	Sphere (0.5)	0,0,0	$< 0, 0, 0 >$	2
Left Wheel	Cylinder (0.5,0.05)	0,0,0	$< -1.57, -1.57, 0 >$	1
Right Wheel	Cylinder (0.5,0.05)	0,0,0	$< -1.57, -1.57, 0 >$	1
Lidar	Box (0.1,0.1,0.1)	0,0,0	$< 0, 0, 0 >$	1e-5

type, origins and axis of rotation for each joint.

Table 3.6: Robot Joint Details.

Joint	Type	Parent Link	Child Link	Origin	$< r, p, y >$	Rotation
1	Continuous	Chassis	Castor	0.15, 0,-0.05	$< 0, 0, 0 >$	1, 0, 1
2	Continuous	Chassis	Left Wheel	-0.1, 0.13, -0.05	$< 0, 0, 0 >$	0, 1, 0
3	Continuous	Chassis	Right Wheel	-0.1, -0.13, -0.05	$< 0, 0, 0 >$	0, 1, 0
4	Fixed	Chassis	Lidar	0,0,0.15	$< 0, 0, 0 >$	N/A

Making use of Table 3.5 and Table 3.6, the xacro file can be completed:

```
  <!-- Base Link -->
  <link name="chassis">
    <collision>
      <origin xyz="0 0 0.05" rpy="0 0 0"/>
      <geometry>
<box size=".4 .2 .1"/>
      </geometry>
    </collision>
    <visual>
      <origin xyz="0 0 0.05" rpy="0 0 0"/>
      <geometry>
<box size=".4 .2 .1"/>
      </geometry>
    </visual>
    <inertial>
```

```
      <origin xyz="0 0 0.05" rpy="0 0 0"/>
      <mass value="1"/>
      <inertia
  ixx=".014" ixy="0" ixz="0"
  iyy=".016" iyz="0"
  izz=".004"/>
    </inertial>
  </link>

  <!--Joint 1 -->
  <joint name="joint1" type="continuous">
    <parent link="chassis"/>
    <child link="castor"/>
    <origin xyz="0.15 0 -0.05 " rpy="0 0 0"/>
    <axis xyz="1 0 1"/>
  </joint>

  <!-- Castor Wheel Link-->
  <link name="castor">
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
<sphere radius="0.05"/>
      </geometry>
    </collision>
    <visual>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
<sphere radius="0.05"/>
      </geometry>
      <material name="black"/>
    </visual>
    <inertial>
```

```
       <origin xyz="0 0 0" rpy="0 0 0"/>
      <mass value="2"/>
      <inertia
  ixx=".002" ixy="0" ixz="0"
  iyy=".002" iyz="0"
  izz=".002"/>
    </inertial>
  </link>

  <!--Joint 2 -->
  <joint name="joint2" type="continuous">
    <parent link="chassis"/>
    <child link="leftWheel"/>
    <origin xyz="-0.1 0.13 -.05" rpy="0 0 0"/>
    <axis xyz="0 1 0"/>
  </joint>

  <!--Left Wheel Link-->
  <link name="leftWheel">
    <collision>
      <origin xyz="0 0 0" rpy="-1.57 -1.57 0"/>
      <geometry>
<cylinder length="0.05" radius="0.05"/>
      </geometry>
    </collision>
    <visual>
      <origin xyz="0 0 0" rpy="-1.57 -1.57 0"/>
      <geometry>
<cylinder length="0.05" radius="0.05"/>
      </geometry>
    </visual>
    <inertial>
      <origin xyz="0 0 0" rpy="-1.57 -1.57 0 "/>
```

```
      <mass value="1"/>
      <inertia
  ixx=".008" ixy="0" ixz="0"
  iyy=".008" iyz="0"
  izz=".001"/>
    </inertial>
  </link>

  <!--Joint 3 -->
  <joint name="joint3" type="continuous">
    <parent link="chassis"/>
    <child link="rightWheel"/>
    <origin xyz="-0.1 -0.13 -.05" rpy="0 0 0"/>
    <axis xyz="0 1 0"/>
    <dynamics damping="0"/>
  </joint>

  <!--Right Wheel Link -->
  <link name="rightWheel">
    <collision>
      <origin xyz="0 0 0" rpy="-1.57 -1.57 0"/>
      <geometry>
<cylinder length="0.05" radius="0.05"/>
      </geometry>
    </collision>
    <visual>
      <origin xyz="0 0 0" rpy="-1.57 -1.57 0"/>
      <geometry>
<cylinder length="0.05" radius="0.05"/>
      </geometry>
    </visual>
    <inertial>
      <origin xyz="0 0 0" rpy="-1.57 -1.57 0"/>
```

```
    <mass value="1"/>
    <inertia
ixx=".008" ixy="0" ixz="0"
iyy=".008" iyz="0"
izz=".001"/>
  </inertial>
</link>

<!--Joint 4 -->
<joint name="joint4" type="fixed">
  <axis xyz="0 1 0" />
  <origin xyz="0 0 .15" rpy="0 0 0"/>
  <parent link="chassis"/>
  <child link="hokuyo_link"/>
</joint>

<!-- Hokuyo Laser Link-->
<link name="hokuyo_link">
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
  <box size="0.1 0.1 0.1"/>
    </geometry>
  </collision>
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
   <box size="0.1 0.1 0.1"/>
    </geometry>
  </visual>
  <inertial>
    <mass value="1e-5" />
    <origin xyz="0 0 0" rpy="0 0 0"/>
```

```
      <inertia ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0" izz="1e-6" />
    </inertial>
  </link>
</robot>
```

Now, to view the robot run the previous command again:

```
roslaunch robot robot.launch
```

The output will be similar to Figure 3.14. By right clicking on any of the joints, click on 'Apply Force/Torque', to get another menu, as seen in Figure 3.15. By changing the torque applied to joints along X, Y, Z axes, the robot will move.

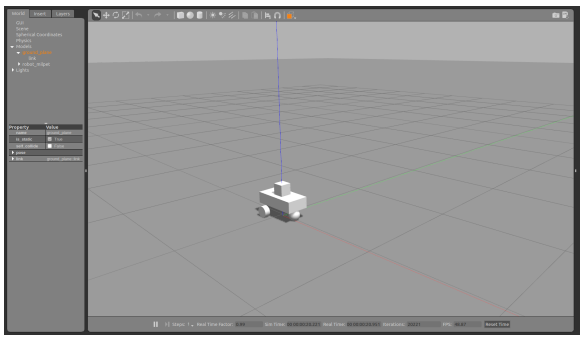


Figure 3.14: Robot created in Gazebo.

9. Writing the Gazebo file

   To give more capabilities to the robot, now make a 'robot.gazebo' file in /catkin_ws/src/robot/urdf. The following lines should be at the top of the file:

```
<?xml version="1.0"?>
<robot>
```

   If desired, add colors to the robot with the following snippet:

```
  <gazebo reference="chassis">
    <material>Gazebo/Red</material>
```

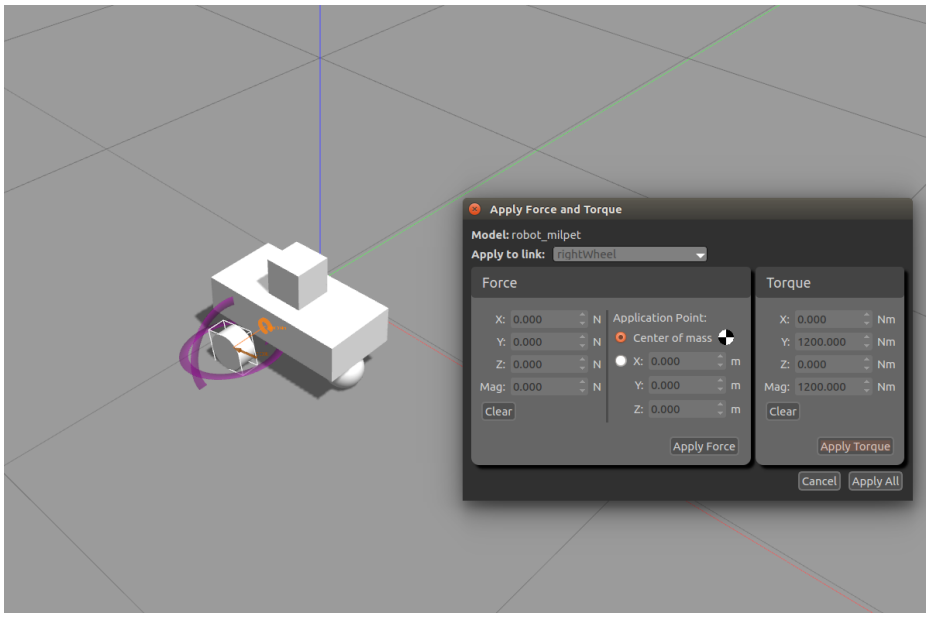


Figure 3.15: Gazebo menu to apply torque and force to different or all joints.

```
  </gazebo>
  <gazebo reference="leftWheel">
    <material>Gazebo/Black</material>
  </gazebo>
  <gazebo reference="rightWheel">
    <material>Gazebo/Black</material>
  </gazebo>
  <gazebo reference="hokuyo_link">
    <material>Gazebo/White</material>
  </gazebo>
  <gazebo reference="castor">
    <material>Gazebo/Black</material>
  </gazebo>
</robot>
```

To link the gazebo file to the xacro file, add the following line before the definition of links in 'robot.xacro':

```
<!-- Import all Gazebo-customization elements, including Gazebo colors -->
  <xacro:include filename="$(find robot_milpet)/urdf/robot.gazebo" />
```

10. Adding the Lidar

    To make the 'hokuyo_link' simulate the behavior of a Hokuyo Lidar sensor, add the following code in 'robot.gazebo':

```
<gazebo reference="hokuyo_link">
  <sensor type="gpu_ray" name="head_hokuyo_sensor">
    <pose>0 0 0 0 0 0</pose>
    <visualize>true</visualize>
    <update_rate>40</update_rate>
    <ray>
      <scan>
        <horizontal>
          <samples>1081</samples>
          <resolution>1</resolution>
          <min_angle>-2.3561</min_angle>
          <max_angle>2.3561</max_angle>
        </horizontal>
      </scan>
      <range>
        <min>0.10</min>
        <max>3.5</max>
        <resolution>0.01</resolution>
      </range>
    </ray>
    <plugin name="gazebo_ros_head_hokuyo_controller" filename="libgazebo_ros_gpu_laser.so
      <topicName>/robot/laser/scan</topicName>
      <frameName>hokuyo_link</frameName>
    </plugin>
  </sensor>
</gazebo>
```

    As seen, all the properties of the Lidar are configurable. The visualization tag when true, allows the Lidar trace to be visible in the simulation; to turn it

off switch it to false. The range, resolution, minimum and maximum angles can be set as well. Run the roslaunch file to see the changes. To see how the Lidar behaves, drag and drop objects from the 'Insert' tab into the simulation window. Figure 3.16 shows the bird's-eye-view of the robot, with a box in the Lidar's range.

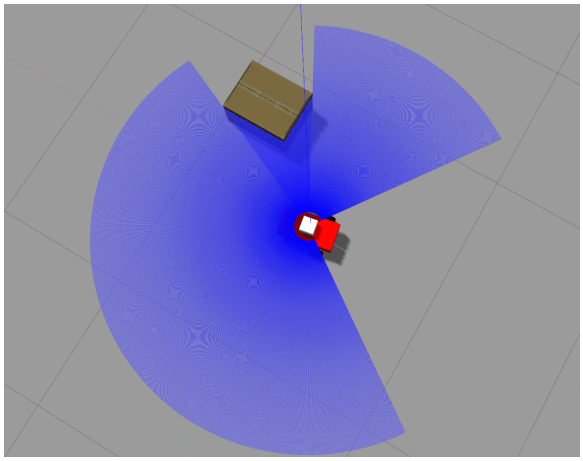


Figure 3.16: Top View of Robot with Visible Lidar Trace, with an Obstacle.

11. Controlling the robot

To move robot, we must add a motor to rotate the wheels. This too, is added in 'robot.gazebo'. The motor controller added here, is a differential drive motor controller. It is between the left and right wheels' joints. Paste the following code below the Lidar plugin:

```
<gazebo>
 <plugin name="differential_drive_controller" filename="libgazebo_ros_diff_drive.so">
    <alwaysOn>true</alwaysOn>
    <updateRate>1000</updateRate>
    <leftJoint>joint2</leftJoint>
    <rightJoint>joint3</rightJoint>
    <wheelSeparation>0.25</wheelSeparation>
    <wheelDiameter>0.1</wheelDiameter>
    <torque>5</torque>
```

```
    <commandTopic>cmd_vel</commandTopic>
    <legacyMode>false</legacyMode>
    <odometryTopic>odom</odometryTopic>
    <odometryFrame>odom</odometryFrame>
    <robotBaseFrame>base_footprint</robotBaseFrame>
  </plugin>
</gazebo>
```

After this is done, open a new terminal, and install the following package:

```
sudo apt-get install ros-kinetic-teleop-twist-keyboard
```

This package will publish twist messages to the *cmd_vel* topic that the motor controller is listening to. Start the simulation in the main terminal and run the following command in the second terminal:

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

Pressing keys, 'u,i,o,j,k,l,n,m,<', in the second terminal, will result in the robot moving. Figure 3.17 shows the controls.

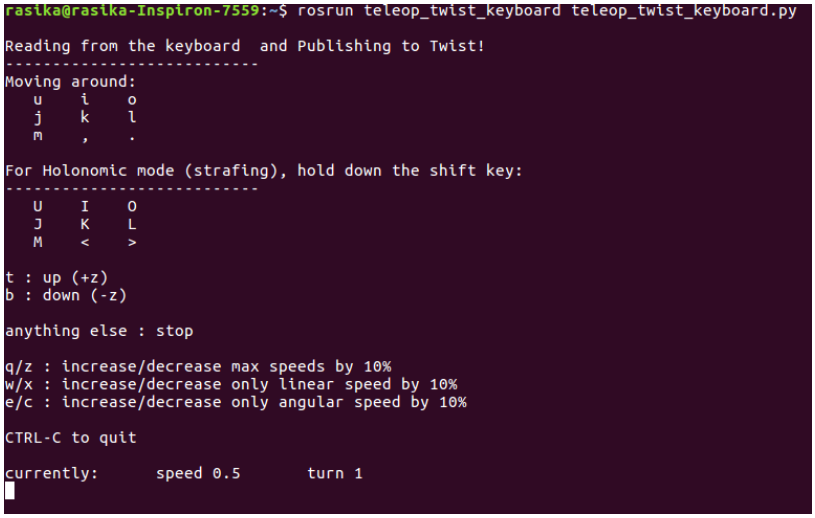


Figure 3.17: Keys to control the robot in Gazebo.

12. Viewing a ROS topic

    Start the simulation; start the 'teleop-twist' package in another terminal, with the command in the previous step, and open a new terminal to run:

    ```
    rostopic list
    ```

    The list of ongoing running topics will be printed on the terminal, as shown in Figure 3.18. */odom, /cmd_vel, /robot/laser/scan* have been setup after adding the Lidar and motorcontroller.

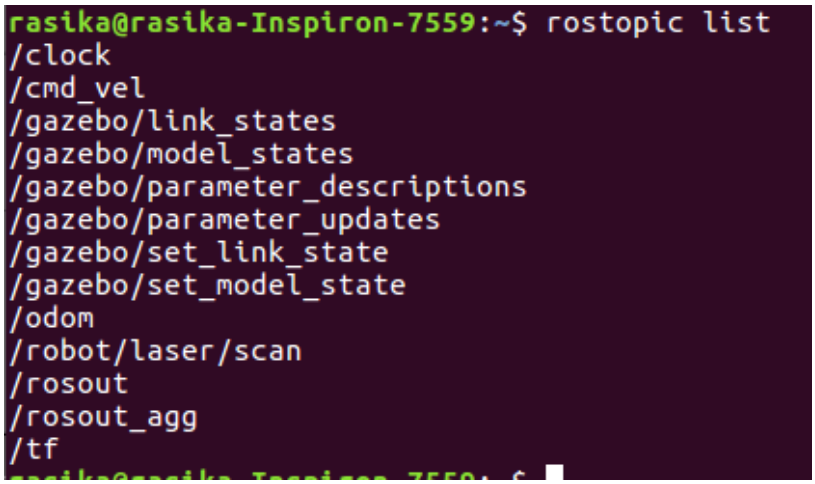


Figure 3.18: List of ROS topics running during Simulation.

Any topic's content can be viewed by running the command:

```
rostopic echo TOPIC_NAME
```

Run the following command and move the robot with keyboard:

```
rostopic echo cmd_vel
```

The linear and angular velocities sent to the robot are now printed on the screen, Figure 3.19.

Run the following command to see the Lidar's scan:

```
rasika@rasika-Inspiron-7559:~$ rostopic echo cmd_vel
WARNING: no messages received and simulated time is active.
Is /clock being published?
linear:
  x: 0.5
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

Figure 3.19: Twist messages published to topic *cmd_vel*.

```
rostopic echo /robot/laser/scan
```

Add obstacles to its view to see the scan change.

13. Viewing a ROS *rqt_graph* To see the nodes and topics running during the simulation, run the following command to get Figure 3.20:

```
rosrun rqt_graph rqt_graph
```

Figure 3.20 shows that the node, *teleop*, publishes messages to the *cmd_vel* topic and */gazebo*, i.e. the motor driver, subscribes to this topic to get its velocity commands.

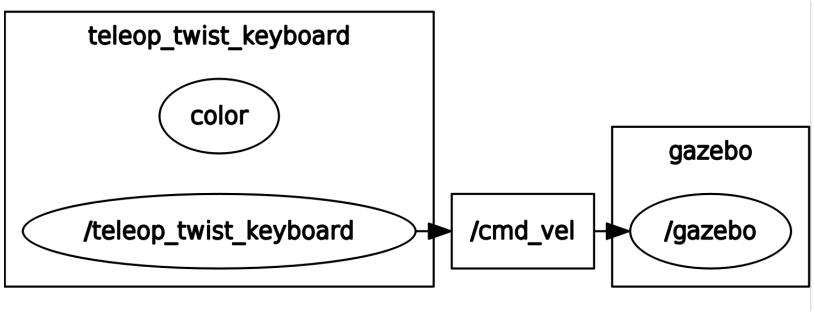


Figure 3.20: Nodes and Topics running during Simulation.

Chapter 4

Human Robot Interaction

Instead of controlling the wheelchair with the joystick, Milpet can be controlled via a smart-phone application, voice commands or the keyboard. The mobile application, which is available for download, could be useful to a caretaker of the patient.

Speech commands like “Milpet, move forward”, “Milpet, take me to the kitchen”, can be given. Milpet also has a debugging interface for developers. This interface allows developers to control Milpet by just clicking buttons. This interface can be expanded to a touch screen for patients with speech disabilities.

4.1 Mobile Application

Milpet can be manually controlled via Bluetooth, using a joystick mobile application. The applications are available for both Android, iOS. Figures 4.1, 4.2 show the mobile app interfaces in Android and iOS, respectively.

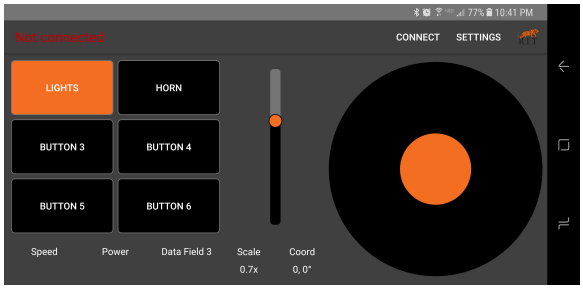


Figure 4.1: Android App Interface.

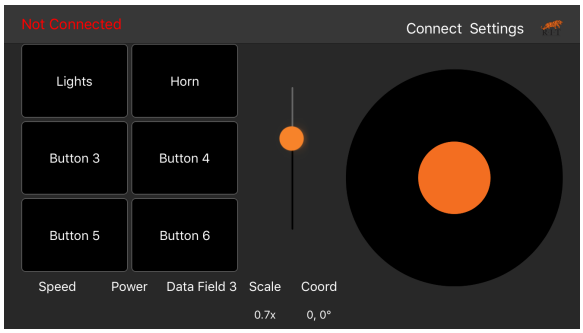


Figure 4.2: iOS App Interface.

4.2 Speech

One of the main ways to interface with Milpet, is through speech. Voice commands are given to Milpet over the connected microphone and converted to text by a speech-to-text package. Every command, barring 'stop', must start with 'Milpet', so that Milpet knows this is a command to interpret. Speech recognition requires a dictionary of probable words, phonemes, a language model and an acoustic model to work.

A dictionary contains the vocabulary of the recognizer, with each word's phonemes; the language model maps different word combinations to a probability. The acoustic model is not needed, but helps in recognizing phonemes, and is trained on a speaker's voice.

Python provides a variety of speech recognition engines and packages in its SpeechRecognition package [31]. In this, Google's API is available too; all recognition happens online and does not require manual set up of the pre-requisite models and dictionary. Unfortunately, this speech-to-text conversion requires constant Internet connection, and so it was deemed not ideal for Milpet.

CMU PocketSphinx [32] is a package that can be used for offline speech recognition. Here, the dictionary and models needed to be set up. The steps for creating the models and dictionary using PocketSphinx are listed below:

1. List all the words that would be used in a .txt file.

2. Upload this file to [33].

3. Download the .lm and .dict file.

4. For the acoustic model record each word in the dictionary; name each recording with the recorded word.

5. Follow the steps [34] to create the acoustic model.

The PocketSphinx model was set up and then tested. However, the recognition speed was not fast enough. Examining the source code, it was realized that since the Python SpeechRecognition was a wrapper for interacting with a variety of speech recognition engines with a single constructor, it initialized all engines, whether online or offline, with the same function. After initializing, whenever a spoken phrase was converted to text, PocketSphinx loaded in the dictionary, language model and acoustic model, to give its output. It was recognized that this loading of models, every time a sound is captured, was the bottle-neck in its speed. Since PocketSphinx was to be used exclusively, these models were loaded during creation of the PocketSphinx object, instead of loading them each time a phrase needs to be converted to text. Doing this, the recognition speed was boosted.

Table 4.1 shows a comparison between Google API, original PocketSphinx and the modified PocketSphnix. To cancel out ambient noise, parameters like damping, pause threshold and phrase threshold were set in the PocketSphinx code.

Table 4.1: Speech Recognition speed (in seconds).

	Google API	PocketSphinx	PocketSphinx (modified)
Speaker			
Phrase:	**Milpet**		
Speaker1	N/A	1.249	0.175
Speaker2	N/A	1.21	0.1175
Speaker3	N/A	1.246	0.1589
Phrase:	**Milpet, take left.**		
Speaker1	1.4813	1.33	0.193
Speaker2	1.4705	1.18	0.202
Speaker3	1.64	1.178	0.32
Phrase:	**Stop**		
Speaker1	1.39	1.076	0.1373
Speaker2	1.2908	1.25	0.1340
Speaker3	1.115	1.518	0.1139
Phrase:	**Milpet, go forward and take right**		
Speaker1	1.944	1.21	0.317
Speaker2	1.08	1.33	0.3723
Speaker3	1.55	1.518	0.4380
Average speed:	1.4401	1.274	0.2232

4.3 Voice Feedback

Milpet makes use of a package provided by ROS, called 'sound_play'. This package, takes in a sentence and converts it to speech. The accent that the speech is outputted with, is the default accent set in the Festival Speech Synthesis System (Festival) package, installed with Ubuntu.

Any feedback message / error message / interaction message that Milpet wants to give the user, is published as a String and priority to the 'dialogue' topic. The voice_out topic, then decides which message to send to the sound_play node, considering the priority. Figure 4.3 shows the ROS diagram depicting the nodes involved.

To make Milpet user friendly, whenever a command is spoken to Milpet, it can

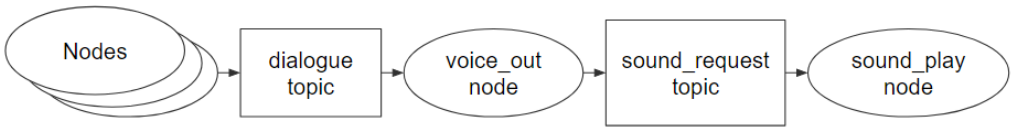


Figure 4.3: Voice out text-to-speech ROS diagram.

reply with a confirmation message. An example scenario, is described in Table 4.2, between a user and Milpet. The type of message published to the 'dialogues' topic, are shown in brackets against the user; the type of message spoken by Milpet is shown in brackets against Milpet.

Table 4.2: Dialogue Scenario with Milpet.

Speaker	Dialogue/ Situation	Message type
User:	Milpet, move forward.	(Basic, 3)
Milpet:	Going forward.	(Feedback)
User:	Stop	(Emergency,1)
Milpet:	Stopping	(Confirmation)
User:	Milpet, take me to Room 237	(Navigation,2)
Milpet:	Planning a path... starting navigation	(Feedback)
—	(Navigates to goal, avoiding obstacles)	—
Milpet:	Goal reached	(Feedback)
User:	Move forward	(Basic,3)
—	(Milpet facing corner)	—
Milpet:	Nowhere to go. Please help me navigate.	(Error)

Chapter 5

Obstacle Avoidance

Avoiding obstacles is an essential task in any autonomous agent. Milpet aims at providing obstacle avoidance capabilities, so that the user can be at ease while navigating environments. Milpet runs with obstacle avoidance when following a path or when wandering freely.

Obstacle avoidance was implemented in stages to reach the final algorithm. After each stage, the robot was tested in either simulation and/or in practice. Observations and results are discussed in the Results section. Each stage builds upon the other, improves functionality or serves as a unit that can be swapped in or out, thereby changing the behavior of the robot.

5.1 System Integration

The Obstacle avoidance algorithm lies at the heart of Milpet's navigation system. Figure 5.1 shows the ROS nodes, topics, etc. involved with the obstacle avoidance node, 'avoider'.

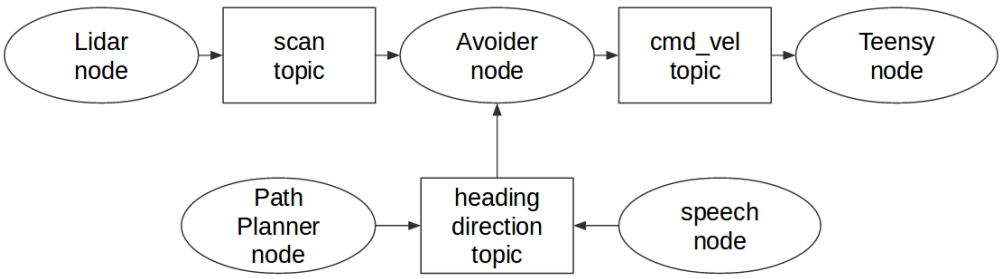


Figure 5.1: Avoider node in ROS.

5.2 Algorithm Development

5.2.1 Stage 1: Blocks Creation

Figure 5.2 shows the Lidar scan view with a threshold and the robot, when no obstacle is present. The shaded area / light blue area shows areas that are free. Segregating the environment into safe and unsafe zones is the first step. Safe zones, $scan_{safe}$, are calculated by thresholding the lidar scan, s; these zones are termed as *blocks*. Blocks through which the robot can fit, are kept, i.e. the block width must be greater than two times the robot radius ($> 2 * robot_{radius}$). In all the figures, the blocks i.e. shaded area, are depicted, and not the entire 270 ° range, unless the entire range is free, as is in Figure 5.2. Figure 5.3 shows two blocks before filtering out smaller block. The following properties of blocks are calculated:

- center orientation of block
- (x,y) Cartesian coordinates and indices i_b, i_e of the start and end of block
- block width
- average Lidar distance reading
- candidate orientation w.r.t. world

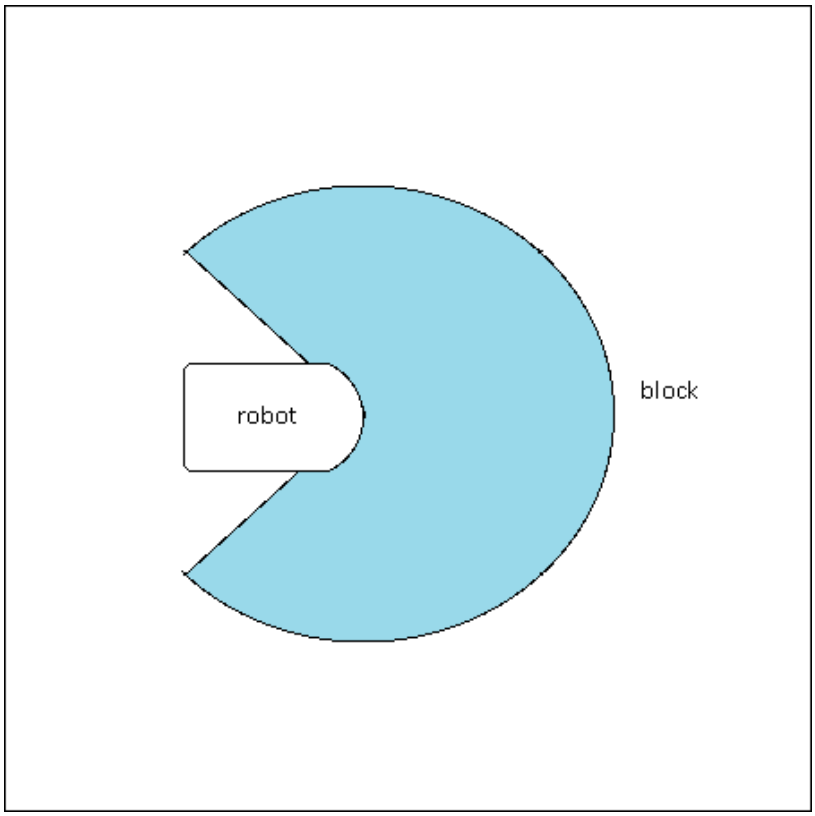


Figure 5.2: Robot's Lidar scan view up to a threshold.

The blocks whose center orientation, $orientation_{block\_center}$, is closest to the current orientation, $orientation_{current}$, is chosen and set as the required orientation, $orientation_{required}$. Picking the center orientation of each block, ensures safety, such that the robot would never hit an obstacle. The robot turns to match $orientation_{required}$ and moves forward. Figure 5.4 shows the different orientation terms associated with the obstacle avoidance algorithm.

5.2.1.1 Simulated Lidar interfacing:

Interfacing the Lidar with the robot is described in Chapter2 in the Gazebo and ROS tutorial.

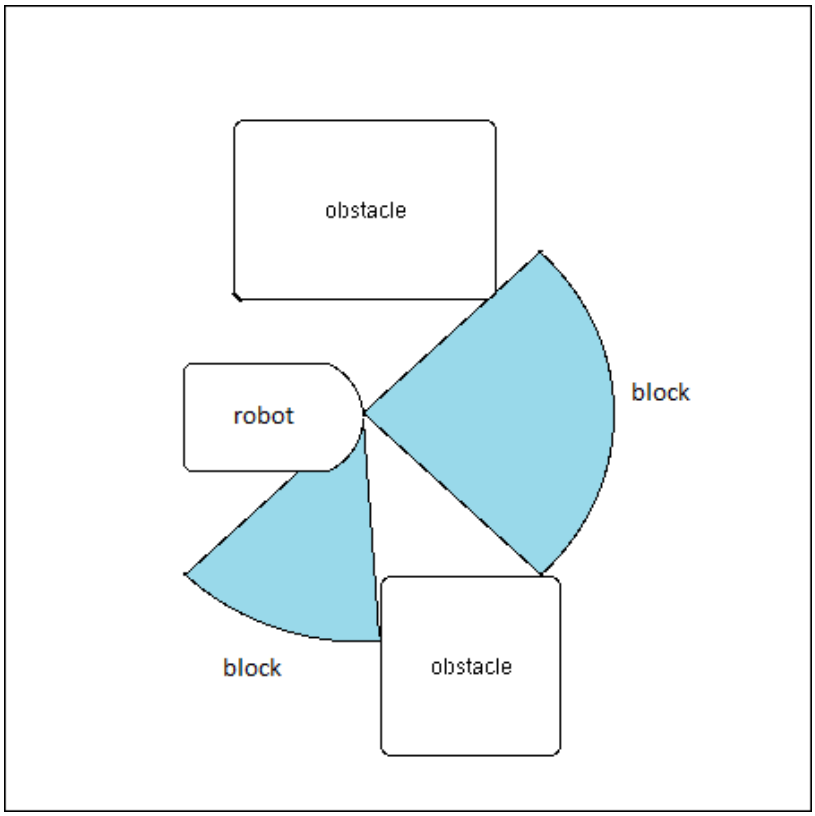


Figure 5.3: Robot's Lidar scan view up to a threshold.

5.2.1.2 Calculating Orientations:

In the simulation, the robot can be made to publish to an '/odom' topic, by setting up the required parameter in the differential drive controller. The required settings are present in the code in the tutorial from Chapter 2. The message that Gazebo publishes to the '/odom' topic is a Quaternion message, $Q(x, y, z, w)$. A Quaternion message is four dimensional message that represent the three dimensional Euler angles, roll, pitch and yaw. Equation (5.1) gives the conversion of Q to the pose (yaw), ϕ_{world}, of the robot from its initialization.

$$\phi_{world} = atan2(2 * w * z, 1 - 2 * z * z) \tag{5.1}$$

On the robot, the encoders publish a Pose2D message, x, y, z, giving the angle z in radians $\in [-2\pi, 2\pi)$. This is converted to $\phi_{world} \in [0, 360)$. Equation (5.2) gives the

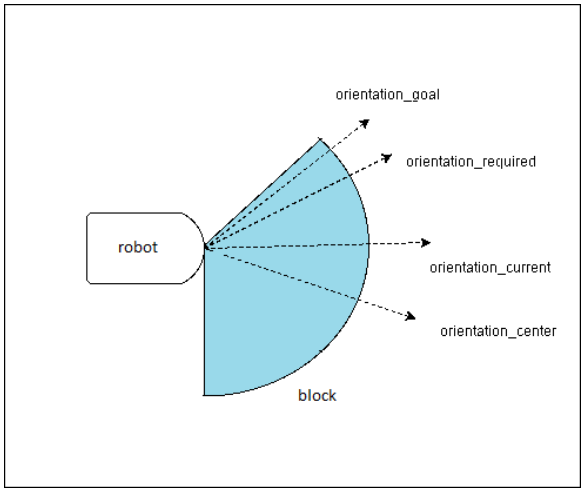


Figure 5.4: Different *orientation* terms associated with Obstacle Avoidance.

$orientation_{current}$. ϕ_{world} and $orientation_{current}$ are the same (5.3), while the former emphasis on the frame and the later emphasis on the real time aspect.

$$\phi_{world} = \begin{cases} radianToDegree(z) + 360, & \text{if } radianToDegree(z) < 0 \\ radianToDegree(z), & \text{otherwise} \end{cases} \tag{5.2}$$

$$orientation_{current} \iff \phi_{world} \tag{5.3}$$

5.2.1.3 Switching between Frames:

There are two frames, F_{robot} and F_{world}, describing the angle of the robot. F_{world} describes the angle w.r.t. the robot's initial position, $0 \leq \forall\ n \in F_{world} < 360$. F_{robot} describes the angle w.r.t. the robot, $-135 \leq \forall\ n \in F_{robot} < 135$; the angle right in front of the robot is 0°.

The block calculation and candidate orientations, are in the F_{robot}, while $orientation_{current}$ is in F_{world}. Thus, a need to convert from one frame to the other is needed. Milpet's Hokuyo Lidar's range, l_{range}, is 270°and its resolution, l_{res} is 0.25°. Equation (5.4) describes how angle ϕ_{robot} is calculated from the Lidar scan. Equation (5.2) described how ϕ_{world} is calculated. Equation (5.5) describes the θ_{robot} is converted to θ_{world},

while the ϕ_{world} is current orientation and θ_{robot} is a candidate direction for a block.

$$\phi_{robot} = -lidar_{range}/2 + idx * lidar_{res} \tag{5.4}$$

$$\theta_{world} = \begin{cases} \phi_{world} - \theta_{robot} - 360, & \text{if } \phi_{world} - \theta_{robot} > 360 \\ \phi_{world} - \theta_{robot} + 360, & \text{if } \phi_{world} - \theta_{robot} < 0 \\ \phi_{world} - \theta_{robot}, & \text{otherwise} \end{cases} \tag{5.5}$$

5.2.2 Stage 2: Goal Heading Direction

Instead of picking a direction closest to $orientation_{current}$, the block's whose $orientation_{center}$ is closest to the goal direction, $orientation_{goal}$, is set as $orientation_{required}$.

5.2.3 Stage 3: Block cut-off

Choosing the direction closest to $orientation_{goal}$, worked well. However, when the robot would be in a large unobstructed space, a discrepancy was noticed: the robot would not turn in the direction of the goal. This was caused by only considering the center orientation of each block. Figure 5.5 describes this scenario.

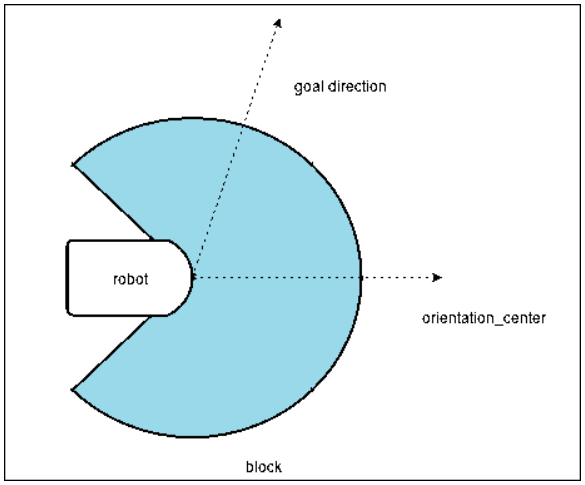


Figure 5.5: Issue with Stage1.

To solve this issue, the concepts of multiple candidate orientations per block, and block cut-off, $block_{cutoff}$ were introduced. Instead of considering only one direction per block, every orientation within the range shown in (eq cutoff) is considered. To avoid getting too close to an obstacle, the $block_{cutoff}$ parameter was introduced. In other words, the entire block's orientations, excluding a minimal range 'cut-off' from each end of the block, is considered for the calculation of the block's candidate direction. After this, all blocks' candidate directions are considered to set the $orientation_{required}$. Figure 5.6 depicts a representation of $block_{cutoff}$. Figure 5.7 shows how the $orientation_{required}$ is chosen.

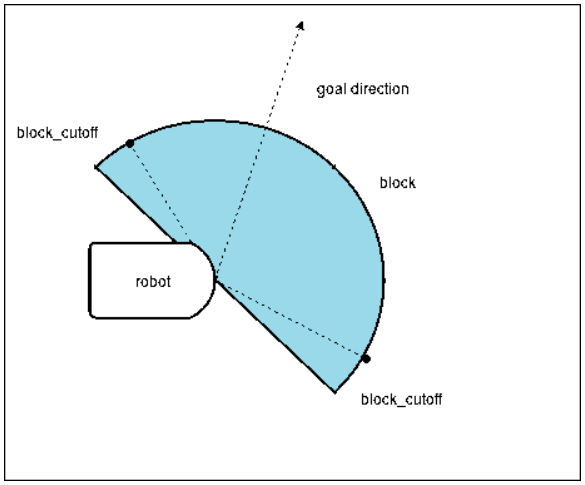


Figure 5.6: Block with $block_{cutoff}$.

5.2.4 Stage 3: Behavior Modes

Following the implementation of Stage 2, behavior modes were introduced. The criteria of how the candidate orientations are evaluated can be swapped in and out; this results in the robot exhibiting a different behavior. Stage 2 describes a 'goal based behavior'. Another behavior is the 'distance based behavior'. Table 5.1 describes different behavior modes.

These are termed as 'behavior modes', since the behavior exhibited by the robot changes based on different scenarios. For example, the 'heuristic based behavior

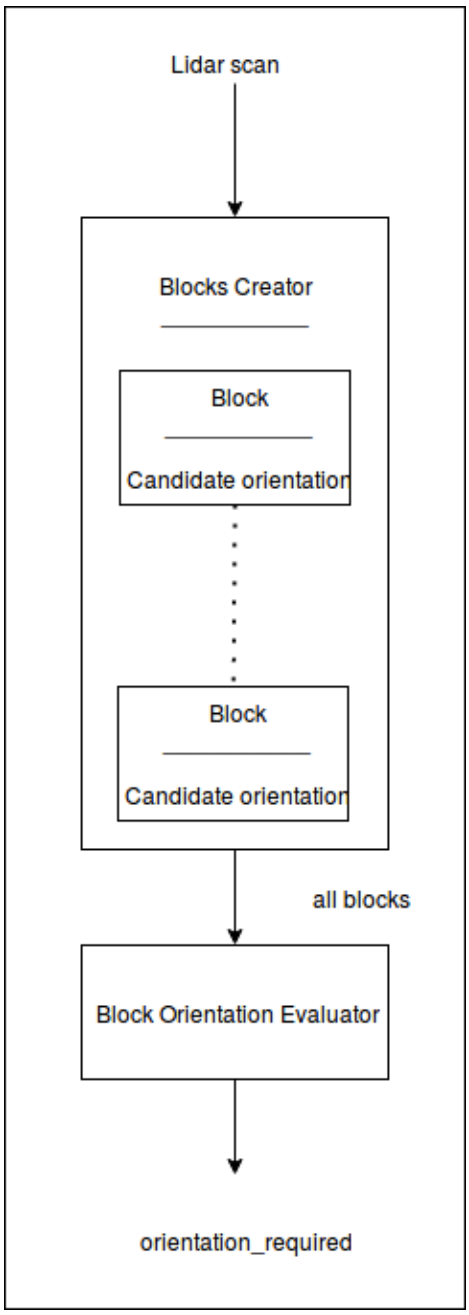


Figure 5.7: Setting $orientation_{required}$.

mode', is a wandering kind of behavior; while the 'goal based behavior mode' is very aggressive in choosing directions towards the goal, which is suitable for directed commands, like, "Milpet, go forward".

Table 5.1: Obstacle Avoidance Behavior Modes.

Behavior Mode	Direction Setting	Resulting Behavior
Distance mode	Chooses block with farthest average distance	Begins to circle around an area, since the clearest area is behind the robot.
Goal mode	Sets direction closest to $orientation_{goal}$	Aggressive, gets stuck in local minima eg. dead-ends, corners.
Heuristic mode	Chooses orientation closest to a weighted sum of previous and goal orientation	Has momentum; which helps in getting out of local minima; can drift away from goal.
Path planning mode	A cross between the heuristic mode and goal mode: with more boas towards the goal mode	Suitable for integration with the entire navigation system.

5.2.4.1 Heuristic based behavior mode:

The heuristic mode, selects a candidate orientation based off of a weighted sum of the previous, current and required orientations. Equation (5.6) describes the heuristic orientation. The heuristic mode also sets up a momentum (5.7), and uses this $orientation_{momentum}$, instead of the $orientation_{goal}$ in the block orientation evaluator, Figure 5.7.

$$orientation_{heuristic} = 0.6{*}orientation_{goal}{+}0.1{*}orientation_{current}{+}0.3{*}orientation_{previous} \tag{5.6}$$

$$orientation_{momentum} = 0.6 * orientation_{required} + 0.4 * orientation_{previous} \tag{5.7}$$

5.2.5 Stage 4: Smoothing

For turning the robot smoothly the linear and angular velocities, v, w, must be controlled appropriately. To do so, a PI inspired feedback loop has been set up for controlling the turning motion of the robot when, $angleDifference$ ($orientation_{current}$, $orientation_{required} < k$); k is generally 30°. Figure 5.17, shows the breakdown of motion, in the main thread. $angleDifference()$ is described in Algorithm 5. The PI inspired feedback loop, is shown in Algorithm 4. Depending on the error between the required orientation and current orientation, w increases/decreases accordingly. Using Algorithm 4 to set w, the robot is able to make smooth turns and drive straight. α, β are parameters analogous to the coefficients of proportional and integral terms, K_p, K_i.

Algorithm 4: Calculating w

```
define angularVelocity:
difference=angleDifference(orientation_current, orientation_required);
error = β * difference ;
error = min (error, w_max/4);
w_previous = w;
w = min (α * difference + error, w_max);
```

Algorithm 5: Circular Angle Difference Function

```
define angleDifference(x1, x2):
if x1 < x2 then
    if abs(x1 − x2) ≤abs(x1 − x2 + 360) then
        return abs(x1 − x2);
    else
        return abs(x1 − x2+360);
else
    if abs(x1 − x2) ≥ abs(x1 − x2 − 360) then
        return abs(x1 − x2-360);
    else
        return abs(x1 − x2);
```

5.2.6 Stage 5: Lidar Filtering

The Hokuyo UST-10LX is a 270° single channel lidar. Each scan,s, is a 1081 long vector. Figure 5.8 illustrates the Lidar's properties. In simulation, the Lidar didn't

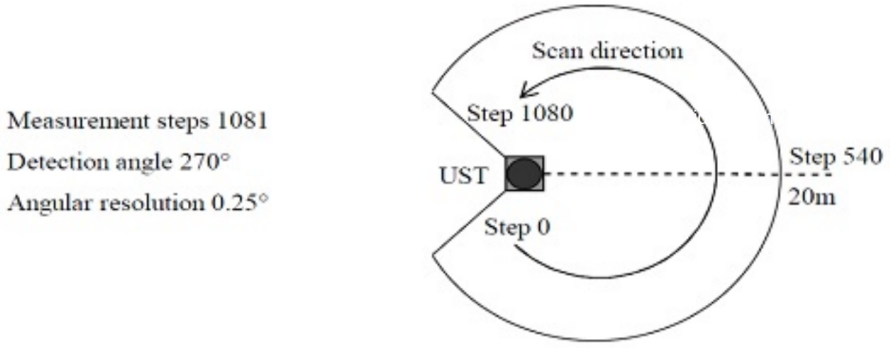


Figure 5.8: Block with $block_{cutoff}$.

have any noise. But in practice the Lidar does have noise. Noise filtering has to be carried out before passing the scan to the obstacle avoidance node. Figure 5.11 describes this. Figures 5.9, 5.10 show ten lidar scans in the same location, before and after filtering. Algorithm 6 describes the lidar filtering method. s is the scan read in from the Lidar and $scan_{count}$ describes the number of scans to consider for filtering. The $remove_{spikes}()$ function filters out the noise when the distance returned by the scan is above 65 in the vector. This value represents an error in recording the scan. The $interpolate()$ function removes the most of the interference caused due to the poles supporting Milpet's desk.

Using (5.8), (5.9)polar data is mapped to the Cartesian system and plotted in Figure 5.12. The green circle shows the area occupied by the robot; the red circle is at a radius of 2 meters. The scan in the figure is taken with a board blocking it in the front, which is visible in the figure.

$$x = s[i] * cos(\phi_{robot} * \pi/180)) \tag{5.8}$$

Algorithm 6: Lidar Filtering Algorithm.

```
1  j=0;
2  while Lidar is on do
3      read in s;
4      if j<scan_count then
5          s = remove_spikes(s);
6          S[j,:]=s;
7          j++;
8      else
9          s=interpolate(median_filter(S),threshold);
10         publish(s);
11         j=0;
```

$$y = s[i] * sin(\phi_{robot} * \pi/180)) \tag{5.9}$$

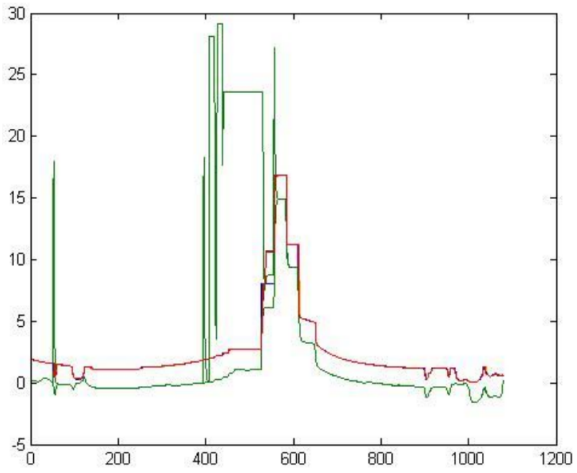


Figure 5.9: Ten raw Lidar scans in the same position.

5.2.6.1 Lidar Interfacing:

Steps for connecting the Lidar to Windows:

1. The static IP of the computer must be set to 192.168.0.XX, where XX is anything but 10.

2. The netmask is 255.255.255.0

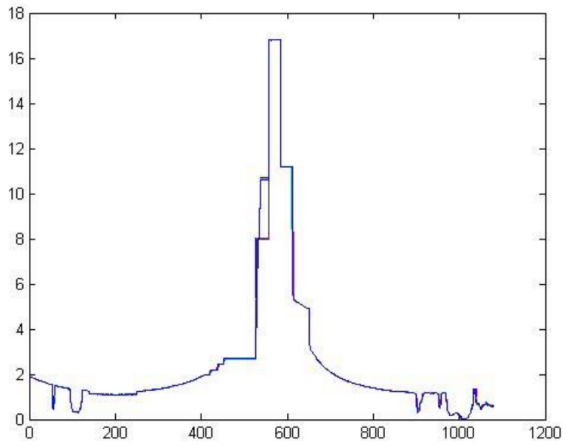


Figure 5.10: Ten filtered Lidar scans in the same position.

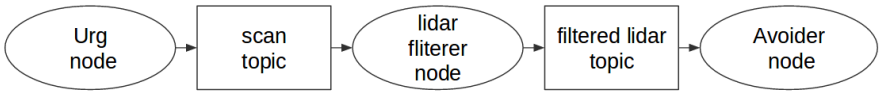


Figure 5.11: ROS nodes and topics for filtering the Lidar.

3. The gateway is 192.168.0.1

4. To change this address run the following as administrator:

```
netsh interface ipv4 set address name=Ethernet static 192.168.0.10

255.255.255.0 192.168.0.1
```

5. Run

```
ipconfig
```

   to cross check changes.

6. Download urgbenri from sourceforge.net and run.

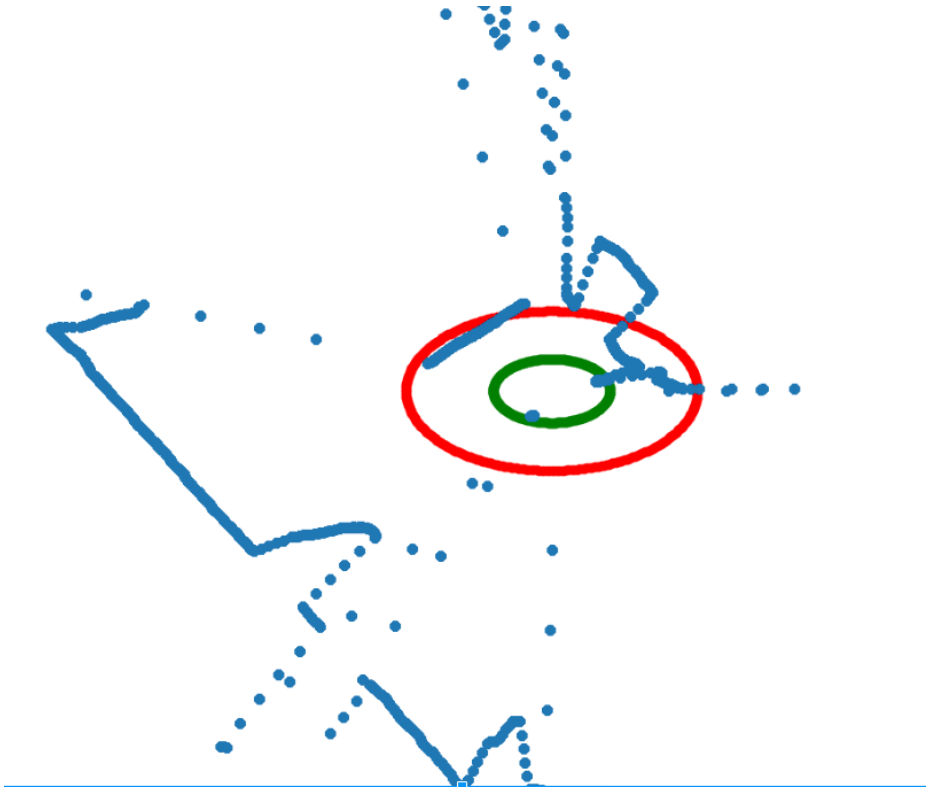


Figure 5.12: Lidar scan mapped to Cartesian system, using (5.8), (5.9).

7. Set the Ethernet IP to 192.168.0.10 (IP of the device) in the field provided.

Steps for connecting the Lidar to Ubuntu:

1. Connect the Lidar to the computer via ethernet and let it try to establish a connection.

2. Click on the network symbol i.e. WiFi symbol on top right.

3. Go to edit connections. Select the wired connection just made.

4. Click on the IPV4 tab

5. set it to manual

6. Set the static IP address of the computer as in Figure 5.14.

7. Plug the ethernet back in.

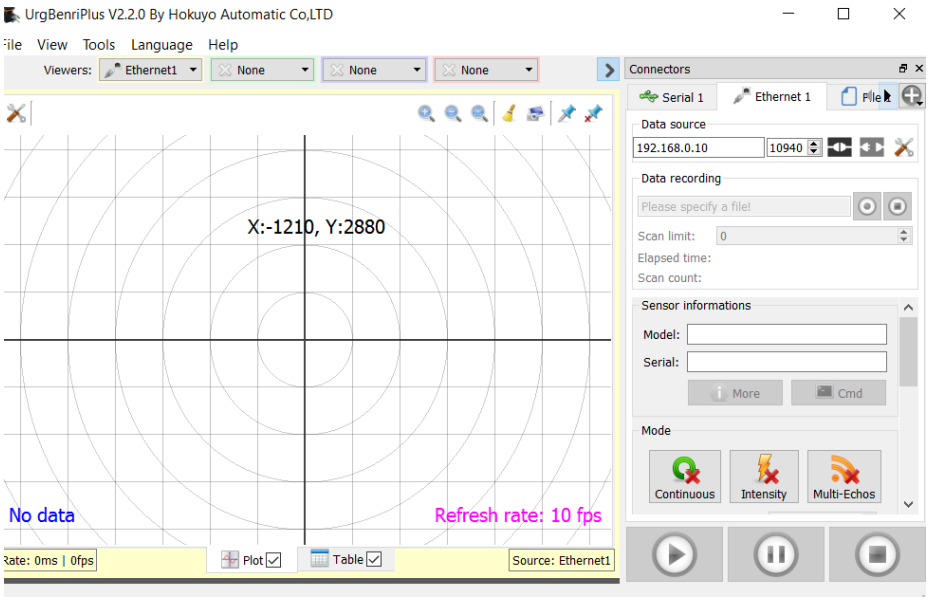


Figure 5.13: Interface for viewing Lidar data on Windows.

8. The connection should get established.

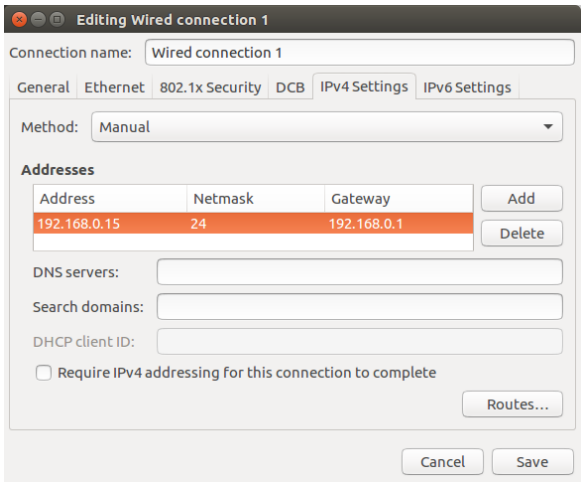


Figure 5.14: Setting up IP address for Hokuyo Lidar on Ubuntu.

To check the connection, make a launch file, lidarTest.launch, in a ROS package (refer to the ROS tutorial in Chapter 3 for details). Paste the below in it:

```
<launch>
<node pkg="urg_node" type="urg_node" name="urg_node">
    <param name="ip_address" value="192.168.0.10"/>
    <param name="frame_id" value="scan"/>
```

```
    <param name="calibrate_time" type="bool" value="true"/>
    <param name="publish_intensity" type="bool" value="false"/>
    <param name="skip" value="0" />
    </node>
</launch>
```

To run the launch file, open a terminal and run:

```
roslaunch packageName lidarTest.launch
```

5.2.7 Stage 6: Improving safety

Despite the $block_{cutoff}$ parameter, the robot would get very close to walls, which is unsafe and undesirable. Another problem area was exposed during testing: The Lidar, Milpet uses for testing, is mounted underneath its desk, thus, while navigating round corners, the Lidar sees free area, before the robot has gone fully around the corner, resulting in the robot grazing or driving into corners. Figure 5.15 describes this scenario.

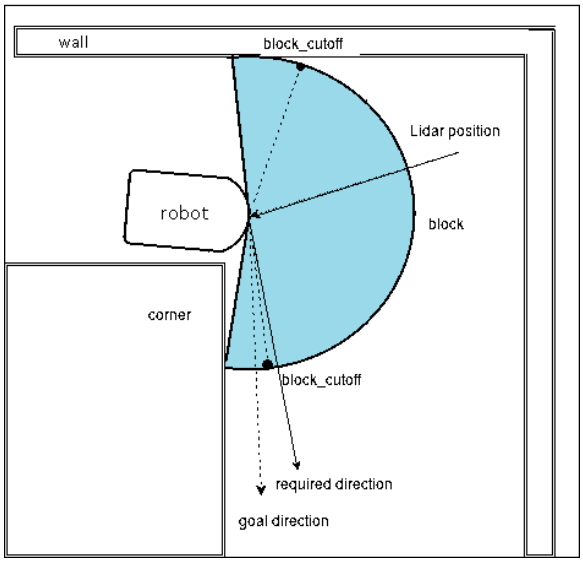


Figure 5.15: Problem Example: Milpet turning around a corner.

To overcome this, the enlargement angle concept from VFH+ [7] was implemented

here. Using the enlargement angle, obstacle at edges of the block are bloated up, by a factor depending on the robot size. As the robot get closer to the obstacle, the enlargement angle increases.

Due to this, the calculation of a block size is affected. Figure 5.16 shows how the representation of the block with the safety bubble.

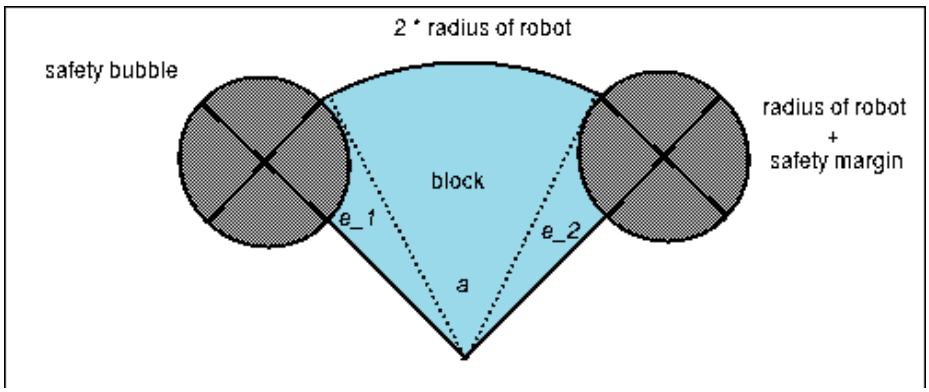


Figure 5.16: Block with Safety Bubbles.

5.2.8 Stage 7: Centering the robot

Implementing Stage 6, the robot exhibits a wall following behavior, this behavior is not as safe as driving down the center of the hallway, and is unfavorable from a system integration stand-point. Milpet's localization module has been trained with data collected in the center of hallways, having Milpet driving close to walls, therein affects the localization module's performance. To resolve this, a bias was added after the candidate direction for the block is chosen. This bias shifts the candidate direction in the direction of block's $orientation_{center}$.

5.3 Final Algorithm Overview

Figure 5.17 has a flowchart of the obstacle avoidance algorithm. There are three main threads,Thread 1 keeps track of $orientation_{required}$ and $orientation_{current}$, along with sending velocity commands to the Teensy. Thread 2 reads and interprets the Lidar

scans in block terms to find the optimal orientation for the robot. Thread 3 updates the $orientation_current$ and Thread 4 connects the obstacle avoidance system to the path planning system, by updating the $orientation_goal$.

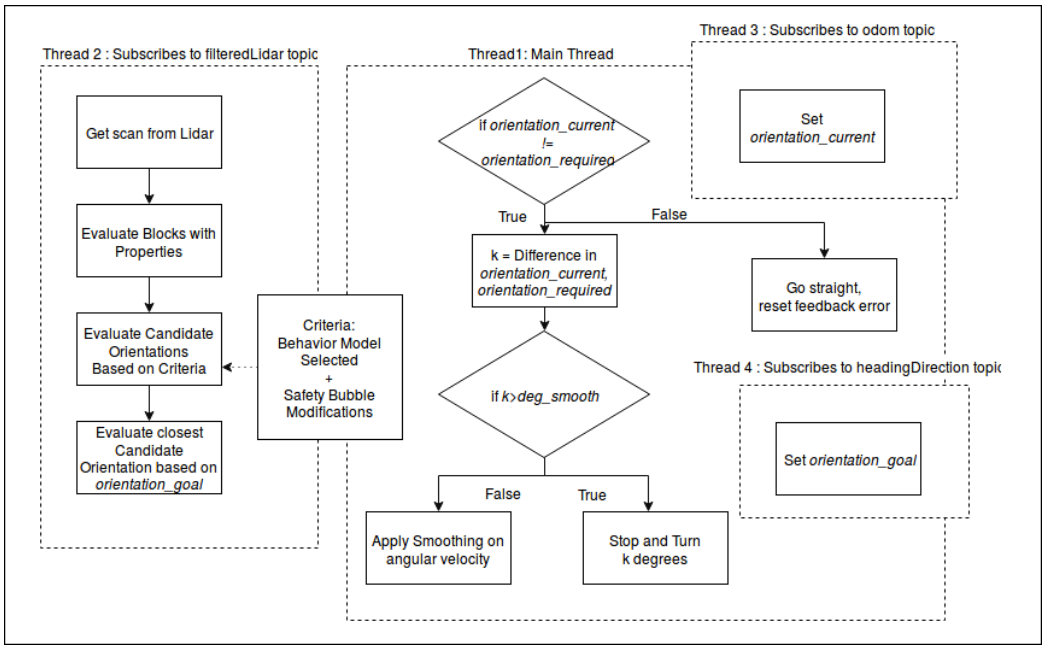


Figure 5.17: Flowchart of Obstacle Avoidance.

5.4 Modularity

The obstacle avoidance code has been designed in a modular way. Figure 5.18 shows the parts of the obstacle avoidance system that can be swapped or modified to change the overall function of the robot. As seen in the figure, the Lidar sensor properties like resolution and range can be changed to suit any single channel Lidar. Robot properties like robot radius, safety margin, minimum passage width, etc. can be set according the robot's requirements. The debugging switch can be enabled for debugging details by developers. By enabling the path planning switch, the obstacle avoidance system is connected to the path planning system. Behavior modes and centering can be set to get a specific behavior.

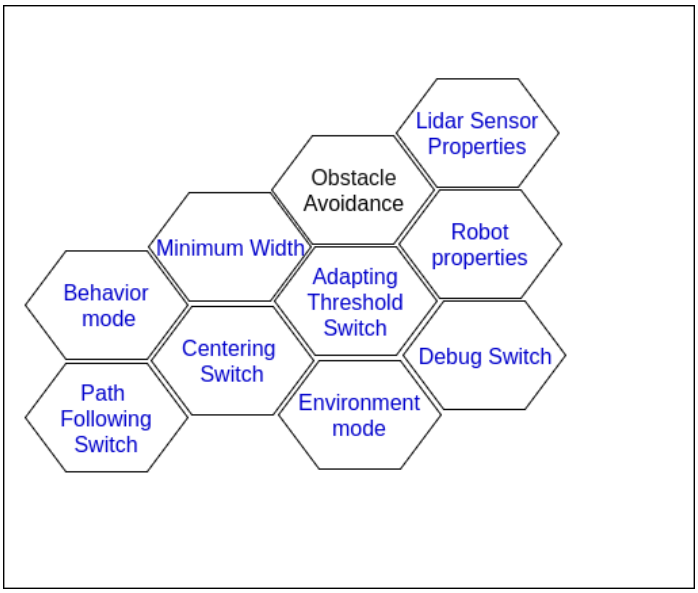


Figure 5.18: Obstacle Avoidance System Components.

Chapter 6

Path Planning

6.1 Algorithm

Milpet uses D* Lite for path planning as described in the background section. D* Lite is suitable in applications where a path may change or get obstructed while traveling towards it, since it is suited for dynamic replanning. The next section steps through an example of getting a path via D* lite.

6.2 Mapping

Milpet makes use of a two dimensional occupancy grid to represent its environment. Figure 8.1 shows the environment Milpet is localized to. An occupancy grid is an array, where each cell represents a square area of the real world. Cells that are black, denote areas the robot cannot go, whereas white cells are free space. Figure 6.1 shows a map of a hallway that Milpet is to navigate. In Figure 6.1 each cell is a four square inch area. This is the resolution of the localization system. This map is used for planning paths for Milpet.

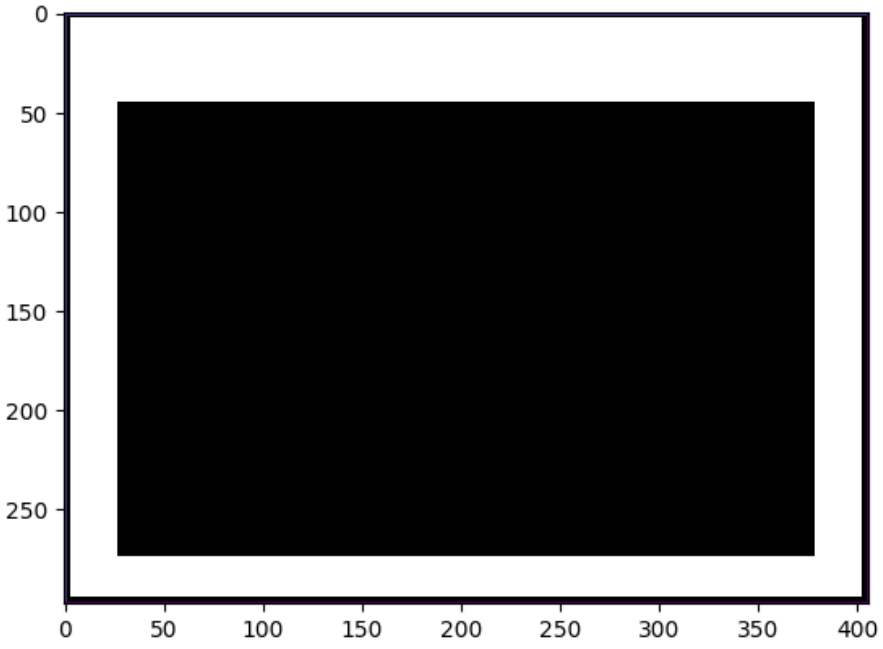


Figure 6.1: Occupancy Grid that Milpet navigates.

6.3 Waypoints

If the entire path is taken into consideration, the robot aims at strictly following the path. This poses the complicated task of replanning, whenever a new obstacle not on the map, obstructs Milpet's path.

Path planning is an expensive task, hence, calling it repeatedly is computationally expensive. If the replanned path is not returned in time, the robot may collide into an obstacle, travel in the wrong direction for a period of time, or unnecessarily stop to deal with the change in the environment.

Thus, lowering the number of times replanning is carried out, is to the advantage of the system. Replanning when there are actual changes to the environment, like a path being blocked off or too crowded to go through, are times when replanning ought to be carried out, and not for every new obstacle that blocks its path. This is

the job of the obstacle avoidance system.

To give this kind of freedom to the obstacle avoidance system, but still make sure the robot goes along the path, the path is broken down into waypoints.

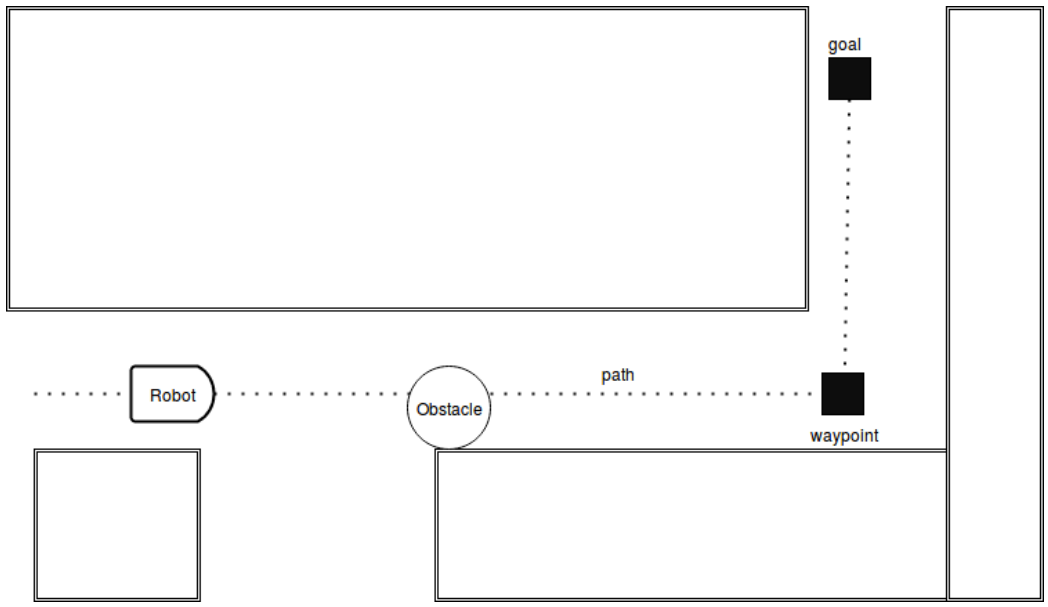


Figure 6.2: Waypoint Example Scenario: Waypoints only at angle changes in path.

Having a waypoint whenever the angle changes along the path, could result in the robot drifting away or going into a room when it travels. Figures 6.2 shows a scenario, where having waypoints at every angle change could be unfavorable. The robot may drift off and it could take longer for it to correct itself and come back on the path. Instead, waypoints that are equidistant from each other is more useful; each waypoint then serves as a check point for the path planning system. Also, determining major changes in angle path can be an expensive task. On the other hand, calculating waypoints, W, at intervals of w, is simpler. Also, the required information for getting equidistant waypoints, is already present in the path planning's cost map and can be used directly. Algorithm 7 describes the waypoint calculation.

6.4 Clusters

The waypoints represent 4 square inch areas, whereas the robot covers an area of approximately 3 square feet. When the robot reaches an area near the waypoint, it essentially, has reached that waypoint. Thus, the area in the vicinity of each waypoint, also act as intermediate goal points. This leads to the concept of waypoint clusters.

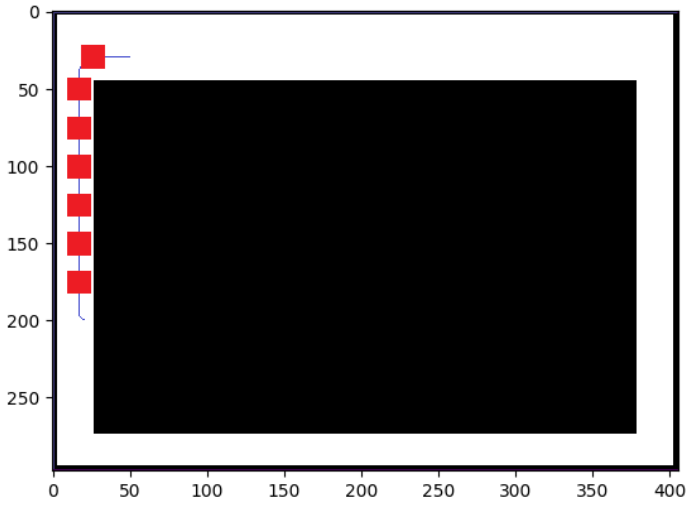


Figure 6.3: Path with Waypoint clusters, and square shaped dilating structire.

The points in the locality of the waypoint make up the respective waypoint's waypoint cluster. Waypoints can thus be calculated by applying a morphological operation called dilation. Dilation bloats up the single cell to the dilating structure. Figure 6.3 shows the path (in green), and its waypoint clusters (in yellow). The dilating structure, d_s=$d_{s1} \times d_{s2}$, can be changed to cover a different spread of area. Figures 6.3, 6.4, 6.5 shows the application of different dilating structures to the waypoints. Algorithm 7 describes how waypoint clusters $W_{cluster}$ is calculated, *path* holds the path to the goal.

[algorithm for clusters]

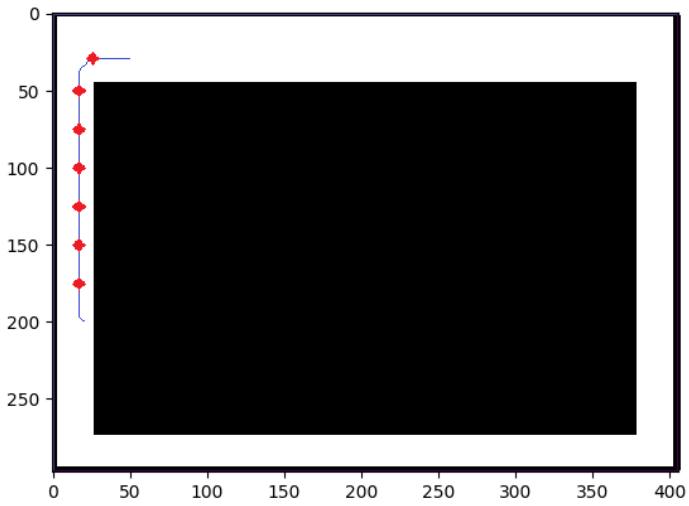


Figure 6.4: Path with Waypoint clusters, and diamond shaped dilating structure.

Algorithm 7: Waypoint and Waypoint Cluster Calculation

$t = f =$ zeros($map_{dimensions}$);
$f[\ path[:,0]\ ,\ path[:,1]\ ] = 1$;
$f = t * g$;
$W =$ where mod(f , w) == 0;
$W_G = t[W] = 1$;
$ds =$ ones(d_{s1} , d_{s2}) ;
$W_{cluster} =$ dilation(W_G, d_s);

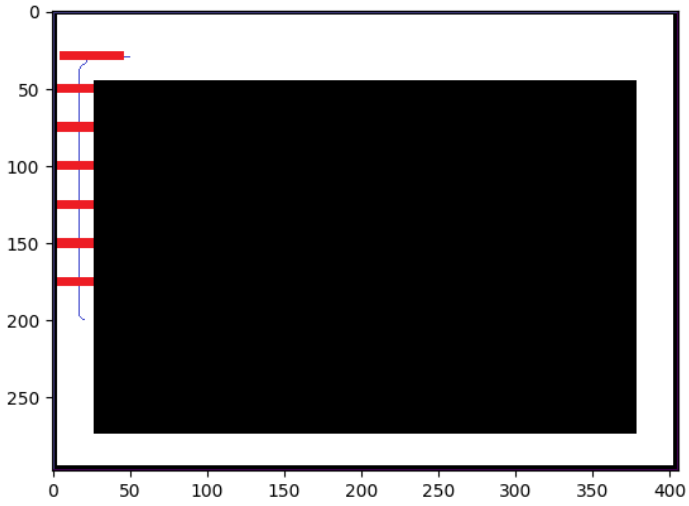


Figure 6.5: Path with Waypoint clusters going across the corridor.

6.5 Following

With the methods for path, waypoint and waypoint cluster calculation in hand, the path planning node can be discussed. The destination Milpet needs to go to, comes from the GUI or speech node, while the start point comes from the localization system. The start and destination are fed to the path planning function and a path is returned. Following this, waypoints are calculated and stored in order. The first waypoint is popped and dilated to get its waypoint cluster. The heading direction is calculated and published to the obstacle avoidance system. The robot travels in this heading direction, until it determines that it has reached the waypoint cluster. The waypoint cluster is met when the location is within the waypoint cluster n times. Having this n, makes sure that the robot has actually reached the location, since the localization system can jump around at times. n is generally set as 4. After it determines that it has reached the intermediate waypoint, it pops the next waypoint and repeats this process. If the goal belongs to the waypoint cluster, then the obstacle avoidance system is signaled to stop.

6.6 System Integration

The path planning system, ties the localization and obstacle avoidance systems together. The path planning system, acts as a master to the obstacle avoidance system, by feeding heading directions to it. Varying w the amount of control the path planner has on the obstacle avoidance system can be changed. Figure 6.6 shows the relation between the systems.

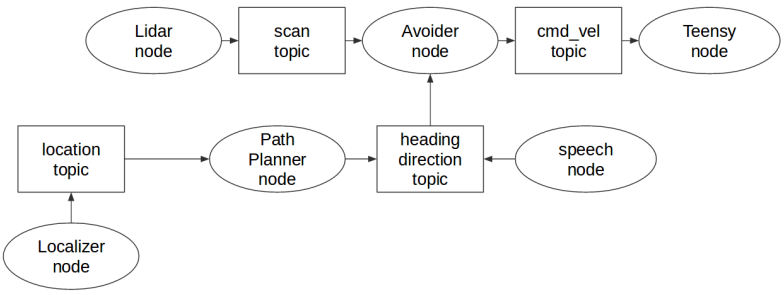


Figure 6.6: Navigation ROS nodes system.

6.7 Centering

The path that the planner gives is the shortest path. This causes the path to be along the wall. This in turn affects the heading direction, causes waypoint clusters to be close to the walls, where the localization is weak. Thus, a need to have the path centered in the corridor was felt. Figure 6.7 shows the path without the centering on. With centering on, we get Figure 6.3. For all previous figures in this section, centering has been applied.

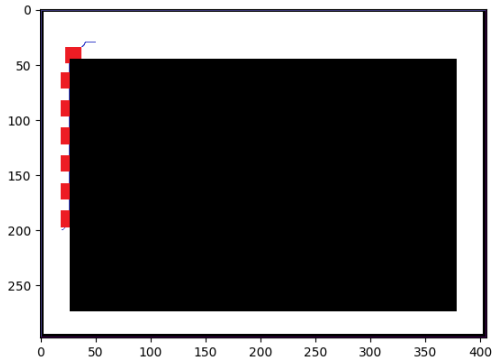


Figure 6.7: Path with Waypoint clusters going across the corridor.

To address this need, the area in the middle of the corridor must have a lesser cost and those near the walls should have a higher cost. To apply this kind of gradient to the map, a square filter was convolved over the map in Figure 6.1, resulting in Figure 6.8, called a gradient map. Figures 6.8, 6.9, 6.10, 6.11 show the gradient map

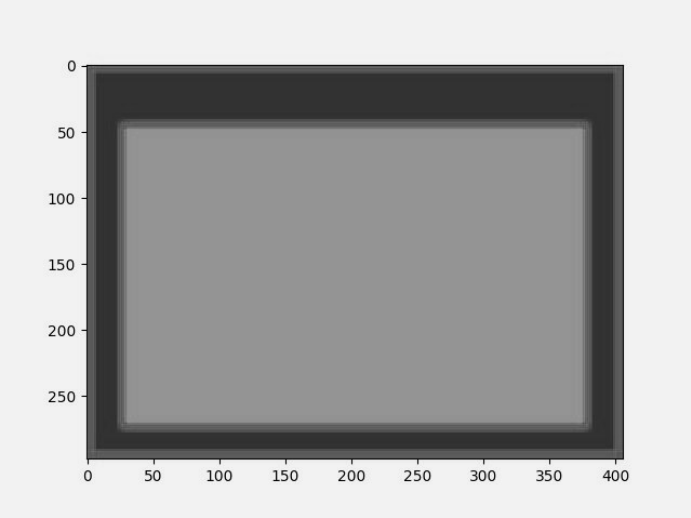


Figure 6.8: Filter size, $f_s = 9$.

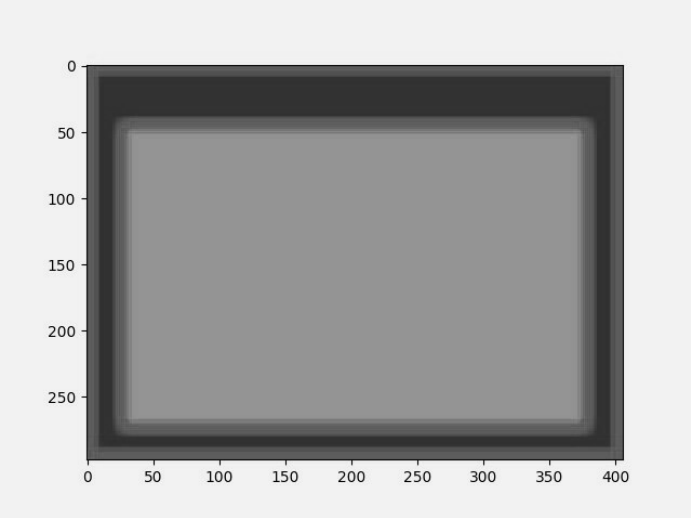


Figure 6.9: Filter size, $f_s = 15$.

at various filter sizes, f_s. The larger the filter size, the more centered the path is. f_s cannot be greater than the number of pixels that represent the width of the narrowest corridor.

To incorporate this gradient map into the path planner, an additional layer of representation was added. The equation to calculate the rhs value is updated to incorporate the gradient map costs.

Figure 6.3 shows the path after implementing this centering technique.

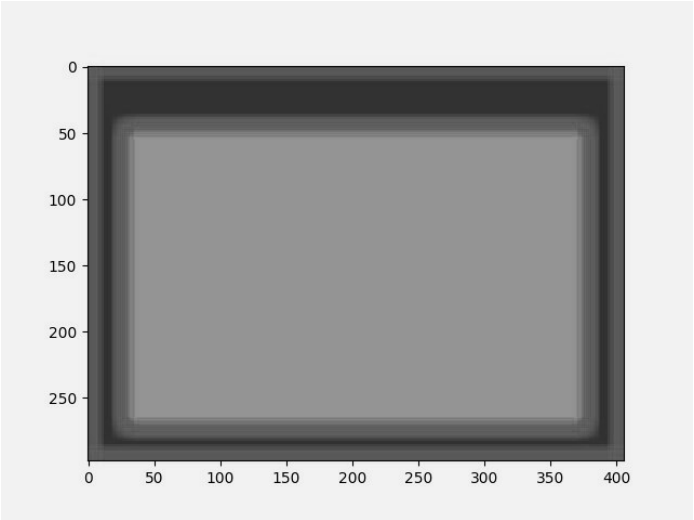


Figure 6.10: Filter size, $f_s = 19$.

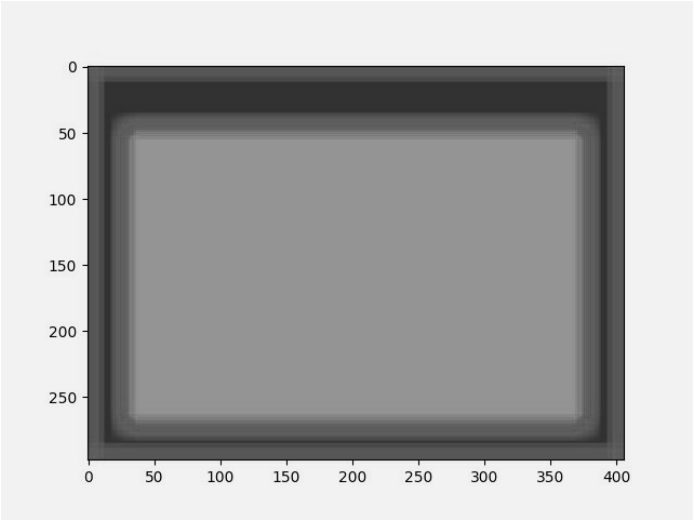


Figure 6.11: Filter size, $f_s = 21$.

Chapter 7

Vision and Lidar Data Fusion

This section deals with fusing the front facing camera data with the Lidar data. By doing this, the robot will have a more comprehensive view of its environment and can react intelligently to different types of objects it identifies. For example, if the hallway the robot is traveling through has a few chairs and tables, the robot can speed up since these are all static objects. When the robot identifies people moving about in the hallway, it can decrease its speed. This way the robot's environment affects its behavior. The front facing camera has a resolution of 0.1875° with a +/-30° field of view, while that of the Lidar is 0.25° for a +/-135° field of view.

YOLOv2 recognizes 80 different classes and provides a bounding box around the object in the image. Each bounding box is accompanied with a label, top, bottom, right, left coordinates and a confidence. This information can then be overlaid with the Lidar data. Figure 7.1 shows the stages in sensor fusion. The output bounding boxes and labels of YOLO along with the filtered Lidar are inputs to Stage 1. Stage 1 adjusts the Lidar and vision data frames to overlay the field of views of one another. Stage 2 then converts the image resolution to that of the scan resolution. Following this, the two frames are matched up in Stage 3. The bounding boxes given by YOLO can at times be larger than the actual object, this results in incorrect marking up of the Lidar scan. To overcome this, thresholding is carried out in Stage 4 to output the fused data in Stage 5.

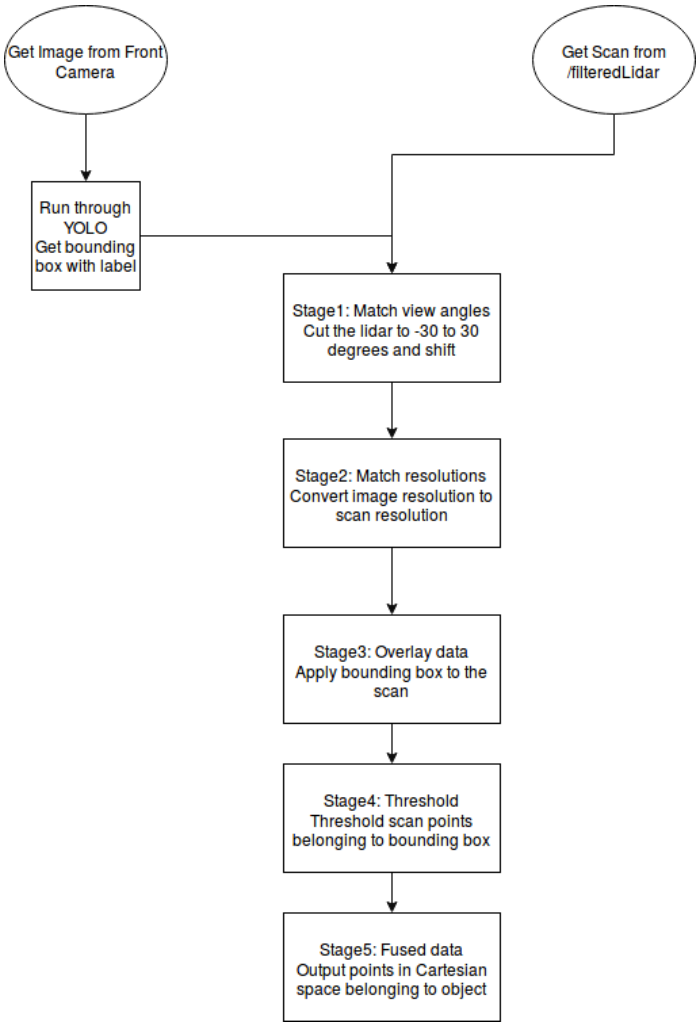


Figure 7.1: Image and Lidar scan data fusion stages.

Figure 7.2 shows the entire scan with a person standing in front.

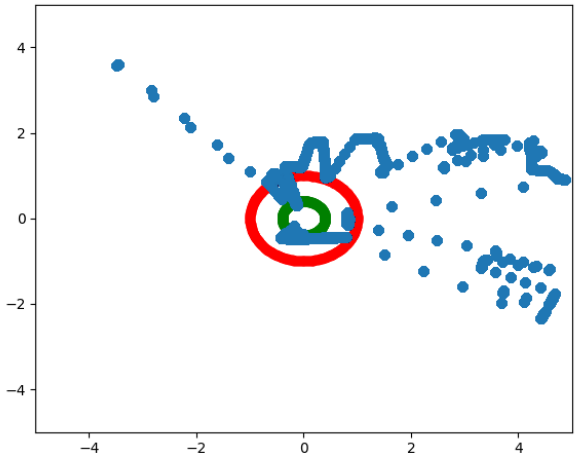


Figure 7.2: Entire scan -135° to 135° : Robot is facing right; blue are points from the Lidar; green circle represents the robot; red circle represents a 2 meter radius.

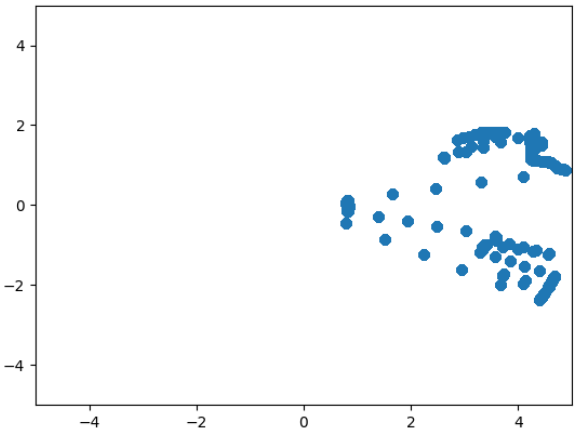


Figure 7.3: Scan cut to -30° to 30° to match that of the camera.

Figure 7.3 shows the scan cut to -30° to 30 °.

Figure 7.4 shows the output after overlaying the bounding box onto the scan. Here, there are extra points not belonging to person class also marked as person. This is caused due to the bounding box being larger at times than the object.

By using the distance information of the marked points, we can now threshold

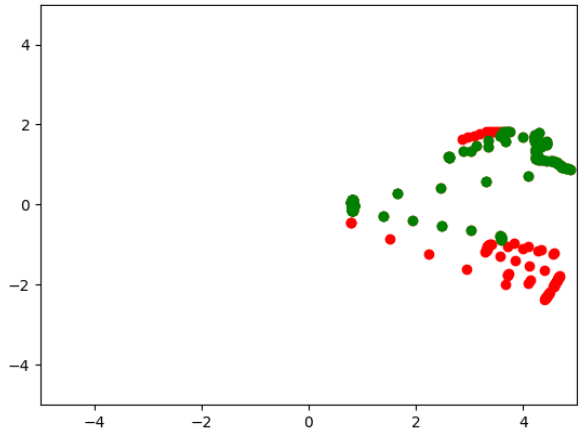


Figure 7.4: Scan marked up: green shows points belonging to person class; red shows points belonging to no object.

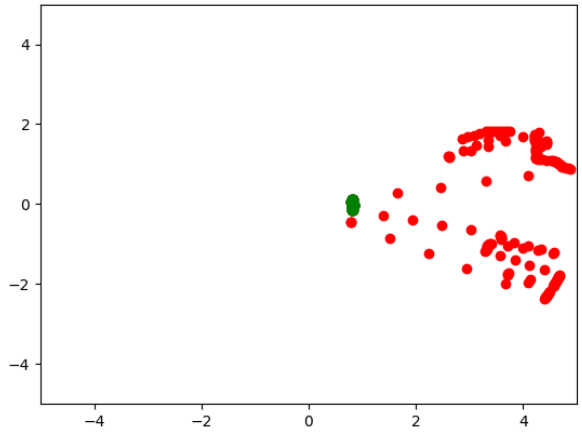


Figure 7.5: Thresholding marked points with distance information.

these points to remove these outliers. This is done by finding the minimum of the marked points for each bounding box and then eliminating points at a distance greater than the threshold from the cluster. This threshold was determined empirically to be 0.4. This is sums up Stage 4. Figure 7.5 shows the output of Stage 4. These points are then outputted.

Chapter 8

Results

This chapter discusses the observed behavior and results of the robot.

8.0.1 Obstacle Proximity

For safety evaluation purposes, each test carried out is accompanied by an obstacle proximity graph. This graph shows the average distance and minimum distance to the closest obstacle, from the four corners of Milpet, and Lidar. These five points have been termed as anchor points.

For safety purposes, getting within 12 inches to an obstacle is termed as a danger zone, and is shown in the obstacle proximity figures. A collision occurs when the distance to an obstacle is zero, this is also depicted in the proximity figures.

8.1 Obstacle Behavior Modes

By changing the behavior modes, the robot's behavior can be changed. The following figures show the effect the mode has on the robot's behavior. Tests were carried out in a hallway shown in Fig. 8.1.

Fig. 8.2, Fig. 8.4 show that by putting the robot in heuristic mode, the robot is able to make turns around corners without getting stuck.

Fig. 8.2, Fig. 8.8, Fig. 8.10 show that with no centering the robot oscillates between the walls or follows a particular wall.

Figure 8.1: Hallway area that Milpet was tested in.

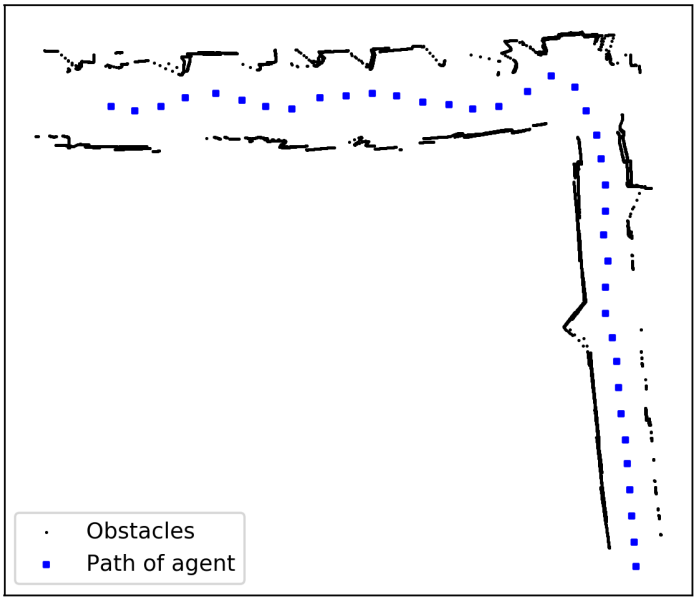


Figure 8.2: Area explored by Milpet in heuristic mode with no centering.

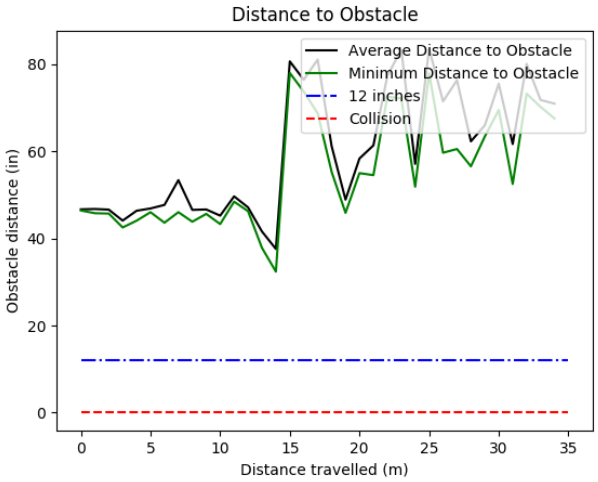


Figure 8.3: Obstacle Proximity for Fig. 8.2.

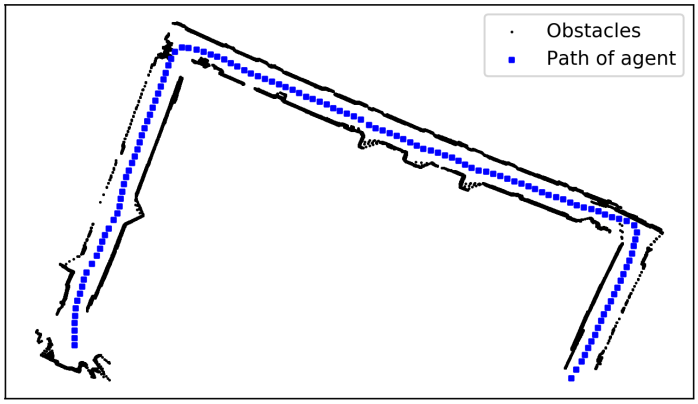


Figure 8.4: Area explored by Milpet in heuristic mode with centering on.

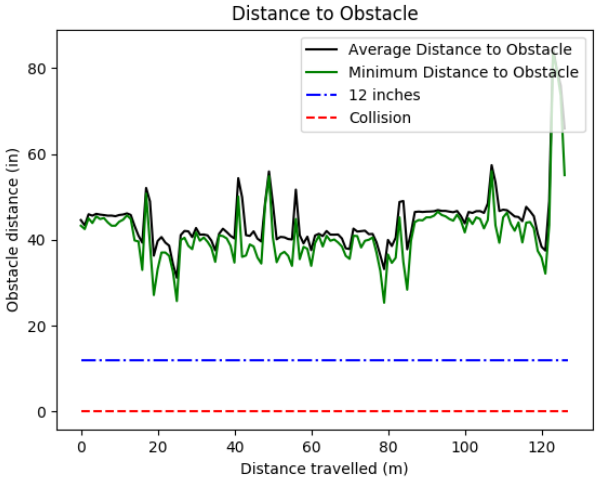


Figure 8.5: Obstacle Proximity for Fig. 8.4.

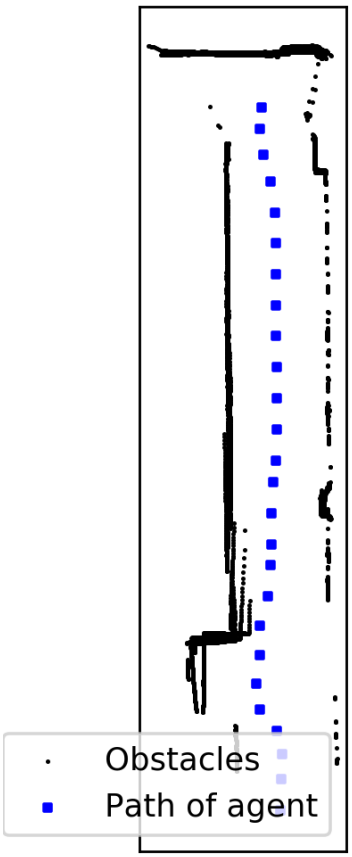


Figure 8.6: Area explored by Milpet in goal mode with centering on and $orientation_{goal}$ = 0.

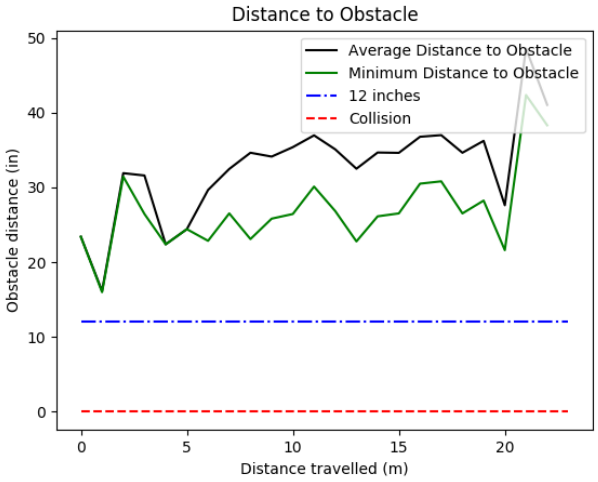


Figure 8.7: Obstacle Proximity for Fig. 8.6.

Fig. 8.8 shows that with the $orientation_{goal}$ set to zero i.e. straight in front of it, the robot stops at the end of the corridor.

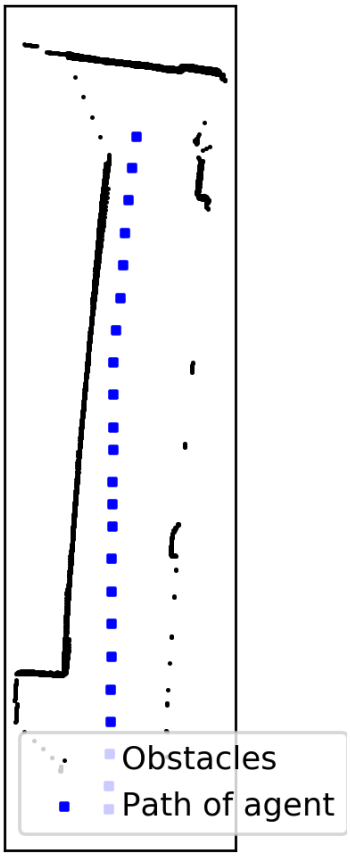


Figure 8.8: Area explored by Milpet in goal mode with no centering and $orientation_{goal} = 0$.

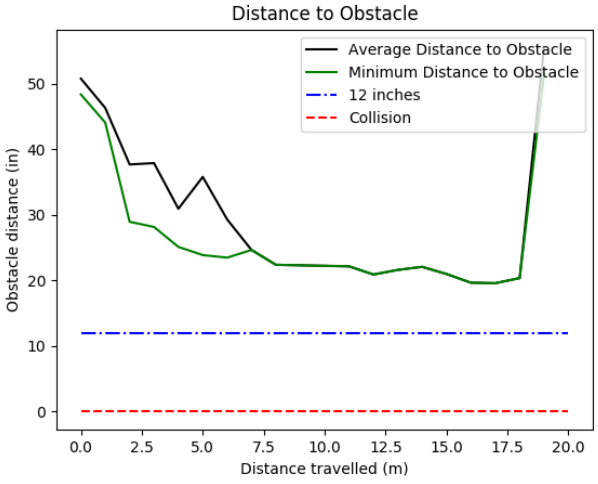


Figure 8.9: Obstacle Proximity for Fig. 8.8.

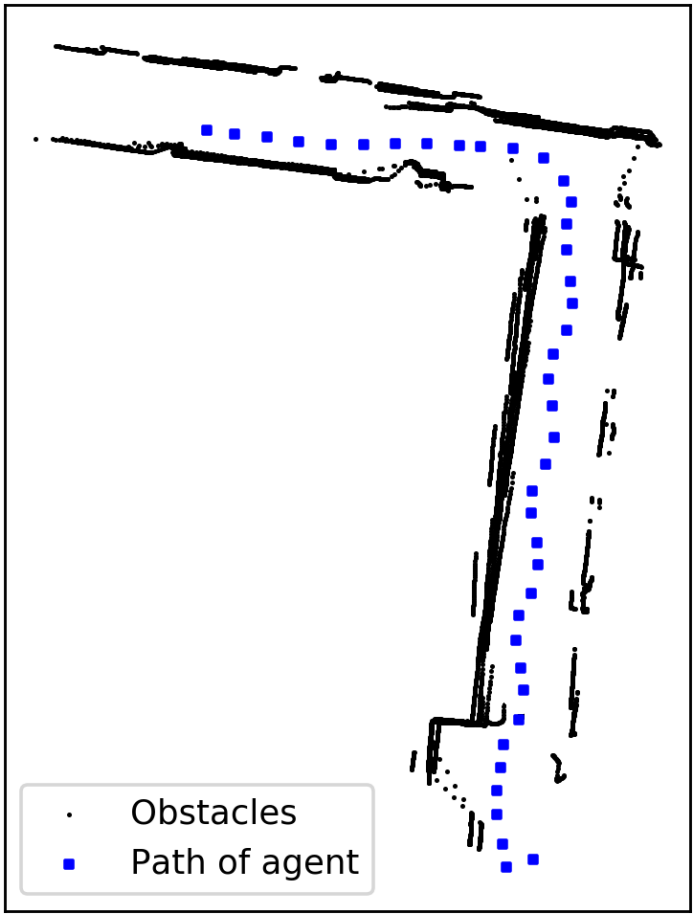


Figure 8.10: Area explored by Milpet in goal mode with no centering and $orientation_{goal}$ = 90.

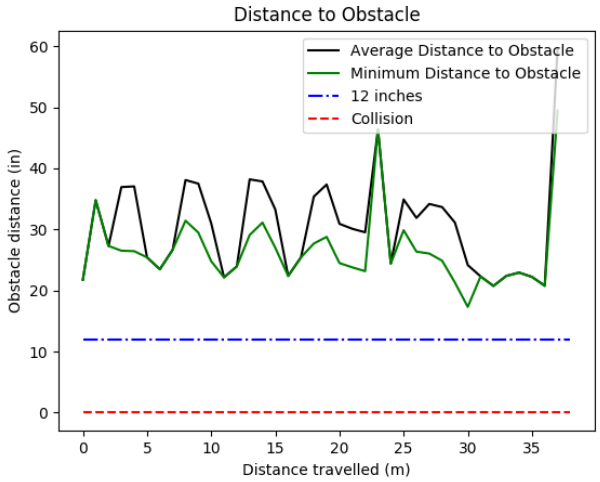


Figure 8.11: Obstacle Proximity for Fig. 8.10.

Fig. 8.10 shows that by setting $orientation_{goal}$ as 90°, the robot goes straight until it can make a 90° turn.

Fig. 8.3, Fig. 8.5, Fig. 8.7, Fig. 8.9, Fig. 8.11 show the obstacle proximity graphs in each mode.

8.1.1 Encoder Free Mode

The obstacle avoidance algorithm gets an understanding of where it is in the world, by using the pose θ, calculated by the encoders. All the figures with the area explored by the robot, have been made using this θ. The encoders are subject to noise and error due wheel slippage, wheel radii measurement error, etc. Due to this, the figures created appear slightly bent or curved.

To demonstrate that the obstacle avoidance algorithm works despite the presence of encoder error, dependency on the encoders has been removed in this encoder free mode. This mode depends merely on the Lidar, however now, the robot has no sense of where it is in the world. Figures 8.12, 8.14 show the robot wandering without getting pose information from the encoders.

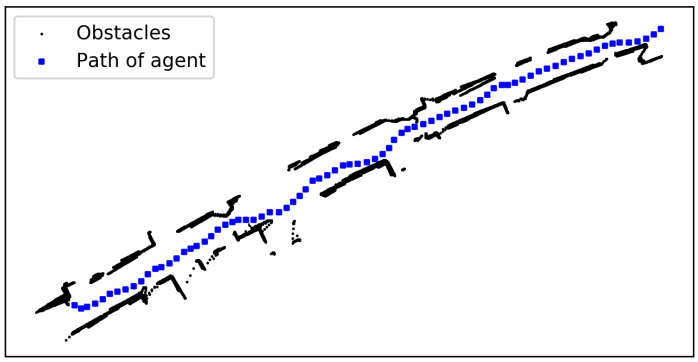


Figure 8.12: Area explored by Milpet in encoder free goal mode with centering and $orientation_{goal} = 0$.

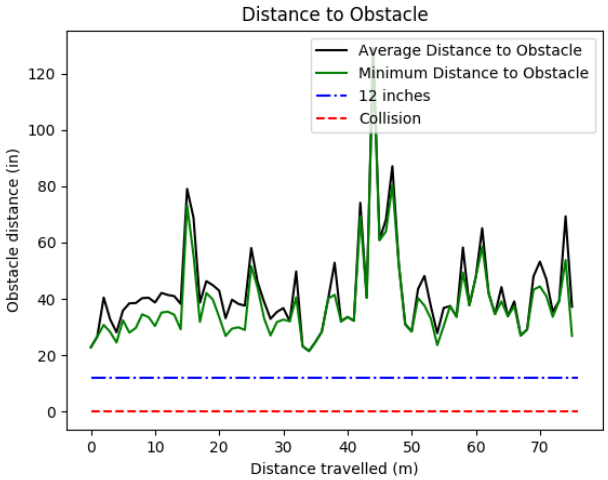


Figure 8.13: Obstacle Proximity for Fig. 8.12.

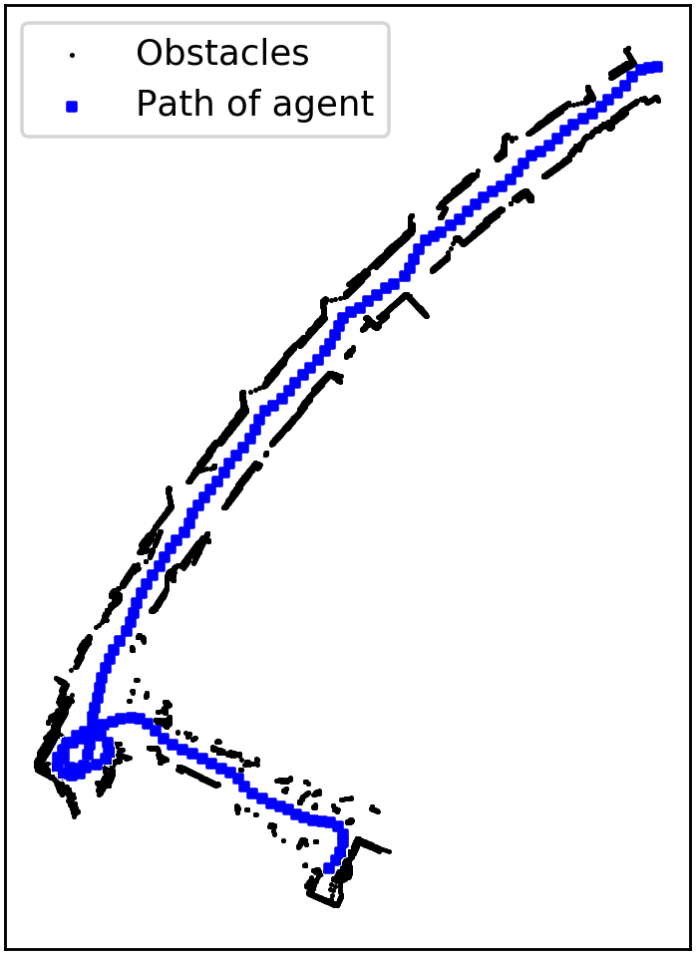


Figure 8.14: Area explored by Milpet in encoder free heuristic mode with centering.

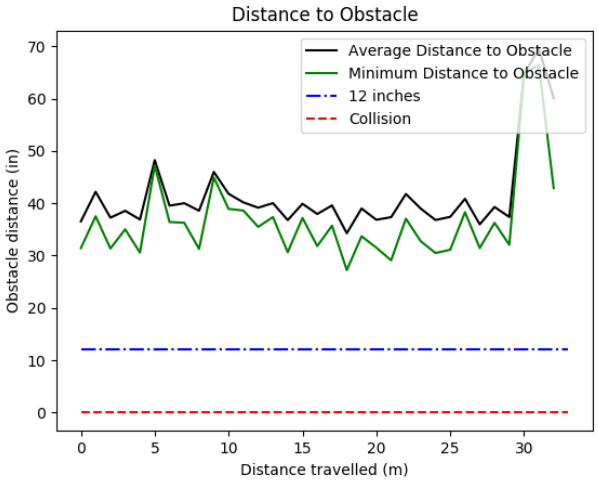


Figure 8.15: Obstacle Proximity for Fig. 8.14.

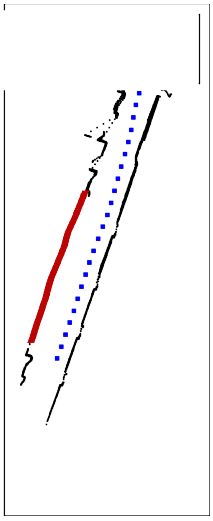


Figure 8.16: Area explored by Milpet: black points are obstacles detected by lidar; red are undetected obstacles (glass); blue is path of robot.

8.1.2 Testing in Glass Areas

Lidar makes use of light to detect distances to obstacles. Since light passes through glass surfaces, the distance to the glass surface cannot be determined. This is a problem for obstacle avoidance systems that make use of Lidars. Tests were carried out to see how the robot behaves in glass environments with both the obstacle avoidance system alone and the entire navigation system. A hallway with one glass wall was used for testing. 40 % of the time the robot got too close to the glass wall, which would have resulted in collision. This is because the Lidar is unable to sense the glass wall and perceives the wall as free space. Figure 8.16 shows a successful attempt through an area with a glass wall. The points in red show the glass. Points in black are obstacles detected by the Lidar. By keeping centering off in obstacle avoidance, the robot performs wall following and is able to make it through the hallway.

8.2 Local Obstacle Avoidance

For demonstrating the obstacle avoidance capabilities of the robot, two scenes were set, as seen in Fig. 8.17 and Fig. 8.18. Each scene had obstacles placed at distances

Figure 8.17: Setup 1 with minimum passage width = 62 inches.

Figure 8.18: Setup 2 with minimum passage width = 50 inches.

equal to the robot's set minimum passage width. This is the minimum distance required for the robot to pass through and ensures that the robot will not attempt to pass through unsafe areas. Setup 1 had a minimum passage width of 62 inches, while Setup 2 had a minimum passage width of 50 inches.

In each setup, different behavior modes were set to see if the robot could make it through safely. Figures 8.19, 8.21, 8.23, 8.25, 8.27 show the path the robot took through its respective setup. Figures 8.20, 8.22, 8.24, 8.26, 8.28 show the obstacle proximity graphs.

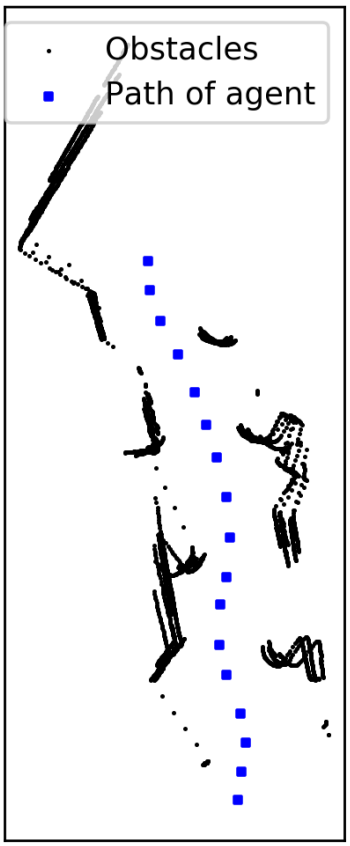


Figure 8.19: Obstacle Avoidance in Setup 1 with heuristic mode and centering on.

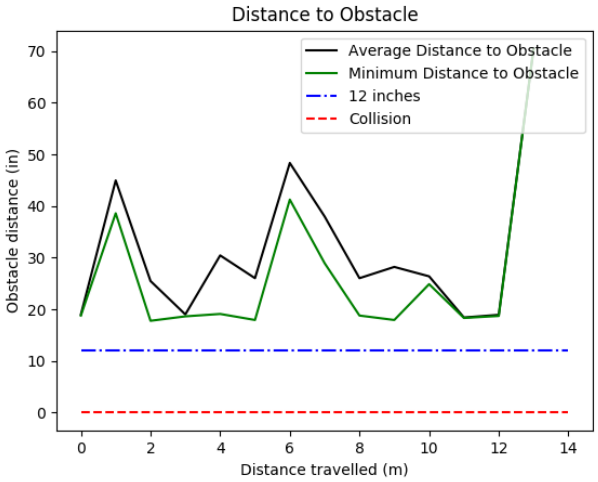


Figure 8.20: Obstacle Proximity for Fig. 8.19.

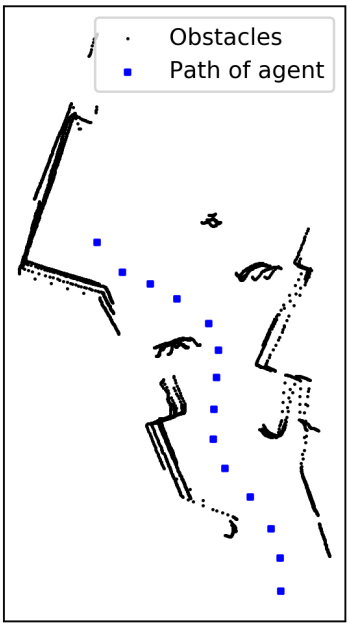


Figure 8.21: Obstacle Avoidance in Setup 2 with goal mode and centering on.

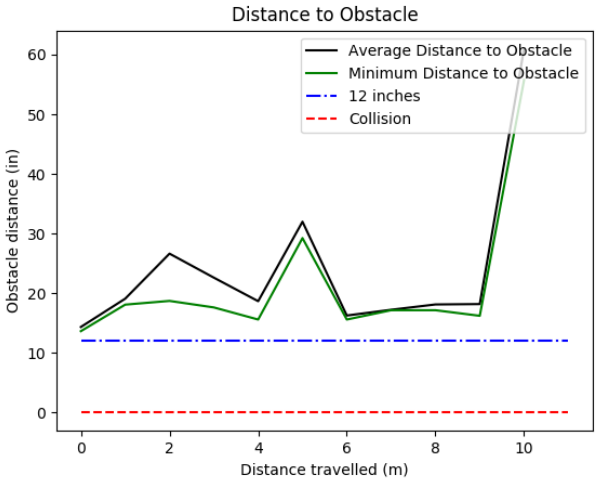


Figure 8.22: Obstacle Proximity for Fig. 8.21.

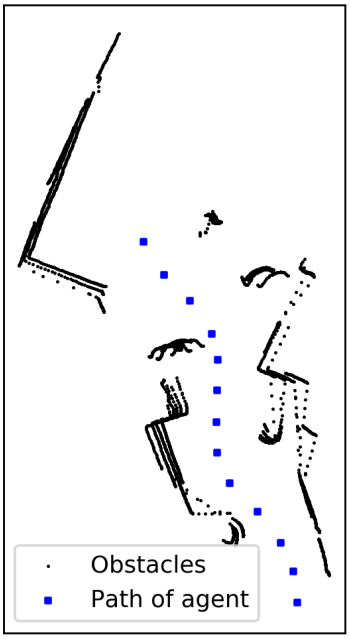


Figure 8.23: Obstacle Avoidance in Setup 2 with goal mode and no centering.

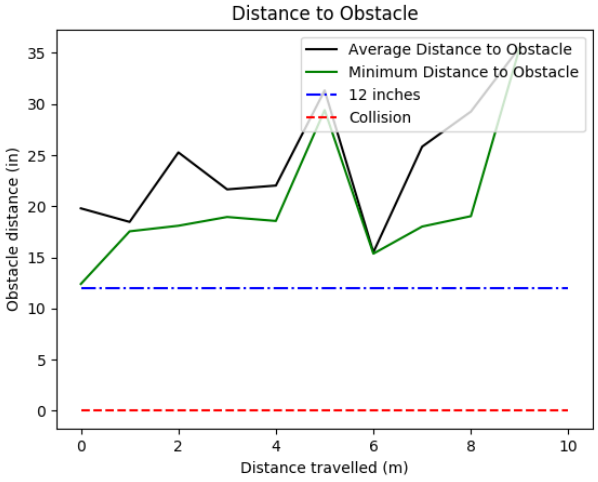


Figure 8.24: Obstacle Proximity for Fig. 8.23.

Figure 8.25: Obstacle Avoidance in Setup 2 with heuristic mode and centering on.

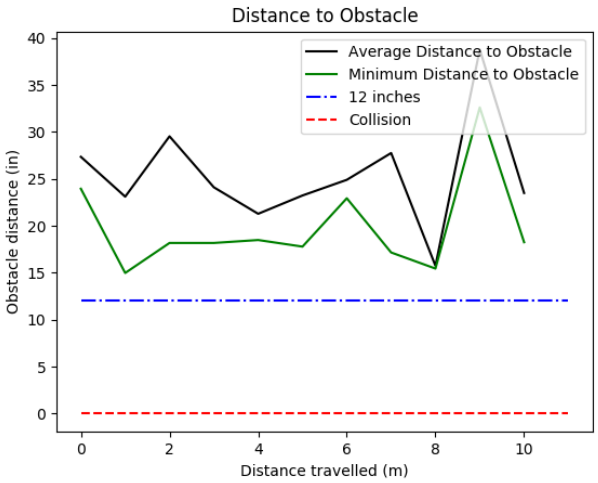


Figure 8.26: Obstacle Proximity for Fig. 8.25.

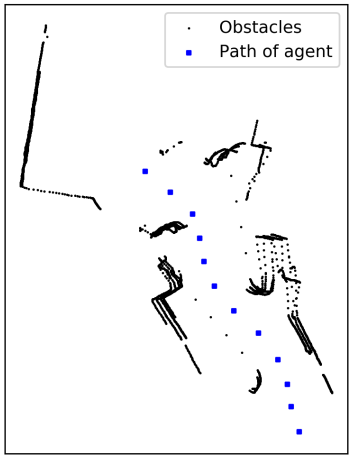


Figure 8.27: Obstacle Avoidance in Setup 2 with heuristic mode and no centering.

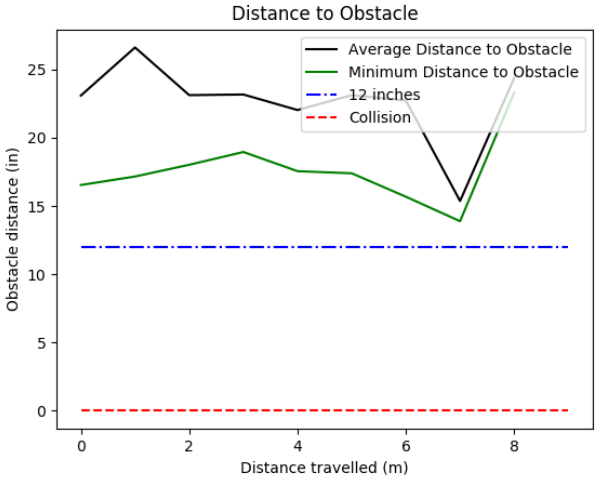


Figure 8.28: Obstacle Proximity for Fig. 8.27.

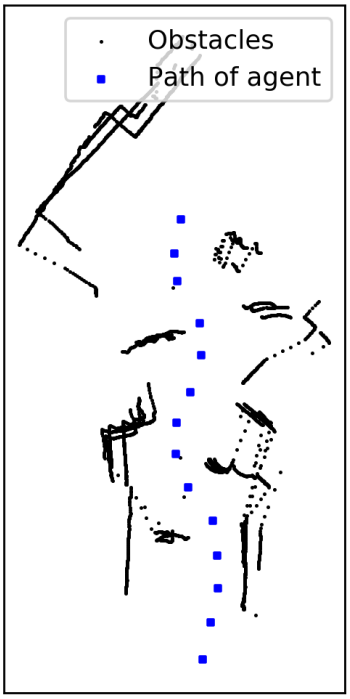


Figure 8.29: Human driving manually through Setup 2.

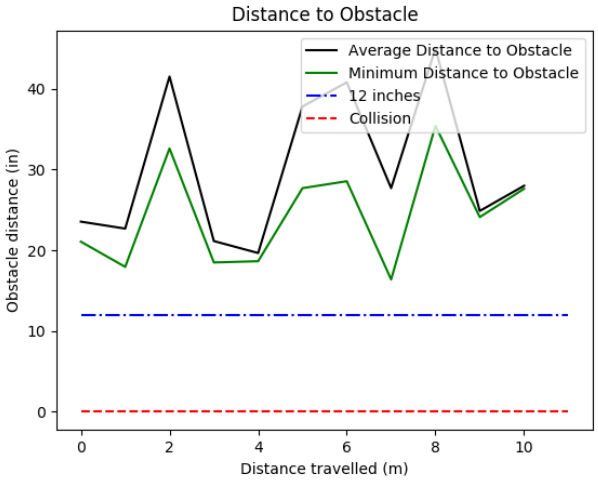


Figure 8.30: Obstacle Proximity for Fig. 8.29.

Fig. 8.29 and Fig. 8.30 show the path and obstacle proximity graph when a person drives through Setup 2.

8.2.1 Avoiding Dynamic Obstacles

Table 8.1: Robot Reaction for Dynamic Obstacles.

Obstacle Speed in (meter per second)	Angle of Approach	Corridor Width (in ft)	Reaction from Robot(Paused/ Collided/ Veered away)	Approximate distance between robot and obstacle when reaction observed (in inches)
0.5	0	7	Paused	35
1	0	7	Paused	20
1.5	0	7	Paused	20
0.5	0	12	Veered away	35
1	0	12	Veered away	30
1.5	0	12	Veered away	28
0.5	+30	7	Paused	35
1	+30	7	Paused	30
0.5	+60	12	Veered away	35
1	+60	12	Veered away	34
1.5	+60	12	Veered away	30
0.5	-30	7	Paused	35
1	-30	7	Paused	30
0.5	-60	12	Veered away	35
1	-60	12	Veered away	30
1.5	+60	12	Veered away	30

Tests for seeing if the robot can avoid dynamic obstacles has been carried out. People have been considered as the dynamic obstacle in this test. At varying angles and speeds the person comes towards the robot and the reaction of Milpet is observed. Table 8.1 sums up the tests carried out. Tests were carried out in two corridors: a wide one of 144 inches and a narrow one of 84 inches. The minimum passage width being used was 62 inches and the robot moved at a linear speed of 0.5 m/s. Obstacles

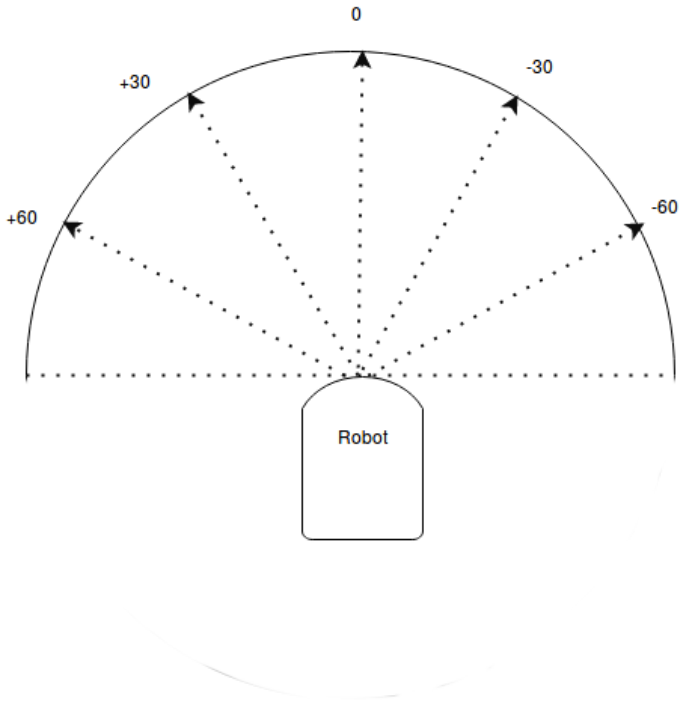


Figure 8.31: Different angles of approach.

approaching straight in front of the robot are said to be oncoming at 0 °. Figure 8.31 shows the different angles of approach for these tests.

8.3 Reaction Time

Safety of Milpet's user is of the utmost importance. When an obstacle suddenly blocks Milpet's path, Milpet should stop as soon as it can, without colliding into the sudden obstacle. By recording how long it takes for Milpet to stop, the reaction times are calculated. Fig. 8.32 shows the reaction time of the robot in seconds, at varying distances from the robot, on encountering a sudden obstacle. The reaction times for Milpet with linear velocities 0.5m/s and 0.3m/s have been measured. From Fig. 8.32, we can see that the robot can stop in less than a second when the sudden obstacle is lesser than 24 inches away from the robot. The reaction time increases as the distance from the robot increases. The reaction time decreases as the speed of the robot increases, for obstacles more than two feet away from the robot.

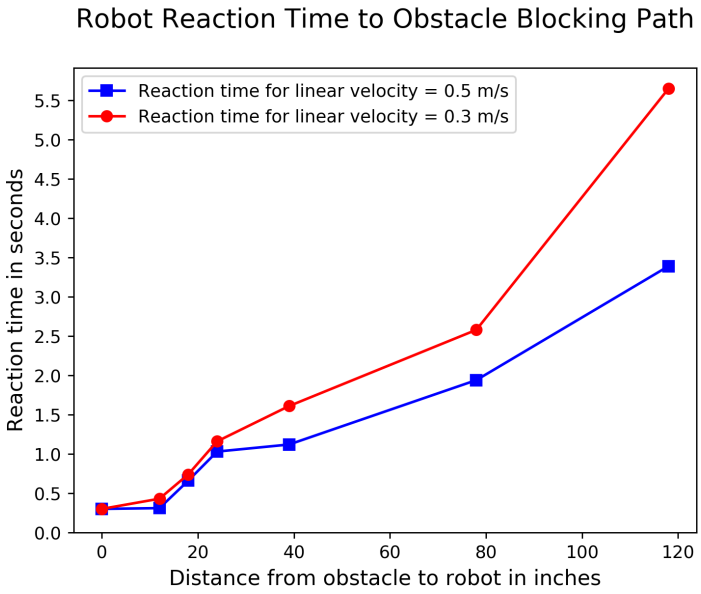


Figure 8.32: Reaction time when a sudden obstacle blocks Milpet's entire path.

8.4 Navigation in Autonomous mode

The results in this section depend on the localization, path planning and obstacle avoidance systems working together. These three systems enable Milpet to exhibit autonomous behavior. By giving a command such as, "Milpet, take me to the kitchen", the autonomous mode is triggered. In the following tests, the distance between each waypoint, w, has been changed and the waypoint cluster size, d_s, for the input speech command: "Milpet, take me to Room 2560".

In Figures 8.33, 8.40, the octagon and diamond represent the start and destination of the path. Using this the waypoint are calculated followed by waypoint clusters shown as squares. Once the next and following waypoint cluster is predicted, the path planning system waits till the localization system predicts the location belonging to one of these clusters. The predicted positions within the clusters have been shown as stars. While following a path, the path planner looks for the next two consecutive waypoint clusters. Due to this even if the localization falls behind for the immediate cluster, the following waypoint cluster can be reached without missing the path. In

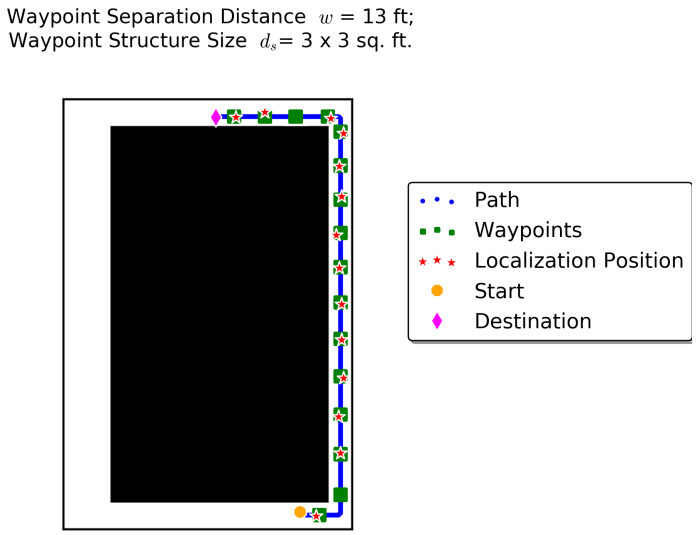


Figure 8.33: Autonomous traveling with $w = 13$ ft and $d_s = 3 \times 3$ sq. ft.

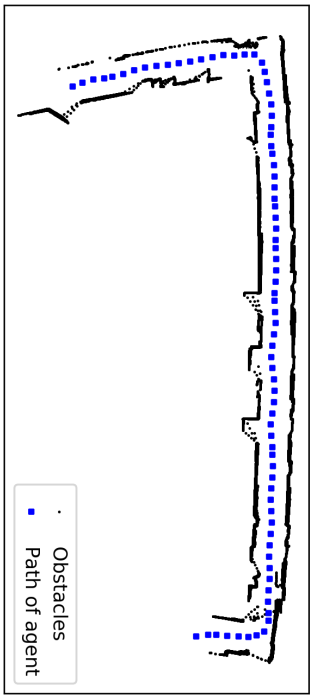


Figure 8.34: Path explored for Fig.8.33.

Figure 8.33, there are waypoint clusters, without a star. These show places where the localization missed a waypoint, but then caught on to the path at the next waypoint, that has two stars.

Table 8.2 shows a summary of the tests carried out. The distance between the

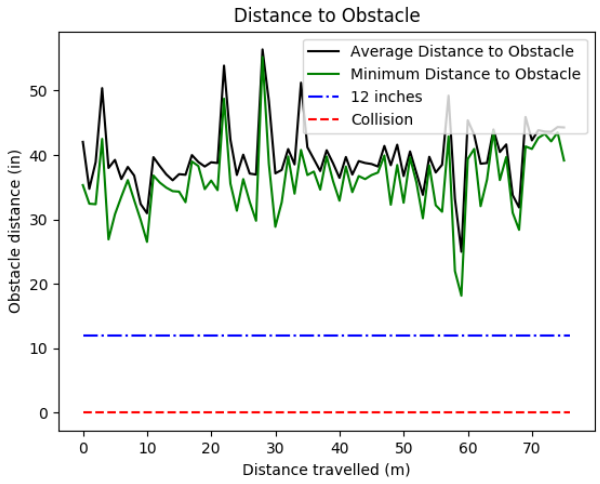


Figure 8.35: Obstacle proximity for path in Fig.8.33.

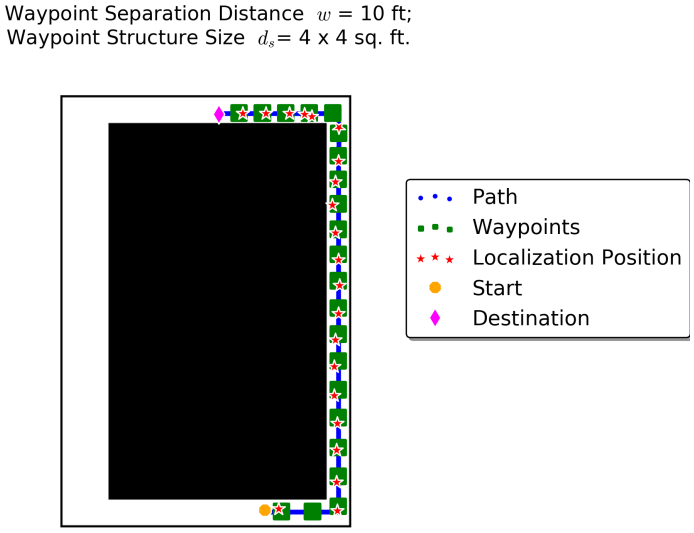


Figure 8.36: Autonomous traveling with $w = 10$ ft and $d_s = 4 \times 4$ sq. ft.

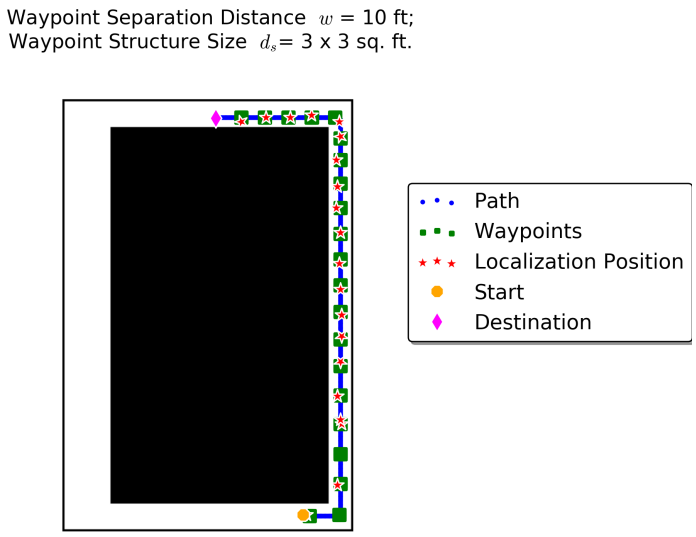


Figure 8.37: Autonomous traveling with $w = 10$ ft and $d_s = 3 \times 3$ sq. ft.

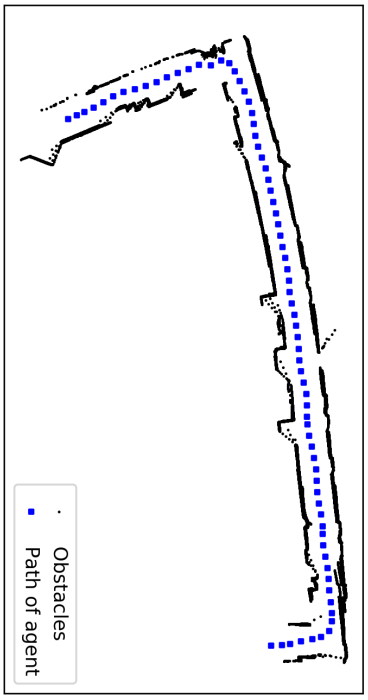


Figure 8.38: Path explored for Fig.8.37.

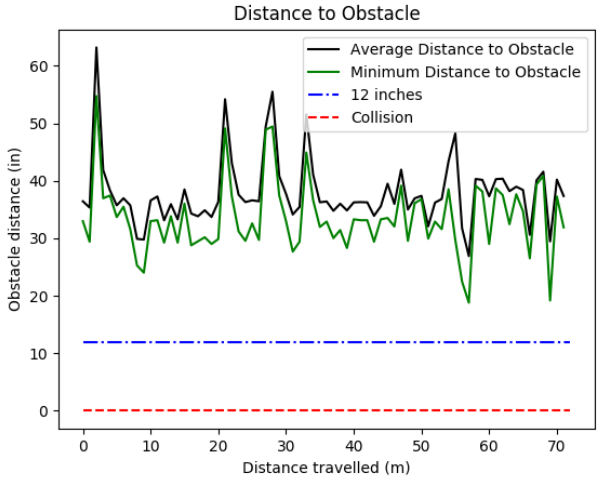


Figure 8.39: Obstacle proximity for path in Fig.8.37.

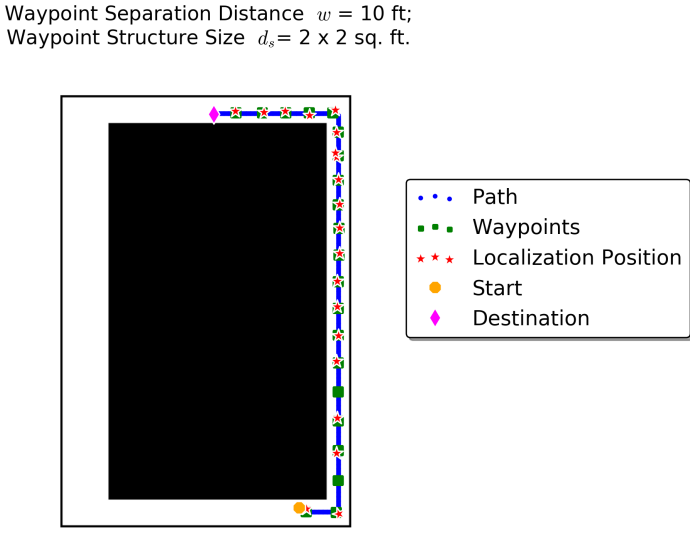


Figure 8.40: Autonomous traveling with $w = 10$ ft and $d_s = 2 \times 2$ sq. ft.

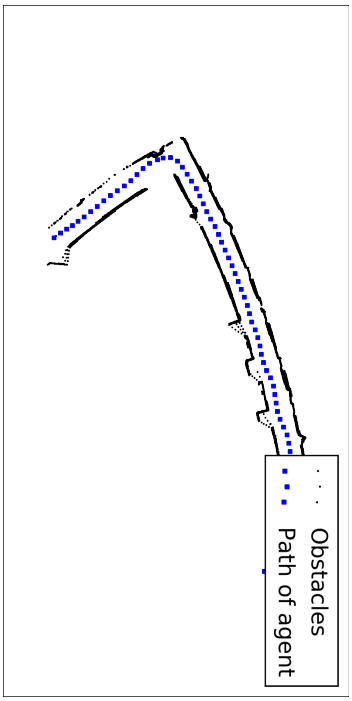


Figure 8.41: Path explored for Fig.8.40.

destination and the location where the robot stopped is termed as destination proximity. It can be seen that the robot reaches the destination with a max offset of 1.4 ft.

Table 8.2: Destination Proximity.

Waypoint Cluster Size d_s (in sq. ft.)	Distance between Waypoint w (in ft.)	Lookahead	Destination Proximity (in ft)
16	13	On	0.8
16	13	Off	1.3
9	13	On	1.4
9	13	Off	0.3
9	10	On	0
9	10	Off	0.2
4	10	On	0
4	10	Off	0

8.5 Smart Wheelchair Comparison

There have been many smart wheelchairs developed over the years, aiming at bettering the life of disabled persons. Smart wheelchairs can be autonomous, semi - autonomous or exhibit obstacle avoidance by augmenting them with different sensors and technologies. The performance amongst wheelchairs is difficult, since each wheelchair has been evaluated against its own performance measures. Table 8.3 describes a series of wheelchairs, their capabilities, sensors and operating modes. For a more detailed survey refer to [35], [36] and [37]. The work described in this study is in the last row. From the entire table, the evaluation metrics of the first wheelchair [38] resemble this study's metrics. Performance is based on number of times the user must takeover in semi-autonomous mode and obstacle clearance. The wheelchair described in [39] determines performance by surveying the user's comfort in the wheelchair. If the wheelchair gets too close to an obstacle, this is termed as a discomfort or invasion of privacy.

Table 8.3: Smart Wheelchair Chart

ID.	Sensors	User Interface	Capabilities	Operating Modes	Evaluations based on	University, Year
[38]	Planar Laser, Wheel Encoders	Touchscreen, Visual Display	N/A	Semi-Autonomous	Frequency of User Intervention, Device Errors, Obstacle Clearance	University of Zaragoza, Spain, 2010
[40]	Laser range finder, wheel encoders, camera	Mobile application, EEG cap, joystick	N/A	Obstacle avoidance	N/A	University of Technology, Sydney, 2012
[41]	Ultrasound, encoders	Keyboard	Dead reckoning	Target following, obstacle avoidance	Stair detection, collision	Shanghai Jiao Tong University, 2014
[42]	Lidar, ultrasonic sensors, encoders	EEG Cap	Localization	Autonomous navigation and obstacle avoidance	Input recognition success	University of Technology, Sydney, 2016
[39]	Lidar	N/A	Human detection	Obstacle avoidance	discomfort, invasion of privacy	Chuo University Tokyo, Japan, 2016
[43]	Lidar, ultrasonic sensors	Analog joystick	SLAM	Autonomous	Collisions, path length, time	California State University, 2017
[44]	N/A	Microphone	Speech recognition	Manual	Input recognition success	Cummins College for Engineering, Women, 2017
Milpet	Lidar, omni-vision camera, front facing camera	Microphone, GUI, joystick mobile app	Speech recognition, voice feedback, object recognition	Autonomous, wandering, manual	Obstacle proximity, collision, destination proximity	Rochester Institute of Technology, 2017

Chapter 9

Conclusion

In conclusion, a navigation system was developed in ROS for Milpet. The path planning and obstacle avoidance systems have been created to work in sync with one another. Each system is modular, making it easy to modify and test new algorithms. Both systems, along with the localization system, have been written in Python, making it easier to debug and maintain in the future. The overall behavior of the robot can be controlled by changing the modes in the obstacle avoidance code. The path planning code was adapted to encompass a few shortcomings of the localization system. Localization is based on image data, and the predicted location can jump between locations on the map. To make the path planning impervious to this, a count variable was added to filter out erroneous predictions.

Smart wheelchairs don't have a standardized method of evaluating performance. Many wheelchairs evaluate performance on input recognition success. Input recognition varies from wheelchair to wheelchair, as each has a different interaction interface. This doesn't give an idea about the actual working of the navigation system. In this study, the measure of obstacle proximity and destination proximity have been used for performance measure. Results for evaluating a smart wheelchair's response to dynamic obstacles, like people, has also been documented. Reaction time for how fast the wheelchair can stop has been measured as well. These are important evaluation criteria and can help in comparing different smart wheelchairs.

Testing was carried out on vision data from the front facing camera and how it can be used to improve Milpet's capabilities and understanding of its environment.

For an autonomous agent that aims at being used for patients in wheelchairs, safety is the most important aspect. The results that Milpet exhibit show that safety is placed first. Having more sensors integrated into the system would make Milpet more reliable and secure. When Milpet reaches areas it can't navigate, it asks for help from the user. Milpet can then be controlled via speech, keyboard or with the bluetooth joystick mobile application.

Chapter 10

Future Work

Milpet can be improved upon in many ways. Overcoming the limitations of the current system, is an area of improvement for Milpet. Obstacles were made sure to have a height of 28 inches, which matches the height of the Lidar. Most tests were carried in areas not having glass walls. For testing the entire navigation system, areas had to be lit, else the localization system, which depends on image data, would fail.

Since the base systems are in place, viz. Localization, Path Planning and Obstacle Avoidance, more sensors and technology can be added to it to make Milpet more intelligent.

For having smarter obstacle avoidance and interaction capabilities, the front facing camera data can be fused with the Lidar's data in real-time, to make Milpet interact and even recognize people.

Milpet's interaction capability can be expanded to make it capable of having conversations with its user at all times, as opposed to only during special cases like when the navigation system cannot find a path to the goal.

Various smart wheelchairs have additional capabilities like guide following, stairway detection and door passage. These can also be added to Milpet.

Bibliography

[1] N. T. Nhu and N. T. Hai, "Landmark-based robot localization using a stereo camera system," *American Journal of Signal Processing*, vol. 5, no. 2, pp. 40–50, 2015.

[2] N.-A. A. T. Education), "Acoustic emission source location techniques."

[3] L. Buşoniu and L. Tamás, *Handling Uncertainty and Networked Structure in Robot Control*. Springer, 2015.

[4] L.-S. J. A. T. Choset, Mills Tettey, "Robotic motion planning: A* and d* search," *Computer Science, Carnegie Mellon University*.

[5] S. Rodrigues, "Obrrt: An obstacle-based rapidly-exploring random tree."

[6] J. Borenstein and Y. Koren, "The vector field histogram-fast obstacle avoidance for mobile robots," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 3, pp. 278–288, Jun 1991.

[7] I. Ulrich and J. Borenstein, "Vfh+: reliable obstacle avoidance for fast mobile robots," in *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146)*, vol. 2, May 1998, pp. 1572–1577 vol.2.

[8] ——, "Vfh*: local obstacle avoidance with look-ahead verification," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, vol. 3, 2000, pp. 2505–2511 vol.3.

[9] D. Fox, W. Burgard, and S. Thrun, "The dynamic window approach to collision avoidance," *IEEE Robotics Automation Magazine*, vol. 4, no. 1, pp. 23–33, Mar 1997.

[10] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015. [Online]. Available: http://arxiv.org/abs/1506.02640

[11] J. Redmon and A. Farhadi, "YOLO9000: better, faster, stronger," *CoRR*, vol. abs/1612.08242, 2016. [Online]. Available: http://arxiv.org/abs/1612.08242

[12] Y. Nasri, V. Vauchey, R. Khemmar, N. Ragot, K. Sirlantzis, and J.-Y. Ertaud, "Ros-based autonomous navigation wheelchair using omnidirectional sensor," *International Journal of Computer Applications*, vol. 133, no. 6, pp. 12–17, 2016.

[13] S. Hemachandra, M. R. Walter, S. Tellex, and S. Teller, "Learning spatial-semantic representations from natural language descriptions and scene classifications," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*. IEEE, 2014, pp. 2623–2630.

[14] S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain," *IEEE Transactions on Robotics*, vol. 21, no. 3, pp. 354–363, June 2005.

[15] J. J. Leonard and H. F. Durrant-Whyte, "Mobile robot localization by tracking geometric beacons," *IEEE transactions on robotics and automation*, vol. 7, no. 3, pp. 376–382, 1991.

[16] U. government, "Gps accuracy."

[17] R. K. McConnell, "Method of and apparatus for pattern recognition," Jan. 28 1986, uS Patent 4,567,610.

[18] D. G. Lowe, "Object recognition from local scale-invariant features," in *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, vol. 2. Ieee, 1999, pp. 1150–1157.

[19] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[20] A. Stentz, "Optimal and efficient path planning for unknown and dynamic environments," DTIC Document, Tech. Rep., 1993.

[21] A. Stentz *et al.*, "The focussed dˆ* algorithm for real-time replanning," in *IJCAI*, vol. 95, 1995, pp. 1652–1659.

[22] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes challenge: A retrospective," *International Journal of Computer Vision*, vol. 111, no. 1, pp. 98–136, Jan. 2015.

[23] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Computer Vision and Pattern Recognition*, 2014.

[24] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.

[25] C. V. F. Videos, "You only look once: Unified, real-time object detection," https://www.youtube.com/watch?v=NM6lrxy0bxs&t=231s.

[26] R. C. Simpson, "Smart wheelchairs: A literature review," *Journal of rehabilitation research and development*, vol. 42, no. 4, p. 423, 2005.

[27] S. Hemachandra, T. Kollar, N. Roy, and S. Teller, "Following and interpreting narrated guided tours," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE, 2011, pp. 2574–2579.

[28] O. S. R. Foundation, "Sphinx knowledge base tool – version 3," www.ros.org.

[29] W. Foundation, "Moment of inertia," https://en.wikipedia.org/wiki/Moment_of_inertia.

[30] O. S. R. Foundation, "Ros joints," http://wiki.ros.org/urdf/XML/joint.

[31] P. S. Foundation, "Speech recognition 3.6.5," https://pypi.python.org/pypi/SpeechRecognition/.

[32] C. M. University, "Cmusphinx: Open source speech recognition," https://cmusphinx.github.io/.

[33] A. Rudnicky, "Sphinx knowledge base tool – version 3," http://www.speech.cs.cmu.edu/tools/lmtool-new.html, accessed: 2017-05-08.

[34] CMUSphinx, "Adaptation of the acoustic model," https://cmusphinx.github.io/wiki/tutorialadapt/, accessed: 2017-05-08.

[35] R. C. Simpson, "Smart wheelchairs: A literature review," *Journal of rehabilitation research and development*, vol. 42, no. 4, p. 423, 2005.

[36] B. M. Faria, L. P. Reis, and N. Lau, "A survey on intelligent wheelchair prototypes and simulators," in *New Perspectives in Information Systems and Technologies, Volume 1*. Springer, 2014, pp. 545–557.

[37] J. Leaman and H. M. La, "A comprehensive review of smart wheelchairs: Past, present and future," *arXiv preprint arXiv:1704.04697*, 2017.

[38] L. Montesano, M. Díaz, S. Bhaskar, and J. Minguez, "Towards an intelligent wheelchair system for users with cerebral palsy," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 18, no. 2, pp. 193–202, 2010.

[39] Y. Sawada and M. Niitsuma, "Dynamic obstacle avoidance based on obstacle type for interactive smart electric wheelchair," in *Industrial Electronics Society, IECON 2016-42nd Annual Conference of the IEEE*. IEEE, 2016, pp. 5891–5896.

[40] H. T. Trieu, H. T. Nguyen, and K. Willey, "Obstacle avoidance for power wheelchair using bayesian neural network," in *Engineering in Medicine and Biology Society, 2007. EMBS 2007. 29th Annual International Conference of the IEEE*. IEEE, 2007, pp. 4771–4774.

[41] J. Zhang, J. Wang, and W. Chen, "A control system of driver assistance and human following for smart wheelchair," in *2014 IEEE International Conference on Robotics and Biomimetics (ROBIO 2014)*, Dec 2014, pp. 1927–1932.

[42] R. Zhang, Y. Li, Y. Yan, H. Zhang, S. Wu, T. Yu, and Z. Gu, "Control of a wheelchair in an indoor environment based on a brain–computer interface and automated navigation," *IEEE transactions on neural systems and rehabilitation engineering*, vol. 24, no. 1, pp. 128–139, 2016.

[43] H. Grewal, A. Matthews, R. Tea, and K. George, "Lidar-based autonomous wheelchair," in *Sensors Applications Symposium (SAS), 2017 IEEE*. IEEE, 2017, pp. 1–6.

[44] S. U. UPase, "Speech recognition based robotic system of wheelchair for disable people," in *2016 International Conference on Communication and Electronics Systems (ICCES)*, Oct 2016, pp. 1–5.

[45] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," 1998.

[46] O. S. R. Foundation, "Gazebo," www.ros.org/gazebo.

[47] S. N. Patel and V. Prakash, "Autonomous camera based eye controlled wheelchair system using raspberry-pi," in *Innovations in Information, Embedded and Communication Systems (ICIIECS), 2015 International Conference on*. IEEE, 2015, pp. 1–6.