The Project Report
Entitled

Autonomous Navigation of Robot in Indoor Environment using SLAM

Submitted in
Partial fulfilment for the award of the Degree
of

Bachelor of Technology
in
Electrical Engineering

by

CHIRAG SOLANKI (U15EE027)
HARSHIT AGHERA (U15EE039)
DEEPESHWAR KUMAR (U15EE054)
SUSHEN KASHYAP (U15EE070)
SARTHAK DESAI (U15EE077)

Under the guidance of

Dr. S.N. Sharma

DEPARTMENT OF ELECTRICAL ENGINEERING
SARDAR VALLABHBHAI NATIONAL INSTITUTE OF TECHNOLOGY
SURAT – 395007

MAY 2019

SARDAR VALLABHBHAI NATIONAL INSTITUTE OF TECHNOLOGY
SURAT-395 007, GUJRAT, INDIA
DEPARTMENT OF ELECTRICAL ENGINEERING
SVNIT

CERTIFICATE

This is to certify that the Project report entitled **"*AUTONOMOUS NAVIGATION OF ROBOT IN INDOOR ENVIRONMENT USING SLAM*"** submitted by **CHIRAG SOLANKI (U15EE027), HARSHIT AGHERA (U15EE039), DEEPESHWAR KUMAR (U15EE054), SUSHEN KASHYAP (U15EE070), SARTHAK DESAI (U15EE077),** is a record of bona fide work carried out by them in partial fulfilment of the requirement for the award of the degree of "**BACHELOR OF TECHNOLOGY IN ELECTRICAL ENGINEERING**".

Date:

Place: SURAT

Signature and date

Supervisor

Signature and date

Examiners

Signature and date

Head of Department

ACKNOWLEDGMENT

We would like to express our utmost gratitude to **Dr. S.N. Sharma**, Head of Department, Electrical Engineering, Sardar Vallabhbhai National Institute of Technology for providing his valuable assistance and guidance throughout the duration of this project.

In addition, a thank you to **Professor M.N. Bhusavalwala**, Associate Professor, Electrical Engineering Department for helping in the procurement of required materials. We would also like to thank the members of **Team DRISHTI**, technical hobby club of SVNIT, Surat, for their cooperation in use of their lab resources.

We would also like to expand our deepest gratitude to all those who have directly and indirectly guided us.

ABSTRACT

The use of Autonomous robotics has increased noticeably in last decade, especially for indoor applications. An autonomous robot can map its surrounding environment and decide upon navigation strategy, hence finding great use in many fields such as military application, disaster management, patient assistance etc. In order to achieve complete independent functioning, entire system can be divided in three subsections.

1. Map generation using Simultaneous Localization and Mapping [SLAM].
2. Localisation using Adaptive Monte Carlo particle filter.
3. Path planning using A* algorithm.

Map generation is based on the gmapping ROS package which works on Rao-Blackwellized Particle Filter. Generated map is then used for goal tracking and path planning purpose. Localisation in this map is done by amcl ROS package. Navigation stack is used for the navigation of robot in the path chosen. All the above-mentioned algorithms are processed on ROS platform based on Ubuntu 16.04 running on Intel i5-8300H processor. These algorithms output command velocity which is then sent to controller using MAVLink protocol. The microcontroller then implements the set velocity based on the unicycle model closed loop control.

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

1.1 Problem Statement

The autonomous navigation problem deals with the current position of the robot with respect to a global map and finds the most cost-effective path to the goal point as specified by the user. In order to effectively execute said process, the robot must have the knowledge of its surroundings as well as its motion and this information is extracted by means of sensors mounted on the robot. Various algorithms are developed for the autonomous navigation problem and the study of particle filter-based SLAM (Simultaneous Localisation And Mapping) algorithm is presented in this report.

1.2 Available Approaches

Autonomous Navigation systems are widely employed in warehouse automation, patient assistance, military guidance systems and is rapidly finding its importance in self-driving vehicles. This report deals mainly with the solutions developed for indoor environment autonomous navigation systems. Presently, the autonomous navigation systems can be broadly classified into three main sub-systems.

1.2.1 Line Following Robots

The line following robot is the simplest autonomous navigation system. In these systems, the path to be traversed is distinctly marked (either black strip on a white surface or vice-versa) and the robot is equipped with a line sensor calibrated to distinguish between the path marked by the strip and the floor surface.

The robot construction is cheap, and the computation power required is very low. This system can be implemented on a low-end microcontroller. As there is only one fixed path, no path planning is involved and hence the most economic path may not be the one traversed. In order to get current robot location, additional sensors must be used. The robot will only travel on the line and no obstacle avoidance is possible, hence it is not suitable for places where installing strips will be difficult and lot of floor movement occurs.

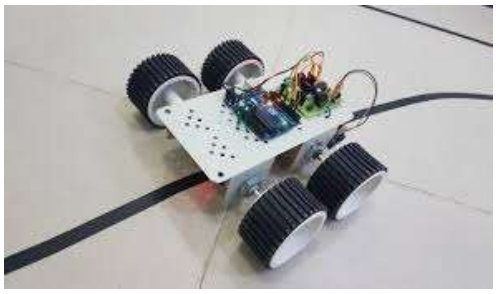


Figure 1.1: Line Follower Robot

1.2.2 Autonomous Navigation System using Local Map

These are slightly more complex than the line following robots. In this system, the robot traverses along a local map, created by the robot using sensor data, between its initial point and destination point. The robot requires its odometry data obtained primarily from wheel encoders which can be fused with other pose determining sensors, like heading from magnetometer, and is generally equipped with object detection sensors for obstacle avoidance.

As all points in the local map are with reference to robot coordinate system, the goal points also need to be given in the robot coordinate system. This can be problematic in situations where the environment is relatively unknown. Also, errors in robot sensor data can lead to erratic goal points, which can lead to accidents. However, as this system is computationally lighter it can be implemented on low-end microcontrollers, and hence is relatively cheaper. The robot construction is also simple. In case of static environments, it can generate the most optimal path and hence they are widely used in small-scale warehouse automation, autonomous room cleaning systems, etc.

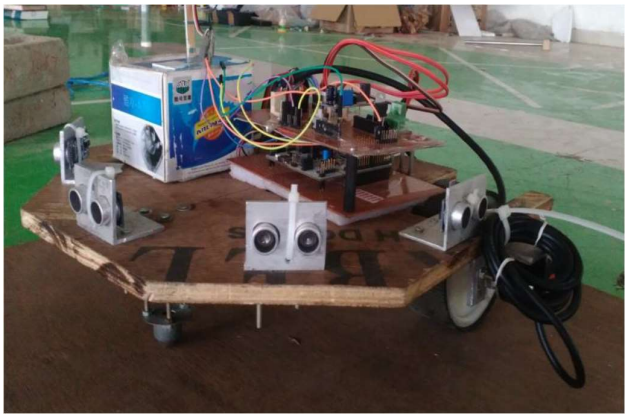


Figure 1.2: Robot used for Autonomous Navigation using Local Map

1.2.3 Autonomous Navigation System using Global Map

These robots have a depth sensor, typically a LIDAR (Light Detection And Ranging) sensor, equipped which helps in identifying environment boundaries and other obstacles which are useful in creating the global map. The robot then traverses within this map based on user input. As the global map is available, the goal points can be specified in terms of global map coordinates. These systems have very heavy processing requirements used for map generation and path planning algorithms and hence are normally equipped with a mid-range processor. As the construction is not simple, these systems are costly.

The robot plans the most optimal path based on its current location and destination accounting for static obstacles as well as dynamic obstacles. Also, because of localisation algorithms, we can monitor robot movement in real time and can get the exact position with respect to map coordinate system at any given instant. These algorithms are generally self-correcting and therefore, random errors are minimised. The map generation is a continuous process which takes place as the robot moves in an environment. As a result, these systems can be employed in unknown environment without any loss of efficiency. Because of the advantages stated above, these systems are of great interest for hobbyists, researchers as well as in industries.

Figure 1.3: Robot used for Autonomous Navigation using Global Map

1.3 System Overview

This report mainly focuses on the autonomous navigation systems which are using global maps. The robot designed is a two wheeled differentially driven robot. It uses a brushed DC motor with attached encoder assembly. The lower control system is running on the Arduino Due board based on the Atmel SAM3X8E ARM Cortex M3 CPU. Codes used in lower control system was written using Arduino IDE. Depth sensing is achieved using a Microsoft Xbox 360 Kinect Sensor. The robot is also equipped with HC-SR04 Ultrasonic Sensor and HMC5883L 3-axis Magnetometer. The upper control system is implemented using ROS (Robot Operating System) Kinetic Kame made for Ubuntu 16.04 (Xenial) release running on Intel Core i5 8300H processor.

Mapping is achieved using gmapping algorithm based on Rao-Blackwellized particle filter. AMCL (Adaptive Monte-Carlo Localisation) is used for solving localisation problem. Path planning is achieved using A* Search algorithm. A PID controller is implemented for motor velocity control.

Communication between lower and upper control system is via USB Serial Communication based on MAVLink protocol.

The robot is powered by two 12V Li-Po (Lithium-Polymer) batteries.

The figure below represents the general block diagram of our system. Detailed explanation of the system will be given in subsequent chapters.

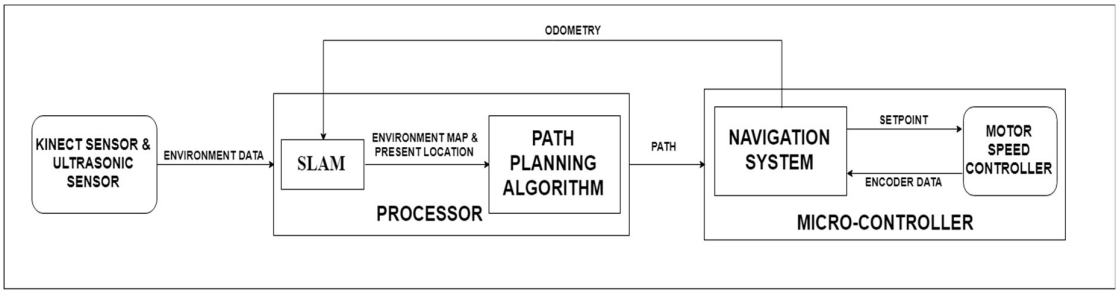


Figure 1.4: Block Diagram of Robot System

2 ROBOT PHYSICAL MODEL

2.1 Mechanical Chassis

The chassis of the robot provides two major functionalities. First is to provide support and weight distribution for the laptop computer which handles the upper control of the system and second is to provide housing for various electronic components and batteries.

Figure 2.1: Robot Chassis

The main body of chassis is made up of two stacked sheets of 4mm thickness aluminium composite panel (ACP), The profile of which is demonstrated in fig 2.2.

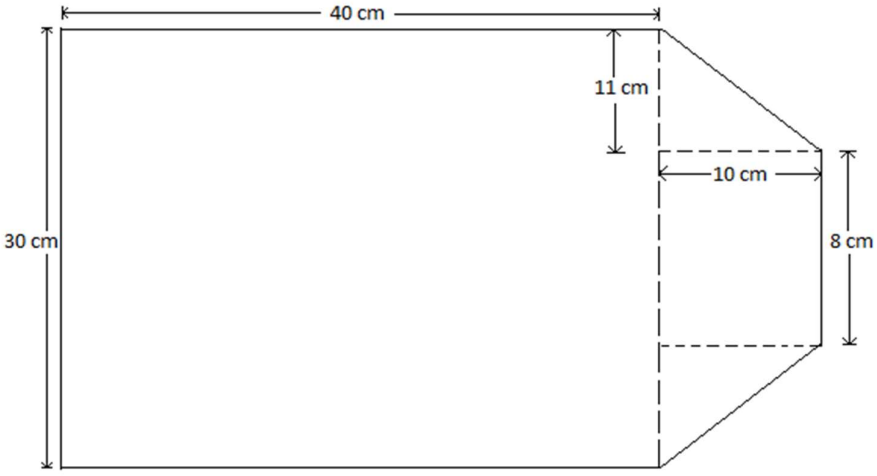
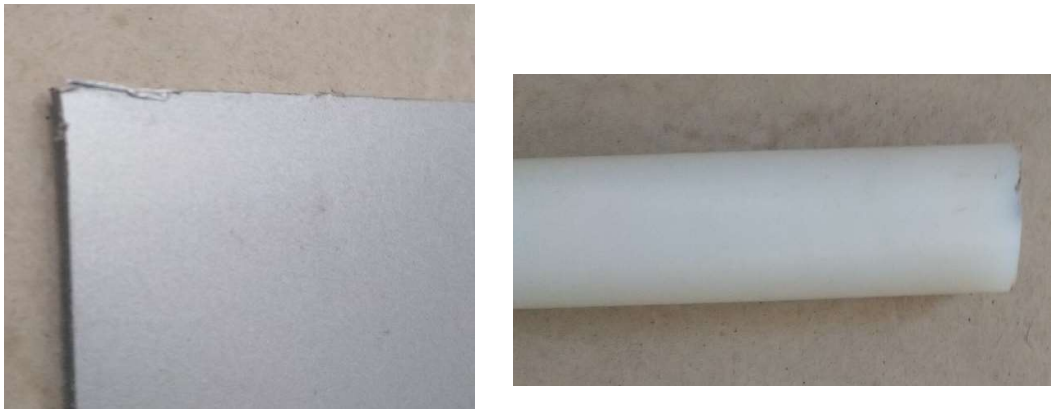


Figure 2.2: Main Body Profile

These sheets are separated from each other by five 12cm height nylon spacers of 2cm diameter each. The spacers are capped on each end's with rubber washers of 3cm diameter and 1cm height.

The spacers are affixed to both the ACP sheets through washers with stainless steel countersunk screws having dimension of 0.6cm face diameter and 4cm length.



(i) (ii)

Figure 2.3: Materials Used (i) Aluminium Composite Panel (ii) Nylon Rod

The placement of the nylon spacers are illustrated fig 2.4.

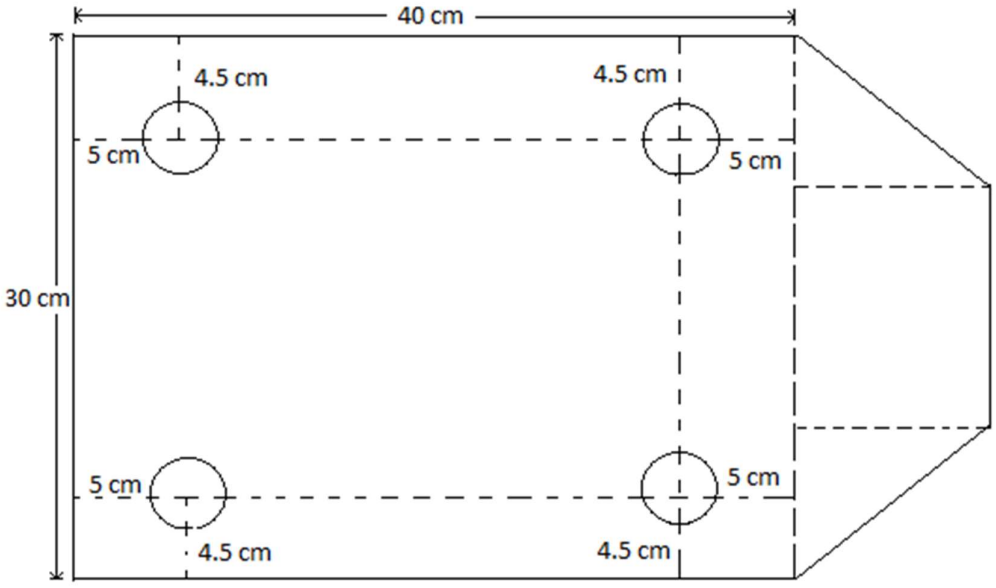


Figure 2.4: Spacer Placements

The lower sheet has an indentation of 2cm × 8cm to accommodate the wheels of the robot and the upper sheet has a window of 12 cm × 12cm to facilitate an access to electronic components mounted on the lower sheet.

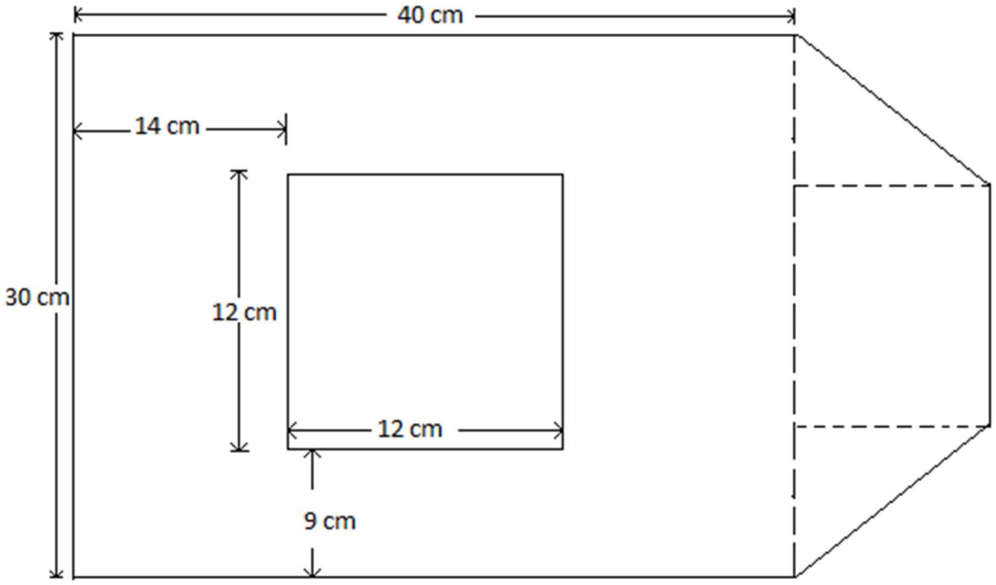


Figure 2.5: Upper Body Sheet

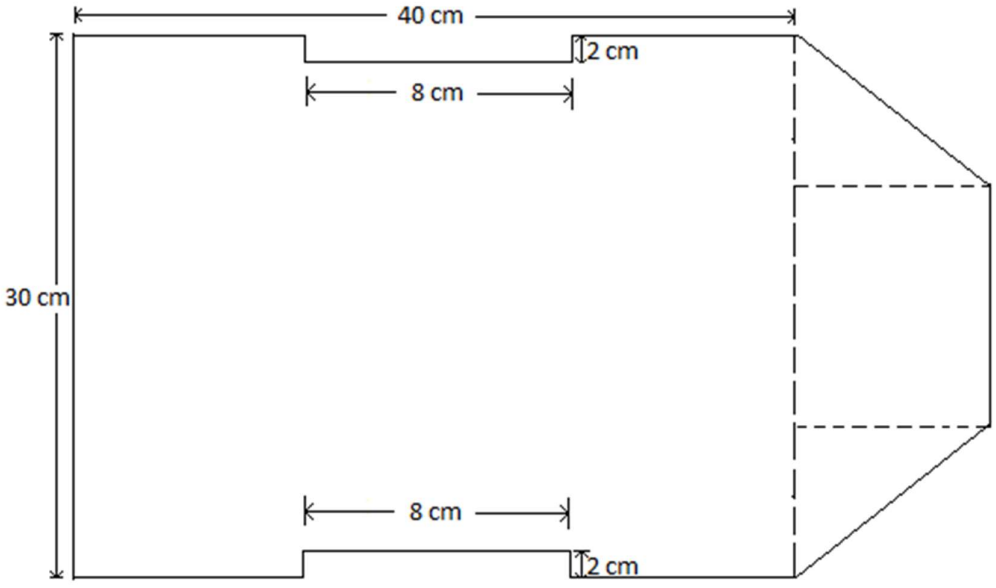


Figure 2.6: Lower Body Sheet

The entire weight of robot including upper control laptop computer is supported by 5 castor wheel and two drive wheels. The castor wheel has a height of 1.3cm. The drive

wheels have a diameter of 7cm and width of 4cm. The periphery of the drive wheel is covered with 2mm rubber grip to provide better traction with the ground.

(i) (ii)

Figure 2.7: Wheels Used (i) Castor wheel (ii)Drive wheel

The castor wheel as fixed such that their vertical axis is shifted 2.5cm inwards with respect to the vertical axis of its nearby spacer as shown in figure 2.11(c).

As illustrated in figure 2.1, Five strips of dimension 4.5 cm × 13 cm are secured at the centre of three front faces of the robot and at front of left and right edge of the robot using 2.5cm × 1.3cm L clamps. Holes are drilled at 5.5cm from the bottom of said strip to accommodate HC-SR04 ultrasonic sensors. Fig 2.8 illustrates this strip.

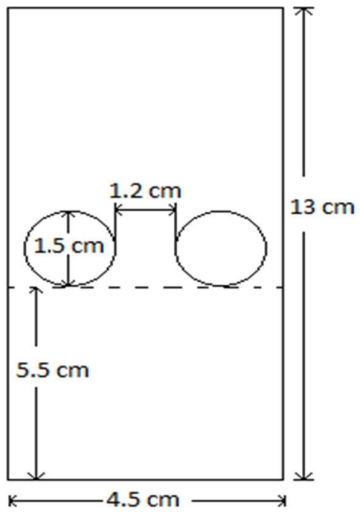


Figure 2.8: HCSR04 Housing Strip

The motor used to provide motion to the robot is a 12V, 100 RPM DC geared motor with a 29,520 pulse per revolution rotary encoder coupled coaxially with the base motor's shaft. The motor has the following mechanical specifications.

General Specifications	
Stall Torque (Kgcm)	35
Shaft Diameter (mm)	6
Gear Box Diameter (mm)	37
Shaft Length (mm)	30
Motor Length (mm)	63
Weight (gms)	180
Gearbox Ratio	1:180

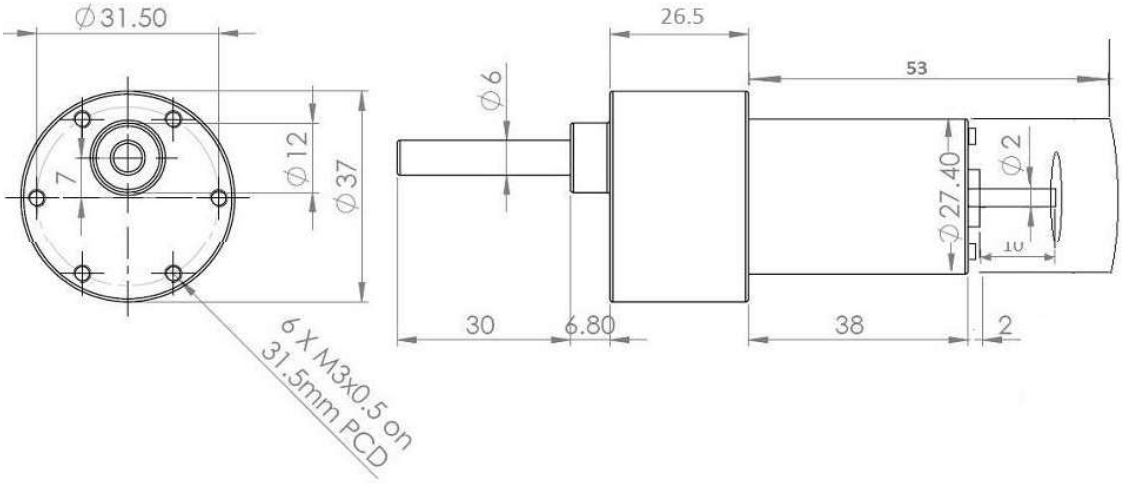


Figure 2.9: Motor Mechanical Design.

Base Motor Specifications	
No-load Speed (RPM)	18000
Nominal Voltage (V)	12
No-load Current (A)	0.8
Rated Current (A)	1.04
Rated Torque (gcm)	151
Stall Current (A)	7.5 (max)

Figure 2.10 Base Motor Electrical Specification

The various point of view for the assembled robot chassis in solid-works CAD software is illustrated in the figures below.

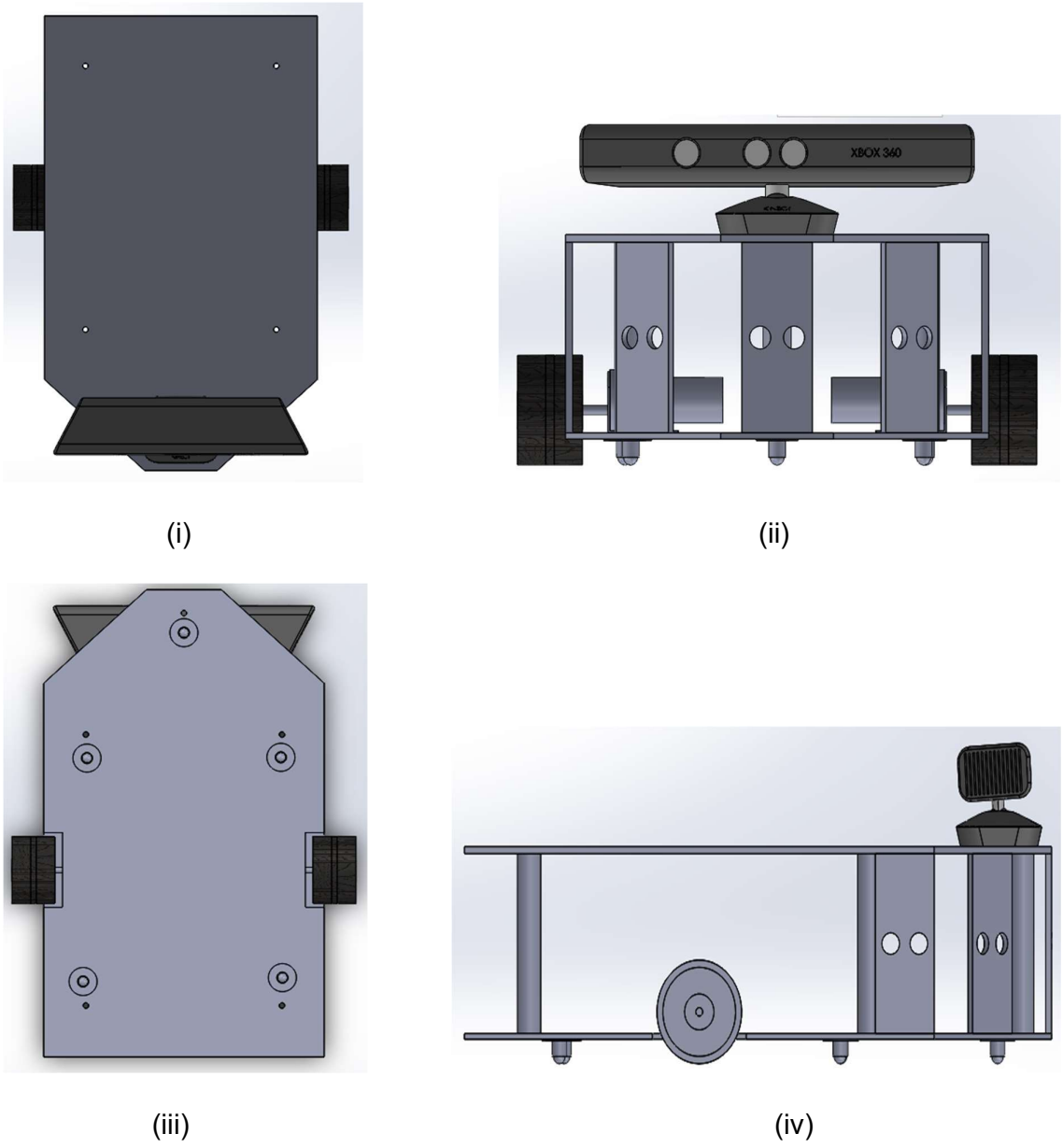


Figure 2.11: Orthogonal View (i) Top view (ii) Front view (iii) Bottom view (iv) Side view

The isometric view of the assembled robot in solid-works CAD software is illustrated in fig 2.12.

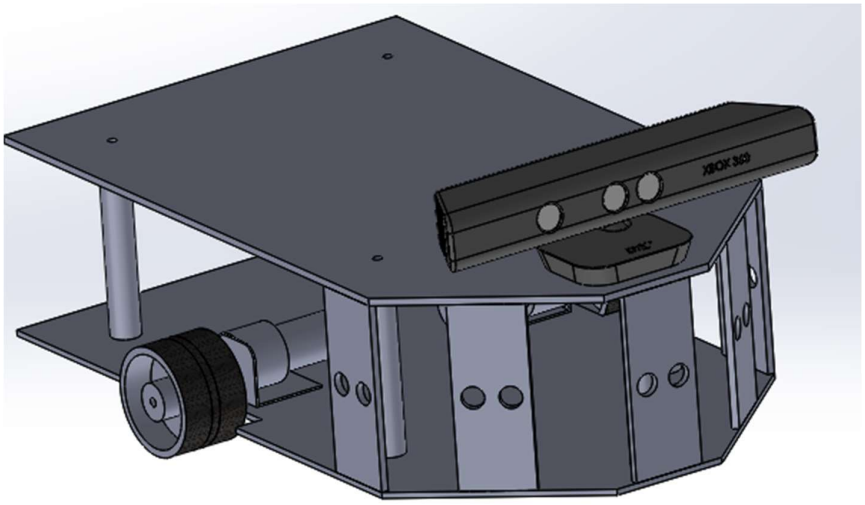


Figure 2.12: Isometric View of Assembled Robot

2.2 Electronics Hardware

2.2.1 Microcontroller: Arduino Due

The lower control system is implemented on an Arduino Due Board based on the 32-bit Atmel SAM3X8E ARM Cortex M3 CPU. It has 54 digital input/output pins (of which 12 can be used as PWM outputs), 12 analog inputs, 4 UARTs (hardware serial ports), a 84 MHz clock, an USB OTG capable connection, 2 DAC (digital to analog), 2 TWI, a power jack, an SPI header, a JTAG header, a reset button and an erase button. This board runs on 3.3V supply and hence logic level converters are necessary for interfacing with other components running on 5V level.

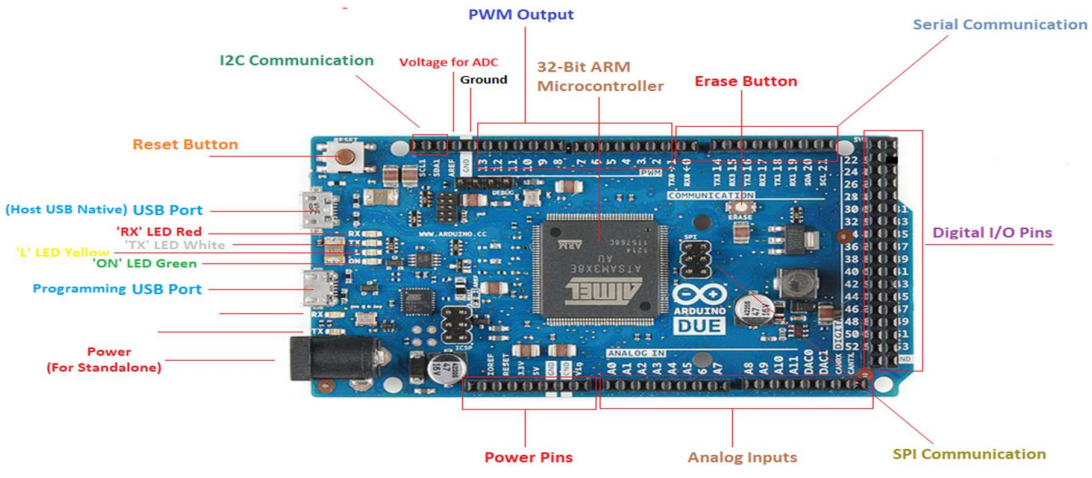


Figure 2.13: Arduino Due Microcontroller

2.2.2 Cytron Motor Driver MDD10A

Cytron MDD10A is a dual channel, n-MOSFET based H-bridge motor driver. It is used to amplify and condition the power signal fed to the motor according to control input. It has the following features:

- Bi-directional control of 2 brushed DC motor.
- Support motor voltage ranges from 5V to 25V.
- Maximum current up to 13A continuous and 30A peak (10 seconds) for each channel.
- Solid state components provide faster response time and eliminate the wear and tear of the mechanical relay.
- Fully NMOS H-Bridge for better efficiency and no heat sink is required.
- Speed control PWM frequency up to 20KHz. Support both locked-antiphase and sign-magnitude PWM operation.
- 2 activation buttons for fast test on each channel.

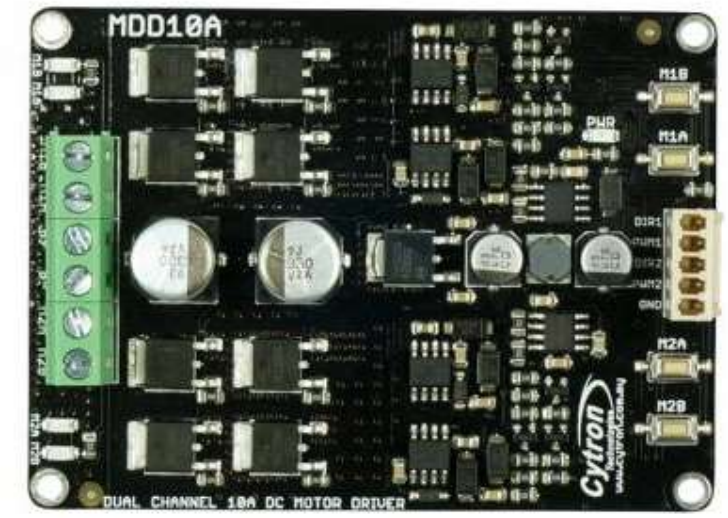


Figure 2.14: Cytron MDD10A Motor Driver

2.2.3 HC-SR04 Ultrasonic Sensor

Ultrasonic Sensor is being used as a protective unit for obstacle detection and avoidance. Ultrasonic ranging module HC - SR04 provides 2cm - 400cm non-contact measurement function, the ranging accuracy can reach to 3mm. The modules include ultrasonic transmitters, receiver and control circuit. The operating voltage is 5V DC

and current is 15mA. The measuring angle is about 15^0. The basic principle of work is:

1. Initialise by giving a 10μs pulse on Trigger Pin.
2. The Module automatically sends eight 40 kHz pulses and detects whether there is a pulse signal back.
3. If the signal is back, measuring time of high output IO duration, which is the time from sending ultrasonic to returning, distance can be known as:

$$Test\ distance\ =\ (high\ level\ time \times velocity\ of\ sound\ \left(\frac{340M}{S}\right))/\ 2$$

Figure 2.15: HC-SR04 Ultrasonic Sensor

2.2.4 HMC5883L Magnetometer

The Honeywell HMC5883L is a surface-mount, multi-chip module designed for low-field magnetic sensing with a digital interface for applications such as low-cost compassing and magnetometry. Its features are:

- 3-Axis Magnetoresistive Sensors and ASIC (Application Specific IC) in a 3.0x3.0x0.9mm LCC Surface Mount Package.
- 12-Bit ADC Coupled with Low Noise AMR Sensors Achieves 2 milli-gauss Field Resolution in ±8 Gauss Fields.
- Low Voltage Operations (2.16 to 3.6V) and Low Power Consumption (100 μA).
- I2C Digital Interface.
- Fast 160 Hz Maximum Output Rate.

The commercially available breakout board for this sensor is being used. The heading obtained from the magnetometer is being used to correct the odometry data from encoder. The heading can be obtained from the magnetometer reading as per the

formula:$heading\ \theta(in\ rad)\ =\ \tan^{-1}\frac{y}{x}$, where x & y are the magnetometer readings of X & Y axes respectively.

Figure 2.16: HMC5883L Magnetometer

2.2.5 Logic Level Converter (LLC)

As the controller is working at 3.3VDC level and the remaining components except magnetometer are working at 5VDC level, we need a bidirectional logic level converter to faithfully step-up 3.3V output signal to 5V and step-down 5V input signal to 3.3V. An n-channel MOSFET (BSS138) is used to design the LLC as shown in figure below.

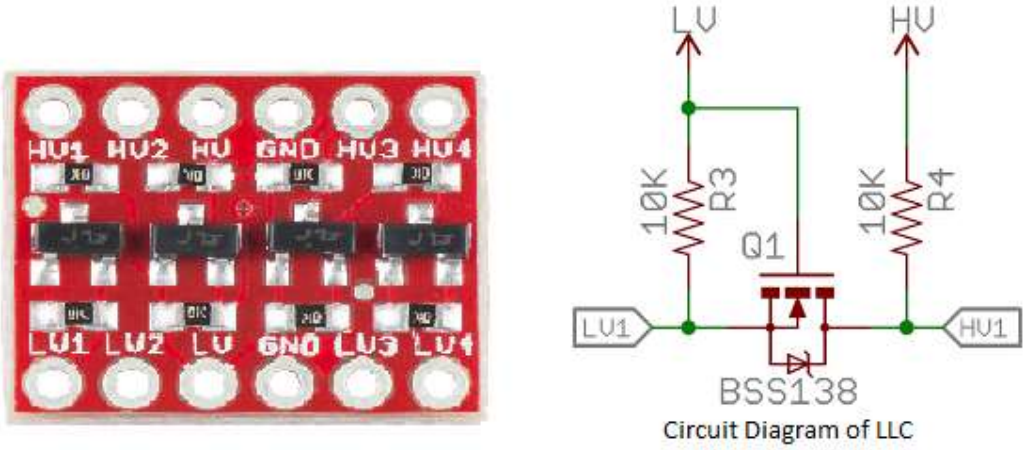


Figure 2.17: Bidirectional Logic Level Converter

2.2.6 12V Li-Po Battery

Two 12V batteries are used to power the entire robot. One battery powers the peripheral components and motor while the other is an isolated source used to power the Kinect sensor. The battery used is a 3000mAh, 11.1V, 3 cell, 30C Orange Li-Po battery. It is equipped with XT-60 connectors.

Figure 2.18: 12V Orange Li-Po Battery

2.2.7 LM2596 DC-DC Buck Converter

In order to get constant 5V and 3.3V DC supply from 12V battery, two DC-DC Buck Converter based on LM2596 IC is used. Its main features are:

Input Voltage	4-35 V
Output Voltage	1.25-30 V
Maximum Output Current	3 A
Switching Frequency	150 kHz
Conversion Efficiency	92%
Output Voltage Ripple	±50 mV

Figure 2.19: LM2596 DC-DC Buck Converter

2.2.8 XL6009 DC-DC Boost Converter

Kinect sensor requires a constant 12VDC supply. A boost converter is used to maintain the 12VDC to Kinect sensor in the event of battery discharge. It has the following features:

Input Voltage	3-32 V
Output Voltage	5-35 V
Maximum Output Current	4 A
Switching Frequency	400 kHz
Conversion Efficiency	94%
Output Voltage Ripple	±50 mV

Figure 2.20: XL6009 DC-DC Boost Converter

2.2.9 Microsoft Xbox 360 Kinect Sensor

The depth sensing task is being performed by the Microsoft Kinect Sensor. It is a 3D Camera developed my Microsoft. It contains two main camera units working together to get the depth data.

- Colour VGA video camera - This video camera aids in facial recognition and other detection features by detecting three colour components: red, green and blue. Microsoft calls this an "RGB camera" referring to the colour components it detects.
- Depth sensor - An infrared projector and a monochrome CMOS (complementary metal-oxide semiconductor) sensor work together to "see" the room in 3-D regardless of the lighting conditions.

Both cameras have a resolution of 640x480 pixels and run at 30FPS. The minimum sensing distance is 1.8m according to the manufacturer. It requires a constant 12V, 1A power being supplied by an isolated 12V battery and it transmits data serially via USB. It also has a microphone array.

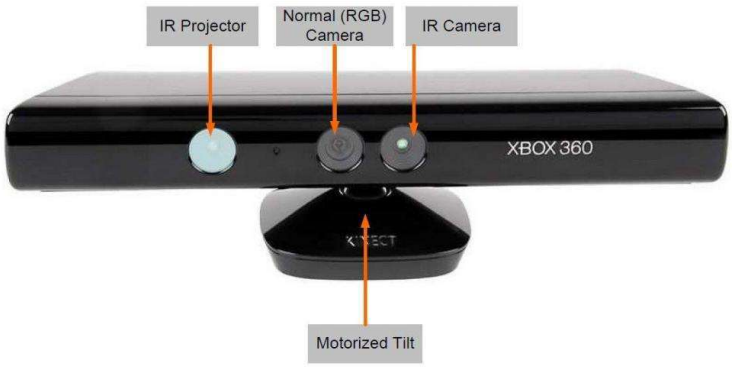


Figure 2.21: Microsoft Xbox 360 Kinect Sensor

2.2.10 Intel Core i5-8300H Processor

It is the main processing unit of the robot. It is a 4 core, 8 thread, 2.3GHz processor. It has cache memory of 8MB.

2.2.11 Power Board

This circuit board contains screw connectors for connecting batteries. It also has the 5V and 3.3V buck converters as well as the 12V boost converter. Screw connectors for Kinect supply and Motor Driver supply, a micro USB port to power Arduino Due and the power connection of control board is also made on this board. Its schematic is shown below.

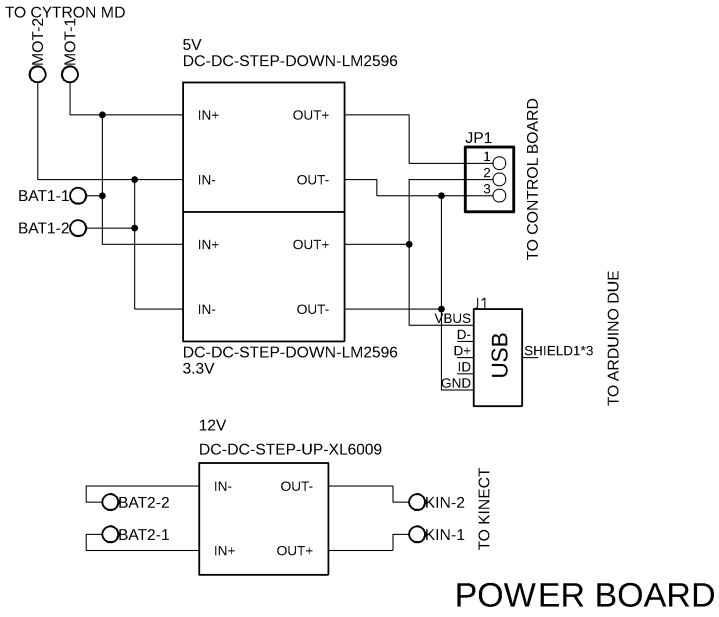


Figure 2.22: Power Board Schematic

2.2.12 Control Board

All the components are interfaced and merged on a circuit board and the connections are soldered. The control board consists of the Arduino Due board, the Logic Level Converters and header pins for encoder, Cytron MDD10A motor driver, five HC-SR04 Ultrasonic Sensors, and the HMC5883L magnetometer. It draws power from the power board connected via header pins. Three power lines, one each for 5V, 3.3V and Ground are drawn on the control board. An external reset button for Arduino Due is also provided. The schematic of control board is shown below.

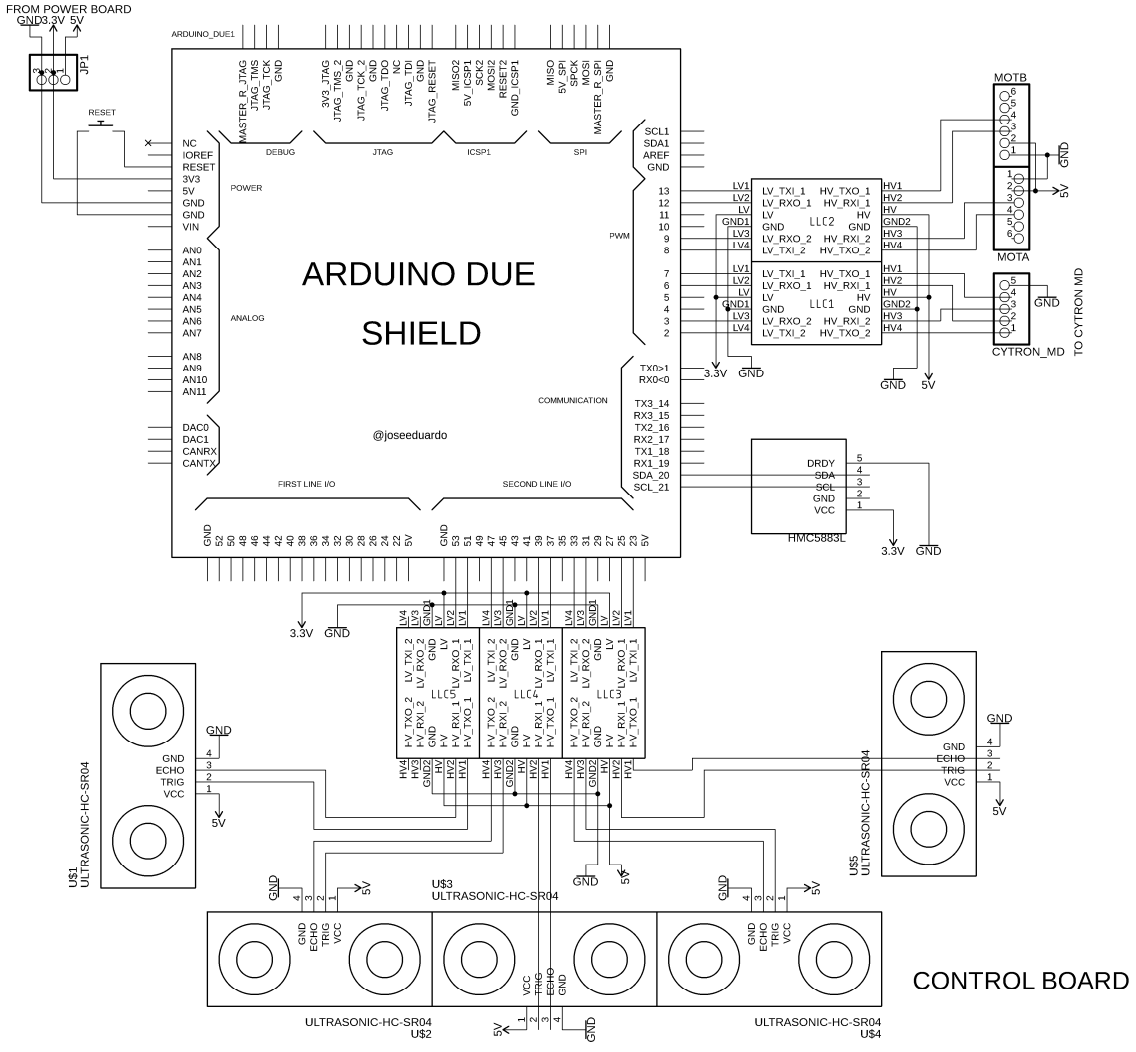


Figure 2.23: Control Board Schematic

3 ROBOT ODOMETRIC MODEL

Two-wheeled differential drive robots have many benefits over four-wheel robots. We generally use discrete time control system. Because of wide use of two-wheeled differential drive robots, there is a need to develop an efficient control system algorithm and models for improving performance. Effective control strategies rely on modelling of system. There are few models developed to control differential drive.

The model takes into account the robot kinematic and dynamic constraints, leading to bounded velocities and accelerations that are compatible with those an actual mobile robot can perform. The main intention is to use the model itself as a controller in a closed-loop control scheme. This approach has several advantages. Having the parameters of the model/controller a clear physical sense, they can be easily tuned to comply with the desired robot dynamic behaviour. Moreover, there is no need to identify the robot parameters, whenever some soft hypothesis on the robot velocities and accelerations are respected. Velocities to follow a path or reach a goal are automatically computed by the model/controller.

3.1 Robot Kinematics

As shown in figure 2.2 parameters R, d and L are robot dimensions and that can be used in modelling further. Consider the mobile robot in Figure 2.2. Using generalized coordinate vector $q = [x\ y\ \theta]$ the robot's posture can be defined on its whole configuration space of xy coordinate system. Hence, we can define our position using q vector at any point of time.

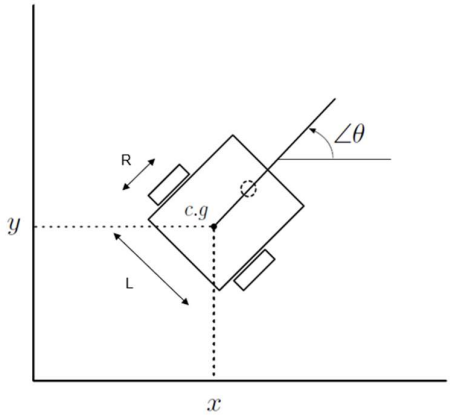


Figure 3.1: State Space Model

Now as shown in figure 3 robot has 2 input parameters $u = [v\ w]$. These two parameters are capable to move robot in our space frame.

Now using discrete time equation

$$\dot{x} = Ax + Bu \tag{1}$$

Here x is our state space parameters q and $\dot{x}$ is time differentiated. A and B are state space matrix.

$$x = [x\ y\ \theta]^T \tag{2}$$

Hence, we can write

$$\dot{x} = [\dot{x}\ \dot{y}\ \dot{\theta}]^T \tag{3}$$

So $\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = A \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + B \begin{bmatrix} v \\ w \end{bmatrix}$

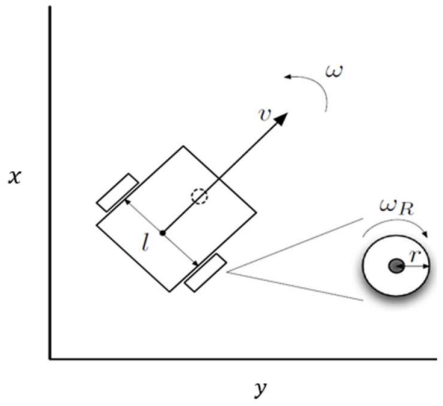


Figure 3.2: V-w model

This is called unicycle model for differential Drive model where we are giving input in terms of v and w. but in our real system v and w are not applicable. We are having left and right wheel velocities which is resulted into left and right wheel velocity.

From kinematics,

$$\dot{x} = v\,cos\theta$$

$$\dot{y} = v\,sin\theta \tag{4}$$

$$\dot{\theta} = w$$

From robot dimensions,

$$v = (R)(v_r + v_l)/2$$

$$w = (v_r - v_l) * R/L \tag{5}$$

Hence,

$$\dot{x} = \frac{(R)(v_r + v_l)}{2} * cos\theta$$

$$\dot{y} = \frac{(R)(v_r + v_l)}{2} * sin\theta$$

$$\dot{\theta} = (v_r - v_l) * R/L \tag{6}$$

Hence,

$$v = R * (v_r + v_l)/2$$

$$w = R * (v_r - v_l)/L \tag{7}$$

So, by solving above equations

$$v_r = (2v + wL)/2R$$

$$v_l = (2v - wL)/2R \tag{8}$$

Hence by controlling v_r and v_l we can change desired v and w which are out input parameters.

Now our control parameters are v_r and v_l only which are lest and right wheel velocities.

Current state can be found form the above model. Distance covered x and y are calculated from below equations,

$$\Delta x = v\ cos\theta\ \delta t$$
$$\Delta y = v\ sin\theta\ \delta t \tag{9}$$
$$\Delta\theta = w\ \delta t$$

Above obtained x, y and θ are current robot state.

4 LOWER CONTROL

4.1 The PID Controller

x, y and θ are current robot state which is obtained by robot state model. For moving robot on xy plane there is need to control v_r and v_l only. The robot dynamics having two motor for left and right wheel each. Control of both motors will give us change in v_r and v_l. v_r and v_l are measured with encoders and used to obtain current x and y value.

There is current x, y and θ and desired x, y and θ . For obtaining goal point error is calculated and corresponding v_r and v_l is obtained. To control v_r, v_l and w. PID is used for close loop control.

A proportional–integral–derivative controller (PID controller) is a control loop feedback mechanism widely used in industrial control systems and a variety of other applications requiring continuously modulated control.

The distinguishing feature of the PID controller is the ability to use the three control terms of proportional, integral and derivative influence on the controller output to apply accurate and optimal control. PID controller, which continuously calculates an error value e(t) as the difference between a desired setpoint r(t) and a measured process variable y(t) and applies a correction based on proportional, integral, and derivative terms. The controller attempts to minimize the error over time by adjustment of a control variable u(t), such as the opening of a control valve, to a new value determined by a weighted sum of the control terms.

$$u(t) = K_{\mathrm{p}} e(t) + K_{\mathrm{i}} \int_0^t e(\mathrm{t}) dt + K_{\mathrm{d}} \frac{de(t)}{dt} \tag{10}$$

where Kp, Ki and Kd, all non-negative, denote the coefficients for the proportional, integral, and derivative terms respectively (sometimes denoted *P*, *I*, and *D*).

Effects of increasing parameters individually,[6,7]

Parameter	Rise time	Overshoot	Steady state error	Settling time
P	Decrease	Increase	Decrease	Small
I	Decrease	Increase	Almost zero	Increase
D	Increase	Decrease	Small change	Decrease

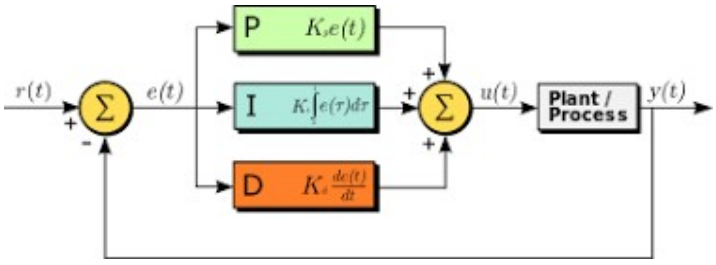


Figure 4.1: PID Controller

4.2 PID Tuning of Motor

Parameter tuning of PID i.e. Kp, Ki and Kd is done by observing real-time response of motor. Arduino Serial Plotter serves as the real-time graph plotting purpose. Screenshots of response of speed of DC motor for a unit step function is included below.[7]

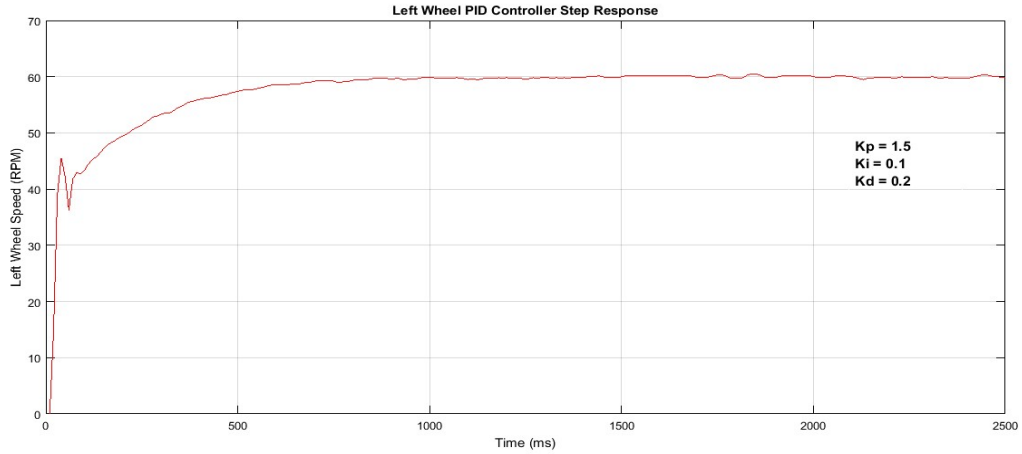


Figure 4.2: Left Wheel PID Controller Step Response

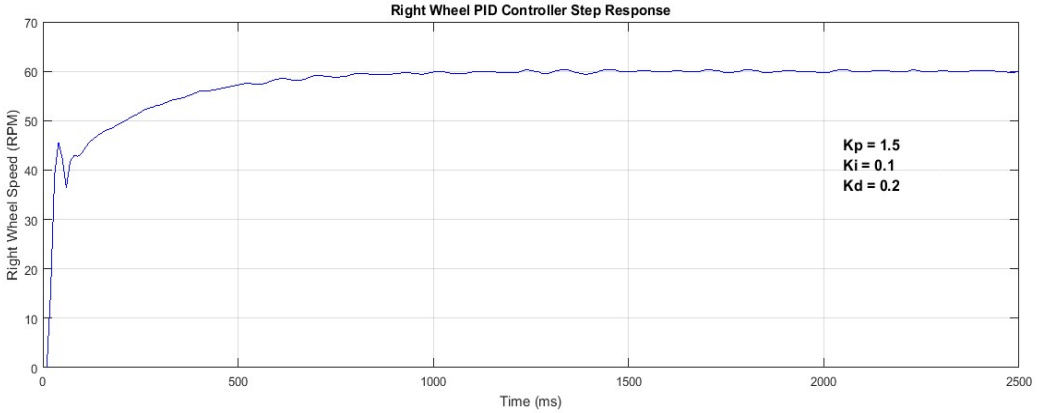


Figure 4.3: Right Wheel PID Controller Step Response

5 COMMUNICATION

Having multiple processing devices always needs inter communication. Embedded devices don't have Operating System, hence choosing unique communication mode and format plays a major role. In this robot Laptop had Ubuntu 16.04 hence we could communicate with serial communication over USB. Similarly, for the microcontroller, data was being sent and received with serial communication over USB.

5.1 Serial Communication over USB

Microcontroller and laptop both support serial communication over USB. For accessing data in serial C programming was used. With C we can directly access serial port data and use it over ROS. ROS also supports C language for generating node and publishing/subscribing to(from) nodes.

5.2 MAVLINK

As we were using serial communication between microcontroller and laptop, collecting data in proper format without loss might be big concern. There must be proper and same format for both ends. MAVLINK protocol is one of the best methods for node to node communication over ROS.

MAVLink(Micro Aerial Vehicle Link) is a very lightweight messaging protocol for communicating with drones (and between onboard drone components).

MAVLink follows a modern hybrid publish-subscribe and point-to-point design pattern: Data streams are sent / published as topics while configuration sub-protocols such as the mission protocol or parameter protocol are point-to-point with retransmission.

Messages are defined within XML files. Each XML file defines the message set supported by a particular MAVLink system, also referred to as a "dialect". The reference message set that is implemented by most ground control stations and autopilots is defined in *common.xml* (most dialects build on top of this definition).

The MAVLink toolchain uses the XML message definitions to generate MAVLink libraries for each of the supported programming languages. Drones, ground control stations, and other MAVLink systems use the generated libraries to communicate. These are typically MIT-licensed and can therefore be used without limits in any

closed-source application without publishing the source code of the closed-source application.[8]

We used following messages with given fields in our robot.

Index	Message ID	Message	Field
1	1	Left wheel PID gains	Kp
			Ki
			Kd
2	2	Right Wheel PID gains	Kp
			Ki
			Kd
3	3	Robot dimension	R
			d
4	31	Desired wheel RPM	Left wheel RPM
			Right Wheel RPM
5	32	Desired command value	v
			w
6	101	Initial Status	Initial heading
			Ultrasonic configuration
			Update frequency
			Encoder PPR
7	131	Robot position change	Delta_x
			Delta_y
			Delta_θ
8	132	Robot sensor reading	Left wheel ticks
			Right wheel ticks
			Current Heading
9	133	Robot Distance reading	Ultrasonic Reading

6 PROBABILITY ROBOTICS ALGORITHMS

6.1 Robot Odometry State Space Model and Map Of Environment

As odometry of all wheeled robot is non-linear equation of input action and state, state space equations are given by

$$x_t = g(u_t, x_{t-1}) + \varepsilon_t$$
$$z_t = h(x_t) + \delta_t \tag{11}$$

$x_t =$ robot position $[x\ y\ \theta]^T$

$u_t =$ distance travelled by the wheels $[r\ l]^T$

$z_t =$ map of environment

$\varepsilon_t =$ process noise

$\delta_t =$ measurement noise

The evolution of state and measurements is governed by probabilistic laws. In general, the state at time x_t is generated stochastically. Thus, it makes sense to specify the probability distribution from which x_t is generated. For most of control theory development of this project, state variable is assumed to have normal probability distribution also known as Gaussian distribution.

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{12}$$

x $=$ state random variable

$\mu =$ mean of distribution

$\sigma =$ variance of distribution

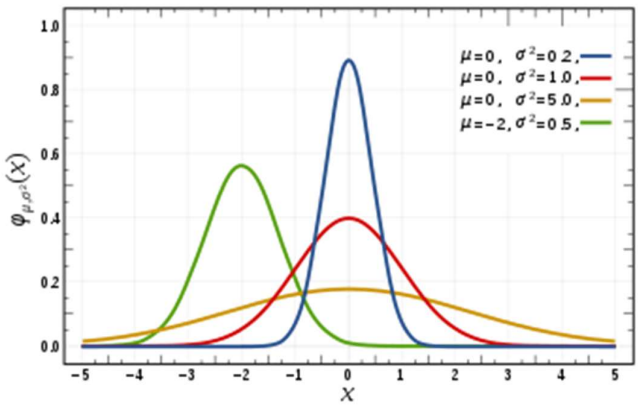


Figure 6.1: Sample Normal Probability Distribution Functions

At first glance, the emergence of state x_t might be conditioned on all past states, measurements, and controls. Here x_t is complete state and process is evidently Markov so, the probabilistic law characterizing the evolution of state might be given by a probability distribution of the following form: [1]

$$p(x_t|x_{0:t-1}, z_{1:t-1}, u_{1:t}) = p(x_t|x_{t-1}, u_t)$$
$$p(z_t|x_{0:t}, z_{1:t-1}, u_{1:t}) = p(z_t|x_t) \quad (13)$$

6.1.1 Bayes Filter

The most general algorithm for calculating beliefs is given by the *Bayes filter* algorithm. This algorithm calculates the belief distribution $\boldsymbol{bel}$ from measurement and control data.

$$\overline{\text{bel}}(x_t) = \int p(x_t|u_t, x_{t-1}, m) bel(x_{t-1}) dx$$
$$bel(x_t) = \eta p(z_t|x_t, m) \overline{bel}(x_t) \quad (14)$$

The Algorithms used in control scheme are developed based on Extended Kalman Filter and Particle Filter. Both of them are derived from Bayes Filter.[2]

6.2 Simultaneous Localization And Mapping

This section addresses one of the most fundamental problems in robotics, the simultaneous localization and mapping problem. This problem is commonly abbreviated as SLAM, and is also known as Concurrent Mapping and Localization, or CML. SLAM problems arise when the robot does not have access to a map of the

environment; nor does it have access to its own poses. Instead, all it is given are measurements $z_{1:t}$ and controls $u_{1:t}$. The term “simultaneous localization and mapping” describes the resulting problem: In SLAM, the robot acquires a map of its environment while simultaneously localizing itself relative to this map. SLAM is significantly more difficult than all robotics problems discussed thus far: It is more difficult than localization in that the map is unknown and has to be estimated along the way. It is more difficult than mapping with known poses, since the poses are unknown and have to be estimated along the way.

From a probabilistic perspective, there are two main forms of the SLAM problem, which are both of equal practical importance. One is known as the online SLAM problem: It involves estimating the posterior over the momentary pose along with the map:

$p(x_t, m|z_{1:t}, u_{1:t})$

Here x_t is the pose at time t, m is the map, and $z_{1:t}$ and $u_{1:t}$ are the measurements and controls, respectively. This problem is called the online SLAM problem since it only involves the estimation of variables that persist at time t. Many algorithms for the online SLAM problem are incremental: they discard past measurements and controls once they have been processed. The second SLAM problem is called the full SLAM problem. In full SLAM, we seek to calculate a posterior over the entire path $x_{1:t}$ along with the map, instead of just the current pose x_t:

$p(x_{1:t}, m|z_{1:t}, u_{1:t})$ [2].

6.2.1 Rao-Blackwellized Particle Filter for SLAM

Recently Rao-Blackwellized particle filters have been introduced as effective means to solve the simultaneous localization and mapping (SLAM) problem. This approach uses a particle filter in which each particle carries an individual map of the environment. Accordingly, a key question is how to reduce the number of particles. It presents adaptive techniques to reduce the number of particles in a Rao-Blackwellized particle filter for learning grid maps. It proposes an approach to compute an accurate proposal distribution taking into account not only the movement of the robot but also the most recent observation. This drastically decrease the uncertainty

about the robot's pose in the prediction step of the filter. Furthermore, we apply an approach to selectively carry out re-sampling operations which seriously reduces the problem of particle depletion. [4]

The key idea of the Rao-Blackwellized particle filter for SLAM is to estimate the joint posterior $p(x_{1:t}, m|z_{1:t}, u_{1:t})$ about the map m and the trajectory $x_{1:t}$ of the robot. This estimation is performed given the observations $z_{1:t}$ and the odometry measurements $u_{1:t-1}$ obtained by the mobile robot. The Rao-Blackwellized particle filter for SLAM makes use of the following factorization.

$$p(x_{1:t}, m|z_{1:t}, u_{1:t-1}) = p(m|x_{1:t}, z_{1:t}) \cdot p(x_{1:t}|z_{1:t}, u_{1:t-1}) \tag{15}$$

This factorization allows us to first estimate only the trajectory of the robot and then to compute the map given that trajectory. Since the map strongly depends on the pose estimate of the robot, this approach offers an efficient computation. This technique is often referred to as Rao-Blackwellization.

One of the most common particle filtering algorithms is the sampling importance resampling (SIR) filter. A Rao-Blackwellized SIR filter for mapping incrementally processes the sensor observations and the odometry readings as they are available. It updates the set of samples that represents the posterior about the map and the trajectory of the vehicle. The process can be summarized by the following four steps:

1) Sampling

   The next generation of particles is obtained from the previous generation by sampling from the proposal distribution. Often, a probabilistic odometry motion model is used as the proposal distribution.

2) Importance Weighting

$$w_t^{(i)} = \frac{p(x_{1:t}^{(i)}|z_{1:t}, u_{1:t-1})}{\pi(x_{1:t}^{(i)}|z_{1:t}, u_{1:t-1})} \tag{16}$$

3) Resampling

   Particles are drawn with replacement proportional to their importance weight. This step is necessary since only a finite number of particles is used to approximate a continuous distribution. Furthermore, resampling allows us to apply a particle filter in situations in which the target distribution differs from the proposal. After resampling, all the particles have the same weight.

4) Map Estimation

   For each particle, the corresponding map estimate $p(m|z_{1:t}, x_{1:t})$ is computed based on the trajectory of that sample and the history of observations.

6.2.1.1 Improved Proposal Distribution for RBPF

Improve of proposal distribution is dependent on weight computation of each particle. Most optimal weight calculation scheme suitable for SLAM purpose is given by [3]

$$w_t^{(i)} = w_{t-1}^{(i)} \cdot \int p(z_t|x')p(x'|x_{t-1}^{(i)}, u_{t-1})dx'$$
$$w_t^{(i)} = w_{t-1}^{(i)} \cdot \eta^{(i)} \tag{17}$$

$$\eta^{(i)} = \sum_{j=1}^{K} p(z_t|m_{t-1}^{(i)}, x_j) \cdot p(x_j|x_{t-1}^{(i)}, u_{t-1}) \tag{18}$$

6.2.1.2 Adaptive Resampling

On the one hand, resampling is necessary since only a finite number of particles are used to approximate the target distribution. On the other hand, the resampling step can remove good samples from the filter which can lead to particle impoverishment. Accordingly, it is important to find a criterion for deciding when to perform the resampling step. It is introduced the so-called effective sample size to estimate how well the current particle set represents the target posterior. In this work, we compute this quantity according to the formulation of Doucet as

$$N_{\text{eff}} = \frac{1}{\sum_{i=1}^{N} (\tilde{w}^{(i)})^2} \tag{19}$$

We resample each time N_{eff} drops below the threshold of N/2 where N is the number of particles. In extensive experiments, we found that this approach drastically reduces

the risk of replacing good particles, because the number of resampling operations is reduced and they are only performed when needed.

6.2.1.3 Overall Process Algorithm

The overall process is summarized in. Each time a new measurement tuple (u_{t-1}, z_t) is available, the proposal is computed for each particle individually and is then used to update that particle. This results in the following steps.

1. An initial guess $x_t'^{(i)} = x_{t-1}^{(i)} \oplus u_{t-1}$ for the robot's pose represented by the particle i is obtained from the previous pose $x_{t-1}^{(i)}$ of that particle and the odometry measurements u_{t-1} collected since the last filter update. Here, the operator $\oplus$ corresponds to the standard pose compounding operator.
2. A scan-matching algorithm is executed based on the map $m_{t-1}^{(i)}$ starting from the initial guess $x_t'^{(i)}$. The search performed by the scan-matcher is bounded to a limited region around $x_t'^{(i)}$. If the scan-matching reports a failure, the pose and the weights are computed according to the motion model (and the steps 3 and 4 are ignored).
3. A set of sampling points is selected in an interval around the pose $\hat{x}_t^{(i)}$ reported scan-matcher. Based on these points, the mean and the covariance matrix of the proposal are computed by pointwise evaluating the target distribution $p(z_t|m_{t-1}^{(i)}, x_j)p(x_j|x_{t-1}^{(i)}, u_{t-1})$ in the sampled positions $\mathcal{X}_j$. During this phase, also the weighting factor $\eta^{(i)}$ is computed as per discussed above.
4. The new pose $x_{t-1}^{(i)}$ of the particle i is drawn from the Gaussian approximation $\mathcal{N}(\mu_t^{(i)}, \Sigma_t^{(i)})$ of the improved proposal distribution.
5. Update of the importance weights.
6. The map $m_t^{(i)}$ of particle i is updated according to the drawn pose $x_t^{(i)}$ and the observation z_t.

After computing the next generation of samples, a resampling step is carried out depending on the value of N_{eff}. [3]

6.3 Adaptive Monte Carlo Localization

This chapter presents a number of probabilistic algorithms for mobile robot localization. Mobile robot localization is the problem of determining the pose of a robot relative to a given map of the environment. It is often called *position estimation* or *position tracking*. Mobile robot localization is an instance of the general localization problem, which is the most basic perceptual problem in robotics. This is because nearly all robotics tasks require knowledge of the location of the robots and the objects that are being manipulated (although not necessarily within a global map). Localization can be seen as a problem of coordinate transformation. Maps are described in a global coordinate system, which is independent of a robot's pose. Localization is the process of establishing correspondence between the map coordinate system and the robot's local coordinate system. Knowing this coordinate transformation enables the robot to express the location of objects of interests within its own coordinate frame—a necessary prerequisite for robot navigation. knowing the pose $x_t = (x\ y\ \theta)^T$ of the robot is sufficient to determine this coordinate transformation, assuming that the pose is expressed in the same coordinate frame as the map. [2]

$$\overline{\text{bel}}(x_t) = \int p(x_t|u_t, x_{t-1}, m) bel(x_{t-1}) dx$$
$$bel(x_t) = \eta p(z_t|x_t, m) \overline{bel}(x_t) \tag{20}$$

AMCL algorithm is based on Particle Filter derived from Bayes Filter. Four steps of particle filter are same as earlier mentioned in SLAM Section. The most important different is instead of simultaneously generating map, in AMCL generated Grid Occupancy Map is used. Availability of Map makes weight calculation process easier. Here notation are same as used earlier in SLAM section.

6.3.1 AMCL Algorithm

1: **Algorithm Augmented_MCL**($\mathcal{X}_{t-1}, u_t, z_t, m$):
2: static w_{slow}, w_{fast}
3: $\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$
4: for $m = 1$ to M do
5: $x_t^{[m]} = \text{sample\_motion\_model}(u_t, x_{t-1}^{[m]})$
6: $w_t^{[m]} = \text{measurement\_model}(z_t, x_t^{[m]}, m)$
7: $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
8: $w_{\text{avg}} = w_{\text{avg}} + \frac{1}{M} w_t^{[m]}$
9: endfor
10: $w_{\text{slow}} = w_{\text{slow}} + \alpha_{\text{slow}}(w_{\text{avg}} - w_{\text{slow}})$
11: $w_{\text{fast}} = w_{\text{fast}} + \alpha_{\text{fast}}(w_{\text{avg}} - w_{\text{fast}})$
12: for $m = 1$ to M do
13: with probability $\max(0.0,\ 1.0 - w_{\text{fast}}/w_{\text{slow}})$ do
14: add random pose to $\mathcal{X}_t$
15: else
16: draw $i \in \{1, \ldots, N\}$ with probability $\propto w_t^{[i]}$
17: add $x_t^{[i]}$ to $\mathcal{X}_t$
18: endwith
19: endfor
20: return $\mathcal{X}_t$

6.4 Robot Navigation Algorithm

One of the most basic things that a robot can do is to move around the world. To do this effectively, the robot needs to know where it is and where it should be going. This is usually achieved by giving the robot a map of the world, a starting location, and a goal location. In the previous lesson, we saw how to build a map of the world from sensor data. Now, we'll look at how to make your robot autonomously navigate from one part of the world to another, using this map and the ROS navigation packages.

Task of path planning for mobile robot is to determine sequence of maneuvers to be taken by robot in order to move from starting point to destination avoiding collision with obstacles.

Sample algorithms for path planning are:

- Dijkstra's algorithm
- A*
- D*
- Artificial potential field method
- Visibility graph method

Path planning algorithms may be based on graph or occupancy grid.

6.4.1 Graph Methods

Method that is using graphs, defines places where robot can be and possibilities to traverse between these places. In this representation graph vertices define places e.g. rooms in building while edges define paths between them e.g. doors connecting rooms. Moreover, each edge can have assigned weights representing difficulty of traversing path e.g. door width or energy required to open it. Finding the trajectory is based on finding the shortest path between two vertices while one of them is robot current position and second is destination.

6.4.2 Occupancy Grid Methods

Method that is using occupancy grid divides area into cells (e.g. map pixels) and assign them as occupied or free. One of cells is marked as robot position and another as a destination. Finding the trajectory is based on finding shortest line that do not cross any of occupied cells.

6.5 Environment Map

real-world applications, as maps are often available a priori or can be constructed by hand. Some application domains, however, do not provide the luxury of coming with an a priori map. Surprisingly enough, most buildings do not comply with the blueprints generated by their architects. And even if blueprints were accurate, they would not contain furniture and other items that, from a robot's perspective, determine the shape of the environment just as much as walls and doors. Being able to learn a map from scratch can greatly reduce the efforts involved in installing a mobile robot, and enable robots to adapt to changes without human supervision. In fact, mapping is one of the core competencies of truly autonomous robots.[11]

6.5.1 Occupancy Grid Mapping Algorithm

The goal of an occupancy mapping algorithm is to estimate the posterior probability over maps given the data:

$p(m \mid z_{1:t}, x_{1:t})$

where m is the map, $z_{1:t}$ is the set of measurements from time 1 to t, and is the set of robot poses from time 1 to t. The controls and odometry data play no part in the occupancy grid mapping algorithm since the path is assumed known.

Occupancy grid algorithms represent the map m as a fine-grained grid over the continuous space of locations in the environment. The most common type of occupancy grid maps is 2d maps that describe a slice of the 3d world.

If we let m_i denote the grid cell with index i (often in 2d maps, two indices are used to represent the two dimensions), then the notation $p(m_i)$ represents the probability that cell i is occupied. The computational problem with estimating the posterior $p(m \mid z_{1:t}, x_{1:t})$ is the dimensionality of the problem: if the map contains 10,000 grid cells (a relatively small map), then the number of possible maps that can be represented by this gridding is $2^{10,000}$ Thus calculating a posterior probability for all such maps is infeasible.

The standard approach, then, is to break the problem down into smaller problems of estimating

$p(m_i \mid z_{1:t}, x_{1:t})$

for all grid cells m_i . Each of these estimation problems is then a binary problem. This breakdown is convenient but does lose some of the structure of the problem, since it does not enable modelling dependencies between neighbouring cells. Instead, the posterior of a map is approximated by factoring it into

$$p(m \mid z_{1:t}, x_{1:t}) = \prod_i (m_i \mid z_{1:t}, x_{1:t})$$

Due to this factorization, a binary Bayes filter can be used to estimate the occupancy probability for each grid cell. It is common to use a log-odds representation of the probability that each grid cell is occupied. [2]

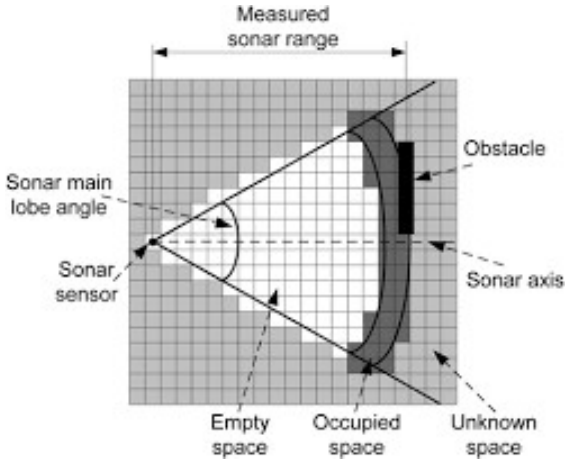


Figure 6.2: Sample Scan of Lidar Generated Occupancy Grid Map

6.5.2 Generated Maps

Parameters for generated map from above discussed algorithms are as follows.

***resolution**: 0.010000*

***size**: 10 m x 10 m*

***negate**: 0*

***occupied_thresh**: 0.65*

***free_thresh**: 0.196*

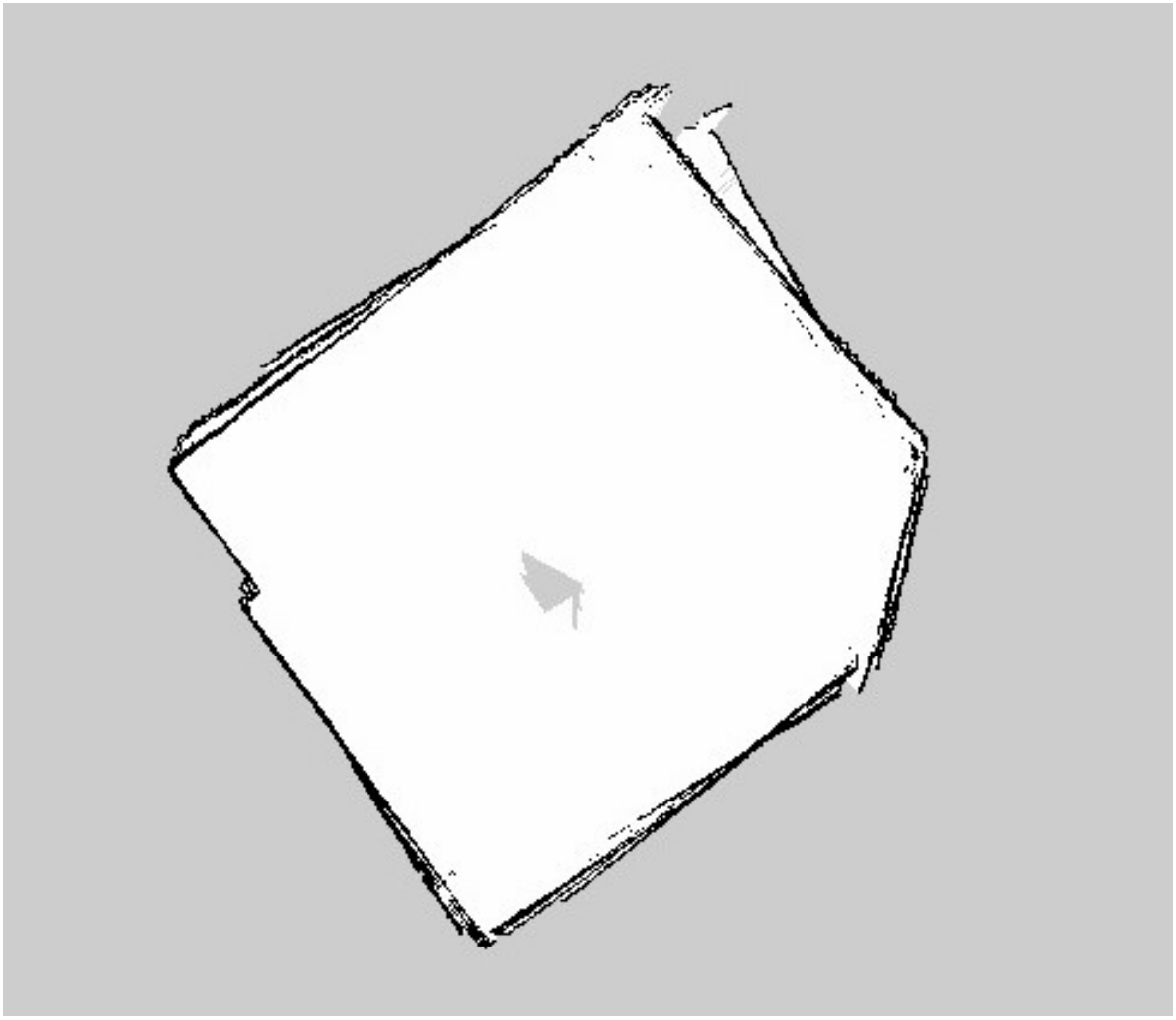


Figure 6.3: Map Generated for Experimental Setup at Swami Vivekanand Bhavan

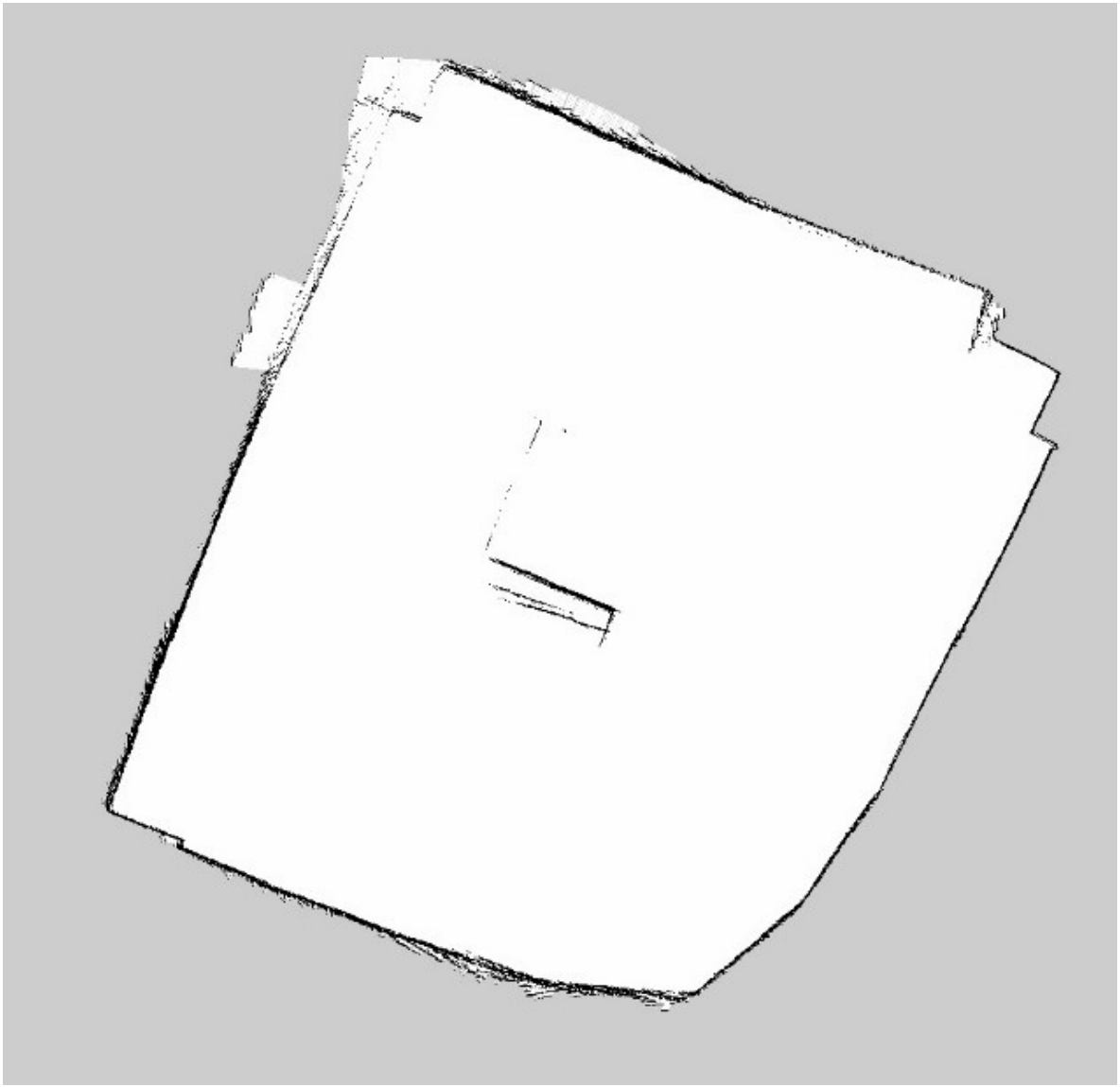


Figure 6.4 Map Generated for Experimental Setup at Control Lab, New Electrical Engineering Department

7 IMPLEMENTATION OF ALGORITHMS ON ROS

7.1 ROS Introduction

ROS is an open-source robot operating system. It is a set of software libraries and tools that help you build robot applications that work across a wide variety of robotic platforms. ROS has two "sides"

- The operating system side, which provides standard operating system services such as:
  - hardware abstraction
  - low-level device control
  - implementation of commonly used functionality
  - message-passing between processes
  - package management
- A suite of user contributed packages that implement common robot functionality such as SLAM, planning, perception, vision, manipulation, etc.

7.2 ROS Core Concepts

- Nodes
- Messages and Topics
- Services
- ROS Master
- Parameters
- Stacks and packages

7.2.1 ROS Nodes

Base unit in ROS is called a node. Nodes are in charge of handling devices or computing algorithms- each node for separate task. Nodes can communicate with each other using topics or services. ROS software is distributed in packages. Single package is usually developed for performing one type of task and can contain one or multiple nodes. Nodes can publish or subscribe to topic. Nodes can also provide or use a service.

- **rosnode**

Rosnode is a command line application for examining which nodes are registered in the system and also checking their statuses.

Using the application looks as follows:

```
rosnode command [node_name]
```

Command could be:

- list - display list of running nodes
- info - display info regarding selected node
- kill - stop selected node

7.2.2 ROS Topics

In ROS, topic is a data stream used to exchange information between nodes. They are used to send frequent messages of one type. This could be a sensor readout or motor goal speed. Each topic is registered under the unique name and with defined message type. Nodes can connect with it to publish messages or subscribe to them. For a given topic, one node cannot publish and subscribe to it at the same time, but there is no restrictions in the number of different nodes publishing or subscribing.

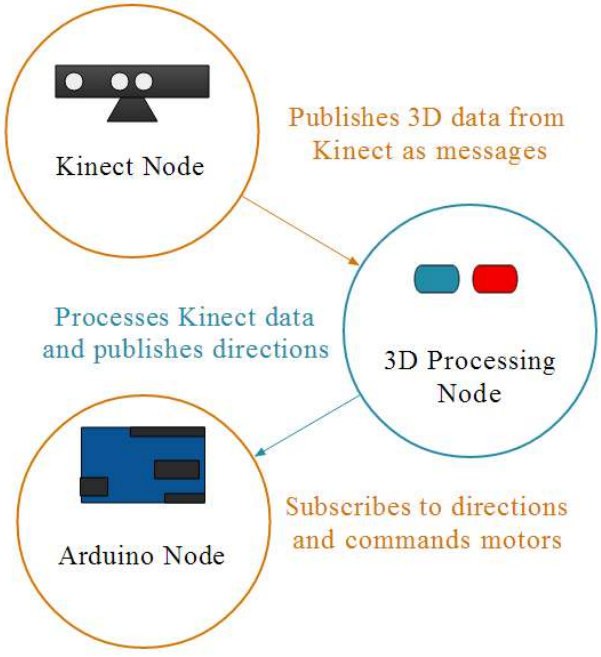


Figure 7.1: Examples of Different ROS Topics

- **rostopic**

Rostopic is a command line application for examining which topics are already being published and subscribed, checking details of the selected topic or reading messages being sent in it.

Using the application looks as follows:

```
rostopic command [topic_name]
```

Command could be:

- list - display list of topics
- info - display info regarding selected topic
- echo - display messages published in the topic

7.2.3 ROS Messages

It is strictly-typed data structures for inter-node communication. For eg., geometry_msgs/Twist is used to express velocity commands:

Vector3 linear
Vector3 angular

Vector3 is another message type composed of:

float64 x
float64 y
float64 z

7.2.4 ROS Master

It provides connection information to nodes so that they can transmit messages to each other. Every node connects to a master at startup to register details of the message streams they publish, and the streams to which that they to subscribe. When a new node appears, the master provides it with the information that it needs to form a direct peer-to-peer connection with other nodes publishing and subscribing to the same message topics.

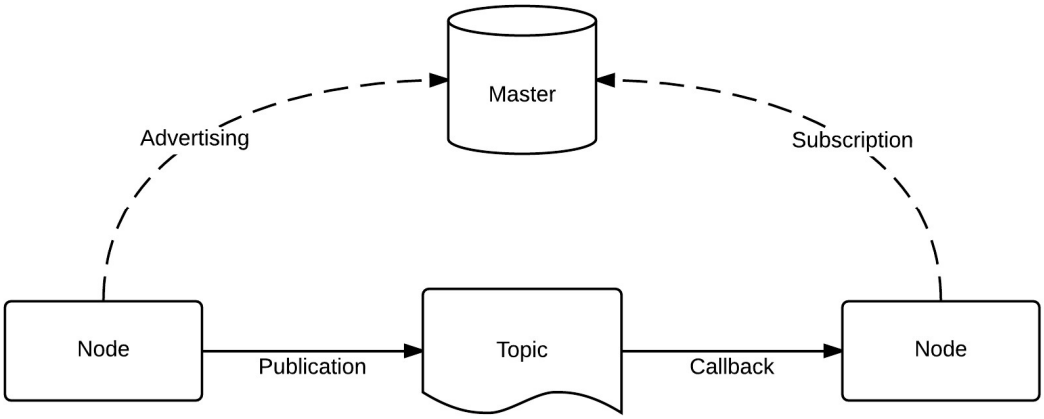


Figure 7.2: ROS Master Node Topic

7.2.5 ROS Packages

Software in ROS is organized in packages. A package might contain ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module. The goal of these packages it to provide this useful functionality in an easy-to-consume manner so that software can be easily reused. In general, ROS packages follow a "Goldilocks" principle: enough functionality to be useful, but not too much that the package is heavyweight and difficult to use from other software.

Packages are easy to create by hand or with tools like *catkin_create_pkg*. A ROS package is simply a directory descended from *ROS_PACKAGE_PATH* that has a *package.xml* file in it. Packages are the most atomic unit of build and the unit of release. This means that a package is the smallest individual thing you can build in ROS and it is the way software is bundled for release respectively.

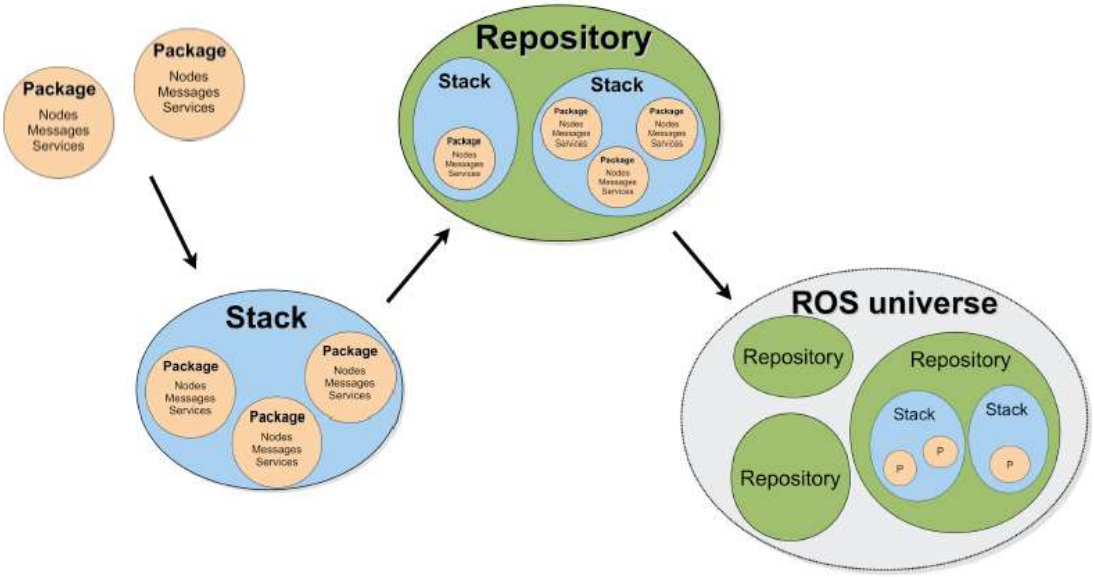


Figure 7.3: ROS Package System

```
ros_workspace
├── build
│   └── ...
├── devel
│   └── ...
└── src
    ├── CMakeLists.txt
    └── tutorial_pkg
        ├── CMakeLists.txt
        ├── include
        │   └── tutorial_pkg
        ├── package.xml
        └── src
```

Figure 7.4: Typical Package File System

The following files within the package file system needs to be modified according to use.

- CMakeLists.txt - these are build instructions for nodes, edit this file to compile any node.
- package.xml - this file contains package metadata like author, description, version or required packages. Package can be built without changing it, but adjust this file to publish the package to others.

7.3 ROS Communication Types

Type	Best used for
Topic	One-way communication, especially if there might be multiple nodes listening (e.g., streams of sensor data)
Service	Simple request/response interactions, such as asking a question about a node's current state
Action	Most request/response interactions, especially when servicing the request is not instantaneous (e.g., navigating to a goal location)

7.3.1 ROS Topics

- Topics implement a *publish/subscribe* communication mechanism, one of the more common ways to exchange data in a distributed system.
- Before nodes start to transmit data over topics, they must first announce, or *advertise,* both the topic name and the types of messages that are going to send.
- Then they can start to send, or *publish,* the actual data on the topic.
- Nodes that want to receive messages on a topic can *subscribe* to that topic by making a request to *roscore*.
- After subscribing, all messages on the topic are delivered to the node that made the request.
- In ROS, all messages on the same topic must be of the same data type.
- Topic names often describe the messages that are sent over them.

7.3.1.1 Topic Publisher

It can manage an advertisement on a specific topic. It is created by calling *NodeHandle::advertise()*

It can register this topic in the master node.

- Example for creating a publisher:

```
ros::Publisher chatter_pub = node.advertise<std_msgs::String>("chatter", 1000);
```

  - First parameter is the topic name
  - Second parameter is the queue size
- Once all the publishers for a given topic go out of scope the topic will be unadvertised. To publish a message on a topic call ***publish()***.

7.3.1.2 Talker and Listener

We'll have a package with two nodes:

- *talker* publishes messages to topic "chatter"
- *listener* reads the messages from the topic and prints them out to the screen

7.3.1.3 Subscribing to a Topic

To start listening to a topic, call the method *subscribe()* of the node handle. This returns a Subscriber object that you must hold on to until you want to unsubscribe.
Example for creating a subscriber:

ros::Subscriber sub = node.subscribe("chatter", 1000, messageCallback);

- 1st parameter is the topic name
- 2nd parameter is the queue size
- 3rd parameter is the function to handle the message

7.3.1.4 rqt_graph

rqt_graph creates a dynamic graph of what's going on in the system. Use the following command to run it:

$ rosrun rqt_graph rqt_graph

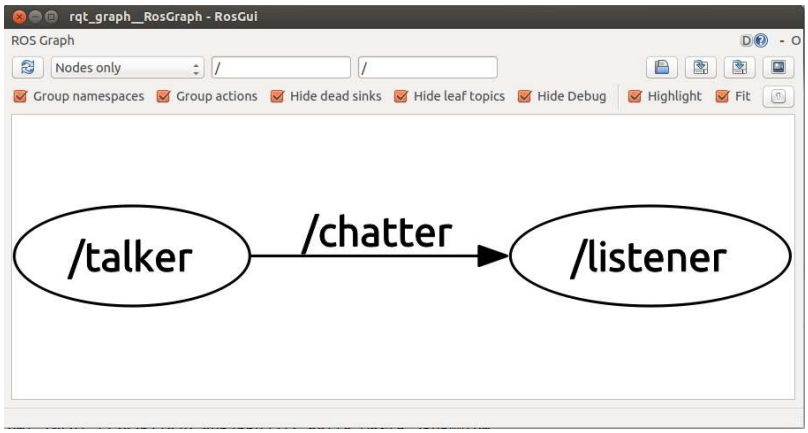


Figure 7.5: Simple ROS Node Graph

7.4 Roslaunch

It is a tool for easily launching multiple ROS nodes as well as setting parameters on the Parameter Server. Roslaunch operates on launch files which are XML files that specify a collection of nodes to launch along with their parameters

By convention these files have a suffix *.launch*

- Syntax:

$ roslaunch PACKAGE LAUNCH_FILE

- roslaunch automatically runs roscore.

7.5 Introducing catkin

Catkin is the official build system of ROS. Before catkin, ROS used the rosbuild system to build packages. Its replacement is catkin on the latest ROS version. Catkin combines CMake macros and Python scripts to provide the same CMake normal workflow. Catkin provides a better distribution of packages, better cross-compilation, and better portability than the rosbuild system.

Catkin workspace is a folder where you can modify, build, and install catkin packages. Let's check how to create an ROS catkin workspace. The following command will create a parent directory called *catkin_ws* and a subfolder called *src*:

$ mkdir -p ~/catkin_ws/src

Switch directory to the src folder using the following command. We will create our packages in the src folder:

$ cd ~/catkin_ws/src

Initialize the catkin workspace using the following command:

$catkin_init_workspace

After you initialize the catkin workspace, you can simply build the package (even if there is no source file) using the following command:

$ cd ~/catkin_ws/

$catkin_make

The catkin_make command is used to build packages inside the src directory. After building the packages, we will see a build and devel folder in catkin_ws. The

executables are stored in the build folder and in the devel folder, there are shell script files to add to the workspace on the ROS environment.[11]

7.6 Kinect

Kinect is connected to the TurtleBot netbook through a USB 2.0 port (USB 3.0 for Kinect v2). Software development on Kinect can be done using the Kinect Software Develoment Kit (SDK), freenect, and OpenSource Computer Vision (OpenCV). The Kinect SDK was created by Microsoft to develop Kinect apps, but unfortunately, it only runs on Windows. OpenCV is an open source library of hundreds of computer vision algorithms and provides support for mostly 2D image processing. 3D depth sensors, such as the Kinect, ASUS, and PrimeSense are supported in the VideoCapture class of OpenCV. Freenect packages and libraries are open source ROS software that provides support for Microsoft Kinect. More details on freenect will be provided in an upcoming section titled Configuring TurtleBot and installing 3D sensor software.

Using kinect, the robot will get the 3D image of its surroundings. The 3D images are converted to finer points called point cloud. The point cloud data will have all 3D parameters of the surrounding.

The main use of kinect on the robot is to mock the functionality of a laser scanner. The laser scanner data is essential for an algorithm called SLAM, used for building a map of the environment. The laser scanner is a very costly device, so instead of buying an expensive laser scanner, we can convert a kinect into a virtual laser scanner.

To get the Kinect working and sending data through ROS using the libfreenect library, follow the following steps:

- Install the libfreenect stack:

  sudo apt install ros-kinetic-freenect-stack

- Connect the Kinect to your computer and test the Kinect stack by running the convenient launch file (depth registration refers to the pairing of colour data with depth points):

  roslaunch freenect_launch freenect.launch depth_registration:=true

- Open up rviz in a separate terminal window (press ctrl + shift + T to open a new terminal tab):

rosrun rviz rviz

- In the rviz options panel on the left, change Global Options>Fixed Frame to *camera_link*
- In rviz add a new PointCloud2. Set its topic to */camera/depth_registered/points.*
- Wait a few seconds for the point cloud to show up. You may need to rotate the viewport around to see it.
- Play around with some of the different topics being sent out by the Kinect.[9,12]

7.7 Robot Frames and Their Transformation

The electronics data acquisition sensors are placed at different appropriate spots on robot body. Acquired environment data has to be processed with respect to robot centre. The effective way to preform the task is to assign frames to all the sensors and map and determine translation parameters between different frames. Number of available translations between different frames is sufficient enough to obtain translation from any to frame to desire frame.

A robotic system typically has many 3D coordinate frames that change over time, such as a world frame, base frame, gripper frame, head frame, etc. The effective way of tackle the problem the **tf** package of **ROS**. tf keeps track of all these frames over time. tf is a package that lets the user keep track of multiple coordinate frames over time. tf maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors, etc between any two coordinate frames at any desired point in time. tf core library is C++ class. A second class provides ROS interface and instantiates the core library. [5]

7.8 Map Server

map_server allows you to load and save maps.

To install the package:

```
$ sudo apt-get install ros-kinetic-map-server
```

To save dynamically generated maps to a file:

```
$ rosrun map_server map_saver [-f mapname]
```

map_saver generates the following files in the current directory:

map.pgm – the map itself

map.yaml – the map's metadata

7.9 GMapping ROS Package

This package contains a ROS wrapper for OpenSlam's Gmapping i.e. Rao-Blackwellized particle filters. The gmapping package provides laser-based SLAM, as a ROS node called slam_gmapping. Using slam_gmapping, you can create a 2-D occupancy grid map from laser and pose data collected by a mobile robot.

7.9.1 Subscribed Topics

`tf` (tf/tfMessage)

- Transforms necessary to relate frames for laser, base, and odometry (see below)

`scan` (sensor_msgs/LaserScan)

- Laser scans to create the map from

7.9.2 Published Topics

`map_metadata` (nav_msgs/MapMetaData)

- Get the map data from this topic, which is latched, and updated periodically.

`map` (nav_msgs/OccupancyGrid)

- Get the map data from this topic, which is latched, and updated periodically

`~entropy` (std_msgs/Float64)

- Estimate of the entropy of the distribution over the robot's pose (a higher value indicates greater uncertainty).

7.10 AMCL ROS Package

AMCL is a probabilistic localization system for a robot moving in 2D as described earlier in algorithm section. This ROS package provide implementation of the adaptive

(or KLD-sampling) Monte Carlo localization approach (as described by Dieter Fox), which uses a particle filter to track the pose of a robot against a known map.

7.10.1 Subscribed Topics

`scan` (sensor_msgs/LaserScan)

- Laser scans.

`tf` (tf/tfMessage)

- Transforms.

`initialpose` (geometry_msgs/PoseWithCovarianceStamped)

- Mean and covariance with which to (re-)initialize the particle filter.

`map` (nav_msgs/OccupancyGrid)

- When the `use_map_topic` parameter is set, AMCL subscribes to this topic to retrieve the map used for laser-based localization.

7.10.2 Published Topics

`amcl_pose` (geometry_msgs/PoseWithCovarianceStamped)

- Robot's estimated pose in the map, with covariance.

`particlecloud` (geometry_msgs/PoseArray)

- The set of pose estimates being maintained by the filter.

`tf` (tf/tfMessage)

- Publishes the transform from `odom` (which can be remapped via the ~odom_frame_id parameter) to `map`.[13]

7.11 Teleoperation

The teleoperation can be run simultaneously with the navigation stack. It will override the autonomous behaviour if commands are being sent. It is often a good idea to teleoperate the robot after seeding the localization to make sure it converges to a good estimate of the position.

7.12 ROS Navigation Stack

The goal of the navigation stack is to move a robot from one position to another position safely (without crashing or getting lost).

It takes in information from the odometry and sensors, and a goal pose and outputs safe velocity commands that are sent to the robot.

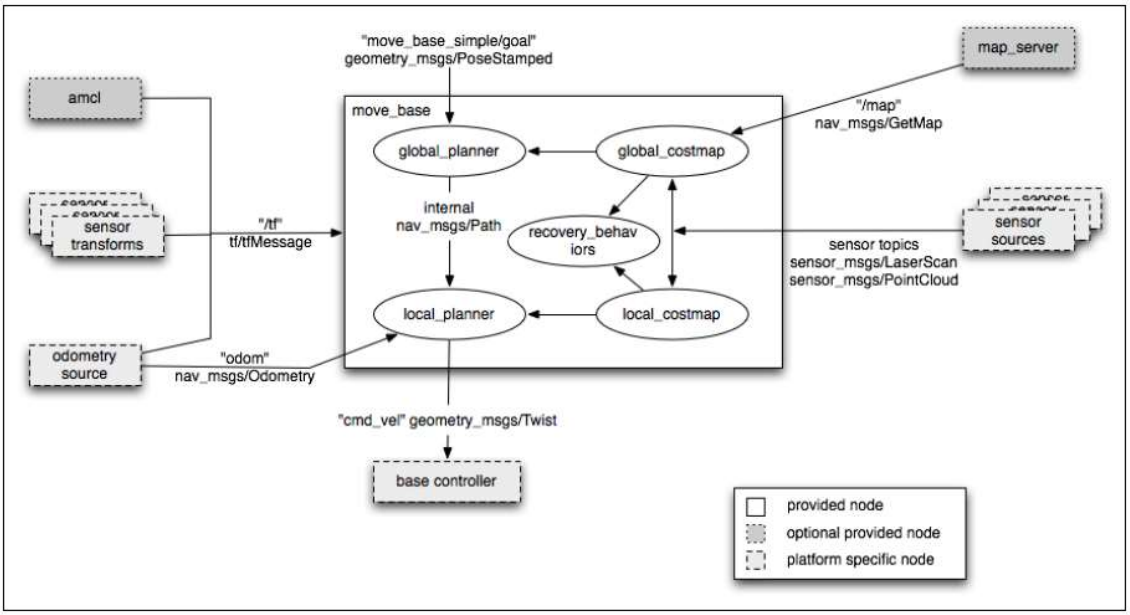


Figure 7.6: Navigation Stack Setup

7.12.1 Navigation Stack Main Components

Package/Component	Description
map_server	offers map data as a ROS Service
gmapping	provides laser-based SLAM
amcl	a probabilistic localization system
global_planner	implementation of a fast global planner for navigation
local_planner	implementations of the Trajectory Rollout and Dynamic Window approaches to local robot navigation
move_base	links together the global and local planner to accomplish the navigation task

7.12.2 Navigation Stack Requirements

Three main hardware requirements:

- The navigation stack can only handle a differential drive and holonomic wheeled robots. It can also do certain things with biped robots, such as localization, as long as the robot does not move sideways.
- A planar laser must be mounted on the mobile base of the robot to create the map and localization. Alternatively, you can generate something equivalent to laser scans from other sensors (Kinect for example).
- Its performance will be best on robots that are nearly square or circular.

7.13 Navigation Planners

Our robot will move through the map using two types of navigation—global and local. The global planner is used to create paths for a goal in the map or a far-off distance. The local planner is used to create paths in the nearby distances and avoid obstacles.

7.13.1 Global Planner

NavFn provides a fast-interpolated navigation function that creates plans for a mobile base. The global plan is computed before the robot starts moving toward the next destination. The planner operates on a costmap to find a minimum cost plan from a start point to an end point in a grid, using Dijkstra's algorithm. The global planner generates a series of waypoints for the local planner to follow.

7.13.2 Local Planner

It chooses appropriate velocity commands for the robot to traverse the current segment of the global path. It combines sensory and odometry data with both global and local cost maps. It can recompute the robot's path on the fly to keep the robot from striking objects yet still allowing it to reach its destination. It implements the Trajectory Rollout and Dynamic Window algorithm.

7.13.2.1 Trajectory Rollout Algorithm

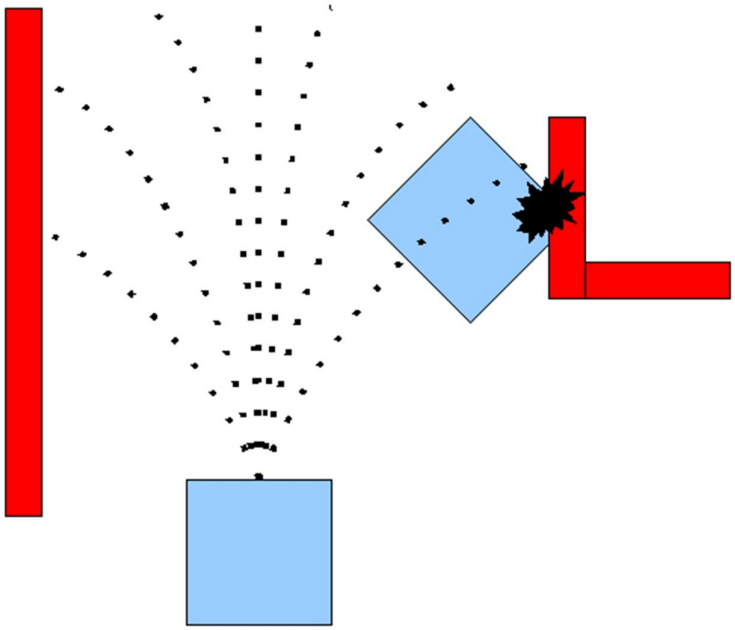


Figure 7.7: Trajectory Rollout Algorithm

1. Discretely sample in the robot's control space (*dx,dy,dθ*)
2. For each sampled velocity, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time
3. Evaluate each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path, and speed
4. Discard illegal trajectories (those that collide with obstacles)
5. Pick the highest-scoring trajectory and send the associated velocity to the mobile base
6. Rinse and repeat

7.13.2.2 Local Planner Parameters

The file *base_local_planner.yaml* contains a large number of ROS Parameters that can be set to customize the behavior of the base local planner.

It grouped into several categories:

- robot configuration
- goal tolerance

- forward simulation
- trajectory scoring
- oscillation prevention
- global plan

7.13.3 Costmap

A data structure that represents places that are safe for the robot to be in a grid of cells. It is based on the occupancy grid map of the environment and user specified inflation radius.

There are two types of costmaps in ROS:

- Global costmap is used for global navigation
- Local costmap is used for local navigation

Each cell in the costmap has an integer value in the range [0 (FREE_SPACE), 255 (UNKNOWN)]. It is managed by the costmap_2d package.

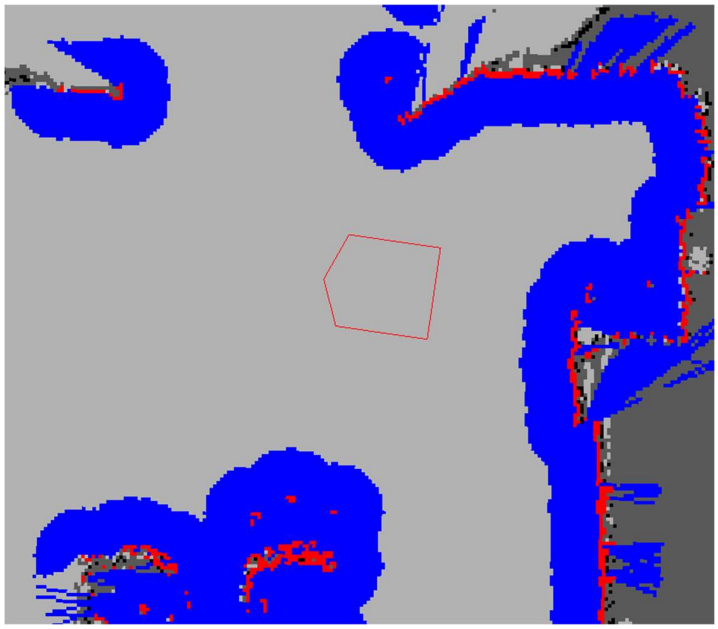


Figure 7.8: Costmap

7.13.3.1 Common parameters for cost map

Common parameters are used both by local and global cost map and defined by following parameters:

obstacle_range: 6.0
raytrace_range: 8.5
footprint: [[0.12, 0.14], [0.12, -0.14], [-0.12, -0.14], [-0.12, 0.14]]
map_topic: /map
subscribe_to_updates: true
observation_sources: laser_scan_sensor
laser_scan_sensor: {sensor_frame: laser_frame, data_type: LaserScan, topic: scan, marking: true, clearing: true}
global_frame: map
robot_base_frame: base_link
always_send_full_costmap: true

This parameter defines properties of used sensor, these are:

- sensor_frame - coordinate frame tied to sensor
- data_type - type of message published by sensor
- topic - name of topic where sensor data is published
- marking - true if sensor can be used to mark area as occupied
- clearing - true if sensor can be used to mark area as clear

7.13.3.2 Parameters for global cost map

These parameters are used only by global cost map. Parameter meaning is the same as for local cost map, but values may be different.

The file for global cost map is look like below:

global_costmap:
update_frequency: 2.5
publish_frequency: 2.5
transform_tolerance: 0.5
width: 15
height: 15
origin_x: -7.5
origin_y: -7.5
static_map: true
rolling_window: true

inflation_radius: 2.5
resolution: 0.1

7.13.3.3 Parameters for Trajectory Planner

These parameters define how far from destination it can be considered as reached. Linear tolerance is in meters, angular tolerance is in radians.

Your final file should look like below:

TrajectoryPlannerROS:
max_vel_x: 0.2
min_vel_x: 0.1
max_vel_theta: 0.35
min_vel_theta: -0.35
min_in_place_vel_theta: 0.25

acc_lim_theta: 0.25
acc_lim_x: 2.5
acc_lim_Y: 2.5

holonomic_robot: false

meter_scoring: true

xy_goal_tolerance: 0.15
yaw_goal_tolerance: 0.25

7.14 Map Updates

The costmap performs map update cycles at the rate specified by the update_frequency parameter. In each cycle:

- Sensor data comes in.
- Marking and clearing operations are perfomed in the underlying occupancy structure of the costmap.
- This structure is projected into the costmap where the appropriate cost values are assigned as described above.
- Obstacle inflation is performed on each cell with a LETHAL_OBSTACLE value.

7.15 Navigation Configuration Files

Configuration File	Description
global_planner_params.yaml	global planner configuration
navfn_global_planner_params.yaml	navfn configuration
dwa_local_planner_params.yaml	local planner configuration
costmap_common_params.yaml global_costmap_params.yaml local_costmap_params.yaml	costmap configuration files
move_base_params.yaml	move base configuration
amcl.launch.xml	amcl configuration

Autonomous Navigation of a Known Map

Amcl_demo.launch – launch file for navigation demo.[12]

7.16 rviz with Navigation Stack

rviz allows you to:

- Provide an approximate location of the robot (when starting up, the robot doesn't know where it is).
- Send goals to the navigation stack.
- Display all the visualization information relevant to the navigation (planned path, costmap, etc.).

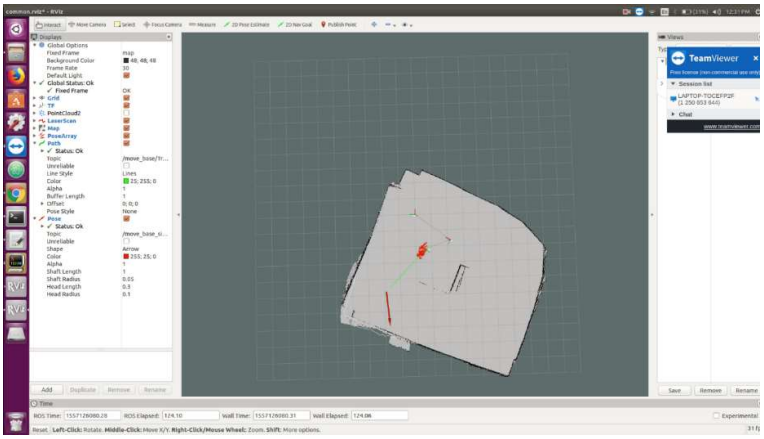


Figure 7.9 Sample rviz setup

7.17 Node structure while SLAM

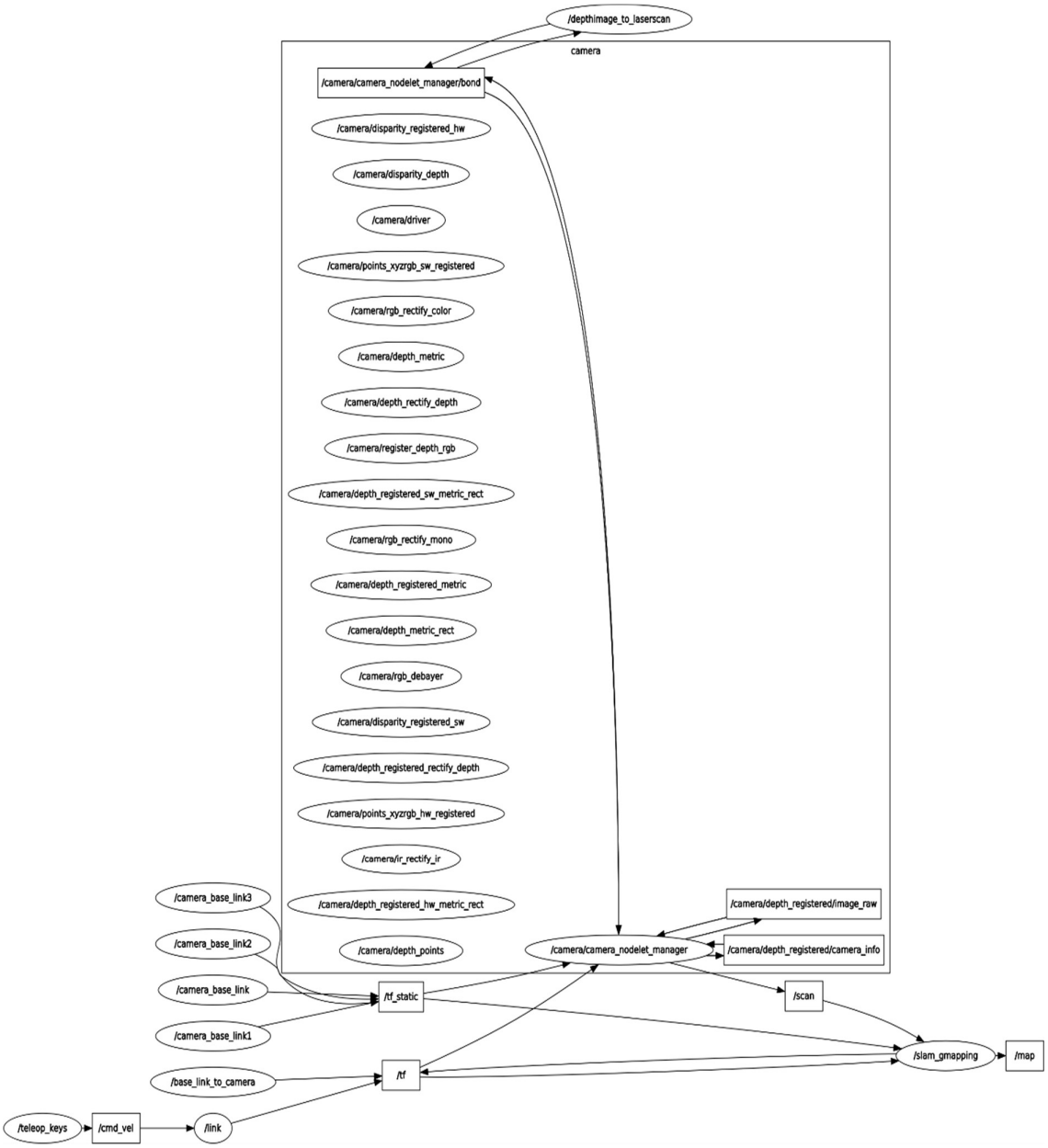


Figure 7.10 Node structure while SLAM

7.18 Node structure while goal tracking

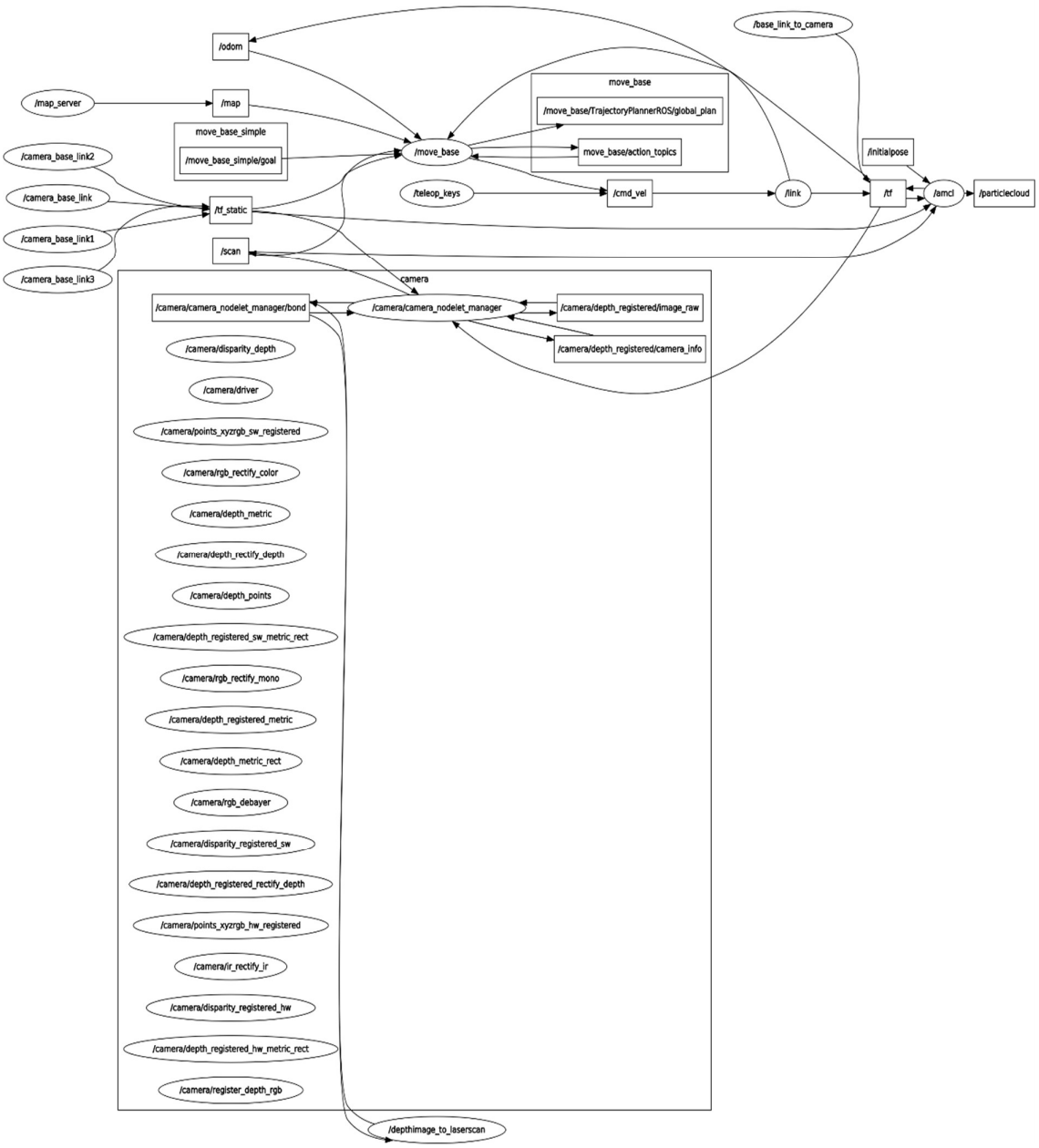


Figure 7.11 Node structure while goal tracking

8 CONCLUSION

Prior discussed probabilistic robotics algorithms can be developed, implemented and tested on ROS with high efficiency and greater simplicity. Because of the open-source nature of ROS, all the algorithms can be easily implemented by suitable ROS packages. All the supporting packages like *tf, rviz, & rqt,*etc. are useful for the accurate simulation and realization of discussed algorithms.

The hardware design used is the most suitable and cheapest models from the ones available. The DC brushed motor gives linear speed control with relatively less additional component requirement. The Kinect Sensor is the most appropriate alternative for the expensive depth sensing devices like LIDAR. It has a simple interface along with low power requirement. The MAVLink protocol is universally accepted for node-to-node communication as it has least computational cost and relatively simple cross-platform implementation.

All three algorithms, viz. gmapping, AMCL and A*-Path Planning, chosen are found to be suitable for autonomous navigation of indoor environment along with dynamic obstacle avoidance and is widely being used in modern day robotics.

REFERENCES

[1] Gallager, Robert G. *Stochastic processes: theory for applications*. Cambridge University Press, 2013.

[2] Thrun, Sebastian, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*. MIT press, 2005.

[3] Grisetti, G., Stachniss, C. and Burgard, W., 2007. Improved techniques for grid mapping with Rao-Blackwellized particle filters. *IEEE transactions on Robotics*, *23*(1), p.34.

[4] Grisetti, G., Stachniss, C. and Burgard, W. (2007, January). OpenSLAM. Retrieved from https://openslam-org.github.io/gmapping.html

[5] Foote, T., 2013, April. tf: The transform library. In *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)* (pp. 1-6). IEEE.

[6] Nagrath, I.J. and Gopal, M., 2008. *Textbook of control systems engineering (Vtu)*. New Age International.

[7] Åström, K.J., Hägglund, T. and Astrom, K.J., 2006. *Advanced PID control* (Vol. 461). Research Triangle Park, NC: ISA-The Instrumentation, Systems, and Automation Society.

[8] Atoev, S., Kwon, K.R., Lee, S.H. and Moon, K.S., 2017, November. Data analysis of the MAVLink communication protocol. In *2017 International Conference on Information Science and Communications Technologies (ICISCT)* (pp. 1-3). IEEE.

[9] Zainuddin, N.A., Mustafah, Y.M., Shawgi, Y.A.M. and Rashid, N.K.A.M., 2014, September. Autonomous navigation of mobile robot using Kinect sensor. In *2014 International Conference on Computer and Communication Engineering* (pp. 28-31). IEEE.

[10] Stentz, A., 1997. Optimal and efficient path planning for partially known environments. In *Intelligent Unmanned Ground Vehicles* (pp. 203-220). Springer, Boston, MA.

[11] Wiki.ros.org. (2019). *Documentation - ROS Wiki*. [online] Available at: http://wiki.ros.org/Documentation [Accessed 7 May 2019].

[12] Joseph, L., 2015. *Mastering ROS for robotics programming*. Packt Publishing Ltd.

[13] Fairchild, C. and Harman, T.L., 2016. *ROS Robotics By Example*. Packt Publishing Ltd.