Localization and 2D Mapping Using Low-Cost Lidar

Master of Science in Technology
University of Turku
Department of Future Technologies
Embedded Computing
2018
Yuan Zhang

Supervisors:
Ethiopia Nigussie, Adj. Prof.
Nanda Thanigaivelan, M.Sc.

UNIVERSITY OF TURKU
Department of Future Technologies

YUAN ZHANG: Localization and 2D Mapping Using Low-Cost Lidar

Master of Science in Technology, 67 p., 31 app. p.
Embedded Computing
December 2018

Autonomous vehicles are expected to make a profound change in auto industry. An autonomous vehicle is a vehicle that is able to sense its surroundings and travel with little or no human intervention. The four key capabilities of autonomous vehicles are a comprehensive understanding of sensor data, knowledge of their positions in the world, building the map of unknown environment, as well as following the planed route and collision avoidance.

This thesis is aimed at building a low-cost autonomous vehicle prototype that is capable of localization and 2D mapping simultaneously. In addition, the prototype should be able to detect obstacles and avoid collision. In this thesis, a Redbot is utilized as a moving vehicle to evaluate collision avoidance functionality. A mechnical bumper in front of the Redbot is used to detect obstacles, and a remote user can send appropriate commands to control the Redbot via Zigbee network, then Redbot acts accordingly, including driving straightly, changing direction to right or left, and stop. Redbot are also used to carry the lidar scanner which consists of Lidar Lite V3 and a servo motor. Lidar data are sent back to a Laptop running ROS via Zigbee network. In ROS, $Hector\_SLAM$ metapackage is adopted to process the lidar data, and realize the functionality of simultaneous localization and 2D mapping.

After implementing the autonomous vehicle prototype, a series of tests are conducted to evaluate the functionality of localization, 2D mapping, obstacle detection, and collision avoidance. The results demonstrated that the prototype is capable of building usable 2D maps of unknown environment, simultaneous localization, obstacle detection and collision avoidance in time. Due to the limited scan range of the low-cost lidar scanner, boundary missing problem can happen. This limitation can be solved through the use of a lidar scanner with larger scan range.

Keywords: autonomous vehicles, 2D lidar, localization, 2D mapping, SLAM, ROS, obstacle detection and avoidance

Contents

List of Figures

List of Tables

List of Acronyms

AV	Autonomous Vehicle
GPS	Global Positioning System
IMU	Inertial Measurement Unit
SLAM	Simultaneous Localization and Mapping
RTLS	Real-Time Locating System
ROS	Robot Operating System
DARPA	Defense Advanced Research Projects Agency
SAE	Society of Automotive Engineers
ACC	Adaptive Cruise Control
MIT	Massachusetts Institute of Technology
SARTRE	Safe Road Trains for the Environment
SoC	System on a Chip
ASIC	Application Specific Intergrated Circuit
FPGA	Field Programmable Gate Array
API	Application Programming Interface

IoT	Interner of Things
RFID	Radio Frequency Identification
WSN	Wireless Sensor Network
IIoT	Industrial Internet of Things
WLAN	Wireless Local Area Networking
UHF	Ultra High Frequency
SHF	Super High Frequency
ISM	Industrial Scientific and Medical
WEP	Wired Equivalent Privacy
WPA	Wi-Fi Protected Access
WPA2	Wi-Fi Protected Access II
BLE	Bluetooth Low Energy
SIG	Special Interest Group
UGV	Unmanned Ground Vehicle
UAV	Unmanned Aerial Vehicle
AUV	Autonomous Underwater Vehicle
BSD	Berkeley Software Distribution
SRR	Short Range Radar
MRR	Medium Range Radar
LRR	Long Range Radar

AEBS	Automatic Emergency Braking Systems
SSL	Solid State Lidar
MAP	Maximum a Posteriori Probability
IR	Infrared Reflectance
PWM	Pulse Width Modulated

Chapter 1

Introduction

Autonomous vehicles (AVs) are expected to make a great change in our transportation system. It has potential influence on traffic safety, congestion and way of traveling[4]. Giants in automotive industry like Daimler, new technology companies like Google and Apple, and educational laboratories are all working on the research and development of autonomous vehicles[5].

One of the major challenges for autonomous vehicle designers is the use of data from different sensors to control the vehicles. Typical sensors used on AVs are lidar, radar, ultrasonic sensors, GPS, inertial measurement unit (IMU) and stereo vision[6][7]. Generally, modern AVs use simultaneous localization and mapping (SLAM) algorithms[8] to detect the surrounding environment and other moving vehicles. Other technologies like real-time locating system (RTLS) and Sensor Fusion[1] are also used in simpler systems.

Mobile robots and robot operating system (ROS) are widely used in R&D phases of AVs. Mobile robots refer to the robots that can move around. Some of the mobile robots are capable of navigating without any guidance while others need the pre-defined route or some guidance devices. With the help of ROS, researchers and engineers can test more algorithms on the mobile robots instead of starting from “reinventing the wheels”.

In this thesis, a mobile robot called RedBot and a single lidar called Lidar-Lite V3 are employed for 2D SLAM system. The hector-SLAM package provided by ROS is adopted to realize the mapping and localization functionalities. The main idea is to develop a low cost system that can prototype the AVs with navigation and mapping functionalities. The following sections describe the motivation, and the research objectives of the thesis in detail.

1.1 Motivation

The initial motivation of this thesis is to prototype an AV with basic functionalities. AVs, also known as self-driving cars, are one of the dramatic innovations in auto industry. It will not only change people's way of traveling, but also affect the way we perceive the world. Prototyping AVs can help engineers to evaluate and test the designs, including hardware and software. As a student of embedded systems, I want to apply what I have learned on such an important process.

After doing some literature research on AVs, I found the key point of AVs is to let the vehicles be aware of the surrounding area as human do, and it is realized by two technology: computer vision and sensor fusion. Among the sensors used by the vehicles, lidar is used most. Besides, lidar is easily accessible to me. Therefore, my second motivation is to apply low cost lidar on controlling the AVs.

SLAM is the algorithm widely used to localize the vehicles with the given sensor data. SLAM can help the vehicles to know where they are in the world. SLAM can also work properly even without IMU and GPS. Hence, my another motivation is to employ SLAM algorithm on vehicles without IMU or places with no GPS.

The last motivation is to use limited resources to prototype an AVs for research purposes. Under some circumstances, we only have restricted fundings and research equipments, a low-cost prototype can solve this problem while achieving the research goals like verifying the improved algorithms or testing the sensor performance.

1.2 Research Objectives

This thesis aims at building a low cost 2D lidar SLAM system using a mobile robot and ROS. The system can produce the 2D map of the surrounding environment of the mobile robot using the lidar data. The mobile robot can avoid the obstacles autonomously and move according to the remote control commands. The system operates wirelessly.

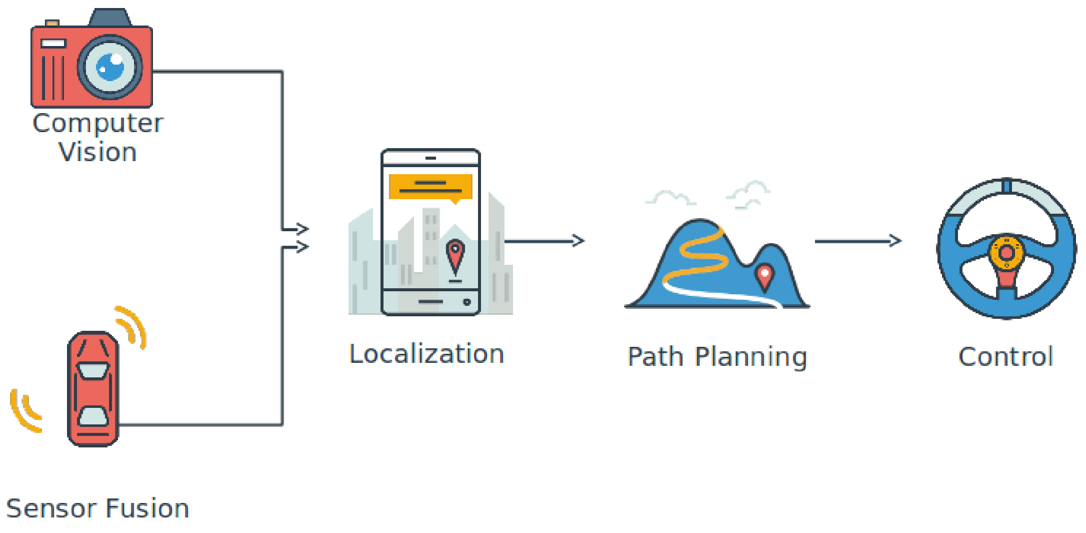


Figure 1.1: AVs' Working Process[1]

There are five typical research areas in AVs (see Figure 1.1):

1. Computer vision: to enable the AVs to see the world as human do;
2. Sensor fusion: to integrate the data from different sensors, and to make the AVs have a comprehensive understanding of the surrounding area;
3. Localization: to let the AVs know where they are in the world with high accuracy;
4. Path Planning: to calculate the best route to where they want to go;
5. Control: to control the AVs to follow the planned trajectory, including avoiding obstacles.

In this thesis, the latter four steps mentioned above are focused. The hardware selected and the technologies used are listed below:

- Redbot from SparkFun Electronics: The mobile robot to move the lidar sensor around.
- Lidar-Lite v3 from GARMIN: The light detection and range sensor to provide the sensor data.
- Arduino Mega: The controller to control the servo and interface the lidar sensor.
- Servo MG995: The servo to rotate the lidar sensor horizontally.
- Xbee pro S1 from DIGI: The Xbee module to form the Zigbee networks for transmitting the sensor data and interfacing the mobile robot wirelessly.
- ROS: The robotics middleware to process the sensor data, map the surrounding area and control the mobile robot.
- SLAM: The algorithm to localize the mobile robot and produce the 2D map according to the sensor data.

1.3 Thesis Structure

This thesis is organized into five chapters. Chapter 1 introduces the general topics, research motivation and objectives of the thesis. Chapter 2 explains the technologies behind AVs briefly and compares the differences among alternative choices of the mobile robots, ROS versions, sensors and SLAM packages. Chapter 3 takes a deeper look into the design and implementation of the system, including the mobile robot, lidar sensor, ROS software and the obstacle detection and avoidance with remote control. Chapter 4 evaluates the system performance with respect to the 2D mapping and obstacle detection and avoidance. This chapter also discusses the limitations of the system. Chapter 5, as the last chapter, summarizes the thesis and suggests some future work.

Chapter 2

Background

This chapter presents the detailed description of the research topics and technologies used in this thesis. This chapter consists of seven sections. The first three sections are the general research area focused in this thesis, i.e. autonomous vehicles, embedded systems and Internet of Things. Meanwhile, the latter four sections describe the specific technologies and hardware used in this thesis, and compare the alternative technologies that can also be employed.

2.1 Autonomous Vehicles

The early experiments on autonomous driving have started since the 1920s[9] and emerging prototypes appeared in the 1950s. More autonomous prototypes came out in the 1980s, including Carnegie Mellon University's Navlab[10] and ALV[11][12] projects in 1984 and Mercedes-Benz and Bundeswehr University Munich's Eureka Prometheus Project[13] in 1987. The Tsukuba Mechanical Engineering Laboratory developed the first real automated car in 1997, in Japan. The Tsukuba automated car was capable of tracking the white street markers and marching at the speed of 30 kilometers per hour. Many companies and R&D groups have built prototypes since the second DARPA Grand Challenge in 2005[13][14][15][16][17]. The U.S. National Automated Highway System program demonstrated a combination of automated vehicles and highway network successfully in 1997 but only on a small scale[18]. Navlab drove 98% of 4501 kilometers autonomously across America in 1995[19], and the record was not broken until 2015 when an Audi drove 99% of over 5472 kilometers with the help of Delphi technology[20]. Many states of the US allowed tests on public roads after that[21]. One recent break was that Waymo announced its autonomous vehicles had traveled for over 13,000,000 kilometers in July 2018[22].

In the autonomous vehicle industry, many terminologies are used to describe the vehicles that can drive themselves. It can cause many safety problems when a fully autonomous self-driving car and one that needs driver assistance technologies are confused by the users[23]. Autonomous, automated, cooperative and self-driving are most used among the terminologies. Autonomous refers to fully self-governing[24] while automated stresses the artificial aids. One approach to realize autonomous is to build communication networks among vehicles. Cooperative stands for the system that has a remote driver. To standardize the degree of automation, Society of Automotive Engineers (SAE) International published a classification system in 2014 and updated it in 2016[25]. SAE's classification defines six automation levels, ranging from fully manual to fully autonomous vehicles. The brief description of each level is given below[25]:

- Level 0: The system handles warnings but has no lasted vehicle control.
- Level 1: The driver and the system work together, for example, adaptive cruise control (ACC) and parking assistance.
- Level 2: The system controls accelerating, braking and steering, but the driver has to monitor the process and be ready to take over the control whenever the system fails.
- Level 3: The driver can turn attention to text or watching a movie when driving, which is called "eyes off". In 2018, Audi A8 Luxury Sedan was the first commercial

car with level 3 automation, but it only works for one-way traffic on high-ways.

- Level 4: No more safety issues. The driver can sleep or sit in the passenger's seat.
- Level 5: No human involvement any more. One possible example is the robotic taxi.

Apart from the ambiguous terminology and classification issue, there are many other challenges to be solved. Technically, the main challenge for designers is to make the AVs to understand the surrounding environment, including other vehicles and the road with the sensory data input, as human do. Generally, SLAM and RTLS are the mainstream algorithms used. Typical sensors used to provide data for these algorithms are lidar, stereo vision, GPS and IMU. Sensor fusion, which integrates data from different sensors, is employed to decrease the uncertainty when each sensor is used individually. Autonomous vehicles also apply deep learning for visual object recognition. Deep neural networks, which is an approach to simulate neurons that activate the network[26], are used to develop the autonomous cars[6]. Recently, one big progress has been made that researchers in MIT announced that their automated car can navigate unmapped roads with the system called MapLite[27]. The system employs GPS, OpenStreetMay and many sensors to control the vehicles[28]. From human side, one big challenge is that human need to build up more confidence on the vehicles with the elevated automation level, .

There are two main fields of application in autonomous vehicles. One is automated trucks. Uber's self-driving truck startup Otto demonstrated their products on the highway in August 2016[29]. Embark, a startup in San Francisco, also announced their collaboration with truck manufacturer Peterbilt in May 2017[30]. Besides, Waymo was said to have a project on self-driving trucks[31]. In March 2018, Starsky Robotics, as the first player to drive in autonomous mode without a driver in the trucks, completed a 11-kilometer trip in Florida. In Europe, Safe Road Trains for the Environment (SARTRE) project is deploying truck Platooning[32]. Another application is in transport systems. In Europe, countries like Belgium, the UK, France, the Netherlands, Spain, Germany and Italy have allowed testing in public traffic, and are planning the transport systems for autonomous vehicles[33][34][35].

The motivation behind the fast development progress of autonomous vehicles is its potential advantages with regard to the following factors:

- Safety: Experts predict that traffic accidents like head-on, road departure, rear-end, side collisions and rollovers that caused by human errors can be substantially reduced with the wider use of AVs[36]. McKinsey estimated 90% accidents can be eliminated and thousands of lives and money can be saved[37].
- Welfare: AVs can reduce labor costs[38] and leave travelers with more leisure time[36]. It also increases the mobility of the young, the elderly and the disabled[39]. With the removal of the steering wheel, the space inside the cabin can be more cozy and flexible[40].
- Traffic: There can be higher speed limits, larger road capacity and less traffic congestion[41]. The traffic flow can be easier to manage even with less traffic policies and road signals. Besides, more traffic data can be provided to better predict the traffic behavior.
- Lower costs: Less traffic accidents can reduce the vehicle insurance costs. Apart from that, higher fuel efficiency can reduce the fuel costs[39].
- Parking space: The need of parking space can be dramatically reduced because the AVs can be used continuously after finishing on journey.
- Related effects: With more and more sharing services of AVs, the number of individually owned cars will be reduced, which can be more environment-friendly[42]. Moreover, illegal behavior like deliberate crash can be avoided.

Despite the above advantages, there are several challenges and limitations of AVs that can hinder its development. One is the liability problem. No consensus has been reached until now on who is responsible for the accidents, the operator company or the one that is in the car[43]. Besides, there are other related government regulations that need to be implemented. There will be a long time that AVs and non AVs are sharing the roads, and this will cause not only technical challenges but also policy problems. Unemployment caused by the wide spread of AVs is also a big challenge. Drivers, workers at repair shops and public transit services will lose their jobs. The other disadvantage is loss of privacy. With the data provided by the third party, anonymous traveling will be impossible, people's daily life will be under surveillance of the AV companies and government[44]. In contrast to one advantage mentioned above, another opinion thinks that AVs can cause more use of fuel polluting the environment even more. People will be inclined to living far away from the cities and working in the city centers due to the travel comfort and convenience[45].

2.2 Embedded Systems

An embedded system is a programmed controlling and operating system. It usually has a dedicated function within an electrical system, and has real-time computing constraints[46]. It is often embedded in devices with hardware and mechanical components. Embedded systems are indispensable in many devices today[47]. 98% of the microprocessors are used in embedded systems[48]. The first modern embedded system was the Apollo Guidance Computer developed by Charles Stark Draper at the MIT. The first embedded system put into mass production was the Autonetics D-17 guidance computer for the Minuteman missile in 1961. After some early applications in the 1960s, embedded systems lowered the price and had an improvement on processing power and functionality. In 1980s, input and output systems and memory were integrated into one chip with microprocessors so that microcontrollers were formed. As the cost fell, microcontrollers took the place of general-purpose computers in many places. Even software prototype and hardware tests are quicker and cheaper in embedded systems compared to redesigning of a circuit.

In comparison with general-purpose counterparts, typical embedded computers have many advantages. It usually has low power consumption, small size, wide operating ranges and low costs at the price of limited processing resources. The programmers need to be more careful with the programming and interaction. But with the help of intelligence mechanisms, such as randomized algorithms and robustness analysis, programmers can utilize sensors and networks to balance the optimal performance and the available resources[49]. Furthermore, embedded systems are designed to do specific tasks, and some have to meet the real-time constraints for safety or usability. Embedded systems can have no user interface at all, or they can even have complex graphical user interfaces that are similar to modern computer desktop. Most embedded devices have buttons, LEDs and LCDs. More complex devices can have graphical screens, and some interfaces can be accessed remotely with a serial or network connection. Embedded systems can be classified into two, the first one is the microprocessors with separate circuits for peripherals and the other one is microcontrollers with on-chip peripherals. For small embedded systems, ready-made computer boards are used, and mostly based on x86. They usually use DOS, Linux or an embedded real-time operating system. For very high volume embedded systems, system on a chip (SoC) based design paradigm is preferred. This kind of system has multiple processors, multipliers, caches and interfaces integrated on a single chip. The typical implementation options for such system is either application-specific integrated circuit (ASIC) or field-programmable gate array (FPGA).

The applications of embedded systems are pervasive throughout everyday life. Components in telecommunications systems, such as telephone switches, cell phones, routers and network bridges, employ numerous embedded systems. Consumer electronics like video game consoles and digital cameras, household appliances such as ovens and dishwashers, and systems like smart home are composed of embedded systems. Transportation systems including planes, trains, and automobiles employ embedded systems to control the mechanical processes and interact with different sensors. Embedded systems are also applied to medical equipment for monitoring vital signals, amplifying sounds of electronic stethoscopes and medical imaging.

Embedded systems apply several types of software architecture. One is the simple control loop. It has a single loop, and some subroutines are called in the loop to manage the hardware or software. Interrupts are also commonly used. For tasks that are simple and need quick responses, interrupts can be triggered by different events, and the event handlers can turn to execute the tasks. Triggers can be a predefined timer or receiving a specific byte. Multitasking systems are used in embedded systems too. One is the cooperation multitasking, also known as non-preemptive multitasking. Similar to the simple loop control, its loop is in an application programming interface (API), and each task runs in its own environment. Another one is the preemptive multitasking, also known as multi-threading. Tasks or threads are switched in a low-level piece of code based on a timer. Many synchronization strategies are used to deal with shared data among different tasks.

2.3 Internet of Things

With the development of autonomous vehicles, the Internet of Vehicle, as a branch of the Internet of Things (IoT), has gained attention[50]. The main idea of IoT is to build a smart network that interconnect “anything”, and can be accessed by “anyone” at “anytime” through “any path” and “any service”[51]. IoT is the network of smart devices like vehicles, household appliances and others with electronics, sensors and wireless modules. The network enables smart devices to connect, collect and exchange data[52]. IoT’s main vision is to build a smart environment that can put more intelligence to the industries, healthcare, cities, transport and many other aspects of our daily life[51].

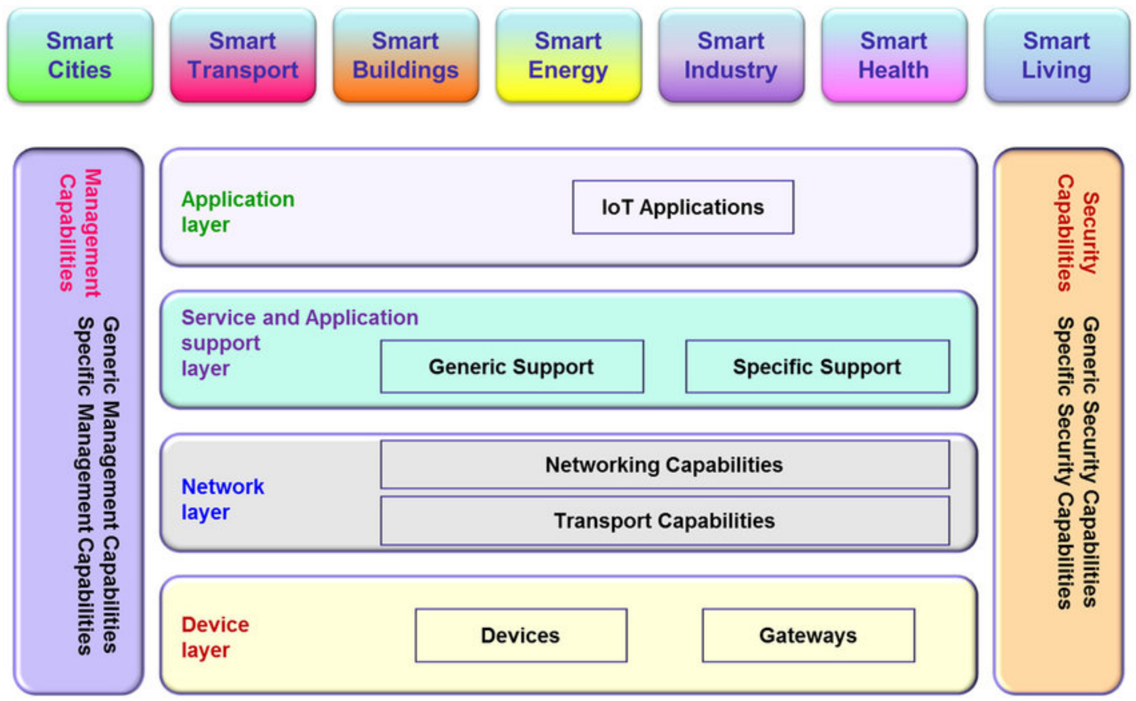


Figure 2.1: IoT Layered Architecture (Source: ITU-T)

As depicted by Figure 2.1, IoT is a new infrastructure with multiple layers. Smart devices deliver their services through four layers: device layer (devices like sensors and RFID, gateways), network layer (transport and networking capabilities for forwarding data to processing centers), service and application support layer (to hide the complex lower layers and provides generic services) and application layer. Each layer has many research topics. Application layer and network layer is introduced in detail in the following sections.

2.3.1 Applications

As illustrated in Figure 2.2, IoT infrastructure can be divided into three dimensions: applications, enabling technologies and general requirements need to be taken into account. IoT has wide applications in our daily life, such as healthcare, energy, building, transport, cities and industries. The enabling technologies include sensors, nanoelectronics,

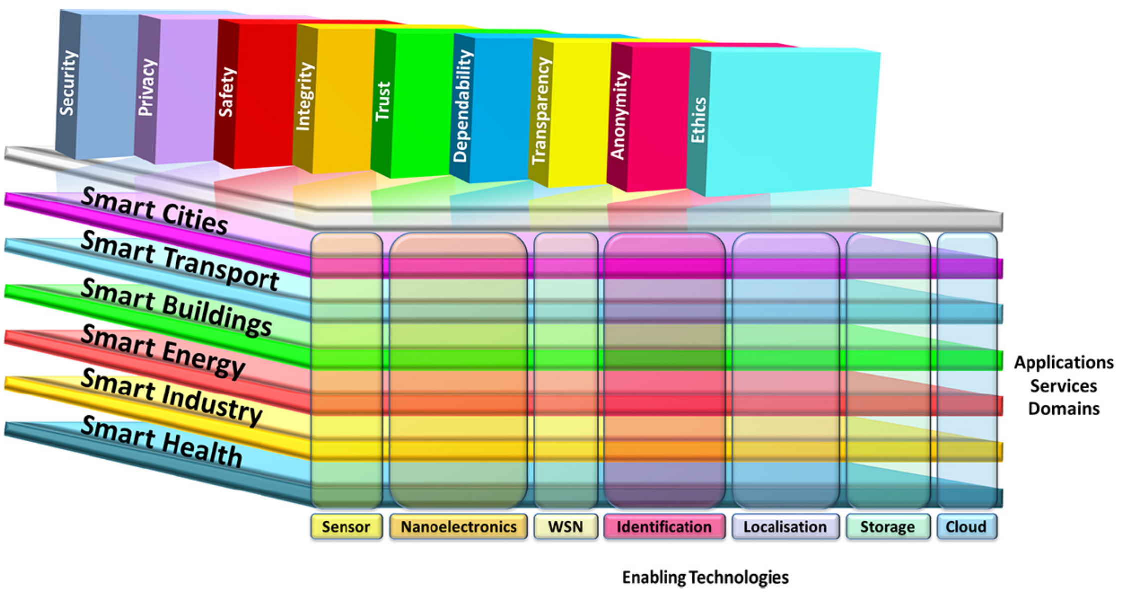


Figure 2.2: IoT 3D Matrix[2]

wireless sensor networks (WSN), identification, localization, storage and Cloud. Some of the main requirements for IoT applications are security, privacy, safety, integrity, trust, dependability, transparency, anonymity and ethics are taken into consideration[2].

Applications in IoT are usually classified into four types: consumer, commercial, industrial and infrastructure[53]. Wearable devices are the typical consumer applications. Others like smart home and elder care also belong to this category. Not as close to our daily life as consumer applications do, commercial applications include Internet of Medical/Health Things for hospitals, Internet of Vehicles for autonomous vehicles and building automation systems. Industrial applications are widely spread in manufacturing and agriculture. In manufacturing industries, it is also known as Industrial IoT (IIoT). In farming, IoT systems utilize many kinds of sensors to collect data on humidity, temperature, wind speed and soil content. The data collected can be analyzed in the systems and assist the farmers to have better decisions, so the risk can be minimized and the quality and quantity of the crops can be improved. In the meantime, infrastructure applications refer to those monitoring and controlling operations on the civil infrastructures like railways and bridges. IoT can benefit the construction by reducing the cost and time, improving the quality and increasing the productivity.

2.3.2 Wireless Sensor Networks

Wireless sensor networks (WSNs) are one of the vital components in the IoT. WSNs not only facilitate the extension of the communication range of IoT networks, but also introduce the computing power of daily necessities. A wireless sensor network refers to a self-configuring network of spatially dispersed sensors (also called motes). These sensors are used to collect data and monitor the environment. The collected data are processed at a central location.

In essence, sensor nodes are small computers with very limited functions. All nodes consist of five main elements: processor, power source, memory, radio and sensors. Processor is in charge of processing locally sensed and forwarded information from other nodes. It usually has three modes: sleep mode to save power, idle mode to be ready for the data from other nodes and active mode when sensing, sending or receiving data. Power source is another important component of sensor nodes. Since many nodes are deployed in remote areas for months, they have to consume as little power as possible. End user can reduce the throughput to extend the lifetime of the power source. Rechargeable batteries, solar panels and capacitors are the common power sources[51]. Memory is used to store the execution program and the collected data. Radio is used to transfer data among nodes. Radio used in WSNs typically has low-rate (10-100 Kbps) and short range (less than 100 meters). Radio communication is a very power-intensive task, so many energy-efficient techniques are adopted to save the power of nodes. Customized algorithms and protocols are adopted to increase the network quality. Three wireless radio protocols are introduced and compared in the following section. Many different sensors are deployed in sensor nodes. There are three main categories: physical, chemical and biological sensors. Sensors can monitor a number of parameters of the ambient environment, such as "temperature, humidity, light, pressure, noise level, acceleration and soil moisture"[51].

There are two different nodes in WSNs. One is called sensor node. It is used to sense the surroundings and transmit the data to another kind of node, i.e. sink node. Sink node is also known as “base station”. Sink nodes collect data from sensor nodes. It can either pass data to the gateway for further processing or function as a gateway itself. A sink node has minimal processing and communication capabilities, but it does not have sensing capabilities. A typical multi-hop wireless sensor network is illustrated in Figure 2.3.

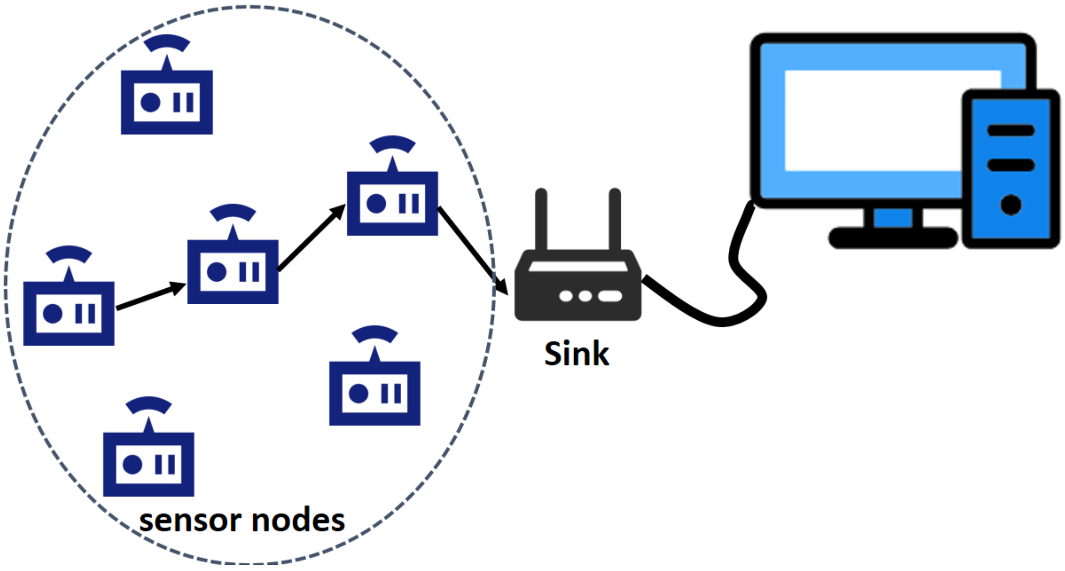


Figure 2.3: Multi-hop Wireless Sensor Network Architecture

2.3.3 Wireless Protocols

The use of wireless technologies is one of the drivers of IoT. Nodes need to utilize wireless protocols to build a communication channel to send and receive data. In this subsection, three wireless communication protocols that are widely used in the IoT field[54] are discussed. For each protocol, a brief description, its pros and cons, its technical features and some application scenarios are introduced.

Wi-Fi

Wi-Fi is a technology developed for device connection of wireless local area networking (WLAN). It is for devices based on the IEEE 802.11 standards. Wi-Fi is the trademark of the Wi-Fi Alliance. It uses the 2.4 gigahertz (12 cm) ultra high frequency (UHF) and 5.8 gigahertz (5 cm) super high frequency (SHF) industrial, scientific and medical (ISM) radio bands. Wi-Fi has several securing methods like wired equivalent privacy (WEP), Wi-Fi protected access(WPA), Wi-Fi protected access II(WPA2). It can be used with or without passwords.

Wi-Fi has been widely used in the IoT because it can utilize the current infrastructure for the new IoT. It works in the range of 10 to 100 meter. The data rate is between 11 to 105 Mbps. It is mostly applied on IoT routers, smart traffic management and office automation. The pros and cons of Wi-Fi for IoT are shown in the Table 2.1.

Table 2.1: Pros & Cons of Wi-Fi

Pros	**Cons**
low cost	high power consumption
easy installation	interference from other devices working in the same bands
scalable	limited range
	less secure

Zigbee

Zigbee is a wireless communication protocol based on IEEE 802.15.4 standard. It is designed for personal area networks with short range, low transfer data rate and low power. Zigbee network is simpler and cheaper than others. The advantages of Zigbee are its low power consumption, its high security, its robustness and its high scalability. It can be utilized with different network topologies, which makes it possible to transmit data over longer distance.

Zigbee operates in 2.4 GHz worldwide, though some devices also use 784 MHz in China, 868 MHz in Europe and 915 MHz in the USA and Australia. Its data rate is 250 Kbps (2.4 GHz band). It is mostly used on monitoring, controlling, medical data collection and wireless sensor networks.

The pros and cons of Zigbee for IoT are shown in the Table 2.2.

Table 2.2: Pros & Cons of Zigbee

Pros	**Cons**
support multiple topologies like mesh, star	low data rate of 250 kbps
long communication range of 1.5 to 2 kilometer	very expensive compared to the license-free protocols
scalable with easy configurations	the frequencies other than 2.4 GHz require license in some countries

Bluetooth Low Energy

Bluetooth low energy (BLE) is a wireless personal area network technology aimed at new applications in the IoT field like healthcare, fitness and home entertainment appliances. Like Bluetooth, BLE is also designed by the Bluetooth Special Interest Group (Bluetooth SIG), but it is intended to reduce energy consumption and cost. BLE has special sleep mode and awake mode to save power consumption, which is very important to IoT devices with limited power supply.

BLE works at 2.4 GHz and the data rate is about 1 Mbps. It is commonly used on mobile phones, smart home systems, wearable devices, healthcare and fitness devices.

The pros and cons of BLE for IoT are shown in the Table 2.3.

Table 2.3: Pros & Cons of BLE

Pros	**Cons**
high data rate of 1Mbps	interference from other devices working in the same band
low power consumption	more initial setup time
medium range of 60 meter	expensive and energy-intensive bluetooth tags

2.4 Mobile Robot

Mobile robots is a subfield of robotics. It refers to robots with locomotion capability. Mobile robots can move themselves around and not stay in a fixed place. Some mobile robots need guidance systems so they can travel in a pre-defined paths, while some are autonomous, i.e. they can navigate in an unknown environment without any help from physical or electro-mechanical guiding devices.

Mobile robots consist of controllers, control software, sensors and actuators. Controllers are mainly microprocessors. Control software can either be assembly language or high-level languages like C, C++ or special real-time software. Dependent on the requirements, the sensors vary from physical ones to chemical ones. Requirements are proximity sensing, collision avoidance, obstacle detection, position location and others.

Mobile robots have become very common in both commercial and industrial scenarios. In hospitals, autonomous mobile robots are used to move materials, and the same use in warehouses. Mobile robots are also one of the research focuses[55]. As consumer products, mobile robots are adopted as entertainment robots or household tools such as vacuum cleaner. Mobile robots' applications can also be found in industrial, military and security field.

Mobile robots can be classified in two ways. One is the environment they travel. Land or home robots are called Unmanned Ground Vehicles (UGVs). Delivery and transportation robots move supplies in a work environment. Robots work in air are named Unmanned Aerial Vehicles (UAVs), while robots work under water are referred to as Autonomous Underwater Vehicles (AUVs). There are also robots specially designed to navigate icy environments called polar robots. Another is the way in which they travel. human/animal-like mechanical legs, wheels or tracks are the three main ways.

Navigation ways are very different among robots. The basic type is a manually tele-operated robot and the robot is completely under control of the driver with control device. Some robots can sense obstacles and avoid collision automatically, which are called guarded tele-op robots. The earliest automated robots followed visual lines to navigate. The autonomously randomized robot can bounce off walls with random motion. With higher automation level, autonomously guided robots can localize themselves and navigate according to the tasks given. The robots with the highest automation level uses a system called sliding autonomy. Three commonly used mobile robots are introduced in the following subsections.

2.4.1 PR2

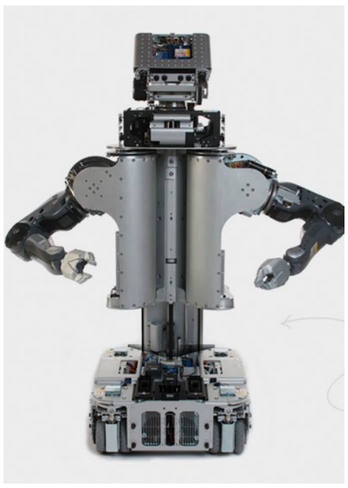


Figure 2.4: PR2

PR2 is a robotic platform for research and development developed by Willow Garage.

The typical model has two arms, a stereo cameras on the top for distance measurement and a tilting laser. As is shown in Figure 2.4, PR2 combines mobility, navigation ability and the ability to grasp and manipulate objects.

2.4.2 TurtleBot

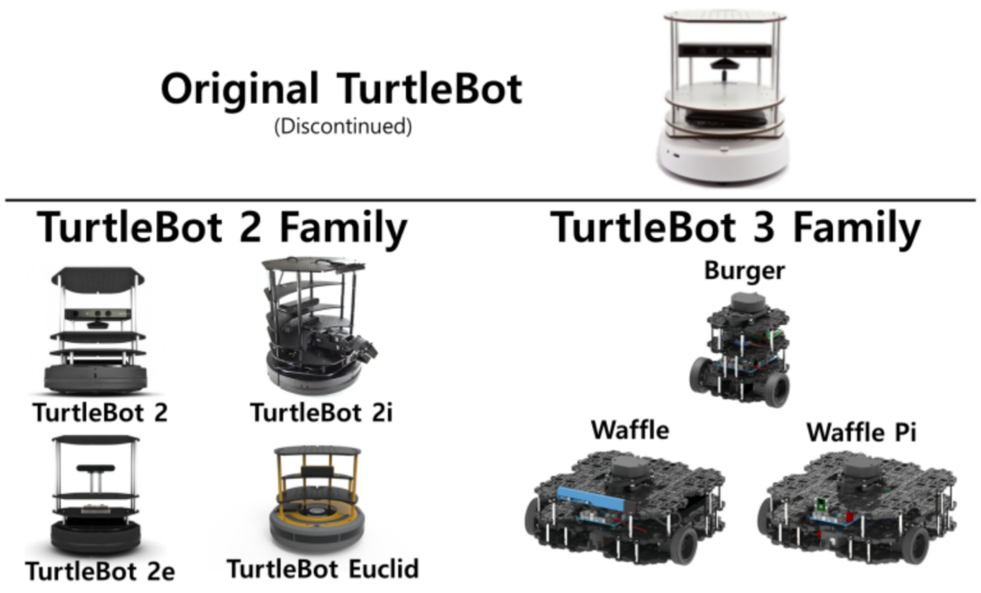


Figure 2.5: TurtleBot Family

TurtleBot is a personal robot kit. It costs relatively lower and it uses open-source software. TurtleBot was created by Melonee Wise and Tully Foote at Willow Garage in November 2010. At first, the creators wanted to build a cheaper platform using ROS, so the first generation of TurtleBot consists of a Kinect and a iRobot Create[56]. TurtleBot provides the basic combination of laptop, sensors and the mobile bases for people work on robots with a reasonable price. It can travel around the house, build a 3D map and create more exciting applications based on the 3D map. Three generations of TurtleBot models are shown in Figure 2.5.

2.4.3 RedBot

RedBot is a platform developed by SparkFun (see Figure 2.6). It integrates the basic robotics and sensors. It is based on the SparkFun RedBoard and can be programmed using Arduino. It consists of top and bottom chassis, two motors and two wheels, two wheel encoders, three line followers, two mechanical bumpers, mainboard, a accelerometer and a buzzer. It is power by four AA batteries. RedBot is the mobile platform used in this thesis, and the details of each part are introduced in Chapter 3.

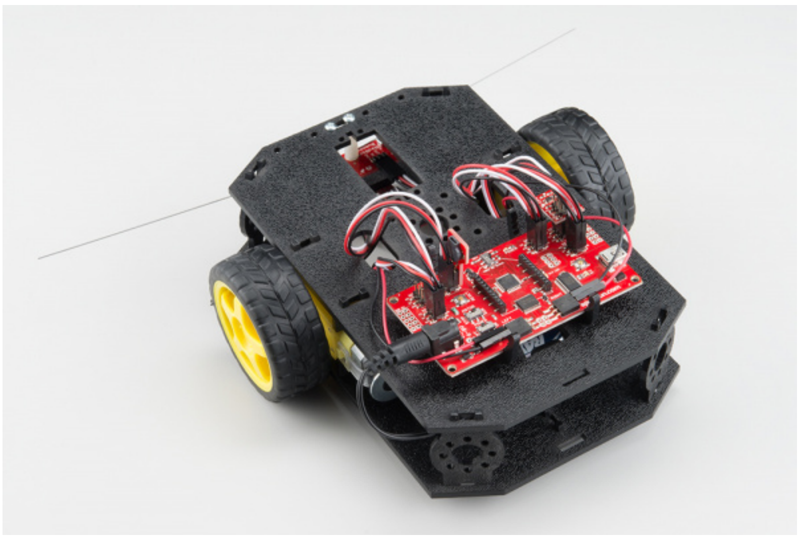


Figure 2.6: RedBot

2.5 Robot Operating System

Robot Operating System (ROS) is robotics middleware. Robotics middleware refers to collection of software frameworks to be used in complex robot control systems, so ROS is not an operating system. It provides several services for heterogeneous computer cluster. It has hardware abstraction and low-level device control. Some commonly used functionality is also implemented. It standardizes message-passing format between processes, as well as manages package. Although it is important to have low latency and real-time reaction, ROS is not a real-time OS (RTOS). The support for real-time systems is provided in ROS 2.0.

ROS has three levels of concepts[3]. The first is the filesystem level. This level covers the resources stored on disk, such as:

- Packages: A package is the main unit to organize software. It can contain runtime processes, libraries, datasets and configuration files. A package is the most atomic thing that can be built and released in ROS.
- Metapackages: Metapackages are special packages. They are used to represent a group of related packages. They are commonly served as a backwards compatible place holder for converted rosbuild stacks.
- Package Manifests: Manifests store matadata about a package, such as its name, version, description, license information and dependencies,
- Repositories: A collection of packages that can be released together with catkin tool bloom.
- Message (msg) types: Definition of the data structures for messages in ROS.
- Service (srv) types: Definition of the request and response data structures for services in ROS.

The second is the Computation Graph level. As is shown in Figure 2.7, there are five components that provide data to the Graph.

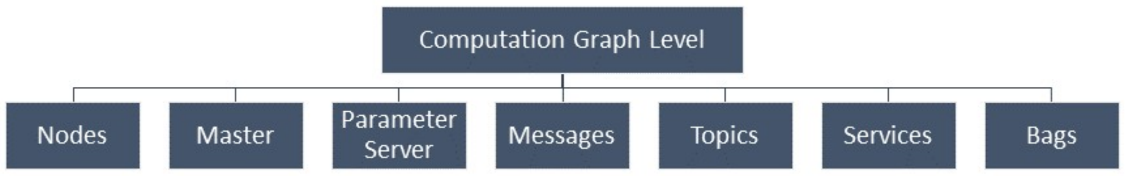


Figure 2.7: ROS Computation Graph Level

- Nodes: Processes performing computation are called Nodes. A robot control system consists of many nodes.
- Master: Each node registers their name and finds each other on Master.
- Parameter Server: It allows data to be stored by key in a central location, and it is part of Master.
- Messages: Communication between processes, i.e. data transferred between processes, are called messages in ROS.
- Topics: Communication tunnels between nodes are referred as topics. Each topic has a unique name. The direction of communication can be set as publish or subscribe. The types of topics are defined to be same as the types of messages.
- Services: ROS service is another way that nodes can communicate. Services allow nodes to send requests and receive responses.
- Bags: Bags are the format to save and play back ROS message data.

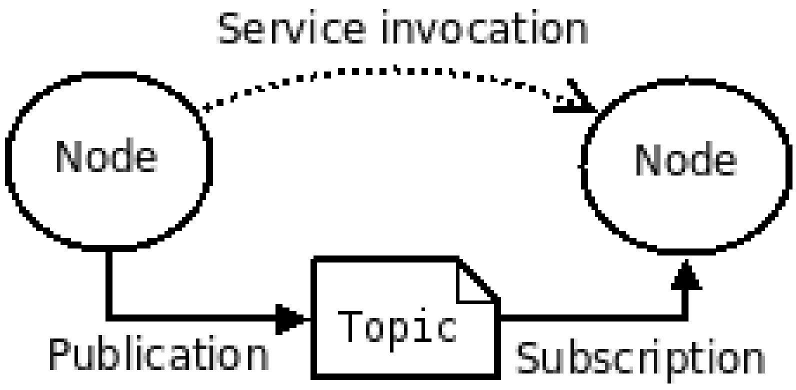


Figure 2.8: ROS Communication Model[3]

The last level is the Community level. ROS community is where ROS users can exchange software and knowledge, including different distributions, repositories are for different institutions to develop and release their own software, ROS Wiki is for documenting information, ROS answers site and ROS blogs.

In the ROS ecosystem, software are separated into three groups:

- tools that are language- and platform-independent, used for building and distributing ROS-based software;
- implementation of ROS client library, such as roscpp, rospy and roslisp;
- packages that can be used in different applications and use one or more ROS client libraries.

Language-independent tools and client libraries are released under the terms of the Berkeley Software Distribution (BSD) license, so they are open source software and can be used for free both commercially and in research. Most of the packages are also licensed under open source licenses. These packages implement the commonly used functionality like hardware drivers, robot models, data types, route planning, environment perception, simultaneous localization and mapping (SLAM), simulation tools and other algorithms. The main ROS client libraries (C++, Python and Lisp) are for a Unix-like system. Client libraries are supported in Ubuntu Linux.

ROS-Industrial is an open source project. It applies capabilities of ROS to manufacturing automation and robotics. It provides interfaces for common industrial manipulators, grippers, sensors, and device networks. It also provides software libraries for automatic 2D/3D sensor calibration, route planning, application called Scan-N-Plan, developer tools like the Qt Creator ROS Plugin, and training curriculum that is specially designed for the needs of manufacturers.

ROS releases many versions that are called distribution. Each distribution includes different set of ROS packages. Each distribution has its own name. Indigo Igloo is more friendly to beginners of ROS for its stability and abundant community support. Catkin, which is a build system, is only supported by Indigo and later. Kinetci Kame is the mostly used distribution with new capabilities. Melodic Morenia is the newest distribution.

2.6 Sensors

Today's semi-autonomous vehicles apply different designs of radar and camera systems, high-resolution and low-cost LIDAR systems are also under development. To realize level 4 and level 5 automation, various numbers and kinds of sensors have to be used. This chapter describes the main features and applications of the commonly used sensors in autonomous vehicles field.

2.6.1 Radar

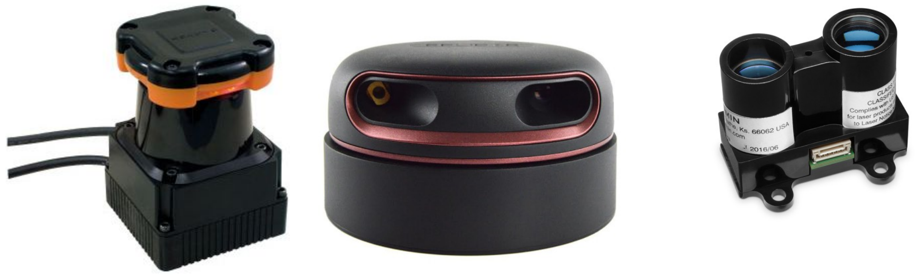
Radar is a detection system which uses radio waves to determine the properties of objects. It can measure range, angle or velocity of objects. A radar system consists of a transmitter to produce the radio waves, a transmitting and receiving antenna, a receiver and a processor. Radar sensors are classified by their operating distance ranges: short range radar (SRR) works from 0.2 to 30 meter range, medium range radar (MRR) works from 30 to 80 meter range and long range radar (LRR) works from 80 to more than 200 meter range. Currently, LRR is used in Adaptive Cruise Control (ACC) and highway Automatic Emergency Braking Systems (AEBS). One advantage of radar is it needs less computation than other sensors, and it works in almost all the environmental conditions like rain, dust or sunlight.

2.6.2 Ultrasonic Sensor

Similar to radar, ultrasonic sensors calculate the distance of an object by measuring the time between sending an ultrasound signal and receiving the echo. It is very sensitive to dirt so it is not suitable for heavy outdoor use. It is widely used in parking. It is also used to aid autonomous navigation.

2.6.3 Lidar

Lidar is an acronym of light detection and ranging. Lidar measures the distance to an object by calculating the time needed by a pulse of laser light to travel to an object and back to the sensor. Lidar is commonly used to make 3D representations of objects according to the different return times and wavelengths. In autonomous vehicles, lidar sensors are placed on the top of the vehicles to provide the 360°3D view of the environment, so the vehicles can be able to detect obstacles and avoid them. There are some aspects that lidar companies are working on. One is to reduce the size and cost. Another is to use longer wavelength to increase the detection range and accuracy. Recently, solid state lidar (SSL) with no moving parts is replacing the scanning lidar for its higher reliability.



(a) Hokuyo Lidar (b) RPLIDAR (c) Lidar Lite

Figure 2.9: Three Commonly Used Lidar Sensors

As is illustrated in Figure 2.9, three lidar sensors that are commonly used for research purpose are introduced:

- Hokuyo Scanning Rangefinder: Hokuyo has many kinds of lidar sensors, the frequently used ones are called scanning laser rangefinders. The detectable range is from 100mm to 30m, it takes 25 millisecond per scan. It works under 12V. It has 270°area scanning range with 0.25°resolution. It uses USB 2.0 interface. However, it costs around USD $4500 each.

- RPLIDAR: RPLIDAR is another commonly used lidar sensor brand. Its working range is from 0.15m to 12m. It can scan 360°with 0.45°to 1.35°resolution. The scan rate is from 5 to 15Hz. It costs USD $380 each.

- LIDAR-Lite: LIDAR-LiTe is a 1D laser rangefinder. It works up to 40 meter. It can be interfaced with I^2C or PWM. It can work together with a spinning module as a scanning lidar. It costs about USD $150 each.

2.6.4 Camera

Camera sensors are indispensable to autonomous systems. It is the only sensors that can capture texture, color and contrast of the objects. It can also capture the details of the environment. It is applied on all sides of an autonomous vehicle. Like human eyes, however, it is susceptible to severe weather and light conditions, so it needs to collaborate with other sensors. With the increasing pixel resolution and the decreasing price, camera sensors are becoming more and more important to autonomous vehicles.

2.6.5 GPS

GPS stands for Global Positioning System. It is a global navigation satellite system owned by the US government. It can provide geolocation and time information. GPS signals can be weak in mountains and buildings. GPS is a very important part in autonomous vehicle system, because it can provide location information and assist the navigation.

2.6.6 Inertial Measurement Unit

An Inertial Measurement Unit (IMU) is an electronic device that can measure an object's specific force, angular rate and the magnetic field around. It is typically used to maneuver vehicles. It is needed for some localization algorithms. It can predict the location and velocity of an object according to the object's previous pose. As a result, it can help an object to localize itself when GPS signals are unavailable, like inside buildings or in tunnels.

2.7 Simultaneous Localization and Mapping

Simultaneous localization and mapping (SLAM) is a very important part in robotic fields. It is a computational problem of building and updating a map of an unknown environment, and keeping track of the robot's location simultaneously at the same time[57][58][59]. There are several algorithms to solve this chicken-and-egg problem, such as particle filter (aka. Monte Carlo methods), extended Kalman filter and GraphSLAM. Since SLAM algorithms are limited to the available resources, it is not aimed at perfection, but at operational compliance. It is widely applied on autonomous vehicles, newer domestic robots and even inside the human body[60].

The SLAM problem is to compute an estimate of the robot's location and a map of the environment with the given sensor observation data over discrete time steps. Statistical techniques, including Kalman filters and particle filters, provide the estimation of the posterior probability function for the pose of the robot and for the parameters of the map. Set-membership techniques, which are based on interval constraint propagation, provide a set which encloses the pose of the robot and a set estimation of the map. Another technique is Maximum a posteriori estimation (MAP), which uses image data to jointly estimate poses and landmark positions to increase map fidelity. It is the technique used by Google's ARCore.

There are several important aspects that can affect SLAM algorithms:

- Map: Maps are classified as topological maps which represent the environment with topology, and grid maps that use arrays of discretized cells to represent a topological world.

- Sensors: Laser scans can provide details of many points within the area with tactile sensors only contain points very close to the agent. Most practical SLAM tasks fall between this two kinds of data. Optical sensors can be one dimension or 2D, even 3D.

- Kinematics model: The kinematics model of the robot improves the estimation of sensing under conditions of inherent and ambient noise.
- Loop closure: It is the problem of recognizing a location that visited previously and updating accordingly.Typical methods compute sensor measure similarity and reset the location priors when a match is found.

Many SLAM algorithms are implemented in ROS libraries, three mostly used are described and compared below:

- Gmapping: Gmapping is a laser-based SLAM. It is substantially a particle filter. It needs odometry and laser scan data to produce 2D occupancy grid map.
- Hector SLAM: Hector SLAM is an algorithm developed by researchers at Technische Universitat Darmstadt. It can be used without odometry and on platforms that can roll or pitch. It benefits from the high rate lidar systems and provides 2D pose estimation at scan rate of the sensors. It is accurate enough although it dose not provide explicit loop closing ability[61].
- Cartographer: Cartographer is an algorithm developed by Google in 2016. It achieves real time loop closure by using a branch-and-bound approach to compute scan-to-submap matches as constraints. It works well with portable platform and limited computational resources[62].

Chapter 3

Design and Implementation

In this chapter, the design and implementation of the low-cost AV prototype with mapping functionality is presented. The prototype is capable of localizing itself and building 2D map of unknown environment simultaneously, and it can avoid collision with remote control command via Zigbee network.

3.1 System Architecture

The system in this thesis is a low-cost AV prototype that is capable of localization and mapping using lidar sensor. As is illustrated in Figure 3.1, the architecture of the system consists of five components:

- RedBot: it is the mobile platform used to carry the lidar scanner to build the map and travel around to test the obstacle detection and avoidance function;
- Lidar scanner: it is is composed of an Arduino Mega, a servo motor and a lidar sensor;
- Zigbee network: it is a multi-point network that made up of three XBee modules to transfer data between RedBot and the laptop running ROS;

- Hector SLAM: a metapackage in ROS to realize the localization and mapping functions;
- Remote control: to control the RedBot by sending commands via serial port and Zigbee network.

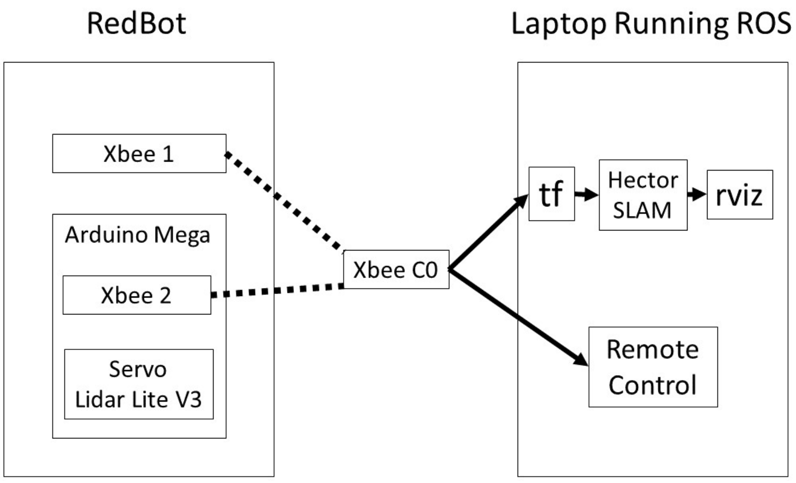


Figure 3.1: System Architecture

The detailed implementation is described in the following sections.

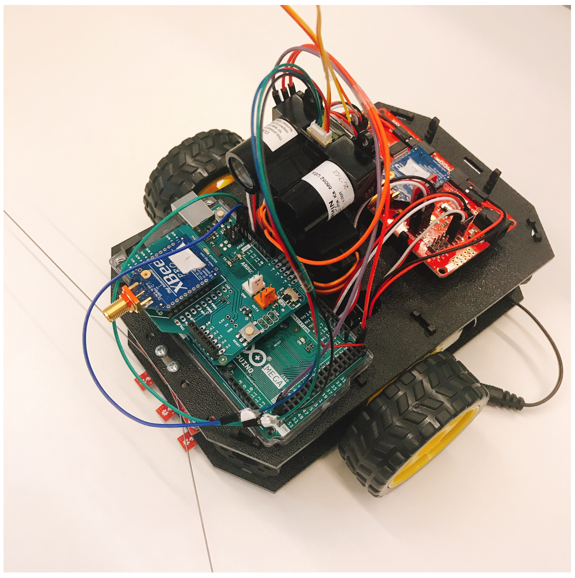


Figure 3.2: Hardware

3.2 Hardware

This section describes the hardware employed in this thesis (see Figure 3.2) and their configurations, including the SparkFun Inventor's Kit for RedBot, a microcontroller, a lidar sensor, a servo and three XBee modules.

3.2.1 RedBot

As is shown in Figure 3.3, the SparkFun inventor's kit for RedBot is composed of many pieces. Each component is described below.

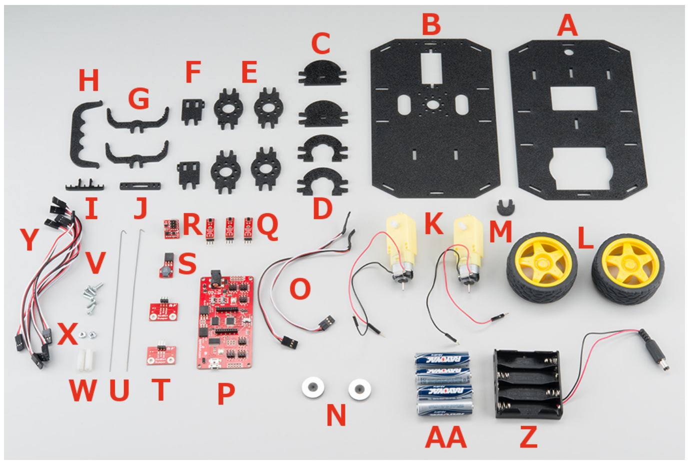


Figure 3.3: The SparkFun Inventor's Kit for RedBot

Mainboard

Mainboard is marked with P in Figure 3.3. It is a combination of a motor driver and Arduino with many headers and connections. It is designed to be very versatile. The headers and pins of the mainboard are depicted in Figure 3.4.

Motors and Wheels

The motors help the RedBot to travel around by turning the rubbery wheels. Motors are marked with K and wheels are marked with L in Firgure 3.3.

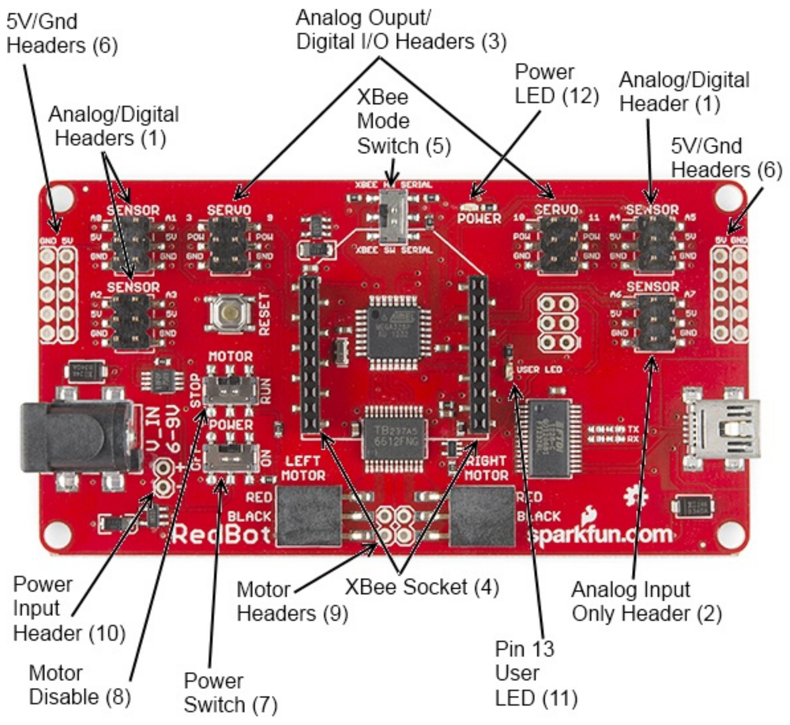


Figure 3.4: RedBot Mainboard

To drive the motors, there are two ways. One is to call function *drive()* (see Listing 3.1), and another is to call functions *rightMotor()* and *leftMoter()* separately (see Listing 3.2). Besides, function *stop()* and *brake()* are slightly different ways to stop the RedBot, *stop()* stops the RedBot immediately, while *brake()* stops the RedBot gradually.

Listing 3.1: Code for Driving two Motors Together

```
#include <RedBot.h>
RedBotMotors motors;
void setup()
{
    // Turn on left and right motors at full speed forward.
    motors.drive(255);
    delay(2000);
    motors.stop();
}
void loop(){}
```

Listing 3.2: Code for Driving two Motors Separately

```
#include <RedBot.h>
RedBotMotors motors;
void setup()
{
    // Turn on right motor clockwise, motorPower=150
    motors.rightMotor(150);
    // Turn on left motor counter clockwise, motorPower=150
    motors.leftMotor(-150);
    delay(2000);
    motors.brake();
}
void loop(){}
```

Wheel Encoders

Wheel encoders are simple add-on sensors, they are marked with N and O in Figure 3.3. It can detect the rotation number of each wheel. The wheel encoders in this kit use multi-pole diametric ring magnets attached to the motor shaft. The ring magnets spin with the motors and can be detected by the hall effect sensors. The encoder counts the number of magnetic poles' change. There are four counts for every one turn of the magnet, and one turn of the wheel is equal to 48 turns of the motor, so the counts per revolution are 196.

The wheel's diameter is denoted as D, and the circumference of the wheel is equal to πD. Hence the traveling distance of the RedBot can be calculated by

$$distance = \frac{counts}{countsPerRev}\pi D \tag{3.1}$$

Code for getting the counts of the wheel encoders are shown in Listing 3.3.

Listing 3.3: Code for Getting Wheel Encoder's Counts

```
#include <RedBot.h>
RedBotEncoder encoder = RedBotEncoder(A2, 10);
//variable to store the counts of left encoder
long lCount = 0;
//variable to store the counts of right encoder
long rCount = 0;
void loop()
{
    lCount = encoder.getTicks(LEFT);
    rCount = encoder.getTicks(RIGHT);
}
```

The function called *travelDistance()* returns the traveling distance of the RedBot. The function called *driveStraight()* uses the wheel encoders to drive a certain distance in a straight line, and the function called *turn()* employs the encoders to help the RedBot turn a 90 degree angle. The full code written for employing the wheel encoders can be found in Appendix A.1.

Line Follower

Line follower is marked with Q in Figure 3.3. It is infrared reflectance (IR) sensors to detect the surface below the RedBot. The way to get readings from the IR sensors is shown in Listing 3.4, and the full code of line follower is in Appendix A.2.

Listing 3.4: Code for Getting Readings from IR Sensor

```
#include <RedBot.h>
RedBotSensor IRSensor1 = RedBotSensor(A3);
void setup()
{
    Serial.begin(9600);
}
void loop()
{
    Serial.print("IR Sensor Readings: ");
    Serial.println(IRSensor1.read());
    delay(100);
}
```

Mechanical Bumpers

Mechanical bumpers are switches that close a circuit when the whisker pushes against an obstacle, as is marked with T and U in Figure 3.3. The code to avoid collision with the help of bumpers are shown in Listing 3.5, and the full code is in Appendix A.3.

Listing 3.5: Code for Mechanical Bumpers

```
#include <RedBot.h>
RedBotMotors motors;
RedBotBumper lBumper = RedBotBumper(3);
RedBotBumper rBumper = RedBotBumper(11);
int lBumperState;
int rBumperState;
```

```
void setup(){}
void loop()
{
    motors.drive(255);
    // default INPUT state is HIGH,
    // it is LOW when bumped
    lBumperState = lBumper.read();
    rBumperState = rBumper.read();
    // left side is bumped
    if (lBumperState == LOW)
    {
        reverse();     // backs up
        turnRight();   // turns
    }
    // right side is bumped
    if (rBumperState == LOW)
    {
        reverse();    // backs up
        turnLeft();   // turns
    }
}
```

Accelerometer

Redbot uses MMA8452Q 3-axis accelerometer with 12 bits of resolution. It is marked with R in Figure 3.3. It is a small add-on sensor. It can detect speed, tilt and bumps. It uses I^2C bus interface. The code to access the X, Y, and Z-axis acceleration and the angles in X-Z, Y-Z, and X-Y planes is shown in Listing 3.6. The RedBot can perceive its motion accordingly.

Listing 3.6: Code for Accelerometer

```
#include <RedBot.h>
RedBotMotors motors;
RedBotAccel accelerometer;
int xAccel, yAccel, zAccel;
float XZ, YZ, XY;
void setup(void){}
void loop(void)
{   // updates the x, y, and z-axis readings
    accelerometer.read();
    // get the X, Y, and Z-axis acceleration
    xAccel = accelerometer.x;
    yAccel = accelerometer.y;
    zAccel = accelerometer.z;
    // the relative angle between the X-Z, Y-Z, and X-Y
    XZ = accelerometer.angleXZ;
    YZ = accelerometer.angleYZ;
    XY = accelerometer.angleXY;
}
```

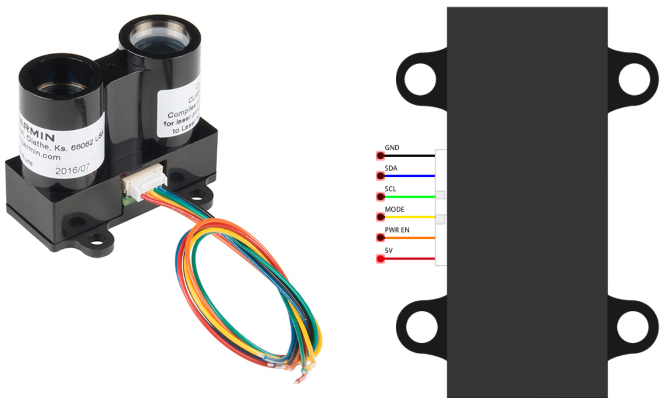
Buzzer

Buzzer is used to make some sounds, and it is marked with S in Figure 3.3. The code to set the buzzer's tone and play time is shown in Listing 3.7.

Listing 3.7: Code for Buzzer

```
#include <RedBot.h>
const int buzzerPin = 9;
void setup()
{
    // configures the buzzerPin as an OUTPUT
    pinMode(buzzerPin, OUTPUT);
}
void loop()
{
    // Play a 1kHz tone on the pin number held in
    // the variable "buzzerPin".
    tone(buzzerPin, 1000);
    delay(125);   // Wait for 125ms.
    noTone(buzzerPin);   // Stop playing the tone.
}
```

3.2.2 LIDAR Sensor



(a) Lidar Lite V3 (b) Interface Pins

Figure 3.5: Lidar Lite V3 and Interfaces

Lidar sensor used in this thesis is the Lidar Lite V3 (see Figure 3.5a). It is mounted on a servo, which is described in subsection 3.2.3. By spinning the lidar sensor, together with the ROS and SLAM algorithm, a 2D distance map can be created. As is shown in Figure 3.5b, it has six wires. A microcontroller can communicate with it over I^2C. It can also use a pulse-width modulated (PWM) signal to denote measured distances. The code to access measured distances is shown in Listing 3.8.

Listing 3.8: Code for Lidar Lite

```
#include <Wire.h>
#include <LIDARLite.h>
// Globals
LIDARLite lidarLite;
int cal_cnt = 0;
void setup()
{
    // Set configuration to default and I2C to 400 kHz
    lidarLite.begin(0, true);
    // basic configuration
    lidarLite.configure(0);
}
void loop()
{
    int dist;
    // At the beginning of every 100 readings,
    // take a measurement with receiver bias correction
    if ( cal_cnt == 0 ) {
        // With bias correction
        dist = lidarLite.distance();
    } else {
        // Without bias correction
        dist = lidarLite.distance(false);
    }
    // Increment reading counter
    cal_cnt++;
    cal_cnt = cal_cnt % 100;
    delay(10);
}
```

3.2.3 Servomotor

In this thesis, servo MG995 made by TowerPro is used to spin the lidar sensor. According to its specification, its operating speed is 0.2sec/60degree (4.8V), 0.16sec/60degree(6.0v). But according to the actual measurement, it works at the speed of 0.01sec/degree when driven by four AA batteries, and it can rotate between 2°and 160°. The code to control the servo to rotate between 2°and 160°continuously, in step of 1°is shown in Listing 3.9.

Listing 3.9: Code for Servo

```
#include <Servo.h>
#include <Wire.h>
#include <LIDARLite.h>
Servo myservo;
LIDARLite myLidarLite;
// variable to store the servo position
int pos = 2;
float dist;
```

```
void setup() {
    Serial.begin(115200);
    // attaches the servo on pin 9
    myservo.attach(9);
    myLidarLite.begin(0, true);
}
void loop() {
    for (pos = 2; pos <= 160; pos += 1) {
        // tell servo to go to position in variable 'pos'
        myservo.write(pos);
        delay(10);
        Serial.print(pos);
        Serial.print("\t");
        Serial.println(myLidarLite.distance());
        delay(10); }
    for (pos = 160; pos >= 2; pos -= 1) {
        myservo.write(pos);
        delay(10);
        Serial.print(pos);
        Serial.print("\t");
        Serial.println(myLidarLite.distance());
        delay(10); }
}
```

3.2.4 XBee

The RedBot employs XBee Pro S1 RF modules (produced by Digi) to communicate wirelessly with its XBee header. In addition, one XBee RF module (see Figure 3.6) is used along with another microcontroller Arduino Mega, and another one is connected to the laptop.

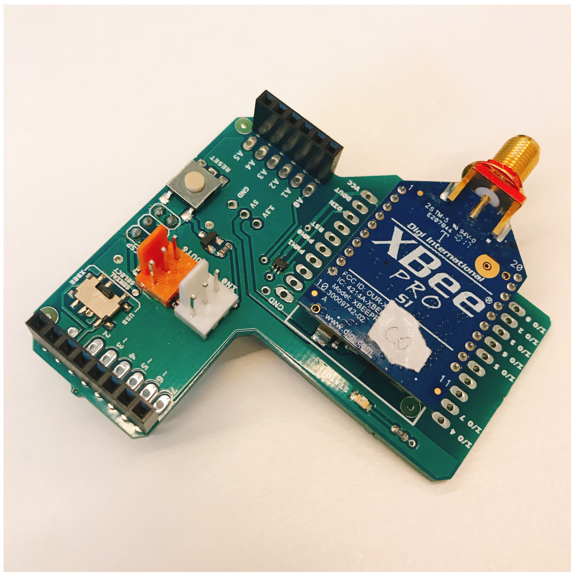


Figure 3.6: Arduino Xbee Shield and RF module

The XBee modules are able to talk to each other with their default settings, using Arduino serial commands. In this thesis, two Arduino boards with XBee modules communicate with a master XBee connected to the laptop running ROS, so each XBee needs to be configured. The detailed configurations are described in section 3.3.

3.2.5 Microcontroller

Due to the limited GPIO capability of the microcontroller board on the RedBot, another microcontroller, Arduino Mega 2569 is employed to control the servo, interface the lidar and send lidar scan information back to the laptop running ROS via XBee. Besides Arduino Mega's GPIO capability, it has lager storage space than Arduino Uno (256kb flash memory), which is needed for large array of float type in the code. In addition, Arduino boards can work as ROS nodes directly using the *rosserial_arduino* package.

3.3 Zigbee Network

There are three XBee modules, which are denoted as XBee C0, XBee 1 and XBee 2 separately, to form the Zigbee network. XBee C0 represents the master XBee module that is connected to the laptop, XBee 1 is the module used on RedBot to send back the signal of obstacle detected at the vicinity to the ROS and receive the commands from serial port, XBee 2 is the module used on Arduino Mega to send back the lidar scanning data to the ROS.

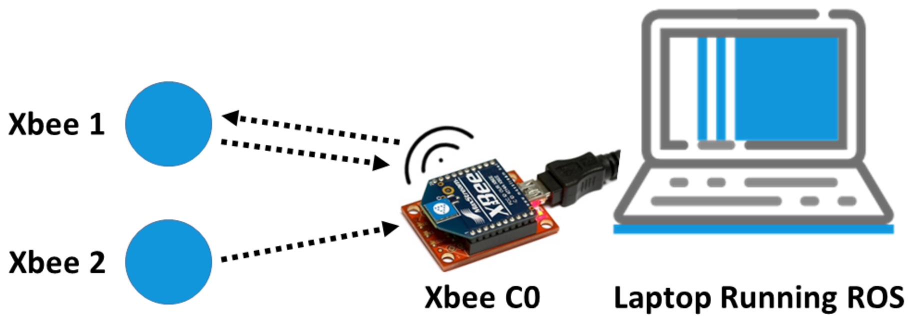


Figure 3.7: Zigbee Network

In this thesis, XBee modules are configured with XCTU. Three XBee module need to be in the same network and channel so that they can communicate. A module's destination address is detonated as DH and DL. When a module's DH is zero and DL is less than 0xFFFF, data sent by this module can only received by the module whose MY parameter is equal to DL. Both XBee 1 and 2 send data to XBee C0, and XBee C0 only needs to send data to XBee 1, so to configure them to work under transparent mode, network parameter ID to be 1331, channel parameter CH to be D, baud rate to be 57600 which is the highest baud rate for XBee to work with ROS, addressing parameter DH to be 0, DL of XBee C0 to be 1 and DL of the other two to be 0. Self address parameter MY to be 0, 1 and 2 for

XBee C0, 1 and 2, respectively. Since every Zigbee network needs a coordinator to form the network in the first place, I set XBee C0 as the coordinator and XBee 1 and 2 as the end devices. The illustration of the Zigbee network is in Figure 3.7.

3.4 ROS Software

As is shown in Figure 3.8, ROS software consists of several nodes, topics (communication channels) and services. There are four main nodes group, lidar scanner node, XBee node, tf node for coordinate frames transformation and hector slam node group. The implementation of each node is described below.

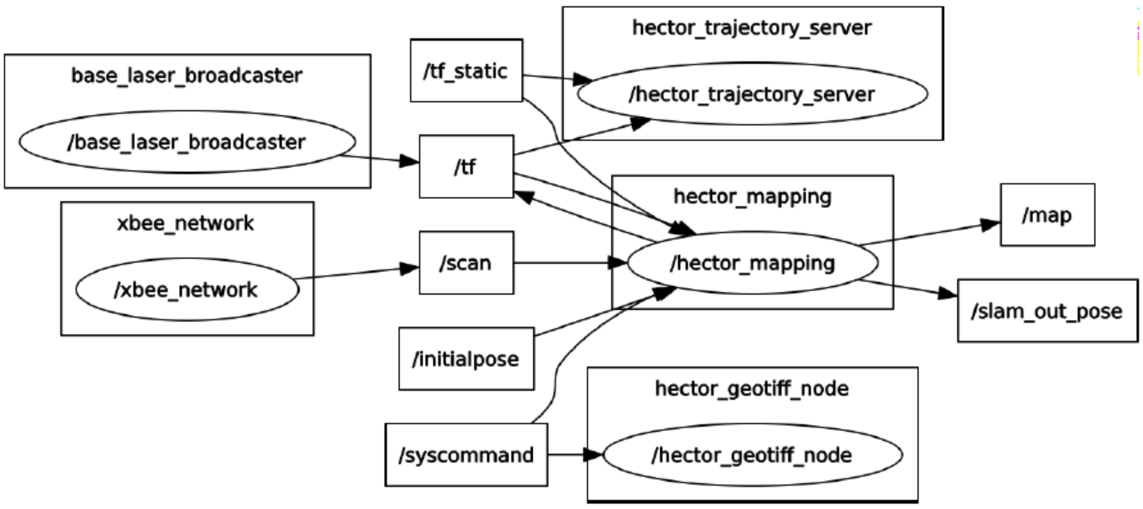


Figure 3.8: ROS Nodes' Graph

3.4.1 Lidar Scanner Node

Acquiring source of $sensor\_msgs/LaserScan$ type data is the first step to build a 2D map with hector SLAM. A lidar scanner node is built with Lidar Lite V3, servo and Arduino Mega 2560. To use the $rosserial\_arduino$ package, the header files

```
#include <ros.h>
#include <sensor_msgs/LaserScan.h>
```

must be included. Then ROS node handle, laser scan message and Publisher *scan_pub* need to be defined:

```
ros::NodeHandle  nh;
sensor_msgs::LaserScan  scan;
ros::Publisher  scan_pub("scan", &scan);
```

In *setup*(), besides the configurations of lidar sensor and servo, parameters of the lidar scanner need to be defined. In *loop*(), the lidar scanner scans from 16°to 160°in a step of 16°, and spins back. The degree increment of 16°is chosen by the buffer size of Zigbee module which can only store limited scan results. The code

```
scan.ranges = ranges;
scan_pub.publish(&scan);
```

publishes the laser scan data to the topic $\backslash scan$. The complete code for lidar scanner is in Appendix B.1.

3.4.2 Zigbee Network

rosserial_xbee package is provided by ROS to allow multi-point communication between rosserial nodes with XBee modules. After configuring XBee modules as section 3.3 describes, ROS node *xbee_network.py* needs to be launched. First launching *roscore* with the command:

```
$ roscore
```

then opening a new terminal, and entering the following commands:

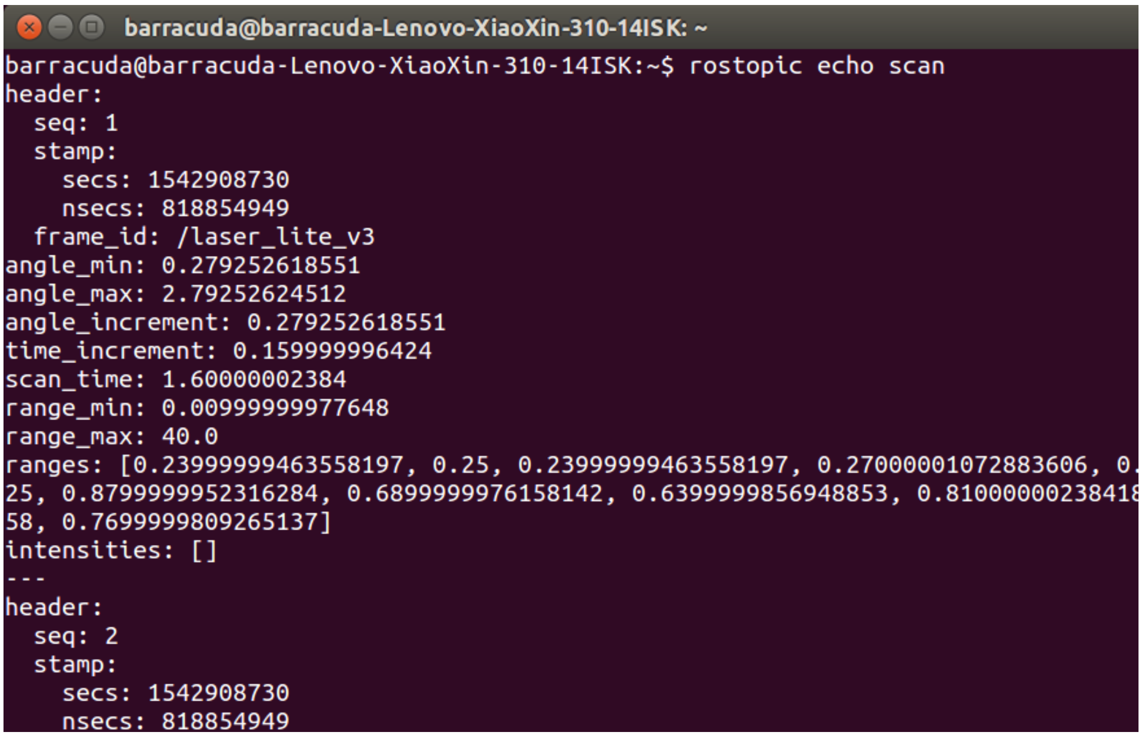
```
$ rosrun  rosserial_xbee  xbee_network.py  /dev/ttyUSB0  1  2
```

$/dev/ttyUSB0$ is the serial port that is connected to XBee C0. The launched node *xbee_network.py* builds the Zigbee network, and forward the data received to the rest of the ROS.

3.4.3 Hector SLAM

After entering the following command in a new terminal, we can subscribe to the rostopic $\backslash scan$ and see the outcome of the scanning (see Figure 3.9).

```
$ rostopic echo scan
```



```
barracuda@barracuda-Lenovo-XiaoXin-310-14ISK: ~
barracuda@barracuda-Lenovo-XiaoXin-310-14ISK:~$ rostopic echo scan
header:
  seq: 1
  stamp:
    secs: 1542908730
    nsecs: 818854949
  frame_id: /laser_lite_v3
angle_min: 0.279252618551
angle_max: 2.79252624512
angle_increment: 0.279252618551
time_increment: 0.159999996424
scan_time: 1.60000002384
range_min: 0.00999999977648
range_max: 40.0
ranges: [0.23999999463558197, 0.25, 0.23999999463558197, 0.27000001072883606, 0.
25, 0.8799999952316284, 0.6899999976158142, 0.6399999856948853, 0.81000000238418
58, 0.7699999809265137]
intensities: []
---
header:
  seq: 2
  stamp:
    secs: 1542908730
    nsecs: 818854949
```

Figure 3.9: Sample Output of Lidar Scanner

hector_slam metapackage includes three main packages:

- *hector_mapping*: the SLAM node;
- *hector_geotiff*: to save map and robot trajectory to geotiff images files;
- *hector_trajectory_server*: to save tf based trajectories.

hector_slam employs *hector_mapping* node for SLAM approach, which can work only with *sensor_msgs/LaserScan* data, i.e. without odometry. This node also can estimate the platform's 2D pose at laser scanner frame rate. The transformation between frames need to set correctly. The tf tree is: $map \rightarrow odom \rightarrow base\_link \rightarrow laser\_base$. According to the documentation of $hector\_slam$, the transformation between map and $odom$ frame is provided, and since odom frame is not used for RedBot, the transformation between $odom$ and $base\_link$ can be set in the launch file with:

```
<launch>
<param name="pub_map_odom_transform" value="true"/>
<param name="map_frame" value="map" />
<param name="base_frame" value="base_frame" />
<param name="odom_frame" value="base_frame" />
</launch>
```

The transformation between $base\_link$ and $laser\_base$ can be set with the static transform publisher:

```
<launch>
<node pkg="tf" type="static_transform_publisher"
name="base_laser_broadcaster"
args="0.1 0 0.2 0 0 1 base_link \laser_lite_v3 100" />
</launch>
```

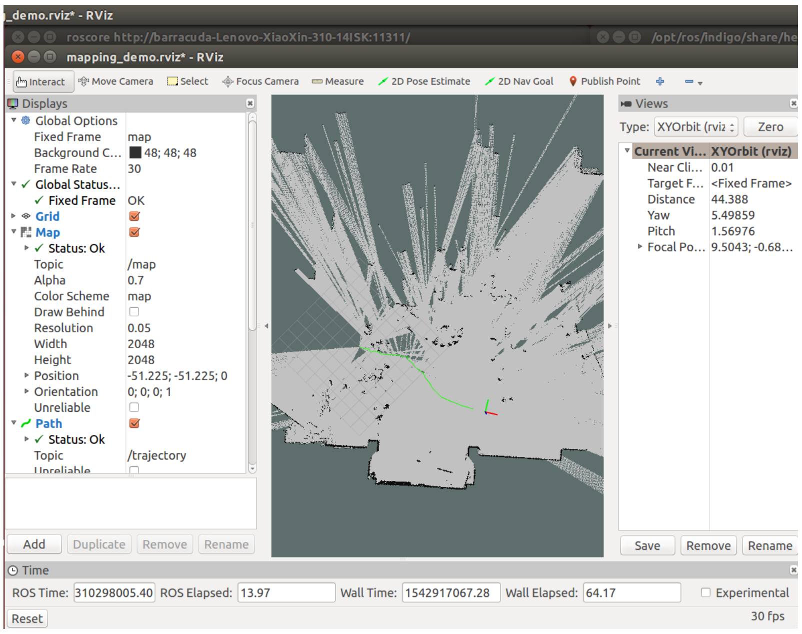
To start the $hector\_slam$ system, launch file $mySLAM.launch$ need to be run:

```
roslaunch hector_slam_launch mySLAM.launch
```

This launch file starts the $hector\_mapping$, $hector\_trajectory\_server$ and $hector\_geotiff$ nodes, as well as the rviz visualization tool. Three launch files used are attached in Appendix B.2, B.3 and B.4.

3.4.4 Rviz Visualization

After starting the rviz visualization tool, the trajectory and the 2D map are updated and shown on the screen simultaneously (see Figure 3.10).



Global Options	
Fixed Frame	map
Background C…	■ 48; 48; 48
Frame Rate	30
✓ Global Status…	
✓ Fixed Frame	OK
Grid	✓
Map	✓
✓ Status: Ok	
Topic	/map
Alpha	0.7
Color Scheme	map
Draw Behind	☐
Resolution	0.05
Width	2048
Height	2048
Position	-51.225; -51.225; 0
Orientation	0; 0; 0; 1
Unreliable	☐
Path	✓
✓ Status: Ok	
Topic	/trajectory
Unreliable	☐

Type: XYOrbit (rviz

Current Vi…	XYOrbit (rviz)
Near Cli…	0.01
Target F…	<Fixed Frame>
Distance	44.388
Yaw	5.49859
Pitch	1.56976
Focal Po…	9.5043; -0.68…

ROS Time: 310298005.40 ROS Elapsed: 13.97 Wall Time: 1542917067.28 Wall Elapsed: 64.17

30 fps

Figure 3.10: Rviz Visualization Tool

3.5 Obstacle Detection and Avoidance with Remote Control

As is mentioned in wiki of navigation stack of ROS[63], the mobile platform should be controlled by sending velocity commands in the form of: x, y and theta velocity, but RedBot is controlled with motor power only. Due to this hardware limitation, navigation function provided by ROS is not used, instead, the RedBot is controlled by sending remote control commands via Zigbee network.

As is shown in the flowchart (Figure 3.11), the RedBot starts to drive forward straightly at the default speed, and can be stopped at anytime. The obstacle detection is realized by the bumpers described in subsection 3.2.1. When the bumpers collide with some obstacles, a signal is triggered to back up and stop the RedBot before the RedBot collides with the obstacle, then the signal of obstacle detected at the vicinity are sent to the laptop. The user decides to turn right or left with the currently 2D map generated by SLAM and sends the commands to the RedBot. After receiving commands from the laptop, RedBot turns a certain degree to the left or right, and keeps driving forward until a new obstacle is detected or a “stop” signal is received. The code of obstacle detection and avoidance with remote control is attached in Appendix B.5.

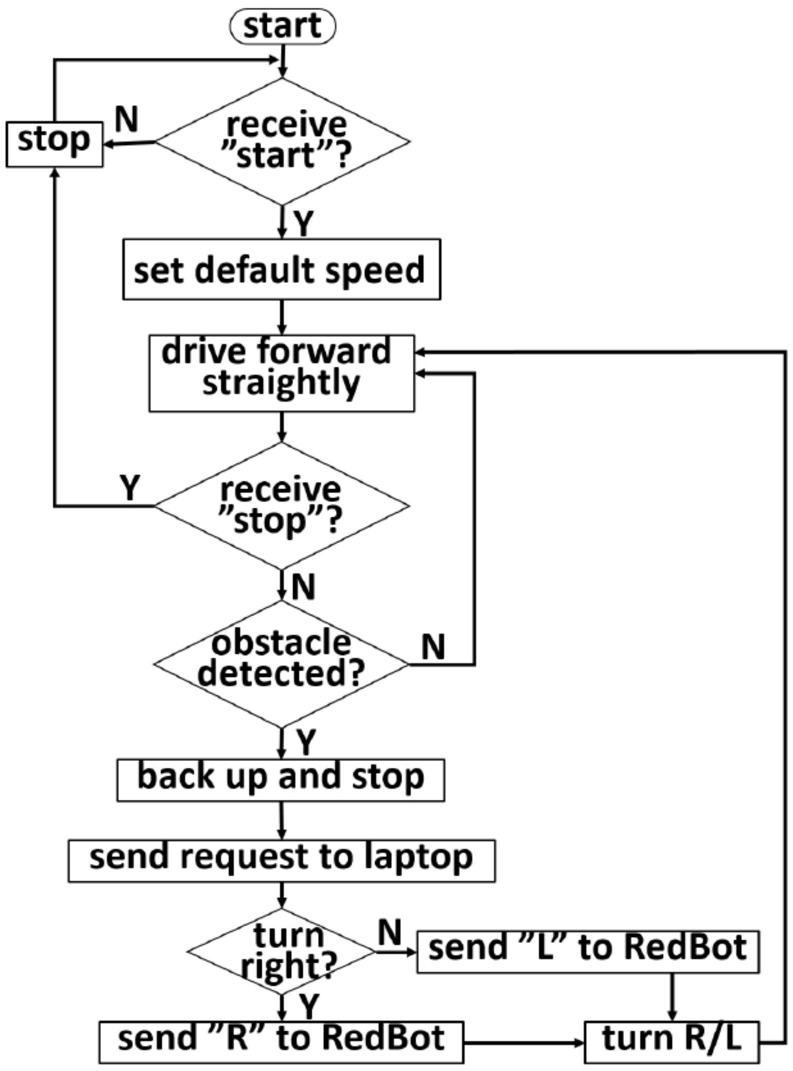


Figure 3.11: Flow Chart of Obstacle Detection and Remote Control

Chapter 4

Evaluation

In this chapter, the test scenarios that are set up for evaluating the functionalities of the prototype, including obstacle detection and avoidance, localization and 2D mapping, as well as the test results are described. The limitations of the prototype are discussed as well.

4.1 Setup and Requirements of Prototype

The prototype built in this thesis is able to localize and build a 2D map using data from lidar sensor. Besides, it can detect obstacles with a mechanical device and avoid collision with remote control commands wirelessly. The prototype consists of a Redbot as a mobile platform, three Zigbee RF modules, a servo motor, a Lidar Lite V3 sensor, and an Arduino mega board. It employs $Hector\_SLAM$ metapackage in ROS to realize the SLAM algorithm. Table 4.1 shows the requirements of the prototype and their test results, all three requirements are achieved.

Table 4.1: Requirements and Test Results

Requirement	**Test Result**
RedBot can send lidar data to laptop with ROS wirelessly.	PASSED
RedBot can be controlled with remote control commands via XBee.	PASSED
ROS can output a usable 2D map of the test site with the lidar data.	PASSED

4.2 Obstacle Detection and Avoidance

As is described in Section 3.5, the prototype can utilize bumpers to detect the obstacles in the vicinity, and travel according to the remote control commands (see Table 4.2). Tests of each command are conducted.

Table 4.2: Remote Control Commands and Corresponding Operations

Remote Control Commands	**Corresponding Operation**
0	stop the RedBot
1	start the RedBot
2	turn right a certain degree and drive straightly
3	turn left a certain degree and drive straightly
7	obstacle detected

First the prototype was put on the ground, and powered on. After pushing the $reset$, the description of the commands were shown in the serial port monitor, and 1 was entered and sent to the Redbot to start the prototype (see Figure 4.2a). Next, Redbot started to drive forward straightly for a while and it backed up and stopped when an obstacle was detected by the bumpers. 7 was received in the serial monitor after that, and 2 was entered and sent to control the Redbot to turn right and keep driving forward (see Figure 4.2b). Redbot did not stop until another obstacle was detected, and another 7 was received. 3

was sent and let the Redbot turn left and keep driving forward (see Figure 4.2c). Finally, 0 was sent to stop the Redbot (see Figure 4.2d).

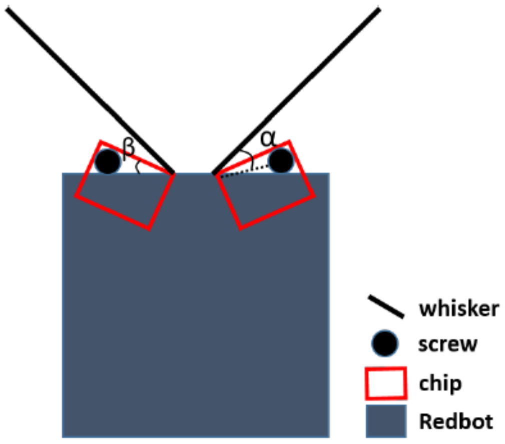


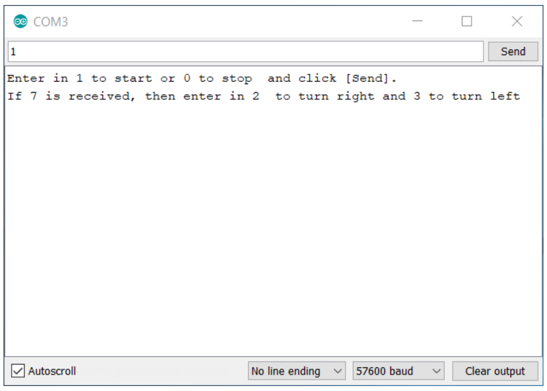
Figure 4.1: Diagram of Whisker Bumper

The sensitivity of the bumpers and the proximity of the obstacle can be adjusted by changing the angle between the whisker and the screw of the bumper (α in Figure 4.1), and between the chip and the Redbot (β in Figure 4.1). After measurement, α is set to 3°and β is set to 30°mechanically to leave enough response time for collision avoidance when the Redbot travels under 1 km/h. Obstacles can be detected when 7.5 cm away from the Redbot under this setting. The value of α and β should be different when testing on surfaces with different friction coefficient.

The tests above verify the functionality of obstacle detection and avoidance with remote control commands.

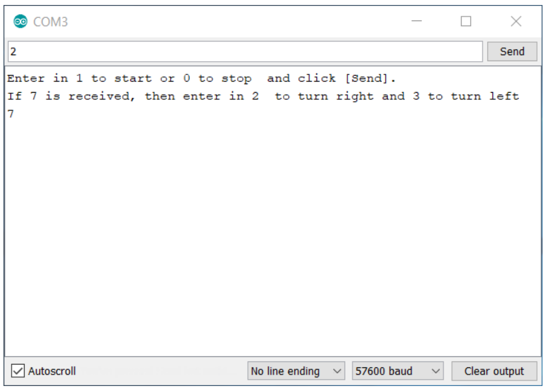
4.3 2D Mapping

The 2D map built by Hector SLAM in ROS is shown in Figure 4.3a. It recovers the test site. The black line in Figure 4.3a is the boundary of the test site which was made up of several boxes in different sizes. Most of the edges of the boxes can be seen in the 2D map. The left boundary is missing because of the limited scan range of the prototype (because



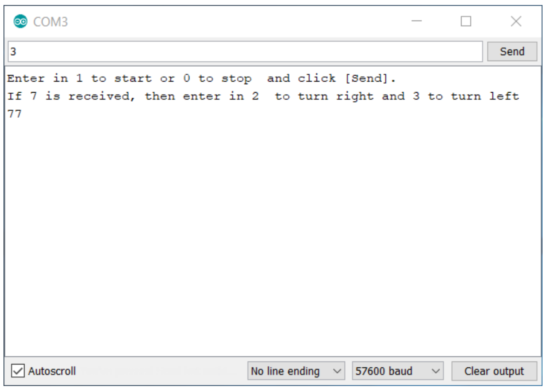
```
1
Enter in 1 to start or 0 to stop  and click [Send].
If 7 is received, then enter in 2  to turn right and 3 to turn left
```

(a) Start the RedBot



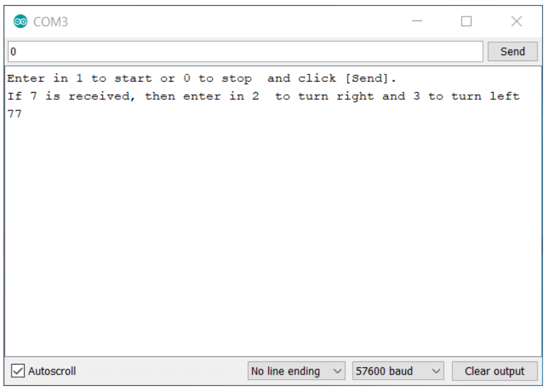
```
2
Enter in 1 to start or 0 to stop  and click [Send].
If 7 is received, then enter in 2  to turn right and 3 to turn left
7
```

(b) Turn Right When Obstacle Detected



```
3
Enter in 1 to start or 0 to stop  and click [Send].
If 7 is received, then enter in 2  to turn right and 3 to turn left
77
```

(c) Turn Left When Obstacle Detected



```
0
Enter in 1 to start or 0 to stop  and click [Send].
If 7 is received, then enter in 2  to turn right and 3 to turn left
77
```

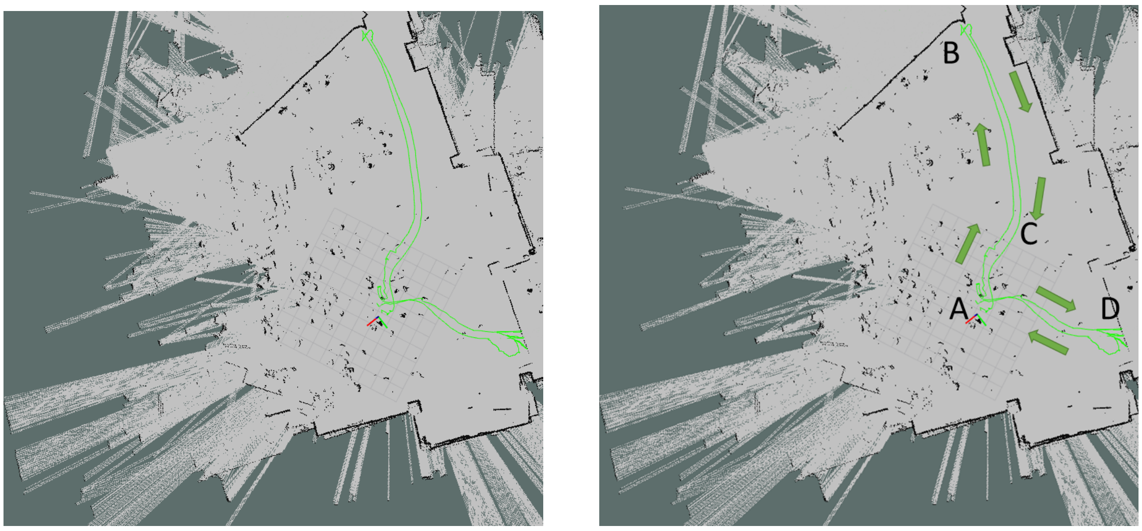
(d) Stop the RedBot

Figure 4.2: Tests on Remote Control Commands

the servo motor can only rotate from 2°to 160°).

The green line in Figure 4.3a is the trajectory of the prototype which proves the localization functionality of the system. Figure 4.3b indicates the moving direction of the prototype. It started from point A to point C, and it was stopped in point C and turned left. After it reached point B and detected the obstacle, it backed up and turned right a few times until it adjusted to the correct direction to point C, it kept driving forward and went back to C, turned right and was back to A. After that, it turned left and drove forward to D, and repeated the same process as it detected obstacle in point B. Finally, it went back to A and stopped.

Through test and the output map evaluates the localization and 2D mapping function-



(a) 2D Map of Test Site with Trajectory

(b) Trajectory Indication

Figure 4.3: 2D Map of Test Site

alities of the system, as well as the capability of obstacle detection and avoidance with remote control commands, and all the requirements mentioned in Table 4.1 are reached.

4.4 Limitations

In the tests mentioned in the previous two sections, some limitations of the system are found. First is the limited scan range causes many blind spots for the prototype. The missing boundaries can only be mapped by more adjustment of the travel direction of the prototype. Second is the unstable connection of I^2C interface of Lidar Lite V3 causes interruption of the system sometimes, and the system need to be reset manually. Third is the low driving speed of the Redbot. As a result, it takes more time than typical SLAM system to build a map of the same-size site. The driving speed is set to be low because of two reasons. One is the limited storage space of XBee RF module, so only 20 points can be sent in each scan. It decreases the mapping accuracy and can affect the correctness of the localization. Making the Redbot to move slower can compensate for the accuracy loss caused by the limited lidar points. Another reason is that the obstacle detection is realized by mechanical bumpers, if the prototype travels too fast, there will be no enough time for it to stop before colliding with the obstacles. The third limitation is the prototype cannot use autonomous navigation of ROS. It causes the complex adjustment process with remote command, and the decisions of the travel direction made by users can be very arbitrary and causes failure of localization and mapping. The last one is that the system can only build 2D map, so the height of obstacles cannot be shown on the map.

Chapter 5

Conclusions

In this thesis, a low-cost prototype of an AV that is capable of building 2D map of its surrounding with lidar sensor is presented. The prototype consists of Redbot, Lidar Lite V3, a servo motor, an Arduino Mega board, three XBee RF modules and $Hector\_SLAM$ package in ROS. The Redbot is the mobile platform to carry the lidar scanner. Lidar scan points are sent back to a laptop running ROS, and $hector\_mapping$ node in ROS converts the lidar data to a 2D map and realizes the localization functionality. $hector\_trajectory\_server$ node saves the trajectory of the prototype. The prototype cannot use the autonomous navigation provided by ROS package due to the hardware limitation. Instead, mechanical bumpers and remote control commands are used to control the prototype. The prototype backs up and stops when the bumpers detect obstacles. A request is sent back to the laptop and a remote user sends back the appropriate commands via Zigbee network, then the prototype acts accordingly.

In conclusion, obstacle detection and avoidance is realized by using mechanical bumpers and remote control commands. Localization and 2D mapping functionality is verified in the tests conducted, and boundary missing problem is caused by limited scan range of lidar. But by adjusting the path of the prototype, usable 2D maps of the unknown environment can be built, as a result, the requirement of building a prototype that is capable of localization and 2D mapping simultaneously, and avoiding collision is achieved.

5.1 Future Work

In the future, other low cost mobile platforms should be explored to work with ROS, including the other SLAM and autonomous navigation packages. Lidar sensors like RPL-IDAR should be employed to realize 360°high frequency scanning. BLE protocol can be utilized to build the communication between the laptops running ROS and the mobile platforms for its high data rate. 3D map can be built based on 2D map and lidar point cloud. Further research on combining sensor fusion and computer vision should be done for improving the AV's perception of the surroundings.

References

[1] Silver, David. How Self-Driving Cars Work. `https://medium.com/udacity/how-self-driving-cars-work-f77c49dca47e`, Dec 2017. [Accessed: 2018-09-17].

[2] Ovidiu Vermesan and Peter Friess. *Internet of things-from research and innovation to market deployment*, volume 29. River publishers Aalborg, 2014.

[3] AaronMR. Wiki: ROS/Concepts. `http://wiki.ros.org/ROS/Concepts`, June 2014. [Accessed: 2018-10-25].

[4] Daniel J Fagnant and Kara Kockelman. Preparing a nation for autonomous vehicles: opportunities, barriers and policy recommendations. *Transportation Research Part A: Policy and Practice*, 77:167–181, 2015.

[5] Autotech. 46 Corporations Working On Autonomous Vehicles. `https://www.cbinsights.com/research/autonomous-driverless-vehicles-corporations-list/`, Sep 2018. [Accessed: 2018-09-14].

[6] Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue, et al. An empirical evaluation of deep learning on highway driving. *arXiv preprint arXiv:1504.01716*, 2015.

[7] Peter Corke, Jorge Lobo, and Jorge Dias. An introduction to inertial and visual sensing, 2007.

[8] H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: part i. *IEEE Robotics Automation Magazine*, 13(2):99–110, June 2006.

[9] Milwaukee Sentinel. Phantom auto'will tour city. *The Milwaukee Sentinel*, page 4, 1926.

[10] Navlab: The Carnegie Mellon University Navigation Laboratory. `http://www.cs.cmu.edu/afs/cs/project/alv/www/index.html`, Dec 2014. [Accessed: 2018-09-28].

[11] Takeo Kanade, Chuck Thorpe, and William Whittaker. Autonomous land vehicle project at cmu. In *Proceedings of the 1986 ACM fourteenth annual conference on Computer science*, pages 71–80. ACM, 1986.

[12] Richard S Wallace, Anthony Stentz, Charles E Thorpe, Hans P Moravec, William Whittaker, and Takeo Kanade. First results in robot road-following. In *IJCAI*, pages 1089–1095. Citeseer, 1985.

[13] Jürgen Schmidhuber. Prof. schmidhuber's highlights of robot car history. *Istituto Dalle Molle di Studi sull'Intelligenza Artificiale*, 2011.

[14] Evan Ackerman. Video friday: Bosch and cars, rovs and whales, and kuka arms and chainsaws. *IEEE Spectrum*, Jan 2013.

[15] Rebecca Rosen. Google's self-driving cars: 300,000 miles logged, not a single accident under computer control. *The Atlantic*, 2012.

[16] BBC News. Nissan car drives and parks itself at Ceatec. `https://www.bbc.com/news/technology-19829906`, Oct 2012. [Accessed: 2018-09-29].

[17] BBC News. Toyota sneak previews self-drive car ahead of tech show. `https://www.bbc.com/news/technology-20910769`, Jan 2013. [Accessed: 2018-09-29].

[18] Matt Novak. The National Automated Highway System That Almost Was. `https://www.smithsonianmag.com/history/the-national-automated-highway-system-that-almost-was-63027245/`, May 2013. [Accessed: 2018-09-29].

[19] Steve Crowe. Back to the future: Autonomous driving in 1995. *Robotics Business Review*, Apr 2015.

[20] Alex Davies. This Is Big: A Robo-Car Just Drove Across the Country. `https://www.wired.com/2015/04/delphi-autonomous-car-cross-country/`, Jun 2017. [Accessed: 2018-09-30].

[21] John Ramsey. Self-driving cars to be tested on Virginia highways. `http://www.richmond.com/news/article_b1168b67-3b2b-5274-8914-8a3304f2e417.html`, Jun 2015. [Accessed: 2018-09-30].

[22] John Ramsey. On the Road – Waymo. `https://waymo.com/ontheroad/`, July 2018. [Accessed: 2018-09-30].

[23] Theo Leggett. Who is to blame for 'self-driving car' deaths? `https://www.bbc.com/news/business-44159581`, May 2018. [Accessed: 2018-10-01].

[24] Panos J Antsaklis, Kevin M Passino, and SJ Wang. An introduction to autonomous control systems. *IEEE Control Systems*, 11(4):5–13, 1991.

[25] SAE International. Automated driving: levels of driving automation are defined in new sae international standard j3016. 2014.

[26] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[27] Andrew J. Hawkins. MIT built a self-driving car that can navigate unmapped country roads. `https://www.theverge.com/2018/5/13/17340494/mit-self-driving-car-unmapped-country-rural-road`, May 2018. [Accessed: 2018-10-02].

[28] Rachel Gordon CSAIL. Self-driving cars for country roads. `http://news.mit.edu/2018/self-driving-cars-for-country-roads-mit-csail-0507`, May 2018. [Accessed: 2018-10-02].

[29] Romain Dillet. Uber acquires Otto to lead Uber's self-driving car effort. `https://techcrunch.com/2016/08/18/uber-acquires-otto-to-lead-ubers-self-driving-car-effort-report-says/`, Aug 2016. [Accessed: 2018-10-04].

[30] Darrell Etherington. Self-driving truck startup Embark raises Peterbilt. `https://techcrunch.com/2017/07/18/self-driving-truck-startup-embark-raises-15m-partners-with-peterbilt/`, Oct 2017. [Accessed: 2018-10-04].

[31] Reuters. Waymo working on self-driving trucks. `https://www.reuters.com/article/us-waymo-selfdriving-truck/waymo-working-on-self-driving-trucks-idUSKBN18T04V`, Jun 2017. [Accessed: 2018-10-04].

[32] Arturo Dávila and Mario Nombela. Sartre: Safe road trains for the environment. In *Conference on Personal Rapid Transit PRT@ LHR*, volume 3, pages 2–3, 2010.

[33] Driverless cars take to the road — Result In Brief. `https://cordis.europa.eu/result/rcn/90263_en.htm`, Jan 2013. [Accessed: 2018-10-04].

[34] Snyder OKs self-driving vehicles on Michigan's roads. http://www.detroitnews.com/article/20131227/POLITICS02/312270067/1148/rss25, Dec 2013. [Accessed: 2018-10-04].

[35] UK to allow driverless cars on public roads in January. https://www.bbc.com/news/technology-28551069, Jul 2014. [Accessed: 2018-10-04].

[36] Tyler Cowen. Can I See Your License, Registration and C.P.U.? https://www.nytimes.com/2011/05/29/business/economy/29view.html, May 2011. [Accessed: 2018-10-04].

[37] Mike Ramsey. Self-Driving Cars Could Cut Down on Accidents, Study Says. https://www.wsj.com/articles/self-driving-cars-could-cut-down-on-accidents-study-says-1425567905, Mar 2015. [Accessed: 2018-10-04].

[38] Chunka Mui. Will Driverless Cars Force A Choice Between Lives And Jobs? https://www.forbes.com/sites/chunkamui/2013/12/19/will-the-google-car-force-a-choice-between-lives-and-jobs/?ss=future-tech#6bd95a983995, Dec 2013. [Accessed: 2018-10-04].

[39] James M Anderson, Kalra Nidhi, Karlyn D Stanley, Paul Sorensen, Constantine Samaras, and Oluwatobi A Oluwatola. *Autonomous vehicle technology: A guide for policymakers*. Rand Corporation, 2014.

[40] Tom Simonite. Self-Driving Motorhome: RV Of the Future? http://vogeltalksrving.com/2014/11/self-driving-motorhome-rv-of-the-future/, Nov 2014. [Accessed: 2018-10-04].

[41] Randal O'Toole. *Gridlock: why we're stuck in Traffic and What to Do About It*. Cato Institute, 2010.

[42] Gary Silberg, Richard Wallace, G Matuszak, J Plessers, C Brower, and Deepak Subramanian. Self-driving cars: The next revolution. *White paper, KPMG LLP & Center of Automotive Research*, page 36, 2012.

[43] Nicholas Negroponte, Randal Harrington, Susan R McKay, and Wolfgang Christian. Being digital. *Computers in Physics*, 11(3):261–262, 1997.

[44] Peter G Neumann. Risks of automation: a cautionary total-system perspective of our cyberfuture. *Communications of the ACM*, 59(10):26–30, 2016.

[45] Max Ufberg. Whoops: The Self-Driving Tesla May Make Us Love Urban Sprawl Again. `https://www.wired.com/2014/10/tesla-self-driving-car-sprawl/`, Jun 2017. [Accessed: 2018-10-04].

[46] Steve Heath. *Embedded systems design*. Elsevier, 2002.

[47] Michael Barr and Anthony Massa. *Programming embedded systems: with C and GNU development tools*. " O'Reilly Media, Inc.", 2006.

[48] Michael Barr. Real men program in c. *Embedded Systems Design*, 22(7):3, 2009.

[49] Cesare Alippi. *Intelligence for embedded systems*. Springer, 2014.

[50] Umar Zakir Abdul Hamid, Hairi Zamzuri, and Dilip Kumar Limbu. Internet of vehicle (iov) applications in expediting the implementation of smart highway of autonomous vehicle: A survey. In *Performability in Internet of Things*, pages 137–157. Springer, 2019.

[51] A Linan Colina, A Vives, A Bagula, M Zennaro, and E Pietrosemoli. Iot in 5 days. *E-Book, March*, page 1, 2015.

[52] ITU. Internet of Things Global Standards Initiative. `https://www.itu.int/en/ITU-T/gsi/iot/Pages/default.aspx`, 2015. [Accessed: 2018-10-15].

[53] Charith Perera, Chi Harold Liu, and Srimal Jayawardena. The emerging internet of things marketplace from an industrial perspective: A survey. *IEEE Transactions on Emerging Topics in Computing*, 3(4):585–598, 2015.

[54] Zeeshan Latif. Wireless protocols for internet of things. `https://medium.com/blueeast/wireless-protocols-for-internet-of-things-7ed0bd860c66`, Jan 2018. [Accessed: 2018-10-17].

[55] Paul M Moubarak and Pinhas Ben-Tzvi. Adaptive manipulation of a hybrid mechanism mobile robot. In *Robotic and Sensors Environments (ROSE), 2011 IEEE International Symposium on*, pages 113–118. IEEE, 2011.

[56] Ackerman, Evan. TurtleBot Inventors Tell Us Everything About the Robot. `https://spectrum.ieee.org/automaton/robotics/diy/interview-turtlebot-inventors-tell-us-everything-about-the-robot`, Mar 2013. [Accessed: 2018-10-23].

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

[58] Tim Bailey and Hugh Durrant-Whyte. Simultaneous localization and mapping (slam): Part ii. *IEEE Robotics & Automation Magazine*, 13(3):108–117, 2006.

[59] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, José Neira, Ian Reid, and John J Leonard. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on Robotics*, 32(6):1309–1332, 2016.

[60] Peter Mountney, Danail Stoyanov, Andrew Davison, and Guang-Zhong Yang. Simultaneous stereoscope localization and soft-tissue mapping for minimal invasive surgery. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 347–354. Springer, 2006.

[61] Stefan Kohlbrecher, Oskar Von Stryk, Johannes Meyer, and Uwe Klingauf. A flexible and scalable slam system with full 3d motion estimation. In *Safety, Security, and Rescue Robotics (SSRR), 2011 IEEE International Symposium on*, pages 155–160. IEEE, 2011.

[62] Wolfgang Hess, Damon Kohler, Holger Rapp, and Daniel Andor. Real-time loop closure in 2d lidar slam. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pages 1271–1278. IEEE, 2016.

[63] Gruhler, Matthias. Wiki: navigation. `http://wiki.ros.org/navigation`, Mar 2017. [Accessed: 2018-11-26].

Appendix A

Code for Programming RedBot

A.1 Wheel Encoders

```
#include <RedBot.h>
RedBotMotors motors;
RedBotEncoder encoder = RedBotEncoder(A2, 10);
int buttonPin = 12;
//192 ticks per wheel rev
int countsPerRev = 192;
// diam = 65mm / 25.4 mm/in
float wheelDiam = 2.56;
// Redbot wheel circumference = pi*D
float wheelCirc = PI*wheelDiam;
// turn 90 degree to left/right
int turnLeft = 0;
int turnRight = 1;

void setup()
```

```
{
    pinMode(buttonPin, INPUT_PULLUP);
    Serial.begin(9600);
}

void loop(void)
{
    // set the power for left & right motors on button press
    if (digitalRead(buttonPin) == LOW)
    {
      driveStraight(12, 150);
      travelDistance();
      turn(turnRight);
      driveStraight(12, 150);
      travelDistance();
      turn(turnLeft);
    }
}

void travelDistance()
{
    long lCount = 0;
    long rCount = 0;
    float lNumRev;
    float rNumRev;
    float distance;
    lCount = encoder.getTicks(LEFT);
    rCount = encoder.getTicks(RIGHT);
    lNumRev = (float)lCount / (float)countsPerRev;
    rNumRev = (float)rCount / (float)countsPerRev;
    distance = (lNumRev + rNumRev)/2*wheelCirc;
    //print the result
    Serial.print("already drove ");
    Serial.print(distance);
    Serial.println(" inches straightly");
}

void driveStraight(float distance, int motorPower)
{
    long lCount = 0;
    long rCount = 0;
    long targetCount;
    float numRev;

    // variables for tracking the left and right encoder counts
    long prevlCount, prevrCount;
    // diff between current encoder count and previous count
    long lDiff, rDiff;

    // variables for setting left and right motor power
    int leftPower = motorPower;
    int rightPower = motorPower;

    // variable used to offset motor power on right vs left
    // to keep straight.
    // offset amount to compensate Right vs. Left drive
    int offset = 5;
    // calculate the target # of rotations
    numRev = distance / wheelCirc;
    // calculate the target count
    targetCount = numRev * countsPerRev;

    // debug
    Serial.print("driveStraight() ");
    Serial.print(distance);
    Serial.print(" inches at ");
    Serial.print(motorPower);
    Serial.println(" power.");

    Serial.print("Target: ");
    Serial.print(numRev, 3);
    Serial.println(" revolutions.");
    Serial.println();

    // print out header
    Serial.print("Left\t");    // "Left" and tab
    Serial.print("Right\t");   // "Right" and tab
    Serial.println("Target count");
    Serial.println("============================");

    encoder.clearEnc(BOTH);     // clear the encoder count
```

```
delay(100);  // short delay before starting the motors.

motors.drive(motorPower);  // start motors

while (rCount < targetCount)
{
    // while the right encoder is less than the target count
    // -- debug print
    // the encoder values and wait -- this is a holding loop.
    lCount = encoder.getTicks(LEFT);
    rCount = encoder.getTicks(RIGHT);
    Serial.print(lCount);
    Serial.print("\t");
    Serial.print(rCount);
    Serial.print("\t");
    Serial.println(targetCount);

    motors.leftDrive(leftPower);
    motors.rightDrive(rightPower);

    // calculate the rotation "speed" as a difference
    // in the count from previous cycle.
    lDiff = (lCount - prevlCount);
    rDiff = (rCount - prevrCount);

    // store the current count as the "previous" count
    // for the next cycle.
```

```
        prevlCount = lCount;
        prevrCount = rCount;

        // if left is faster than the right,
        // slow down the left / speed up right
        if (lDiff > rDiff)
        {
          leftPower = leftPower - offset;
          rightPower = rightPower + offset;
        }
        // if right is faster than the left,
        // speed up the left / slow down right
        else if (lDiff < rDiff)
        {
          leftPower = leftPower + offset;
          rightPower = rightPower - offset;
        }
        // short delay to give motors a chance to respond.
        delay(50);
    }
    // now apply "brakes" to stop the motors.
    motors.brake();
}

void turn(int direct)
{
    long lCount = 0;
    long rCount = 0;
    long Count = 0;
    long targetCount;
    float numRev = 2.4;

    // variables for setting left and right motor power
    int leftPower = 0;
    int rightPower = 0;

    // calculate the target count
    targetCount = numRev * countsPerRev;

    //debug
    // print out header
    Serial.print("Left\t");
    Serial.print("Right\t");
    Serial.print("Target count\t");
    Serial.println("Direction");
    Serial.println("============================");
    // clear the encoder count
    encoder.clearEnc(BOTH);
    // short delay before starting the motors.
    delay(100);

    if(direct == turnRight)
    {
      leftPower = 200;
```

```
    rightPower = 80;
  }
  else
  {
    leftPower = 80;
    rightPower = 200;
  }

  while(Count < targetCount)
  {
      if(direct == turnLeft)
          Count = encoder.getTicks(RIGHT);
      else
          Count = encoder.getTicks(LEFT);

      lCount = encoder.getTicks(LEFT);
      rCount = encoder.getTicks(RIGHT);
      Serial.print(lCount);
      Serial.print("\t");
      Serial.print(rCount);
      Serial.print("\t");
      Serial.print(targetCount);
      Serial.print("\t");
      if(direct == turnLeft)
          Serial.println("turn left");
      else
          Serial.println("turn right");
```

```
        motors.leftDrive(leftPower);
        motors.rightDrive(rightPower);
        delay(50);
    }
    motors.brake();
}
```

A.2 IR Sensor

```
#include <RedBot.h>
// initialize a left sensor object on A3
RedBotSensor left = RedBotSensor(A3);
// initialize a center sensor object on A6
RedBotSensor center = RedBotSensor(A6);
// initialize a right sensor object on A7
RedBotSensor right = RedBotSensor(A7);

// constants that are used in the code. LINETHRESHOLD is
// the level to detect if the sensor is on the line or not.
// If the sensor value is greater than this
// the sensor is above a DARK line.
//
// SPEED sets the nominal speed

#define LINETHRESHOLD 800
// sets the nominal speed. Set to any number from 0 - 255.
```

```
#define SPEED 60

RedBotMotors motors;
int leftSpeed;
int rightSpeed;

void setup()
{
    Serial.begin(9600);
    delay(2000);
    Serial.println("IR Sensor Readings: ");
    delay(500);
}

void loop()
{
    Serial.print(left.read());
    Serial.print("\t");  // tab character
    Serial.print(center.read());
    Serial.print("\t");  // tab character
    Serial.print(right.read());
    Serial.println();

    // if on the line drive left and right at the same speed
    // (left is CCW / right is CW)
    if(center.read() > LINETHRESHOLD)
    {
        leftSpeed = -SPEED;
        rightSpeed = SPEED;
    }

    // if the line is under the right sensor,
    // adjust relative speeds to turn to the right
    else if(right.read() > LINETHRESHOLD)
    {
        leftSpeed = -(SPEED + 50);
        rightSpeed = SPEED - 50;
    }

    // if the line is under the left sensor,
    // adjust relative speeds to turn to the left
    else if(left.read() > LINETHRESHOLD)
    {
        leftSpeed = -(SPEED - 50);
        rightSpeed = SPEED + 50;
    }

    // if all sensors are on black or up in the air, stop the
    // motors. otherwise, run motors given the control speeds
    // above.
    if((left.read() > LINETHRESHOLD) &&\
    (center.read() > LINETHRESHOLD)\
    && (right.read() > LINETHRESHOLD) )
    {
        motors.stop();
    }
    else
    {
        motors.leftMotor(leftSpeed);
        motors.rightMotor(rightSpeed);

    }
    // add a delay to decrease sensitivity.
    delay(20);
}
```

A.3 Bumpers

```
#include <RedBot.h>
RedBotMotors motors;
// initialzes bumper object on pin 3
RedBotBumper lBumper = RedBotBumper(3);
// initialzes bumper object on pin 11
RedBotBumper rBumper = RedBotBumper(11);
// variable to store the button Pin
int buttonPin = 12;
// state variable to store the bumper value
int lBumperState;
// state variable to store the bumper value
int rBumperState;
```

```
void setup()
{
  // nothing here.
}

void loop()
{
  motors.drive(255);
  // default INPUT state is HIGH,
  // it is LOW when bumped
  lBumperState = lBumper.read();
  rBumperState = rBumper.read();
  // left side is bumped/
    if (lBumperState == LOW)
  {
    reverse();     // backs up
    turnRight();   // turns
  }
  // right side is bumped/
  if (rBumperState == LOW)
  {
    reverse();    // backs up
    turnLeft();   // turns
  }

}
```

```
// reverse() function: backs up at full power
void reverse()
{
  motors.drive(-255);
  delay(500);
  motors.brake();
  delay(100);
}

// turnRight() function: turns RedBot to the Right
void turnRight()
{
  motors.leftMotor(-150);   // spin CCW
  motors.rightMotor(-150); // spin CCW
  delay(500);
  motors.brake();
  delay(100);
}

// turnRight() function: turns RedBot to the Left
void turnLeft()
{
  motors.leftMotor(+150);   // spin CW
  motors.rightMotor(+150); // spin CW
  delay(500);
  motors.brake();
  delay(100);
}
```

Appendix B

Code and File for ROS Nodes

B.1 Code for Lidar Scanner

```
 //Lidar + ROS Publisher + servo
#include <ros.h>
#include <sensor_msgs/LaserScan.h>
#include <Wire.h>
#include <LIDARLite.h>
#include <Servo.h>

LIDARLite myLidarLite;
Servo myservo;

const int degree_increment = 16; //degree
const int measureTime_increment = degree_increment * 10; //ms
const int num_readings = 160/degree_increment;
const double pi = 3.141592;
// variable to store the servo position
```

```
int pos = degree_increment;

//ROS node handle
ros::NodeHandle  nh;

// ROS Serial Laser scan message definition
sensor_msgs::LaserScan scan;
// definition of the ROS publisher for the laser scan data
ros::Publisher scan_pub("scan", &scan);
// Frame ID used in the ROS topics
char frameid[] = "/laser_lite_v3";
float ranges[num_readings];

void setup()
{
  Serial.begin(57600);
  // Set configuration to default and I2C to 400 kHz
  myLidarLite.begin(0, true);
  // Change this number to try out alternate configurations
  myLidarLite.configure(0);
  myservo.attach(9);
  /* ROS related */
  nh.initNode();
  nh.advertise(scan_pub);
  scan.angle_min = degree_increment*pi/180.0;//2 degree
  scan.angle_max = 160.0*pi/180.0;//160 degree
  scan.angle_increment = degree_increment*pi/180.0;
  scan.time_increment = measureTime_increment/1000.0;
  scan.scan_time = num_readings*measureTime_increment/1000.0;
  scan.range_min = 0.01;
  scan.range_max = 40.0;
  scan.ranges_length = num_readings;
}

void loop()
{
  scan.header.stamp = nh.now();
  scan.header.frame_id = frameid;

  // goes from 16 degrees to 160 degrees
  // in steps of 16 degree
  for(pos = degree_increment; pos <= 160; pos += degree_increment)
  {
      myservo.write(pos);
      delay(measureTime_increment);
      ranges[pos/degree_increment-1] = myLidarLite.distance()/100.0
      nh.spinOnce();
  }
  scan.ranges = ranges;
  scan_pub.publish(&scan);

  nh.spinOnce();

  // goes from 160 degrees to 16 degrees
  // in steps of 16 degree
  for(pos = 160; pos > 0; pos -= degree_increment)
  {
      myservo.write(pos);
      delay(measureTime_increment);
      ranges[pos/degree_increment -1] = myLidarLite.distance()/100.0
      nh.spinOnce();
  }
  scan.ranges = ranges;
  scan_pub.publish(&scan);

  nh.spinOnce();
}
```

B.2 Launch File of Hector SLAM

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

<launch>

  <arg name="geotiff_map_file_path" default="$(find
  hector_geotiff)/maps"/>

  <param name="/use_sim_time" value="false"/>

  <node pkg="rviz" type="rviz" name="rviz"
    args="-d $(find hector_slam_launch)/rviz_cfg/
    mapping_demo.rviz"/>

  <include file="$(find hector_mapping)/launch/
  mapping_default.launch"/>

  <include file="$(find hector_geotiff)/launch/
  geotiff_mapper.launch">
    <arg name="trajectory_source_frame_name"
    value="scanmatcher_frame"/>
    <arg name="map_file_path"
    value="$(arg geotiff_map_file_path)"/>
  </include>

</launch>
```

B.3 Launch File of Hector Mapping

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

<launch>
  <arg name="tf_map_scanmatch_transform_frame_name"
  default="scanmatcher_frame"/>
  <arg name="base_frame" default="base_link"/>
  <arg name="odom_frame" default="nav"/>
  <arg name="pub_map_odom_transform" default="true"/>
  <arg name="scan_subscriber_queue_size" default="5"/>
  <arg name="scan_topic" default="scan"/>
```

```
<arg name="map_size" default="2048"/>

<node pkg="hector_mapping" type="hector_mapping"
name="hector_mapping" output="screen">

  <!-- Frame names -->
  <param name="map_frame" value="map" />
  <param name="base_frame" value="$(arg base_frame)" />
  <param name="odom_frame" value="$(arg base_frame)" />

  <!-- Tf use -->
  <param name="use_tf_scan_transformation" value="true"/>
  <param name="use_tf_pose_start_estimate" value="false"/>
  <param name="pub_map_odom_transform"
   value="$(arg pub_map_odom_transform)"/>

  <!-- Map size / start point -->
  <param name="map_resolution" value="0.050"/>
  <param name="map_size" value="$(arg map_size)"/>
  <param name="map_start_x" value="0.5"/>
  <param name="map_start_y" value="0.5" />
  <param name="map_multi_res_levels" value="2" />

  <!-- Map update parameters -->
  <param name="update_factor_free" value="0.4"/>
  <param name="update_factor_occupied" value="0.9" />
  <param name="map_update_distance_thresh" value="0.4"/>
```

```
    <param name="map_update_angle_thresh" value="0.06" />
    <param name="laser_z_min_value" value = "-1.0" />
    <param name="laser_z_max_value" value = "1.0" />

    <!-- Advertising config -->
    <param name="advertise_map_service" value="true"/>

    <param name="scan_subscriber_queue_size"
     value="$(arg scan_subscriber_queue_size)"/>
    <param name="scan_topic" value="$(arg scan_topic)"/>

    <!-- Debug parameters -->
    <!--
      <param name="output_timing" value="false"/>
      <param name="pub_drawings" value="true"/>
      <param name="pub_debug_output" value="true"/>
    -->
    <param name="tf_map_scanmatch_transform_frame_name"
     value="$(arg tf_map_scanmatch_transform_frame_name)" />
  </node>

  <node pkg="tf" type="static_transform_publisher"
  name="base_laser_broadcaster"
   args="0.1 0 0.2 0 0 0 base_link /laser_lite_v3 100"/>
</launch>
```

B.4 Launch File of Hector Geotiff

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

<launch>
  <arg name="trajectory_source_frame_name"
   default="/base_link"/>
  <arg name="trajectory_update_rate"
   default="4"/>
  <arg name="trajectory_publish_rate"
   default="0.25"/>
  <arg name="map_file_path"
   default="$(find hector_geotiff)/maps"/>
  <arg name="map_file_base_name"
   default="hector_slam_map"/>

  <node pkg="hector_trajectory_server"
   type="hector_trajectory_server"
   name="hector_trajectory_server" output="screen">
    <param name="target_frame_name"
     type="string" value="/map" />
    <param name="source_frame_name" type="string"
     value="$(arg trajectory_source_frame_name)" />
    <param name="trajectory_update_rate" type="double"
     value="$(arg trajectory_update_rate)" />
    <param name="trajectory_publish_rate" type="double"
     value="$(arg trajectory_publish_rate)" />
  </node>
  <node pkg="hector_geotiff" type="geotiff_node"
  name="hector_geotiff_node" output="screen"
  launch-prefix="nice -n 15">
    <remap from="map" to="/dynamic_map" />
    <param name="map_file_path" type="string"
    value="$(arg map_file_path)" />
    <param name="map_file_base_name" type="string"
    value="$(arg map_file_base_name)" />
    <param name="geotiff_save_period" type="double"
    value="0" />
    <param name="draw_background_checkerboard"
    type="bool" value="true" />
    <param name="draw_free_space_grid"
    type="bool" value="true" />
    <param name="plugins" type="string"
    value="hector_geotiff_plugins/TrajectoryMapWriter" />
  </node>

</launch>
```

B.5 Code for Obstacle Detection and Avoidance with Remote Control

```
//code for collision detection and avoidance with remote control
#include <RedBot.h>
```

```
RedBotMotors motors;
// variable for setting the drive power
int leftPower;
int rightPower;
RedBotEncoder encoder = RedBotEncoder(A2, 10);
// initialzes bumper object on pin 3
RedBotBumper lBumper = RedBotBumper(3);
// initialzes bumper object on pin 11
RedBotBumper rBumper = RedBotBumper(11);
// state variable to store the bumper value
int lBumperState;
// state variable to store the bumper value
int rBumperState;


void setup(void)
{
  Serial.begin(57600);
  Serial.println("Enter in 1 to start or 0 to stop\
  and click [Send].");
  Serial.print("If 7 is received, then enter in 2\
  to turn right and 3 to turn left");
  Serial.println();
}

void loop(void)
{
```

```
if(Serial.available() > 0)
{
  switch(Serial.parseInt())
  {
      //start the RedBot
      case 1:
      {
          driveStraight(60);
      }
      //turn right
      case 2:
      {
          turnRight();
          driveStraight(60);
          break;
      }

      //turn left
      case 3:
      {
          turnLeft();
          driveStraight(60);
          break;
      }

      //stop the RedBot
      default:
        {
            motors.brake();
            delay(100);  // short delay to let robot fully stop
            break;
        }
    }
  }
}

void driveStraight(int motorPower)
{
    long lCount = 0;
    long rCount = 0;
    float numRev;

    // variables for tracking the left and right encoder counts
    long prevlCount, prevrCount;
    // diff between current encoder count and previous count
    long lDiff, rDiff;

    // variables for setting left and right motor power
    int leftPower = motorPower;
    int rightPower = motorPower;

    // variable used to offset motor power on right vs left
    //to keep straight.
    // offset amount to compensate Right vs. Left drive
```

```
int offset = 5;

encoder.clearEnc(BOTH);    // clear the encoder count
delay(100);  // short delay before starting the motors.

motors.drive(motorPower);  // start motors

// default INPUT state is HIGH, it is LOW when bumped
lBumperState = lBumper.read();
// default INPUT state is HIGH, it is LOW when bumped
rBumperState = rBumper.read();
// collision detected
while((lBumperState == HIGH) && (rBumperState == HIGH))
{
    //check if "stop" command is received
    if(Serial.available() > 0)
    {
        //if "stop" is received, stop the RedBot
        if(Serial.parseInt() == 0)
        {
            motors.brake();
            delay(100);  // short delay to let robot fully stop
            break;
        }
    }
    lCount = encoder.getTicks(LEFT);
    rCount = encoder.getTicks(RIGHT);
```

```
motors.leftDrive(leftPower);
motors.rightDrive(rightPower);

// calculate the rotation "speed" as a difference
//in the count from previous cycle.
lDiff = (lCount - prevlCount);
rDiff = (rCount - prevrCount);

// store the current count as the "previous" count
//for the next cycle.
prevlCount = lCount;
prevrCount = rCount;

// if left is faster than the right, slow down the left
// and speed up right
if (lDiff > rDiff)
{
    leftPower = leftPower - offset;
    rightPower = rightPower + offset;
}
// if right is faster than the left, speed up the left
// and slow down right
else if (lDiff < rDiff)
{
    leftPower = leftPower + offset;
    rightPower = rightPower - offset;
```

```
        }
        // short delay to give motors a chance to respond.
        delay(50);
        //update the bumper state
        lBumperState = lBumper.read();
        rBumperState = rBumper.read();
        delay(50);
    }
    //collision detected
    //back up and stop
    reverse();
    //send requset back
    Serial.print(7);
}

// reverse() function -- backs up at full power
void reverse()
{
  motors.drive(-60);
  delay(500);
  motors.brake();
  delay(100);  // short delay to let robot fully stop
}

// turnRight() function -- turns RedBot to the Right
void turnRight()
{
  motors.leftMotor(-100);   // spin CCW
  motors.rightMotor(-100); // spin CCW
  delay(500);
  motors.brake();
  delay(100);   // short delay to let robot fully stop
}

// turnRight() function -- turns RedBot to the Left
void turnLeft()
{
  motors.leftMotor(+100);   // spin CW
  motors.rightMotor(+100); // spin CW
  delay(500);
  motors.brake();
  delay(100);   // short delay to let robot fully stop
}
```