Robot Designed for Socially Acceptable Navigation

by

Michael F. Everett

S.B., Massachusetts Institute of Technology (2015)

Submitted to the Department of Mechanical Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Mechanical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

© Massachusetts Institute of Technology 2017. All rights reserved.

Author . . . .
Department of Mechanical Engineering
May 12, 2017

Certified by . . . .
Jonathan P. How
Richard C. Maclaurin Professor of Aeronautics and Astronautics
Thesis Supervisor

Read by . . . .
David L. Trumper
Professor of Mechanical Engineering
Thesis Reader

Accepted by . . . .
Rohan Abeyaratne
Graduate Officer, Department of Mechanical Engineering

Robot Designed for Socially Acceptable Navigation

by

Michael F. Everett

Submitted to the Department of Mechanical Engineering
on May 12, 2017, in partial fulfillment of the
requirements for the degree of
Master of Science in Mechanical Engineering

Abstract

Autonomous (self-driving) vehicles are increasingly being tested on highways and city streets. But there is also a need for robots that can navigate through environments like sidewalks, buildings, and hallways. In these situations, the robots must interact and cooperate with pedestrians in a socially acceptable manner. The "rules of the road" no longer apply – there are no lanes or street signs, and pedestrians don't use turn signals when cutting through crowds. This thesis describes the hardware and software architecture of a robot that was developed for this application. This thesis also proposes a 2nd generation robot with tighter budget and size constraints. Finally, this thesis presents a novel collision avoidance method that extends the Reciprocal Velocity Obstacle (RVO) framework to consider the impact of planning decisions on future world states.

Thesis Supervisor: Jonathan P. How
Title: Richard C. Maclaurin Professor of Aeronautics and Astronautics

Thesis Reader: David L. Trumper
Title: Professor of Mechanical Engineering

Acknowledgments

The author would like to thank several individuals for an extraordinary experience over the first six years at the Massachusetts Institute of Technology.

First, the author thanks Professor Jonathan How for his support, motivation, and advice throughout graduate school. The author would also like to acknowledge and thank Professor David Trumper as the official reader from the Department of Mechanical Engineering.

Second, the author thanks each of the members of the Aerospace Controls Laboratory, and especially Steven Chen, Justin Miller, Brett Lopez, and Shayegan Omidshafiei.

Third, the author thanks the financial supporters of his graduate education, including Ford Motor Company and Boeing.

The author also thanks the many friends that have made MIT such a fun place, including Brendon Chiu and Meghan Torrence.

Finally, the author thanks his family members: Mom ('79 SB, '81 SM), Dad ('76 SB, '91 PhD), Tim, Katie ('12 SB, '13 MEng), and Patrick ('17 SB) for their support.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

1 Introduction **17**

1.1 Motivation 18

1.2 Literature Review 20

1.2.1 Academic Research 20

1.2.2 Commercial Robots 22

1.2.3 Summary 22

1.3 Contributions 23

2 Generation 1 Hardware **25**

2.1 Chassis 25

2.2 Sensors 27

2.2.1 Lidar 27

2.2.2 Monocular Camera 30

2.2.3 Intel Realsense 31

2.2.4 Bumper 32

2.2.5 Sensor Hub 33

2.2.6 Other Sensors 35

2.3 Computers 36

2.4 Summary 37

3 Software **39**

3.1 Software Architecture 39

3.1.1 Block Diagram 40

3.1.2 Localization . . . 40
3.1.3 Perception & Costmap . . . 42
3.1.4 Path Planning . . . 44
3.1.5 Control . . . 50
3.1.6 Task & Goal Selection . . . 54
3.2 Summary . . . 56

4 Generation 2 Robot **57**
4.1 Constraints . . . 57
4.2 CAD Models . . . 58
4.2.1 Sensor Plate Design . . . 58
4.3 Pedestrian Detection with Cameras near Ground . . . 60
4.4 Lidar Performance . . . 61
4.4.1 Mapping . . . 61
4.4.2 Localization . . . 66
4.5 Summary . . . 68

5 Topology Aware Reciprocal Collision Avoidance (TARCA) **71**
5.1 Background . . . 71
5.1.1 Geometric Approaches to Collision Avoidance . . . 72
5.1.2 Limitations of ORCA . . . 76
5.1.3 Extension: Receding Horizon Control . . . 78
5.1.4 Extension: Forward Simulation . . . 78
5.2 Avoidance Topologies . . . 79
5.3 Forward Simulation Strategy . . . 80
5.3.1 TARCA . . . 80
5.3.2 Comparison to PHOP . . . 81
5.4 Evaluation Metric . . . 82
5.4.1 TARCA . . . 82
5.4.2 Comparison to PHOP . . . 83
5.5 Results . . . 85

5.5.1 Cooperation 85
5.5.2 Planning Horizon 87
5.6 Conclusions 87
5.7 Future Work 88
5.8 Summary 88

6 Conclusion & Future Work 89
6.1 Summary 89
6.2 Future Work 90

A Notes for Future Users of 1st Generation Platform 93
A.1 Hardware 93
A.1.1 Battery 97
A.1.2 Internal Electronics Bay 97
A.2 Software 100
A.2.1 Running the System 100
A.2.2 ROS Launch File Structure 104
A.2.3 Software Diagrams 105
A.2.4 Miscellaneous Software Notes 105

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

1-1 Some robots that operate in pedestrian-rich environments 20

2-1 Clearpath Jackal (unmodified) 25

2-2 Velodyne Lidar examples 28

2-3 RPLidar A2 29

2-4 Genius F100 Webcam [1] 30

2-5 Realsense R200 RGB-D Camera 31

2-7 Raw perception data 33

2-8 1st Generation Robot (with Sensor Hub) 34

3-1 Software Block Diagram 40

3-2 Particle Filter Localization [2]. Initially in the top left, red particles are spread throughout the map. As the robot drives (panels to right), particles concentrate. By final panel (bottom right), particles have converged to one pose. 42

3-3 Motion Primitives: blue lines extending from yellow robot represent all possible directions to consider traveling. 45

3-4 Pruned Motion Primitives: Blue lines only extend to first obstacle (purple, black regions). 47

3-5 Diffusion Map Transformation 48

3-6 Pure Pursuit Diagram [3] 52

3-7 Pure Pursuit Kinematics [3] 52

3-8 Differential Drive possible motions [4] 54

4-1 Sensor Plate Design 58
4-2 Lidar FOV 59
4-3 Realsense FOV 59
4-4 Sonar FOV 60
4-5 Realsense & Sonar 61
4-6 Maps Generated with Full and Limited Angular FOV (Velodyne) . . 62
4-7 4 RPLidar Positions 63
4-8 Map Generated with RPLidar at 0.9 m 64
4-9 Map Generated with RPLidar at 0.3 m 65
4-10 Map Generated with RPLidar at 0.1 m 65
4-11 Estimated Path (RPLidar at 0.1 m) 67
4-12 Localization Error (RPLidar at 0.1 m) 67
4-13 Estimated Path (RPLidar at 0.9 m) 69
4-14 Localization Error (RPLidar at 0.9 m) 69
4-15 Estimated Path (RPLidar at 0.3 m) 70
4-16 Localization Error (RPLidar at 0.3 m) 70

5-1 Collision Cone [5]: Gray area is set of blue agent velocities that lead to collision with stationary red agent 73
5-2 Velocity Obstacle [5]: Dark gray area is set of blue agent velocities that lead to collision with *moving* red agent 73
5-3 Velocity Obstacle [5]: Relative velocity $\boldsymbol{v}_A - \boldsymbol{v}_B$ within light gray region is equivalent to absolute velocity $\boldsymbol{v}_A$ within light gray region 73
5-4 Velocity Obstacle (Simplified): Relative position $\boldsymbol{p}_{ij}$ and sum of radii $r_i + r_j$ define VO with inward normal vector $\boldsymbol{n}_{ij}$. $\boldsymbol{d}_{ij}$ is the shortest vector from the relative velocity $\boldsymbol{v}_{ij}$ to the VO boundary. 74
5-5 Velocity Obstacle (Time Limited): VO from Fig. 5-4 is relaxed if colllisions more than τ seconds in the future are ignored. The VO no longer includes the small region near the origin. 75

5-6 ORCA (Multi-Agent): On the left, ORCA can avoid multiple agents simultaneously. On the right, the original set of feasible velocites is a circle with radius v_{max}. A single linear constraint is applied for each agent's VO, resulting in the slashed, convex polygon of safe velocities. 77
5-7 Velocity Obstacle (with Topologies): The boundary of the VO now consists 80
5-8 PHOP Average Reward for 3 Actions (0°,+20°,-20°) converges over time 84
5-9 Varying Host Agent Weight 85
5-10 Varying Time Horizon: 3 Algorithms Compared 86
5-11 Varying Radius: 3 Algorithms Compared 87

A-1 1st Generation Jackal (Side View) 94
A-2 KVM Switch 96
A-3 Velodyne Connectors 96
A-4 Jackal Sides Showing Connectors 97
A-5 Labeled Electronics Bay 98
A-6 USB Hubs 98
A-7 Jackal Sides Showing Connectors 99
A-8 Software Diagram: Perception on Brix 105
A-9 Software Diagram: Perception on Nuc 106
A-10 Software Diagram: Pedestrian Tracking 1/3 106
A-11 Software Diagram: Pedestrian Tracking 2/3 107
A-12 Software Diagram: Pedestrian Tracking 3/3 107
A-13 Software Diagram: Localization 108
A-14 Software Diagram: Costmap 108
A-15 Software Diagram: Planner (Static World) 109
A-16 Software Diagram: Planner (Dynamic World) 109
A-17 Software Diagram: Control (Static World) 110
A-18 Software Diagram: Control (Dynamic World) 110

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

2.1 Vehicle Specs. Only the Jackal can drive faster than human walking speed (1.3m/s). All of these robots are similar size. 27

2.2 Typical commercially-available sensors for autonomous systems. Price is correlated with range. 27

2.3 Lidar Specs. Velodyne VLP-16 costs almost 20x as much as the RPLidars, but its maximum range is much greater. The RPLidars have better minimum range though. 29

A.1 Power Consumption 95

A.2 USB Hubs 99

A.3 ROS Launch Files 104

A.4 Computer Connection Info 111

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

Autonomous (self-driving) vehicles are increasingly being tested on highways and city streets. But there is also a need for robots that can navigate through environments like sidewalks, buildings, and hallways. In these situations, the robots must interact and cooperate with pedestrians in a socially acceptable manner. The "rules of the road" no longer apply – there are no lanes or street signs, and pedestrians don't use turn signals when cutting through crowds. This thesis presents a robot capable of socially acceptable navigation.

This chapter provides motivation, literature review, and a concise list of the contributions in this thesis.

Chapter 2 describes the sensor suite and other hardware components of the first generation socially acceptable robot.

Chapter 3 describes the software system of the robot, including the Localization, Perception, Path Planning, Control, and Goal Selection systems.

Chapter 4 presents the design of a second generation robot with tighter budget and size constraints. It also presents results on the robot's ability to generate and localize to a map with different lidar sensors.

Chapter 5 presents a novel motion planning strategy that extends the VO framework to account for future impact of current planning decisions. This algorithm, Topology Aware Reciprocal Collision Avoidance, is compared to existing approaches, such as Optimal Reciprocal Collision Avoidance (ORCA), and Progressive Hindsight

Optimization (PHOP).

Chapter 6 concludes the thesis with a summary and some ideas for future work.

Appendix A presents very detailed explanation of the robot for someone. This resource is provided specifically for future users of the device. It includes labeled hardware diagrams, input/output diagrams for each software subsystem, and other general notes.

1.1 Motivation

Humans have a natural intuition for how to navigate through busy places like airports and malls, such as how much space to leave between one another and when to pass people that are moving slowly. Robots, on the other hand, are not good at this task. According to a recent news article [6], "Robots don't do well in crowds...robots are really awkward." Some existing approaches are presented in the next section, but in general, a robot that can drive quickly through pedestrian-rich environments is still an open research area.

There are many applications where a robot that can maneuver through crowds could be beneficial. For example, a robot that follows a person and carries heavy luggage would be make airport travel much more convenient. Or a robot that delivers packages or food to people's doorsteps would reduce the need for humans to do such a repetitive task.

This type of robot could be used in emergency situations where a medical team must navigate through a crowd of people or vehicles to assist a patient. In this scenario, the robot must carefully optimize for both speed and safety.

There is also an application in shared personal transportation devices like bike sharing. Nearly 60,000 people used the Hubway station at Mass Ave and Beacon St in Boston in 2015 [7]. However, these devices become unevenly distributed throughout the city each day. In the morning, the first customers drive into the city, and by the time the later customers arrive, there are no transportation devices left. Currently the customers are either out of luck, or the transportation company must manually

re-distribute the vehicles to the peak customer location. For example, Hubway drives 4-5 vans around the city from 6am to 10pm 7 days a week for this purpose [7]. If the bikes were able to drive autonomously, a customer could ride a bike into the city for work, and it could drive itself back to be available for someone else to use, instead of sitting idly until the end of the day when someone rides it back out of the city.

Existing autonomous vehicles do not solve the problem of personal and cargo transportation through crowded environments for several reasons. First, the robots typically use expensive sensors. While these sensors may be safety-critical on a vehicle that will drive on a highway at 70mph, they are not necessary and cost-prohibitive for this application. Second, many of the robots simply are not designed to carry both people and cargo. Third, the robots do not navigate with pedestrians naturally. So, a robot that uses inexpensive sensors, can carry people and cargo, and can navigate through crowded environments would be novel and could be used for the several applications mentioned earlier.

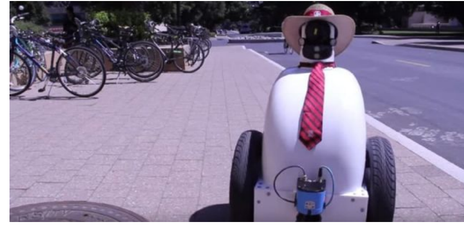
There is no published research into the impact of high-level design decisions on robot performance for this application. Researchers typically build one robot that suits their needs and adheres to their budget. But, nobody has analyzed how the performance of the system could be improved if the budget were increased or a sensor were placed in a new location. Further, with a limited budget, each remaining component must be used to its full potential. Therefore, a solid understanding of how the system's performance is affected by these early design decisions would be useful to anyone building a robot for autonomous navigation in the future.

A robot in a pedestrian-rich environment should behave appropriately for each type of obstacle. That is, a robot should take extra care to not interfere with a human, whereas a robot could get very close to another robot because the other robot can not get annoyed or frustrated like its human counterpart.

The robot should be aware of different human behaviors, such as where people are looking. The robot should leave extra room for people who are staring at their cell phones, as opposed to a person who is clearly aware of the robot's presence.

This ability to plan intelligently around different types of agents does not ad-

(a) Obelix [8] [9]



(b) Jackrabbot [10]

(c) MAGIC [11]

(d) Starship [12]

Figure 1-1: Some robots that operate in pedestrian-rich environments

equately exist in the literature, so a new motion planning algorithm is created in Chapter 5.

1.2 Literature Review

There are many companies and academic groups that are currently developing robots for use in public places with many pedestrians. Some are shown in Fig. 1-1.

1.2.1 Academic Research

A group at Stanford led by Savarese is developing a socially aware robot called Jackrabbot. Their approach learns social etiquette by observing humans. Since a large amount of data is required for machine learning, they propose methods of data collection [13] and a method for pedestrian path prediction [14]. The robot wears a necktie and sometimes a cowboy hat to look more "human." In videos published a couple months prior to the writing of this thesis, the researchers explain that the

robot is either driven manually or has a simple collision avoidance algorithm running onboard. There is also a collection of sensors including at least 2 Lidars, stereo cameras, and RGB-D cameras.

A group led by Burgard developed a robot for this environment called Obelix [9]. The robot recently drove several kilometers through downtown Freiburg, Germany autonomously, with an entire news crew following behind to document the event. The robot is 1.6m tall and has a maximum speed of 1m/s. Four lidars are used, as well as GPS, to map the area ahead of time. They present novel methods including calibration of the several sensors, and SLAM with such a large map. They track dynamic obstacles by comparing changes between 1s worth of successive occupancy grid maps. They monitor the IMU to limit the robot's speed if it is driving over a bumpy surface to prevent tipping over. Their planner has three layers: one to connect the local maps, one to run Dijkstra's algorithm on a local map, and one to determine a safe velocity given the dynamic obstacles in the area. The cost of each node in the local map is pre-computed as a sum of travelled distance and free space, and if the perception system deems an edge un-traversable during execution, it is assigned infinite cost. The robot's impressive trek through the city was fully autonomous, except for at crosswalks and once due to localization loss. Some stated limitations of their approach include navigating around pedestrians, sensing blind spots, and areas with few features for the laserscan-based localization strategy.

A group at University of Michigan led by Olson developed a robot called MAGIC that was shown driving through a hallway with several pedestrians. The path planning algorithm by Mehta et al. dynamically switches between three options to navigate through crowds: stop and wait, follow someone moving in the same direction, or go solo [15]. Further demonstrations show the robot driving in a grassy field with several people wandering around [16]. The MAGIC robot uses a VLP-16 lidar (described in Chapter 2) for perception.

A group at Duy Tan University in Vietnam developed a robot to help pedestrians cross streets, like a human crossing guard. The robot uses ultrasonic sensors to detect objects.

1.2.2 Commercial Robots

On the commercial side, a company called Starship has logged over 20,000km with their robot for local delivery of goods. Starting a few months before the writing of this thesis, the robots have begun making deliveries autonomously, with a person nearby for safety. Their public videos do not show the robot driving through pedestrian-rich environments.

Another start-up, Dispatch, has a ground robot for local delivery as well. Their sensing suite includes a VLP-16 lidar. Again, the limited public information and videos focus on the robot's delivery capabilities, and do not demonstrate performance in crowded areas.

Even Domino's Pizza is building a robot to replace their delivery drivers. The robot seems to still be in an early development phase by a military contractor.

1.2.3 Summary

Many of these robots have led to interesting publications that push forward the state-of-the-art in robotics. However, it is not clear from the demonstrations of these robots that they are truly capable of autonomous navigation in real, crowded environments. Often a robot is shown navigating in a laboratory setting next to the researchers that created it. In the case of Obelix, the distance travelled is impressive, yet the robot is driving on an empty sidewalk for a great deal of the journey [17]. The capability of the commercial robots is difficult to gauge, since the videos often focus on the robot's appearance with brief clips of the robot driving. The literature lacks a robot that has been clearly demonstrated to drive in a realistic environment autonomously for an extended period of time, at the average human walking speed.

In addition, many of the robots do not use commercially-viable sensor suites. Many use extremely expensive sensors, like the Velodyne VLP-16, or a combination of several Hokuyo lidars. The sensors alone on these devices cost close to $10,000. It has not been shown that a robot can still navigate autonomously at high speed with less expensive sensors.

Lastly, the utility of the robot depends on its ability to carry people or cargo. The academic robots are not designed with this application in mind, but the commercial robots certainly do.

1.3 Contributions

The main contributions of this work are

1. A thorough description of a fully autonomous ground robot that can drive through pedestrian-rich environments at the average human walking speed

2. An analysis of some performance trade-offs of a 2nd generation robot design with tighter size and budget constraints - specifically the ability to generate and localize to a map

3. A novel motion planning algorithm, TARCA, that considers the impact of passing other agents on different sides to achieve more socially acceptable navigation

4. Documentation for future users of the robot, including detailed software architecture diagrams, labeled hardware diagrams and operation instructions

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

Generation 1 Hardware

This chapter presents the hardware of the 1$^{\text{st}}$ generation socially acceptable robot. It describes the chassis, sensors and onboard computers.

2.1 Chassis

The robot is built on a Clearpath Jackal, pictured in Fig. 2-1, which is a differential-drive ground vehicle capable of driving on indoor and some outdoor terrain. This robot can be purchased for about $12,000.

The chassis contains 2 motors which drive the 4 wheels using a belt drive, meaning the speed of the wheels on each side of the vehicle can be commanded independently. There are encoders on the motors. There is also an internal tray to hold extra

Figure 2-1: Clearpath Jackal (unmodified)

computers, electronics, or for cable management. The Jackal provides an IMU and GPS but these are not very useful for this application. The IMU is too noisy and GPS is unavailable indoors, and inconsistent in urban environments.

There is a main circuit board which contains most of the control and power electronics for the robot. There is a mini-USB jack which can be used to connect any external computer. The Jackal simply shows up as a USB device on the external computer. There is also a user power board with 5, 12, and 24V lines which is useful for the various sensors that require these voltages.

Clearpath Robotics provides open-source ROS packages which handle the motor control, battery voltage monitoring, and other low-level tasks. This means the external computer must simply supply `std_msgs/Twist` commands through ROS containing a desired linear speed and desired angular speed. This is converted into wheel speed commands at the software level (described in Section 3.1.5), and then the robot's on-board electronics control the motors to those reference speeds using feedback from the encoders.

In experiments for this thesis, it became clear that speed is a critical aspect of a robot that navigates around humans. If the robot drives very slowly, people will do all of the collision avoidance and simply walk around the robot. That is both uninteresting (from a research standpoint) and not useful for the application of carrying cargo nearby a human. Since humans walk at an average speed of 1.2m/s [18], the robot must be able to drive at least this fast. The robot must carry computers, sensors, and a battery, so a large payload capacity is also a requirement.

Some alternatives to the Jackal are described in Table 2.1. All 3 robots are similar in size. The Jackal is the only one that can drive fast enough, and it has the highest payload capacity. It is also the more expensive than the other robots.

[1] The advertised max speed is 0.6m/s. By changing firmware parameters, it is possible to drive up to 0.9m/s, but the vehicle does not stop consistently above the advertised speed.

Table 2.1: Vehicle Specs. Only the Jackal can drive faster than human walking speed (1.3m/s). All of these robots are similar size.

	Clearpath Jackal	Pioneer P3-AT	Turtlebot2
Max Speed [m/s]	2.0	0.6[1]	0.65
Max Payload [kg]	20	12	5.0
Size [mm x mm x mm]	508 x 430 x 250	508 x 497 x 277	354 x 354 x 420
Price [$]	12,000	4,000	1,000

Table 2.2: Typical commercially-available sensors for autonomous systems. Price is correlated with range.

	Price [$]	Range [m]	Data Format
3D Lidar	8-60k	2-100+	3D Pointcloud
2D Lidar	500	0.15-16	2D Plane of Ranges
Realsense (RGB-D)	100	0.5-3	3D Pointcloud & RGB Image
Sonar	5-50	1-5	Single Range Value
Monocular Camera	20-500	N/A	RGB Image

2.2 Sensors

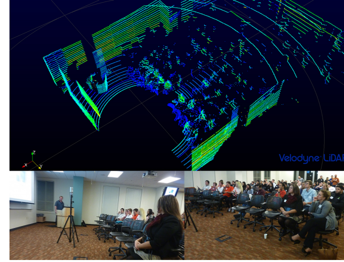
The robot uses sensors to perceive the surrounding world and make intelligent decisions about where to drive. There is no single sensor today that can measure every signal required for navigation. Thus, a suite of sensors is necessary.

The sensors on this robot are organized in Table 2.2 for convenience and comparison. Clearly, as the price of the sensor decreases, the range and quality of information provided by the sensor decreases as well. Each of the sensor types are described in the following subsections. At the end of this section, some other common sensors that do not suit this application are mentioned for completeness.

2.2.1 Lidar

Arguably the most important sensor in the recent advancement in autonomous vehicles is Lidar. Almost every company testing vehicles today has at least one Velodyne Lidar on their car, because of its field of view and robustness to changes in the envi-

(a) Different sizes of Velodyne Lidars [19]. The rightmost one, VLP-16 is used on the 1st generation robot.



(b) Example pointcloud from a 3D Lidar [20]

Figure 2-2: Velodyne Lidar examples

ronment like lighting and inclement weather. The sensor consists of 1 or more lasers spinning on an axis. The range is calculated by the known speed of light, divided by the time elapsed between the light being emitted and being reflected back to the sensor. If a single laser spins, a 2D laser scan is generated, with a distance measurement at many points along a plane. If multiple lasers are mounted at different angles along the spinning axis, a 3D pointcloud is created. The 3D pointcloud of high-end Lidar is close to a perfect representation of the entire scene (out to >100m), showing details as fine as people's hands. Both types of Lidar sensors are commercially available, with the 3D ones being much more expensive.

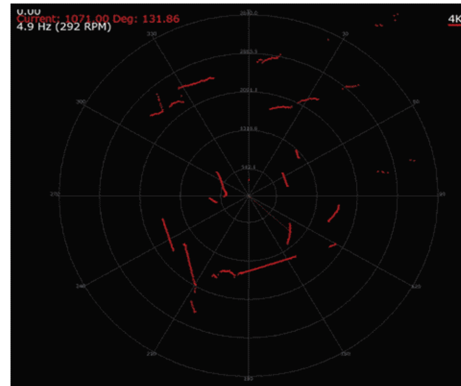
Lidar is useful for two common tasks for a robot:

- seeing walls, so the robot can estimate its position on a map, and
- seeing obstacles, so the robot can plan a path which avoids them.

Velodyne

For the socially acceptable robot, a Velodyne VLP-16 (also known as the Puck) is mounted in the center of the robot. This sensor has a horizontal field of view of 360°, a maximum range of 100m, and a ±15° vertical field of view. This means the Velodyne can see objects in 3D all around the vehicle out to 100m, which explains why it is such a popular sensor. A downside is the $8,000 price tag. Another limitation is the minimum range and vertical field of view, which creates a circular blind spot with radius 1-2m around the vehicle.

(a) Sensor is only 2" tall [21]



(b) Example laserscan from a 2D Lidar [21]

Figure 2-3: RPLidar A2

Table 2.3: Lidar Specs. Velodyne VLP-16 costs almost 20x as much as the RPLidars, but its maximum range is much greater. The RPLidars have better minimum range though.

	Velodyne VLP-16		RPLidar A2M4		RPLidar A2M6	
	Advertised	Observed	Advertised	Observed	Advertised	Observed
Max Range [m]	100	-	6	7.0	16	14
Min Range [m]	-	1	0.15	0.14	0.2	0.2
Frequnecy [Hz]	5-20	10	5-15	12.5	5-15	10
Horiz Resolution [°]	0.1-0.4	-	0.9	1.0	0.9	1.0
Weight [g]	830	>1200	190	200	190	200
Power [W]	8	-	2.3	-	2.3	-
Voltage [V]	9-18	12	5	5	5	5
Price [$]	8,000	-	450	-	560	-

RPLidar

An alternative Lidar sensor is produced by the Chinese company Slamtec. Its main advantage is its low cost of $450, nearly 20x cheaper than the Velodyne VLP-16. The compromise, however, is its sensing performance. The RPLidar only outputs a 2D laser scan, as shown in Fig. 2-3b. Other companies produce 2D Lidars for >$1,000.

The specs of the two Lidars are compared in Table 2.3. Note that the weight of the Velodyne is advertised with only the sensor, but the observed weight includes the required electronics box, and a custom aluminum mount.

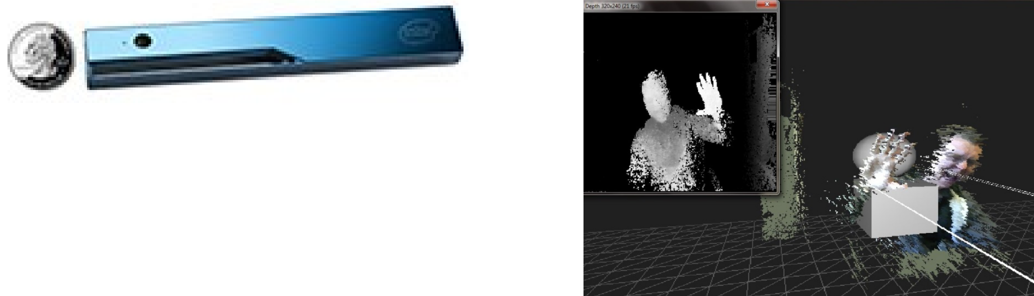
Figure 2-4: Genius F100 Webcam [1]

2.2.2 Monocular Camera

While Lidar is good at measuring distance to and shape of obstacles, it is not as useful for actually inferring what type of obstacle it is. For example, a person and a pillar do not look very different to a 2D Lidar that simply sees the cross section. But those two types of obstacles have very different implications for path planning, since the person might move over time, while the pillar is going to stay still.

The computer vision community has successful methods for this task, namely object classification. Given a camera's image, the objects in the image can be labeled. However, with a monocular camera (similar to a human using only one eye), depth cannot be perceived. The combination of monocular cameras with Lidar was covered thoroughly in [22], and a brief overview of the pedestrian detection strategy on this robot is presented in Section 3.1.3. Some object classification methods require the camera to be mounted horizontally, so the images are not too skewed. However, state-of-the-art machine learning algorithms can still classify objects when there is skew.

For the 1st generation robot, 4 Genius F100 Webcams are mounted (3 in the front, 1 in the back) to cover a large amount of the area around the vehicle. Their 120°



(a) Camera with US quarter for scale [23] (b) Example pointcloud from a Realsense [24]

Figure 2-5: Realsense R200 RGB-D Camera

field of view reduces the number of cameras needed, as compared to other typical webcams that have smaller fields of view. The price tag of this camera is $40 each - two orders of magnitude lower than the Velodyne Lidar.

2.2.3 Intel Realsense

While the cameras augment the Lidar by labeling the blobs it sees, there is still the issue of the Lidar's 1-2m blind spot around the vehicle. This is arguably the most critical region for a robot that avoids collisions with obstacles. In a static, unchanging world, one can assume that any obstacles will be seen by the Lidar when outside of the blind spot, and then remembered even if the robot gets very close. However, in the environments that the socially acceptable robot operates in, there are people moving around all the time, and that assumption is invalid. It is imperative to perceive the world in all directions that the robot would consider driving.

The blind spot is filled in by three Intel Realsense R200 cameras, which are RGB-D sensors. The Realsense calculate depth of points in the image and generates a 3D pointcloud by using infrared stereo cameras. There is also an infrared transmitter that projects texture to make the stereo registration problem easier. This is an enhancement on the monocular camera which has no depth information. The Realsense

(a) Top View of Bumper

(b) Side View of Bumper

R200 cost $100 each, and are very slim and lightweight. They are smaller and less expensive than the Microsoft Kinect, another common RGB-D sensor. There is also an RGB camera on the R200, which is not used in the 1st generation robot.

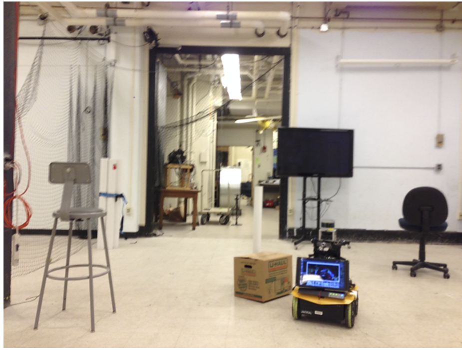
The sensors are mounted similarly to the webcams, but pointed slightly downward. The position is chosen so that the sensors can see both the front of the robot, and as far as the minimum sensing range of the Lidar. The R200 has a maximum range of 3-4m, although some users have seen ranges closer to 20m [25]. Regardless, because the sensor is angled toward the ground, the maximum range is sufficient. Furthermore, this mounting position makes filtering of the somewhat noisy depth measurement easier, because the floor is a known reference that can be assumed as a flat plane.

2.2.4 Bumper

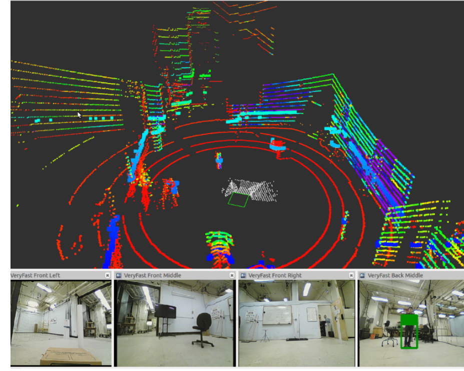
A custom bumper is designed for the front and back of the Jackal. It is shown in Section 2.2.4. This bumper contains touch sensors that act as an open/closed switch when relaxed/compressed. This is the final safety measure for the Jackal, because if any of the touch sensors are compressed, an emergency stop command is issued. This is in case there was a failure to detect an obstacle by other means.

The design shown in Section 2.2.4 includes an RPLidar (8m) and 4 touch sensors. The bumper also allows for mounting of other sensors such as sonars, or could be a place for Realsense cameras in the future. The advantage of a modular bumper is that it can be removed from the Jackal chassis, redesigned at low cost, and re-attached, as opposed to drilling many permanent holes in the vehicle.

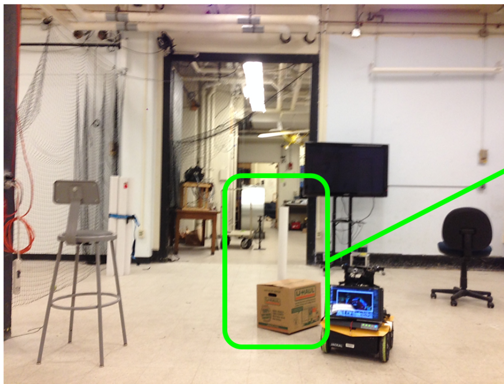
The bumper has an area for its own microcontroller (currently an Arduino Mega) which processes the touch sensors and outputs an emergency stop command if nec-

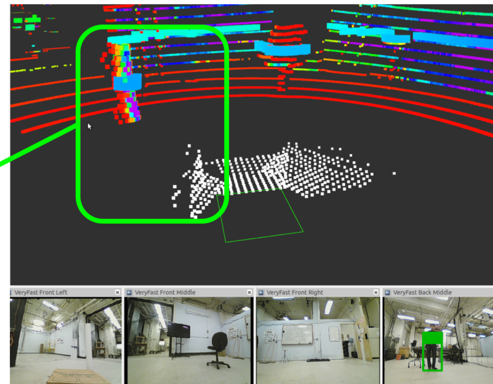


(a) Scene with obstacles at different distances from robot



(b) Sensor data of the scene in (a)





(c) Zoomed in: carboard box and white pole in green rectangles, seen by Realsense and Velodyne, respectively

Figure 2-7: Raw perception data

essary. The microcontroller could run ROS, using the open-source Arudino ROS software, but with the low amount of data being processed, it is easier to communicate simply with serial commands.

Theoretically, this bumper could be manufactured and sold to Jackal customers, as it is completely modular and requires only a USB connection for power. A significant portion of the bumper design was done by undergraduate Justin Chiu '18, as part of a UROP.

2.2.5 Sensor Hub

The sensors are arranged in such a way that maximizes each sensor's strength, while filling in blind spots. In the Sensor Hub (i.e., 1st generation) design, all of the sensors

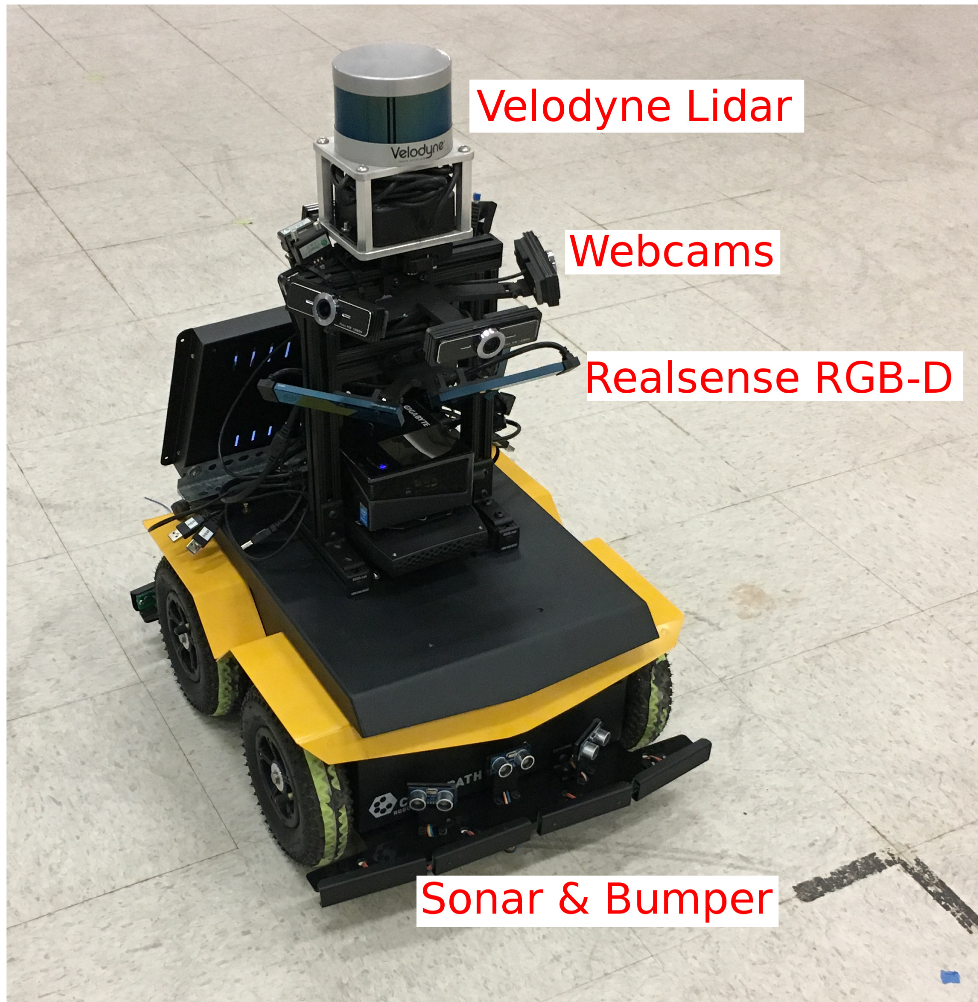


Figure 2-8: 1st Generation Robot (with Sensor Hub)

(except the bumper) are mounted in the center of the robot, as seen in Fig. 2-8.

The Velodyne is mounted about 0.6m above the ground in the center of the robot. The height is necessary for two reasons. First, if placed lower, some of the lasers would be angled such that they intersect with the robot, which is a complete waste of photons. Second, there is much more clutter near the floor, such as chair legs, feet, etc. and it is necessary to see the walls and other permanent features that help the robot recognize where it is on the map.

The Realsenses are mounted right below the Velodyne, angled downward so that they see the front of the robot, and the edge of the Velodyne's blind spot. A good example of this is in Fig. 2-7c's green rectangle, where the cardboard box is too close to the vehicle to be seen by Velodyne, but is seen by the Realsense, and vice versa for

the white pole that is slightly further away. In the image on the right, the deformed white pixels in the bottom corner of the green rectangle represent the cardboard box, and the cylinder of points near the top of the green rectangle represents the pole.

To keep the cameras relatively horizontal (not pitched up or down), they are also mounted on the Sensor Hub so their overlap is minimal, and coverage is maximal. There is also a camera facing backward to detect pedestrians that are behind the robot.

An advantage of the Sensor Hub design is a central structure containing all major sensors. There is space to mount computers discreetly as part of the Hub. A limitation of this design becomes clear when considering an application of the socially acceptable robot, a package carrier, since the main cargo area of the robot is occupied by sensors. This problem is addressed in the 2nd generation robot.

2.2.6 Other Sensors

There are of course other sensors than the ones mentioned in the preceding sections. Each have issues that limit their utility on this robot.

High-quality monocular cameras are becoming useful for state-of-the-art SLAM and collision avoidance methods, but they require a significant amount of computation and often a GPU. On this robot, the even the inexpensive webcam image is downsampled to reduce computation and USB data flow. So, a more expensive monocular camera would not necessarily improve performance for the current software architecture. The performance of monocular-SLAM algorithms is not yet very robust [26], compared to Lidar localization methods.

Stereo cameras are commonly used because they can measure depth in the image frame. The Realsense R200 is actually an IR stereo camera, but this paragraph focuses on RGB stereo cameras. A company recently released the Zed stereo camera[2] for $500, but it requires significant computation on the host computer. The Realsense, on the other hand, does the stereo registration on its own chip, easing the computation load on the host computer. The Zed camera is advertised to measure depth up to 20m.

[2] `https://www.stereolabs.com/`

Lidar, on the other hand, can measure up to 16m (albeit only in 2D) at the same price point with practically no computation required by the vehicle computer Finally, the drawback of all vision systems is the lack of robustness to changing lighting conditions.

Some autonomous vehicles use radar for obstacle detection and others use Lidar. Radar provides a much coarser measurement than Lidar (Fig. 2-2b) which is sufficient for large autonomous vehicles that spend the majority of their time driving in the same direction as other vehicles, and simply need to maintain a safe following distance. On the other hand, a robot that navigates through pedestrian-rich environments requires very fine detail about the objects around it, because people walk in every direction. A robot needs an accurate estimate of the pedestrians' positions and velocities to navigate with the strategy presented in this thesis. Also, there aren't many radar systems for this type of robot that are commercially available today. This level of accuracy and availability makes Lidar a better choice for this application.

2.3 Computers

There are 2 external computers used (in addition to the Jackal's onboard control unit). One requirement for the object recognition software is a discrete graphics card (GPU), and state-of-the-art computer vision algorithms often require a GPU manufactured by NVIDIA. Along with obvious size and power limitations, the GPU constraint severely limits what computers are useful for the socially acceptable robot.

One such computer is the Gigabyte BRIX Pro GB-BXi7G3-760 with an i7-4710 and GTX760 GPU. This computer's size (60 x 128 x 116mm), discrete GPU, and internal power electronics make it a great choice for this robot. The Brix costs about $800 in total. However, the company has stopped making computers of this form factor.

Another popular computer for mobile robots is the Intel Nuc. A recent version of this is called the Skull Canyon, consisting of a i7-6770HQ CPU and powered USB3.0 ports. However, there is no discrete NVIDIA GPU on the Nuc.

For the 1st generation robot, a Brix and a Nuc are mounted on the outside of the

chassis. This position handles heat management by default, instead of cutting holes into the Jackal's chassis and installing fans to keep the temperature at a safe level for the computers. The Brix runs most of the software, while the Nuc runs the Realsenses and has plenty of computation left for future expansion. The two computers are connected with an Ethernet cable. The Brix also does all communication with the Jackal hardware, and is called the ROS Master[3].

Another option considered is a computer built on a mini-ATX motherboard with desired components. The computer could go inside the chassis' computer tray, as the Jackal was initially designed. However, the size of the power supply needed to power a modern GPU and other components is the same size as the Brix, so the Brix+Nuc option is preferable at this time.

Other computation options are explored for the 2nd generation robot, including a laptop and other GPUs.

2.4 Summary

This chapter described the hardware of the 1st generation robot. The chassis, sensors, and computers were presented.

[3]Information about running ROS with multiple machines is found at `http://wiki.ros.org/ROS/Tutorials/MultipleMachines`. The two computers can run different versions of Ubuntu and ROS (i.e. 14.04/Indigo on Brix and 16.04/Kinetic on Nuc)

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

Software

This chapter presents the software on the socially acceptable robot. The backbone of the software is the Robot Operating System (ROS), which is an open-source library for C++ and Python. It is mainly used to handle communication between different files and computers. Some of the best features of ROS are

1. the extensive library of open-source packages
2. the ability to write C++ and Python, and have these files communicate
3. launch files, so many scripts can run at once with the proper parameter

For a brief explanation of ROS terminology, files are called `nodes`, which can either `publish` or `subscribe` to `topics`, which are pieces of information like sensor data or variable estimates. A much more thorough description is found online[1].

3.1 Software Architecture

The system is designed to convert inputs,

- Task (e.g. "Follow Person A", "Go To Position X", etc.)
- Sensor Data

into wheel commands.

[1] `http://wiki.ros.org`

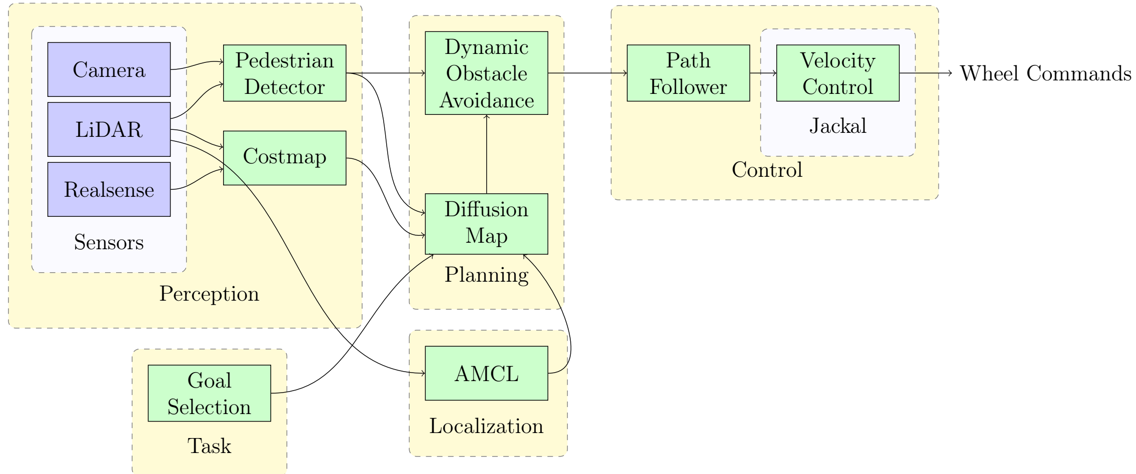


Figure 3-1: Software Block Diagram

3.1.1 Block Diagram

The block diagram in Fig. 3-1 shows the main components of the socially acceptable robot's software architecture.

3.1.2 Localization

Localization is the problem of inferring where an object is in the world. For the socially acceptable robot, the problem is to estimate the robot's pose (position & orientation) in a map. A map of the environment is made ahead of time, and then during execution, the robot uses Lidar and odometry to estimate where in that map the vehicle is.

The ROS package `AMCL`[2] is used for this task. AMCL implements a particle filter for localization. Initially, a set of particles is distributed throughout the map, which represent each of the possible poses the robot could have. The algorithm executes an update step, where for each particle, the current 2D laser scan (from the Lidar) is compared to what the map says the robot should see at that pose. Particles where the laser scan matches well with the map are kept, and the remaining particles are discarded. The probability distribution of poses goes from uniform across the map, to a series of peaks in areas where the robot is likely to be.

[2] `http://wiki.ros.org/amcl`

As the robot drives, the wheel encoders report odometry estimates, roughly corresponding to the robot velocity. These estimates have some noise associated with them, indicated by their covariance. With each new odometry estimate, the existing particles are projected forward according to the odometry, and new particles are added around the existing particles to account for noise. The higher the covariance in the odometry estimate, the more the new particles are spread out.

Thus, the particles are spread out by the odometry, and narrowed down by the laser scans, in a repeating loop. After several seconds of driving, particles will hopefully converge, and only exist near the robot's true position on the map.

This approach is useful because it handles the uncertainty associated with the sensors. A limitation of it is the possibility of narrowing down to the wrong set of particles, because there is no way to recover except re-sampling the whole map again. This limitation is rarely seen in experiments conducted for this thesis. There are existing approaches to recognize when the localization procedure has converged to the wrong position, but those are not discussed here.

The main parameters that can be tuned in `AMCL` are the 4 `odom_alphas` that set how much particles spread out. If they are set too low, the algorithm trusts the wheel encoders too much, and will not have diverse enough particles to select from. Over time, the robot's estimated pose may drift from the true pose, and there will not be diverse enough particles to correct for it. Conversely, if the parameters are set too high, two issues arise. First, a spurious laser scan could cause a particle far from the vehicle to appear as the most likely robot position, causing a jump in the vehicle's pose estimate. While spurious laser scans occur independently of the choice of this parameter, the impact of a bad measurement would be less significant if all of the particles were limited to a small region than if the particles were spread out. Second, the particles will never converge to a small area around the vehicle, because they will spread out faster than the laser scan can narrow them down. This makes it difficult to accurately estimate the vehicle's pose, which is required for other tasks as indicated in Fig. 3-1.

The procedure is visualized in Fig. 3-2. At the beginning (top left), red particles

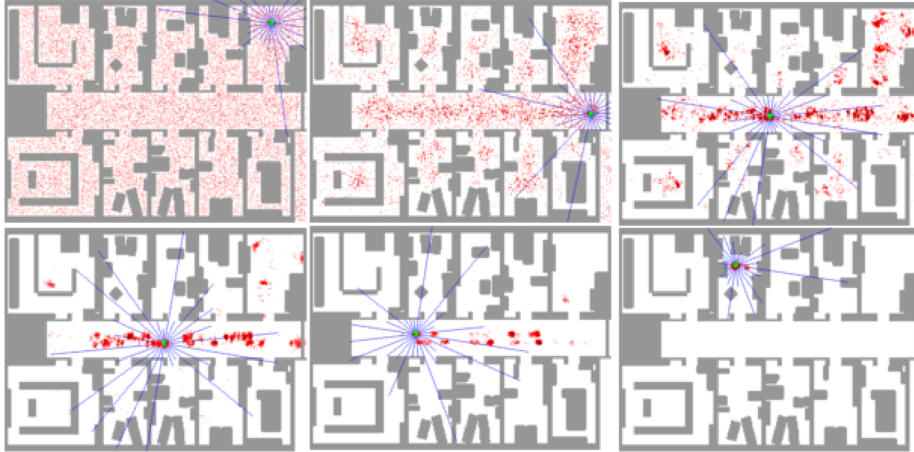


Figure 3-2: Particle Filter Localization [2]. Initially in the top left, red particles are spread throughout the map. As the robot drives (panels to right), particles concentrate. By final panel (bottom right), particles have converged to one pose.

are evenly spread out around the map. The blue lines represent range measurements (like a very discretized Lidar). As the robot moves, the red particles pool in certain regions. Near the end (bottom middle), there are only a handful of particle groups. Finally (bottom right), the robot is well-localized in the map since all particles are near the robot.

3.1.3 Perception & Costmap

Each sensor has its own data format (e.g. point cloud, laser scan, range measurement). To combine all of these into one coherent structure, an occupancy grid map is used. A ROS package called `costmap_2d`[3] implements the occupancy grid map. A single parameter file defines all of the sensor types and formats for the costmap node.

Note that the `costmap_2d` package has many detailed features that are not used for this application, such as probability (cost) in each grid cell, unknown space, obstacle persistence, etc. For this planning architecture, it is only important that a grid cell is occupied or free.

[3] `http://wiki.ros.org/costmap_2d`

Pedestrian Detection

As mentioned briefly in Section 2.2, and thoroughly in [22], pedestrians are detected by a combination of Lidar and camera data.

A 2D laser scan (either from a true 2D Lidar, or by extraction of a 2D slice from a 3D Lidar pointcloud) undergoes a few operations to begin this procedure. First, the scan is filtered against the map, so that points that line up with walls and pre-known static objects are discarded from consideration as pedestrians. Then the Dynamic Means clustering algorithm [27] is applied to the remaining points, which finds clusters in the data according to tunable birth/death rates, minimum cluster size, and other parameters. This list of clusters is now just a list of blobs that could be a wide range of things, like chairs, desks, people, or even other robots.

Simultaneously, the Very Fast pedestrian detector [28] operates on the camera images to find bounding boxes around people. The bounding boxes are transformed from the image plane to angles around the robot using geometric constants, like angles and positions of sensors.

The two processes are then merged. Each cluster is assigned a likelihood of being a pedestrian, based on a calculation of how well a person in the image plane lines up with the cluster position. These likelihoods are updated each time step, so over time clusters that never align with pedestrians in the image frame will have very low likelihood values, and vice versa for clusters that are very likely pedestrians.

Above a certain likelihood threshold, and with the requirement that the cluster has some velocity (because people who are not moving are just normal static objects), a list of pedestrians is generated. This list of pedestrians is continually growing and shrinking as people move in and out of the robot's surrounding area.

Some useful data are also measured by the pedestrian detection subsystem. The size (maximum distance between points in the cluster), position (mean of cluster points), and velocity (derivative of cluster position over time) are easily calculated. Further, a list of past positions is kept for each pedestrian, which can be used to predict future motion. Each of these pedestrian data are used in the path planning

procedure.

3.1.4 Path Planning

The path planning strategy is what makes this robot socially acceptable. While perception, control, and localization are necessary to sense obstacles, follow paths, and understand position in a global sense, the motion planning framework is the central area of innovation that allows this robot to drive at such high speed in pedestrian-rich environments.

Many existing motion planning algorithms assume the world is static. This is obviously not the case in environments with many pedestrians and other dynamic objects, but it turns out the assumption is still valid if either the robot or the obstacles are moving much faster than one another. To consider the two extremes, imagine a robot that drives at a snail's pace through a crowd. The people will generally move around the robot to get to their destinations. So, the robot can simply plan its path ignoring the pedestrians because they will be long gone by the time the slow robot actually gets close to them. On the other hand, an fast, agile drone flying 100mph through a crowd of pedestrians can treat the people as static obstacles, because their speeds are so slow compared to the drone's.

The regime that is even more interesting to study is that of the robot and dynamic obstacles which have similar speeds. For a robot that is designed to drive alongside humans, this is the exact regime of operation.

Path Planning Overview

In this robot's motion planning strategy, a fixed set of motion primitives is generated ahead of time. Each motion primitive is a short path, consisting of several points spaced out along a direction starting at the vehicle and ending a few meters ahead. The motion primitives are all of the possible paths that the robot will consider driving. Fig. 3-3 shows the blue motion primitives emanating from the vehicle.

During execution, the planner operates in three main steps in a Receding Horizon

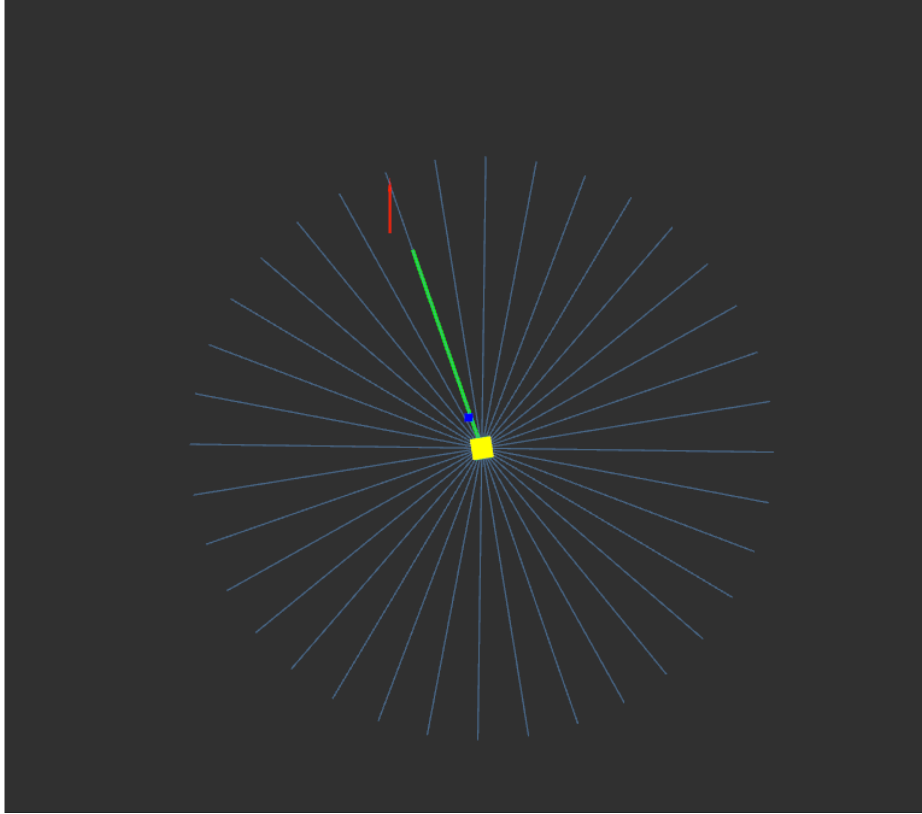


Figure 3-3: Motion Primitives: blue lines extending from yellow robot represent all possible directions to consider traveling.

Control (also known as Model Predictive Control) framework. In this framework, a detailed plan for a short distance in front of the vehicle is generated, and the remaining distance to the global goal is estimated, known as the cost-to-go. The RHC framework fits nicely into systems with a limited sensing horizon, because there is little reason to plan a path all the way to a goal position miles away if the robot can only see a few feet ahead. The robot chooses an action with the intention of executing it indefinitely, but the planning cycle repeats so that each action is only truly executed for a fraction of a second. In this way, the planning horizon is always moving forward along with the robot, giving rise to the name Receding Horizon.

The RHC implementation on this robot in three steps. First the robot finds all of the dynamically-feasible motion primitives that do not collide with static obstacles. Then it selects a subgoal a few meters ahead of the robot, choosing from the safe motion primitives with the lowest cost-to-go. Finally, it uses the information about

nearby pedestrians and its goal to select a desired heading angle and speed. Each of these steps are described in the following subsections.

Static Motion Primitive Pruning

The motion primitives are straight lines, evenly spaced every 10°. Other existing motion planners use motion primitives that are known to be dynamically feasible using a vehicle dynamics model, either ahead of time [29] or online. As it will become clear, this dynamic feasibility constraint is helpful for ensuring that the robot does not bother considering paths that it can't follow. As later described in Section 3.1.5, for example, the robot can't drive straight laterally without first turning in place. However in this implementation of the robot, the motion primitives are limited to straight lines in order to more easily interface with the dynamic obstacle planner. This limitation is sure to be addressed in future work.

The next step in narrowing down which of the paths is the best is called pruning. The vehicle's dimensions are projected along the motion primitive, simulating how much space the vehicle will occupy if it does indeed drive down a particular motion primitive. Starting at the vehicle's current position, the future positions of vehicle are queried on the costmap (described in) to check for collision with obstacles. If at any point a future position collides with a position that is considered occupied, the motion primitive is pruned, or cut, at that point, and the algorithm moves onto the next motion primitive. If there is no collision along the entire motion primitive, then there is no need to prune. After iterating through each of the primitives, a set of primitives that only extend as far as is safe to travel without collision with static obstacles has been generated. Fig. 3-4 shows the motion primitives overlayed on the costmap, where black areas represent occupied space, white is free space, and it can be seen that the blue primitives only extend to the nearest obstacle.

The next step is to determine which of these pruned motion primitives is actually beneficial in terms of getting the robot to its goal position. The metric that is used is Diffusion distance, from [29], which approximates the distance between any two positions in a known, static map. As opposed to simply using Euclidean distance, which

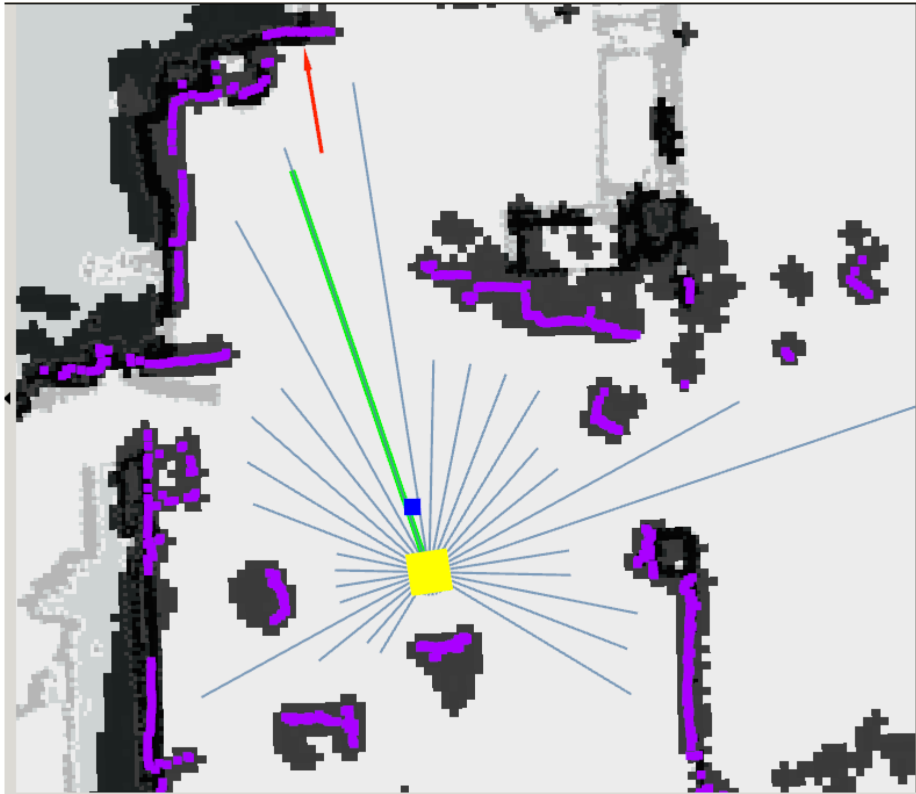
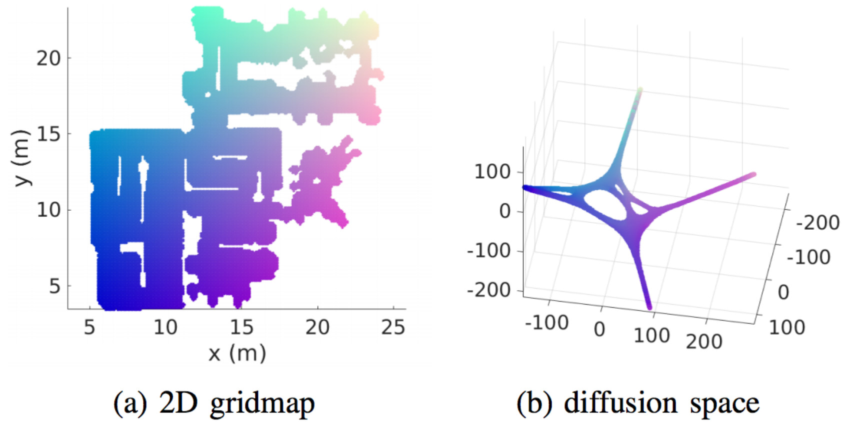


Figure 3-4: Pruned Motion Primitives: Blue lines only extend to first obstacle (purple, black regions).



(a) 2D gridmap (b) diffusion space

Figure 3-5: Diffusion Map Transformation

is not a good indicator of path length when there are walls and strange floorplans, the Diffusion distance takes into account these features to give a better cost-to-go estimate. This concept is depicted in Fig. 3-5 from the paper by Chen et al. where points in the occupancy grid map are transformed into a new Diffusion space that captures the effect of walls on path length between any two points in the map.

In a motion planning strategy that simply treats the world as static, the point in the set of pruned motion primitives with lowest diffusion distance to the final goal would be selected. The performance of this strategy is pretty good, as described in the beginning of this section, but only in the regime where the robot is driving very slowly or there are not many pedestrians. This is because at every planning cycle, the path that gets the robot closest to the goal is selected, and over time this leads to a path that tends toward the goal.

In this robot's strategy, though, the output of the static motion planner must consider the algorithm that handles dynamic obstacles. If the lowest Diffusion distance path is chosen, there is no flexibility to avoid dynamic obstacles. Conversely, if all of the pruned motion primitives are given as options, there is no way for the dynamic obstacle planner to know which way will actually get the robot to its goal position.

Therefore a set of motion primitives is sent to the dynamic obstacle planner with the following criteria:

- collision-free with static obstacles

- have a strictly decreasing diffusion distance along the path
- have a pruned length of at least several meters

These criteria ensure a few conditions. Regardless of what the dynamic obstacle planner decides is the best path, it is guaranteed to be safe, because it can't select from collision-causing paths[4]. In addition, the robot will make progress toward the goal because the diffusion distance is decreasing. Finally, the robot will not consider paths that drive it directly toward nearby obstacles, because those primitives will be pruned below the critical length.

Motion Planning around Dynamic Obstacles

The dynamic obstacle planner is the subject of two papers by Chen et al. [30], [31]. The planner in those works is named Collision Avoidance with Deep Reinforcement Learning (CADRL), and a Socially Aware extension, called (SA-CADRL). In this thesis, the details of CADRL are omitted; it is described at a high level and mainly considered as a system with inputs and outputs.

Briefly, at the core of CADRL is a policy that is generated with Reinforcement Learning on simulated data of agents who have random start and goal positions. The global objective is to develop a policy that minimizes the time to goal for every agent. The value function represents the time until the agent reaches its goal, based on its position, speed, heading angle, and those same states of the other agents in the world. While each of those states can be measured, there are also hidden states such as other agents' goals that can not explicitly be measured. Since this value function has a very high number of states, a deep neural network is used to store the many parameters.

Though it takes several hours to train the network to learn an optimal policy, querying the value function is extremely fast - on the order of 10ms. The robot can therefore calculate the expected time to goal if it takes any of its safe actions that were determined earlier in real time. It will choose the action with highest value, and will send that action to the controller to execute.

[4]Pedestrians within 1m of the vehicle are considered static obstacles so that this safety guarantee holds.

A benefit of the two-part planning strategy is that only actions that are guaranteed to not collide with static obstacles will be considered by the value network query operation. Therefore the dynamic planner need not know about static obstacles explicitly. This is a convenient trick for the current version of the dynamic planner that does not handle static obstacles, but in future work the planner will be expanded. It is hypothesized that a planner that knows more about the environment will perform better than one which is given only information about moving objects.

The SA-CADRL planner adds social norms into the training phase, so that agents are penalized for passing one another on the wrong side. For example in the US, pedestrians walking in opposite directions pass on each person's left, and when walking in the same direction, the faster person passes on the slower person's left. This subtle penalty on simulated agents was shown to generate policies that follow the social norms.

The combined static and dynamic path planners operate on the socially acceptable robot in real time. The robot was shown successfully navigate autonomously at human walking speed for nearly 1 hour without human intervention.

3.1.5 Control

There are two controllers running on the robot. The first calculates desired linear and angular speed from a desired position or path, and the second controls the actual wheel speeds to match the desired linear and angular speeds.

Point Tracking

The CADRL planner outputs a desired heading angle and speed $(\psi_{desired}, v)$ at each planning step (10Hz). Since the velocity controller accepts linear and angular *speed*, the outer-loop controller outputs the linear speed exactly, and does proportional control on the heading angle to output an angular speed. So, at each control step (100Hz),

the controller calculates

$$\begin{cases} \dot{x} = v & (3.1) \\ \dot{\psi} = k \cdot (\psi_{desired} - \psi_{current}) & (3.2) \end{cases}$$

and sends $\dot{x}$ and $\dot{\psi}$ to the velocity controller. $\psi_{current}$ is estimated using the localization method described in Section 3.1.2.

Path Following

An alternative control strategy is used when the planner outputs a path, which is a list of positions. This controller was used with a previous version of the planner, before CADRL. That planner is described in [32]. This path following control strategy is from [3]. The performance of this controller is sufficient for path following at the relatively low accuracy required for collision avoidance.

To summarize, at each control step (30 Hz), the algorithm finds the point on the path that is at most L_1 away. If nothing is found, the closest point on the path is used. The speed V, L_1 distance, and angle η define the lateral acceleration a_s,

$$a_s = 2\frac{V^2}{L_1}\sin\eta \tag{3.3}$$

according to the diagram in Fig. 3-6. This lateral acceleration defines the vehicle's turning rate, as seen in Fig. 3-7. The vehicle's speed is whatever the path generator decides (typically 1.2m/s for jackal) - the path follower isn't modifying the speed.

If the vehicle is moving (positive speed), the L_1 distance is extended to be proportional to the speed ($L_1 = L_{1_{gain}} \cdot V$), instead of the default $L_{1_{nom}}$. This means the vehicle will look for nearby points when stopped, to try and get onto the path quickly, but look far ahead when driving fast, to avoid oscillating on each side of the path.

The only parameters to tune are therefore $L_{1_{nom}}$ and $L_{1_{gain}}$. To tune parameters:

- $L_{1_{nom}}$ is set close to the vehicle. Increasing its value causes more corner-cutting,

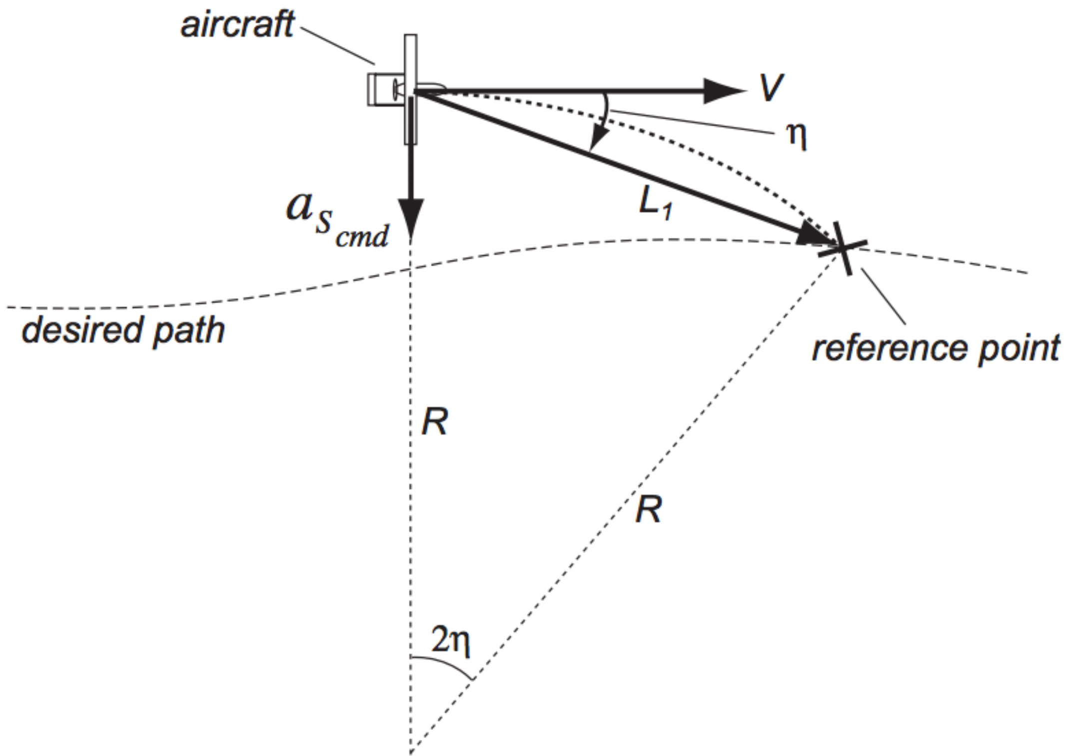


Figure 3-6: Pure Pursuit Diagram [3]

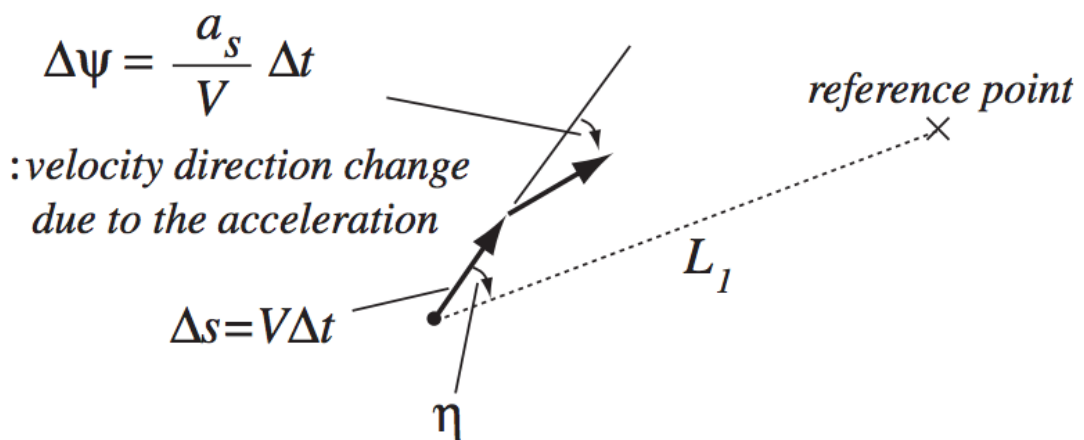


Figure 3-7: Pure Pursuit Kinematics [3]

and decreasing its value causes less progress along the path when starting from a stop.

- Increasing $L_{1_{gain}}$ causes further lookahead distance when driving. Since $L_{1_{gain}} = \frac{L_1}{V}$, $L_{1_{gain}}$ is the maximum distance the robot could travel while not on the path, divided by the maximum speed. So if the robot is going to drive fast, the designer can lower $L_{1_{gain}}$ for a given desired L_1.

Velocity Control

The velocity controller takes in a desired linear and angular robot speed, $(\dot{x}, \dot{\psi})$, and calculates the angular speed of the right and left wheels, $(\omega_{left}, \omega_{right})$. This control is directly done by the ROS package `diff_drive_controller`[5], but it is useful to describe how it works.

Because the two wheels must rotate around the center of rotation at the same angular speed [33], $\dot{\psi}$, the linear speed of each wheel is defined as

$$\begin{cases} v_{left} = \dot{\psi}\left(R_{ICC} - \frac{d_{wheel_sep}}{2}\right) & (3.4) \\ v_{right} = \dot{\psi}\left(R_{ICC} + \frac{d_{wheel_sep}}{2}\right) & (3.5) \end{cases}$$

using the distance to the Instant Center of Curvature for the rotation, R_{ICC}, and the constant distance between wheels, d_{wheel_sep}.

Using the standard relation $v = r\omega$, (in the form of $\dot{x} = \dot{\psi}R_{ICC}$ and $v_{left} = w_{left}r_{wheel}$), the two wheel angular speeds, ω_{left} and ω_{right} are calculated as

$$\begin{cases} \omega_{left} = \frac{1}{r_{wheel}}\left(\dot{x} - \frac{\dot{\psi} \cdot d_{wheel_sep}}{2}\right) & (3.6) \\ \omega_{right} = \frac{1}{r_{wheel}}\left(\dot{x} + \frac{\dot{\psi} \cdot d_{wheel_sep}}{2}\right) & (3.7) \end{cases}$$

where r_{wheel} is the constant wheel radius and $\dot{x}$ is the desired linear speed.

Some possible motions of a differential drive robot are depicted in Fig. 3-8. Clearly,

[5] `http://wiki.ros.org/diff_drive_controller`

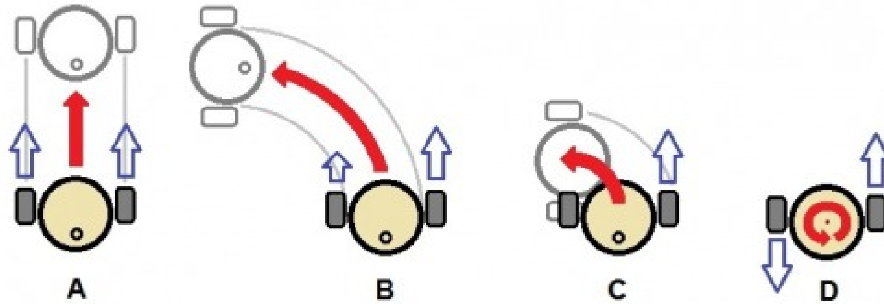


Figure 3-8: Differential Drive possible motions [4]

the robot can track arcs of different radii, as well as straight and turn-in-place maneuvers. It cannot, however, move laterally without turning first.

3.1.6 Task & Goal Selection

As described in Appendix A, the goal for the robot can be selected in several ways, namely:

- a specific point on the map
- a random selection from a predetermined set of points on the map
- a specific selection from a predetermined set of points on the map
- the position of a particular person

The first three options are straightforward. A goal position is chosen and the robot drives toward this static goal until it arrives. The fourth option is more interesting and leads to the "Follow Me" mode of operation.

"Follow Me" Mode

This mode is a culmination of the many software systems described so far. A prototype of this behavior is demonstrated in video clips.

The person to be followed stands in front of the robot, and an operator looks for that person's ID on the computer monitor. After typing in the ID, the algorithm uses the corresponding person's position estimate (from the mean position of the cluster of points in the laserscan labeled as the person) to calculate a goal position over time. Instead of the exact position of the person, the goal is set to be 2m behind the person

(along the line between the person and robot), so that the robot does not get too close to the person it is following. There are two reasons for the 2m buffer: the robot will lose track of the person (due to the 2m blind spot of the lidar) if it gets too close, and a robot that is always trying to drive right up to someone's feet looks unnatural. It looks much better when the robot stops smoothly, leaving some personal space for the person it is tracking. The dynamic goal automatically moves as the person walks around, and the robot is able to follow along as long as the person walks at a reasonable speed (below 1.2 m/s).

One obvious limitation of this system is the dependence on pedestrian ID. The issue with following a single pedestrian ID is that if a person is occluded briefly, they could be re-clustered under a new ID upon reappearance. Or, if two pedestrians cross paths, their IDs sometimes swap because the clustering algorithm has no motion model. To alleviate the first issue, if the pedestrian ID no longer exists in the list of pedestrians, the algorithm looks through the positions of existing pedestrians and if there is one that is close to the last known position of the lost pedestrian, that nearby pedestrian takes over as the one that will be tracked.

This mode of operation was designed with the constraint that the person to be tracked need not explicitly communicate its position to the robot. Many person-following robots require the user to wear a wristband that relays information back to the robot [34].

To further improve robustness with this constraint, future work on this mode of operation will

- include an automated method to convey to the robot that a certain person should be followed, such as a gesture, an app to begin the service, etc.
- include computer vision techniques to maintain a set of features unique to the person to be tracked, which will improve the algorithm's ability to correctly re-gain a person it has lost sight of, and to correctly choose between multiple people that have nearby positions

The two pieces could, for example, be combined so that a person walks up to the device, uses their phone to pay for the service, and during this time, the cameras

build up a set of features that can be re-used to disambiguate between pedestrians.

The "Follow Me" mode can be applied to any human-support task, where the robot must follow the human autonomously. Some examples are a robot that carries packages while a delivery person guides it from destination to destination, or a robotic cameraman, which films a person as they try new skateboard tricks.

3.2 Summary

This chapter described the software system of the robot, including the Localization, Perception, Path Planning, Control, and Goal Selection systems.

Chapter 4

Generation 2 Robot

This chapter presents the 2nd generation robot that meets new budget and size constraints.

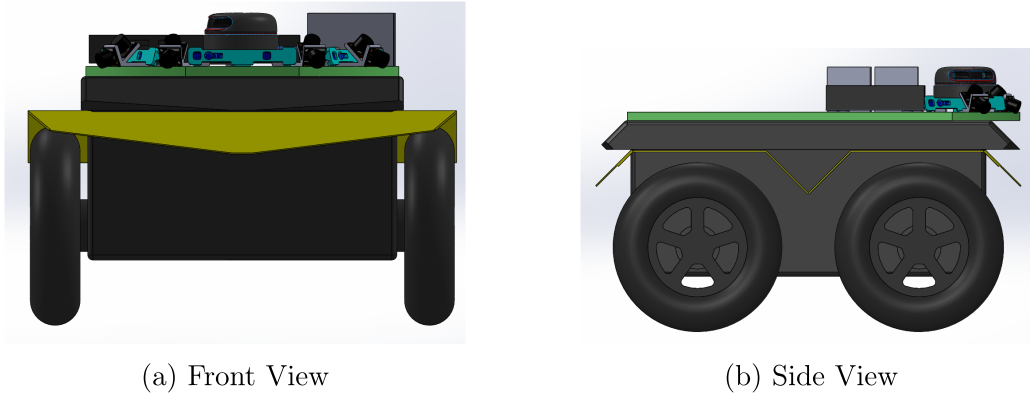
4.1 Constraints

While the robot presented in Chapters 2 and 3 meets the performance requirements of a robot for socially acceptable navigation, it has three major limitations that are addressed in this chapter.

The first is the position of the sensors. If the robot is to carry a person or some other cargo, the sensors must be placed in positions that still leave space for the cargo. The 1st generation robot has all of the sensors in the middle, and while they are centrally located for maximum perception coverage, there is no space left for cargo.

The second is the cost of the sensor package. The $8,000 Velodyne VLP-16 is simply not economically practical for this robot. The $100 Realsense, $30 sonars, and $500 RPLidar are much more reasonable price points. The four $40 webcams can also be removed if the Realsense are mounted accordingly, because the Realsense contains an RGB camera as well.

Third, the sensor and computation package for the 1st generation robot is very intertwined with the chassis of this particular robot. If the chassis of the robot were



(a) Front View (b) Side View

Figure 4-1: Sensor Plate Design

modified, many auxiliary components (like USB hubs, cables, etc.) would need to be removed and re-positioned on the new robot. In the 2nd generation design, the sensor and computation package will be a self-contained unit that only requires 2 types of connections to the robot:

- USB to control wheel speed
- 5, 12, & 19V power connections

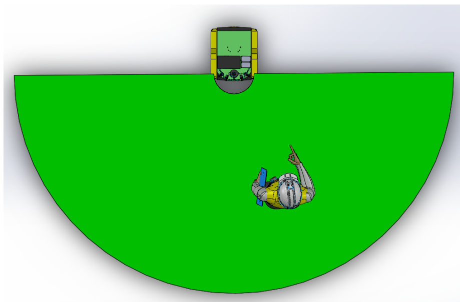
The power electronics can easily be incorporated into the vehicle's MCU board that controls the motor speeds, so this is a very reasonable requirement. The battery is assumed to be included in the robot chassis along with other essentials like motors.

4.2 CAD Models

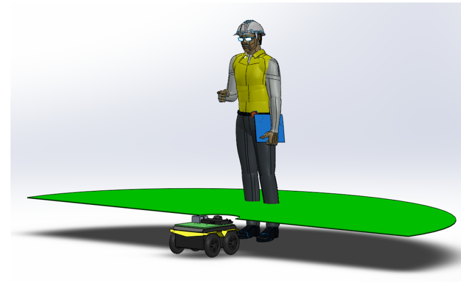
Several designs were created to satisfy the new constraints. One design, known as the sensor post, serves as both a handlebar for passengers to hold onto, as well as an elevated structure to mount sensors. Another design, known as the sensor plate, compacts all computers and sensors into a thin plate only a few inches thick, spanning the entire base of the robot. This allows unobstructed cargo space on top of the plate.

4.2.1 Sensor Plate Design

The front and side view of the sensors are shown in Fig. 4-1. The thin height is one of the definining features of this design.

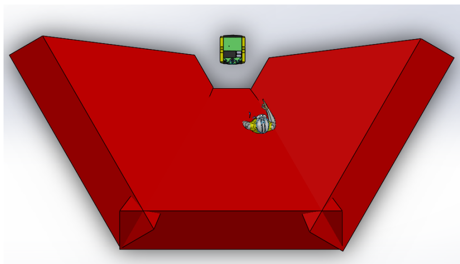


(a) Lidar Top View

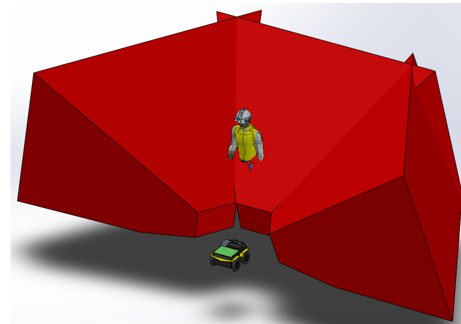


(b) Lidar at shin height

Figure 4-2: Lidar FOV



(a) Realsense Top View



(b) Realsense FOV

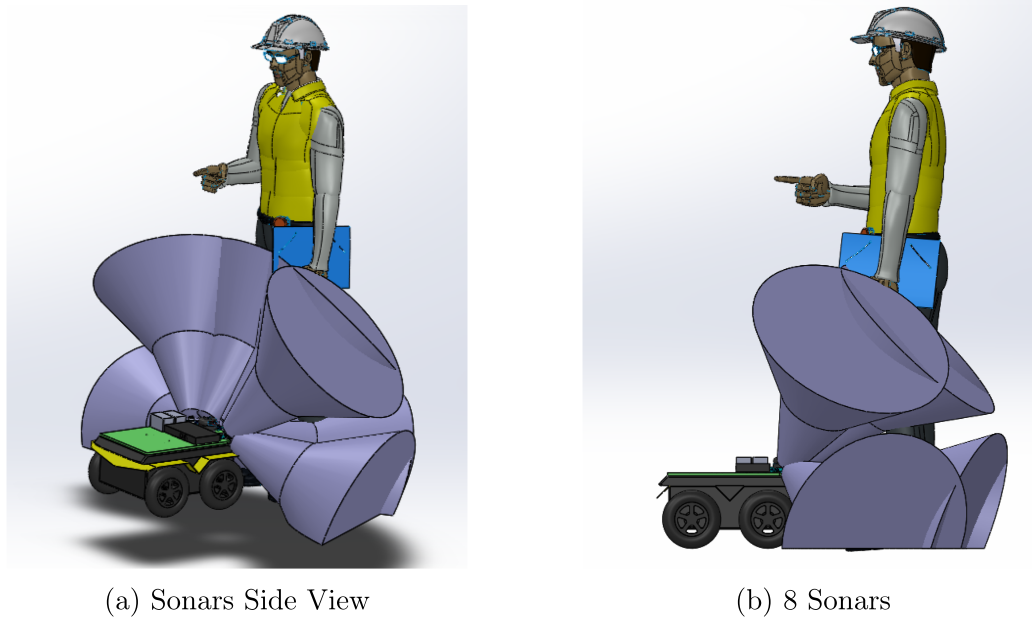
Figure 4-3: Realsense FOV

The low height of sensors in this design is the primary concern. Fig. 4-2 shows that the 2D Lidar's FOV (green) intersects with the engineer's shins, so it is still well within its field of view. If the Lidar is raised, it could intersect with the torso, but the configuration shown seems sufficient for pedestrian detection.

The Realsenses must be pitched upward (as described in Section 4.3). Their FOV (red) is sufficient to detect the engineer in Fig. 4-3, as his entire bottom half is within the middle sensor's perception region. The minimum sensing distance of 0.5m is clearly shown in Fig. 4-3a.

The light purple in Fig. 4-4 show the FOV of the 8 sonar sensors. The maximum range of these sensors is much lower, but so is the minimum range. When the engineer is standing right in front of the robot, his lower half is within the sonar FOV, so he will be seen by the robot.

The images in Fig. 4-5 show the FOV of the Realsense and sonars simultaneously, making clear why the two are needed in tandem. The sonars fill in the region in



(a) Sonars Side View

(b) 8 Sonars

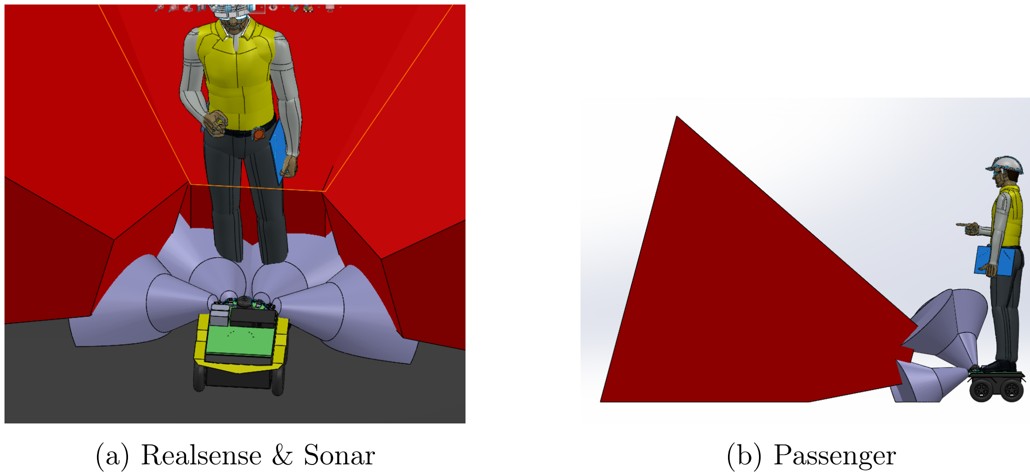
Figure 4-4: Sonar FOV

front of the robot where the Realsense cannot see. In Fig. 4-5b, the robot's passenger can ride with confidence that most obstacles in front of the vehicle will be within one of the sensors' FOV. While there is a blind spot right in front of the passenger's torso, only a strangely shaped obstacle would be able to enter that region without intersecting with any of the FOVs.

4.3 Pedestrian Detection with Cameras near Ground

The cameras in this design are low to the ground and tilted upward. The Very Fast pedestrian detector does not perform well when the image is skewed. It relies on the cameras being mounted horizontally.

A more recent algorithm will be used instead, called the Single-Shot Detector[35]. Its advertised frame rate on a Titan X GPU is 46fps. To compare with the Very Fast detector, the frame rate on the Gigabyte Brix is about 120fps (30fps for 4 cameras), whereas SSD runs at about 10fps. Therefore a computer with a better GPU than the Brix's NVIDIA 760 must be used. A modern laptop with a 10-series GPU can run SSD at 30fps which should be sufficiently fast.



(a) Realsense & Sonar (b) Passenger

Figure 4-5: Realsense & Sonar

4.4 Lidar Performance

This section investigates the performance impact of different Lidar mounting positions. Specifically, the sensor height and angular field of view are varied. The metrics used to evaluate the system's performance are the quality of the map produced by the SLAM algorithm, and the error in localization compared to the 1st generation robot's sensor setup. For the purpose of relative comparison, the 1st generation robot's sensor setup (Velodyne VLP-16 mounted atop the sensor hub) is considered "ground truth," since it is not practical to install a motion capture system in an entire building.

Two uses of the Lidar are Mapping and Localization, and the Lidar's performance at these tasks is described in the following sections.

4.4.1 Mapping

In the mapping problem, a robot is driven manually around a large building (the first floor of MIT's Building 32) with a Velodyne on top. Data is collected from the Lidar and the wheel encoders, and this data is processed offline on a different computer. The data is processed by the ROS package `SLAM_Karto`[1]. This package solves the SLAM problem to produce a 2D map of the environment as the robot drives around. It includes standard capabilities such as loop closure.

[1]`http://wiki.ros.org/slam_karto`

(a) 360°   (b) 180°

Figure 4-6: Maps Generated with Full and Limited Angular FOV (Velodyne)

Angular Field of View

Intuitively, a sensor that sees 360° should yield better performance than one that sees 180°, because it produces more information. To test this hypothesis, the same dataset is processed once as originally recorded, and then again after clipping the lidar data to only report measurements within the frontward-facing 180°.

As seen in Fig. 4-6, the two maps have the same general shape. However, the map in Fig. 4-6b indicates poor performance because the walls do not overlap in the red circled area. The map was generated by driving the robot from the top left corner, all the way to the upper right, and back to the top left. Therefore on the return trip, there was some drift in the measurement which led to the misaligned walls in the map. The map in Fig. 4-6a resembles the floorplan of the building it was driven in. This excellent performance justifies the use of the 360° Velodyne data as ground truth as proposed earlier.

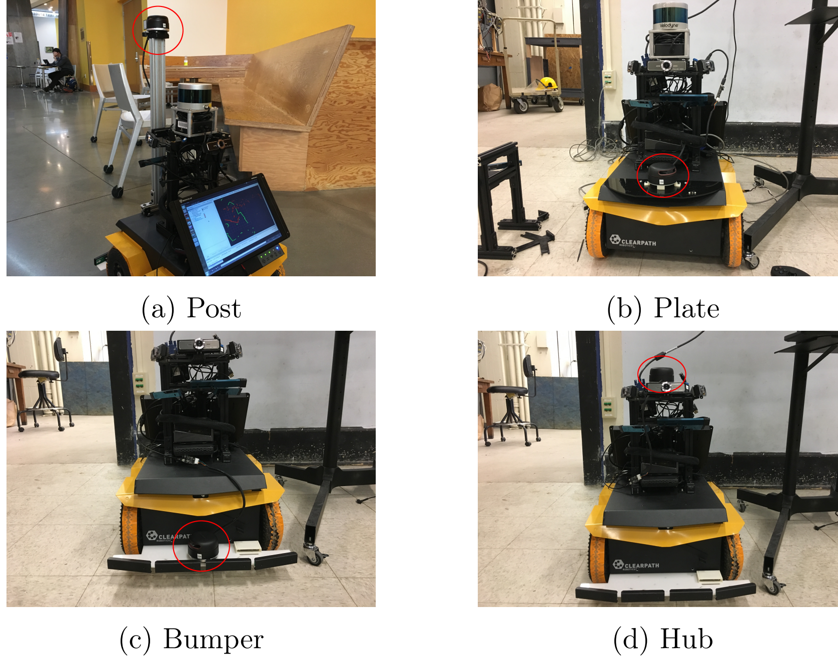
Sensor Type

In addition to varying the FOV and Height, the performance of the RPLidar is compared to the Velodyne in this section.

Sensor Height

The RPLidar is placed at 4 different heights:

- on the front bumper



(a) Post (b) Plate

(c) Bumper (d) Hub

Figure 4-7: 4 RPLidar Positions

- on the Jackal's top plate
- in the same position as the Velodyne,
- and on a stick extending above the Velodyne

These locations are 0.1, 0.3, 0.6, and 0.9m above the ground, respectively. Note that in the last two positions, it possible to use the full $360^{/circ}$ of the sensor, but in the first two the sensor is blocked from behind by the rest of the robot. See Fig. 4-7 for a diagram of the 4 positions on the robot.

The conjecture is that a sensor positioned higher off the ground will yield a better map, because it will see permanent features like walls, and mostly miss items that move, like tables, chairs, and people.

For each height, the map produced by the RPLidar is shown next to the map produced by the Velodyne during the same drive.

As seen in Fig. 4-8, the maps made by the two sensors are very similar. Both sensors fail on the loop closure in the bottom left, but the two maps have mainly the same features otherwise. There was likely an issue with the odometry during this particular run. At this height, the RPLidar seems to be a viable alternative to the

(a) RPLidar

(b) Velodyne

Figure 4-8: Map Generated with RPLidar at 0.9 m

Velodyne for mapping. Note that the entire 360° of the RPLidar was used.

In Fig. 4-9, the map is even further from the truth, with large drift in the bottom of the map.

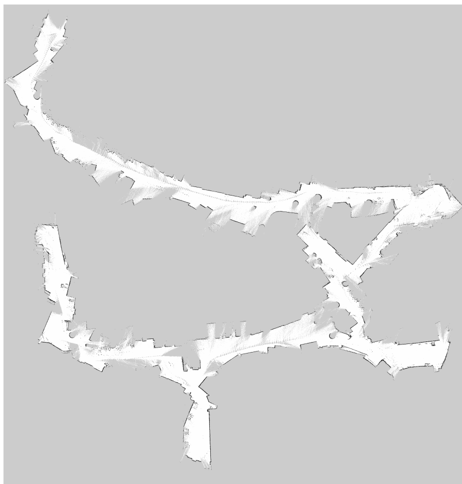
In Fig. 4-10, the map hardly resembles the ground truth. This was likely due to the sensor's vertical FOV intersecting with the ground, causing an artificially low maximum range. The vertical FOV limits how close to the ground the sensor can be positioned, although this could be augmented by pitching the sensor upward. However, that complicates the data transformation process, and does not mesh well with the construction of a 2D map.

In summary, for the mapping problem, these experiments indicate the RPLidar is an ok substitute for the Velodyne if it is positioned above 0.6 m, but not very useful at a lower height. The quality of the map (e.g., sharpness of walls) is slightly worse when using the RPLidar, but the main geometry of the building is successfully represented.

(a) RPLidar

(b) Velodyne

Figure 4-9: Map Generated with RPLidar at 0.3 m



(a) RPLidar

(b) Velodyne

Figure 4-10: Map Generated with RPLidar at 0.1 m

4.4.2 Localization

Although the RPLidar may not be able to make its own maps at the low height, it has been proposed that a map could be made once with the expensive Velodyne, and then during actual device execution, the RPLidar can be used to simply localize to the Velodyne's map. That problem is explored in this section.

There are a couple of localization evaluation methods to consider. If the ground truth map made by the Velodyne is used for all lidars, then it decouples the localization problem from mapping. Every sensor's performance is evaluated on an identical map. However, the Velodyne has an advantage because the map was recorded at its exact height, so features in the map should perfectly line up with features seen during localization. For the RPLidar, on the other hand, the features seen when it is mounted near the ground will not be the same as the features in the map. Looking at Fig. 4-8, there is not much difference between the maps made with different sensors at different heights, so the first comparison strategy will be used.

One map is used for all tests (the one from Fig. 4-9b) For each position of the RPLidar, the robot is driven in a loop around the building, starting in the bottom left corner, driving to the right and up, then turning around and driving back. For each sensor, the 360° and 180° are evaluated when applicable.

The estimated paths are shown in Fig. 4-11 for the RPLidar near the ground, at 0.1 m. There is only one area where the RPLidar's estimated position differs much, seen by the blue line near the (30, 15) point.

The error between the 360° Velodyne estimated position and the estimated position using other sensors is shown in Fig. 4-12. There is some initial transient error that dies off for all sensors. The 5m spike corresponds to the point mentioned in the previous paragraph. Otherwise, the error is never above 1m, and it is even better for the 180° Velodyne. This data indicates that localization on the scale required for this application is very good with any sensor.

In Fig. 4-13, clearly the RPLidar position diverges near the end when the RPLidar is mounted on the post. In Fig. 4-15 and Fig. 4-16, the RPLidar position diverges

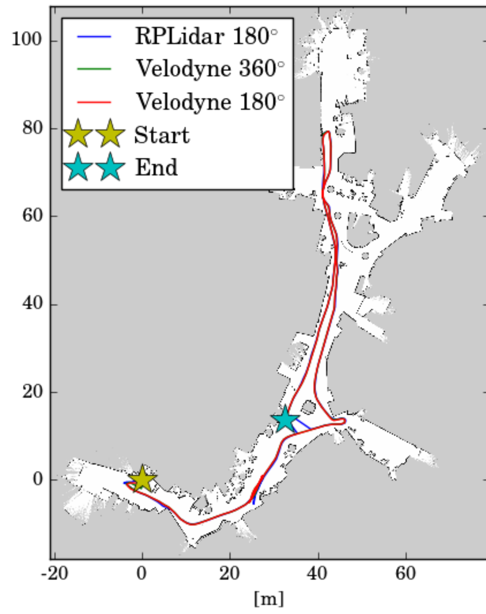


Figure 4-11: Estimated Path (RPLidar at 0.1 m)

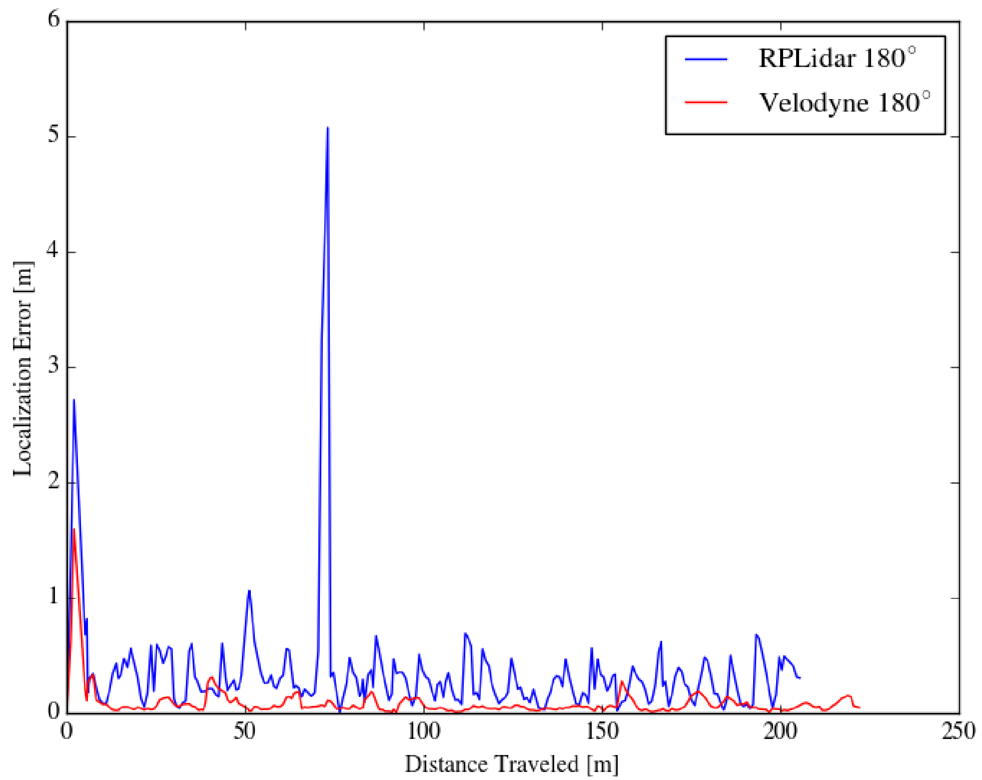


Figure 4-12: Localization Error (RPLidar at 0.1 m)

severely.

4.5 Summary

In summary, the robot is able to localize to the map very well with the Velodyne. The ability to localize with the RPLidar depends on the sensor placement. Interestingly, the ability to localize seems to be best with the RPLidar near the ground. However, this needs to be evaluated with more trials to say anything definitive about sensor height.

The common mode of failure is that none of the particles in the particle filter correspond to the true robot's position, so it attempts to re-distribute the particles. A better strategy would be to use known, unique landmarks in the map and only try to re-distribute particles if it sees a landmark and the estimated pose is not close to the landmark's known pose. With the current strategy, there is not much hope to re-localize globally if localization is ever lost.

The main takeaway is that the RPLidar can localize to a map with little error most of the time during drives of 100s of meters. However, the current localization strategy must be slightly adjusted to account for scenarios where the sensor does not see many walls due to its reduced range. In these scenarios, the robot's position jumps, which could likely be filtered out.

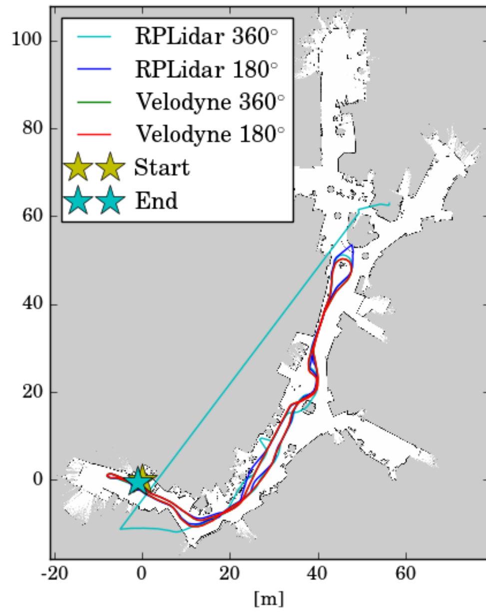


Figure 4-13: Estimated Path (RPLidar at 0.9 m)

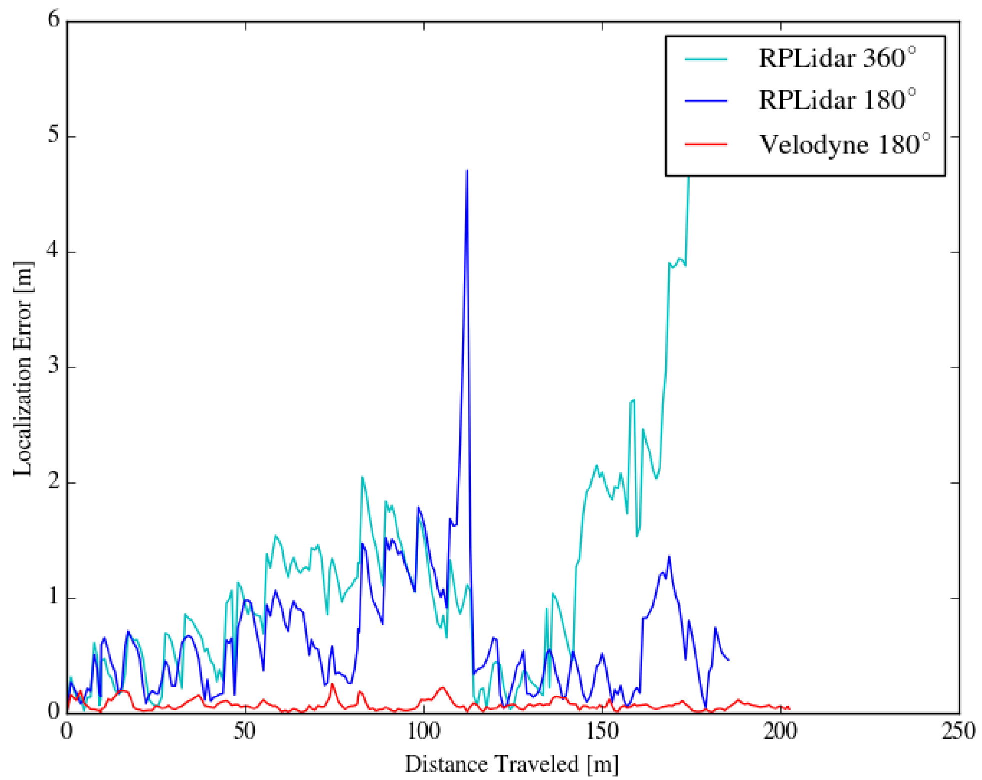


Figure 4-14: Localization Error (RPLidar at 0.9 m)

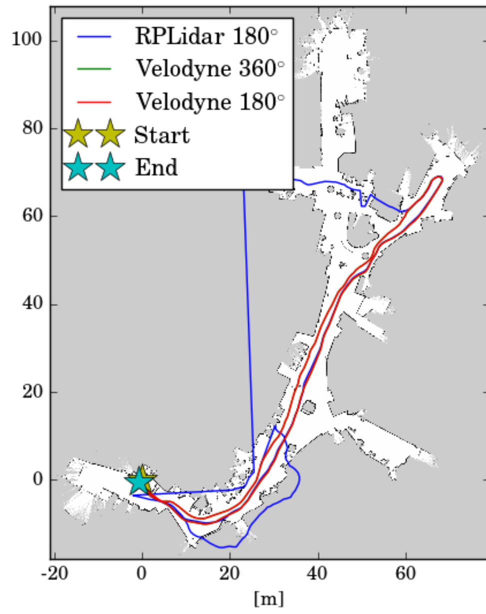


Figure 4-15: Estimated Path (RPLidar at 0.3 m)

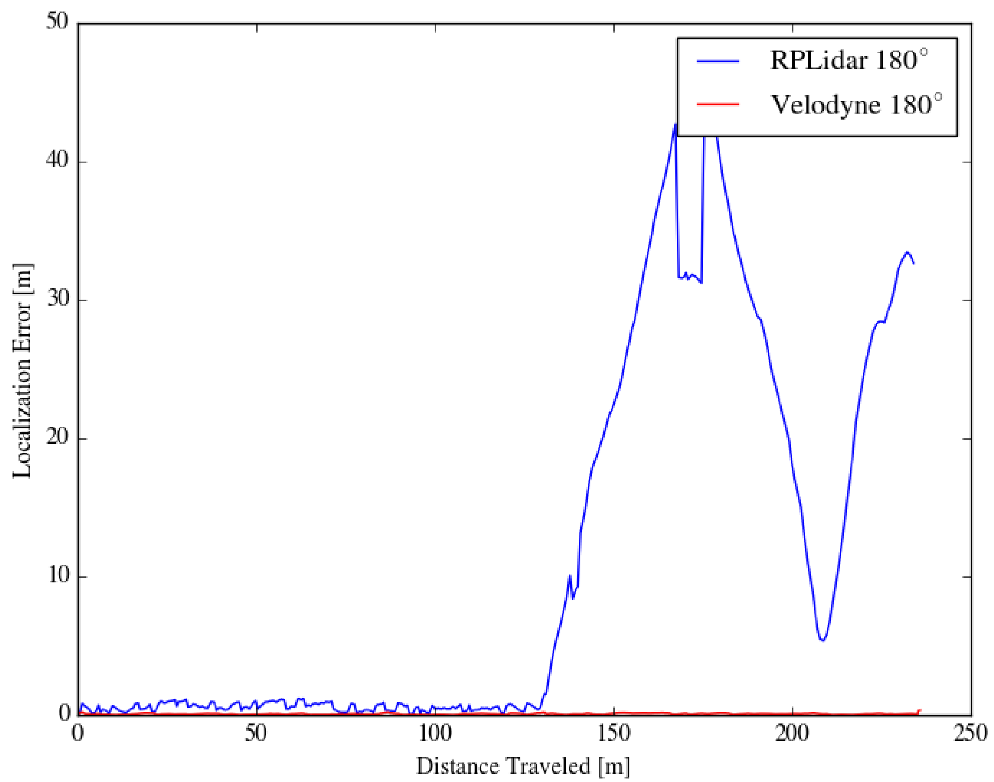


Figure 4-16: Localization Error (RPLidar at 0.3 m)

Chapter 5

Topology Aware Reciprocal Collision Avoidance (TARCA)

This chapter presents a novel motion planning strategy. It is an extension of Reciprocal Velocity Obstacles (RVO) and Optimal Reciprocal Collision Avoidance (ORCA), which are two common geometry-based collision avoidance algorithms. This algorithm aims to develop better motion plans using a prediction of the motion of nearby agents.

5.1 Background

An autonomous robot that drives in a world with pedestrians has two main objectives: it must make progress toward its goal and avoid collisions with other agents and obstacles.

The robotics community has developed many motion planning strategies, however existing methods do not fully address this scenario of a robot in a world with pedestrians.

One such class of motion planners assumes that the agents can communicate. For this application, unlike in human-human and robot-robot interactions where communication can occur in a common language, there is no natural way for the human and robot to communicate explicitly. Instead, the robot and human must infer what one

another is thinking based on implicit signals such as relative positions, velocities, and heading.

Other motion planning algorithms assume that a centralized entity with full knowledge of the world is able to send instructions to everyone. This is obviously unrealistic for this application because the pedestrians' paths are not controllable. Full knowledge of the world is also not obtainable, because the robot cannot know certain pedestrian traits such as goal locations.

Many motion planning algorithms assume that every agent follows the same policy. While not truly possible here, pedestrians can be modeled as agents using the same policy as the robot, but with different tuneable parameters that could be estimated online.

5.1.1 Geometric Approaches to Collision Avoidance

The definition of a collision is an intersection of two agents' reserved area. In the following methods, the agents are assumed to be round, however similar analysis can be done for other shapes. The size of the agents can be the robot's true size or include a buffer region if desired.

Collision Cone

Fig. 5-1[5] shows the simple case of 2 agents, A and B. The black dashed circle around Agent B shows the sum of the two agents' radii, at position $\boldsymbol{p}_B$. If A's center is anywhere within that circle, there is a collision between the two agents. Therefore the gray region represents the set of directions of travel that will cause collision. In the velocity space, if A chooses a velocity anywhere within the gray triangle, there will be a collision at some time in the future. This gray triangle is called a collision cone.

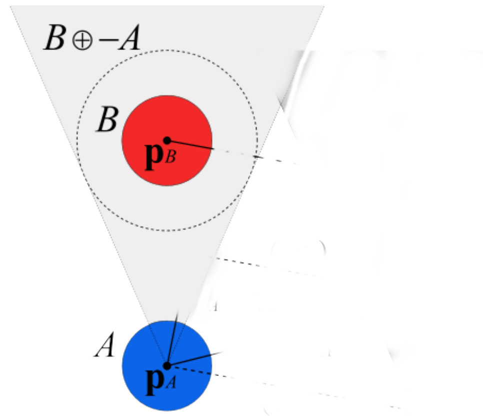


Figure 5-1: Collision Cone [5]: Gray area is set of blue agent velocities that lead to collision with stationary red agent

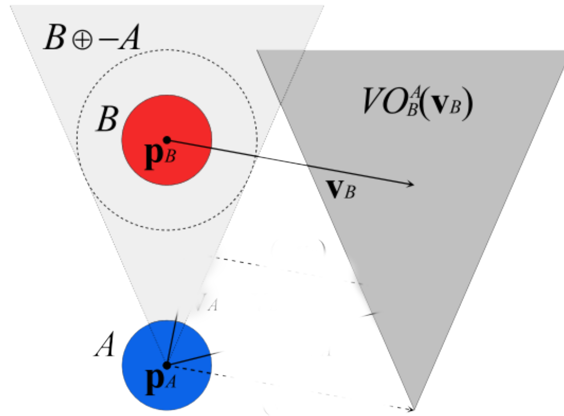


Figure 5-2: Velocity Obstacle [5]: Dark gray area is set of blue agent velocities that lead to collision with *moving* red agent

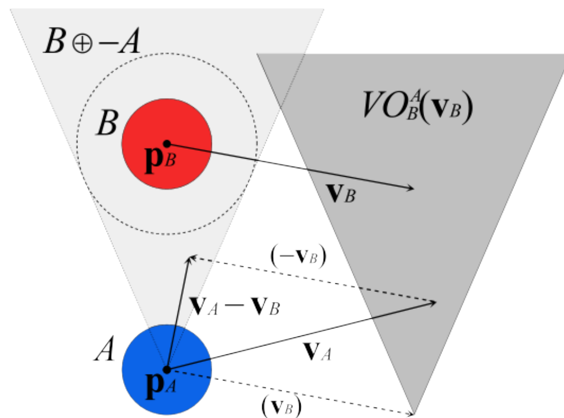


Figure 5-3: Velocity Obstacle [5]: Relative velocity $\boldsymbol{v}_A - \boldsymbol{v}_B$ within light gray region is equivalent to absolute velocity $\boldsymbol{v}_A$ within light gray region

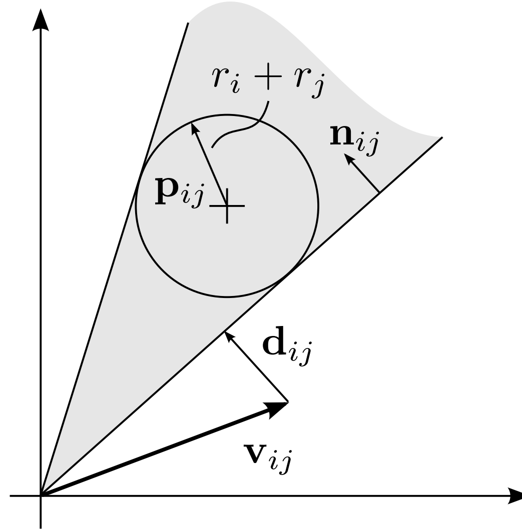


Figure 5-4: Velocity Obstacle (Simplified): Relative position $\boldsymbol{p}_{ij}$ and sum of radii $r_i + r_j$ define VO with inward normal vector $\boldsymbol{n}_{ij}$. $\boldsymbol{d}_{ij}$ is the shortest vector from the relative velocity $\boldsymbol{v}_{ij}$ to the VO boundary.

Velocity Obstacle

If B is moving with velocity $\boldsymbol{v}_B$, the collision cone is translated by the vector $\boldsymbol{v}_B$, as seen in Fig. 5-2. This shifted collision cone is known as the Velocity Obstacle (VO), because it is the set of velocities for agent A, $\boldsymbol{v}_A$ that will lead to collision with agent B.

Fig. 5-3 shows that the condition that A's absolute velocity $\boldsymbol{v}_A$ is within the VO is equivalent to the condition that their relative velocity, $\boldsymbol{v}_A - \boldsymbol{v}_B$ is inside the collision cone. It is often easier to visualize the VO than the CC because it involves absolute velocities instead of relative ones, but they have the same ramifications.

The diagram and notation is now simplified by assuming agent A is at the origin, so the relative position $\boldsymbol{p}_{ij}$ is what matters. The VO is the gray region in Fig. 5-4. $r_i + r_j$ is the sum of the two agents' radii. $\boldsymbol{n}_{ij}$ is the inward-facing vector normal to the VO boundary. For a particular relative velocity $\boldsymbol{v}_{ij}$, the vector d_{ij} is the shortest distance to the VO boundary, so it will face inward if the velocity is outside of the VO (safe), and outward if the velocity is inside the VO (unsafe).

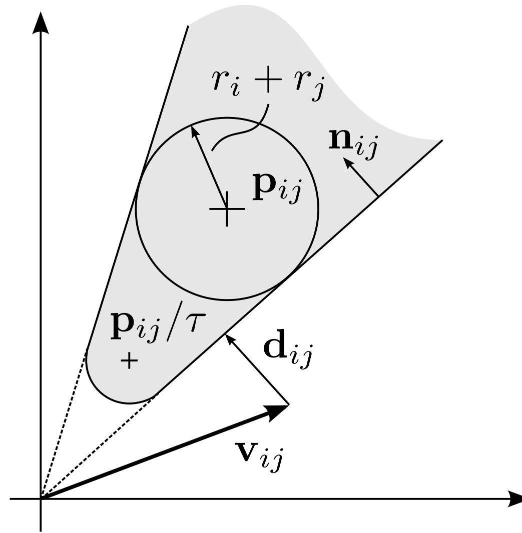


Figure 5-5: Velocity Obstacle (Time Limited): VO from Fig. 5-4 is relaxed if colllisions more than τ seconds in the future are ignored. The VO no longer includes the small region near the origin.

Limited Time Velocity Obstacle

All of the preceding definitions considered collisions over an infinite time scale. To be more realistic, often a guarantee of no collision within a certain time, τ, is sufficient. This relaxes the Velocity Obstacle, shown in Fig. 5-5, and a small set of velocities near the origin are now considered safe. This corresponds to a small relative velocity between the two agents, such that it will take a long time for the two to collide.

Reciprocal VO

The VO and Collision Cone are calculations of the set of safe velocities, but provide little guidance for how the agents should use the calculation. Consider two agents who are traveling toward each other, so their desired velocities are within the VOs. A simple idea is for each agent to add the minimum adjustment to its desired velocity such that the new adjusted velocity is outside of the VO. However if both agents apply this rule, they will both exert 100% of the collision avoidance effort, because they do not expect the other agent to change its velocity.

[5] presents a more efficient strategy, where each agent applies 50% of the minimum adjustment to its desired velocity such that the new adjusted velocity is outside of the VO. If both agents effort adds to 100%, the collision will be avoided. Thus, it is possible for one agent to do less of the avoidance, if the other agent does more. But in the Reciprocal VO algorithm (RVO), the two agents each do 50%. This weighting parameter is referred to as λ.

Optimal Reciprocal Collision Avoidance (ORCA)

An optimal formulation of the RVO method is presented in [36], known as ORCA.

In the ORCA method, the VOs are linearized. The "pizza slice" VO is simplified into a line that splits the plane into two halves. One half is safe and the other half is unsafe.

Assuming the vehicle can achieve any velocity within a certain magnitude, v_{max}, the safe velocity set begins as a circle with radius v_{max}. Each agent in the world's VO adds one linear constraint to this circle.

Multi-Agent Case

The preceding methods considered two agents. In general, a VO can be calculated for every pair of agents in the environment. The set of safe velocities is then the every velocity that is outside of every VO.

At the end, the set of safe velocities remains as a convex polygon remains, as seen by the slashed region in Fig. 5-6 [36].

5.1.2 Limitations of ORCA

ORCA is extremely fast to calculate. Some other classes of motion planning algorithms make predictions about future positions of other agents, which can be time-intensive. ORCA, on the other hand, has been shown to handle scenarios with 1000s of agents in real time. In theory, ORCA is the best decision rule if other agent's size, position, and velocity are known perfectly, and the other agent is also running

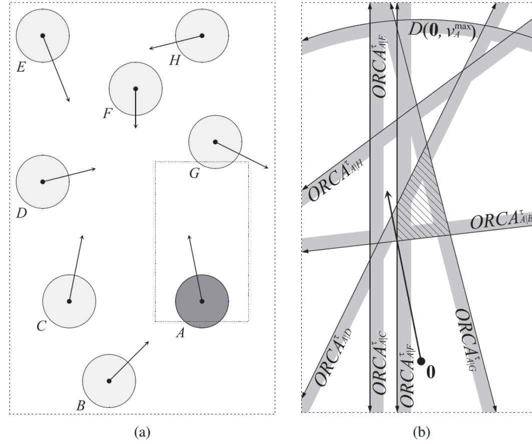


Figure 5-6: ORCA (Multi-Agent): On the left, ORCA can avoid multiple agents simultaneously. On the right, the original set of feasible velocites is a circle with radius v_{max}. A single linear constraint is applied for each agent's VO, resulting in the slashed, convex polygon of safe velocities.

ORCA.

A limitation of ORCA is the number of tunable parameters that affect its output. What started as a simple geometric argument has led to an algorithm where λ, τ, r_i, v_{max}, etc. must be selected. These parameters certainly vary from person to person, if the ORCA model is to be fit to pedestrians' behaviors. As mentioned earlier, pedestrian's parameter values could be estimated online, as shown in [37] but by the time enough data has been observed, the interaction with that agent may be over.

Another limitation is the assumption that all agents choose velocities simultaneously. With robots this may be feasible (albeit difficult to coordinate), but with humans it is impossible.

Yet another limitation is that the agents behave in a reactive manner – it is baked into the algorithm's definition. The desired velocity is of secondary importance to collision avoidance, so there is no guarantee that progress to the goal will be made. In experiments, agents are often "pushed around" by one another, rather than the desired cooperative behavior.

5.1.3 Extension: Receding Horizon Control

So far, this chapter has focused on the calculation of a set of unsafe velocities, given a particular relative position, velocity and size of pairs of agents. The underlying assumption was that the agent selects a safe velocity based on the current world configuration, with the intention of applying that velocity forever. Except for the τ parameter, there is no concept of execution time.

This is not a viable strategy for a real robot operating in a world with pedestrians for many reasons. For one, the pedestrians are not following ORCA (or any deterministic planner), so their behavior can not be modeled indefinitely. More fundamentally, they will certainly not maintain a constant velocity, and the VO is only valid for one particular relative velocity. Another reason is the robot's inability to perfectly track the chosen velocity. Disturbances such as wheel slip and imperfect control algorithms are not accounted for in the ORCA formulation. Yet another reason is field of view limitations. The robot's VO can be calculated using the pedestrians within the perception radius, but if the robot drives and new pedestrians appear, they must be accounted for.

For these reasons, the ORCA algorithm is often applied in a Receding Horizon Control framework [38]. This means that at time t_0, the safe velocity is calculated using the world state as estimated by the sensors, and the velocity is executed for $t_{execute}$ seconds. Then, at time $t_0 + t_{execute}$, the new world state is estimated, a new safe velocity is calculated, and that velocity is executed for another $t_{execute}$ seconds. This strategy is more robust to un-modeled changes in the environment.

In theory, the τ parameter could be reduced down to $t_{execute}$, because the chosen velocity only needs to be safe until a new velocity is chosen. To account for a safety margin, however, this is usually not done in practice.

5.1.4 Extension: Forward Simulation

To improve the reactive behavior of ORCA, a method was proposed by [39] known as Progressive Hindsight Optimization (PHOP). In the PHOP algorithm, each agent

discretizes the feasible velocity space into n_{vel} velocities at fixed angles. It then simulates what would happen if the agent were to execute ORCA for a certain amount of time, using one of the n_{vel} velocities as its preferred velocity. This forward simulation also includes propagation of the other agents in the world, assuming that they are running ORCA. Since ORCA is so fast to compute relative to the planning rate required for real-time robotics, there is plenty of time to propagate many different velocities for many time steps into the future.

After simulating the world for each of the n_{vel} velocities, PHOP selects the best action using the cost function presented below in Eq. (5.1).

5.2 Avoidance Topologies

The strategy proposed in this thesis makes use of the forward simulation concept, but differs in the selection of actions to propagate forward, and the evaluation metric to determine the best action.

One characteristic of an interaction between two agents is the side on which they pass one another. That is, an agent can pass the other on the right or left, or the agent can choose to approach the other agent head-on, with the hope that the other agent will move out of the way.

This notion of avoidance topologies can be naturally applied to the VO framework. Recall that the time-limited VO looks like a truncated "pizza slice."

The two linear and one round boundaries can be relaxed into three linear boundaries. The two linear boundaries remain as before, and the round boundary is replaced by a tangent line. This is visualized in Fig. 5-7, by the lines with normal vectors $\mathbf{n}_{ij}^h$ for Head-on, $\mathbf{n}_{ij}^l$ for Left side, and $\mathbf{n}_{ij}^r$ for Right side.

Therefore, each side of the pizza can be used to constrain the passing topology.

This avoidance topology strategy can be applied

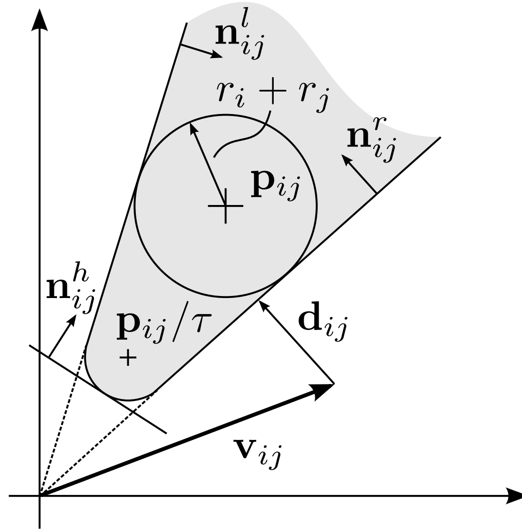


Figure 5-7: Velocity Obstacle (with Topologies): The boundary of the VO now consists

5.3 Forward Simulation Strategy

In this section, a novel strategy to simulate the world forward is presented. The differences between the strategy and an existing approach are then compared.

5.3.1 TARCA

In Topology Aware Reciprocal Collision Avoidance (TARCA), the world is forward simulated with each of the 3 topologies per nearby agent.

Algorithm 1 describes the strategy. If there is time left after the initial simulation for every pair of neighbors and topologies, the loop is repeated with a finer-grain simulation, where each time step is half as long. This strategy is designed so that even if there are many agents, the algorithm will at least try each topology with a coarse simulation once. Each time a plan is simulated forward, a cost function is applied to the predicted world state

```
Data: World state, planning_horizon
Result: Best action executed
preferred_velocity = max speed in direction of goal;
agent_ids = list of neighbors sorted by proximity;
simulation_step_size = planning_horizon / 10;
while time_used < time_limit do
    for topo in topologies do
        world = forward_simulate(action,simulation_step_size,
          planning_horizon);
        cost = calculate_cost(world);
        costs[agent_id, topo] = cost;
    end
    agent_id = next item in agent_ids;
    if done with all agents then
        simulation_step_size *= 2;
        reset to closest agent_id;
    end
end
lowest_cost = min(costs);
best_action = [agent_id, topo] for lowest_cost;
safe_vel = rvo_with_topologies(preferred_velocity, agent_id, topo);
execute(safe_vel);
```

Algorithm 1: How to choose best action

5.3.2 Comparison to PHOP

Instead of forward propagation of a set of fixed angles, TARCA considers a set of actions that changes based on the environment. For an extreme example, if there are no other agents in the world, every velocity is safe, so the agent can safely execute its preferred velocity. It would be wasteful for the agent to forward simulate any velocities except the one that gives the optimal path to the goal (with respect to time, distance, fuel, etc.), because the preferred velocity will be chosen with certainty.

For a less extreme example, consider the case of two agents moving toward each other, so that their preferred velocities point directly at one another. The only safe interactions are to pass on the left, pass on the right, or go head-on and slow down before colliding. There is no reason to consider traveling in reverse in this example. Also, the amount of adjustment to the preferred velocity, the d_{ij} vector, depends on the distance to the other agent and each agent's size.

PHOP considers a fixed set of preferred velocities to simulate forward. It does not change this set of velocities based on the environment. Therefore it does not naturally capture the notion of passing on the left and right.

5.4 Evaluation Metric

Both TARCA and PHOP simulate the world forward to determine the future world state given a chosen action at the current time step. To compare world states, an evaluation metric is developed. The initial action that leads to the world state with lowest cost is executed.

5.4.1 TARCA

In TARCA, the cost of a world state is based on the difference between each agent's end position, and the position they would have achieved had there been no other agents around, based on a simulation for t_{sim} seconds.

For the user agent, the preferred velocity is known, so it can be used for the calculation of user agent cost, J_0. For other agents, the cost J_i involves their current velocity since the preferred velocity is unknown. Certain agents' costs are weighted more heavily than others, where agent i has cost weight w_i. The agent costs are

$$J_0 = (\mathbf{p}_0(t_0 + t_{sim}) - (\mathbf{p}_0(t_0) + \mathbf{v}_{0,pref}(t_0) \cdot t_{sim})) \tag{5.1}$$

$$J_i = (\mathbf{p}_i(t_0 + t_{sim}) - (\mathbf{p}_i(t_0) + \mathbf{v}_i(t_0) \cdot t_{sim}) \tag{5.2}$$

The total cost of the world state, J is the weighted sum of all n_{agents} agent costs,

$$J = \sum_{i=0}^{n_{agents}} w_i \cdot J_i \tag{5.3}$$

5.4.2 Comparison to PHOP

The PHOP reward function only considers the state of the host agent. A metric of progress toward the goal Section 5.4.2 is divided by an energy metric Section 5.4.2 to generate a total reward Section 5.4.2. The action with the biggest reward is executed.

$$\begin{aligned} Progress &= \mathbf{v} \cdot \frac{\mathbf{g} - \mathbf{p}}{\|\mathbf{g} - \mathbf{p}\|} && (5.4) \\ Energy &= 1 + \|\mathbf{v}_{pref}\| && (5.5) \\ Reward &= \frac{Progress}{Energy} && (5.6) \end{aligned}$$

The first limitation of this approach is that it only considers the host agent. The agent will have no concept of its impact on other agents, which is critical in a robot for socially acceptable navigation. It also does not allow for the agent to dynamically adjust its agressiveness.

A second limitation is that over the duration of one simulation step, the average reward for all actions converges to the same value. This is a result of PHOP's average reward strategy.

The agent using PHOP will simulate forward several different actions for a certain amount of time, t_{sim}. At the start of each planning step, the agent calculates the reward if each action is executed for the full simulation time. If there is still computation time remaining, the agent does a second iteration of forward simulation. This iteration, the simulation is partitioned into 2 halves, and a plan is created for every pair of actions (where action A is executed for $t_{sim}/2$, and then action B is executed for the remaining $t_{sim}/2$). The reward associated with the world state at the end of each plan is attributed to the first action in the plan. The reward is averaged with the other plans that have the same first action. If there is still computation time remaining, a 3$^{\text{rd}}$ iteration of simulation occurs, and so on.

Consider an example where there are only 3 actions (A,B,C). After several iterations, the plans associated with action A include:

$\{A\}$

$\{A, B\}$,$\{A, C\}$

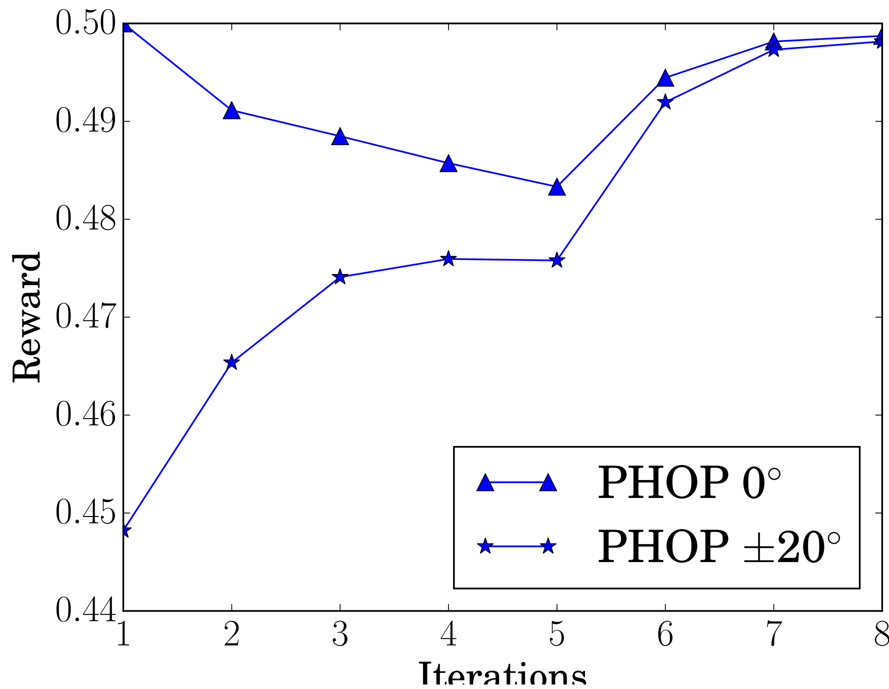


Figure 5-8: PHOP Average Reward for 3 Actions (0°,+20°,-20°) converges over time

$\{A, B, A\}$,$\{A, B, B\}$,$\{A, B, C\}$,$\{A, C, A\}$,$\{A, C, B\}$,$\{A, C, C\}$,...
Clearly, as the number of iterations increases, the plan with action A for the full simulation is only one of very many plans. And most of the plans are not really that related to the action A, because A may only be executed briefly at the beginning of the simulation.

This concept is plotted in Fig. 5-8, where an agent has 3 possible actions, and an open path to the goal. The 0° action is clearly the best, because there is no reason for the agent to go in a direction except straight to the goal. In the first iteration, the average reward for 0° is much greater than the reward for ±20°. But, after several iterations, the 3 actions converge to a similar reward because they are diluted.

In conclusion, the partitioning strategy allows the agent to explore a diverse set of paths, but the averaging strategy leads to an ambiguous choice for the best action.

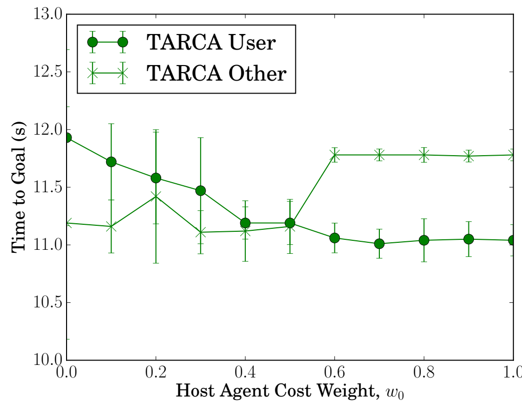


Figure 5-9: Varying Host Agent Weight

5.5 Results

5.5.1 Cooperation

Variation of Host Agent's Cost Weight, w_0

Consider the case of a pair of agents driving toward each other. The weight term w_i for each agent's cost can be varied to penalize the impact of a plan on an agent's preferred velocity. The two agent's weights are set such that $w_0 + w_1 = 1$, and w_0 is varied from 0 to 1.

Fig. 5-9 demonstrates that the weight term has an effect on the two agents' paths. For low w_0, the agent does not weigh its own weight much, so it chooses a plan to minimize the disturbance to the other agent. It allows the the other agent to go straight to its goal. Conversely, for high w_0, the agent mostly ignores the impact its plan has on the other agent, so the agent drives straight to the goal and indirectly forces the other agent to do all of the collision avoidance.

When w_0 is near 0.5, the two agents have almost the same time to goal, meaning they distribute the collision avoidance task equally. Also note that the time to goal for the slowest agent is minimized near 0.5 because neither agent has to do a large collision avoidance maneuver.

This result implies that for standard operation where load should be shared equally, w_0 and w_1 should be the same. In certain scenarios where the agent should

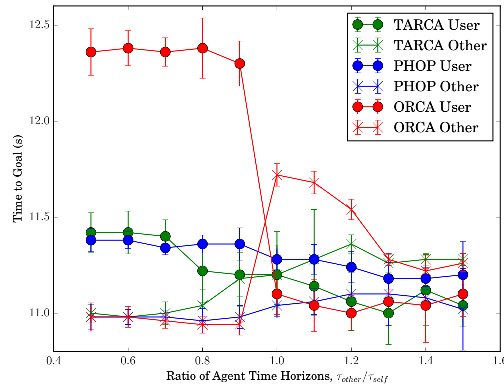


Figure 5-10: Varying Time Horizon: 3 Algorithms Compared

not care about its impact on the environment and simply wants to achieve a minimum time to goal, w_0 should be very high. In other scenarios where the agent is not in a hurry and does not want to affect other agents, w_0 should be small.

Variation of Other Agent's Time Horizon, τ

Next, instead of w_i, the time horizon of the other agent is varied, and both agents are given equal weight, $w_0 = w_1 = 1$. The time horizon of the other agent is varied from slightly below the host agent's time horizon, to slightly above. Recall that the time horizon, τ is the amount of time an agent guarantees collision-free travel. If τ is small, the agent does not consider collisions until they are very close to occuring, so it can be thought of as a highly reactive agent. Conversely, an agent with large τ is conservative and even agents very far away have an effect on the VO.

The time to goal for each agent is shown in Fig. 5-10 for each of the 3 algorithms (ORCA, PHOP, TARCA).

Variation of Other Agent's Radius, r_{other}

Next, instead of τ, the radius of the other agent, r_{other} is varied. The time to goal for each agent is shown in Fig. 5-11 for each of the 3 algorithms (ORCA, PHOP, TARCA).

In ORCA, as the other agent's radius grows, its time to goal also increases. For

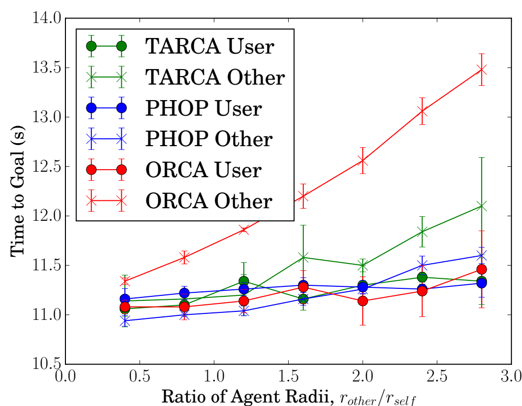


Figure 5-11: Varying Radius: 3 Algorithms Compared

TARCA and PHOP, this growth still exists but is less significant.

5.5.2 Planning Horizon

An important parameter in RHC is the planning horizon. A short planning horizon only considers the impact of an action for a short duration, so locally optimal plans will be selected. A long planning horizon allows some local suboptimality, if the long-term effect of a plan is beneficial. For example, when crossing a busy intersection, it may be worthwhile to stop and wait for a gap to open up, and then proceed at full speed. But a planner with a short planning horizon would consider the stop and wait action to be too costly, without understanding that it is useful in the long-term. Of course, the tradeoff with a large planning horizon is the extra computation time it requires.

TARCA includes a planning horizon parameter that can be adjusted. Future work will show the impact this parameter has on performance.

5.6 Conclusions

The new method proposed in this chapter, TARCA, extends the VO framework to select an action based on the predicted outcome of many actions. The original ORCA work selects an action with the intention of executing it forever. Another existing

approach, PHOP, was the first to extend VOs with predictions, but has several limitations for the application of a robot that navigates autonomously in pedestrian-rich environments.

TARCA improves on ORCA's sensitivity to model parameters, as shown earlier in Fig. 5-10 and Fig. 5-11. Its robustness to model parameters is comparable to PHOP.

TARCA improves on PHOP by adding an ability to weigh the agent's impact on others. Results in Fig. 5-9 demonstrate that the agent can dynamically adjust how much it cares about its impact on others. This is essential in environments with both robots and humans. A robot can impede another robot's progress without much concern, but if a robot gets in a human's path, the human will be annoyed and inconvenienced.

5.7 Future Work

There are a few areas of future work for TARCA. First, the algorithm should be implemented on a real robot to test its performance with humans who are not exactly ORCA agents. The estimation of the human agents' ORCA parameters could possibly be done online. Finally, it would be worthwhile to add non-holomic constraints to the codebase, using [40]. This work reasons about the dynamic constraints of a differential-drive robot, but there is no open-source implementation of the algorithm. This type of robot cannot track every velocity perfectly, and adding these constraints to the VOs would make the collision avoidance guarantees theoretically sound.

5.8 Summary

The new method proposed in this chapter, TARCA, extends the VO framework to select an action based on the predicted outcome of many actions. The actions include passing each of the nearby agents on the left, right, or head-on.

Chapter 6

Conclusion & Future Work

This chapter summarizes the material presented in the thesis. It also proposes future areas of research and development.

6.1 Summary

Chapter 1 introduced the need for a robot that can drive in pedestrian-rich environments in a socially acceptable manner. This technology can be applied to a wide range of vehicles, including personal transportation devices and cargo delivery robots. There are several academic and commercial groups working on a similar type of robot, but none have publicly demonstrated an ability to drive a robot in a realistic, pedestrian-rich environment, at human walking speed, for an extended period of time without human intervention or teleoperation.

Chapter 2 described the hardware of the 1st generation robot designed for this application. The choice of chassis, sensors, and computers is described.

Chapter 3 explains the software of the robot. A high-level block diagram is presented, as well as explanation of the Localization, Perception, Path Planning, Control, and Goal Selection systems. While the Localization, Perception, and Control systems use standard libraries, the Path Planning strategy is novel and allows the robot to drive safely at such high speed.

Chapter 4 explains the design and budget constraints of the 2nd generation robot.

It then presents a new robot sensing design that is sleeker and less expensive than the 1st generation design. The performance tradeoff is briefly evaluated for the Localization system.

Chapter 5 proposed a new motion planning algorithm, TARCA, which extends the VO framework to select an action based on the predicted outcome of many actions. The actions include passing each of the nearby agents on the left, right, or head-on. Comparisons to existing algorithms, ORCA and PHOP, are presented. This algorithm allows the agent to dynamically adjust the extent which its planned paths impact other agents. This is critical for a robot operating in pedestrian-rich environments, where some pedestrians are paying attention, and others are distracted.

Appendix A contains detailed documentation for the 1st generation robot that will be useful to anyone who uses the platform in the future. There are labeled pictures of each region on the robot, information on power requirements, and other useful notes. There is also a section on the software which includes instructions to put the robot in a demo mode. There are block diagrams with input/outputs for each subsystem in the software architecture. This is especially useful to someone who is interested in writing a new version of a subsystem, while maintaining compatibility with the rest of the codebase.

6.2 Future Work

There are many steps remaining to develop an even more robust, and commercially-viable robot. Some of the next steps involve implementation of existing ideas, and others involve creating new algorithms.

The next step in hardware development is to fully build the 2nd generation design, and verify the sensor FOVs are as predicted in Section 4.2.1. Another future project could involve mounting this system on a different robot chassis that is capable of carrying a human passenger.

One of the next software challenges is to include vehicle dynamics in the path planning strategy. Currently, the planner does not consider physical constraints such

as stopping distance or maximum acceleration.

Another software challenge is to model the perception delay in the path planner. There is up to 1 second delay in the system currently, and this negatively affects performance because plans are generated on old information.

Another project is to add a multi-class object classifier into the perception system, and to use that information in the planner. For example, a dog should be avoided differently than a chair, but currently anything that is not detected as a pedestrian is treated as a static obstacle.

From a system performance perspective, it would be beneficial to develop a metric that captures how socially acceptable a robot's interactions with pedestrians are. Because there is not really a unit to describe this quantity, this will likely be a statistical argument, such as Robot A had unacceptable interactions 10However the definition of socially acceptable interaction must be fully described. One proposed method is to use the same rules that the path planner's neural network is trained on. Data from the real robot can be passed through this same cost function to provide a system level performance metric.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix A

Notes for Future Users of 1st Generation Platform

A.1 Hardware

This section contains a series of notes that will be useful to anyone using the 1st generation platform in the future.

There is a 14" monitor onboard for debugging and display. This monitor is optional and can be removed. The monitor cuts off the edges of the screen when using HDMI, so VGA is recommended.

The two computers are connected to a KVM switch, so that the user can have one keyboard/mouse/monitor that switches between computers with the click of a button.

The internal computer bay contains many components.

- several powered USB hubs that connect to the Brix
- 2x 19V, 10A regulators for the Nuc and Brix
- 2 screw terminal blocks for 5 and 12V
- the robot's original user power board from that provides 5V @ 5A, 12V @ 10A, and 24V @ 20A

Each device's power consumption is detailed in Table A.1. The battery is 29V fully charged, and 24V empty. The battery voltage can be checked with the command:

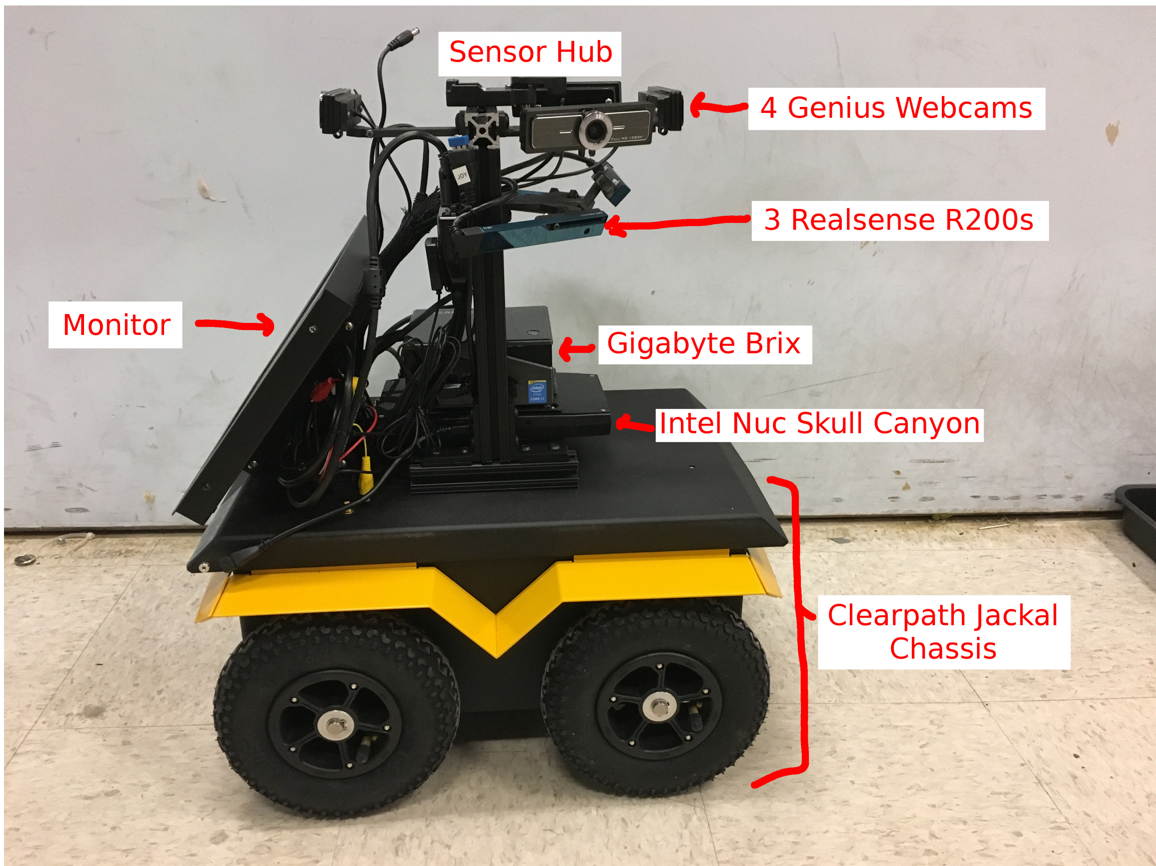


Figure A-1: 1st Generation Jackal (Side View)

Table A.1: Power Consumption

	Voltage [V]	Current [A]	Power [W]
Motors	24	2-5	50-100
Brix	19	10	190
Nuc	19	-	-
Velodyne	12	0.6	8
Webcam	5 (USB)	low	low
Realsense	5 (USB)	0.3/camera	1.6/camera
Bumper	5 (USB)	low	low
Monitor	12	0.8 nom, 2 startup	10

```
$ rostopic echo /status
```

while the ROS service is running.

The robot has lights that indicate it is on and connected. A solid blue light means the robot is on. A blinking blue light means it is turning off within a few seconds. The robot turns off if the computer is disconnected, or if the power button is pressed. The button on the far left is the motor switch. If solid green, the motors are on. If solid off, the motors are off, which can be helpful to guarantee the robot won't move. There is a light to show that the ROS node is running on the computer and connected to the robot. This is the light next to the two arrows. There is also a light to show that the computer has internet connection.

Occasionally if the computer does not boot fast enough, the robot will turn itself off because it thinks the computer was unplugged. This is freakishly annoying but is usually fixed by trying once or twice again.

To switch which of the 2 computers is connected to the monitor and keyboard, a KVM switch is installed on the sensor hub. The KVM switch connects to both computers' VGA outputs and USB ports, and the switch selects which signals are connected from input to output. Fig. A-2 is labeled to show the important ports of the KVM switch. There is a second USB port on the KVM switch input, but it is not connected to either computer so its use is not recommended and it is taped over.

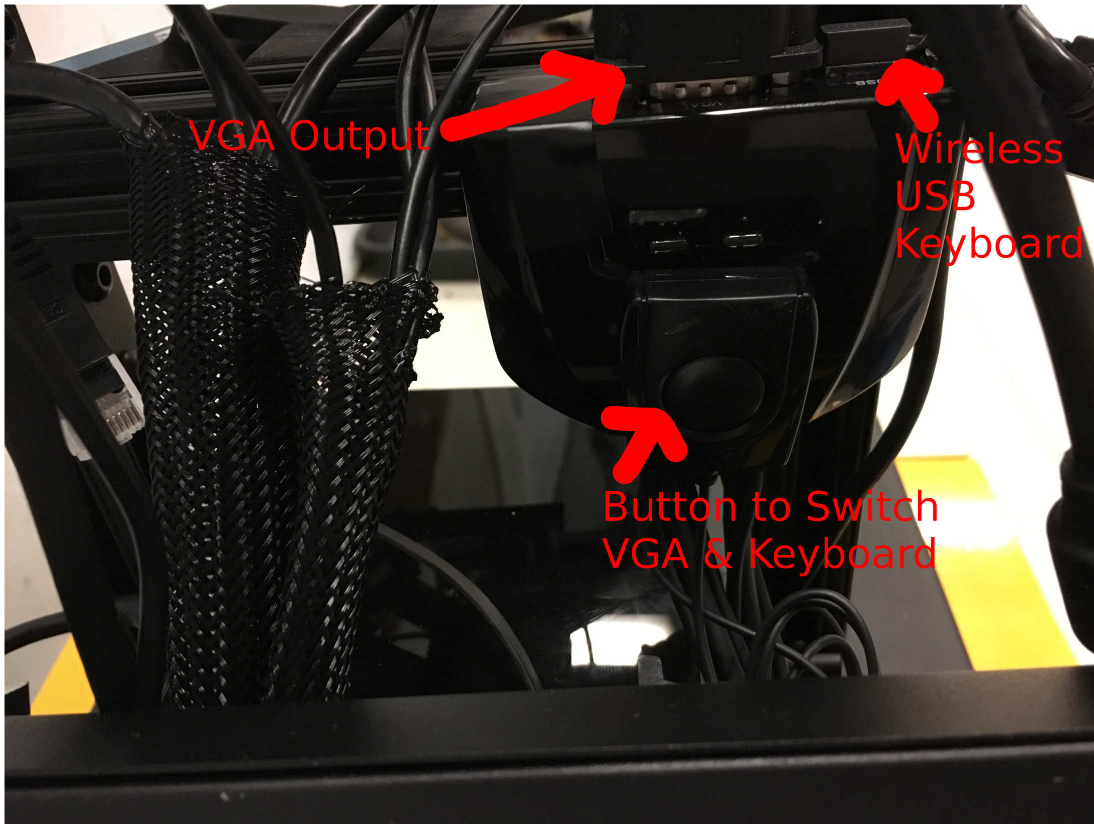


Figure A-2: KVM Switch

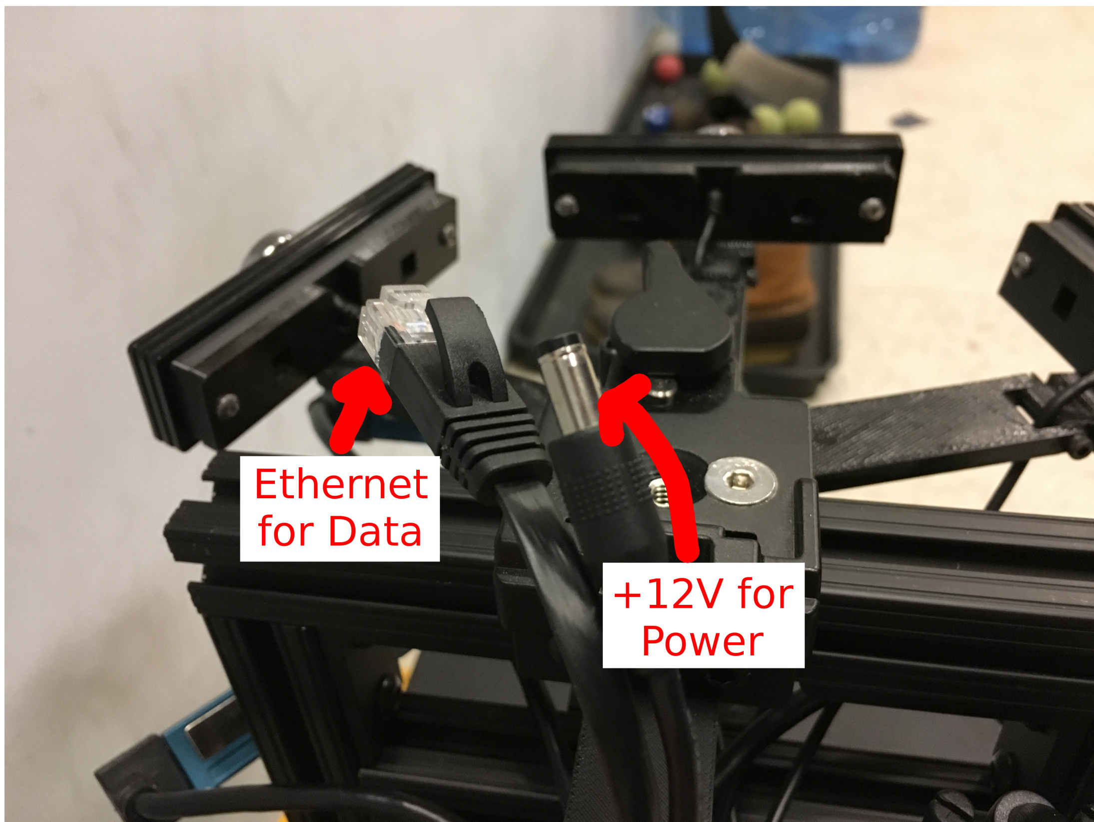
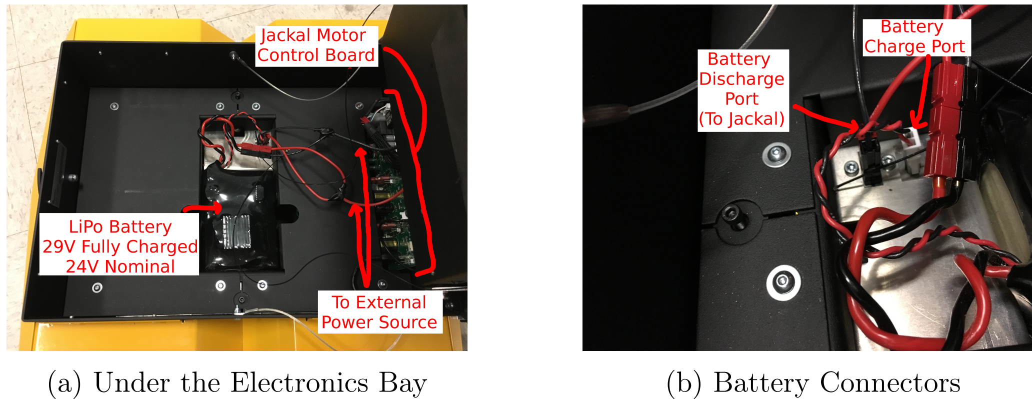


Figure A-3: Velodyne Connectors



(a) Under the Electronics Bay

(b) Battery Connectors

Figure A-4: Jackal Sides Showing Connectors

The Velodyne is connected with 2 cables as shown in Fig. A-3. The tripod mount allows the Velodyne to be stored in a safe place separate from the robot, and securely added during robot execution.

A.1.1 Battery

The battery used is a 24V LiPo battery. Its nominal voltage is 24V, but it gets up to 29V fully charged. When it is below approximately 24V, the Jackal powers down automatically. It can be charged with a 29V power supply, but the recommended charge rate is limited to 3A.

A.1.2 Internal Electronics Bay

USB

There are 3 USB 3.0 hubs in one corner of the electronics bay. The top two plug into the Brix, and the bottom one plugs into the Nuc. The top one is labeled Brix Hub A, the middle one is Brix Hub B, and the bottom is labeled Nuc Hub. Each hub receives 5V from the terminal block so that it can power devices beyond the typical USB spec.

The exact ports that the USB devices are plugged into is important. It is possible to change the order of these, but not recommended.

Table A.2 describe the configuration of devices in ports, with the ports numbered

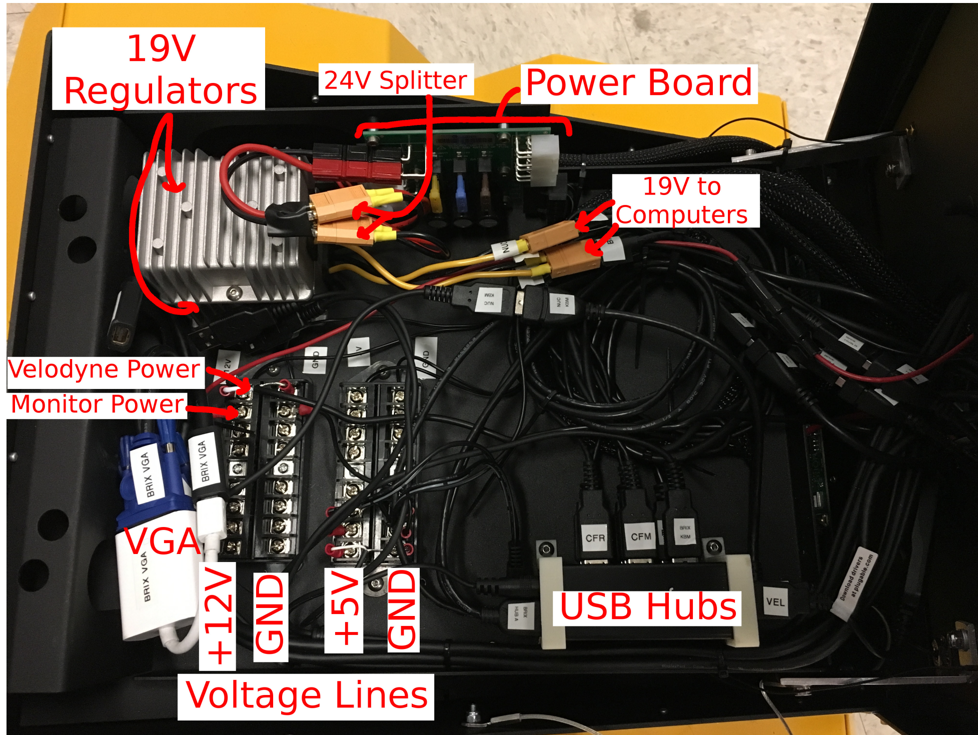


Figure A-5: Labeled Electronics Bay

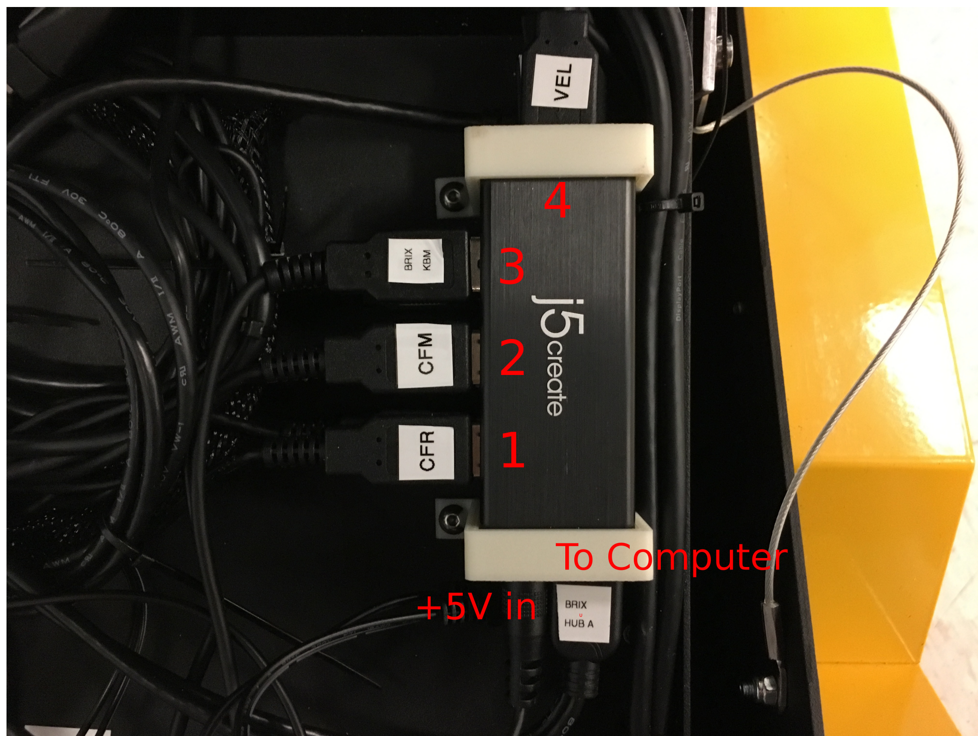
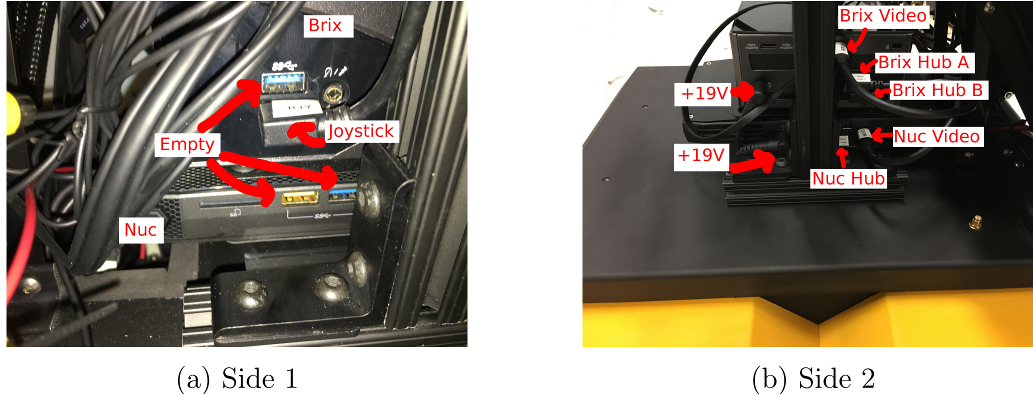


Figure A-6: USB Hubs

Table A.2: USB Hubs

	Port 1	Port 2	Port 3	Port 4
Top	CFR	CFM	Brix KBM	VEL
Middle	CFL	CBM	RPLidar	JACKAL
Bottom	RFL	RFM	RFL	Nuc KBM



(a) Side 1 (b) Side 2

Figure A-7: Jackal Sides Showing Connectors

as shown in Fig. A-6. The labels on the devices are shorthand for

- CFM = Camera Front/Back Middle/Left/Right
- RFM = Realsense Front Middle/Left/Right
- VEL = Velodyne
- JACKAL = Jackal's motor control board
- x KBM = Keyboard+Mouse for computer x, connected to KVM switch

On the computer level, the USB devices and hubs should be plugged into the exact ports as shown in Fig. A-7. There are a few unused ports as seen in Fig. A-7a, which leaves room for future expansion such as new sensors.

If the ports are changed, a file should be updated, located at `~/ford_ws/src/ford/jackal/usbCustomRulesJackalFord.rules`. That file is linked to the `/etc/udev` directory. Udev rules are a Linux concept that permanently set certain attributes to USB devices. Normally when one plugs a device into USB, it is assigned a device name, such as `/dev/video0`. A udev rule could name it `/dev/usb_cam_front_left` instead. This rule could be assigned based on the exact port of the USB hub that

the device is plugged into (using the KERNEL attribute), or many other attributes. But in the case where there are many identical devices (like 4 identical webcams), the only unique attribute is the USB port.

The other limitation for changing the USB ports is the available bandwidth. The webcams are USB2.0, and only 2 can be plugged into any one USB hub. The uvcvideo drivers for these inexpensive cameras are not great, and the cameras reserve excessive bandwidth by default. The following command can be executed to allow multiple webcams per USB hub, however 2 is still the upper limit:

```
$ sudo rmmod uvcvideo
$ sudo modprobe uvcvideo quirks=128
```

. This instruction is added to uvcvideo module's options file so that it is executed once every time the computer boots up.

There are many issues with the Realsense R200 cameras and Ubuntu 14.04. Therefore they are plugged into the Nuc which is runnning Ubuntu 16.04. It is not recommended to try multiple Realsense cameras with 14.04. It involves updating the kernel and other steps, and was not demonstrated to run robustly for extended time periods.

A.2 Software

A.2.1 Running the System

Start the Code

To start all of the processes for autonomous driving, run each of the commands in separate terminals:

```
$ drive # launches joystick, connection to robot's MCU board
$ control # launches sensors, localization, control, etc.
$ nn # launches CADRL (neural network) planner
```

These are aliases for `roslaunch` commands which can be found in the `/.bashrc` file.

Then, to start the realsense sensors on the Nuc, run this in a new terminal:

```
$ ssh nuc-jackal
$ rs # launch 3 realsense cameras
```

Set a Goal Position

It is easy to set a new goal position for the jackal to drive to. After starting all the nodes, it takes 10-20 seconds for the diffusion map to be loaded (since it is a large file). Then, click the 2D Nav Goal button on the top bar, and click a position in the white part of the map. If the goal was picked successfully, a set of blue lines will extend from the yellow vehicle, indicating the pruned motion primitives. Sometimes this process fails because

- the diffusion map hasn't loaded.
  - Wait longer
- a point outside of the diffusion map is selected
  - Select a new point. A warning is shown in the terminal. Note that the diffusion map does not perfectly line up with the white/black static occupancy grid map, so some points near edges may not have a valid diffusion cost.

There are other ways to set the goal position. The path planner subscribes to the `/move_base_simple/goal` topic, so any node that publishes this can set a new goal.

A node is provided that randomly selects a goal from a set of pre-defined positions. The positions are defined in the file `~/ford_ws/src/ford/jackal/goal_setter/param/random_goals.yaml`. The first indentation determines the map that the goals are associated with (different areas of MIT campus, Ford campus, etc.). The letters can be changed to strings that describe the goal if desired. The positions are [x,y] positions on the map. The easiest way to set up new goal positions is to drive the robot to a desired position, run:

```
$ rostopic echo /JA01/pose/position
```

and add that position to the .yaml file. The `random_goal` node is started as:

```
$ random_goal
```

This node demonstrates the robot's ability to drive for extended periods of time without any human input.

Instead of random goals, the user sometimes wants to manually set a new position from that list of goals. A node for this purpose is run with:

```
$ goal
```

A python script will prompt the user to enter one of the letters or strings associated with a goal in the .yaml file, and after entering the name and pressing enter, it will set that goal. When the robot gets close to the new goal, it will stop and prompt the user for a new goal. This node is useful to demonstrate the robot's ability to navigate around a series of stations in a known area, according to instructions from an external source.

In testing, the author found it much easier to enter a goal position by typing one letter, than using rviz to scroll around the map and select an exact point. The former method also guarantees the goal position will be on the diffusion map since it was verified upon creation of the `goals.yaml` file.

Follow a Person

Since any node can set the goal by publishing to the `/move_base_simple/goal`, the author created a node to follow a person by publishing the goal as a person's current position.

The node is run with the command

```
$ roslaunch goal_setter follow_goal.launch
```

The node's algorithm is described in a previous section of this thesis. The node has its limitations, because if the person is occluded it may lose track of them, or if multiple people cross paths the robot may switch between which person it is tracking.

In practice, the node works pretty well when there is one person in front of it walking at a reasonable speed.

Further improvements could include computer vision techniques so the robot recognize certain features on a person that could limit the confusion between multiple people in nearby position, and could help if the robot loses track of its leader.

Generate a Map

There are 3 steps to generate a new map. The first is the occupancy grid map, which outputs a grayscale image. The second is to generate a diffusion map from the occupancy grid map. The third is to update the launch file so that it points to the new map.

Occupancy Grid Map

Run the following commands in new terminals:

```
$ drive
$ roslaunch ford_ros jackal_map_maker.launch
```

Then drive the robot around and build up a good map. The map_maker launch file starts the sensors (velodyne) and the SLAM node, so be sure these are not commented out in the launch file code.

After the area is mapped, save the map using the command:

```
$ roscd ford_ros/maps/<map_folder_name>
$ rosrun map_server map_saver -f <map_filename>
```

The `map_saver` command saves a `.pgm` grayscale image of the map, and a `.yaml` file with map parameters such as the origin, resolution, and associated image filename.

Diffusion Map

The code to generate a diffusion map is written in MATLAB, and the files are converted to python so that ROS can use them. This process is not fully automated,

Table A.3: ROS Launch Files

	Contents
jackal_control.launch	path follower, costmap, diffusion map planner
jackal_localization.launch	amcl, robot pose publisher
jackal_ped_tracking.launch	laserscan map filter, scan clustering, VeryFast ped detector, cluster classifier, ped manager, follow_me
jackal_rviz.launch	rviz visualization software
jackal_sensors.launch	webcams, velodyne, sonars, transforms

so the easiest method is to email the grayscale image to the author to generate the diffusion map.

There is also a readme file located at ~`/ford_ws/src/diffusion_maps/readme.txt` which provides instructions to generate a diffusion map.

Launch File

The ~`/ford_ws/src/ford/ford_ros/launch/jackal.launch` file should be updated to include the new maps. A new `arg` can be created at the top of the file (using the examples for MIT buildings). Then, a new entry for the map and diffusion map filenames should be added.

A.2.2 ROS Launch File Structure

The launch files are organized so that related nodes are grouped together. Launch files are contained in the directory ~`/ford_ws/src/ford/ford_ros/launch/jackal`. See Table A.3 for a description of what each launch file contains. The `jackal.launch` file launches all the sub-launch files listed in the table, and the map server. It is located at ~`/ford_ws/src/ford/ford_ros/vehicles/jackal.launch`. The transforms (between reference frames of sensors) file is also in the `vehicles` directory.

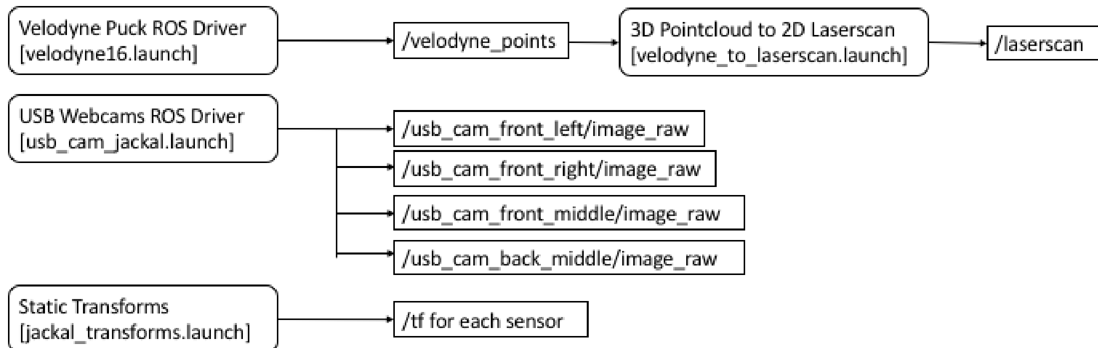


Figure A-8: Software Diagram: Perception on Brix

A.2.3 Software Diagrams

The software diagrams in this section provide broad input/output descriptions of the many software nodes. They are somewhat simplified to show how the data generally transfers between systems. A true understanding of the software involves a thorough read of the actual code. As an example, the robot's position is often omitted from the block diagrams, even though the code may use it for a minor calculation.

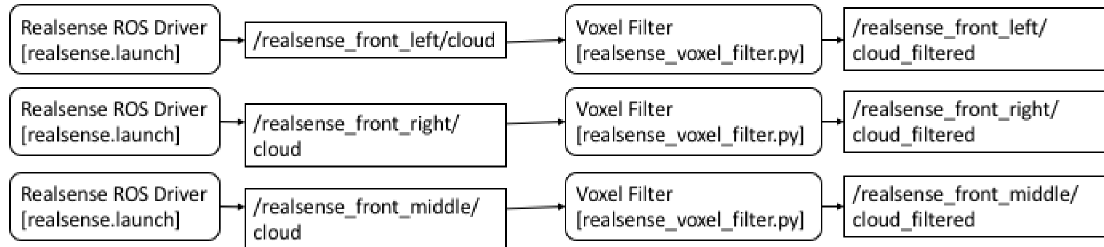
A.2.4 Miscellaneous Software Notes

The two computers are connected through an Ethernet cable, and are both on the "Jackal Local" network with static IP addresses. Each computer can also be connected to the Internet with wifi in addition to this local connection.

Each computer has these IP addresses in the `/etc/hosts` file, so the following commands allow the user to open the other computer's terminal:

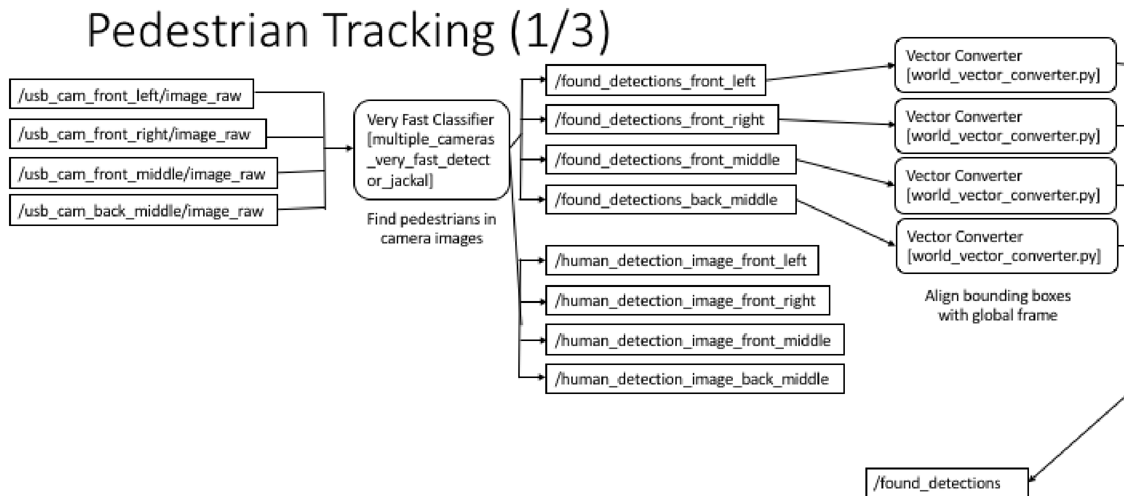
```
$ ssh nuc-jackal # from brix
$ ssh gigabyte-jackal # from nuc
```

Perception (Runs on Nuc)



~/ford_ws/src/acl_sensors/scripts/launch_mult_realsense.sh

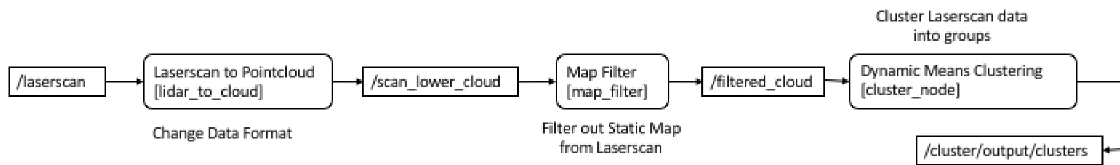
Figure A-9: Software Diagram: Perception on Nuc



~/ford_ws/src/ford_ros/launch/jackal/jackal_ped_tracking.launch

Figure A-10: Software Diagram: Pedestrian Tracking 1/3

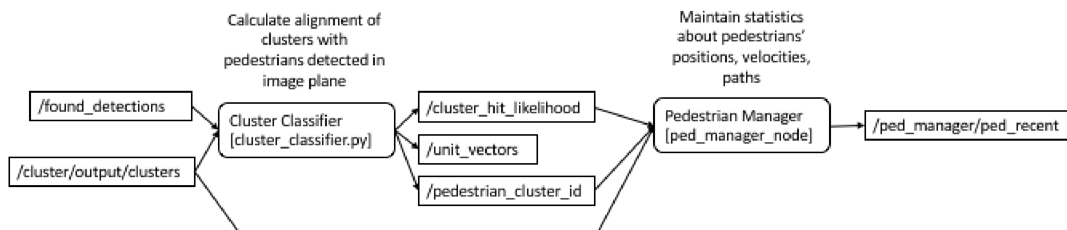
Pedestrian Tracking (2/3)



~/ford_ws/src/ford_ros/launch/jackal/jackal_ped_tracking.launch

Figure A-11: Software Diagram: Pedestrian Tracking 2/3

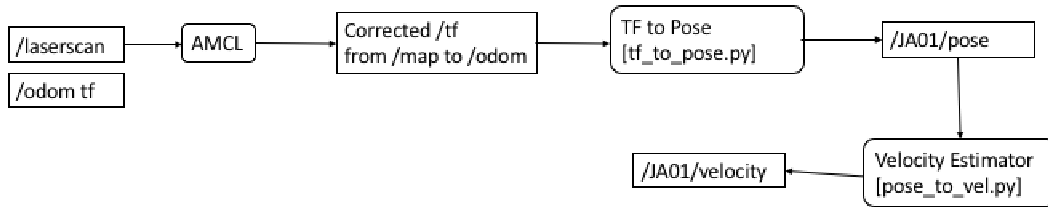
Pedestrian Tracking (3/3)



~/ford_ws/src/ford_ros/launch/jackal/jackal_ped_tracking.launch

Figure A-12: Software Diagram: Pedestrian Tracking 3/3

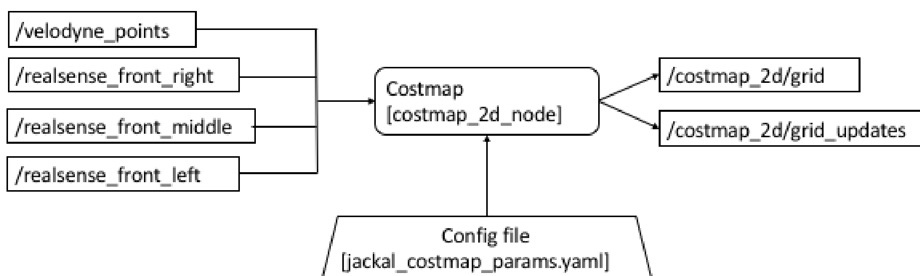
Localization



`~/ford_ws/src/ford_ros/launch/jackal/jackal_localization.launch`

Figure A-13: Software Diagram: Localization

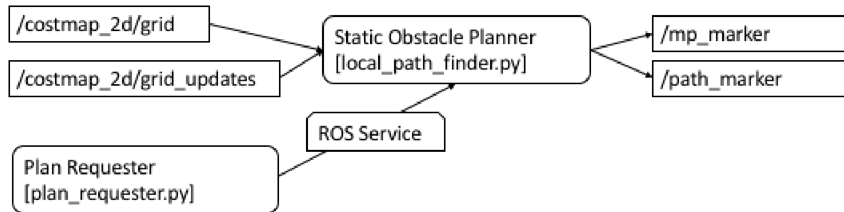
Costmap



`~/ford_ws/src/ford_ros/launch/jackal/jackal_control.launch`

Figure A-14: Software Diagram: Costmap

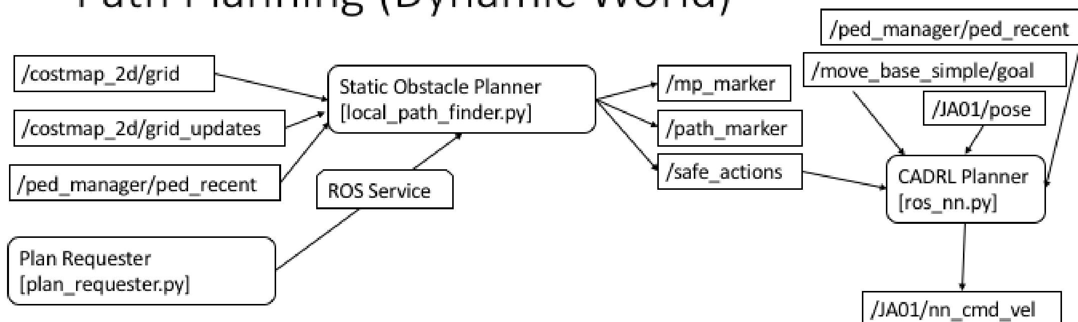
Path Planning (Static World)



~/ford_ws/src/ford_ros/launch/jackal/jackal_control.launch

Figure A-15: Software Diagram: Planner (Static World)

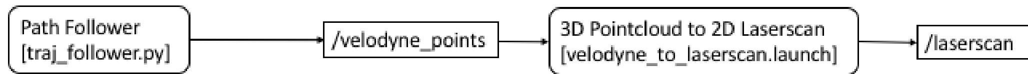
Path Planning (Dynamic World)



~/ford_ws/src/ford_ros/launch/jackal/jackal_control.launch

Figure A-16: Software Diagram: Planner (Dynamic World)

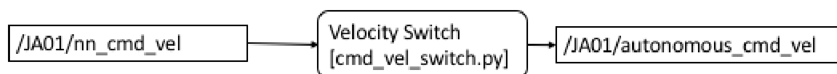
Control (with Static World Planner)



~/ford_ws/src/ford_ros/launch/jackal/jackal_control.launch

Figure A-17: Software Diagram: Control (Static World)

Control (with Dynamic World Planner)



~/ford_ws/src/ford_ros/launch/jackal/jackal_control.launch

Figure A-18: Software Diagram: Control (Dynamic World)

Table A.4: Computer Connection Info

	IP Address	user@hostname	Ubuntu	ROS
Brix	192.168.0.101	swarm@gigabyte-jackal	14.04	Indigo
Nuc	192.168.0.102	swarm@nuc-jackal	16.04	Kinetic

Since the computers have the same username, `swarm`, no username is necessary in the `ssh` command.

The Brix connects to the Jackal hardware. The Nuc is onboard to process the realsense and leave room for future computation. There are too many USB devices to connect everything to one computer. Even though the Brix has 4 USB 3.0 ports, they are all connected to the same bus.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Genius Webcam. http://cdn.toptenreviews.com/rev/scrn/medium/56302-genius-wide-cam-f1001.jpg [accessed Feb 26, 2017].

[2] RSE Lab Washington. https://rse-lab.cs.washington.edu/projects/mcl/animations/global-floor.gif [accessed Feb 27, 2017].

[3] Sanghyuk Park, John Deyst, and Jonathan P How. Performance and lyapunov stability of a nonlinear path following guidance method. *Journal of Guidance, Control, and Dynamics*, 30(6):1718–1728, 2007.

[4] RobotShop. http://robotshop.com/letsmakerobots/files/userpics/u17353/diffdrive.jpg [accessed Apr 2, 2017].

[5] J. van den Berg, Ming Lin, and D. Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. In *IEEE International Conference on Robotics and Automation, 2008. ICRA 2008*, pages 1928–1935, May 2008.

[6] Corinne Purtill. https://qz.com/893068/a-robot-that-isnt-awkward/ [accessed Mar 11, 2017].

[7] Hubway. https://www.thehubway.com/about/media-kit [accessed April 15, 2017].

[8] YouTube. https://i.ytimg.com/vi/Q4mFi1CSsFU/maxresdefault.jpg [accessed Mar 14, 2017].

[9] Rainer Kummerle, Michael Ruhnke, Bastian Steder, Cyrill Stachniss, and Wolfram Burgard. Autonomous robot navigation in highly populated pedestrian zones. *J. Field Robot.*, 32(4):565–589, June 2015.

[10] Jackrabbot. https://2e0a24317f4a9294563f-26c3b154822345d9dde0204930c49e9c.ssl.cf1.rackcdn.com/9888126_jackrabbot-a-social-robot-for-crowded-spaces_1a064fec_m.jpg?bg=827691 [accessed Mar 14, 2017].

[11] YouTube. https://www.youtube.com/watch?v=RY46hXqKcpU [accessed Mar 14, 2017].

[12] Starship. http://www.roboticstrends.com/images/wide/swiss-post-starship-technologies.jpg [accessed Mar 14, 2017].

[13] Alexandre Robicquet, Amir Sadeghian, Alexandre Alahi, and Silvio Savarese. *Learning Social Etiquette: Human Trajectory Understanding In Crowded Scenes*, pages 549–565. Springer International Publishing, Cham, 2016.

[14] Alexandre Alahi, Kratarth Goel, Vignesh Ramanathan, Alexandre Robicquet, Li Fei-Fei, and Silvio Savarese. Social lstm: Human trajectory prediction in crowded spaces. In *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition*, 2016.

[15] Dhanvin Mehta, Gonzalo Ferrer, and Edwin Olson. Autonomous navigation in dynamic social environments using multi-policy decision making. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, October 2016.

[16] Dhanvin Mehta, Gonzalo Ferrer, and Edwin Olson. Fast discovery of influential outcomes for risk-aware MPDM. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, May 2017.

[17] YouTube Obelix. `https://www.youtube.com/watch?v=WIsnTCtDcUM` [accessed April 15, 2017].

[18] Tim J Gates, David A Noyce, Andrea R Bill, Nathanael Van Ee, and TJ Gates. Recommended walking speeds for pedestrian clearance timing based on pedestrian characteristics. In *Proceeding of TRB 2006 Annual Meeting*, 2006.

[19] Velodyne. `http://velodynelidar.com/images/news/family.png` [accessed Feb 26, 2017].

[20] Velodyne. `http://informedinfrastructure.com/wp-content/uploads/2014/11/Velodyne_SVYP_LiDAR-Audience.jpeg` [accessed Feb 26, 2017].

[21] RPLidar. `http://www.robotshop.com/media/catalog/product/cache/1/image/900x900/9df78eab33525d08d6e5fb8d27136e95/r/p/rplidar-a2-360-laser-scanner-29.jpg` [accessed Feb 28, 2017].

[22] Andres Hasfura. Pedestrian detection and tracking for mobility on demand. Master's thesis, Massachusetts Institute of Technology, 2016.

[23] Intel. `http://click.intel.com/media/catalog/product/cache/1/image/9df78eab33525d08d6e5fb8d27136e95/d/s/ds4_base_1_1.jpg` [accessed Feb 26, 2017].

[24] Intel. `https://software.intel.com/sites/default/files/generating-3d02.jpg` [accessed Feb 26, 2017].

[25] Brett T. Lopez and Jonathan P. How. Aggressive 3-d collision avoidance for high-speed navigation. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017. Manuscript submitted for publication.

[26] Georges Younes, Daniel C. Asmar, and Elie A. Shammas. A survey on non-filter-based monocular visual SLAM systems. *CoRR*, abs/1607.00470, 2016.

[27] Trevor Campbell, Miao Liu, Brian Kulis, Jonathan P. How, and Lawrence Carin. Dynamic clustering via asymptotics of the dependent dirichlet process. In *Advances in Neural Information Processing Systems (NIPS)*, 2013.

[28] R. Benenson, M. Mathias, R. Timofte, and L. Van Gool. Pedestrian detection at 100 frames per second. In *CVPR*, 2012.

[29] Yu Fan Chen, Shih-Yuan Liu, Miao Liu, Justin Miller, and Jonathan P. How. Motion planning with diffusion maps. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016.

[30] Yu Fan Chen, Miao Liu, Michael Everett, and Jonathan P. How. Decentralized non-communicating multiagent collision avoidance with deep reinforcement learning. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2017.

[31] Yu Fan Chen, Michael Everett, Miao Liu, and Jonathan P How. Socially aware motion planning with deep reinforcement learning. *arXiv preprint arXiv:1703.08862*, 2017.

[32] Yufan Chen, Shih-Yuan Liu, Miao Liu, Justin Miller, and Jonathan P. How. Motion Planning with Diffusion Maps. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Daejeon, Korea, October 2016.

[33] Gregory Dudek and Michael Jenkin. *Computational Principles of Mobile Robotics*. Cambridge University Press, New York, NY, USA, 2000.

[34] Dronethusiast. `http://www.dronethusiast.com/drones-that-follow-you/` [accessed Apr 4, 2017].

[35] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single shot multibox detector. In *ECCV*, 2016.

[36] Jur Van Den Berg, Stephen J Guy, Ming Lin, and Dinesh Manocha. Reciprocal n-body collision avoidance. In *Robotics research*, pages 3–19. Springer, 2011.

[37] Sujeong Kim, Stephen J. Guy, Wenxi Liu, Rynson W. H. Lau, Ming C. Lin, and Dinesh Manocha. *Predicting Pedestrian Trajectories Using Velocity-Space Reasoning*, pages 609–623. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[38] J. Bellingham, A. Richards, and J. P. How. Receding horizon control of autonomous aerial vehicles. In *American Control Conference (ACC)*, volume 5, pages 3741–3746, 2002.

[39] Julio Godoy, Ioannis Karamouzas, Stephen J Guy, and Maria Gini. Anytime navigation with progressive hindsight optimization. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 730–735. IEEE, 2014.

[40] J. Alonso-Mora, A. Breitenmoser, M. Rufli, P. Beardsley, and R. Siegwart. Optimal reciprocal collision avoidance for multiple non-holonomic robots. In *Proc. Int. Symp. on Distributed Autonomous Robotics Systems*, 2010.