Cairo University

Faculty of Engineering

Aerospace Department

Mars Rover

Project Supervisions

Prof. Gamal M. El-Bayoumi

Prof. Ayman H. Kassem

August 2020

Team members

Karim Ahmed Kamal El-din

Anas Abd-Elmenam Farrag

Mohamed Ismail Ibrahim

Acknowledgment

First and foremost, we would like to thank God Almighty for giving us the strength, knowledge, ability and opportunity to undertake this project and to persevere and complete it satisfactorily. Without his blessings, this achievement would not have been possible.

In our journey in this project, we have found a family member, heartfelt support and guidance at all times, so special thanks to **Prof. Gamal El-Bayoumi** for his invaluable guidance, motivation and suggestions to solve our problems. We also feel grateful for **Prof. Ayman H. Kassem** for ensuring that the fire keeps burning and for assisting us in collation of data for our project despite his managerial role in leading our department.

We take pride in acknowledging the insightful guidance of **Eng. Mohamed Samir** for sparing his valuable time whenever we approached him and showing us the way ahead.

We have a great pleasure to express our gratitude to **Dr. Diaa El-Haq** who share with us the financial responsibility.

Our acknowledgement would be incomplete without thanking our **Professors and Teaching Assistants** who teach us in our college journey. We are one step away from beginning a new chapter. So, it's the time to send them a special thank for their effort and time.

Our Families who no word can describe our feelings toward them. Their sacrifices, endless love and support are the main reason to become here. So, we hope the happiness and satisfaction for them.

MARS ROVER PROJECT TEAM

Abstract

A mars rover model equipped with autonomous model that allow rover to navigate without human interactions.

An autonomous exploration operates in unknown environment fusing Visual Odometry data from: Kinect camera, LIDAR, and Wheel encoders, by extended Kalman filter or Particle filter to generate more accurate Odometry. This enable the rover to navigate without human interface and create a map using SLAM algorithm.

Autonomous algorithms used to optimize rover navigation and exploration missions supported with maps and cost maps generated from SLAM algorithm to use in A* motion planning algorithm.

Full simulation model applied on ROS simulation tools RVIZ and Gazebo to help testing and optimizing all navigation algorithms SLAM and motion planning

Contents

List of figures

Chapter 1

1. Introduction

1.1 Introduction

Thousands of years ago, human was concerned into the space and tried to discover it. The only way for them was observations and their fiction. Many theories were deduced only by observations, also without calculations.

After the great progress in vast fields of science; like: Mathematics, Physics and many other fields; scientists start to understand how our Planet and the other planets moves in specific orbits and how large is the Universe which contains massive number of Galaxies, they needed to take a closer look to that outer space, and thinking about how to get benefits from that large Universe.

Nowadays, after the massive progress in technology, scientists need to study other planets that lies in our solar-system, but is it safe to send astronauts to the nearest planet that lies 0.52 AU? (1 AU = $1.496x10^{8}$)

1.2 Why do we use robots?

To reduce the cost and risk for human exploration of Mars, robots are sent in exploration missions. We can send robots to explore space without having to worry so much about their safety. Of course, we want these carefully built robots to last. We need them to stick around long enough to investigate and send us information about their destinations. But even if a robotic mission fails, the humans involved with the mission stay safe.

Sending a robot to space is also much cheaper than sending a human. Robots don't need to eat or sleep or go to the bathroom. They can survive in space for many years and can be left out there—no need for a return trip.

Plus, robots can do lots of things that humans can't. Some can withstand harsh conditions, like extreme temperatures or high levels of radiation. Robots can also be built to do things that would be too risky or impossible for astronauts.

1.3 Why do we need autonomous vehicles?

The large distance between the Planets makes the communication and data transfer take long time. Also, the power problem, as we need to save power as much as possible for its mission.

The communication between Earth ground station and the robot on Mars is not continuous. It depends on specific communication sessions which are scheduled as that depends on communication satellites position. At these sessions, the rover starts sending its data packages and receive new mission to do.

Mars Rover has the ability to its mission safely, as it can build its own map for the environment around it, avoid any dangerous collisions so that the robot could last as much time as possible fully functional.

1.4 Objectives of Mars Rover

The main purpose of sending robots to Mars is to send human safely to it in the near future. Also, searching for water is the most thing that the scientists need in such exploration missions. Searching for and characterize a variety of rocks and soils that hold clues to past water activity.

We can say that the objectives of the exploration missions are:

- Determine whether life ever arose on Mars.
- Characterize the climate of Mars.
- Characterize the geology of Mars.
- Prepare for human exploration.

But how does Mars Rover move to do its missions safely?

Moving safely from rock to rock or location to location is a major challenge because of the communication time delay between Earth and Mars, which is about 20 minutes on average. Unlike a remote-controlled car, the drivers of rovers on Mars cannot instantly see what is happening to a rover at any given moment and they cannot send quick commands to prevent the rover from running into a rock or falling off of a cliff.

1.4.1 Navigation

Navigation is the most important task for mobile robots. We want to know where we are, and we need to be able to make a plan for how to reach a goal destination.

Of course, these two problems are not isolated from each other, but rather closely linked. If a robot does not know its exact position at the start of a planned trajectory, it will encounter problems in reaching the destination.

1.4.1.1 Mapping and Localization

One of the central problems for driving robots is localization. For many application scenarios, we need to know a robot's position and orientation at all times.

It's more practical to use sensors on the robot and using it to build a map and localize itself in that location. The rover uses sensors such as: LIDAR and stereo camera to build a map and update it with constant frequency, to know its place with respect to the built map. Building maps and representing obstacles would be discussed in the next chapters.

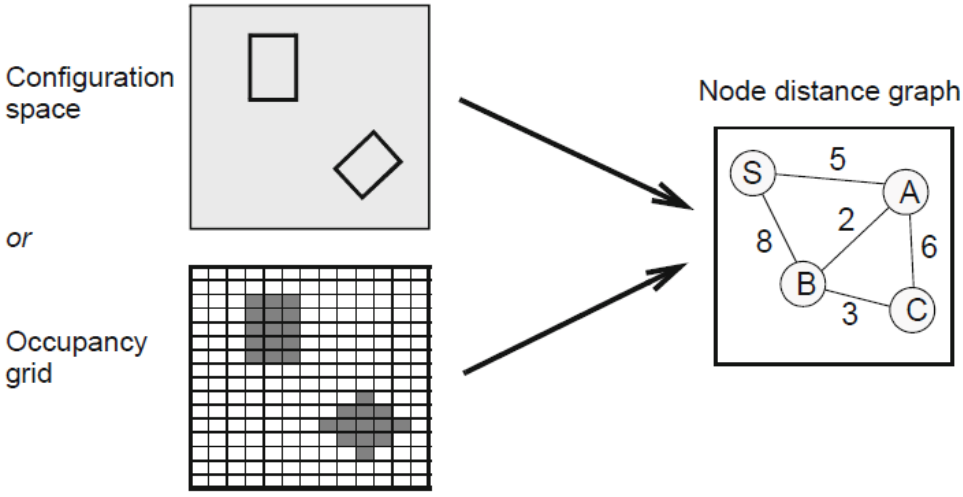


Figure 1.1 basic environment representation

1.4.1.2 Path planning

Path-planning is an important primitive for autonomous mobile robots that lets robots find the shortest – or otherwise optimal – path between two points. Path-planning requires a map of the environment and the robot to be aware of its location with respect to the map.

In the next chapters Path planning algorithms would be discussed in detail.

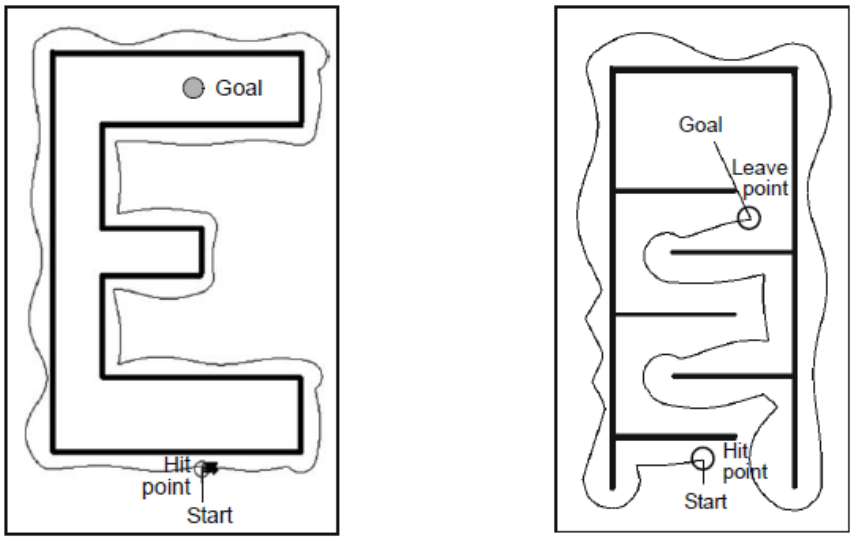


Figure 1.2 Path planning examples

1.4.2 Robotic arm

The Robotic Arm holds and maneuvers the instruments that help scientists get up-close and personal with Martian rocks and soil.

Much like a human arm, the robotic arm has flexibility through three joints: the rover's shoulder, elbow, and wrist. The arm enables a tool belt of scientists' instruments to extend, bend, and angle precisely against a rock to work as a human geologist would: grinding away layers, taking microscopic images, and analyzing the elemental composition of the rocks and soil.

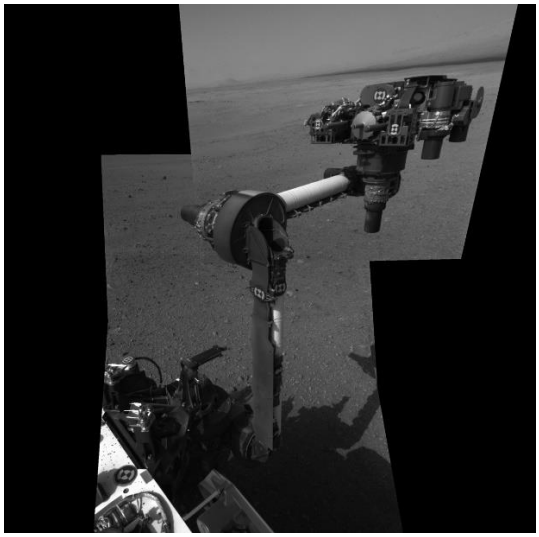


Figure 1.1.3 NASA Mars Rover Robotic arm

The Robotic arm has an important role, as it is used in drilling into different rocks to analyze them, which considered from the most important missions of the robot.

1.5 ROS (Robot Operating System)

The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.

Can we say that ROS is mature enough for practical applications?

In reality, ROS is that plumbing, a rich and mature set of tools, a wide-ranging set of robot-agnostic capabilities provided by packages, and a greater ecosystem of additions to ROS.

We can make sure that ROS is mature enough, since ROS was used on Robonaut 2 (R2) aboard the International Space Station (ISS). NASA uses ROS in many prototypes to get benefits from that platform.

Figure 1.1.4 R2 Robot

ROS processes are represented as nodes in a graph structure, connected by edges called topics. ROS nodes can pass messages to one another through topics, make service calls to other nodes, provide a service for other nodes, or set or retrieve shared data from a communal database called the parameter server. A process called the ROS Master makes all of this possible by registering nodes to itself, setting up node-to-node communication for topics, and controlling parameter server updates. Messages and service calls do not pass through the master, rather the master sets up peer-to-peer communication between all node processes after they register themselves with the master. This decentralized architecture lends itself well to robots, which often consist of a subset of networked computer hardware, and may communicate with off-board computers for heavy computation or commands.

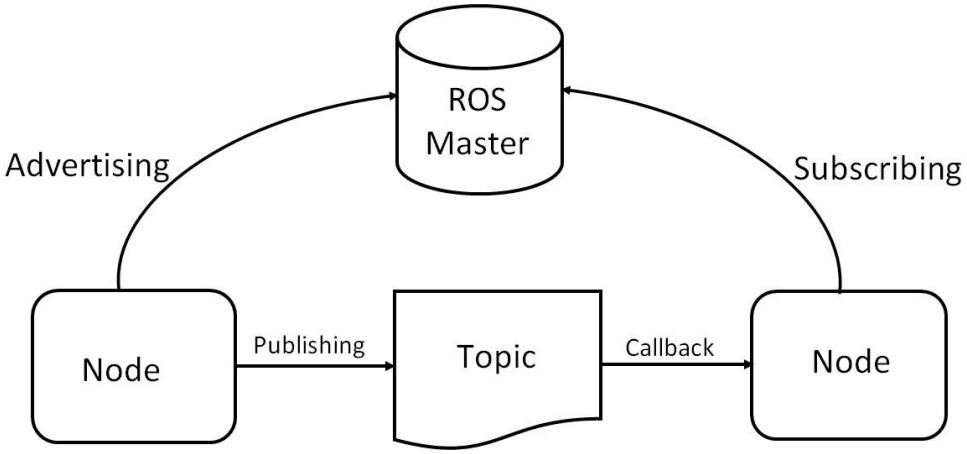


Figure 1.5 ROS

Our project mainly depends on ROS. In the next Chapters, ROS would be discussed in details from scratch, then our ROS codes implementations would be discussed clearly.

1.6 Project Objectives

In our project, we focused on:

- Mechanical design.
- Mapping and localization.
- Path Planning and navigation.
- Motors control.

We tried to discuss every single step in that project as a tutorial, to make that project useful for anyone to get start with autonomous navigation.

Our recommendation for future work, is to work on robotic arm Project, to get a complete project of Mars Rover.

Chapter 2

2. Mechanical system and kinematics

2.1 Introduction

According to the International Standards Organization's definition of an industrial Robot, mobile robot can be defined as "A mobile robot is an autonomous system capable of traversing a terrain with natural or artificial obstacles. Its chassis is equipped with wheels/tacks or legs and possibly a manipulator setup mounted on the chassis for handling of work pieces, tools or special devices. Various preplanned operations are executed based on a preprogrammed navigation strategy taking into account the current status of the environment."

Mobile robots can be classified by significant properties as:

- Locomotion (Legged, wheeled, limbless, etc.)
- Suspension (Rocker-bogie, independent, soft, etc.)
- Steering (Skid, Ackerman, explicit)
- Usage Area (Rough Terrain, even surface, etc.)

Our Rover uses a differential mechanism to maintain a relatively even weight distribution on all wheels when driving over uneven terrain. The Rover main body houses the electronics and batteries and, serves as a mounting place for the sensor suite. The Rocker bogie suspension system lifts the body above the wheels to maximize ground clearance improving mobility over obstacles. These structures are designed to be as compact as possible to conserve space while protecting the internal components against foreign debris and collisions with obstacles

2.2 Objective of mechanical Design

The primary design goal for the rover is simplification. The mobility system used to get the rover to its destination must be energy efficient, light weight, and robust. The suspension system should be able to traverse uneven terrain. This is possible because the top speed of the rover will never be fast enough for the tires to leave contact with the ground while driving over bumps.

2.2.1 Design Criteria

Locomotion

Locomotion is a process, which moves a rigid body. There is no doubt that a mobile robot's most important part is its locomotion system which determines the stability and capacity while traversing on rough terrain. The difference of robotic locomotion is distinct from traditional types in that it has to be more reliable without human interaction. While constructing a robot, designer must have decided on the terrain requirements like stability criteria, obstacle height, and surface friction. There is no only one exact solution while comparing the mobility systems.

There are several types of locomotion mechanisms were designed depending on nature of the terrain. Locomotion systems can be divided into groups as; wheeled, tracked, legged (walking robots), limbless (snake and serpentine robots) and hopping robots. Wheeled rough terrain mobile robots are called as "Rover". Comparing to walking robots and snake robots. Another advantage of wheeled locomotion is navigation. Wheeled robot's position and orientation can be calculated more precisely than tracked vehicles. Opposite to wheeled locomotion, legged locomotion needs complex control algorithms for positioning.

2.3 Suspension

Wheeled locomotion's main component is its suspension mechanism which connects the wheels to the main body or platform. This connection can be in several ways like springs, elastic rods or rigid mechanisms. Most of the heavy vehicles like trucks and train wagons use leaf springs. For comfortable driving, cars use a complex spring, damping and mechanism combination. Generally, exploration robots are driven on the rough surface which consists of different sized stones and soft sand. For this reason, car suspensions are not applicable for rovers. The requirements of a rover suspension are; as simple and lightweight as possible. Connections should be without spring to maintain equal traction force on wheels, distribute load equally to each wheel for most of the orientation possibilities to prevent from slipping Soft suspension systems. With spring reduce vibrations and effects of impacts between wheel and ground. However, reaction force of pressed spring increases the force that transmits from wheel to ground. When climbing over an obstacle, higher wheel's traction force is more than the lover one which causes slippage.

2.4 Rocker and bogie suspension

The rocker-bogie suspension system is a passive springless and symmetric mechanism. Each side of the rocker-bogie has a rocker and a bogie: the rocker is connected to the rear wheel and the middle wheel and

the front wheel are connected by the bogie. The two sides of rocker-bogie are connected by the differential bar attached to the main body, which ensures that the six wheels are in contact with the ground all the time providing a stable platform for the scientific instruments and sensors as shown in fig 2.1

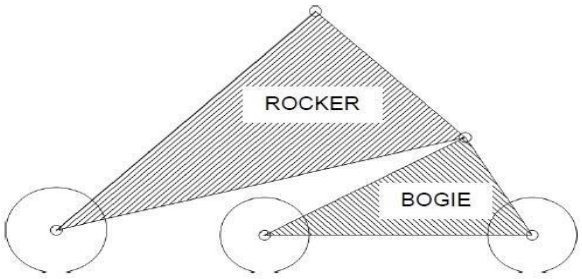


Figure 2.1 Rocker and bogie

The rocker-bogie suspension system is good at dealing with obstacles and excellent travers ability. However, the rocker-bogie based robots must move at a very low average speed to ensure the stability of traveling

The rover uses a similar "rocker-bogie" suspension system that was used on Mars Exploration Rovers. The suspension system is how the wheels are connected to the rest of the rover and control how the rover interacts with the terrain.

The suspension system has three main components:

Differential: Connects to the left and right rockers and, to the rover body by a pivot in the center of the rover's top deck.

Rocker: One each on the left and right side of the rover. Connects the front wheel to the differential and the bogie in the rear.

Bogie: Connects the middle and rear wheels to the rocker.

A rover is considered to have a high degree of mobility in natural terrain if it can surmount obstacles that are large in comparison to the size of its wheels. A rover must have enough traction from its rear wheels to push the front wheels against an obstacle with enough force so that they can climb up it. A four wheeled rover can’t climb obstacles larger than a wheel radius because the rear wheels do not have enough traction. Without traction the wheels will slip and there will not be enough forward thrust to keep the front wheels in contact with the obstacle. The rocker bogie suspension can surmount obstacles head on that are larger than a wheel diameter because it uses an extra set of wheels to provide more forward

thrust. The extra wheels also reduce the normal force on each wheel by about 1/6 the weight of the rover. Less forward thrust is required because the front wheels only have to lift.

The rocker bogie suspension is capable of a high degree of mobility. It has a ground clearance larger than a wheel diameter, unlike articulated body vehicles. The single rigid body is more stable for sensor mounting and thermal control. The suspension mechanisms and joints are above the wheels reducing the chances that
the rover will get caught on an obstacle. It can also perform multiple types of
steering as seen in fig 2.2: Crabbing, Zero Radius, Differential, Zero Radius and Ackerman.

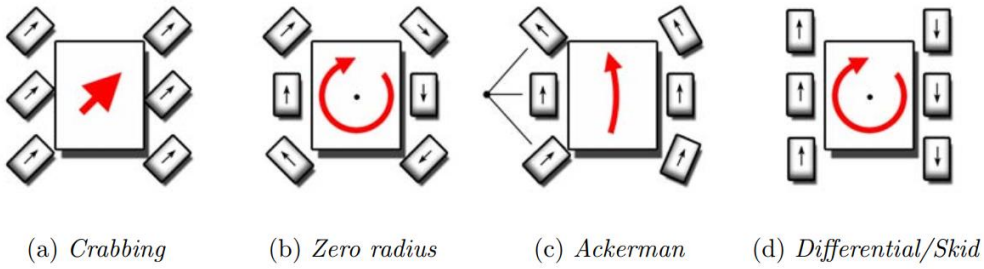


Figure 2.2 Types of steering

This mobility system requires that each wheel be driven by a separate motor and steering mechanism, increasing the overall complexity. Rovers that use the rocker bogie suspension can have 8 or 10 motors just for mobility all of which are exposed to the environment including the drive train. Harmonic drives coupled to the motors are used to increase torque rather than planetary or spur gear boxes because they save space and weight. During operation they have high static friction and can lock up in cold temperatures which will overload the motors causing them to fail prematurely. Sojourner had heating units on each motor to keep them within the operating limits in fear that the extreme cold of the Martian atmosphere might damage them

2.5 Obstacle limit

A rover's obstacle limit generally compared with robot's wheel size. in four- wheel drive off-road vehicles, limit is nearly half of their wheel diameter. It is possible to pass over more than this height by

pushing driving wheel to obstacle which can be called as *climbing*. Step or stair climbing is the maximum limit of obstacles. The contact point of wheel and obstacle is at the same height with wheel center for this condition.

Field tests show that Mars mobile robots should be able to overcome at least 1.5 times height of its wheel diameter as shown in fig 2.3 This limitation narrows the mobile robot selection alternatives and forces scientists to improve their current designs and study on new rovers.

All the researches show that most of the rover designs have a climbing capacity between 1.5 diameters and 2 diameters of wheel. To reach higher capacities, active climbing methods are required.

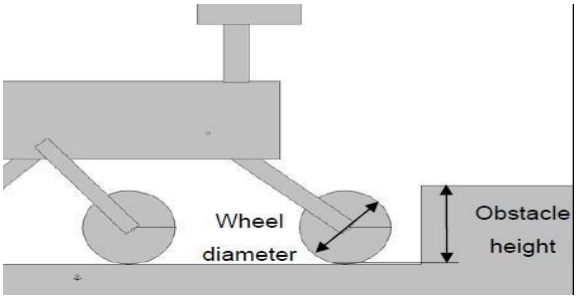


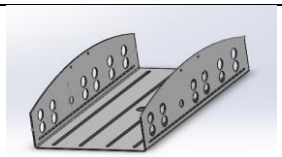
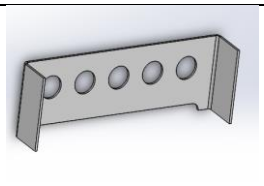
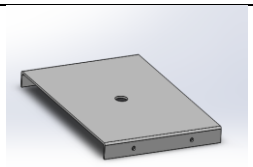
Figure 2.3 obstacle height

2.6 WALL-E

These parts were cut and bended by CNC machining.

2.6.1.1 Items Specifications

Table 2.1 Items Specifications of Wall-E Body

No.	Item	Image
1	Base of body	
2	Side of body	
3	Upper part of body	

Assembly of Body
By integrate all last parts.

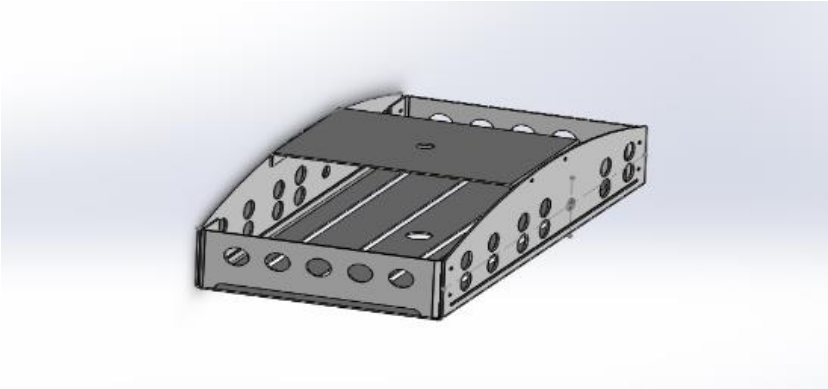


Figure 2.4 Wall-E Body

2.7 Motor Wheel Mounting

The function of motor-wheel mounting is transfer the torque output from the motors to the wheels. A compact light weight design is needed to maintain efficiency and reduce the power consumed when driving. It must be able to withstand high loads in forward and reverse directions when the rover is climbing over obstacles for many kilometers.

Wheel torque

The wheel diameter and weight of the rover are critical dimensions that affect the amount of torque required to traverse obstacles as shown in fig 2.5 The following assumes that the wheel has a mechanical grip on the obstacle using the grousers on the tire and does not slip. This is done to obtain the maximum amount of torque needed. The wheel diameter is 120mm. Only a fraction of the mass of the rover will need to be lifted by each wheel because all six wheels are always in contact with the ground. This is an advantage of having a spring less suspension. To be sure that it will have enough torque each wheel should lift one sixth of the rover's mass (2.5kg). The weight of the rover will be calculated using Earth gravity (9.81 m/s^2) since all testing will be done here.

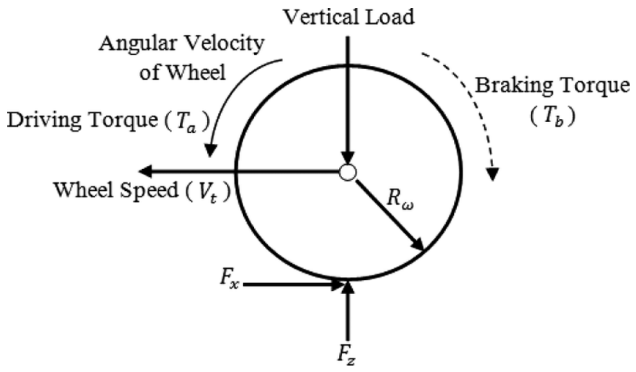


Figure 2.5 Wheel torque

$$vertical\ load = \frac{m_r * g}{6} \qquad \text{eq 1}$$

$$T_b = vertical\ load * R_w \qquad \text{eq 2}$$

Eq 1 is the amount of force on the wheel due to the mass of the rover (mr).
Calculation of torque from Eq 2 where R_W is the radius. We could choose the required motor and wheels.

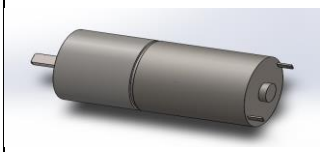
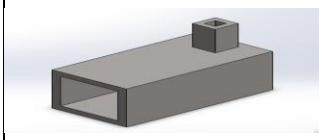
Wheel motion

While driving on a flat surface, if there is no slipping, wheel center will move on a line parallel to the surface with constant velocity. Although, obstacle geometries can be different, most difficult geometry which can be climbed by wheel is stair type rectangular obstacle.

In figure height of the obstacle is same or less than the half diameter of the wheel. For this condition, the wheel's instant center of rotation (IC1) is located at the contact point of the obstacle and wheel. Trajectory of the wheel centers' during motion generates a soft curve, thus, horizontal motion of the wheel center does not break.

2.8 Motor-Wheel Assembly

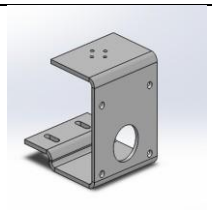
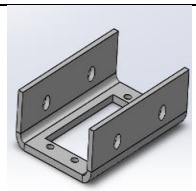
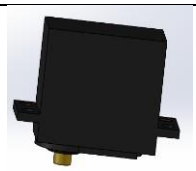
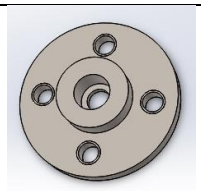
Table 2.2 Items of wheel, motor and housing

No.	Item	Dimension	Image
1	DC Motor	Diameter (0.96) in Length (2.08) in	
2	Motor housing	Inner (0.98×.98×2.5) in Outer (1.3×1.3×2.5) in	
3	Wheel	Outer diameter (120) mm	

By integrating item (1) with item (3) and then fixed them inside item (2).

Using CNC machine, we could cut sheet metal, and by bending we reached to these shapes:

Table 2.3 items of wheel, motor and housing

No.1	Item	Image
1	Motor Holder	
2	Servo holder	
3	servo	
4	Servo disc	

By integrating all last parts:

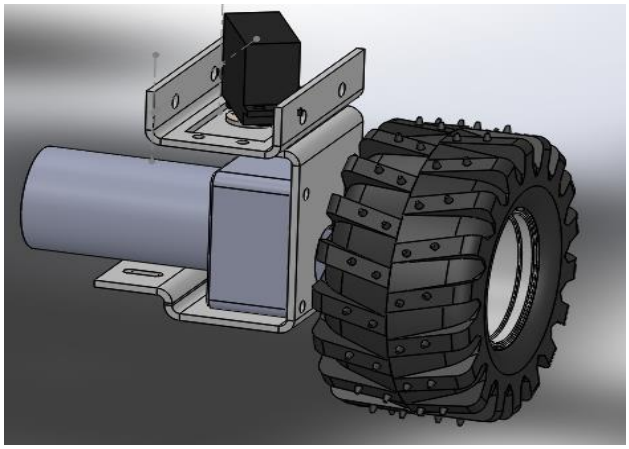


Figure 2.6 wall-E motor wheel mounting

2.9 Rocker and Bogie Suspension System

Using CNC machining we could cut sheet metal we reached to this shape:

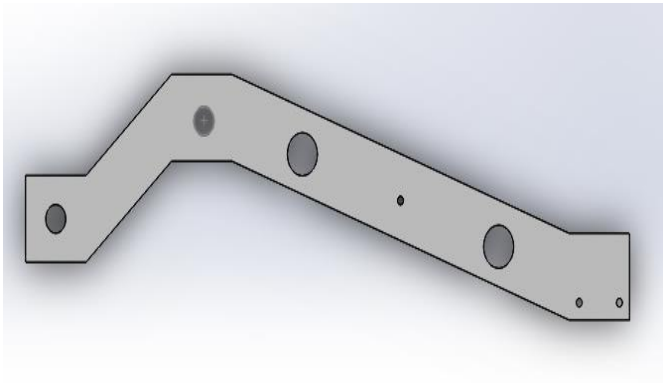


Figure 2.7 Wall -E Rocker

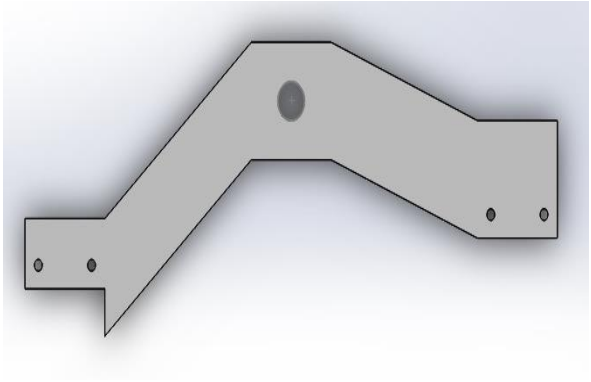


Figure 2.8 Wall-E Bogie

- Using CNC machining we could cut sheet metal to reach to these shapes:

Differential pivot parts:

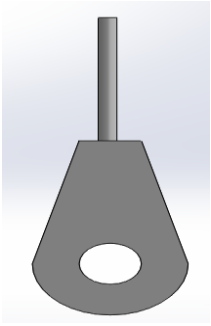


Figure 2.11 part 1

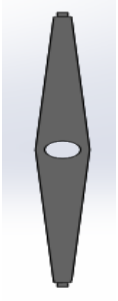


Figure 2.9 pivot

Figure 2.10 spherical joint

- By integrate all parts of rover we can reach to the final shape of rover.

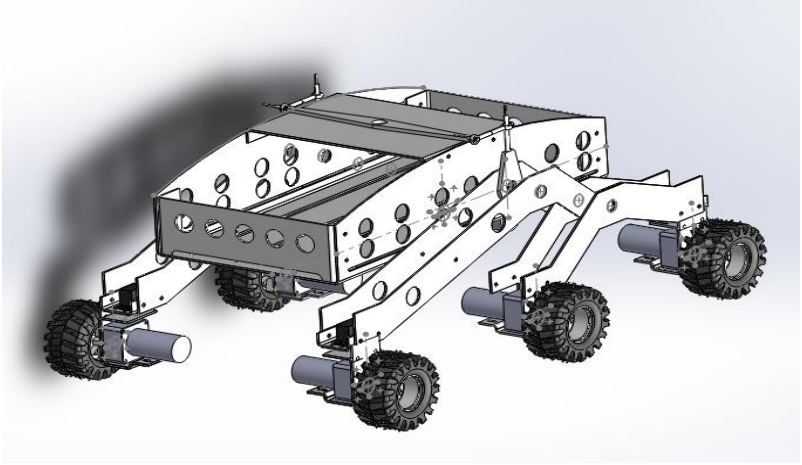


Figure 2.12 Rover Assembly

2.10 Modification of Rocker and Bogie suspension System

We changed the design of the rocker bogie system from the previous one by replacing the pivot with the differential gear box

The main design difference, as it pertains to the suspension system, is the addition of a differential gear box as shown in fig 2.13

The differential is composed of three identical beveled gears as shown in fig 2.14 situated 90-degrees from each other at the center of the rover. Each gear is affixed to a 6-inch steel drive shaft that is mounted to the rover body by 2 mounted sleeve bearings. One gear connected to the left, one gear connected to the right, and the last gear assembled onto the main platform. The two rods facing opposite of each other on the left and right of the robot help serve as axels for the wheel assemblies. The axils are connected to the wheel assemblies by 2 parallel aluminum tubes that make up the rocker bogie suspension system.

The main advantage of this type of suspension system is that the load on each wheel is nearly identical and thus allows an equal distribution of force regardless of wheel position. Taking into consideration the harsh terrain found on Mars this type of suspension system provides a better alternative to that of the common 4-wheel drive soft suspensions found on most automobiles.

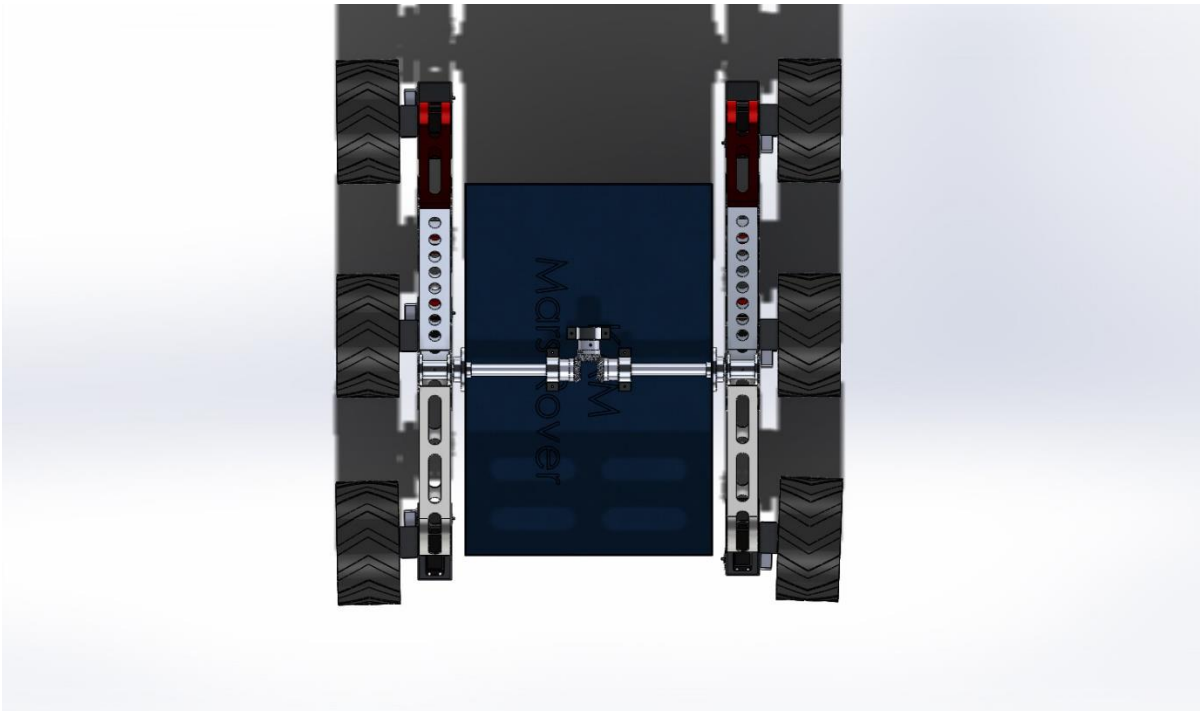


Figure 2.13 the differential gearbox

Figure 2.15 shaft connects bogie to gear

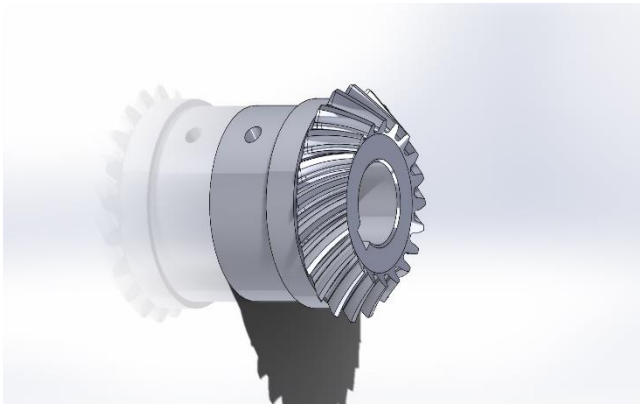


Figure 2.14 beveled gear

We did that change for more stability of the rover and we tried to transform the SolidWorks design to URDF to make the simulation of the rover on gazebo but we faced another problem on that. we believe that change should be applied to the Rover.

ROVER INTEGRATION

By integrate all parts of rover and the modification we made we can reach to the final shape of rover.

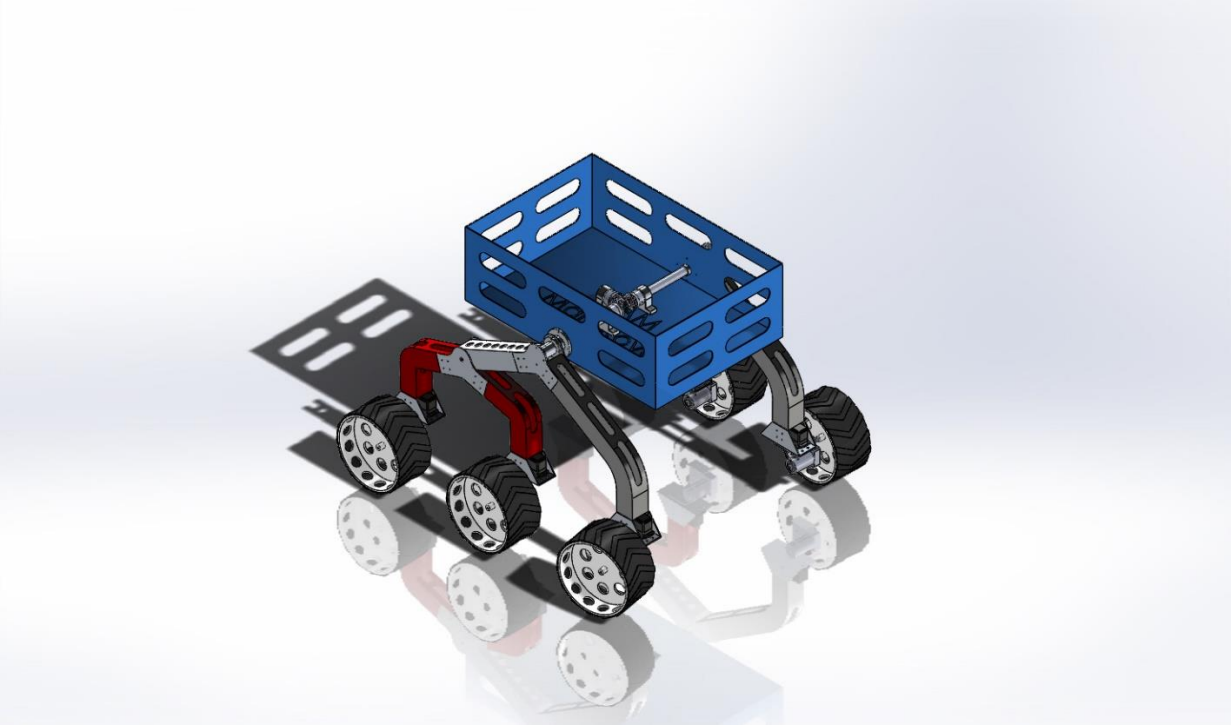


Figure 2.16 The assembly of the rover with differential gear suspension system

2.11 Working Drawing

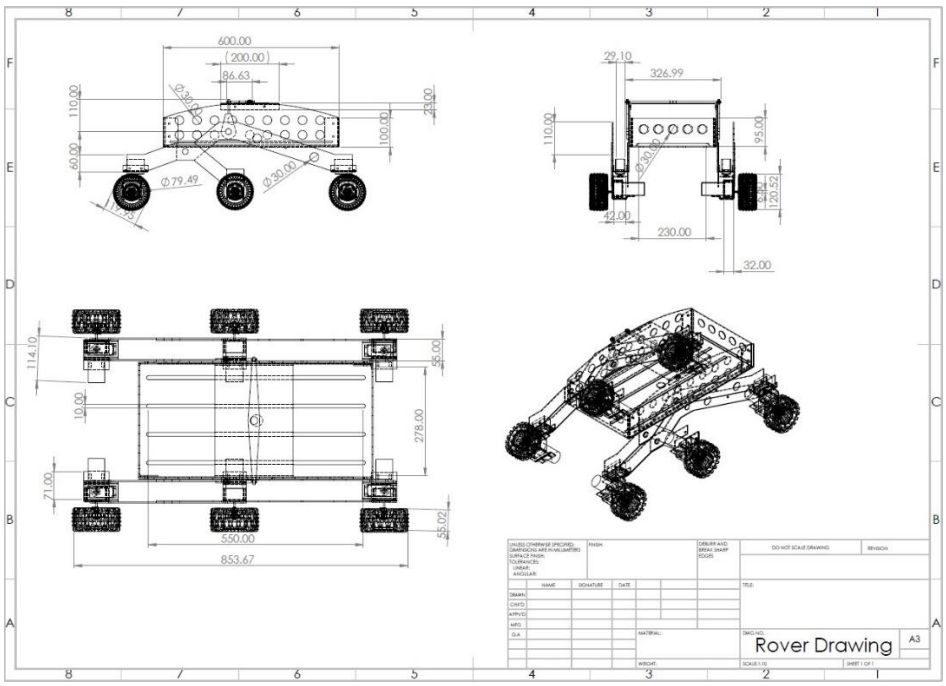


Figure 2.17 The Drawing on the solid works

2.12 Stress Analysis

We check the stability by making structure analysis using ANSYS as shown in fig 2.18 and 2.19

The Yield strength of aluminum alloy is 250 MPa

Modeling: Load Applied on Differential joint and, Displacement supports in the wheels

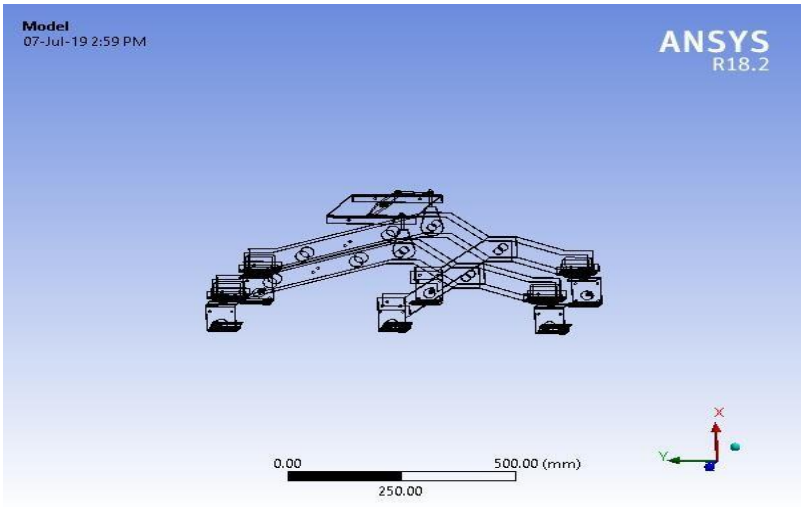


Figure 2.18 Ansys Geometry

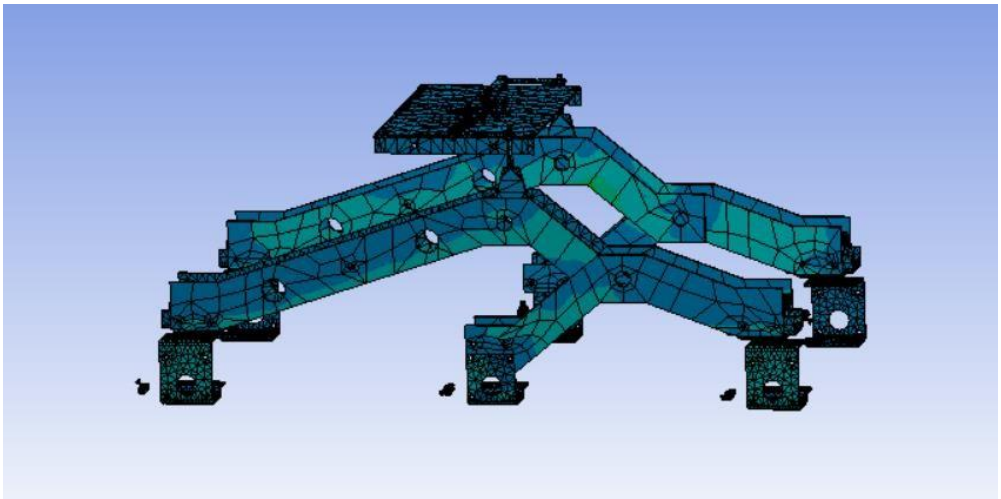


Figure 2.19 Ansys Model

Ansys Results: The maximum stress is less than yield strength, and the resulting structure is stable

2.13 Modeling

2.13.1 Overview

The effectiveness of a wheeled mobile robot has been proven by NASA by sending a semi-autonomous rover "Sojourner" landed on Martian surface in 1997. Future field mobile robots are expected to traverse much longer distance over more challenging terrain than Sojourner, and perform more difficult tasks. Other examples of rough terrain applications for robotic can be found in hazardous material handling applications, such as explosive ordnance disposal, search and rescue.

Corresponding to such growing attention, the Researches are varying from mechanical design, performance of the robot, control system, navigation systems, path planning, field test, and so on. However, there are very few dynamics of the robot. This is because the field robots are considered too slow to encounter dynamic effect. And the high mobility of the robot, moving in 3 dimensions with 6 degrees of freedom (X, Y, Z, pitch, yaw, roll), makes the kinematics modeling a challenging task than the robots which move on flat and smooth surface (3 degrees of freedom: X, Y, rotation about Z axis) In rough terrain, it is critical for mobile robots to maintain maximum traction. Wheel slip could cause the robot to lose control and trapped. Traction control for low-speed mobile robots on flat terrain has been studied by Reister and Unseren using pseudo velocity to synchronize the motion of the wheels during rotation about a point. Sreenivasan and Wilcox have considered the effects of terrain on traction control by assume knowledge of terrain geometry, soil characteristics and real-time measurements of wheel-ground contact forces. However, this information is usually unknown or difficult to obtain directly.

Quasi-static force analysis and a fuzzy logic algorithm have been proposed for a rocker-bogie robot. Knowledge of terrain geometry is critical to the traction control. A method for estimating wheel-ground contact angles using only simple on-board sensors has been proposed. A model of load-traction factors and slip-based traction has been developed. The traveling velocity of the robot is estimated by measure the PWM duty ratio driving the wheels. Angular velocities of the wheels are also measured then compare with estimated traveling velocity to estimate the slip and perform traction control loop.
In this research work, a small six-wheel robot with Rocker-Bogie suspension is designed, built and tested. The method to derive mathematical modeling such as, the wheel- ground contact angle estimation and kinematics modeling also described. Finally, a traction control system is developed and implemented on this robot.

2.13.2 Rover test bed

In this research, the robot test bed named "Lonotech 10" is built. Its dimensions are 800x600x480 mm^3, consists of six wheels, three on each side. Four steering mechanisms are equipped to the front and rear wheels.

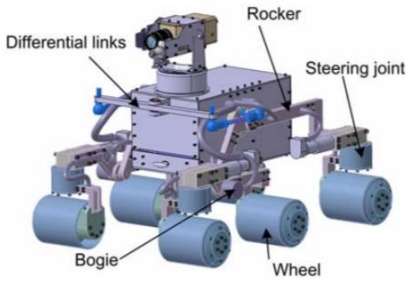


Figure 2.20 Lomotech 10

All independently actuated wheels are connected by the Rocker-Bogie suspension, a passive suspension that works well at low-velocity. This suspension consists of two rocker arms connected to the sides of the robot body. At one end of each rocker is connected to pivot of the smaller rocker, the bogie, and the other end has a steerable wheel attached. Two wheels are attached to the end of these bogies. The rockers connected to the body via a differential link. This configuration maintains the pitch of the body equal to the average angle between the two rockers. This mechanism also provides an important mobility characteristic of the robot: one wheel can be lifted vertically while other wheels remain in contact with the ground.

In order to climb over an obstacle, the front wheels are forced against the obstacle by the middle and rear wheels. Then the rotations of the front wheels lift the front of the robot up and climb over the obstacle.

The middle wheels are pressed against the obstacle by the rear wheels and pulled by the fronts. Finally, the rear wheels are pulled by the front and middle wheels.

The robot equipped with various sensors for navigation. Most of the sensors are mounted on the pan-tilt mechanism, including CCD Camera, laser pointer, and distance measuring sensor. Three encoders are attached to the suspension to sense rocker and bogie angles. Inclinometers and accelerometer are also installed. The accelerometer is used to calculate the acceleration and velocity of the robot as the feedback data of the control system.

2.13.3 Wheel-Ground Contact Angle Estimation

To formulate a kinematic model for the robot, the wheel-ground contact angles must be known. But it is difficult to make a direct measurement of the angles, so a method for estimating the contact angles based on Iagnemma and Dubowsky is implemented in this section.

In kinematics modeling and contact angle estimation, we introduce the following assumptions:

1) Each wheel makes contact with the ground at a single point.
2) No side slip and rolling slip between a wheel and the ground.

Consider the left bogie on uneven terrain, the bogie pitch, μ_1, is defined with respect to the horizon. The wheel center velocities v_1 and v_2 are parallel to the wheel-ground tangent plane. The distance between the wheel centers is L_B.

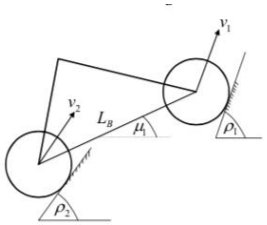


Figure 2.21 The Left Bogie on uneven terrain

The kinematics equations can be written as follows:

$$\nu_1 cos(\rho_1 - \mu_1) = \nu_2 \, cos(\rho_2 - \mu_1)$$

$$\nu_1 sin(\rho_1 - \mu_1) - \nu_2 \, sin(\rho_2 - \mu_2) = L_B \dot{\mu_1}$$

Combining two equations:

$$sin[(\rho_1 - \mu_1) - (\rho_2 - \mu_1)] = \frac{L_B \dot{\mu_1}}{v_1} cos(\rho_2 - \mu_1)$$

Define: $a_1 = (L_B \dot{\mu_1}) / \nu_1$ and $b_1 = \nu_2 / \nu_1$

Contact angles of the wheel 1 and 2 are given by

$$\rho_1 = \mu_1 + \arcsin\left(\frac{{a_1}^2 - {b_1}^2}{2a_1}\right)$$

$$\rho_2 = \mu_1 + \arcsin\left(\frac{1 + {a_1}^2 - {b_1}^2}{2a_1}\right)$$

In order to compute the contact angle of the rear wheel, we need to know the velocity of the bogie joint first:

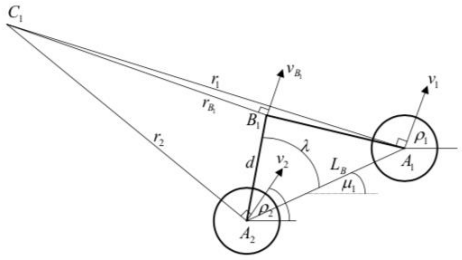


Figure 2.22 Instantaneous center of rotation of the left bogie

The velocity of the bogie joint can be written as:

$$\nu_{B_1} = r_{B_1} \dot{\mu}_1$$

Where $$r_{B_1} = \sqrt{{r_2}^2 + d^2 - 2 r_2 d cos(90 + \rho_2 - \mu_1 - \lambda)}$$

$$r_1 = \frac{L_B \sin(90 + \rho_2 - \mu_1)}{\sin(\rho_1 - \rho_2)}$$

$$r_2 = \frac{L_B \sin(90 - \rho_2 + \mu_1)}{\sin(\rho_1 - \rho_2)}$$

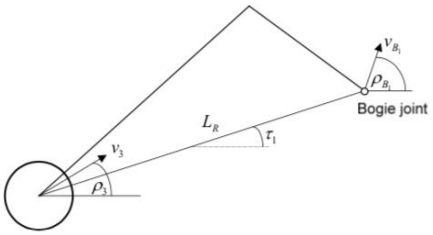


Figure 2.23 Left rocker on uneven terrain

Contact angles of the wheel 3 is

$$\rho_3 = \arccos\left(\frac{v_{B_1}}{v_3}\cos\left(\rho_{B_3} - \tau_1\right)\right)$$

In the same way, we repeated these procedures with the right side:

$$\rho_4 = \mu_2 + \arcsin\left(\frac{{a_2}^2 - {b_2}^2}{2a_2}\right)$$

$$\rho_4 = \mu_2 + \arcsin\left(\frac{1 + {a_2}^2 - {b_2}^2}{2a_2}\right)$$

$$\rho_6 = \arccos\left(\frac{v_{B_2}}{v_6}\cos\left(\rho_{B_2} - \tau_2\right)\right)$$

2.13.4 Forward Kinematics

We define coordinate frames as in figure 2.24 and 2.25. The subscripts for the coordinate frames are as follows: *O*: robot frame, *D*: differential joint, B_i: left and right bogie (i = 1,2),

S_i : steering of left front, left rear, right front, right rear wheels (i = 1,3,4,6) and A_i :axle of all wheels (i = 1 − 6).

Other quantities shown in figure 2.24 and 2.25 are ψ_i :steering angles (i = 1,3,4,6), β :rocker angle, γ_1 and γ_2 :left and right bogie angle respectively.

Following the Denavit-Hartenburg notations, table 1 provides the parameters corresponding to various coordinate frames. This transformation consists of a rotation Θ about Z-axis, a translation *d* along the Z-axis, a translation *a* along the X-axis and a rotation α about the X-axis. The transformation matrix for adjacent coordinate frame *i* to *j* can be written as follows:

$$T_{ji} = \begin{bmatrix} \cos(\Theta_j) & -\sin(\Theta_j)\cos(\alpha_j) & \sin(\Theta_j)\sin(\alpha_j) & a_j\cos(\Theta_j) \\ \sin(\Theta_j) & \cos(\Theta_j)\cos(\alpha_j) & -\cos(\Theta_j)\sin(\alpha_j) & a_j\sin(\Theta_j) \\ 0 & \sin(\alpha_j) & \cos(\alpha_j) & d_j \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where Θ_j , α_j, a_j and d_j are the D-H parameters given for coordinate frame j.

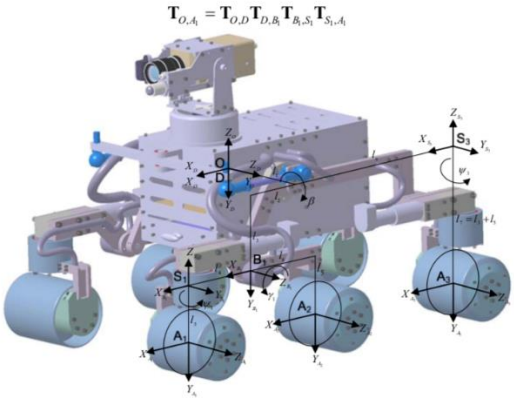


Figure 2.24 Left coordinate frame

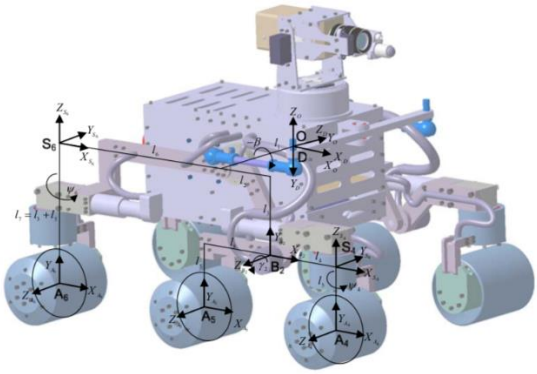


Figure 2.25 Right coordinate frame

The transformations from the robot reference frame (O) to the wheel axle frames (A_i) are obtained by cascading the individual transformations. For example, the transformations for the wheel 1 are

$$T_{O,A} = T_{O,D}T_{D,B}T_{B,S}T_{S,A}$$

In order to capture the wheel motion, we need to derive two additional coordinate frames for each wheel, contact frame and motion frame. Contact frame is obtained by rotating the wheel axle frame (A_i) about the Z-axis until the X-axis of contact frame parallel to the ground then followed by a 90 degree

rotation about the X-axis. The Z-axis of the contact frame (C_i) points away from the contact point as shown in figure 2.26.

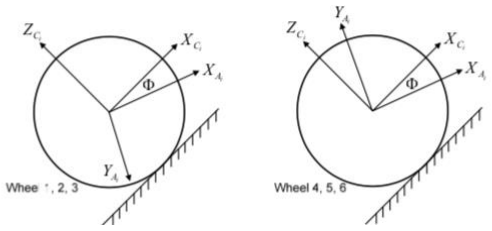


Figure 2.26 Contact coordinate frame

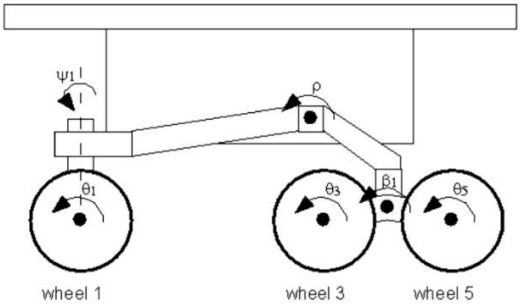


Figure 2.27 Schematic diagram of rocker-bogie shown joint angles and wheel-rolling angles

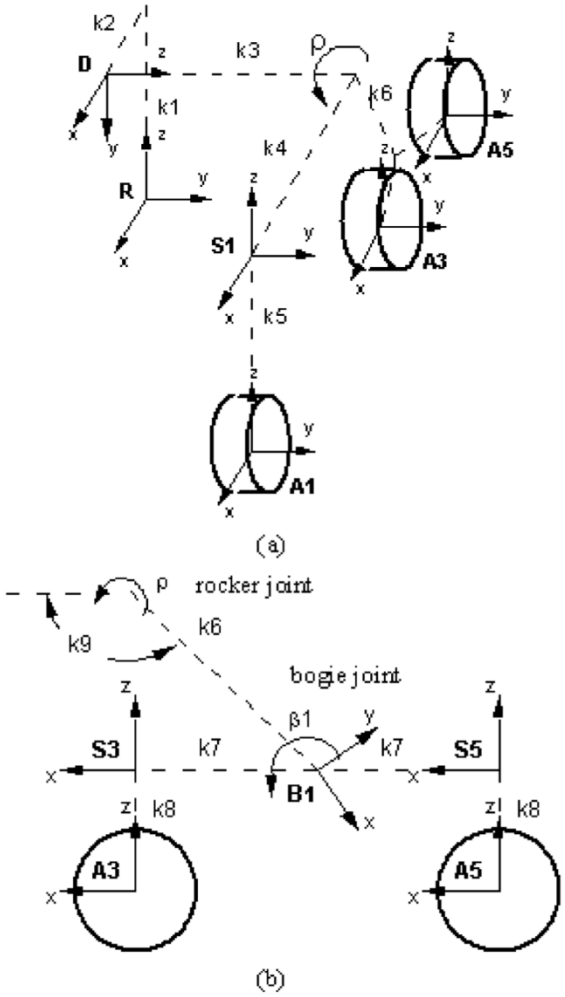


Figure 2.28 Coordinate frames for rover's left side

Table 2.4 Bogie and Differential Parameters

Frame	γ	d	a	α
D	0	K_1	K_2	-90
B_1	$K_9 + \rho$	K_3	K_6	0
B_2	$K_9 - \rho$	$-K_3$	K_6	0

Table 2.5 Steering Parameters

Frame	γ	d	a	α
S_1	ρ	K_3	K_4	90
S_2	$-\rho$	$-K_3$	K_4	90
S_3	$\beta_1 - k_9$	0	K_7	90
S_4	$\beta_2 - k_9$	0	K_7	90
S_5	$\beta_1 - k_9$	0	$-K_7$	90
S_6	$\beta_2 - k_9$	0	$-K_7$	90

Table 2.6 Axle Parameters

Frame	γ	d	a	α
A_1	Ψ_1	$-K_5$	0	0
A_2	Ψ_2	$-K_5$	0	0
A_3	0	$-K_8$	0	0
A_4	0	$-K_8$	0	0
A_5	0	$-K_8$	0	0
A_6	0	$-K_8$	0	0

The transformation matrices for contact frame can be derived by using Z-X-Y Euler angle.

$$T_{A,C} = \begin{bmatrix} Cp_iCr_i - Sp_iSq_iSr_i & Sq_iCp_iSr_i + Sp_iCr_i & -Cq_iSr_i & 0 \\ -Cq_iSp_i & Cq_iCp_i & Sq_i & 0 \\ Sq_iSp_iCr_i + Cp_iCr_i & -Sq_iCp_iCr_i + Sp_iSr_i & Cq_iCr_i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where p_i , q_i and r_i are rotation angle about X, Y and Z respectively.

The wheel motion frame is obtained by translating along the negative Z-axis by wheel radius (R_w) and translating along the X-axis for wheel roll (R_w θ_i).

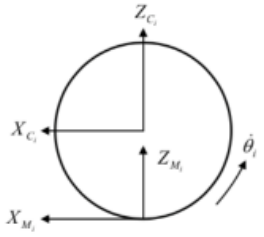


Figure 2.29 Wheel motion frame

The transformation matrices for all wheels can be written as follows:

$$T_{O,M1} = T_{O,D}T_{D,B1}T_{B1,S1}T_{S1,A1}T_{A1,C1}T_{C1,M1}$$

$$T_{O,M2} = T_{O,D}T_{D,B1}T_{B1,A2}T_{A2,C2}T_{C2,M2}$$

$$T_{O,M3} = T_{O,D}T_{D,S3}T_{S3,A3}T_{A3,C3}T_{C3,M3}$$

$$T_{O,M4} = T_{O,D}T_{D,B2}T_{B2,S4}T_{S4,A4}T_{A4,C4}T_{C4,M4}$$

$$T_{O,M5} = T_{O,D}T_{D,B2}T_{B2,A5}T_{A5,C5}T_{C5,M5}$$

$$T_{O,M6} = T_{O,D}T_{D,S6}T_{S6,A6}T_{A6,C6}T_{C6,M6}$$

In order to obtain the wheel Jacobian matrices, we must express the motion of the robot to the wheel motion frame, by applying the instantaneous transformation $\dot{T}_{\dot{O},\dot{M}}$ as follows:

$$\dot{T}_{\dot{O},O} = T_{\dot{O},\dot{M}}\dot{T}_{M,O}$$

Where derivative of transformation matrices, and instantaneous transformations

$\dot{T}_{\dot{O},O}$ is found to have the following form

$$\dot{T}_{\dot{O},O} = \begin{bmatrix} 0 & -\dot{\phi} & \dot{p} & \dot{x} \\ \dot{\phi} & 0 & -\dot{r} & \dot{y} \\ -\dot{p} & \dot{r} & 0 & \dot{z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Once the instantaneous transformations of each wheel are obtained, we can extract a set of equations relating the robot's motion in vector form $[\dot{x}\ \dot{y}\ \dot{z}\ \dot{\phi}\ \dot{p}\ \dot{r}]$ to the joint angular rates.

The resulting equation for wheels 1 and 2 is

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\phi}_x \\ \dot{\phi}_y \\ \dot{\phi}_z \end{bmatrix} = \begin{bmatrix} -b_i k_1 & J_{x,\psi} & J_{x,\theta} & J_{x,\delta} \\ 0 & J_{y,\psi} & J_{y,\theta} & J_{y,\delta} \\ b_i k_2 & J_{z,\psi} & J_{z,\theta} & J_{z,\delta} \\ 0 & J_{\phi x,\psi} & 0 & J_{\phi x,\delta} \\ b_i & 0 & 0 & J_{\phi y,\delta} \\ 0 & J_{\phi\, z,\psi} & 0 & J_{\phi z,\delta} \end{bmatrix} \times \begin{bmatrix} \dot{\rho} \\ \dot{\psi}_i \\ \dot{\theta}_i \\ \dot{\delta}_i \end{bmatrix}$$

The resulting equation for wheels 3, 4, 5 and 6 is

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\phi}_x \\ \dot{\phi}_y \\ \dot{\phi}_z \end{bmatrix} = \begin{bmatrix} -b_i k_1 & J_{x,\beta 1} & J_{x,\beta 2} & k_{10} C\sigma_i & J_{x,\delta} \\ 0 & 0 & 0 & 0 & 0 \\ b_i k_2 & J_{z,\beta 1} & J_{z,\beta 2} & -k_{10} S\sigma_i & J_{z,\delta} \\ 0 & 0 & 0 & 0 & 0 \\ b_i & J_{\phi y,\beta 1} & J_{\phi y,\beta 2} & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} \dot{\rho} \\ \dot{\beta}_1 \\ \dot{\beta}_2 \\ \dot{\theta}_i \\ \dot{\delta}_i \end{bmatrix}$$

2.13.5 Inverse Kinematics

The purpose of inverse kinematics is to determine the individual wheel rolling velocities which will accomplish desired robot motion. The desired robot motion is given by forward velocity and turning rate. In this section, we will develop all 6 wheels rolling velocities with geometric approach to determine steering angle of steerable wheels.

2.13.5.1 Wheel Rolling Velocities

The resulting equation for wheels 1 and 2 is

$$\dot{x} = -b_i k_1 \dot{\rho} + J_{x,\psi} \dot{\psi} + J_{x,\theta} \dot{\theta} + J_{x,\delta} \dot{\delta}_i$$

$$\dot{\theta} = \frac{\dot{x}}{J_{x,\theta}} + \frac{b_i k_1}{J_{x,\theta}} \dot{\rho} - \frac{J_{x,\psi}}{J_{x,\theta}} \dot{\psi} - \frac{J_{x,\delta}}{J_{x,\theta}} \dot{\delta}_i$$

The resulting equation for wheels 3, 4, 5 and 6 is

$$\dot{x} = -b_i k_1 \dot{\rho} + J_{x,\beta 1}\dot{\beta}_1 + J_{x,\beta 2}\dot{\beta}_2 + J_{x,\delta}\dot{\delta}_i + k_{10} C \sigma_i \dot{\theta}_i + J_{x,\delta}\dot{\delta}_i$$

$$\dot{\theta}_i = \frac{\dot{x}}{k_{10} C \sigma_i} + \frac{b_i k_1}{k_{10} C \sigma_i}\dot{\rho} - \frac{J_{x,\beta 1}}{k_{10} C \sigma_i}\dot{\beta}_1 - \frac{J_{x,\beta 2}}{k_{10} C \sigma_i}\dot{\beta}_2 - \frac{J_{x,\delta}}{k_{10} C \sigma_i}\dot{\delta}_i$$

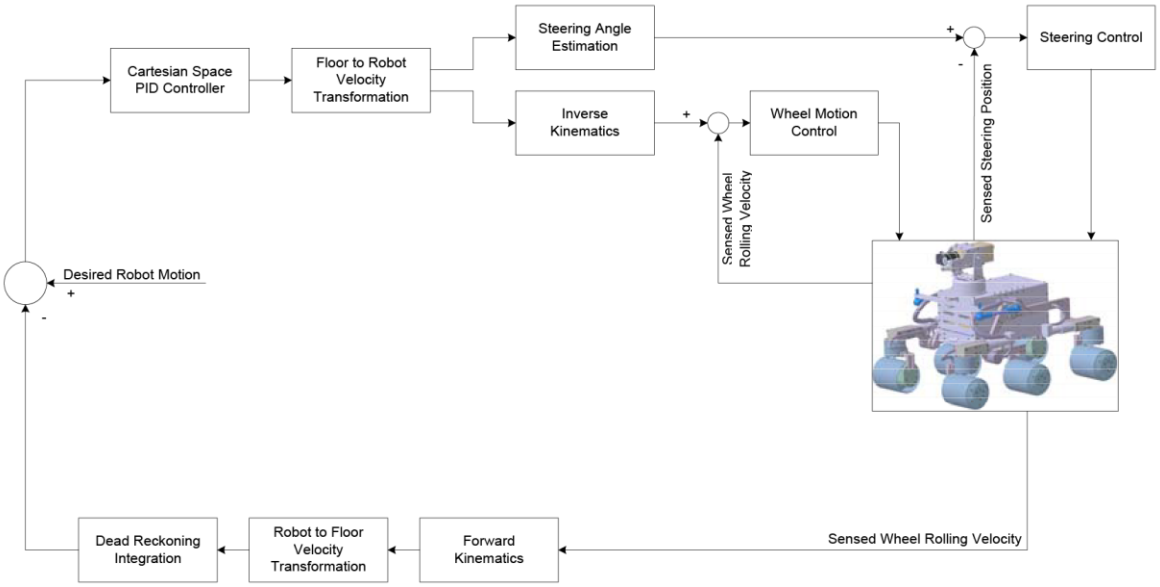


Figure 2.30 Robot control schematic

2.13.5.2 Steering Angles

Center of rotation, called turning center, is estimated based on the two non-steerable middle wheels. This turning center will be used to determine the steering angles of the four corner wheels. We can derive coordinate of the wheel centers respect to the robot reference frame as follows:

$$X_{C1} = l_2 \cos\beta + l_3 \sin\beta + l_4 \cos(\beta - \gamma_1) + l_5 \sin(\beta - \gamma_1)$$

$$X_{C2} = l_2 \cos\beta + l_3 \sin\beta - l_8 \cos(\beta - \gamma_1) + l_5 \sin(\beta - \gamma_1)$$

$$X_{C3} = -l_6 \cos\beta + (l_3 + l_5)\sin\beta$$

$$X_{C4} = l_2 \cos(-\beta) - l_3 \sin(-\beta) + l_4 \cos(-\beta + \gamma_2) + l_5 \sin(-\beta + \gamma_2)$$

$$X_{C5} = l_2 \cos(-\beta) - l_3 \sin(-\beta) - l_8 \cos(-\beta + \gamma_2) + l_8 \sin(-\beta + \gamma_2)$$

$$X_{C6} = -l_6 \cos(-\beta) - (l_3 + l_5)\sin\beta$$

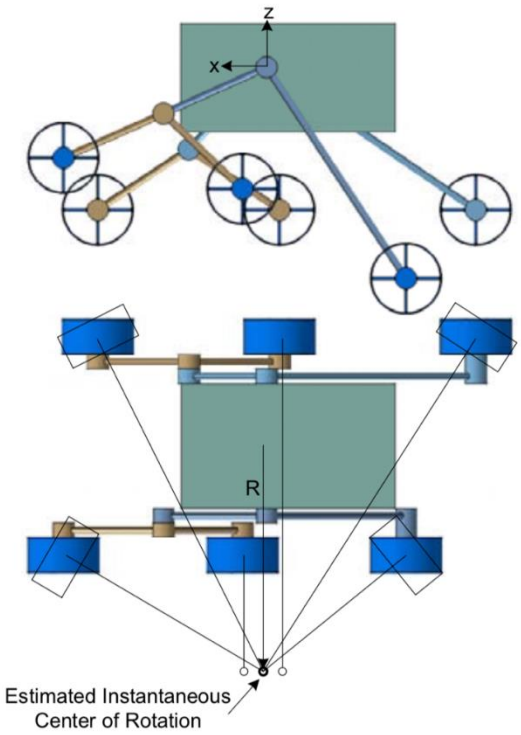


Figure 2.31 Instantaneous center of rotation

From figure shown, the instantaneous center of rotation can be estimated by averaging the x position of the middle wheels. The distance in Z-axis is neglected because there is only 1 degree of freedom per each steering. If the wheel's axis is steered to intersect with the center of rotation on the X-axis, the angle in Z direction is coupled and cannot be controlled.

Using the estimated center of rotation, the desired steering angle for each steerable wheel can be determined. Define R is a turning radius, x_R is the distance in X-direction of the center of rotation with respect to the robot reference frame, l_1 is the distance from the robot reference frame to steering joint in Y-direction. The desire steering angles are

$$\Psi_1 = \arctan\left(\frac{X_{C1} - X_R}{R - L_1}\right) \; For\ wheel\ 1$$

$$\Psi_3 = \arctan\left(\frac{X_{C3} - X_R}{R - L_1}\right) \; For\ wheel\ 3$$

$$\Psi_4 = \arctan\left(\frac{X_{C4} - X_R}{R - L_1}\right) \; For\ wheel\ 4$$

$$\Psi_6 = \arctan\left(\frac{X_{C6} - X_R}{R - L_1}\right) \; For\ wheel\ 6$$

2.13.5.3 Ackerman Steering

Control Syste

There are two main issues we need to address when designing the control mechanics for the rover. One is the direction that each of the corner wheels needs to point in order for the rover to be able to turn correctly. The other is how fast each of the individual drive motors needs to spin (it's not the same speed at each wheel when the rover is turning).

Drive Motor calculations

The 6 wheels rover design employs Ackerman steering, which can be seen in Figure 2.32. The rover steers about a point which lies on the line that passes through the two center wheels. All 6 wheels need to align their center axis towards the point around which the rover will turn. Notice that we observe a more drastic steering angle for the wheels closer to that point than for the wheels that are further away.

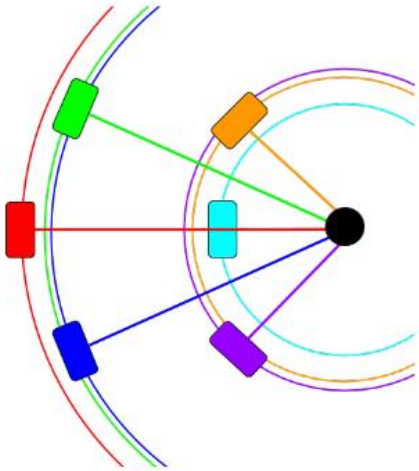


Figure 2.32 Rover turning about a point

Let's go step-by-step though how to calculate the speed of each of the drive wheels. To start, we will look at the rover which will be turning about a point P at radius r, shown in Figure 2.33

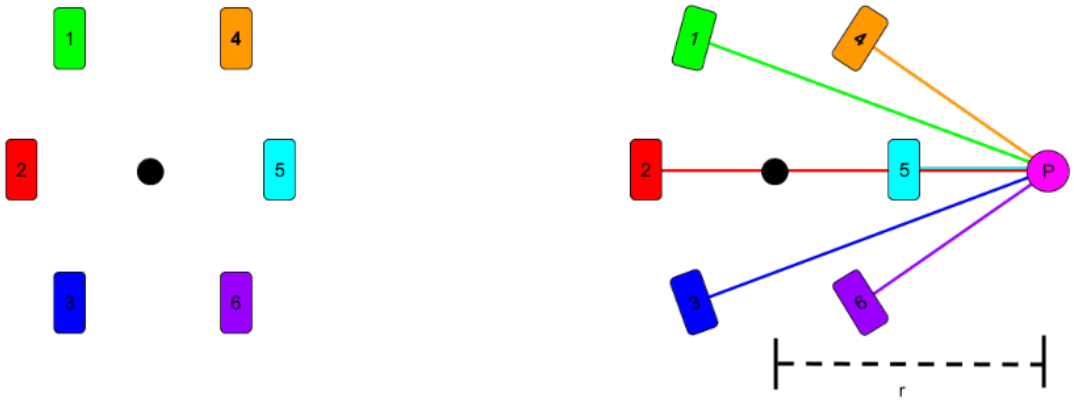


Figure 2.33 Ackerman Steering

We start by sending a signal from the controller for the rover to drive forward. Let's call the speed at which it drives X (for now, we set X to its maximum value of 100). By convention, this will tell the middle of the rover that it should move forwards at the max speed of 100. However, if the rover is turning, each wheel will need to spin at a slightly different rate to avoid slipping or "scrubbing". Because no wheel can spin faster than speed 100 and because the wheels have differing distances they must travel along their arcs during the turn, we send the max speed of 100 to the wheel furthest away from the point we are turning around. In the case above, this is wheel #2. Therefore, we now set wheel #2 to spin at speed 100 as shown in Figure 2.34.

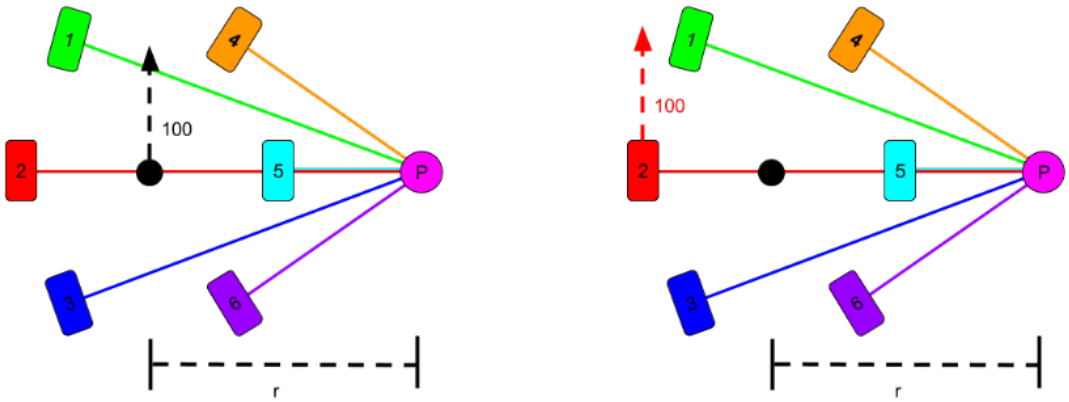


Figure 2.34 Rover distributing speed to fastest/ furthest wheel

Next, we examine how to distribute speed to the other wheels. We will make use of arc lengths of the circles the wheels turn about which is given by the relation in:

$$S = R\theta$$

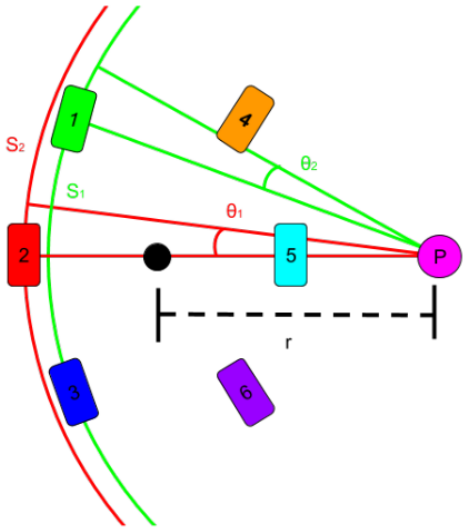


Figure 2.35 Arc lengths of driving curves

In order for the entire rover to drive cohesively each wheel must travel the same number of degrees from the following equations in the same time, so we can solve for the relation between each of the wheels now. (In the equations below, θ is the angle that the wheel traverses, S is the arc length that the wheel travels, and R is the radius or distance from the wheel to the turning point)

$$\theta_1 = \frac{S_1}{R_1}$$

$$\theta_2 = \frac{S_2}{R_2}$$

$$\theta_1 = \theta_2$$

$$S_1 = \frac{S_1 \times R_1}{R_2}$$

From above Equation we can see that the ratio of the distance traveled (and thus the speed) of each of the wheels is going to be given by the ratio of the radius of the arc that each wheel is turning about. The last step is to figure out the radius of that arc for each wheel.

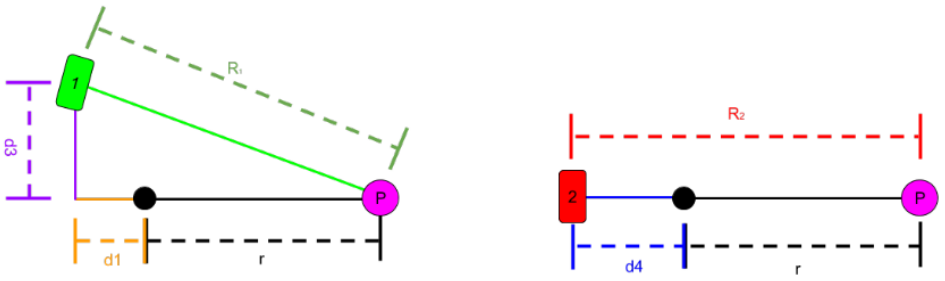


Figure 2.36 Radius of arc for turning

the distances d 1, and d 3 are geometric relations based on the mechanical lengths of the robot itself. Putting this all together, let's look at an example of calculating speed and turning distance.

Variable	Physical description
d_1	Horizontal distance between middle of rover and the corner wheels.
d_2	Vertical distance between the middle of rover and back corner wheels.
d_3	Vertical distance between the middle of the rover and the front corner wheels.
d_4	Horizontal distance between the middle of the rover and the center wheels.

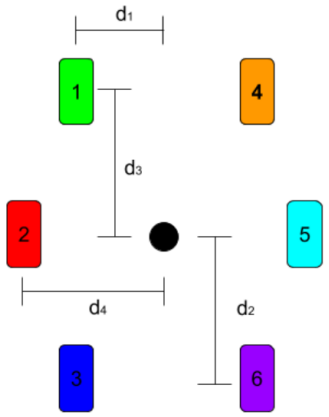


Figure 2.37 Physical distance of wheel geometry

For the version of the rover described in the build documents, the distances are as follows:

$$d_1 = 27\text{cm}$$

$$d_2 = d_3 = 35\text{cm}$$

$$d_4 = 27\text{cm}$$

Using Equations 3 and 4 we get the following relations for the speed distribution. We substitute Equation 4 into 3, which solves for the arc length distance. This is analogous to the speed of each wheel, as speed is just speed= $\frac{distance}{time}$. We denote speed with V (for velocity), and we now have:

$$V_1 = X\frac{R_1}{R_2} = X\frac{\sqrt{d_3^2 + (d_1 + r)^2}}{r + d_4}$$

$$V_2 = X$$

$$V_3 = X\frac{R_3}{R_2} = X\frac{\sqrt{d_2^2 + (d_1 + r)^2}}{r + d_4}$$

$$V_4 = X\frac{R_4}{R_2} = X\frac{\sqrt{d_3^2 + (d_1 - r)^2}}{r + d_4}$$

$$V_5 = X\frac{R_5}{R_2} = X\frac{\sqrt{d_3^2 + (d_1 + r)^2}}{r + d_4}$$

$$V_6 = X\frac{R_6}{R_2} = X\frac{r - d_4}{r + d_4}$$

2.13.5.4 Corner Motor Positions

To signal turning, the rover will take values from the controller from -100 to 100. This value represents the percentage of the minimum (tightest) turning radius that the wheels can handle, with negative numbers turning to the left and positive numbers to the right. The higher the magnitude of the number, the tighter turning radius, i.e. -100 means the tightest turn possible to the left, 0 means straight ahead, and 100 means the tightest turn possible to the right. This "tightest" turning radius is determined by a combination of the geometric distances of the rover as well as the physical hard stop limitations of the corner motors. In our system, each of the corner motor is allowed to turn up to 45 degrees before hitting a physical hard stop, as shown in Figure 2.38.

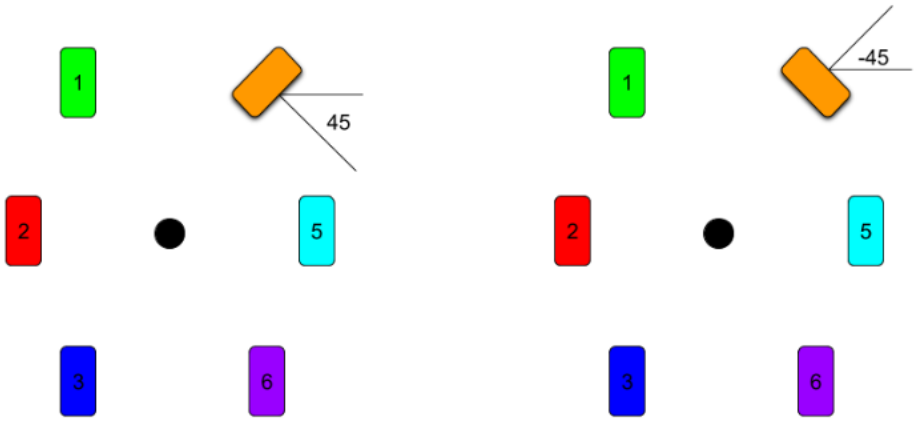


Figure 2.38 Corner angle limitations

In order to figure out the tightest turning radius, we need to solve for the closest place along the rover's center axis that the corner wheel can point given its 45 ◦ limitation. This is shown in Figure 2.39:

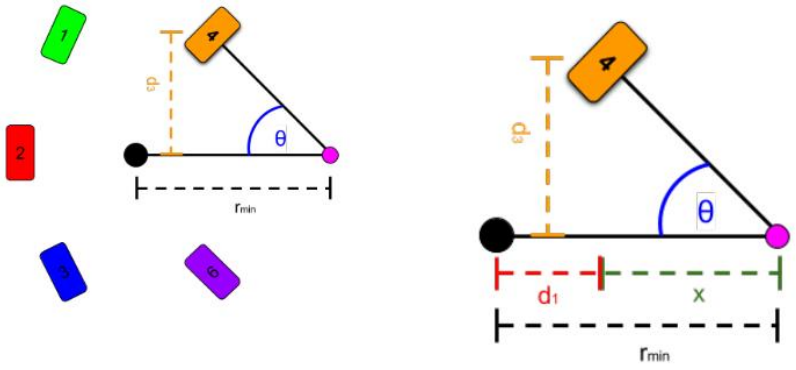


Figure 2.39 Minimum turning radius calculations

$$\tan(\theta) = \frac{d_3}{x}$$

$$r_{min} = d_1 + x$$

$$r_{min} = d_1 + \frac{d_3}{\tan(\theta)}$$

$$r_{min} = 27 + \frac{35}{\tan(45^{\circ})}$$

$$r_{min} = 62\text{cm}$$

2.13.5.5 Kinematics of our Rover (Bicycle Model)

While going through the Sebastian Thrun‘s AI for Robotics class, I came across a programming assignment that required implementing basic kinematics of a simple robot in 2D space. The robot uses bicycle model. Here is the underlying geometry that give rise to the basic equations used in the class.

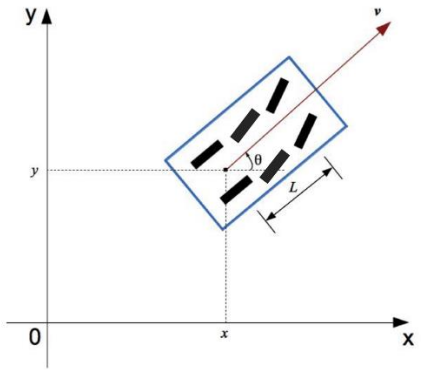


Figure 2.40 A robot (car) movement in 2D space

As shown in Fig.2.40 , the robot lies in a global Cartesian coordinate space and is characterized by its position and heading, (x, y, θ), where θ is the heading relative to x-axis. The length of the robot is L and its velocity along heading direction θ is υ.

Let's assume the robot's origin (center of rear axle in Fig. 2.40) moves a distance d along the heading direction. We are interested in the turn angle that the robot incurred during the movement. Fig. 2.41 shows the bicycle model of the robot, the angular velocity of the robot can be expressed as

$$\dot{\theta} = \frac{\upsilon}{R}$$

Where R is the radius of turn.

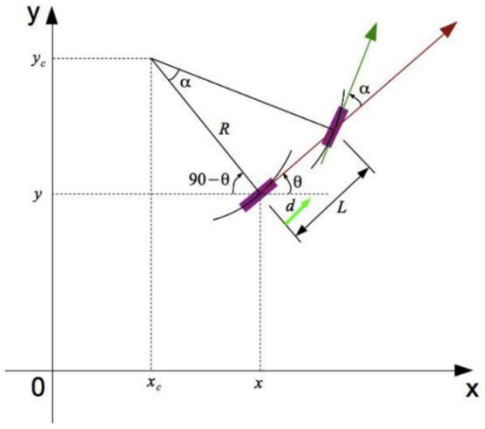


Figure 2.41 Bicycle model of the robot (car)

We can also write,

$$\tan(\alpha) = \frac{L}{R} \rightarrow R = \frac{L}{\tan(\alpha)}$$

Using above 2 equations, the turn angle is given by

$$\beta = \int \dot{\theta} dt = \frac{d}{L}\tan(\alpha) = \frac{d}{R}$$

Where integration is performed over the time during which the rear wheel moves distance d.

Coordinates of center of turn can be easily computed using Fig. 1.35,

$$X_C = X - Rsin(\theta)$$

$$Y_C = Y + R\,Cos(\theta)$$

New heading of the robot after the turn can be obtained by adding β and original headingθ. Updated robot position and heading are given by

$$X' = X_C + Rsin(\beta + \theta)$$

$$y' = y_C - Rcos(\beta + \theta)$$

$$\theta' = (\theta + \beta)\ mod(2\pi)$$

When β → 0 and R → ∞ . We can approximate this as just a motion in straight line along heading direction. So,

$$X' = X + dcos(\theta)$$

$$y' = y + dsin(\theta)$$

$$\theta' = (\theta + \beta)\ mod(2\pi) \approx \theta \bmod (2\pi)$$

Chapter 3

3. Motors System Identification and Control

3.1 Introduction

Representing a system mathematically is one of the most important things in our lives. In our project, we have six DC motors to move our rover and four servo motors will be used for steering. Here, we will discuss how to identify the transfer function for each motor.

But why would we do that if we use identical motors?

In experimental work, we found that the three motors are almost identical, but there are some differences among them. Also, some other external factors are acting on the motors, which we will discuss in the next sections.

3.2 Review on DC-Motor TF

We already knew the transfer function of DC-Motor is presented as shown:

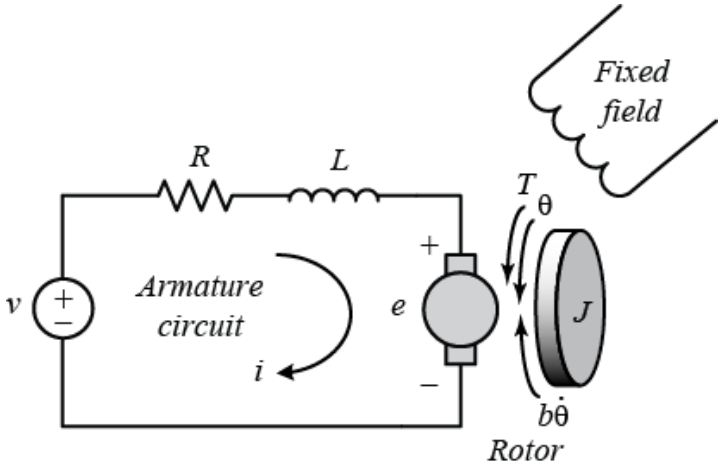


Figure 3.1 DC-Motor schematic

$$\frac{\dot{\theta}\,(s)}{V(s)} = \frac{K}{(Js+b)(Ls+R)+K^2}\ [\frac{rad/sec}{V}]$$

Where:

(J) moment of inertia of the rotor

(b) motor viscous friction constant

(Ke) electromotive force constant

(Kt) motor torque constant

(R) electric resistance

(L) electric inductance

But if we see our experimental complete setup, would this equation be enough for representing the motors?

Figure 3.2 Walle Mars Rover

It's clear that, there are many other factors that affect the motor rotation, such as, friction and load distributions. That's why we tend to use the MATLAB to describe our systems.

3.3 Transfer function Estimation using MATLAB

In this section, we discuss in details how to use MATLAB and the requirements to get your transfer function like magic.

3.3.1 Steps

a) Experiment setting.
b) Use known inputs and get output readings.
c) Use Inputs and outputs in MATLAB system-identification toolbox.
d) Analyze the results.
e) Repeat the experiment to get the best results.

Experiment setting

For convenience, we will discuss the setting on one motor only.

Components:

- 12 V DC geared motor: 5000 rpm reduced to 83 rpm to get much torque.
- Encoders: as a senor of motor rpm.
- Monster Motor Shield VNH2SP30: control the motor speed through Arduino.
- 2 Arduino Boards: microcontroller to control the motor and get readings from encoders.

- 12 V lipo battery: to supply power for different components.
- 2 RF module: to make wireless connection between the motors and the Laptop.
- Jumper wires: for different connections.

Figure 3.3 Dc Geared motor

Figure 3.4 encoders

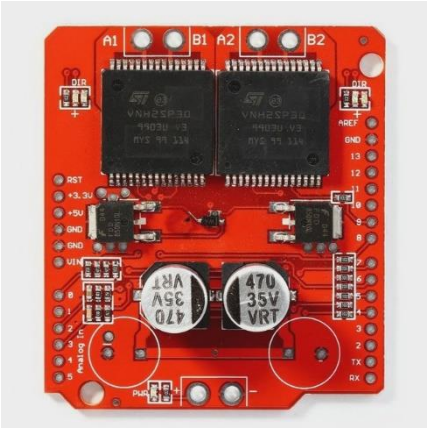


Figure 3.5 Monster motor shield (H-bridge)

Figure 3.6 Arduino uno

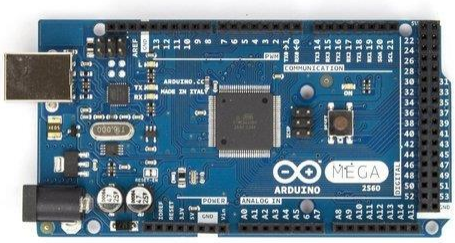


Figure 3.7 Arduino Mega

Figure 3.8 lipo battery

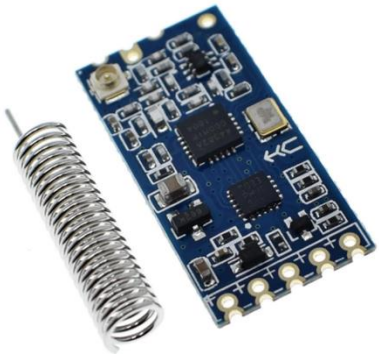


Figure 3.9 RF module

Figure 3.10 jumper wires

Note that: we need Arduino Mega to connect all six motors and RF module at the same time and another Arduino to connect it with the other RF module to get the reading to our computer without direct wired connections.

Connections:

First step: we connect the motor to the H-bridge according to the data sheet given:

No.	Colour	Description
VCC		Power Supply
GND		Ground
A0		Enable for motor 1
A1		Enable for motor 2
A2		Current sensor for motor 1
A3		Current sensor for motor 2
D7		Clockwise for motor 1
D8		Counterclockwise for motor 1
D4		Clockwise for motor 2
D9		Counterclockwise for motor 2
D5		PWM for motor 1
D6		PWM for motor 2

Figure 3.11 Monster shield H-bridge connection guide

We wouldn't use current sensor pins as there is no need for them in that experiment.

Second step: we connect the Arduino Mega with the motor encoder pins, noting that: you should use interrupt pins in Arduino Mega (2,3,18,19,20,21).

Third step: we connect the RF module to Arduino mega to transmit reading that we will get from the motor.

Fourth step: we connect the RF module to Arduino uno that is connected to our Laptop to receive readings.

Fifth step: upload the code to Arduino mega to get ready for executing when the power is supplied.

Finally: we connect the battery to the motor and Arduino mega.

Inputs and Outputs

In our first experiment we used PWM input for the motors as a sin wave with absolute value

$$(\quad PWM = 255 * |\sin(2\pi f * no.of\ iterations * T_{sampling})| \quad)$$

We tried many sampling times, and the best results were between 400 and 500 msec.

As System-Identification needs much readings for getting the best results, so, we used 500 input samples and got 500 readings out of the encoders.

ENCODER READINGS:

There are several different types of rotary encoders. I'm restricting this discussion to the simpler "incremental" encoders. These are sometimes called *quadrature* or *relative* rotary encoders.

These encoders have two sensors and output two sets of pulses. The sensors, which can be magnetic (hall effect) or light (LED or Laser), produce pulses when the encoder shaft is rotated.

As there are two sensors in two different positions, they will both produce the same pulses, however, they will be out of phase as one sensor will pulse before the other one. Which sensor goes first is determined by the direction of rotation.

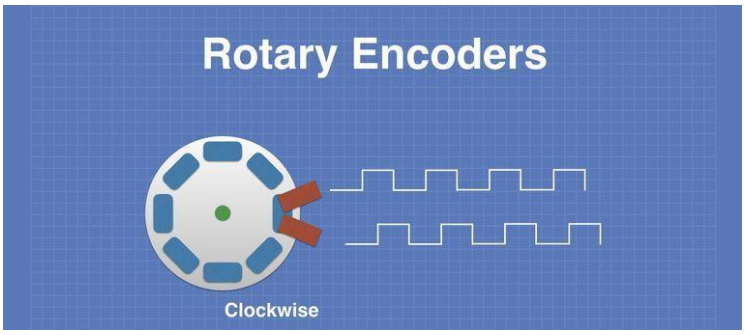


Figure 3.12 the encoder shaft is spinning clockwise. The sensor on the top is triggered before the bottom one, so the top set of pulses precedes the bottom set.

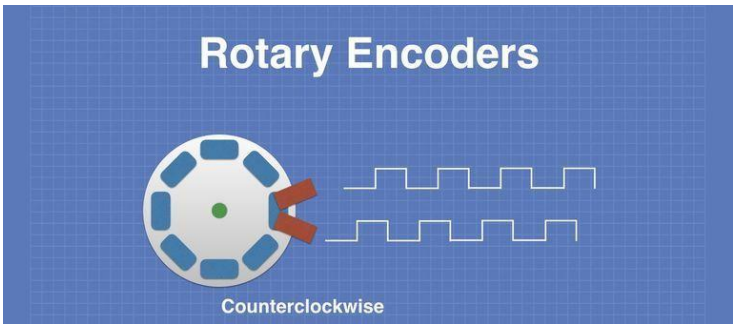


Figure 3.13 the encoder shaft is rotated counterclockwise then the bottom set of pulses will be delivered before the top set.

In our case, the encoder is mounted on the side before the gearbox, so, the maximum speed is 5000 rpm. That's why we needed to calibrate the readings before using them in Identification process, using the next relation:

$$rpm = \frac{encodercounts}{Time\ of\ reading\ in\ msec}\frac{60}{1000}\frac{1}{number\ of\ encoder\ disk\ spots}\frac{1}{60.24(gear\ ratio)}$$

Now, we have our Inputs and outputs. We are ready for the system Identification.

3.3.2 System Identification

Steps:

- Open MATLAB
- Upload your Inputs and Outputs to the workspace.

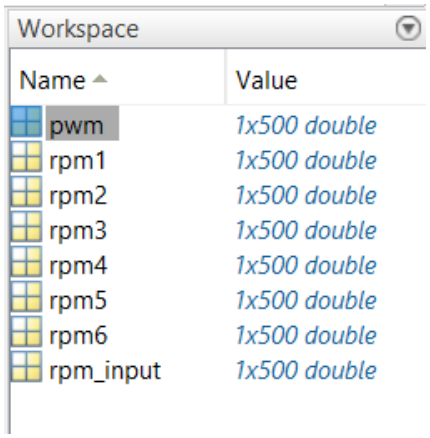


Figure 3.14 arrays of input and 6 motors outputs

- Remap the input PWM to rpm, so that we have both input and outputs having the same units.

$$rpm_{input} = pwm * \frac{83(max.vel)}{255(8bit)}$$

- Start system Identification tool using "systemIdentification" command.
- Click on Import data and choose, Time domain data; as shown:

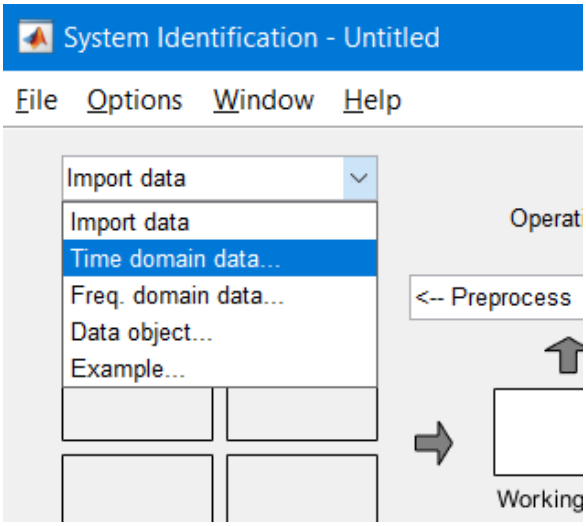


Figure 3.15 Importing data

- Write your input and output as shown in your workspace, and make the starting time equals zero and write the sampling time used in your experiment. Then click import.

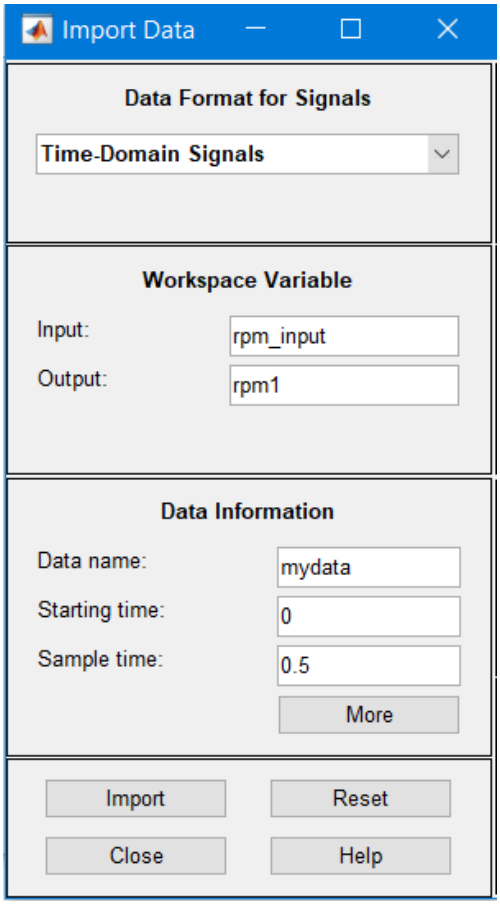


Figure 3.16 importing data and specifying starting and sampling time

- Choose Quick start as shown.

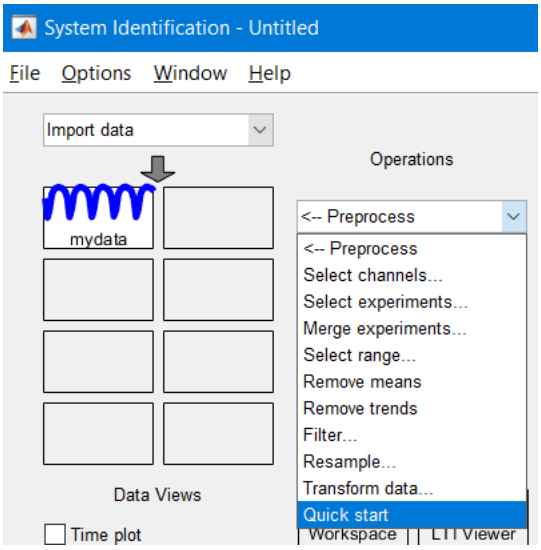


Figure 3.17 Quick start

Get different functions and choose the proper one

- Click estimate transfer function models and choose no. of poles and zeroes. Also, you can add time delay to it.

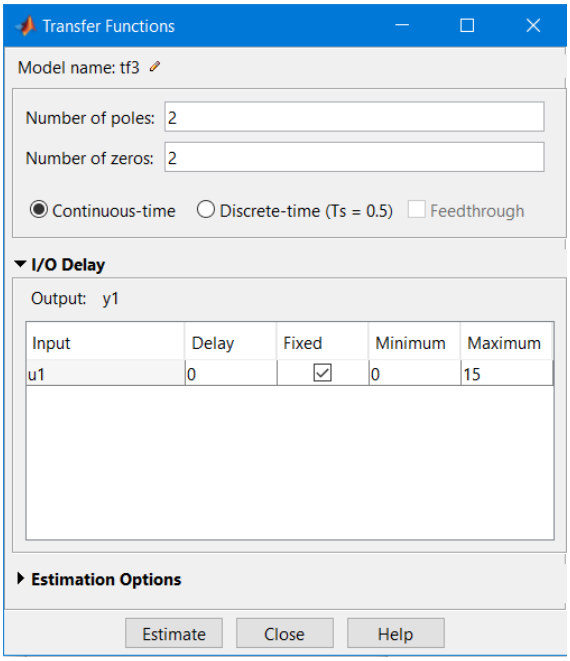


Figure 3.18 Estimating transfer function

- Using the checkbox Model output, you can see the best fit of your trials and choose from them. Double click on each tf in the shown box to see each transfer function estimated.

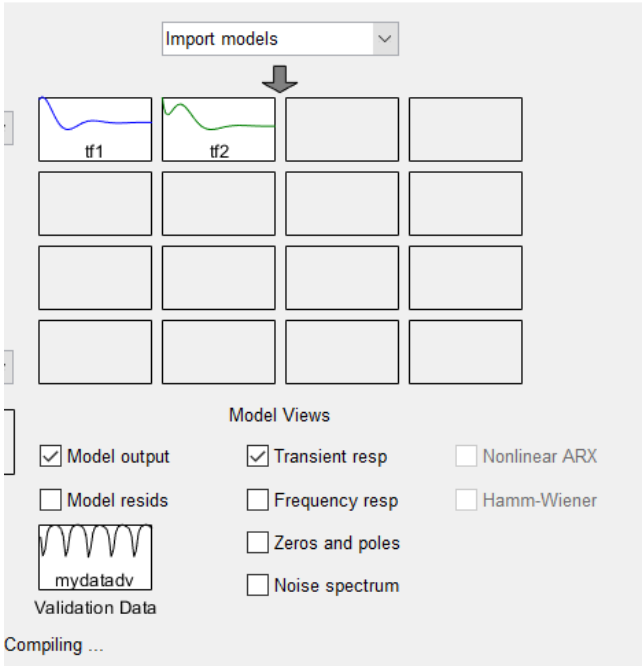


Figure 3.19 outputs

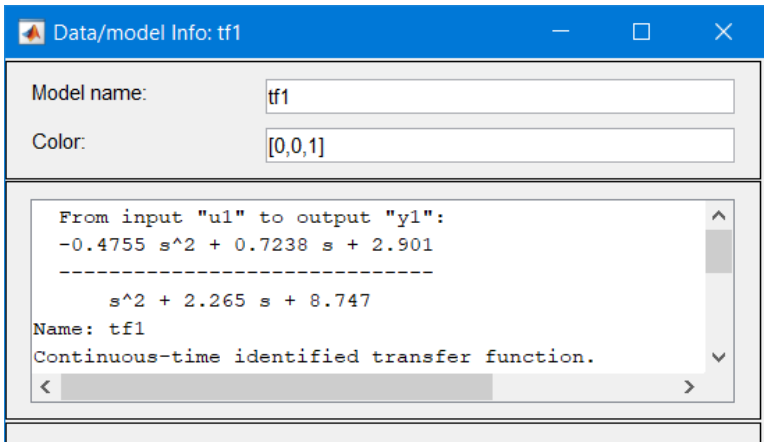


Figure 3.20 Transfer function

- You can compare between different TF from their Model output. Also, you can see the transient response of each one and compare them.

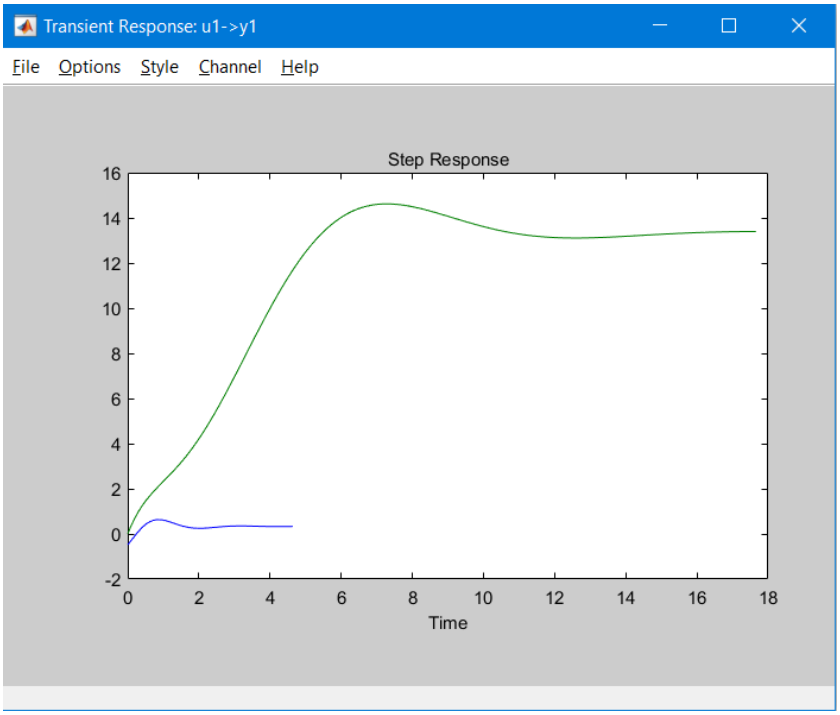


Figure 3.21 Step response

Note that: each response stops after reaching steady state in this step response representation.

Repeating the experiment

For the best results, we should repeat this experiment more times to get the best results. In our case, we will do this step after the rover become completely ready and all its components mounted.

3.4 Results

Motor 1

Best fit: 87 %

$$\frac{3.45s^2 + 0.2214\,s + 7.162}{s^3 + 2.321\,s^2 + 2.831\,s + 6.57}$$

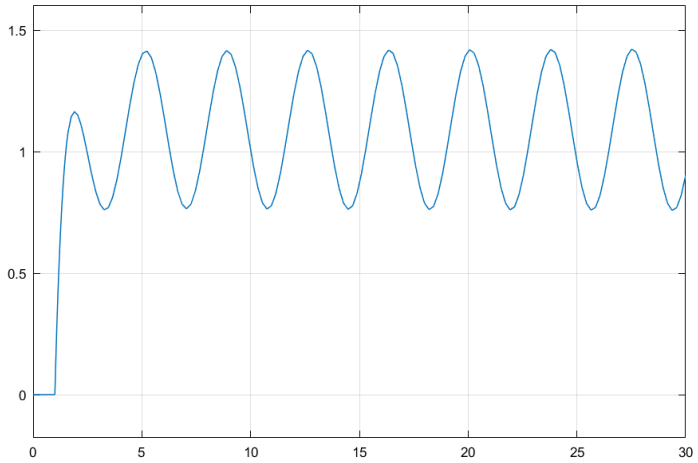


Figure 3.22Step response

Best fit: 73%

$$\frac{3.32s^2 + 2.431}{s^3 + 1.117\,s^2 + 8.276\,s + 2.233}$$

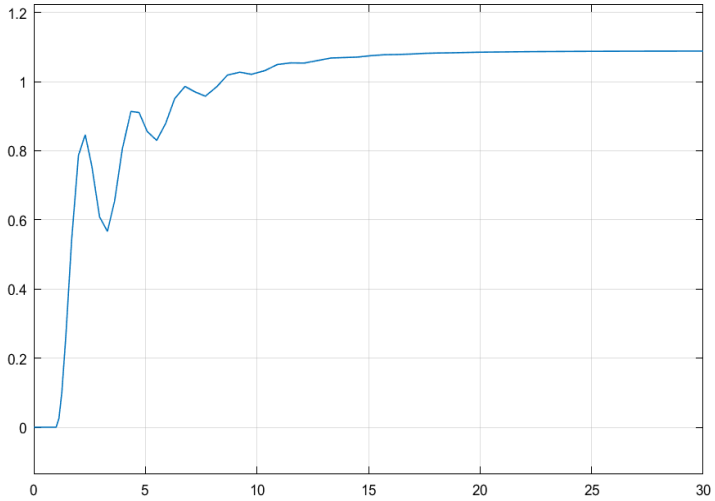


Figure 3.23 Step response

Motor 2

Best fit: 86%

$$\frac{3.119s^2 + 0.1959\,s + 6.611}{s^3 + 2.211\,s^2 + 2.854\,s + 6.29}$$

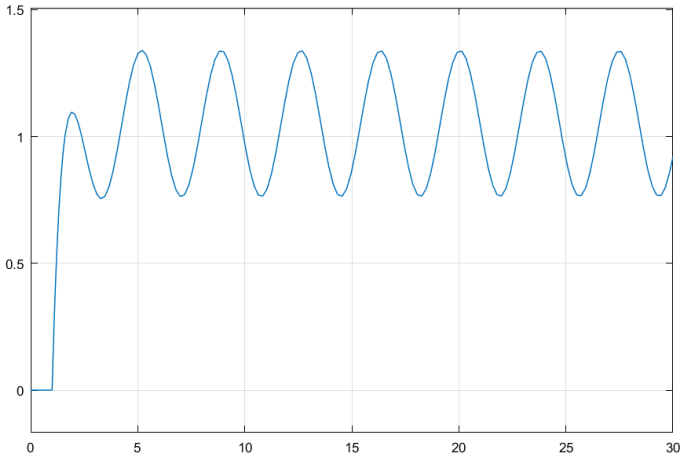


Figure 3.24 Step response

Best fit: 75%

$$\frac{6.017\ s + 4.357}{s^3 + 2{,}159\ s^2 + 12.94\ s + 4.148}$$

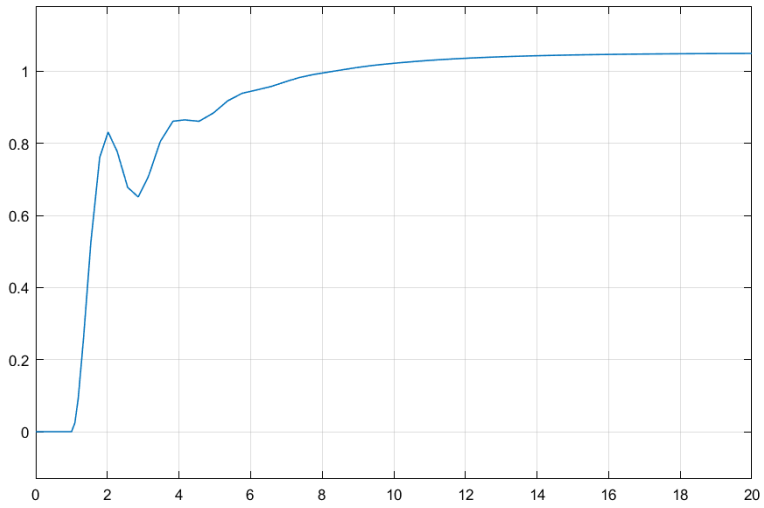


Figure 3.25 Step response

Motor 3

Best fit: 87%

$$\frac{3.361s^2 + 0.2292\ s + 7.01}{s^3 + 2.248\ s^2 + 2.876\ s + 6.465}$$

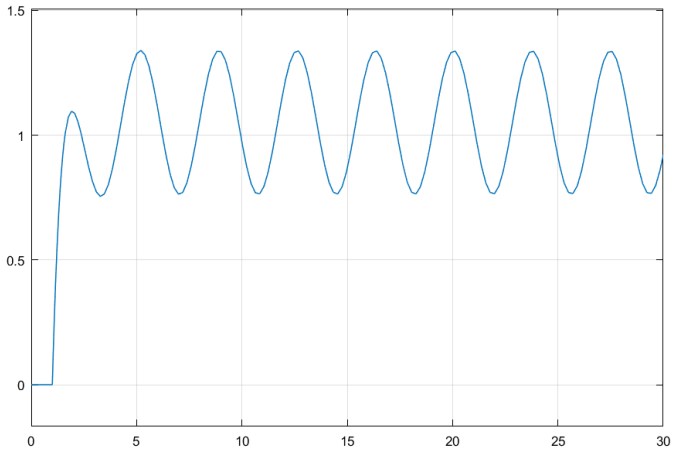


Figure 3.26 Step response

Best fit: 73 %

$$\frac{3.212\ s + 2.397}{s^3 + 1.091\ s^2 + 8.175\ s + 2.213}$$

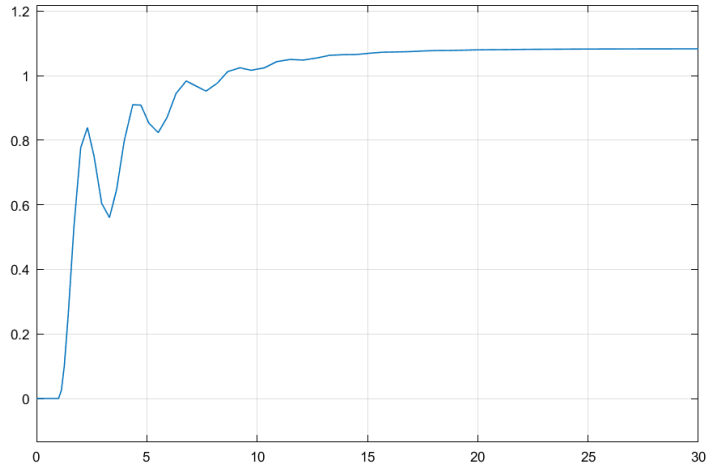


Figure 3.27 Step response

Motor 4

Best fit: 87 %

$$\frac{3.569s^2 + 0.1706\,s + 7.26}{s^3 + 2.47\,s^2 + 2.718\,s + 6.714}$$

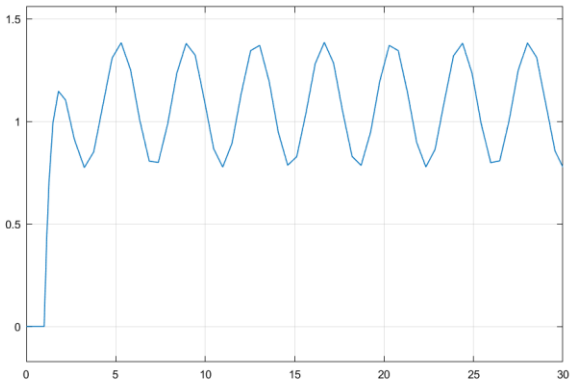


Figure 3.28 Step response

Best fit: 73%

$$\frac{3.558\,s + 2.502}{s^3 + 1.2\,s^2 + 8.521\,s + 2.316}$$

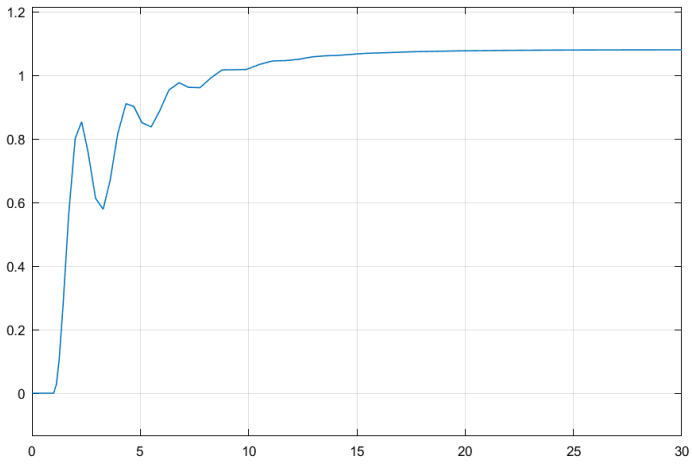


Figure 3.29 Step response

Motor 5

Best fit: 85%

$$\frac{3.336s^2 + 0.174\,s + 6.887}{s^3 + 2.169\,s^2 + 2.883\,s + 6.254}$$

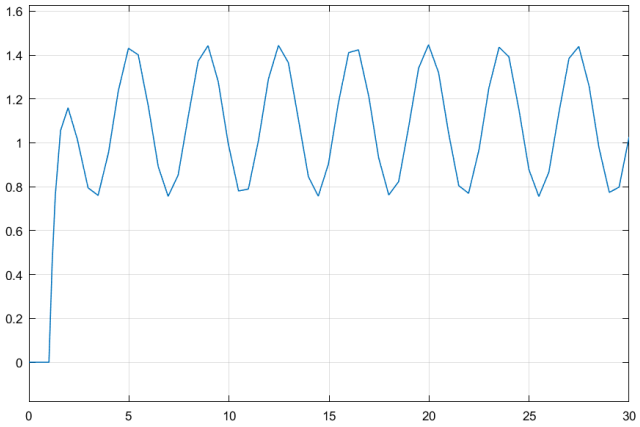


Figure 3.30 Step response

Best fit: 73 %

$$\frac{3.209\,s + 2.471}{s^3 + 1.073\,s^2 + 8.157\,s + 2.245}$$

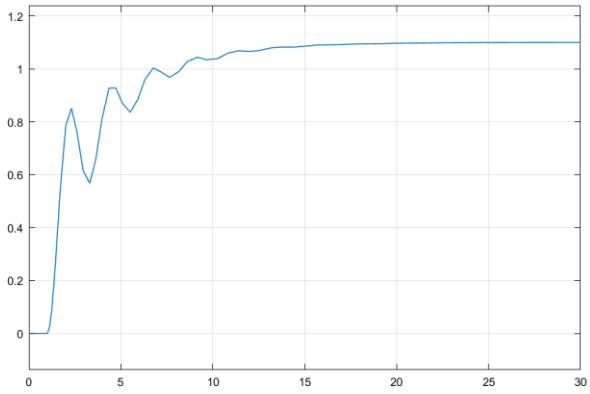


Figure 3.31 Step response

Motor 6

Best fit: 87%

$$\frac{3.864\,s^2 + 0.1733\,s + 7.848}{s^3 + 2.676\,s^2 + 2.641\,s + 7.068}$$

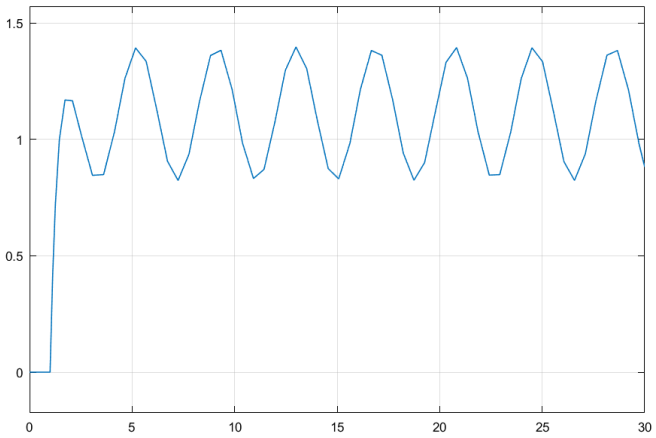


Figure 3.32 Step response

Best fit: 73%

$$\frac{3.862\,s + 2.615}{s^3 + 1.276\,s^2 + 8.708\,s + 2.359}$$

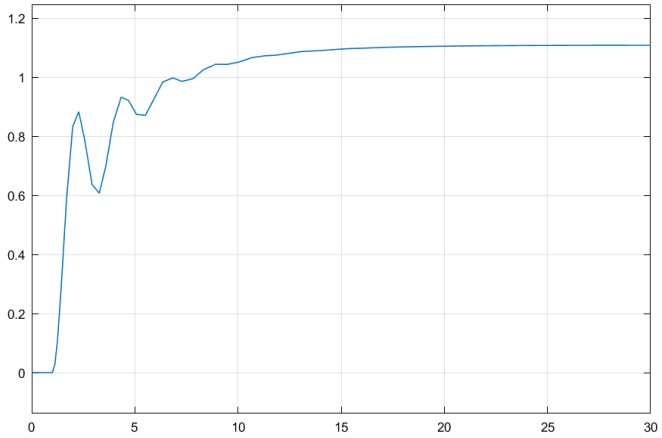


Figure 3.33 Step response

3.5 Controller Design

3.5.1 Discussion

This Part distinguishes between PID controllers for 3 kinds of transfer functions for the same DC motor but with different Fit to the real model

Here we choose motor number one to be our model so we did system identification for it and get three transfer functions with the following fits to real model

- 87%

$$T = \frac{3.45s^2 + 0.2214\,s + 7.162}{s^3 + 2.321\,s^2 + 2.831\,s + 6.57}$$

- 73%

$$\mathrm{T} = \frac{3.32s^2 + 2.431}{s^3 + 1.117\,s^2 + 8.276\,s + 2.233}$$

- 67%

$$T = \frac{6.227}{s^2 + 2.949\,s + 11.23}$$

we perform Simulink model to simulate the transfer function in closed loop form with unity feed-back and PID controller

then using PID tuner to estimate the gains Kp, Ki and Kd

here we show the results of each model

3.5.2 Results

87% TF

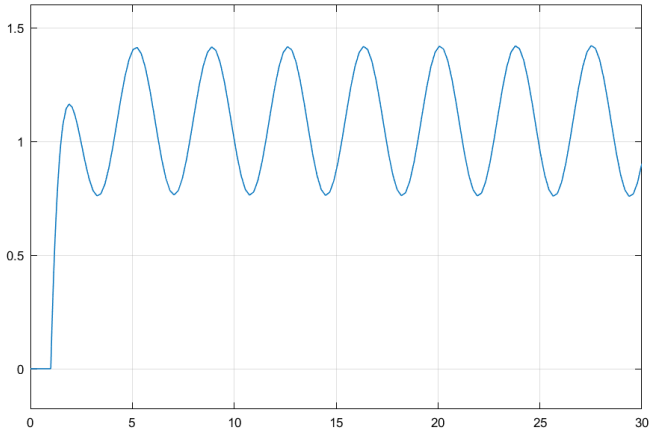


Figure 3.34 Open Loop Response

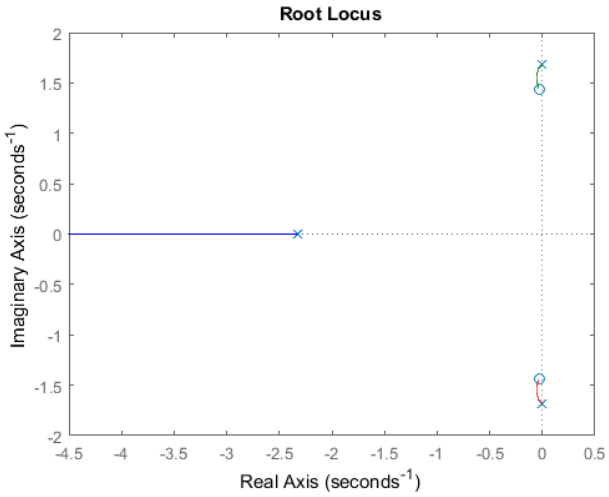


Figure 3.35 root locus

From root locus and open loop response we could see that it's critically stable.

The following figure show the response after put PID controller:

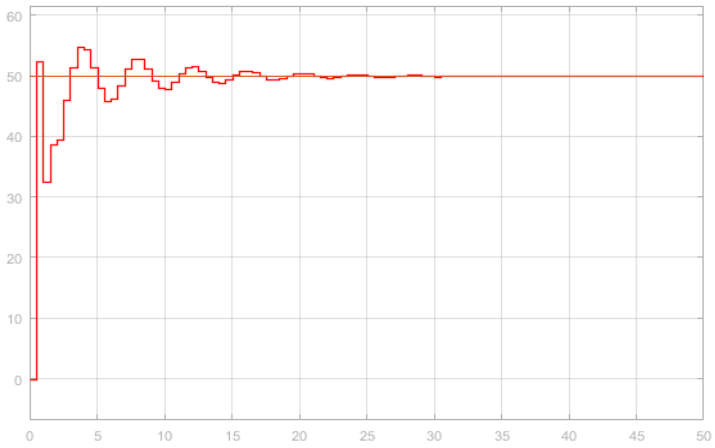


Figure 3.36 Closed Loop Response

The controller gains are

Kp	1.04781660677059
Ki	1.06497416859211
Kd	0

The Response parameters

Rise time	0
Settling time	14.5
Overshoot	13.6%

Conclusion

It seems that the PID controller achieves advance with the stability of the system

73% TF

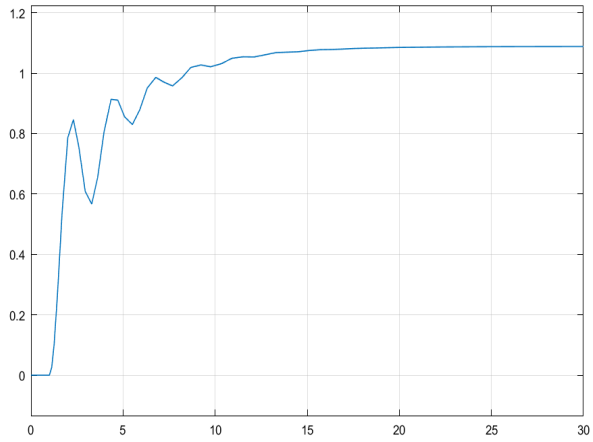


Figure 3.37 Open loop Response

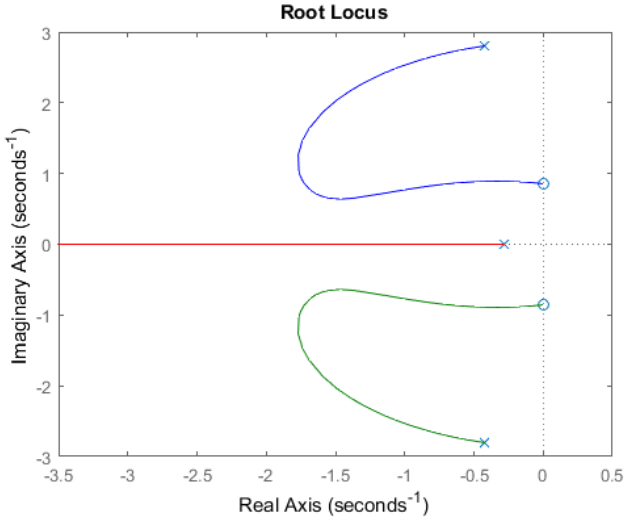


Figure 3.38 Root locus

It seems to be more stable than the previous TF.

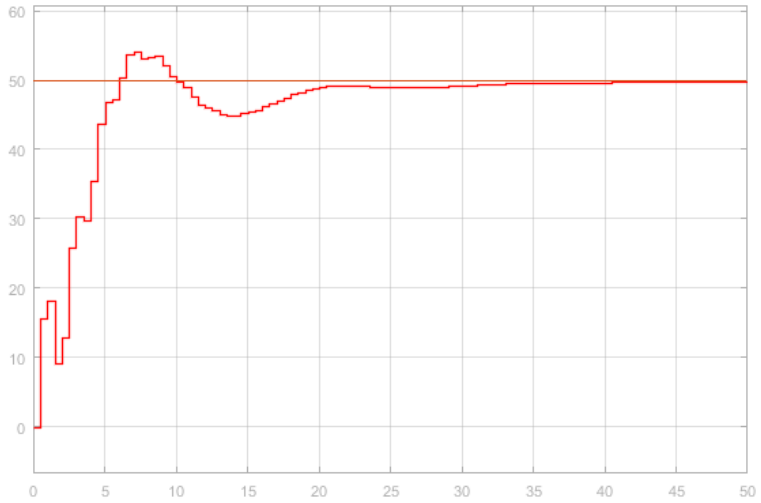


Figure 3.39 Closed Loop Response

The controller gains are

Kp	1.8931035044146
Ki	0.255532213735752
Kd	4.4831607785217

The Response parameters

Rise time	4.5
Settling time	27.5
Overshoot	8.12%

67% TF

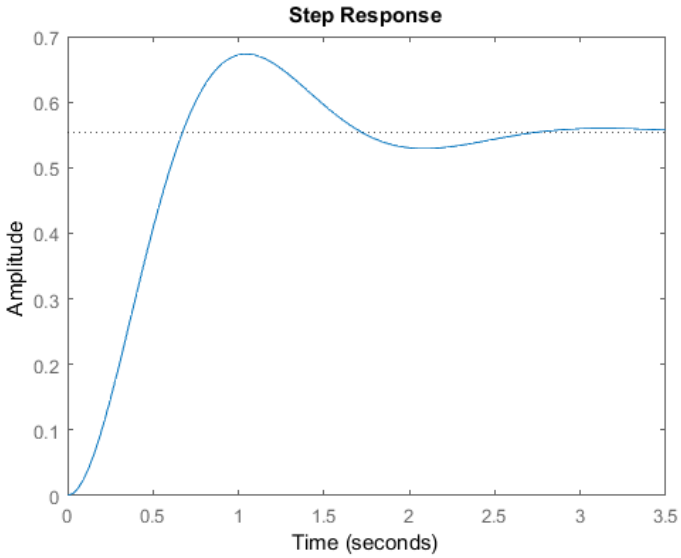


Figure 3.40 Open loop response

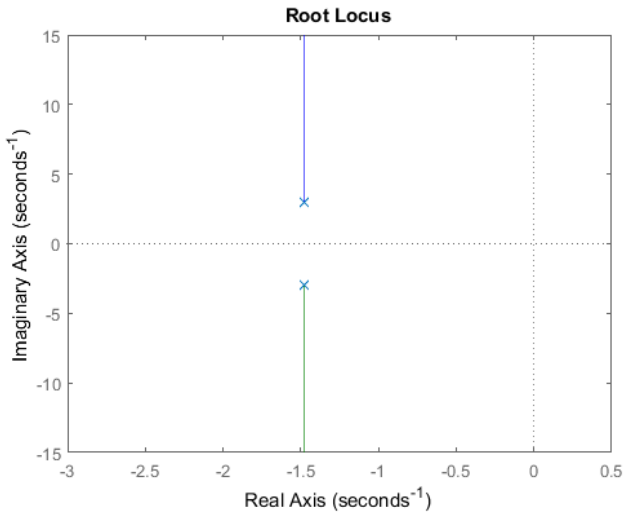


Figure 3.41 Root locus

Stable system

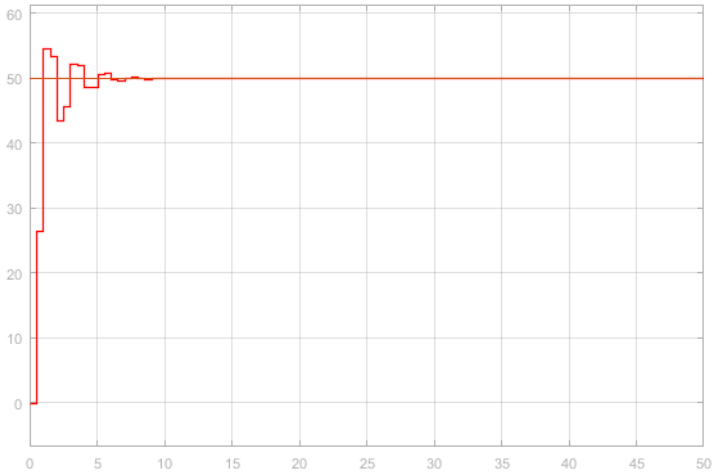


Figure 3.42Closed Loop Response

The controller gains are

Kp	1.29916627119768
Ki	2.43875307724585
Kd	0

The Response parameters

Rise time	0.5
Settling time	5
Overshoot	9%

Chapter 4

4. Introduction to ROS (Robot Operating System)

4.1 Introduction to ROS

4.1.1 Overview

ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management.

- A software framework for programming robots
- Prototypes originated from Stanford AI research, officially created and developed by Willow Garage starting in 2007
- Currently maintained by Open Source Robotics Foundation
- Consists of infrastructure, tools, capabilities, and ecosystem

Distro	Release date	Poster	*Tuturtle*, turtle in tutorial	EOL date
ROS Melodic Morenia (**Recommended**)	May 23rd, 2018			May, 2023 (Bionic EOL)
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame	May 23rd, 2016			April, 2021 (Xenial EOL)
ROS Jade Turtle	May 23rd, 2015			May, 2017
ROS Indigo Igloo	July 22nd, 2014			April, 2019 (Trusty EOL)
ROS Hydro Medusa	September 4th, 2013			May, 2015
ROS Groovy Galapagos	December 31, 2012			July, 2014

Figure 4.1 ROS distributions

Its usefulness is not limited to robots, but the majority of tools provided are focused on working with peripheral hardware. So, fundamentally, you can think of it as a collection of software, but there are other components as well. Modular design helps to reduce code complexity in comparison to monolithic systems. A system which is distributed in this manner is more tolerant to faults as crash at one node does not result in crash of the whole system. What is more, implementation details are hidden as nodes expose minimal API.

This also enables communication of nodes written in different programming languages or running on different machines which are connected over network. ROS does not replace your computer operating system. It is a collection of software that runs on top of the operating system you are running on your computer. Usually that will be Ubuntu Linux.

ROS currently only runs on Unix-based platforms. Software for ROS is primarily tested on Ubuntu and Mac OS X systems.

ROS Consists of the following components

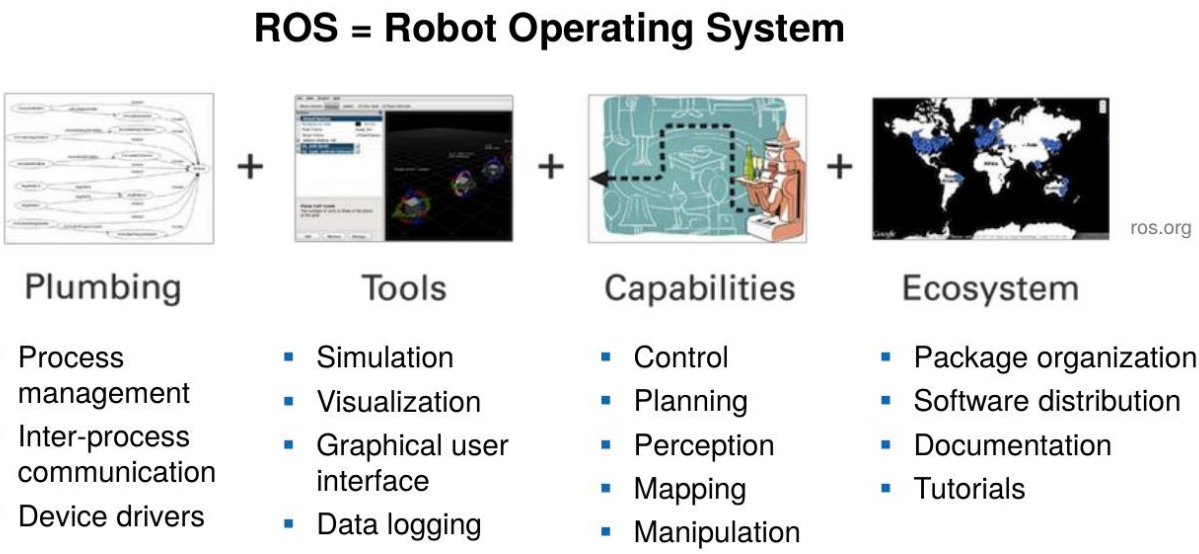


Figure 4.2The ROS Equation

- Plumping

It gives you a way to organize and a way to manage your code. But also, very importantly, it provides communication between multiple pieces of code that are all running.

A robotic system has many components and all of those components end up having to talk to each other. So actually, a robot is made up of lots of individual components in hardware, and you can think about that in software as well.

- **Tools**

Introspection, you can always look and see what data is being passed around between your nodes. Visualization tools, the visualizer itself, RVIZ, which is a tremendously powerful tool which can visualize lots of types of data that's relevant in robotics, Also it provides interfacing with Gazebo Simulator through gazebo ROS pkgs.

Many other examples are on the ROS Wiki. ROS gives you many tools for debugging and for, in general, making your robot project move forward.

- **Capabilities**

ROS is split up in more than 2000 packages, each package providing specialized functionality. The number of tools connected to the framework are probably its biggest power. It provides functionality for hardware abstraction, device drivers, communication between processes over multiple machines, tools for testing and visualization, and much more.

- **Ecosystem**

ROS has a tremendous number of users. It has seen huge acceptance, especially in the research community. There are hundreds and even thousands of labs that are actively using ROS, contributing code and developing new packages. It's seeing more acceptance in industry, as well. There is a very active support forum for Q&A at answers.ros.org.

4.1.2 General Concepts

ROS has three levels of concepts: the Filesystem level, the Computation Graph level, and the Community level. We are going to mention the first two concepts, for more details about the latter, check the official ROS Wiki.

ROS Filesystem Level

The filesystem level concepts mainly cover ROS resources that may be encountered on disk, such as:

- **Packages**: Packages are the main unit for organizing software in ROS. A package may contain ROS runtime processes (nodes), a ROS-dependent library, datasets, configuration files, or anything else that is usefully organized together.
- **Metapackages**: Metapackages are specialized Packages which only serve to represent a group of related other packages. It is a specific type of package which has no content, just dependencies. Package Manifests: Manifests (package.xml) provide metadata about a package, including its name, version, description, license information, dependencies, and other meta information like exported packages.
- **Message** (msg) types: Message descriptions, stored in my_package/msg/MyMessageType.msg, define the data structures for messages sent in ROS.
- **Service** (srv) types: Service descriptions, stored in my_package/srv/MyServiceType.srv, define the request and response data structures for services in ROS.

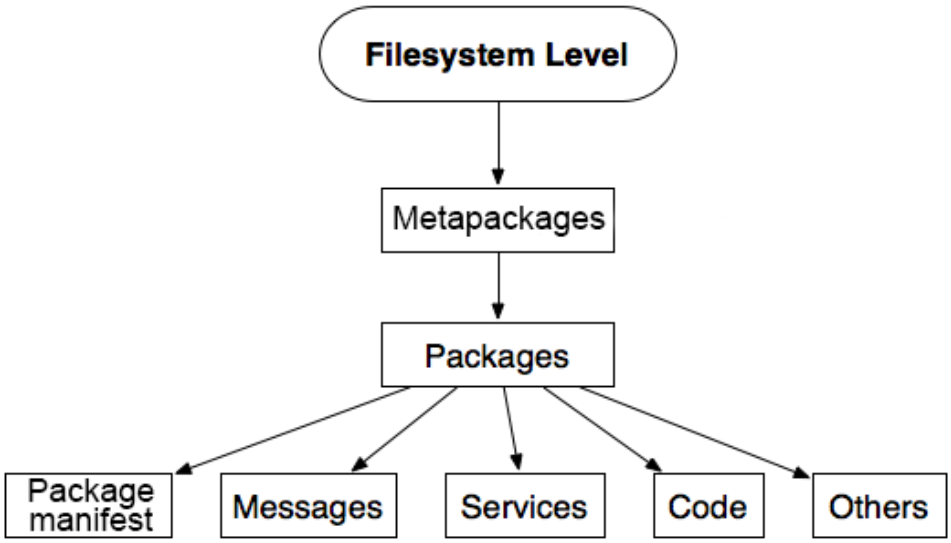


Figure 4.3:ROS Filesystem Level

ROS Computation Graph Level

The Computation Graph is the peer-to-peer network of ROS processes that are processing data together. This is how things are connected in ROS. The basic Computation Graph concepts of ROS are nodes, Master, Parameter Server, messages, services, topics, and bags, all of which provide data to the Graph in different ways.

Nodes: Nodes are processes that perform computation. ROS is designed to be very modular. So, a robot control system usually comprises many nodes. For example, one node controls a laser range-finder, one node controls the wheel motors, one node performs localization, one node performs path planning. A ROS node is written with the use of a ROS client library, such as roscpp or rospy.

Master: The ROS Master provides name registration and lookup to the rest of the Computation Graph. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services.

Parameter Server: The Parameter Server allows data to be stored by key in a central location. It is currently part of the Master.

Messages: Nodes communicate with each other by passing messages. A message is simply a data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays (much like C structs).

Topics: Messages are routed via a transport system with publish / subscribe semantics. A node sends out a message by publishing it to a given topic. The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. Logically, one can think of a topic as a strongly typed message bus. Each bus has a name, and anyone can connect to

the bus to send or receive messages as long as they are the right type. The idea is to decouple the production of information from its consumption

Services: The publish / subscribe model is a very flexible communication paradigm, but its many-to-many, one-way transport is not appropriate for request / reply interactions, which are often required in a distributed system. Request / reply is done via services, which are defined by a pair of message structures: one for the request and one for the reply. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply.

Bags: Bag files are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data, such as sensor data, that can be difficult to collect but is necessary for developing and testing algorithms.

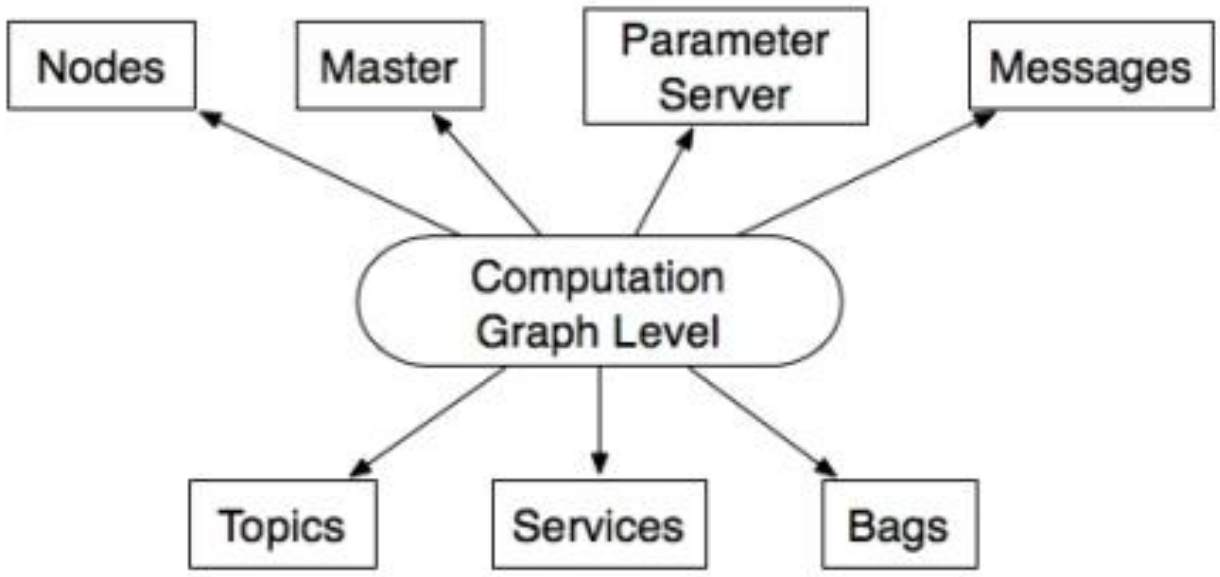


Figure 4.4: ROS Computation Graph Level

Base unit in ROS is called a node. Nodes are in charge of handling devices or computing algorithms- each node for separate task. Nodes connect to other nodes directly; the Master only provides lookup information, much like a DNS server. Nodes that subscribe to a topic will request connections from nodes that publish that topic, and will establish that connection over an agreed upon connection protocol. The most common protocol used in a ROS is called TCPROS, which uses standard TCP/IP sockets. Single package is usually developed for performing one type of task and can contain one or multiple nodes. For More explanation how to write it check the ROS Wiki

The ROS Master acts as a server in the ROS Computation Graph. It stores topics and services registration information for ROS nodes. Nodes communicate with the Master to report their registration information. As these nodes communicate with the Master, they can receive information about other registered nodes and make

connections as appropriate. The Master will also make callbacks to these nodes when this registration information changes, which allows nodes to dynamically create connections as new nodes are run. Nodes must know network address of master on startup (ROS_MASTER_URI)

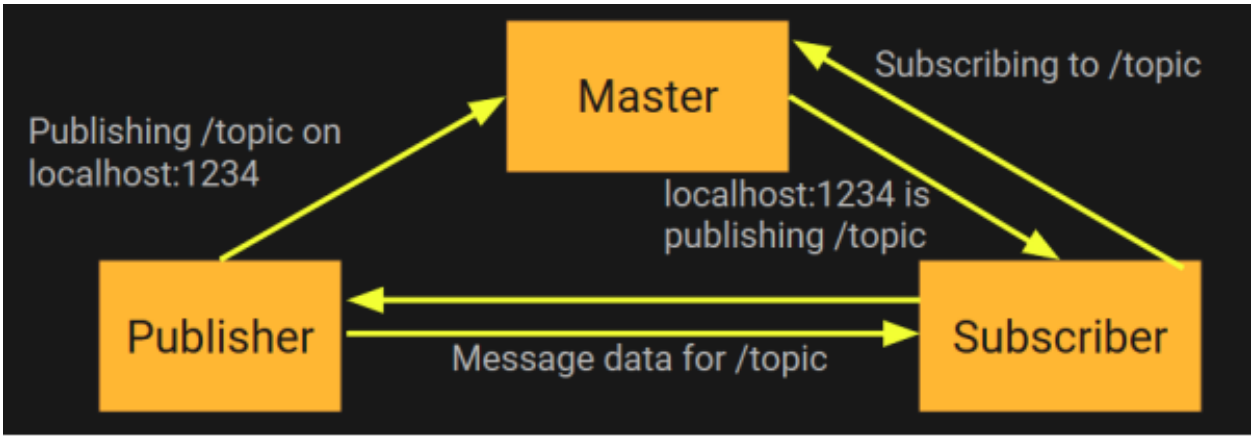


Figure 4.5: ROS Master connecting two nodes that are talking and listening to each other

4.1.3 Installation and Configuration (ROS Melodic)

Configure your Ubuntu repositories

Configure your Ubuntu repositories to allow "restricted," "universe," and "multiverse." You can follow the Ubuntu guide for instructions on doing this.

Setup your sources.list

Setup your computer to accept software from packages.ros.org.

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

Set up your keys (might be different, check official ROS Wiki)

```
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key
C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

If you experience issues connecting to the keyserver, you can try substituting `hkp://pgp.mit.edu:80` or `hkp://keyserver.ubuntu.com:80` in the previous command.

4.1.4 Installation

First, make sure your Debian package index is up-to-date:

```
sudo apt update
```

There are many different libraries and tools in ROS. We provided four default configurations to get you started. You can also install ROS packages individually.
In case of problems with the next step, you can use following repositories instead of the ones mentioned above ros-shadow-fixed

- Desktop-Full Install: (Recommended) : ROS, rqt, rviz, robot-generic libraries, 2D/3D simulators and 2D/3D perception

```
sudo apt install ros-melodic-desktop-full

sudo apt install ros-melodic-PACKAGE

sudo apt install ros-melodic-slam-gmapping
```

To find available packages, use:

```
apt search ros-melodic
```

Initialize rosdep

Before you can use ROS, you will need to initialize rosdep. rosdep enables you to easily install system dependencies for source you want to compile and is required to run some core components in ROS.

```
sudo rosdep init

rosdep update
```

Environment setup

It's convenient if the ROS environment variables are automatically added to your bash session every time a new shell is launched:

```
echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc

source ~/.bashrc
```

If you have more than one ROS distribution installed, ~/.bashrc must only source the setup.bash for the version you are currently using.
If you just want to change the environment of your current shell, instead of the above you can type:

```
source /opt/ros/melodic/setup.bash
```

Dependencies for building packages

Up to now you have installed what you need to run the core ROS packages. To create and manage your own ROS workspaces, there are various tools and requirements that are distributed separately. For example, rosinstall is a frequently used command-line tool that enables you to easily download many source trees for ROS packages with one command.

To install this tool and other dependencies for building ROS packages, run:

```
sudo apt install python-rosinstall python-rosinstall-generator python-wstool build-essential
```

4.1.5 Navigating the ROS Filesystem

Quick Overview of Filesystem Concepts

A quick overview of the above mentioned concepts

- Packages: Packages are the software organization unit of ROS code. Each package can contain libraries, executables, scripts, or other artifacts.
- Manifests (package.xml): A manifest is a description of a package. It serves to define dependencies between packages and to capture meta information about the package like version, maintainer, license, etc...

Filesystem Tools

Code is spread across many ROS packages. Navigating with command-line tools such as ls and cd can be very tedious which is why ROS provides tools to help you.

Using rospack

rospack allows you to get information about packages. In this tutorial, we are only going to cover the find option, which returns the path to package.

Usage

```
$ rospack find [package_name]
```

Using roscd

roscd is part of the rosbash suite. It allows you to change directory (cd) directly to a package or a stack.

Usage

```
$ roscd [locationname[/subdir]]
```

To verify that we have changed to the roscpp package directory, run this example:

```
$ roscd roscpp
```

Now let's print the working directory using the Unix command pwd:

Note that roscd, like other ROS tools, will only find ROS packages that are within the directories listed in your ROS_PACKAGE_PATH. To see what is in your ROS_PACKAGE_PATH, type:

```
$ echo $ROS_PACKAGE_PATH
```

Your ROS_PACKAGE_PATH should contain a list of directories where you have ROS packages separated by colons. A typical ROS_PACKAGE_PATH might look like this:

```
/opt/ros/kinetic/base/install/share
```

Similarly to other environment paths, you can add additional directories to your ROS_PACKAGE_PATH, with each path separated by a colon ':'

Subdirectories

roscd can also move to a subdirectory of a package or stack.

Try:

```
$ roscd roscpp/cmake
```

roscd log

roscd log will take you to the folder where ROS stores log files. Note that if you have not run any ROS programs yet, this will yield an error saying that it does not yet exist.

If you have run some ROS program before, try:

```
$ roscd log
```

Using rosls

rosls is part of the rosbash suite. It allows you to ls directly in a package by name rather than by absolute path.

Usage:

```
$ rosls [locationname[/subdir]]
```

Review

You may have noticed a pattern with the naming of the ROS tools:

- rospack = ros + pack(age)
- roscd = ros + cd
- rosls = ros + ls

This naming pattern holds for many of the ROS tools.

4.1.6 ROS Package

What makes up a catkin Package?

For a package to be considered a catkin package it must meet a few requirements:

- The package must contain a catkin compliant package.xml file.
    - That package.xml file provides meta information about the package.
- The package must contain a CMakeLists.txt which uses catkin.
    - If it is a catkin metapackage it must have the relevant boilerplate CMakeLists.txt file.
- Each package must have its own folder
    - This means no nested packages nor multiple packages sharing the same directory.

The simplest possible package might have a structure which looks like this:

```
my_package/
  CMakeLists.txt
  package.xml
```

Packages in a catkin Workspace

The recommended method of working with catkin packages is using a catkin workspace, but you can also build catkin packages standalone. A trivial workspace might look like this:

```
workspace_folder/        -- WORKSPACE
  src/                   -- SOURCE SPACE
    CMakeLists.txt       -- 'Toplevel' CMake file, provided by catkin
    package_1/
      CMakeLists.txt     -- CMakeLists.txt file for package_1
      package.xml        -- Package manifest for package_1
    ...
    package_n/
```

```
CMakeLists.txt     -- CMakeLists.txt file for package_n

package.xml        -- Package manifest for package_n
```

Before continuing with this tutorial create an empty catkin workspace by following the Creating a workspace for catkin tutorial.

Creating a catkin Package

This tutorial will demonstrate how to use the catkin_create_pkg script to create a new catkin package, and what you can do with it after it has been created.

First change to the source space directory of the catkin workspace you created in the Creating a Workspace for catkin tutorial:

```
# You should have created this in the Creating a Workspace Tutorial

$ cd ~/catkin_ws/src
```

Now use the catkin_create_pkg script to create a new package called 'beginner_tutorials' which depends on std_msgs, roscpp, and rospy:

```
$ catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
```

This will create a beginner_tutorials folder which contains a package.xml and a CMakeLists.txt, which have been partially filled out with the information you gave catkin_create_pkg.

catkin_create_pkg requires that you give it a package_name and optionally a list of dependencies on which that package depends:

```
# This is an example, do not try to run this

# catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

catkin_create_pkg also has more advanced functionalities which are described in catkin/commands/catkin_create_pkg.

Building a catkin workspace and sourcing the setup file

Now you need to build the packages in the catkin workspace:

```
$ cd ~/catkin_ws

$ catkin_make
```

After the workspace has been built it has created a similar structure in the devel subfolder as you usually find under

/opt/ros/$ROSDISTRO_NAME.

To add the workspace to your ROS environment you need to source the generated setup file:

```
$ . ~/catkin_ws/devel/setup.bash
```

4.1.7 Package Dependencies

First-order dependencies

When using catkin_create_pkg earlier, a few package dependencies were provided. These first-order dependencies can now be reviewed with the rospack tool.

```
$ rospack depends1 beginner_tutorials
```

```
roscpp

rospy

std_msgs
```

As you can see, rospack lists the same dependencies that were used as arguments when running catkin_create_pkg.These dependencies for a package are stored in the package.xml file:

```
$ roscd beginner_tutorials

$ cat package.xml
```

```
<package format="2">

...

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>roscpp</build_depend>

  <build_depend>rospy</build_depend>

  <build_depend>std_msgs</build_depend>

...

</package>
```

Indirect dependencies

In many cases, a dependency will also have its own dependencies. For instance, rospy has other dependencies.

```
$ rospack depends1 rospy
```

```
genpy

.....
```

A package can have quite a few indirect dependencies. Luckily rospack can recursively determine all nested dependencies.

```
$ rospack depends beginner_tutorials

cpp_common

rostime

.....
```

Customizing Your Package

This part of the tutorial will look at each file generated by catkin_create_pkg and describe, line by line, each component of those files and how you can customize them for your package.

Customizing the package.xml

The generated package.xml should be in your new package. Now lets go through the new package.xml and touch up any elements that need your attention.

Description tag

First update the description tag

```
<description>The beginner_tutorials package</description>
```

Change the description to anything you like, but by convention the first sentence should be short while covering the scope of the package. If it is hard to describe the package in a single sentence then it might need to be broken up.

Maintainer tags

Next comes the maintainer tag:

```
<!-- One maintainer tag required, multiple allowed, one person per tag
<!-- Example:  -->
<!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->
<maintainer email="user@todo.todo">user</maintainer>
```

This is a required and important tag for the package.xml because it lets others know who to contact about the

package. At least one maintainer is required, but you can have many if you like. The name of the maintainer goes into the body of the tag, but there is also an email attribute that should be filled out:

```
<maintainer email="you@yourdomain.tld">Your Name</maintainer>
```

License tags

Next is the license tag, which is also required:

```
<!-- One license tag required, multiple allowed, one license per tag -->
<!-- Commonly used license strings: -->
<!--   BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->
<license>TODO</license>
```

You should choose a license and fill it in here. Some common open source licenses are BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, and LGPLv3. You can read about several of these at the Open Source Initiative. For this tutorial we'll use the BSD license because the rest of the core ROS components use it already:

```
<license>BSD</license>
```

Dependencies tags

The next set of tags describe the dependencies of your package. The dependencies are split into build_depend, buildtool_depend, exec_depend, test_depend. For a more detailed explanation of these tags see the documentation about Catkin Dependencies. Since we passed std_msgs, roscpp, and rospy as arguments to catkin_create_pkg, the dependencies will look like this:

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
```

All of our listed dependencies have been added as a build_depend for us, in addition to the default buildtool_depend on catkin. In this case we want all of our specified dependencies to be available at build and run time, so we'll add a exec_depend tag for each of them as well:

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
```

```
<exec_depend>roscpp</exec_depend>
<exec_depend>rospy</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

Final package.xml

As you can see the final package.xml, without comments and unused tags, is much more concise:

```
<?xml version="1.0"?>
<package format="2">
  <name>beginner_tutorials</name>
  <version>0.1.0</version>
  <description>The beginner_tutorials package</description>

  <maintainer email="you@yourdomain.tld">Your Name</maintainer>
  <license>BSD</license>
  <url type="website">http://wiki.ros.org/beginner_tutorials</url>
  <author email="you@yourdomain.tld">Jane Doe</author>

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>

  <exec_depend>roscpp</exec_depend>
  <exec_depend>rospy</exec_depend>
  <exec_depend>std_msgs</exec_depend>

</package>
```

Customizing the CMakeLists.txt

Now that the package.xml, which contains meta information, has been tailored to your package, you are ready to move on in the tutorials. The CMakeLists.txt file created by catkin_create_pkg will be covered in the later tutorials about building ROS code.

Now that you've made a new ROS package, let's build our ROS package.

Building Packages

As long as all of the system dependencies of your package are installed, we can now build your new package.

Note: If you installed ROS using apt or some other package manager, you should already have all of your dependencies.

Before continuing remember to source your environment setup file if you have not already. On Ubuntu it would be something like this:

```
# source /opt/ros/%YOUR_ROS_DISTRO%/setup.bash

$ source /opt/ros/melodic/setup.bash            # For melodic for instance
```

Using catkin_make

catkin_make is a command line tool which adds some convenience to the standard catkin workflow. You can imagine that catkin_make combines the calls to cmake and make in the standard CMake workflow.

Usage:

```
# In a catkin workspace

$ catkin_make [make_targets] [-DCMAKE_VARIABLES=...]
```

For people who are unfamiliar with the standard CMake workflow, it breaks down as follows:

Note: If you run the below commands it will not work, as this is just an example of how CMake generally works.

```
# In a CMake project

$ mkdir build

$ cd build

$ cmake ..

$ make

$ make install  # (optionally)
```

This process is run for each CMake project. In contrast catkin projects can be built together in workspaces. Building zero to many catkin packages in a workspace follows this work flow:

```
# In a catkin workspace

$ catkin_make
```

```
$ catkin_make install  # (optionally)
```

The above commands will build any catkin projects found in the src folder. This follows the recommendations set by REP128. If your source code is in a different place, say my_src then you would call catkin_make like this:

Note: If you run the below commands it will not work, as the directory my_src does not exist.

```
# In a catkin workspace

$ catkin_make --source my_src

$ catkin_make install --source my_src  # (optionally)
```

For more advanced uses of catkin_make see the documentation: catkin/commands/catkin_make

Building Your Package

If you are using this page to build your own code, please also take a look at the later tutorials (C++)/(Python) since you may need to modify CMakeLists.txt.

You should already have a catkin workspace and a new catkin package called beginner_tutorials from the previous tutorial, Creating a Package. Go into the catkin workspace if you are not already there and look in the src folder:

```
$ cd ~/catkin_ws/

$ ls src
```

```
beginner_tutorials/  CMakeLists.txt@
```

You should see that there is a folder called beginner_tutorials which you created with catkin_create_pkg in the previous tutorial. We can now build that package using catkin_make:

```
$ catkin_make
```

You should see a lot of output from cmake and then make, which should be similar to this:

```
Base path: /home/user/catkin_ws

Source space: /home/user/catkin_ws/src

Build space: /home/user/catkin_ws/build

Devel space: /home/user/catkin_ws/devel

Install space: /home/user/catkin_ws/install
```

```
.....
```

Note that catkin_make first displays what paths it is using for each of the 'spaces'. The spaces are described in the REP128 and by documentation about catkin workspaces on the wiki: catkin/workspaces. The important thing to notice is that because of these default values several folders have been created in your catkin workspace. Take a look with ls:

```
$ ls
```

- build
- devel
- src

The build folder is the default location of the build space and is where cmake and make are called to configure and build your packages. The devel folder is the default location of the devel space, which is where your executables and libraries go before you install your packages.

4.1.8 Nodes

A node really isn't much more than an executable file within a ROS package that either publishes or subscribes to one or several topics. ROS nodes use a ROS client library to communicate with other nodes. Nodes can publish or subscribe to a Topic. Nodes can also provide or use a Service.

Quick Overview of Graph Concepts

- Nodes: A node is an executable that uses ROS to communicate with other nodes.
- Messages: ROS data type used when subscribing or publishing to a topic.
- Topics: Nodes can publish messages to a topic as well as subscribe to a topic to receive messages.
- Master: Name service for ROS (i.e. helps nodes find each other)
- rosout: ROS equivalent of stdout/stderr
- roscore: Master + rosout + parameter server (parameter server will be introduced later)

Client Libraries

ROS client libraries allow nodes written in different programming languages to communicate:

- rospy = python client library

- roscpp = c++ client library

roscore

There can be only one roscore in a ROS network. roscore is the first thing you should run when using ROS unless your using a launch file, launch files start a roscore themselves.

Please run:

```
$ roscore
```

You will see something similar to:

```
... logging to ~/.ros/log/9cf88ce4-b14d-11df-8a75-00251148e8cf/roslaunch-machine_name-13039.log

Checking log directory for disk usage. This may take awhile.

.....

started core service [/rosout]
```

If roscore does not initialize, you probably have a network configuration issue or another roscore is running. See Network Setup - Single Machine Configuration

If roscore does not initialize and sends a message about lack of permissions, probably the ~/.ros folder is owned by root, change recursively the ownership of that folder with:

```
$ sudo chown -R <your_username> ~/.ros
```

Using rosnode

Open up a new terminal, and let's use rosnode to see what running roscore did... Bear in mind to keep the previous terminal open either by opening a new tab or simply minimizing it.

Note: When opening a new terminal your environment is reset and your ~/.bashrc file is sourced. If you have trouble running commands like rosnode then you might need to add some environment setup files to your ~/.bashrc or manually re-source them.

rosnode displays information about the ROS nodes that are currently running. The rosnode list command lists these active nodes:

```
$ rosnode list
```

- You will see:

- /rosout

This showed us that there is only one node running: rosout. This is always running as it collects and logs nodes' debugging output.

The rosnode info command returns information about a specific node.

```
$ rosnode info /rosout
```

This gave us some more information about rosout, such as the fact that it publishes /rosout_agg.

```
--------------------------------------------------------------------------------
Node [/rosout]
Publications:
 * /rosout_agg [rosgraph_msgs/Log]

Subscriptions:
 * /rosout [unknown type]

Services:
 * /rosout/get_loggers
 * /rosout/set_logger_level

contacting node http://machine_name:54614/ ...
Pid: 5092
```

Now, let's see some more nodes. For this, we're going to use rosrun to bring up another node.

Using rosrun

rosrun allows you to use the package name to directly run a node within a package (without having to know the package path).

Usage:

```
$ rosrun [package_name] [node_name]
```

So now we can run the turtlesim_node in the turtlesim package.

Then, in a new terminal:

```
$ rosrun turtlesim turtlesim_node
```

You will see the turtlesim window:

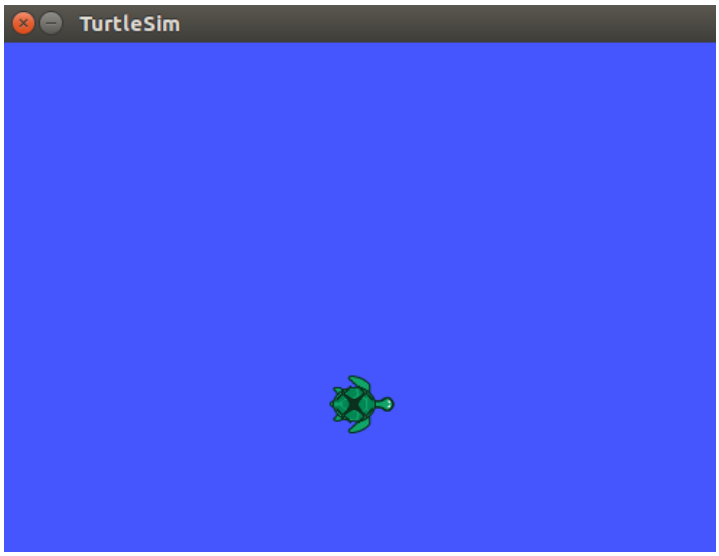


Figure 4.6: Running Turtlesim Node of Turtlesim Package

NOTE: The turtle may look different in your turtlesim window. Don't worry about it - there are many types of turtle and yours is a surprise!

Then, in a new terminal:

```
$ rosrun turtlesim turtle_teleop_key
```

This node drives the turtle around with arrow keys, i.e. it publishes messages of the type geometry_msgs/Twist on the /cmd_vel topic. The turtle itself subscribes to /cmd_vel topic

You should see

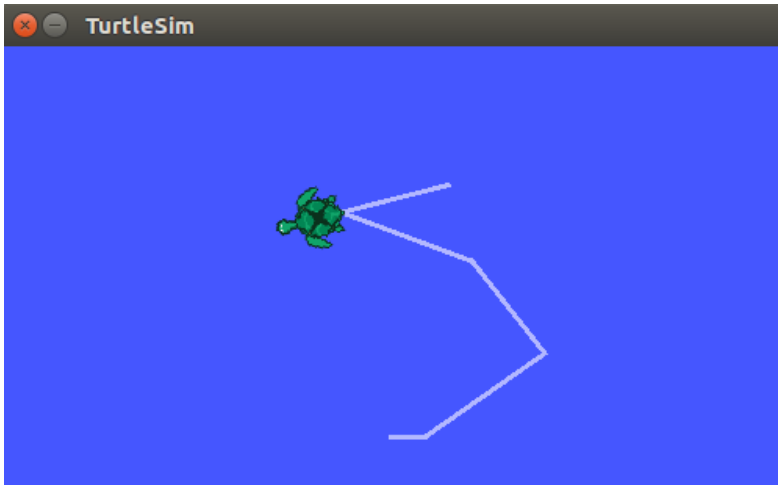


Figure 4.7: Driving around the turtle with turtle_teleop Package

Review

What was covered:

- roscore = ros+core : master (provides name service for ROS) + rosout (stdout/stderr) + parameter server (parameter server will be introduced later)
- rosnode = ros+node : ROS tool to get information about a node.
- rosrun = ros+run : runs a node from a given package.

4.1.9 ROS Topics

The turtlesim_node and the turtle_teleop_key node are communicating with each other over a ROS Topic. turtle_teleop_key is publishing the key strokes on a topic, while turtlesim subscribes to the same topic to receive the key strokes. Let's use rqt_graph which shows the nodes and topics currently running.

Note: If you're using electric or earlier, rqt is not available. Use rxgraph instead.

Using rqt_graph

rqt_graph creates a dynamic graph of what's going on in the system. rqt_graph is part of the rqt package. Unless you already have it installed, run:

```
$ sudo apt-get install ros-<distro>-rqt
```

```
$ sudo apt-get install ros-<distro>-rqt-common-plugins
```

replacing <distro> with the name of your ROS distribution (e.g. indigo, jade, kinetic, lunar ...)

In a new terminal:

```
$ rosrun rqt_graph rqt_graph
```

You will see something similar to:

Figure 4.8: rqt_graph showing two nodes running, publishing msgs on /turtle1/command_velocity and the other node subscribes to the msgs being sent

If you place your mouse over /turtle1/command_velocity it will highlight the ROS nodes (here blue and green) and topics (here red). As you can see, the turtlesim_node and the turtle_teleop_key nodes are communicating on the topic named /turtle1/command_velocity.

4.1.10 ROS Messages

Communication on topics happens by sending ROS messages between nodes. For the publisher (turtle_teleop_key) and subscriber (turtlesim_node) to communicate, the publisher and subscriber must send and receive the same type of message. This means that a topic type is defined by the message type published on it. The type of the message sent on a topic can be determined using rostopic type.

Using rostopic type

rostopic type returns the message type of any topic being published.

Usage:

```
rostopic type [topic]
```

For ROS Hydro and later,

```
$ rostopic type /turtle1/cmd_vel
```

You should get

```
geometry_msgs/Twist
```

We can look at the details of the message using rosmsg:

```
$ rosmsg show geometry_msgs/Twist
```

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

4.1.11 ROS Services

Services are another way that nodes can communicate with each other. Services allow nodes to send a **request** and receive a **response**.

Using rosservice

rosservice can easily attach to ROS's client/service framework with services. rosservice has many commands that can be used on services, as shown below:

Usage:

```
rosservice list         print information about active services
rosservice call         call the service with the provided args
rosservice type         print service type
rosservice find         find services by service type
```

```
rosservice uri        print service ROSRPC uri
```

rosservice list

```
$ rosservice list
```

The list command shows us that the turtlesim node provides nine services: reset, clear, spawn, kill, turtle1/set_pen, /turtle1/teleport_absolute, /turtle1/teleport_relative, turtlesim/get_loggers, and turtlesim/set_logger_level. There are also two services related to the separate rosout node: /rosout/get_loggersand /rosout/set_logger_level.

```
/clear

/kill

/reset

....
```

Let's look more closely at the clear service using rosservice type:

rosservice call

Usage:

```
rosservice call [service] [args]
```

Here we'll call with no arguments because the service is of type empty:

```
$ rosservice call /clear
```

This does what we expect, it clears the background of the turtlesim_node.

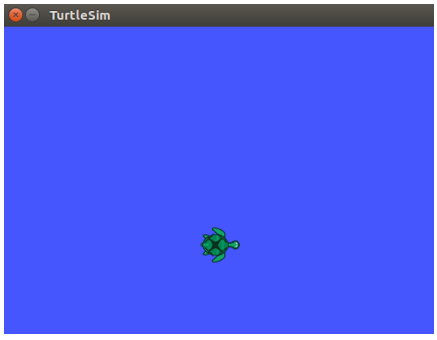


Figure 4.9: Turtlesim node after calling /clear service

Using rosparam

rosparam allows you to store and manipulate data on the ROS Parameter Server. The Parameter Server can store integers, floats, boolean, dictionaries, and lists. rosparam uses the YAML markup language for syntax. In simple cases, YAML looks very natural: 1 is an integer, 1.0 is a float, one is a string, true is a boolean, [1, 2, 3] is a list of integers, and {a: b, c: d} is a dictionary. rosparam has many commands that can be used on parameters, as shown below:

Usage:

```
rosparam set            set parameter
rosparam get            get parameter
rosparam load           load parameters from file
rosparam dump           dump parameters to file
rosparam delete         delete parameter
rosparam list           list parameter names
```

Let's look at what parameters are currently on the param server:

Using roslaunch

roslaunch starts nodes as defined in a launch file.

```
$ roslaunch [package] [filename.launch]
```

First go to the beginner_tutorials package we created and built earlier:

```
$ roscd beginner_tutorials
```

If roscd says something similar to ROSCD: NO SUCH PACKAGE/STACK 'BEGINNER_TUTORIALS' , you will need to source the environment setup file like you did at the end of the create_a_workspace tutorial

```
$ cd ~/catkin_ws
$ source devel/setup.bash
$ roscd beginner_tutorials
```

Then let's make a launch directory

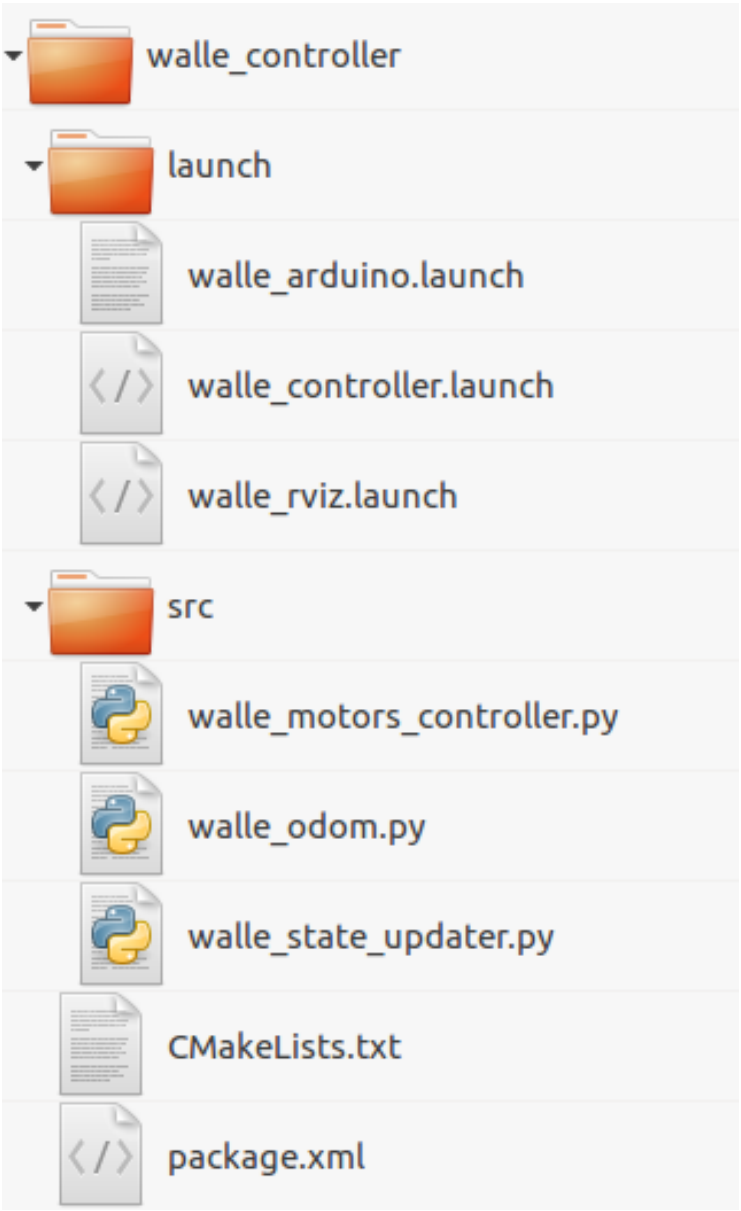
```
$ mkdir launch
$ cd launch
```

4.1.12 Real Examples

Important Note: XML is widely used throughout ROS, which is very easy to catch up with. XML (Extended Markup Language) is somewhat similar to HTML, but in XML you can use whatever tags you want and there's a parser program that takes these files and extract the information from them

Wall-E Controller Package

This packages contains the inverse kinematics model, PWM output to motors (via rosserial package) and a small control loop that feeds the wheel encoder reading back and produces control action to motors to achieve the desired speed



Launch files

Here we can launch multiple nodes at once, no need to start a roscore

Nodes

Here is where the real work is done, these nodes are platform-specific, which means it's not easy to use some other similar package to do the same job, you have to write them yourself

One example for such nodes is walle_motors_controllers.py which includes the inverse kinematics model for the rover

As for CmakeLists.txt and package.xml, these files are what distinguish a folder from being a catkin package or just some regular folder and they have been extensively explained in the previous sections. In the following text, we will be exposed to one of each of the above files

walle_controller.launch

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

  <group ns="walle_controller">
    <node pkg="walle_controller" type="walle_odom.py" name="walle_odom" output="screen">
      <param name="rate" value="10" />
      <param name="robot_wheel_separation_distance" value="0.14" />
      <param name="robot_wheel_radius" value="0.03" />
      <param name="robot_wheel_ticks" value="20" />
      <param name="frame_id" value="/odom" />
      <param name="child_frame_id" value="base_link" />
    </node>

    <node pkg="diff_drive_controller" type="diff_drive_controller.py" name="diff_drive_controller"
output="screen">
      <param name="rate" value="10" />
      <param name="timeout_idle" value="50" />
      <param name="robot_wheel_separation_distance" value="0.14" />
      <param name="robot_wheel_radius" value="0.03" />
    </node>
    <node pkg="walle_controller" type="walle_motors_controller.py" name="walle_motors_controller"
output="screen">
      <param name="rate" value="10" />
      <param name="Kp" value="0.1" />
      <param name="Ki" value="0.05" />
      <param name="Kd" value="0.001" />
      <!-- Max angular vel computed by counting the number of encoder ticks over some time.
```

```
      rad/s = # ticks * rad/tick / dt
       -->
    <param name="motor_max_angular_vel" value="3.00" />
    <param name="motor_min_angular_vel" value="0.01" />
    <param name="motor_cmd_max" value="255" />
    <param name="motor_cmd_min" value="10" />
    <param name="robot_wheel_radius" value="0.03" />
    <param name="pid_on" value="True" />
    <!-- <param name="gopigo_on" value="False" /> -->
  </node>

  <node pkg="walle_controller" type="walle_state_updater.py" name="walle_state_updater" output="screen">
    <param name="rate" value="10" />
    <param name="err_tick_incr" value="5" />
    <param name="robot_wheel_radius" value="0.03" />
    <!-- <param name="gopigo_on" value="False" /> -->
  </node>
 </group>
</launch>
```

This is a real launch file that we are using in our package, for more details about launch files check the official ROS Wiki for launch files. We are going to walk through the launch file assuming you have checked the ROS Wiki.

First off, the <node> tags is used to call a ROS node either from an official ROS package or a package that you have created yourself. This tag has attributes (pkg, type, name, output, etc. …) and elements <param> and sometimes <remap> which will be mentioned later on.

```
<node pkg="walle_controller" type="walle_odom.py" name="walle_odom" output="screen">
    <param name="rate" value="10" />
.......
    <param name="frame_id" value="/odom" />
    <param name="child_frame_id" value="base_link" />
  </node>
```

Here we are calling a node named walle_odom.py which lies in a package named walle_controller, inside of the ROS network, this node will be named “walle_odom”, and it outputs to terminal.

This node takes several parameters, each of which is passed as follows

<param name=” ” value=” “/>

In the source code of the node, this parameter is used by the function

```
rospy.get_param("~param name", default value)
```

The default value is used in case the parameter wasn't passed in the launch file. These parameters should correspond to meaningful things in your code.

Next is the ROS nodes, writing a publisher node, subscriber node or a publisher and subscriber node should be straightforward as it's the same procedure followed throughout the ROS network either using C++ or Python so it's best to check the ROS Wiki. After checking the ROS Wiki, following up with any node should be a straightforward task provided that you know the task of this node.

Wall-E Navigation Package

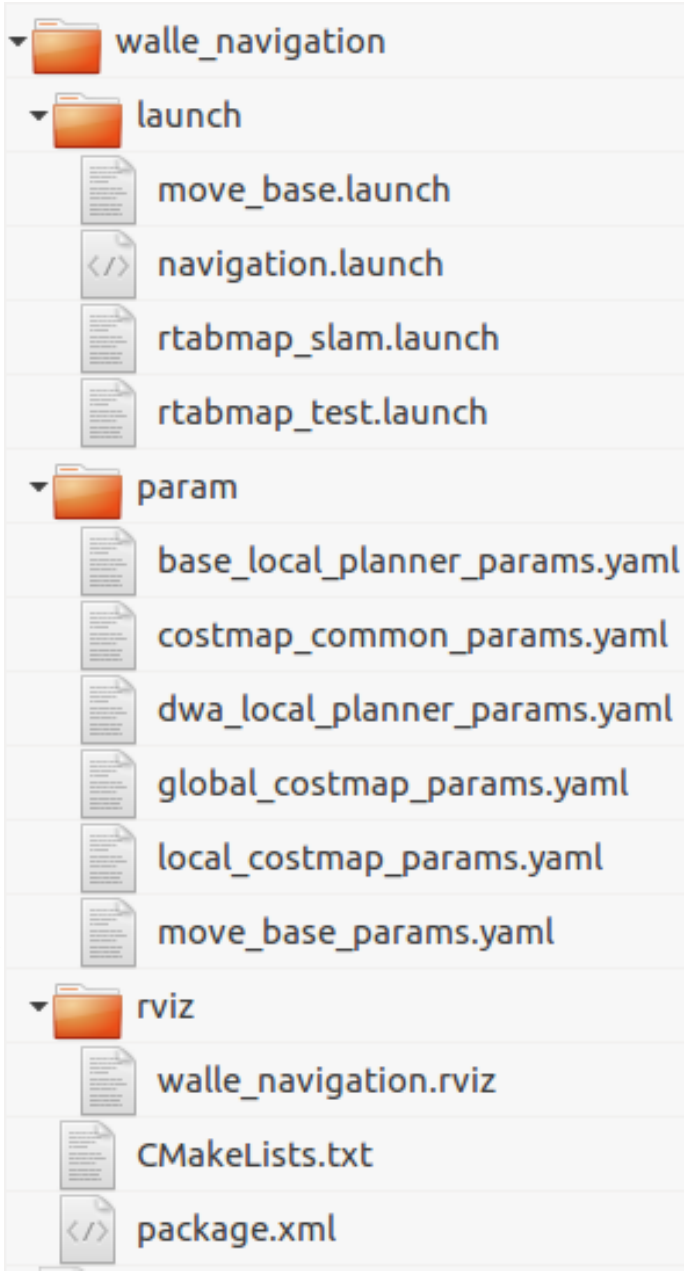
This package contains the autonomous navigation system for driving the robot in an unknown environment by creating a map and then sending a goal point to which the robot will autonomously navigate on a generated trajectory.

Launch files

Here we can launch multiple nodes at once, no need to start a roscore

Parameters

.yaml files include params for packages that use these params instead of writing all the params in the launch file



The launch files in this package follow the same procedure as the last one except that here we are using official ROS packages and passing the parameters to them but the parameters are tuned for our case. This tuning process shouldn't be an easy task.

Here, we can see .yaml files, this is another way of passing parameters to a launch file, it's used here as there are so many parameters and writing all of them in launch file would make the launch file no readable at all. So, they are all typed down in .yaml files which are loaded in the launch file.

costmap_common_params.yaml

```
obstacle_range: 5.0
raytrace_range: 6.0
max_obstacle_height: 1.0
min_obstacle_height: 0.05
# track_unknown_space: true
footprint: [[-0.26,0.35], [0.26,0.35], [0.26,-0.4], [-0.26,-0.4]]
#robot_radius: 0.105
inflation_radius: .4
footprint_padding: 0
transform_tolerance: 2.0
cost_scaling_factor: 100.0
map_type: costmap
observation_sources: scan point_cloud
scan: {sensor_frame: base_scan, data_type: LaserScan, topic: /scan, marking: true, clearing: true}
point_cloud: {sensor_frame: base_scan, data_type: PointCloud2, topic: /camera/depth/points, marking: true,
clearing: true}
```

These parameters are loaded in the launch file to be used by move_base package afterwards for path planning.

There is no src folder here as we are using official ROS packages, these packages are mentioned in detail in the following chapter.

The .rviz file is a configuration file for RViz which is used to view multiple topics without using the "Add" button in RViz to save time. This file can be generated by saving the configuration when exiting RViz and after adding all the desired topics.

4.1.13 Robotic Simulators

Overview

A robotics simulator is used to create application for a physical robot without depending on the actual machine, thus saving cost and time. In some case, these applications can be transferred onto the physical robot (or rebuilt) without modifications.

The term robotics simulator can refer to several different robotics simulation applications. For example, in mobile robotics applications, behavior-based robotics simulators allow users to create simple worlds of rigid objects and light sources and to program robots to interact with these worlds.

Comparison

Modern simulators tend to provide the following features:

- Fast robot prototyping
- Physics engines for realistic movements. Most simulators use ODE (Gazebo, LpzRobots, Marilou, Webots) or PhysX (Microsoft Robotics Studio, 4DV-Sim).
- Realistic 3d rendering. Standard 3d modeling tools or third party tools can be used to build the environments.
- Dynamic robot bodies with scripting. C, C++, Perl, Python, Java, URBI, MATLAB languages used by Webots, Python used by Gazebo.

The following table shows a comparison among various above the mentioned simulators developed by either open-source community or commercial corporations

Software	Main programming language	Formats support	Extensibility	External APIs	Robotics middleware support	Primary user interface	Headless simulation
Actin	C++	SLDPRT, SLDASM, STEP, OBJ, STL, 3DS, Collada, VRML, URDF, XML, ECD, ECP, ECW, ECX, ECZ,	Plugins (C++), API	Unknown	ROS	GUI	Yes (ActinRT)
ARS	Python	Unknown	Python	Unknown	None	Unknown	Unknown
AUTOMAPPPS	C++, Python	STEP, IGES, STL	Plugins (C), API	XML, C	Socket	GUI	Yes
Gazebo	C++	SDF[6]/URDF[7], OBJ, STL, Collada	Plugins (C++)	C++	ROS, Player, Sockets (protobuf messages)	GUI	Yes
MORSE	Python	Unknown	Python	Python[8]	Sockets, YARP, ROS, Pocolibs, MOOS	Command-line	Yes[9]
OpenHRP	C++	VRML	Plugins (C++), API	C/C++, Python, Java	OpenRTM-aist	GUI	Unknown
RoboDK	Python	SLDPRT, SLDASM, STEP, OBJ, STL, 3DS, Collada, VRML, URDF, Rhinoceros_3D, ...	API[10], Plug-In Interface[11]	Python, C/C++, C#, Matlab, ...	Socket	GUI	Unknown
SimSpark	C++, Ruby	Ruby Scene Graphs	Mods (C++)	Network (sexpr)	Sockets (sexpr)	GUI, Sockets	Unknown
V-Rep	LUA	OBJ, STL, DXF, 3DS, Collada,[12] URDF[7][13]	API, Add-ons, Plugins	C/C++, Python, Java, Urbi, Matlab/Octave	Sockets, ROS	GUI	Yes[14]
Webots	C++	WBT, VRML, X3D	API, PROTOs, Plugins (C/C++)	C, C++, Python, Java, Matlab, ROS	Sockets, ROS, NaoQI	GUI	Yes[15]
4DV-Sim	C++	3DS, OBJ, Mesh	Plugins (C++), API	FMI/FMU, Matlab	ROS, Sockets, Plug & Play interfaces[16]	GUI	Yes
OpenRAVE	C++, Python	XML, VRML, OBJ, Collada	Plugins (C++), API	C/C++, Python, Matlab	Sockets, ROS, YARP	GUI, Sockets	Yes
Software	**Main programming language**	**Formats support**	**Extensibility**	**External APIs**	**Robotic middleware support**	**Primary user interface**	**Headless simulation**

4.1.14 Gazebo

Gazebo development began in the fall of 2002 at the University of Southern California. The original creators were Dr. Andrew Howard and his student Nate Koenig. The concept of a high-fidelity simulator stemmed from the need to simulate robots in outdoor environments under various conditions. As a complementary simulator to Stage, the name Gazebo was chosen as the closest structure to an outdoor stage. The name has stuck despite the fact that most users of Gazebo simulate indoor environments. So, Gazebo is a 3D dynamic simulator with the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. While similar to game engines, Gazebo offers physics simulation on a much higher degree of fidelity, a suite of sensors, and interfaces for both users and programs.

Typical uses of Gazebo include:

- Testing robotics algorithms
- Designing robots
- Performing regression testing with realistic scenarios

A few key features of Gazebo include:

- Multiple physics engines,
- A rich library of robot models and environments,
- A wide variety of sensors,
- Convenient programmatic and graphical interfaces

GUI

The graphical user interface shown as follow

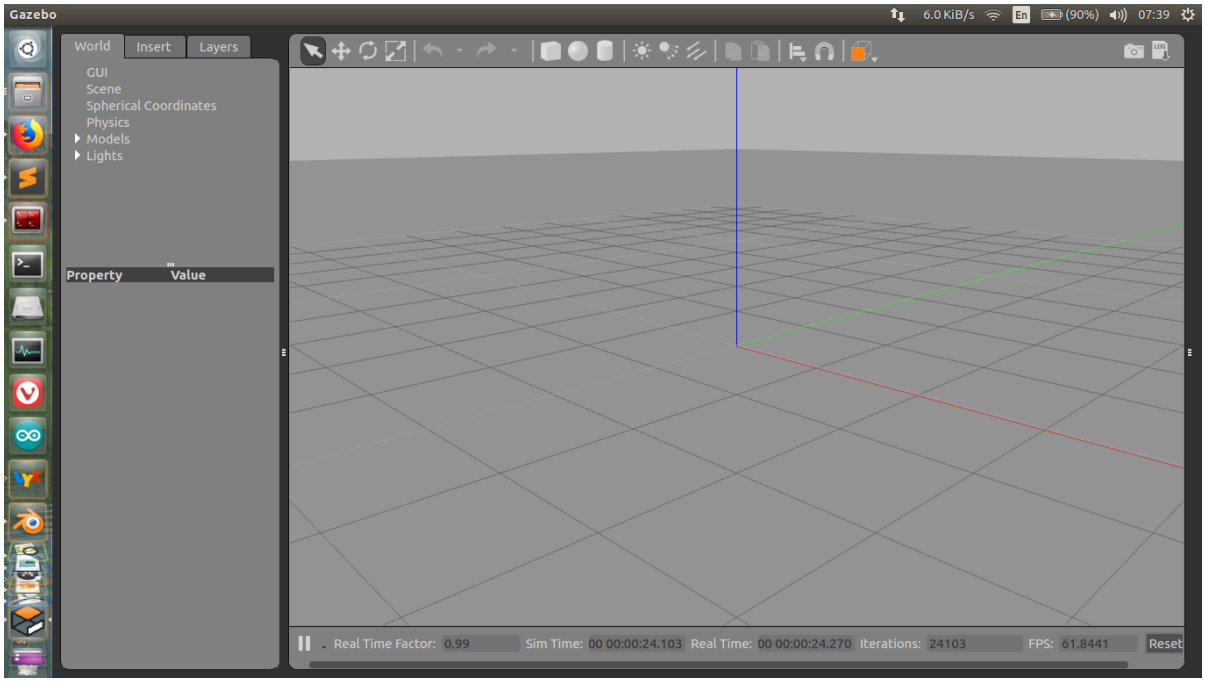


Figure 4.10: Gazebo GUI

Important panels

Both side panels right and left can be displayed, hidden or resized by dragging the bar that separates them from the Scene.

The left panel appears by default when you launch Gazebo. There are three tabs in the panel:

- **WORLD**: The World tab displays the models that are currently in the scene, and allows you to view and modify model parameters, like their pose. You can also change the camera view angle by expanding the "GUI" option and tweaking the camera pose.
- **INSERT**: The Insert tab is where you add new objects (models) to the simulation. To see the model list, you may need to click the arrow to expand the folder. Click (and release) on the model you want to insert, and click again in the Scene to add it.

- **LAYERS**: The Layers tab organizes and displays the different visualization groups that are available in the simulation, if any. A layer may contain one or more models. Toggling a layer on or off will display or hide the models in that layer.

This is an optional feature, so this tab will be empty in most cases.

Upper toolbar

The main Toolbar includes some of the most-used options for interacting with the simulator, such as buttons to: select, move, rotate, and scale objects; create simple shapes (e.g. cube, sphere, cylinder); and copy/paste. Go ahead and play around with each button to see how it behaves.

Buttom toolbar

The

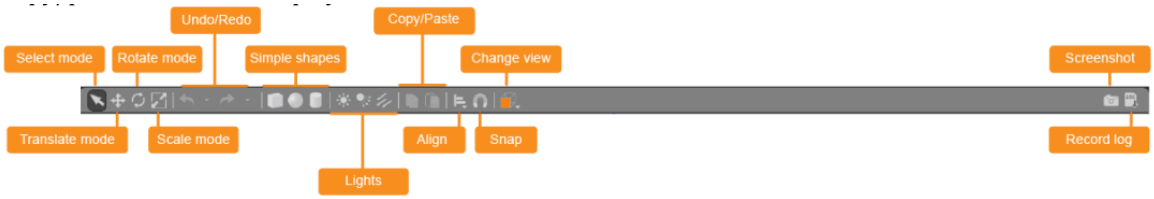


Figure 4.11: Upper Toolbar

Bottom Toolbar displays data about the simulation, like the simulation time and its relationship to real-life time. "Simulation time" refers to how quickly time is passing in the simulator when a simulation is running. Simulation can be slower or faster than real time, depending on how much computation is required to run the simulation.

"Real time" refers to the actual time that is passing in real life as the simulator runs. The relationship between the simulation time and real time is known as the "real time factor" (RTF). It's the ratio of simulation time to real time. The RTF is a measure of how fast or slow your simulation is running compared to real time.

The state of the world in Gazebo is calculated once per iteration. You can see the number of iterations on the right side of the bottom toolbar. Each iteration advances simulation by a fixed number of seconds, called the step size. By

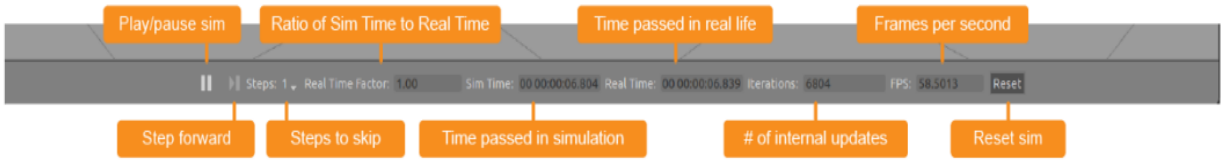


Figure 4.12: Bottom toolbar

default, the step size is 1 ms. You can press the pause button to pause the simulation and step through a few steps at a time using the step button.

Mouse control

The mouse is very useful when navigating in the Scene. We highly recommend using a mouse with a scroll wheel. Below are the basic mouse operations for navigating in the Scene and changing the view angle. Right-clicking on models will open a context menu with various.

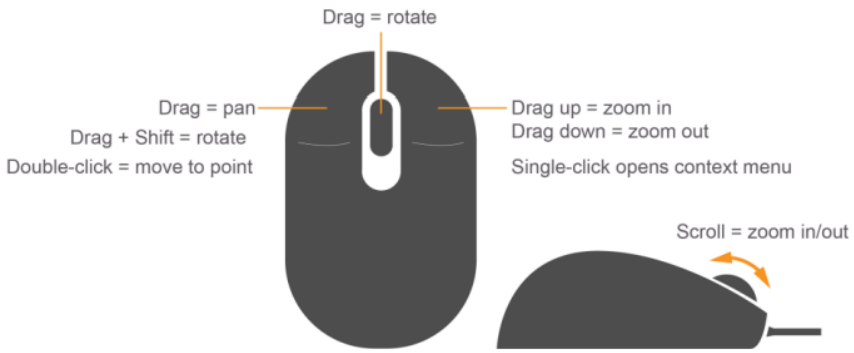


Figure 4.13 Mouse control

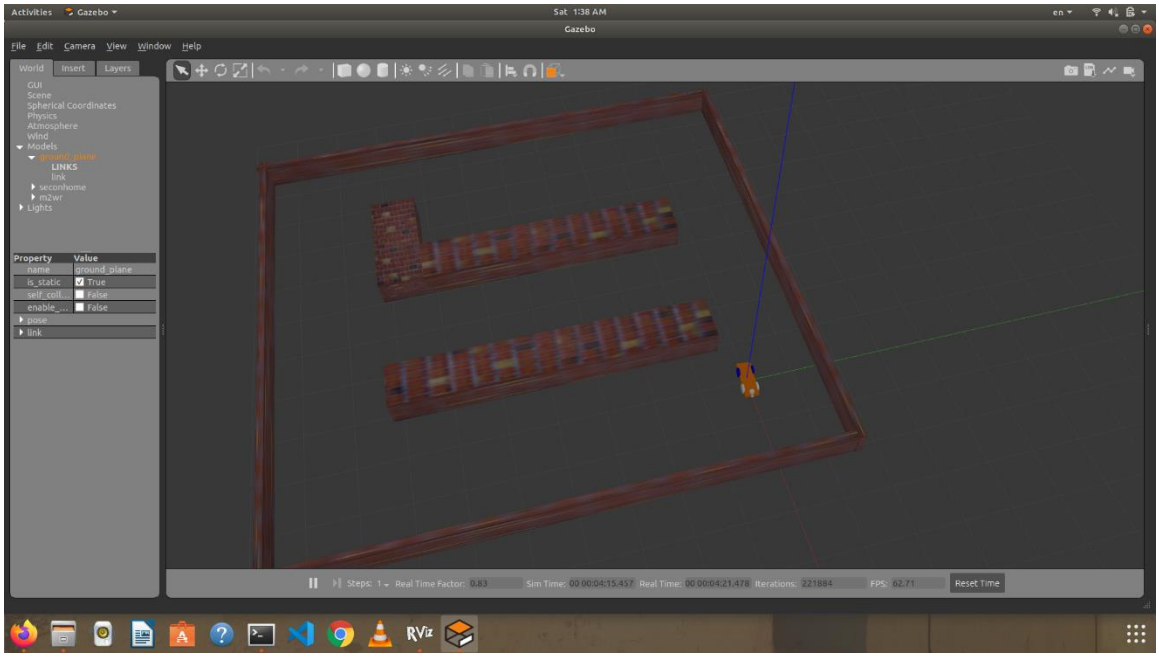


Figure 4.14 Our Model Simulated in Gazebo

Robot Visualizer – RViz

RViz is a 3D visualizer for displaying sensor data and state information from ROS. Using rviz, you can visualize Baxter's current configuration on a virtual model of the robot. You can also display live representations of sensor values coming over ROS Topics including camera data, infrared distance measurements, sonar data, and more.

Start rviz from a properly initialized environment (i.e. roscore is running) using

```
$ rosrun rviz rviz
```

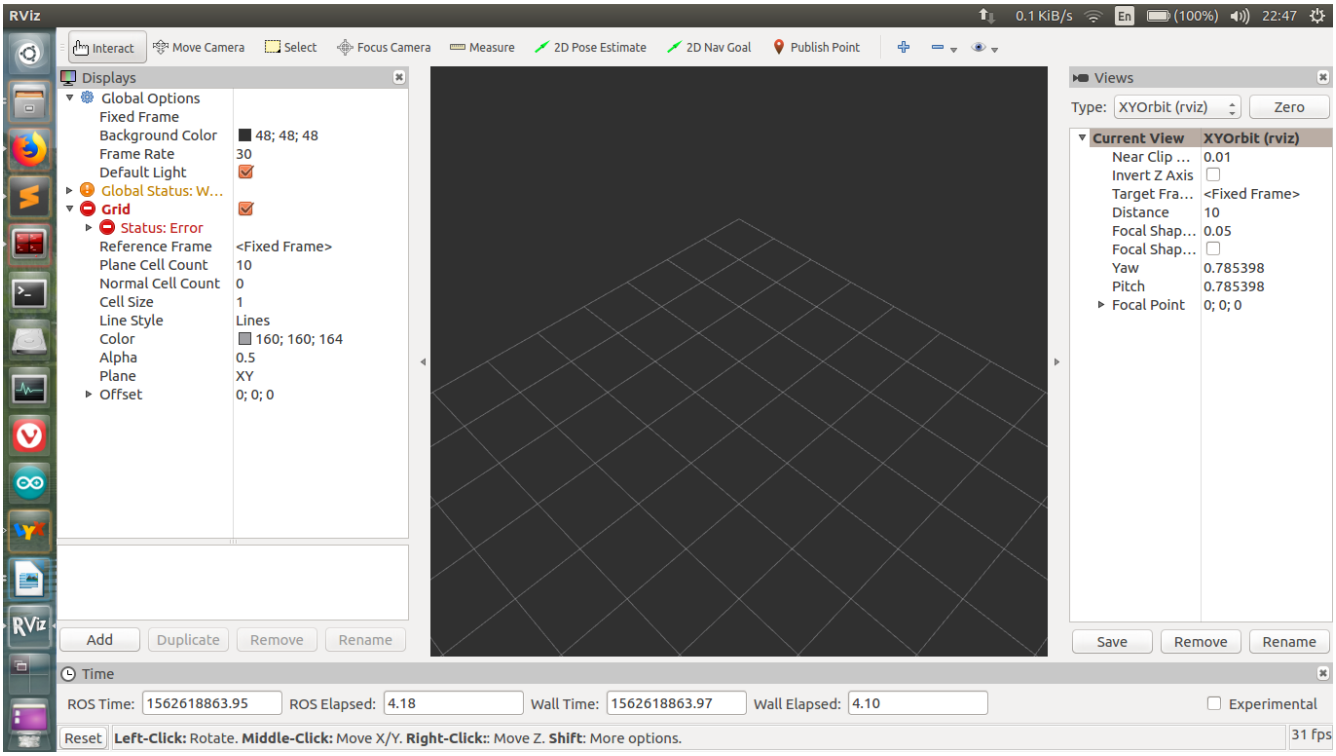


Figure 4.15: RVIZ user Interface

Display Types supported in RViz

Name	Description	Messages Used
Axes	Displays a set of Axes	
Effort	Shows the effort being put into each revolute joint of a robot.	sensor_msgs/JointStates
Camera	Creates a new rendering window from the perspective of a camera, and overlays the image on top of it.	sensor_msgs/Image, sensor_msgs/CameraInfo

Grid	Displays a 2D or 3D grid along a plane	
Grid Cells	Draws cells from a grid, usually obstacles from a costmap from the navigation stack.	nav_msgs/GridCells
Image	Creates a new rendering window with an Image. Unlike the Camera display, this display does not use a CameraInfo. VERSION: DIAMONDBACK+	sensor_msgs/Image
InteractiveMarker	Displays 3D objects from one or multiple Interactive Marker servers and allows mouse interaction with them. VERSION: ELECTRIC+	visualization_msgs/InteractiveMarker
Laser Scan	Shows data from a laser scan, with different options for rendering modes, accumulation, etc.	sensor_msgs/LaserScan
Map	Displays a map on the ground plane.	nav_msgs/OccupancyGrid
Markers	Allows programmers to display arbitrary primitive shapes through a topic	visualization_msgs/Marker, visualization_msgs/MarkerArray
Path	Shows a path from the navigation stack.	nav_msgs/Path
Point	Draws a point as a small sphere.	geometry_msgs/PointStamped
Pose	Draws a pose as either an arrow or axes.	geometry_msgs/PoseStamped
Pose Array	Draws a "cloud" of arrows, one for each pose in a pose array	geometry_msgs/PoseArray
Point Cloud(2)	Shows data from a point cloud, with different options for rendering modes, accumulation, etc.	sensor_msgs/PointCloud, sensor_msgs/PointCloud2
Polygon	Draws the outline of a polygon as lines.	geometry_msgs/Polygon
Odometry	Accumulates odometry poses from over time.	nav_msgs/Odometry
Range	Displays cones representing range measurements from sonar or IR range sensors. VERSION: ELECTRIC+	sensor_msgs/Range

RobotModel	Shows a visual representation of a robot in the correct pose (as defined by the current TF transforms).	
TF	Displays the tf transform hierarchy.	
Wrench	Draws a wrench as arrow (force) and arrow + circle (torque)	geometry_msgs/WrenchStamped
Oculus	Renders the RViz scene to an Oculus headset	

Robot Description for Gazebo and RViz

Overview

In order to simulate/visualize a robot within the ROS framework, we have to describe the kinematic chain of a robot so that physics engines in Gazebo can simulate the robot with a high degree of accuracy. At this point, it should be clear that RViz is used just for visualization of physical quantities that being sent and received to and from the robot in the form of messages. On the other hand, Gazebo is a full 3D simulator that simulates the reality.

Describing the kinematic chain (i.e. links and joints) of the robot can be accomplished with many XML-based files e.g. URDF, SDF, SMURF, etc. In the following section we will be exposed to URDF and SDF

URDF vs. SDF

The Unified Robot Description Format (URDF) is an XML specification to describe a robot. URDF is used in ROS to write robot description for use in simulation and visualization software.

We attempt to keep the specification of robots as general as possible, but obviously the specification cannot describe all robots. The main limitation at this point is that only tree structures can be represented, ruling out all parallel robots. Also, the specification assumes the robot consists of rigid links connected by joints; flexible elements are not supported. The specification covers:

- Kinematic and dynamic description of the robot
- Visual representation of the robot
- Collision model of the robot

While URDFs are a useful and standardized format in ROS, they are lacking many features and have not been updated to deal with the evolving needs of robotics.

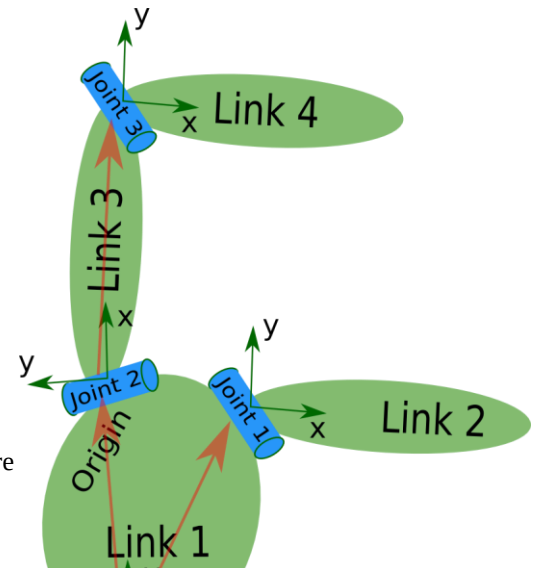
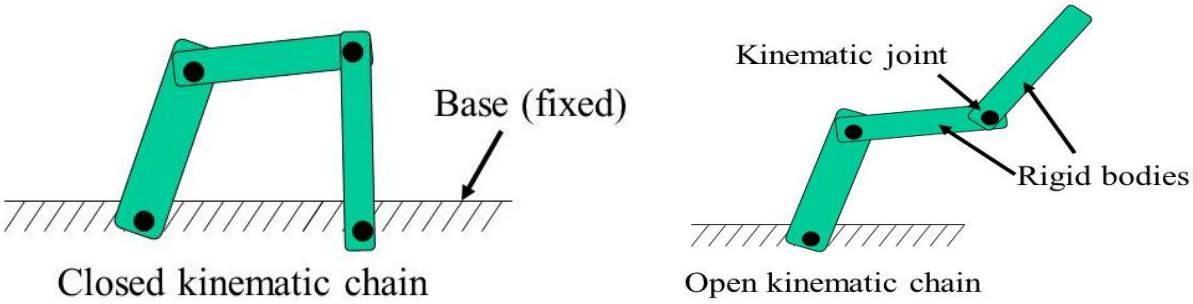


Figure 4.16: Simple example of an open chain kinematic model

- URDF can only specify the kinematic and dynamic properties of a single robot in isolation.
- URDF can not specify the pose of the robot itself within a world. It is also not a universal description format since it cannot specify joint loops (parallel linkages and closed kinematic chains), and it lacks friction and other properties.



- Additionally, it cannot specify things that are not robots, such as lights, heightmaps, etc.

Figure 4.17: Closed Kinematic Chain vs. Open Kinematic Chain

On the implementation side, the URDF syntax breaks proper formatting with heavy use of XML attributes, which in turn makes URDF more inflexible. There is also no mechanism for backward compatibility.
To deal with this issue, a new format called the Simulation Description Format (SDF) was created for use in Gazebo to solve the shortcomings of URDF. SDF is a complete description for everything from the world level down to the robot level. It is scalable, and makes it easy to add and modify elements. The SDF format is itself described using XML, which facilitates a simple upgrade tool to migrate old versions to new versions. It is also self-descriptive. So, Gazebo switched to Scene Definition Format (SDF). While URDF can be parsed into SDF and used with Gazebo simulator, it is not possible to convert SDF to URDF. To use robot described with URDF in Gazebo, special measures has to be taken into account. There are efforts in the community to update URDF format or provide a way to convert from SDF to URDF, but currently, no straightforward solution exists.

Our Robot Description

Because our robot is a closed chain kinematic model, it can be only simulated ***properly*** using SDF! As URDF doesn't support joint loops (the kinematic model has to be an open chain)
This means that we have to write two separate files:

- URDF for RViz
- SDF for Gazebo

As it's not possible to convert the ***proper*** model, SDF to URDF, which is a real tiresome.
That's why we looked for a more automated way to export the URDF or SDF files and the following section introduces a great open-source 3D modeling software that has an a fully featured addon for exporting many formats

that describe the robot physically.

Note that there's a software called sw2urdf_exporter, which exports the URDF file from the SolidWorks CAD file with some strict settings, this software is so buggy as it hasn't been updated since 2012, it always exports faulty URDF files.

TF (Transform Frames) Package

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.

A robotic system typically has many 3D coordinates frames that change over **time**, such as a world frame, base frame, etc. tf keeps track of all these frames over time. TF can operate in a **distributed system**. This means all the information about the coordinate frames of a robot is available to all ROS components on any computer in the system.

When opening RVZ, we can use the "Add" button to add a TF broadcaster that shows all the frames in a robot model, a robot model can be set by

```
$ rosparam set robot_description %YOUR URDF File%
```

After navigating to the path of the URDF file and executing

```
$ roslaunch urdf_tutorial display.launch model:=walle.urdf
```

RViz opens as follows and the nodes robot_state_publisher and joint_state_publisher are launched as well.

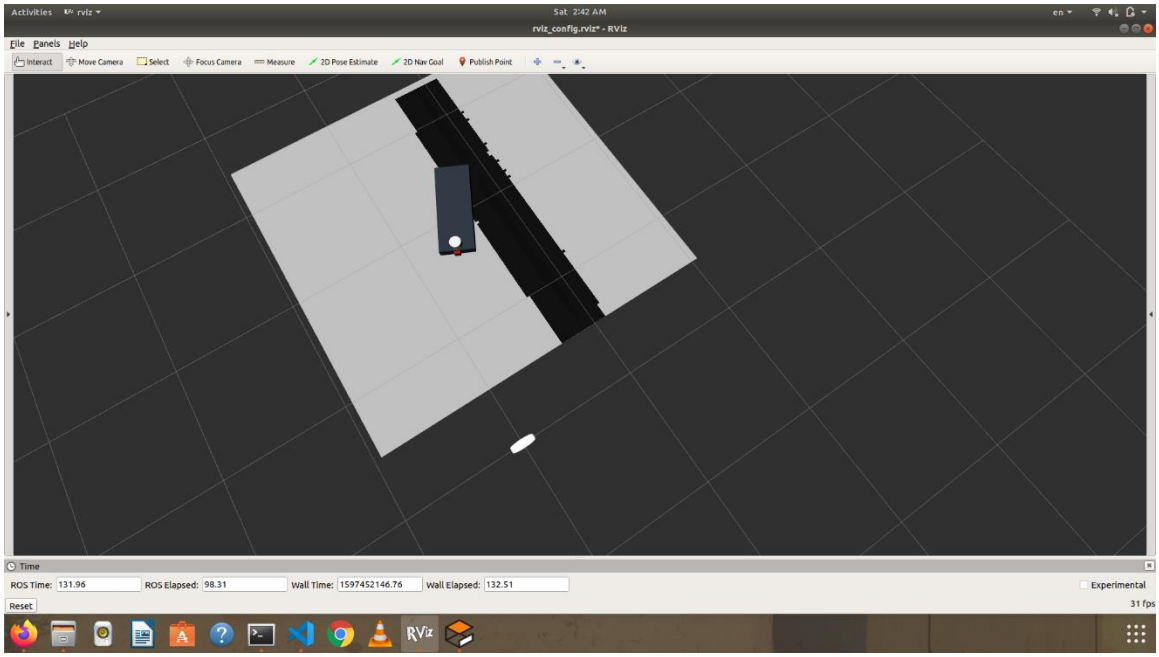


Figure 4.18: Our model in RVIZ

It should be noted that TF can get the transforms either from a tf broadcaster or after parsing the URDF file, it extracts the frames of the robot.

By running the following in another terminal

```
$ rosrun tf view_frames
$ evince frames.pdf
```

A pdf file opens with the TF tree of our robot. The following figures show the transformation tree between different link, the first tree is an open chain model (faulty tree) an when simulated in Gazebo (which uses a physics engine), the body falls down and touches the ground as it lacks the joint loop. The true TF tree should look like the second one.

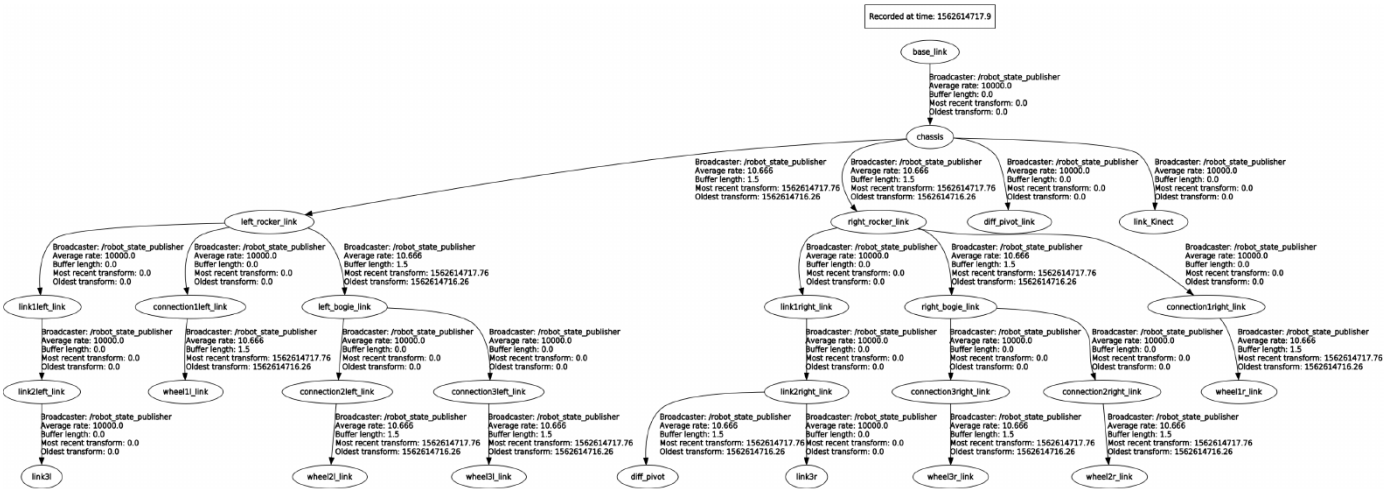


Figure 4.20: The closed chain kinematic model tf tree, lacks joint loops (URDF)

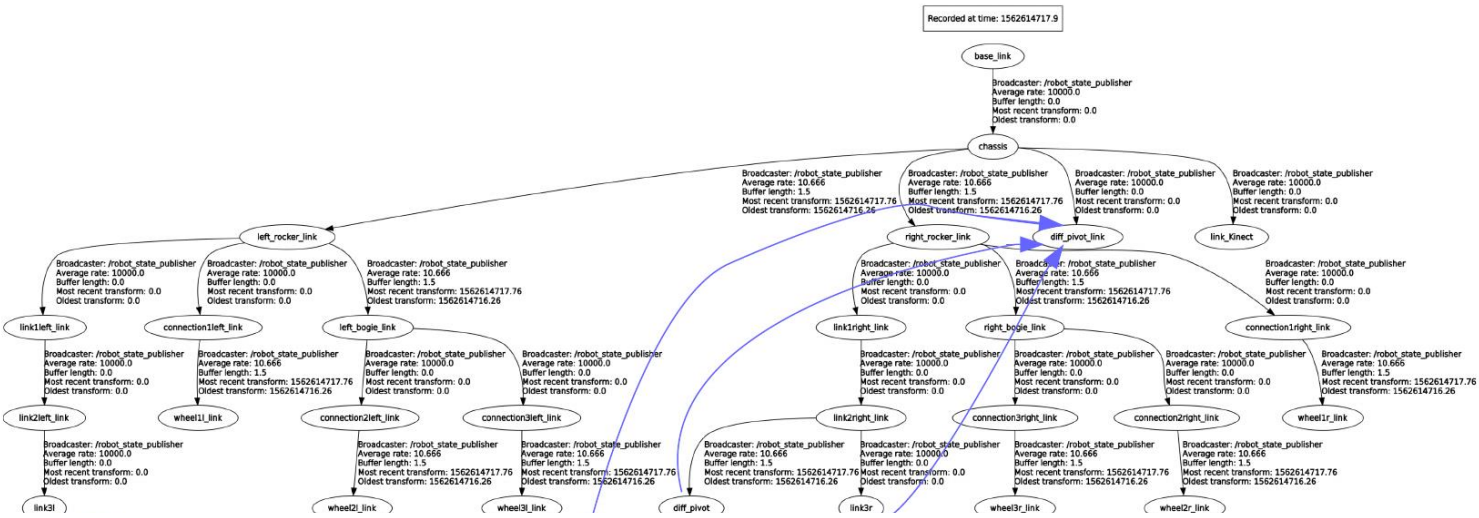


Figure 4.19: The open chain kinematic model tf tree, lacks joint loops (URDF)

LIDAR

which stands for **Light Detection and Ranging** - originated in the 1960s, shortly after the advent of lasers and was first used by the American National Center for Atmospheric Research in meteorology to measure clouds. Since then, the technique - also known as laser scanning or 3D scanning - has been used in applications from geography to forestry, and from atmospheric physics to laser altimetry.

It has been widely used in archaeology to map dig-sites and large areas of land, identifying things that

Figure 4.21: ydlidar X4

couldn't be seen from the ground. The National Oceanic and Atmospheric Administration in America has used it to map shorelines and the surface of the Earth, and NASA utilized the technology in 1971 when Apollo 15 astronauts mapped the surface of the moon using a laser altimeter.

How do they work?

The technique employs ultraviolet (UV), visible or near infrared (IR) light to image objects and map their physical features. Several measurements are taken in quick succession to yield a complex map of the surface at high resolution.

LIDAR measures the distance to a target using active sensors which emit an energy source for illumination, instead of relying on sunlight. It fires rapid pulses of laser light at a surface – anything up to 150,000 pulses a second – usually IR to map land, or water-penetrating green light to measure the seafloor or riverbed.

When the light hits the target object, it is reflected back to a sensor which measures the time taken for the pulse to bounce back from the target. The distance to the object is deduced by using the speed of light to calculate the distance traveled accurately. The result is precise three-dimensional information about the target object and its surface characteristics.

4.1.15 Applications

LIDAR was first used in vehicles in the early 2000s, mostly in the Grand DARPA Challenge – it is only in the last five years, or so that progress has been made concerning self-driving cars. Google and Uber are just a two of many names developing self-driving vehicles; their cars feature a bulky box on top of the roof which spins continuously giving 360° visibility and precise, in-depth information about the exact distance to an object to an accuracy of ±2cm. This box is the LIDAR system; it consists of a laser, scanner and optics and a specialized GPS receiver, especially important if the system is moving.

In the case of self-driving cars, LIDAR is used to generate huge 3D maps – which was its intended original use - that the car can then navigate through. It is also used – particularly by Google – to detect pedestrians and cyclists, traffic signs and other nearby obstacles.

Of course, such autonomous technology isn't without its pitfalls – take the recent fatality in Arizona where the technology failed to pick up a pedestrian crossing the road, for example. However, as LIDAR becomes more sophisticated, it will be increasingly capable of detecting and tracking objects. Improvements will mean higher resolution imagery will be possible and it will be able to operate at longer ranges so that the technology is capable of differentiating between someone walking, or someone on a bike, their speed, and direction.

4.1.16 Working Ydlidar X4 on Ros

Receive data from your LiDAR

In order to connect the LiDAR to you PC, you must first take care of the power supply. Depending of your device, it may require 5VDC or 12/24VDC, for instance.

5VDC supply voltage is usually supported by a USB connector plugged into your PC, you just have to wire it and the LiDAR will be ready to spin.

For higher supply voltage, you need an external alimentation (low frequency generator), a transformer/converter wired to main supply, or a battery.

Once you have connected your LiDAR with its power supply, you need to connect the data transmitter. It could be through the same cable as USB supplier, with a different one, with a Rx/Tx cable, a UART cable or an Ethernet cable. Usually, an adapter is sold with the LiDAR.

Read data from your LiDAR

Once your LiDAR is ready to be used, you have to check if you have the permissions on data input port:

Once you have connected the data transmitter cable on the USB or Ethernet port, type the following command line to check the permissions:

```
$ LS -L /DEV/TTY
```

You should see a new item labelled ACMX or USBX, X being a figure equal or higher than zero (depending on how many ports are already in-use).

Your output should be in the form:

```
$ CRW-RW-XX- 1 ROOT DIALOUT 166, 0 2016-09-12 14:18 /DEV/TTYACM0
```

or

```
$ CRW-RW-XX- 1 ROOT DIALOUT 166, 0 2016-09-12 14:18 /DEV/TTYUSB0
```

- If XX is rw, then the laser is configured properly.
- If XX is –, then the laser is not configured properly and you need to change permissions like below:

```
$ SUDO CHMOD A+RW /DEV/TTYACM0
```

or

```
$ SUDO CHMOD A+RW /DEV/TTYUSB0
```

Once the permissions are configured, you have to download the package of your LiDAR manufacturer

Go in launch folder and find the launch file name which match with your LiDAR version and launch it:

$ roslaunch votre_package votre_launch.launch

To check that the LiDAR is publishing to /scan, use

$ rostopic list

All active topics will be listed, check that /scan is present. Next, check the messages being published to /scan by using:

$ rostopic echo /scan

$ ROSRUN RVIZ RVIZ

The result should be a line mapping distances from the LiDAR in a rectangular coordinate system as shown in fig

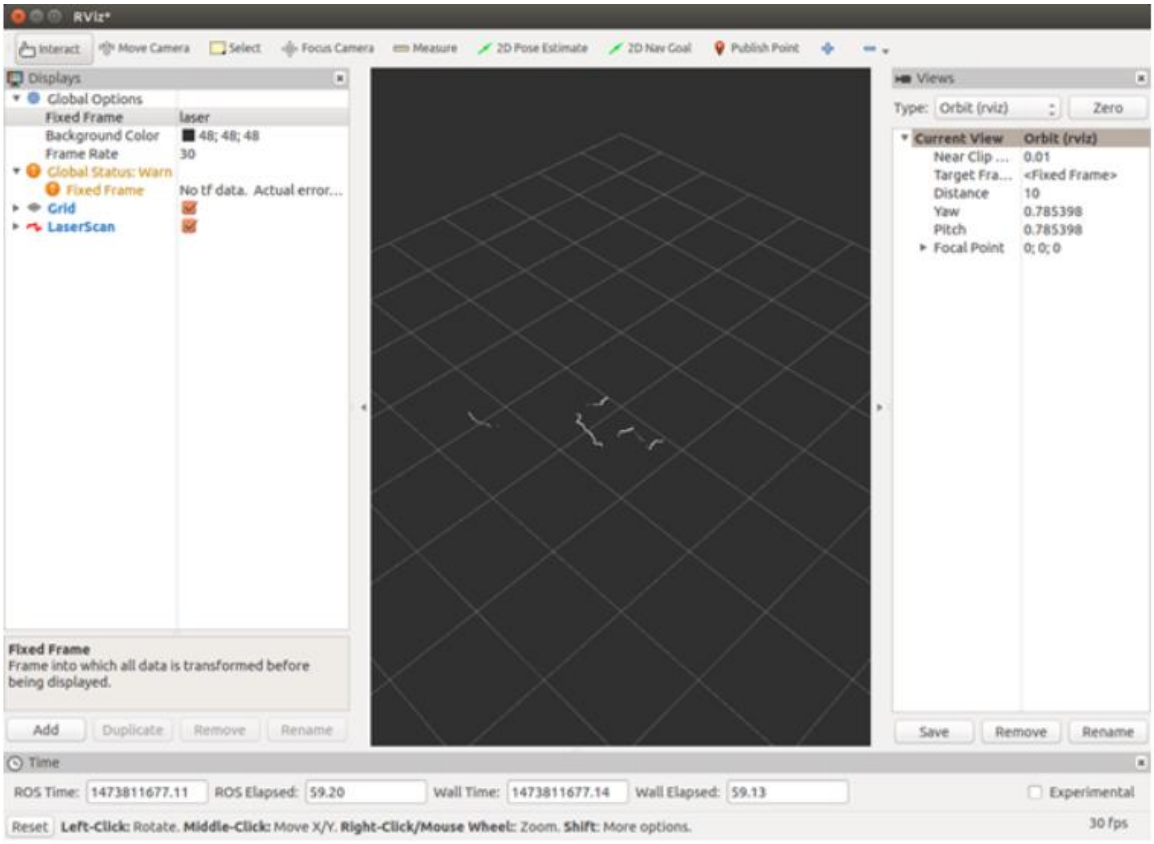


Figure 4.22: The result from lidar in rviz

4.2 Image Processing

Besides, wheel odometry method troubles errors data due to wheel slip. Therefore, we are focused on Visual odometry because this is stable in indoor where brightness is not changed rarely. Visual odometry solves the SLAM problem using a combination of image feature and depth data from laser sensors. Image feature is a process to extract the image where the color change is strong.

Kinect was developed by Microsoft as a motion sensing device for video games, but it works well as a mapping tool. Kinect is equipped with an RGB camera, a depth camera, an array of microphones, and a tilt motor. The RGB camera acquires 2D color images in the same way in which our smart phones or webcams acquire color video images. The Kinect 96 microphones can be used to capture sound data and a 3-axis accelerometer can be used to find the orientation of the Kinect. These features hold the promise of exciting applications, but unfortunately, this book will not delve into the use of these Kinect sensor capabilities.

Figure 4.23 kinect camera figure out

So, we use Kinect camera to provide us the computer vision with help of LIDAR. Kinect camera is able to provide us with the point cloud view that give an image with its depth

We use OPENNI package in ROS to get the full depth images from the camera. This package is developed by Ros and all required from user is to calibrate the camera he has and it is a simple process

But in our case, we are focus on simulation so we write a plugin in gazebo that provide a depth camera in the rover in the environment this plugin is simple

Camera plugin in gazebo

First, to create a camera plugin in gazebo or any sensor in gazebo, there are two important steps:

- Create plugin structure (sensor shape and its connection and joints to the base link or rover) And this written down in XARO files

- Create the plugin main work or in other way what the sensor type and its limits

Here in the following figure shows the sensor(camera) structure in the front base of the rover

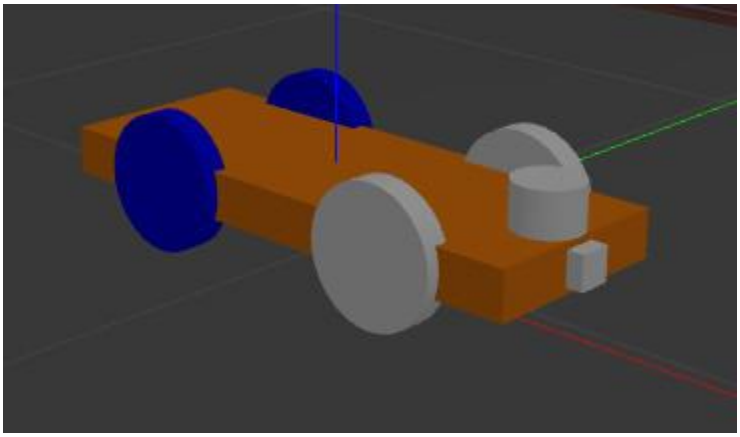


Figure 4.24 camera and lidar plugin

And here in the following table are the specs(limits) of the sensor which written down in .gazebo files and it written with XML markup language

#	Property	value
1	Update rate	30
2	Horizontal fov	1.3962634
3	Width	800
4	height	800
5	format	R8G8B8
6	Noise type	gaussian
7	mean	0.0
8	Stddev(standard deviation)	0.007

After these steps we were able to have a camera in our model and sense the environment and the following figure from our model with our plugin camera

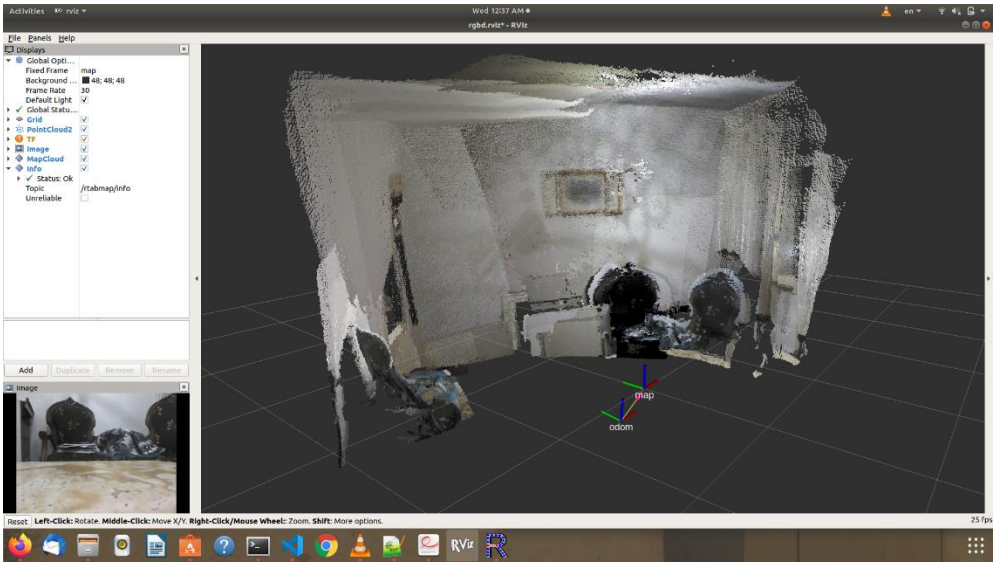


Figure 4.25 Real Kinect camera image and 3D map

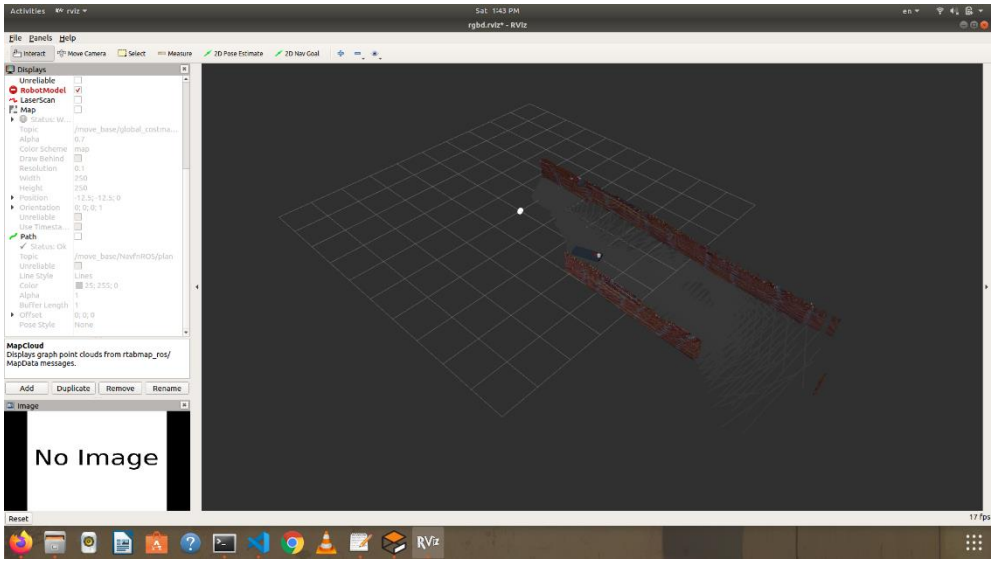


Figure 4.26 Simulated environment Mapping using Kinect

Chapter 5

5. SLAM (Simultaneous localization and mapping)

5.1 Introduction

simultaneous localization and mapping (SLAM) is the computational problem of constructing or updating a map of an unknown environment while simultaneously keeping track of an agent's location within it.

SLAM is hard, because:

- a map is needed for localization and
- a good pose estimate is needed for mapping.

Where:

- **Localization:** inferring location given a map.
- **Mapping**: inferring a map given locations.
- **SLAM:** learning a map and locating the robot simultaneously.

We can say that SLAM is a **chicken-or-egg** problem as:

- a map is needed for localization and
- a pose estimate is needed for mapping.

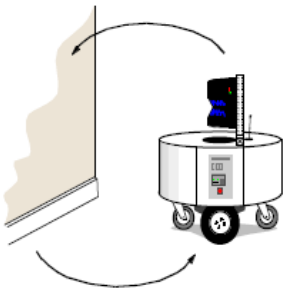


Figure 5.1 SLAM problem

SLAM algorithms are used in many applications that we use in different fields. SLAM is central to a range of indoor, outdoor, in-air and underwater applications for both manned and autonomous vehicles.

Examples:

- **At home**: vacuum cleaner, lawn mower.
- **Air**: surveillance with unmanned air vehicles.
- **Underwater**: reef monitoring.
- **Underground**: exploration of mines.
- **Space**: terrain mapping for localization.

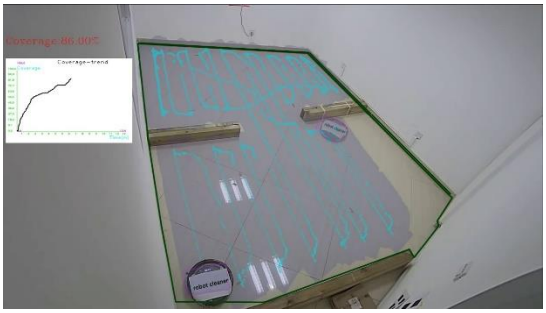


Figure 5.2 2D Indoor SLAM of a Vacuum cleaner

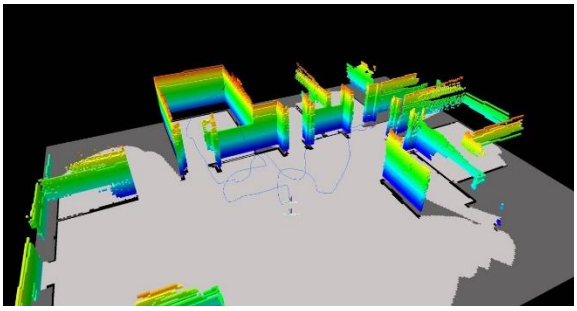


Figure 5.3 3D SLAM of UAV

Figure 5.5 Undersea

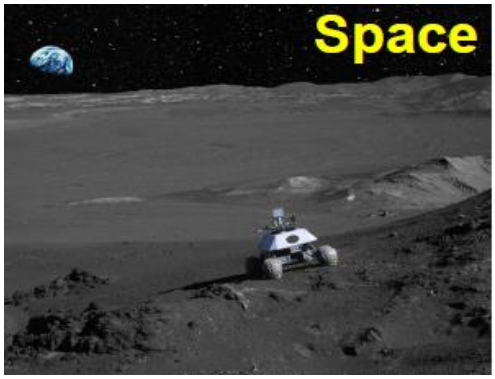


Figure 5.4 Space

Let's know first what is the meaning of Localization and Mapping.

5.2 Localization

One of the central problems for driving robots is localization. For many application scenarios, we need to know a robot's position and orientation at all times. For example, a cleaning robot needs to make sure it covers the whole floor area without repeating lanes or getting lost, or an office delivery robot needs to be able to navigate a building floor and needs to know its position and orientation relative to its starting point. This is a non-trivial problem in the absence of global sensors.

5.2.1 Localization using external sensors

Local positioning system using beacons

The localization problem can be solved by using a global positioning system. In an outdoor setting this could be the satellite-based GPS. In an indoor setting, a global sensor network with infrared, sonar, laser, or radio beacons could be employed. These will give us directly the desired robot coordinates as shown in the next figure.

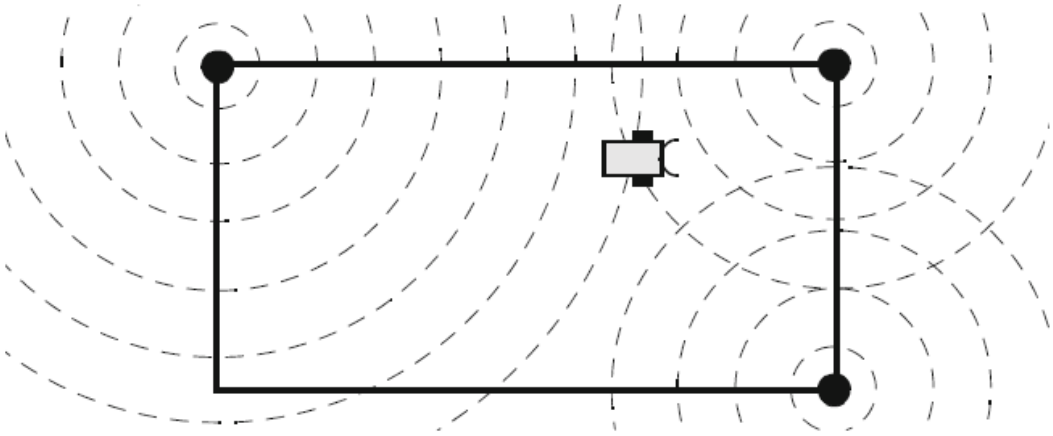


Figure 5.6 Global positioning system

Let us assume a driving environment that has a number of synchronized beacons that are sending out sonar signals at the same regular time intervals, but at different (distinguishable) frequencies. By receiving signals from two or three different beacons, the robot can determine its local position from the time difference of the signals' arrival times.

Using two beacons can narrow down the robot position to two possibilities, since two circles have two intersection points. For example, if the two signals arrive at exactly the same time, the robot is located in the middle between the two transmitters. If, say, the left beacon's signal arrives before the right one, then the robot is closer to the left beacon by a distance proportional to the time difference. Using local position coherence, this may already be sufficient for global positioning. However, to be able to determine a 2D position without local sensors, three beacons are required.

Only the robot's position can be determined by this method, not its orientation. The orientation has to be deducted from the change in position (difference between two subsequent positions), which is exactly the method employed for satellite-based GPS, or from an additional compass sensor.

Using global sensors is in many cases not possible because of restrictions in the robot environment, or not desired because it limits the autonomy of a mobile robot. On the other hand, in some cases it is possible to convert a system with global sensors as in Figure 16.1 to one with local sensors. For example, if the sonar sensors can be mounted on the robot and the beacons are converted to reflective markers, then we have an autonomous robot with local sensors.

Home beacons:

Another idea is to use light emitting homing beacons instead of sonar beacons, i.e. the equivalent of a lighthouse. With two light beacons with different colors, the robot can determine its position at the intersection of the lines from the beacons at the measured angle. The advantage of this method is that the robot can determine its position *and* orientation. However, in order to do so, the robot has either to perform a 360° rotation, or to possess an omni-directional vision system that allows it to determine the angle of a recognized light beacon.

For example, after doing a 360° rotation in the next figure, the robot knows it sees a green beacon at an angle of 45° and a red beacon at an angle of 165° in its local coordinate system.

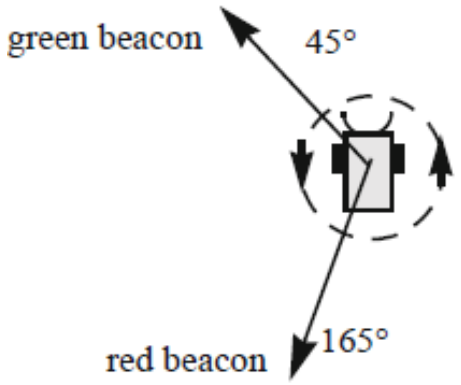


Figure 5.7 Beacon measurement

We still need to fit these two vectors in the robot's environment with known beacon positions (see Figure 16.3). Since we do not know the robot's distance from either of the beacons, all we know is the angle difference under which the robot sees the beacons (here: 165°– 45° = 120°).

As can be seen in the next figure, top, knowing only two beacon angles is not sufficient for localization. If the robot in addition knows its global orientation, for example by using an on-board compass, localization is possible When using three light beacons, localization is also possible without additional orientation knowledge.

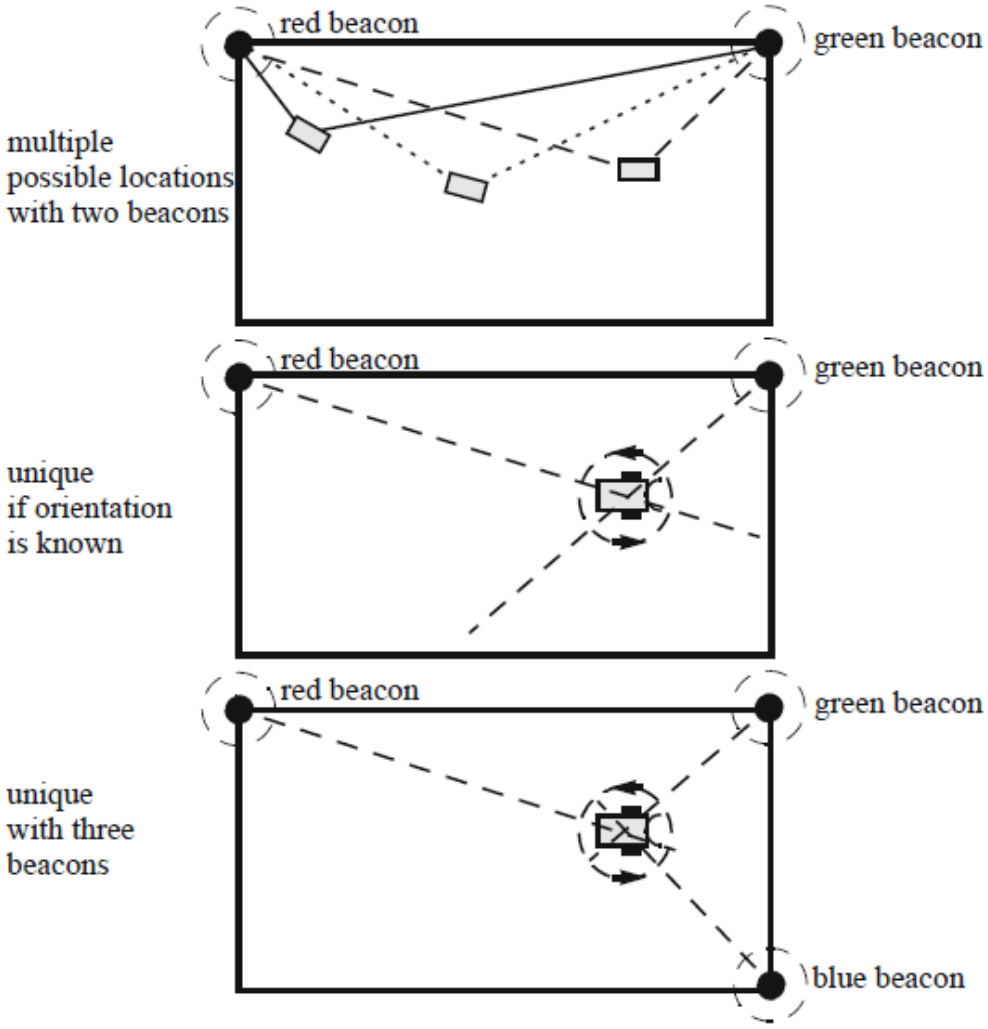


Figure 5.8 Homing beacon

Dead reckoning:

In many cases, driving robots have to rely on their wheel encoders alone for short-term localization, and can update their position and orientation from time to time, for example when reaching a certain waypoint. So-called "dead reckoning" is the standard localization method under these circumstances. Dead reckoning is a nautical term from the 1700[s] when ships did not have modern navigation equipment and had to rely on vector-adding their course segments to establish their current position.

Dead reckoning can be described as *local polar coordinates*, or more practically as *turtle graphics geometry*. As can be seen in the next figure, it is required to know the robot's starting position

and orientation. For all subsequent driving actions (for example straight sections or rotations on the spot or curves), the robot's current position is updated as per the feedback provided from the wheel encoders.

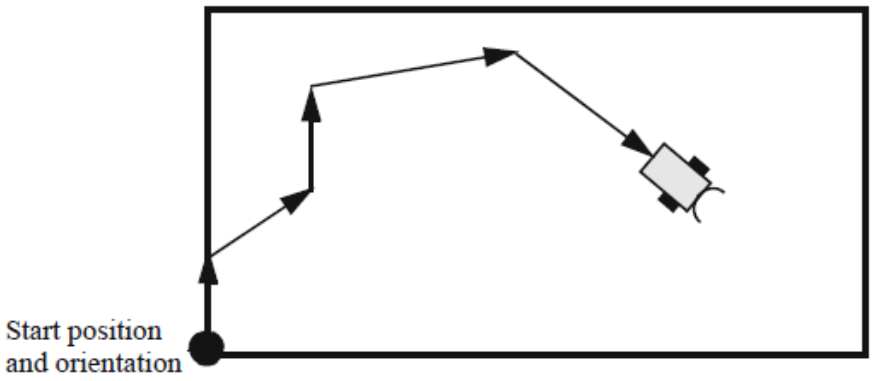


Figure 5.9 dead reckoning

Obviously, this method has severe limitations when applied for a longer time. All inaccuracies due to sensor error or wheel slippage will add up over time. Especially bad are errors in orientation, because they have the largest effect on position accuracy.

This is why an on-board compass is very valuable in the absence of global sensors. It makes use of the earth's magnetic field to determine a robot's absolute orientation. Even simple digital compass modules work indoors and outdoors and are accurate to about 1°.

5.3 Probabilistic Localization

All robot motions and sensor measurements are affected by a certain degree of noise. The aim of probabilistic localization is to provide the best possible estimate of the robot's current configuration based on all previous data and their associated distribution functions. The final estimate will be a probability distribution because of the inherent uncertainty.

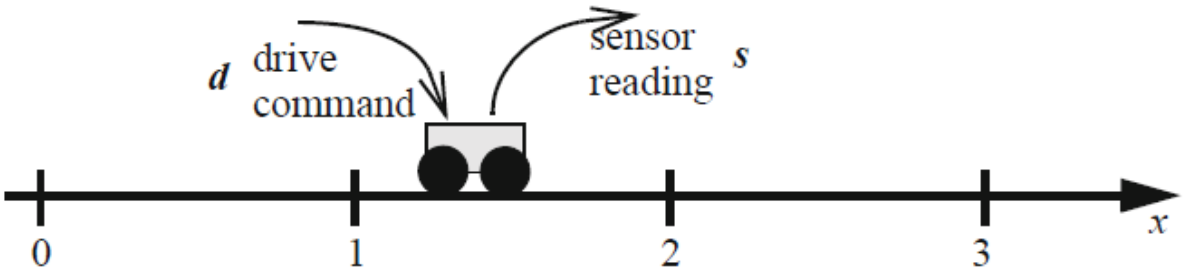


Figure 5.10 Uncertainty in actual position

As an example:

Assume a robot is driving in a straight line along the *x* axis, starting at the true position *x*=0. The robot executes driving commands with distance *d*, where *d* is an integer, and it receives sensor data from its on-board global (absolute) positioning system *s* (e.g. a GPS receiver), where *s* is also an integer. The values for *d* and $\Delta s = s - s'$ (current position measurement minus position measurement before executing driving command) may differ from the true position $\Delta x = x - x'$.

The robot's driving accuracy from an arbitrary starting position has to be established by extensive experimental measurements and can then be expressed by a PMF (probability mass function), e.g.:

$$p(\Delta x = d - 1) = 0.2; p(\Delta x = d) = 0.6; p(\Delta x = d + 1) = 0.2$$

Note that in this example, the robot's true position can only deviate by plus or minus one unit (e.g. cm); all position data are discrete.

In a similar way, the accuracy of the robot's position sensor has to be established by measurements, before it can be expressed as a PMF. In our example, there will again only be a possible deviation from the true position by plus or minus one unit:

$$p(x = s - 1) = 0.1; p(x = s) = 0.8; p(x = s + 1) = 0.1$$

Assuming the robot has executed a driving command with *d*=2 and after completion of this command, its local sensor reports its position as *s*=2. The probabilities for its actual position *x* are as follows, with *n* as normalization factor:

$$p(x=1) = n \cdot p(s=2 \mid x=1) \cdot p(x=1 \mid d=2, x'=0) \cdot p(x'=0)$$
$$= n \cdot 0.1 \cdot 0.2 \cdot 1 = 0.02n$$
$$p(x=2) = n \cdot p(s=2 \mid x=2) \cdot p(x=2 \mid d=2, x'=0) \cdot p(x'=0)$$
$$= n \cdot 0.8 \cdot 0.6 \cdot 1 = 0.48n$$
$$p(x=3) = n \cdot p(s=2 \mid x=3) \cdot p(x=3 \mid d=2, x'=0) \cdot p(x'=0)$$
$$= n \cdot 0.1 \cdot 0.2 \cdot 1 = 0.02n$$

Positions 1, 2 and 3 are the only ones the robot can be at after a driving command with distance 2, since our PMF has probability 0 for all deviations greater than plus or minus one. Therefore, the three probabilities must add up to one, and we can use this fact to determine the normalization factor *n*:

$$0.02n + 0.48n + 0.02n = 1$$
$$\Rightarrow n = 1.92$$

Now, we can calculate the probabilities for the three positions, which reflect the robot's belief:

p(x=1) = 0.04;
p(x=2) = 0.92;
p(x=3) = 0.04

So, the robot is most likely to be in position 2, but it remembers all probabilities at this stage.

Continuing with the example, let us assume the robot executes a second driving command, this time with *d*=1, but after execution its sensor still reports *s*=2. The robot will now recalculate its position belief according to the conditional probabilities, with *x* denoting the robot's true position after driving and *x'* before driving:

$$p(x=1) = n \cdot p(s=2 \mid x=\mathbf{1}) \cdot$$
$$\mathbf{[}\ p(x=1 \mid d=1, x'=1) \cdot p(x'=1)$$
$$+p(x=1 \mid d=1, x'=2) \cdot p(x'=2)$$
$$+p(x=1 \mid d=1, x'=3) \cdot p(x'=3)\ \mathbf{]}$$
$$= n \cdot 0.1 \cdot (0.2 \cdot 0.04 + \mathbf{0} \cdot 0.92 + \mathbf{0} \cdot 0.04)$$
$$= 0.0008n$$

$$p(x=2) = n \cdot p(s=2 \mid x=\mathbf{2}) \cdot$$
$$\mathbf{[}\ p(x=2 \mid d=1, x'=1) \cdot p(x'=1)$$
$$+p(x=2 \mid d=1, x'=2) \cdot p(x'=2)$$
$$+p(x=2 \mid d=1, x'=3) \cdot p(x'=3)\ \mathbf{]}$$
$$= n \cdot 0.8 \cdot (0.6 \cdot 0.04 + 0.2 \cdot 0.92 + \mathbf{0} \cdot 0.04)$$
$$= 0.1664n$$

$$p(x=3) = n \cdot p(s=2 \mid x=\mathbf{3}) \cdot$$
$$\mathbf{[}\ p(x=3 \mid d=1, x'=1) \cdot p(x'=1)$$
$$+p(x=3 \mid d=1, x'=2) \cdot p(x'=2)$$
$$+p(x=3 \mid d=1, x'=3) \cdot p(x'=3)\ \mathbf{]}$$
$$= n \cdot 0.1 \cdot (0.2 \cdot 0.04 + 0.6 \cdot 0.92 + 0.2 \cdot 0.04)$$
$$= 0.0568n$$

Note that only states x = 1, 2 and 3 were computed since the robot's true position can only differ from the sensor reading by one. Next, the probabilities are normalized to 1.

$$0.0008n + 0.1664n + 0.0568n = 1$$

$$\rightarrow \quad n = 4.46$$

→ p(x=1) = 0.0036

p(x=2) = 0.743

p(x=3) = 0.254

These final probabilities are reasonable because the robot's sensor is more accurate than its driving, hence p(x=2) > p(x=3). Also, there is a very small chance the robot is in position 1, and indeed this is represented in its belief.

The biggest problem with this approach is that the configuration space must be discrete. That is, the robot's position can only be represented discretely. A simple technique to overcome this is to set the discrete representation to the minimum resolution of the driving commands and sensors, e.g. if we may not expect driving or sensors to be more accurate than 1cm, we can then express all distances in 1cm increments. This will, however, result in a large number of measurements and a large number of discrete distances with individual probabilities.

To solve this problem, we use SLAM using particle filter.

5.3.1 Particle Filter

A technique called *particle filters* can be used to address this problem and will allow the use of non-discrete configuration spaces. The key idea in particle filters is to represent the robot's belief as a set of N particles, collectively known as M. Each particle consists of a robot configuration *x* and a weight $w \in [0,1]$.

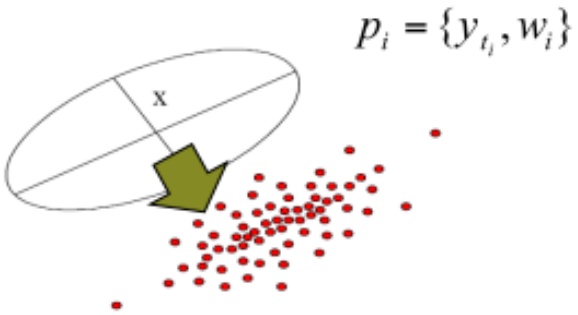


Figure 5.11 Probability distribution as particles

The key idea is to estimate a probability density over the state-space conditioned on the data. This posterior is typically called the belief and is denoted:

$$Bel(x_t) = p(x_t|d_{0..t})$$

Where $Bel(x_t)$ is Belief, x_t is the robot state at time t, and $d_{0..t}$ if the data from time 0 to time t.

For mobile robots, we distinguish two types of data: perceptual data such as laser range measurements, and odometry data, which carries information about robot motion. Denoting the former by o (for observation) and the latter by a (for action), we have:

$$Bel(x_t) = p(x_t|o_t, a_{t-1}, o_{t-1}, \dots)$$

After some simplification we have:

$$Bel(x_t) = \eta\, p(o_t|x_t) \int p(x_t|x_{t-1}, a) Bel(x_{t-1})\, dx_{t-1}$$

Where η is **Normalization Constant (**Make sure everything adds up to 1),

$p(o_t|x_t)$ is **Sensor Model (**Compute how likely your measurements were given updated particles),

$p(x_t|x_{t-1}, a)$ is **Motion Model (**Simulate noisy dynamics of particles based on control input),

and $Bel(x_{t-1})$ is **Previous Belief**.

Initialization

Here we see a part of a map previously generated. The black pixels on the map denote the walls and the grey pixels denote free space.

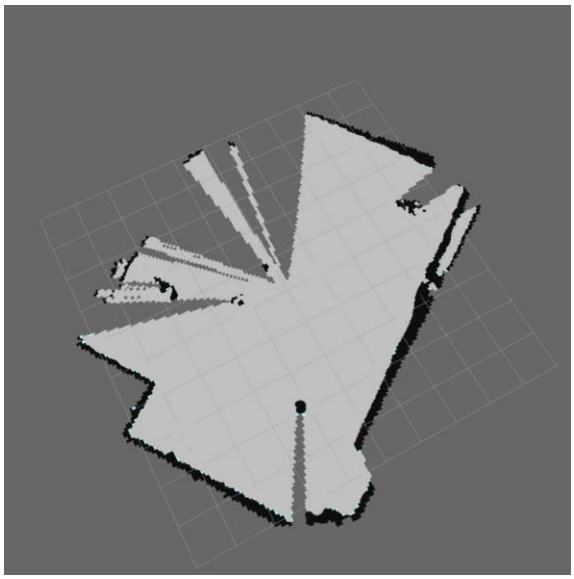


Figure 5.12 2D map generated by gapping

The red arrow indicates our initial pose obtained from user input. Let's use particle filters to solve the pose tracking problem and localize more accurately in the map.

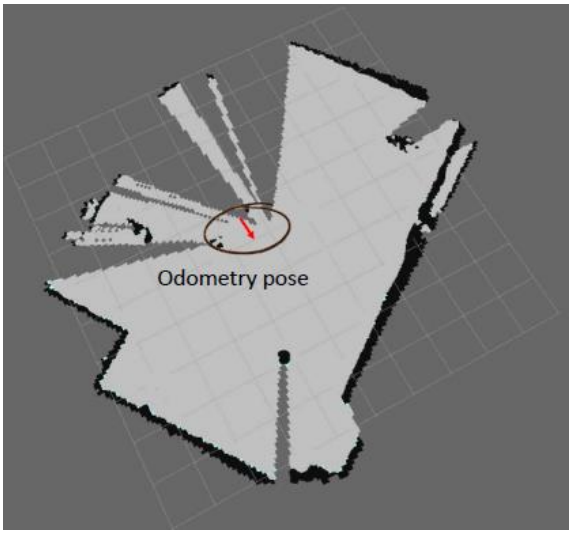


Figure 5.13 initialization of the first pose by user

First, we need to generate a set of hypotheses for our first position. These the discrete particles drawn from a Gaussian distribution of mean being the initial guess and with a small covariance.

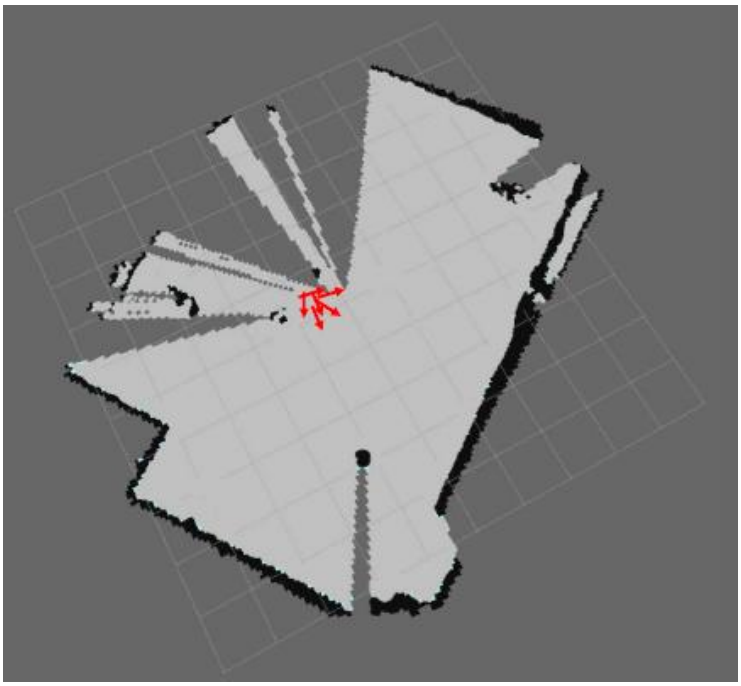


Figure 5.14 hypothesis (guesses) for the first location

Motion Model

Applying the measured control input to the motion model (with noise), yields an updated set of possible poses.

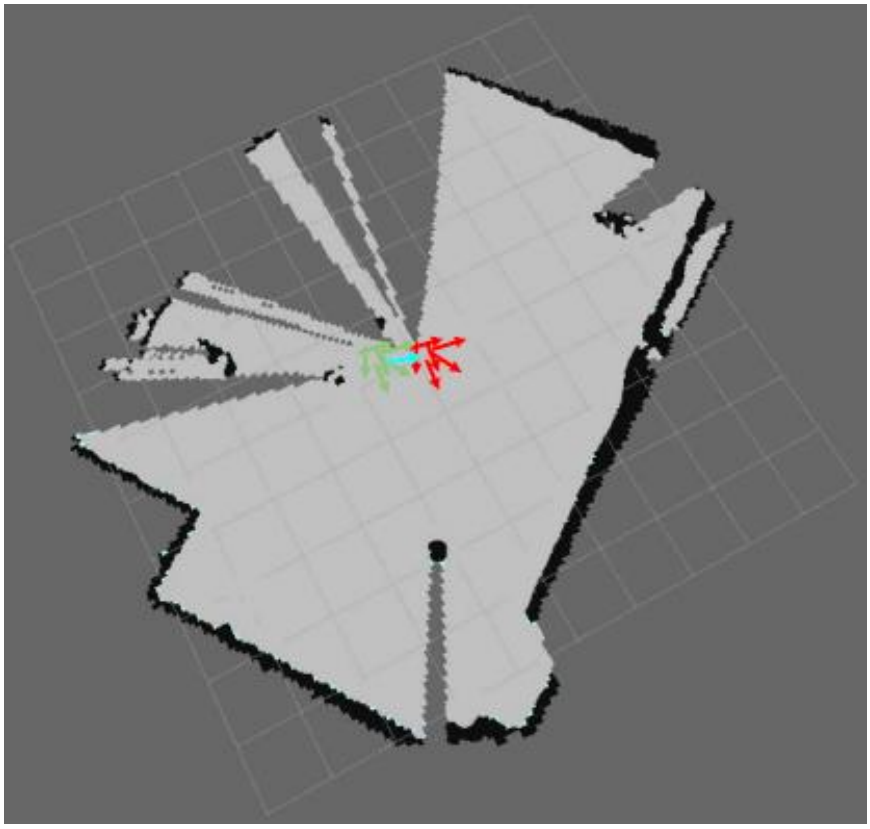


Figure 5.15 control input updates possible poses

Sensor Model

For each particle we can create a ‘fake’ laser scan by ray casting against the map at the particles pose. We predict what should the laser scan readings if this pose is right.

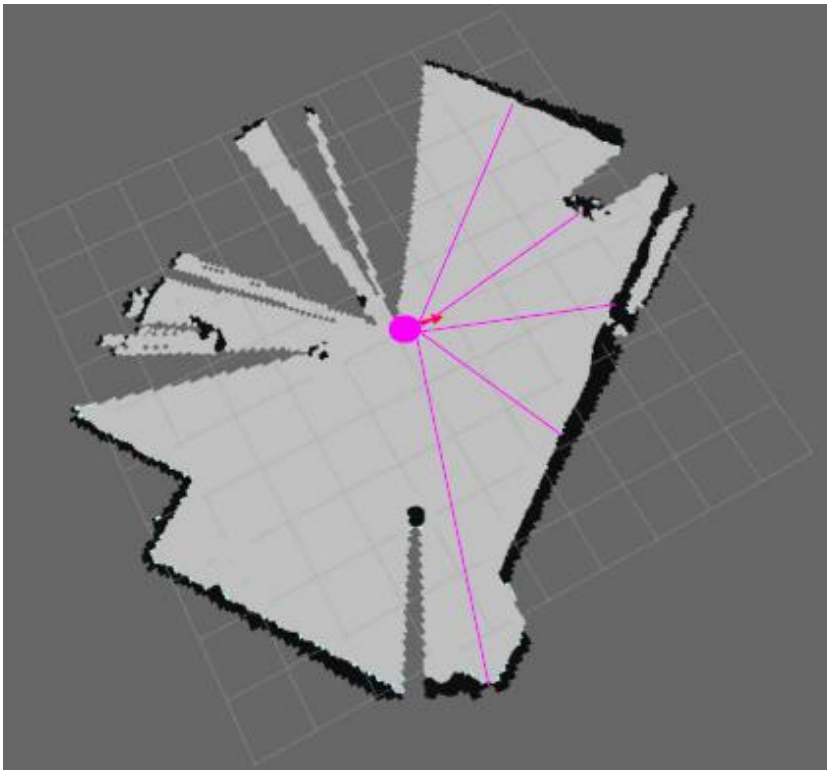


Figure 5.16 Sensor Model

Computing Particle Weights:

Recall, we have the 'real' laser scan which the rover observed (shown in green in the next figure). Note we don't know where the green dot is but we do know the range measurements.

We can compute a score for the fake laser scan:

$$p(o_t, x_t^i) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{(o_t - r^i)^2}{2\sigma^2}}$$

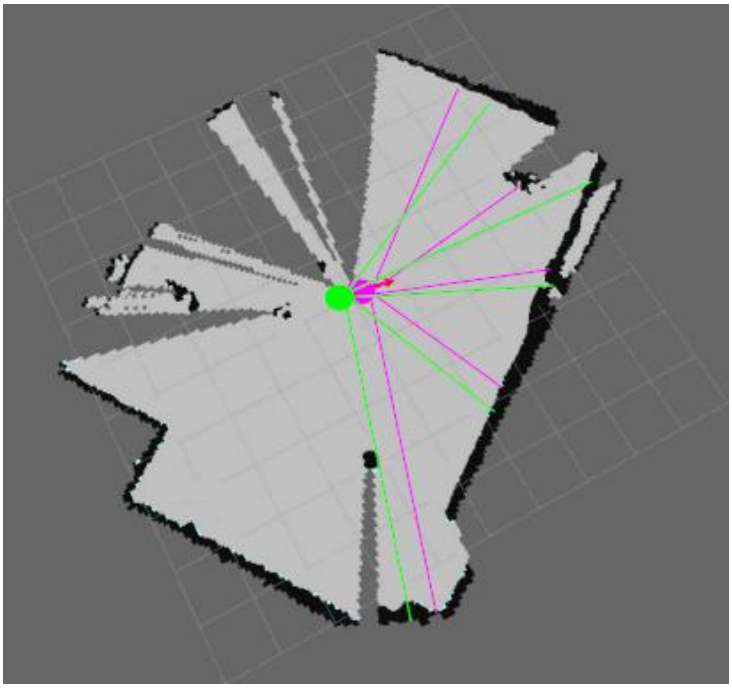


Figure 5.17 Particle weights

As an example, if the predicted pose is as shown in the next figure, so we must have readings of the map as shown.

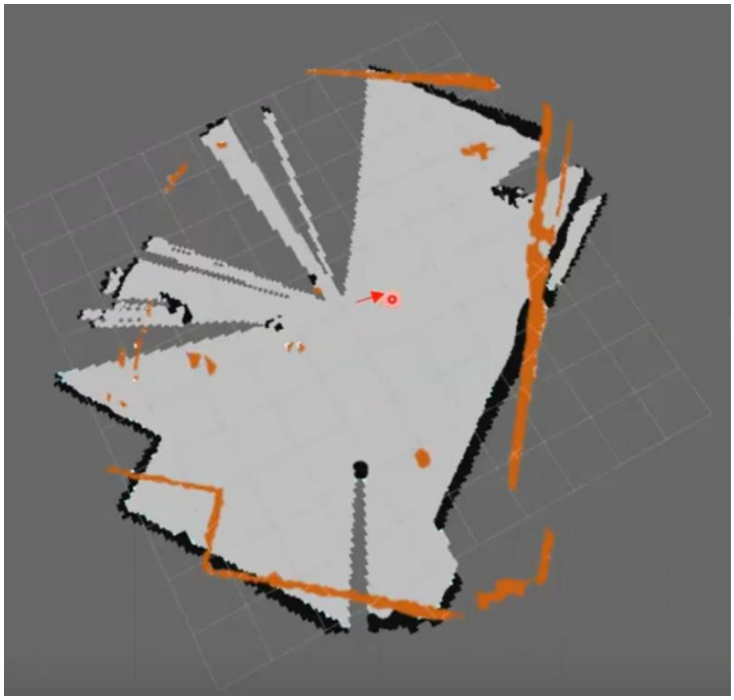


Figure 5.18 predicted pose and corresponding laser readings

The next figure shows best fit pose with the map:

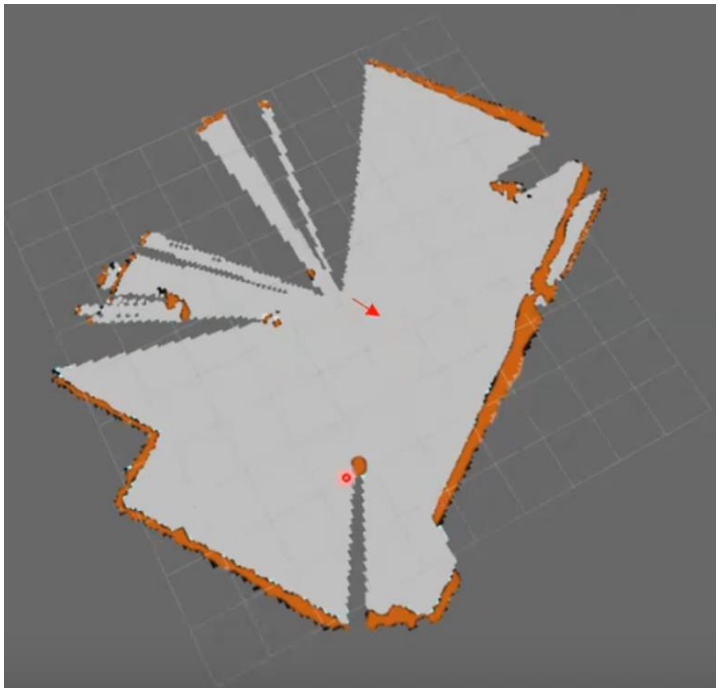


Figure 5.19 best fitted pose

5.4 Environment representation (2D Mapping)

The two basic representations of a 2D environment are *configuration space* and *occupancy grid*. In configuration space, we are given the dimensions of the environment plus the coordinates of all obstacle walls (e.g., represented by line segments). In an occupancy grid, the environment is specified at a certain resolution with individual pixels either representing free space (white pixels) or an obstacle (black pixels). These two formats can easily be transformed into each other. For transforming a configuration space into an occupancy grid, we can "print" the obstacle coordinates on a canvas data structure of the desired resolution. For transforming an occupancy grid into a configuration space, we can extract obstacle line segment information by combining neighboring obstacle pixels into individual line segments.

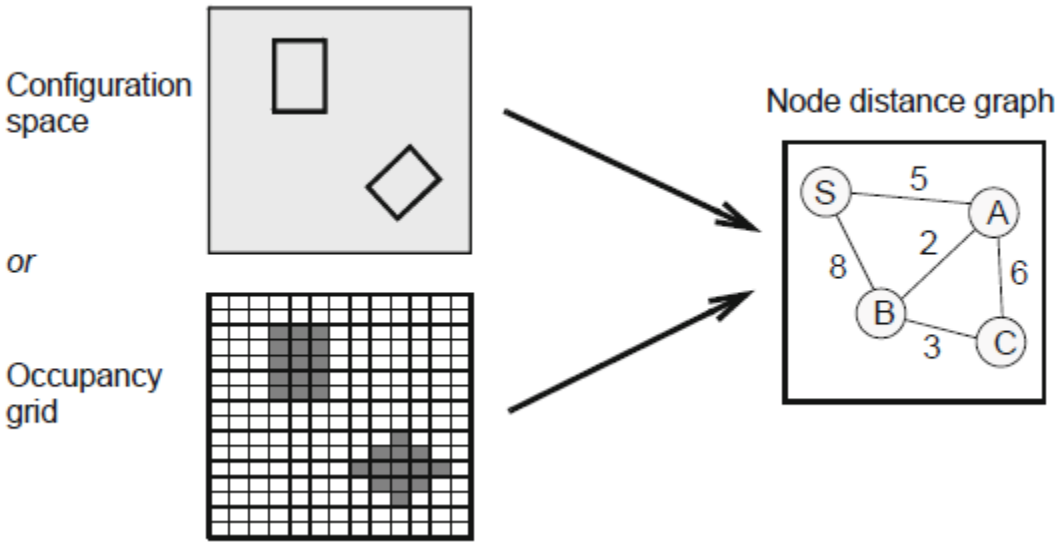


Figure 5.20 Basic environment representations

While many navigation algorithms work directly on the environment description (configuration space or occupancy grid), some algorithms, such as Dijkstra and A*, require a distance graph as input. A distance graph is an environment description at a higher level. It does not contain the full environment information, but it allows for an efficient initial path-planning step (e.g., from room to room) that can be subsequently refined (e.g., from x,y-position to x,y- position).

A distance graph contains only a few individually identified node positions from the environment and their relative distances. Neither of these two basic environment formats leads directly to a distance graph, and so we are interested in algorithms that can automatically derive a distance graph.

The brute force solution to this problem would be starting with an occupancy grid and treating each pixel of the grid as a node in the distance graph. If the given environment is in configuration space, it can easily be converted to occupancy grid by "printing it" on a canvas at the desired resolution.

However, this approach has a number of problems. First, the number of nodes in the resulting distance graph will be huge, and so this will be infeasible for larger environments or finer grids. Second, path planning in such a graph will result in suboptimal paths, as neighboring pixels have been transformed into neighboring graph nodes and therefore only support turning angles that are multiples of ±45º (eight nearest neighbors) of multiples of ±90º (four nearest neighbors).

Using a quadtree will improve this situation on both counts; it will have significantly fewer nodes and does not impose the turning angle restriction. To generate a quadtree, the given environment (in either configuration space or occupancy grid) is recursively divided into four quadrants. If a quadrant is either completely empty (free space) or completely covered by an obstacle, it becomes a terminal node, also called a *leaf*. Those quadrant nodes that contain a mix of free space and obstacles will be further divided in the next recursive step This procedure continues until either all nodes are terminal or until a maximum resolution is reached.

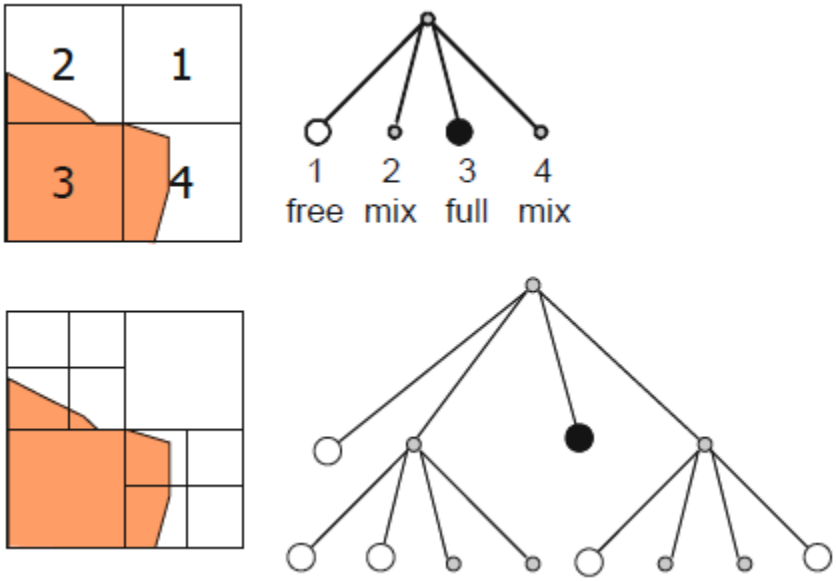


Figure 5.21 Quadtree construction

All free nodes of the quadtree (or more precisely their center positions) can now be used as nodes in the distance graph. We construct a complete graph by linking each node with each other, then eliminate those edges for which the corresponding two nodes cannot be linked through a direct line because of a blocking obstacle (e.g. lines c–e and b–e in the next figure). For the remaining edges we determine their relative distances by measuring in the original environment and enter these values into the distance graph (Figure 16.10, right). As the final path-planning step, we can now use, e.g., the A* algorithm on the distance graph.

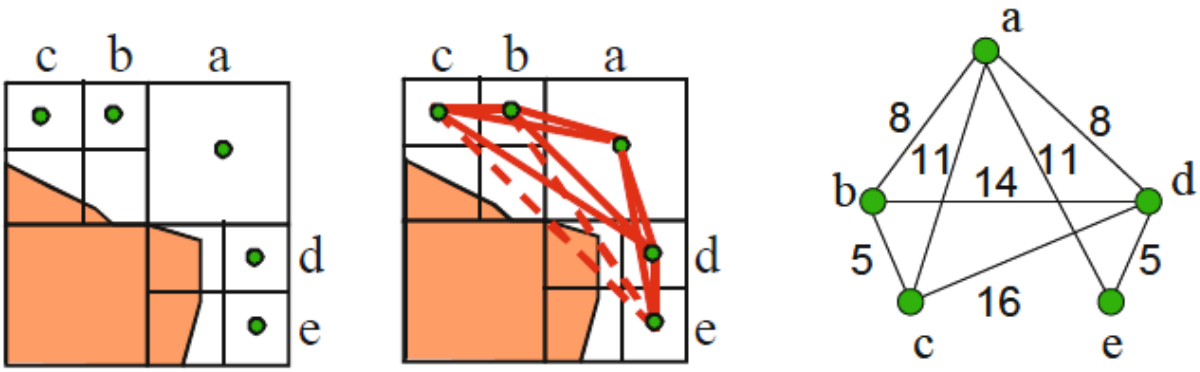


Figure 5.22 Distance graph construction from quadtree

5.4.1 2D Grid map using Laser sensor (LIDAR)

In ROS chapter, we described the LIDAR sensor in details. Here we will show how we use it in building 2D grid map using Slam-gmapping package using ROS distribution. This package uses Laser scan readings to build a 2D map and localization of the rover inside it. It uses the particle slam which is discussed before.

Figure 5.23 LIDAR sensor (laser sensor)

First we use YDLIDAR ROS package to get readings from the LIDAR (laser_scan topic).Then we use gmapping ROS package which uses readings from the LIDAR by subscribing to laser_scan topic which is now published by the YDLIDAR ROS package.

Let’s launch our Gazebo ROS simulation world.

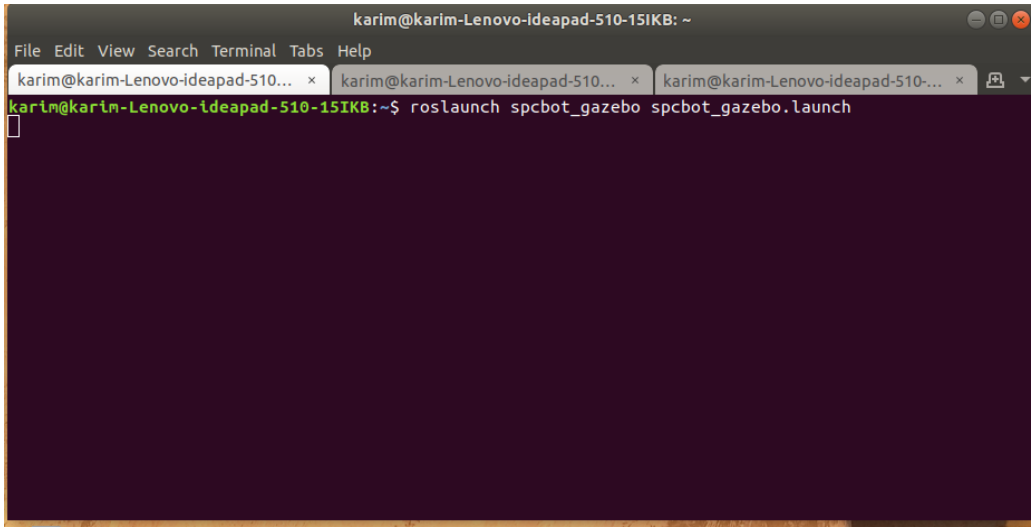


Figure 5.24 launch Gazebo simulation package

Let's take a look for the world that we used in our Gazebo simulation:

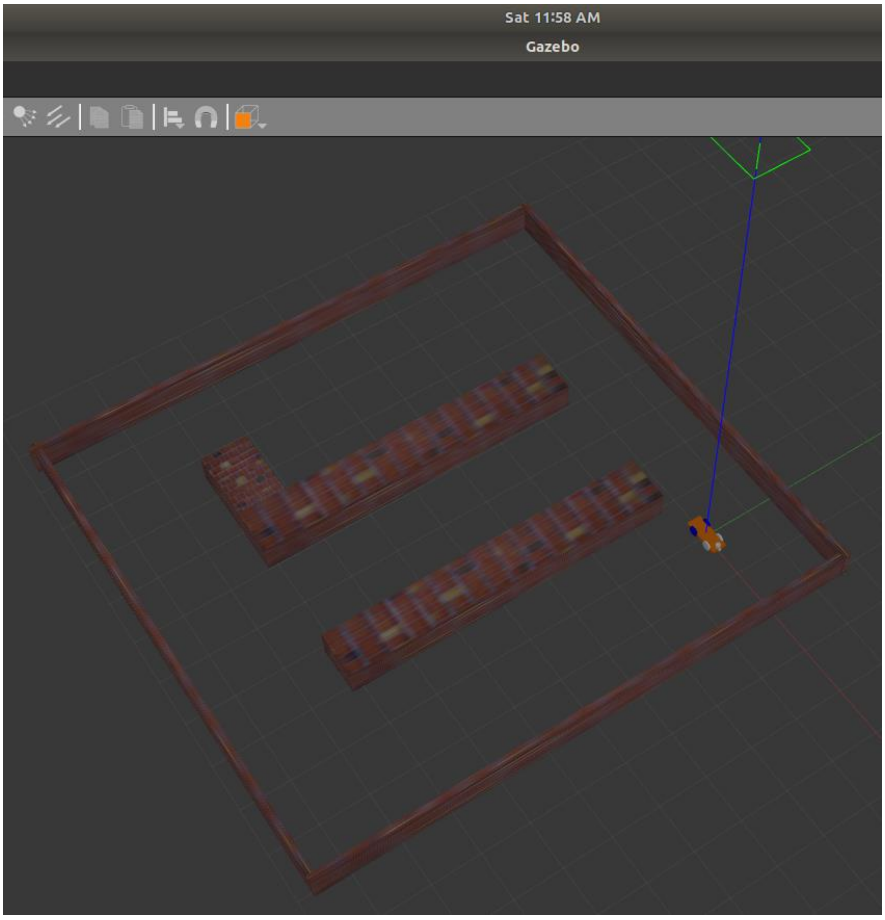


Figure 5.25 World simulated by Gazebo simulation

Let's launch our ROS package and see how the map would look like in RVIZ simulation ROS simulator. The next figure shows the laser scanner (LIDAR) readings that we got from YDLIDAR package.

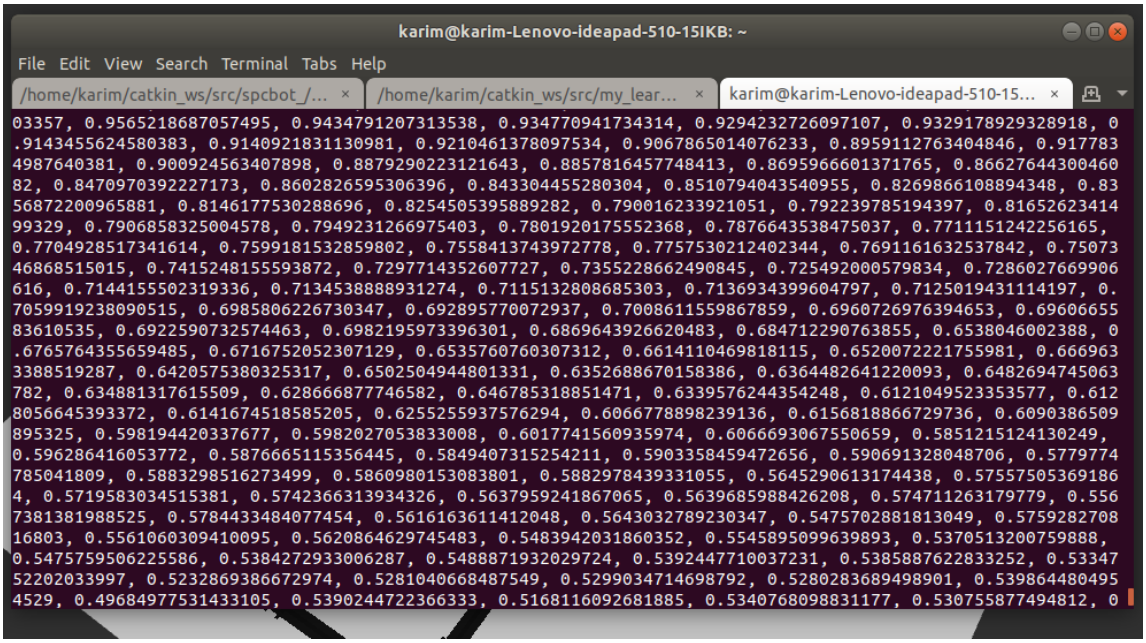


Figure 5.26 LIDAR readings

The next figure shows the simulated LIDAR readings that we use in gmapping ROS package to build our map.

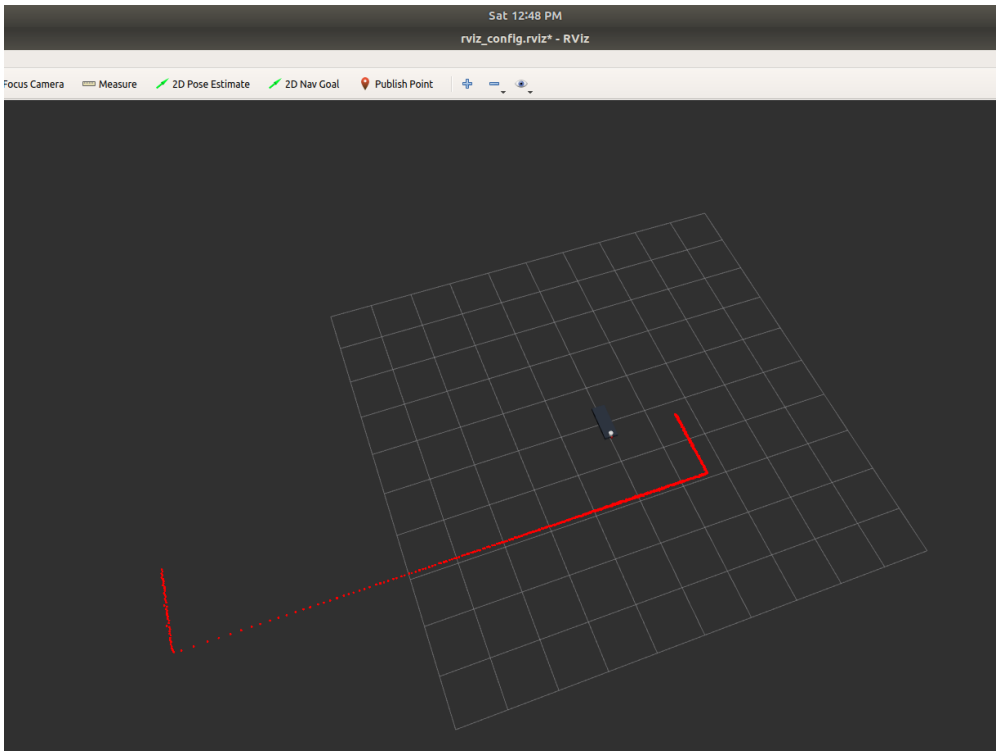


Figure 5.27 Laser scanned area (180º)

When starting the simulation, we will see the map as shown in the next figure.

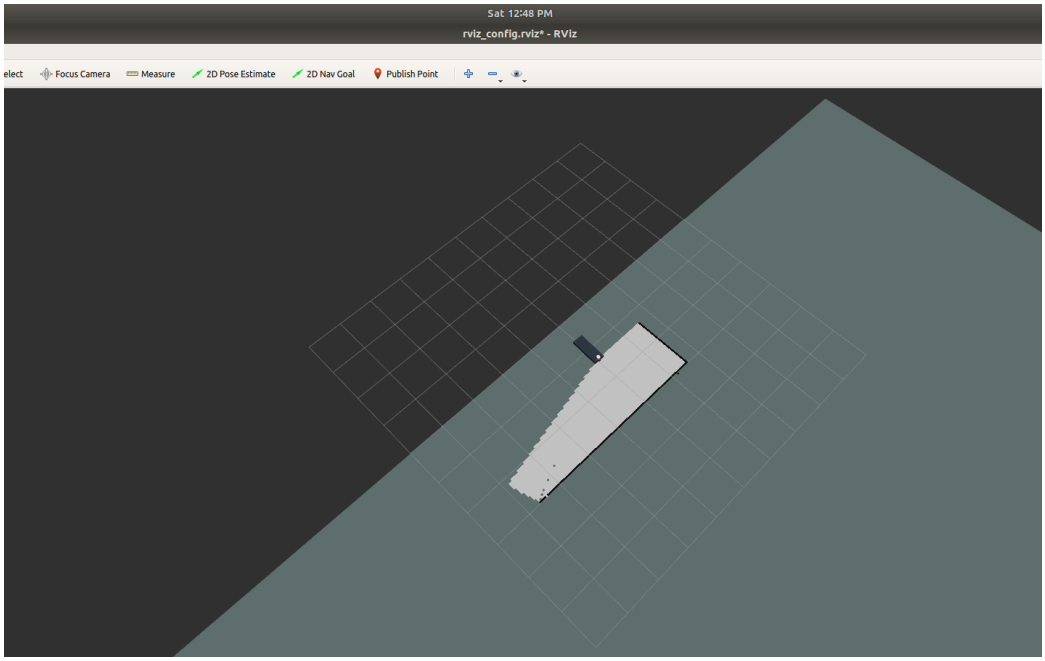


Figure 5.28 initial map

It’s better to build a complete map before using path planning algorithm, so we use exploration algorithms which would be discussed in the next chapter. The next figure shows the complete built map by gmapping.

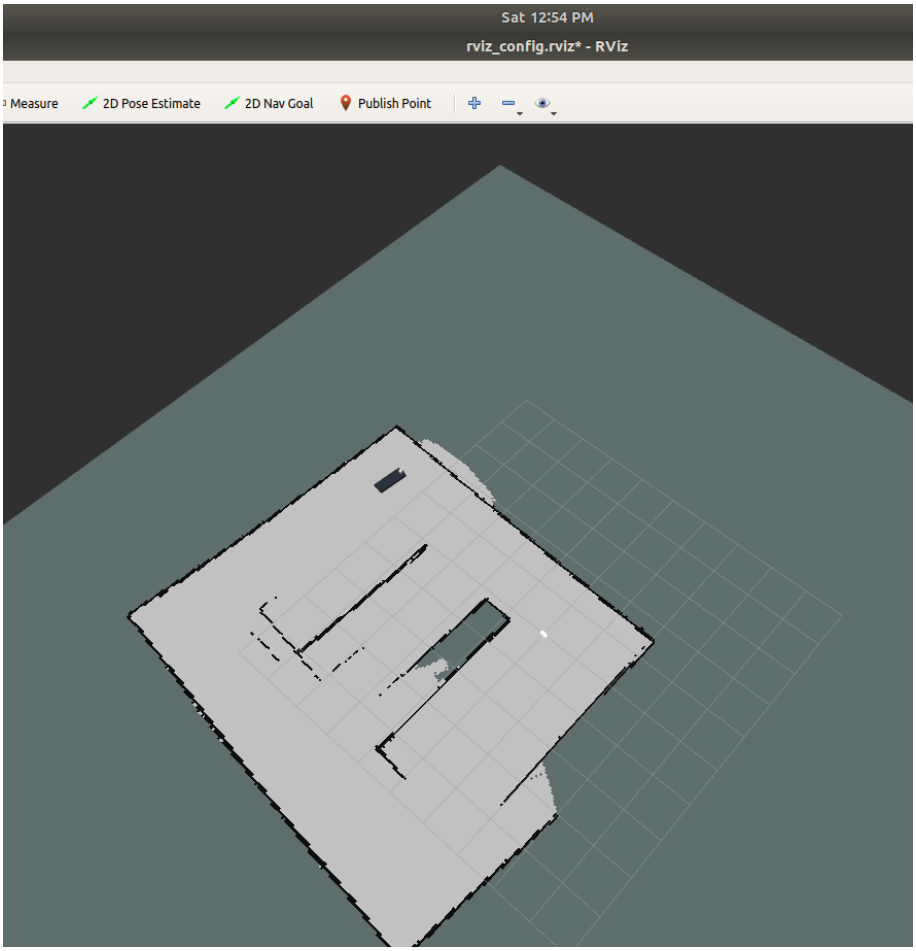


Figure 5.29 complete map built by LIDAR

After building the map we can get to the next step which is path planning as discussed in the next chapter.

The following figure shows the Global costmap built for the whole map which is used in path planning and navigation (Discussed in the next chapter).

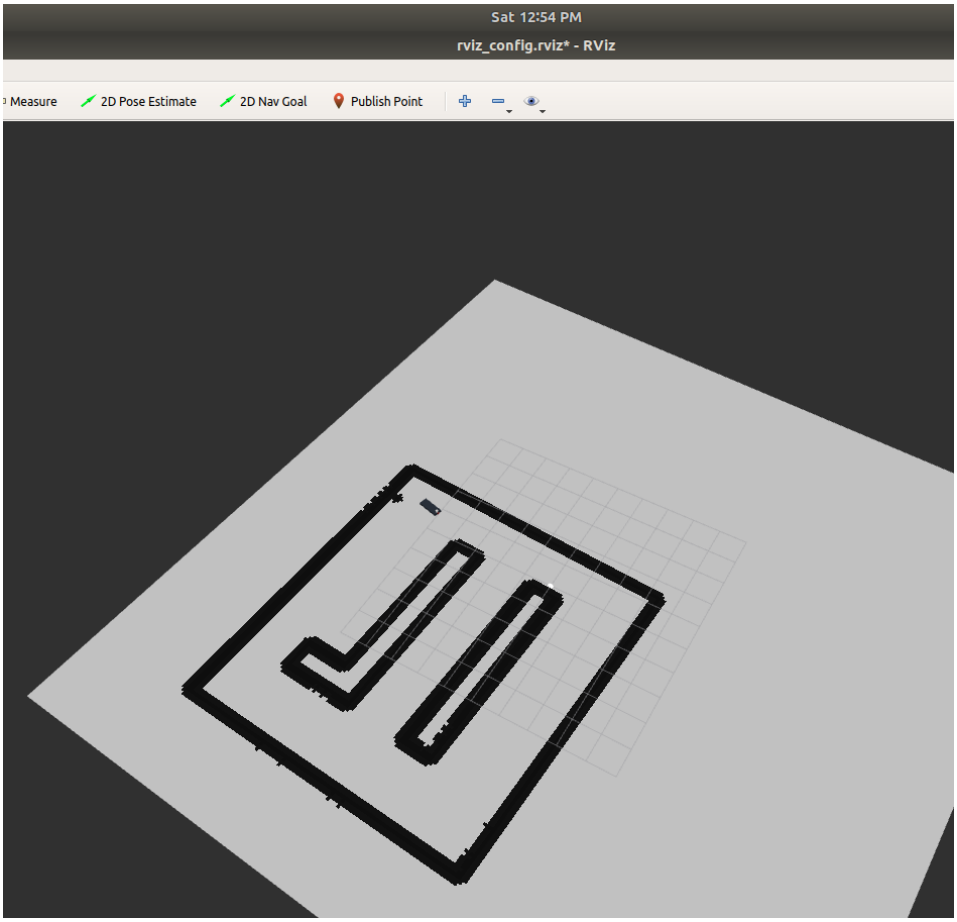


Figure 5.30 Global costmap

5.4.2 3D Grid map using RGB-D sensor (xbox360 Kinect)

In this part we will build a 3D map using depth camera (xbox360 Kinect). The Kinect camera gets colored images and depth of that images.

Figure 5.31 Kinect RGB-depth camera

In 3D slam we use two packages, the first to get readings from the Kinect and the second one to perform RGB-D SLAM.

The Package which is used to get readings from the Kinect camera; or sensors of the same type; is called OpenNI.

The next figure shows the readings come from the camera as we subscribe RGBD image topic using the terminal.

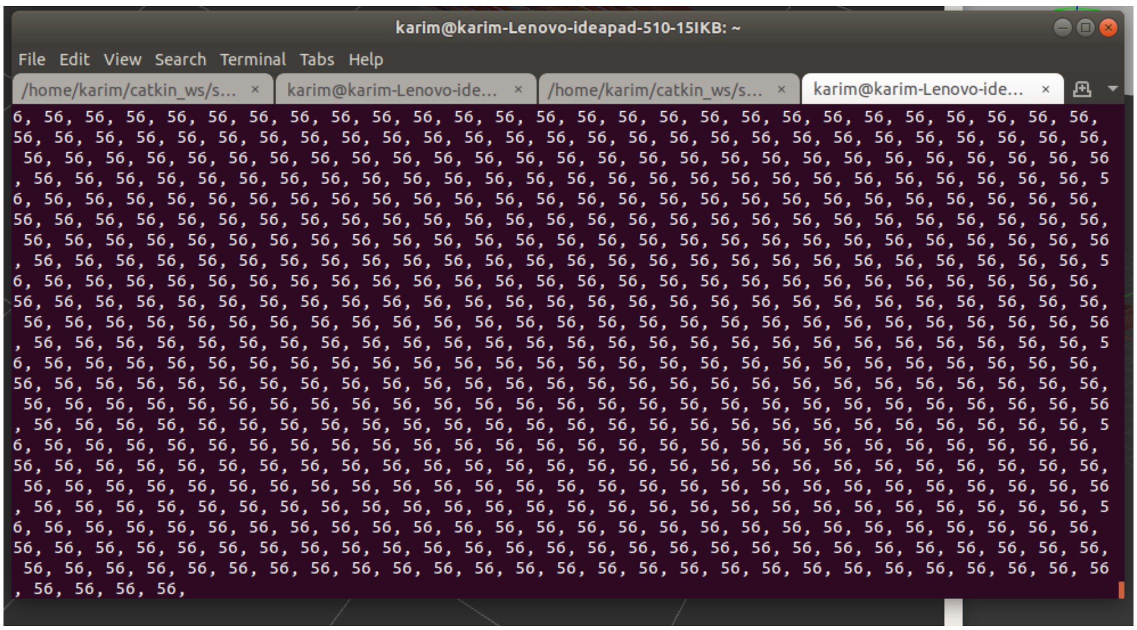


Figure 5.32 RGB-D readings from Kinect camera

The second package used is RTABmap_ros (Real-Time Appearance-Based Mapping). It uses this reading that we get from the Kinect camera and perform RGB-D SLAM on it to get accurate map. Let's take a quick look on the RGB-D SLAM.

<u>RGB-D SLAM:</u>

RGB-D mapping depends on loop closure detection approach. Loop closure detection is the process involved when trying to find a match between the current and a previously visited locations in SLAM (Simultaneous Localization and Mapping).

Over time, the amount of time required to process new observations increases with the size of the internal map, which may affect real-time processing. RTAB-Map is a novel real-time loop closure detection approach for large-scale and long-term SLAM. RTAB-map approach is based on efficient memory management to keep computation time for each new observation under a fixed time limit, thus respecting real-time limit for long-term operation. Results

demonstrate the approach's adaptability and scalability using two custom data sets and ten standard data sets.

Let’s see a realistic 3D map build using RTABmap_ros package. This package can be used to generate a 3D point clouds of the environment and/or to create a 2D occupancy grid map for navigation.

Figure 5.33 pointcloud example

The next figures show a realistic point cloud image taken from Kinect camera.

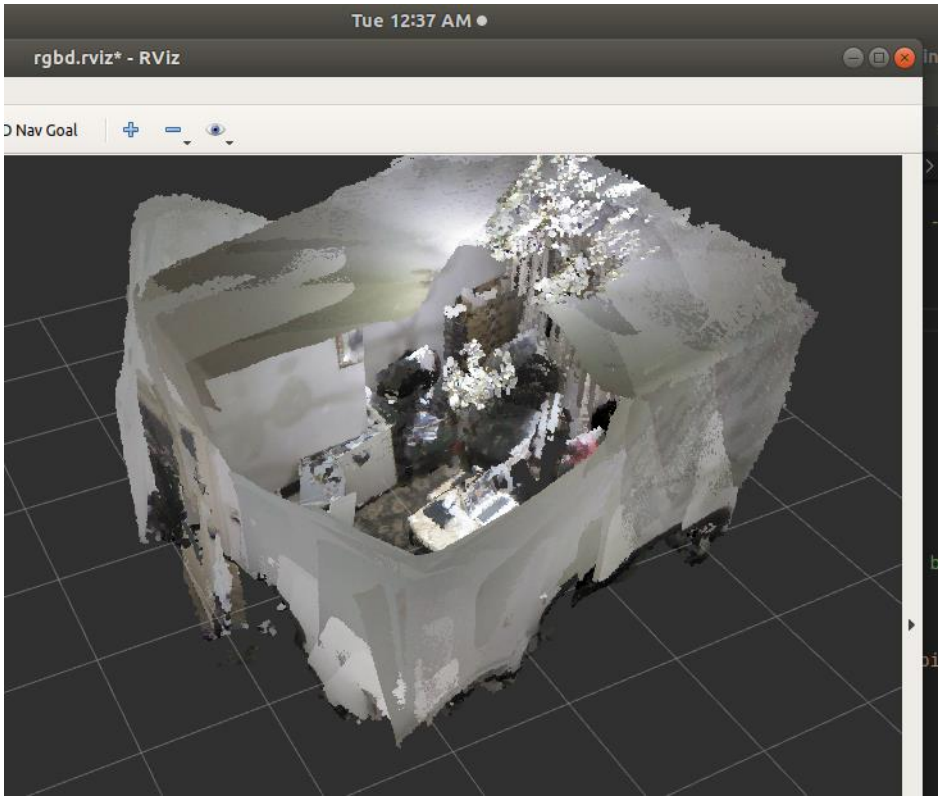


Figure 5.34 Full 3D pointcloud map of a room

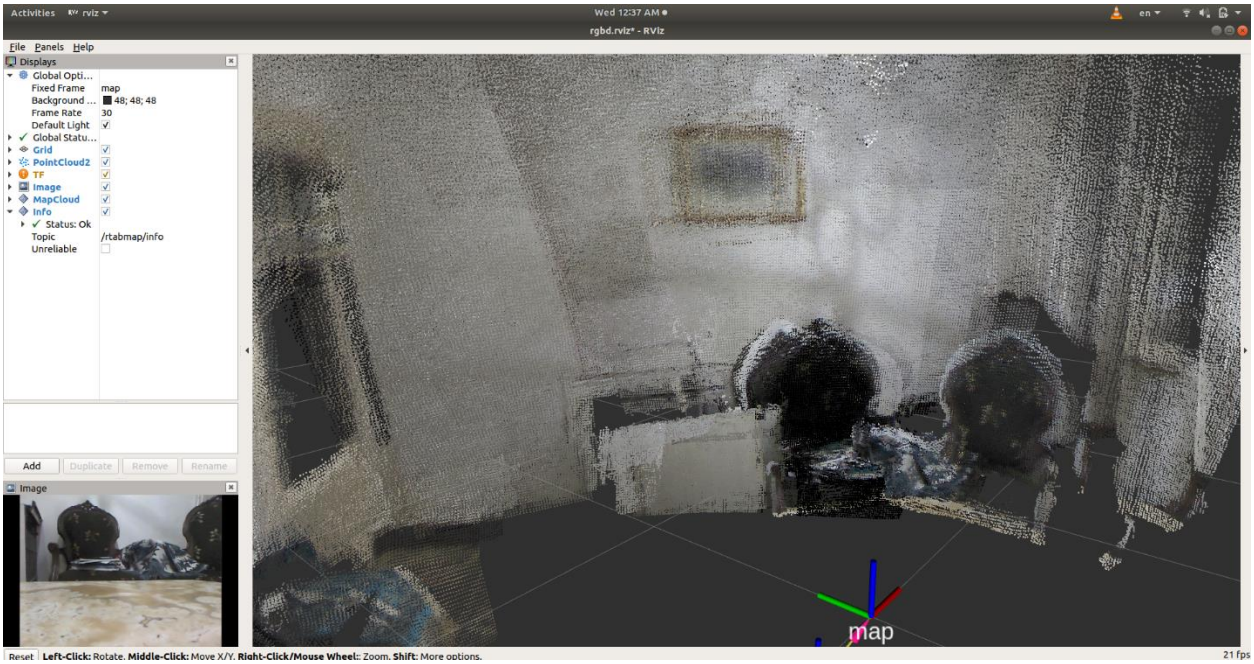


Figure 5.35 a view inside the 3D map of the room

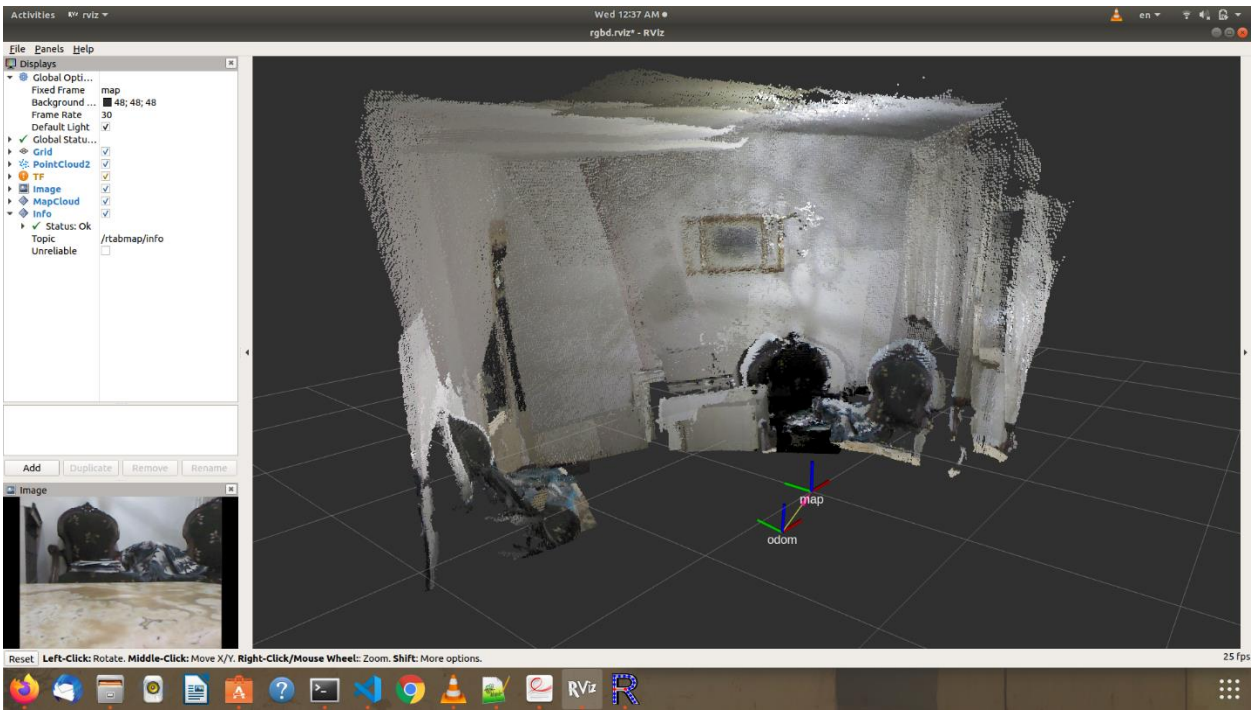


Figure 5.36 a vier inside the 3D map of the room

Using the same package for mapping using simulated world Gazebo simulation as shown:

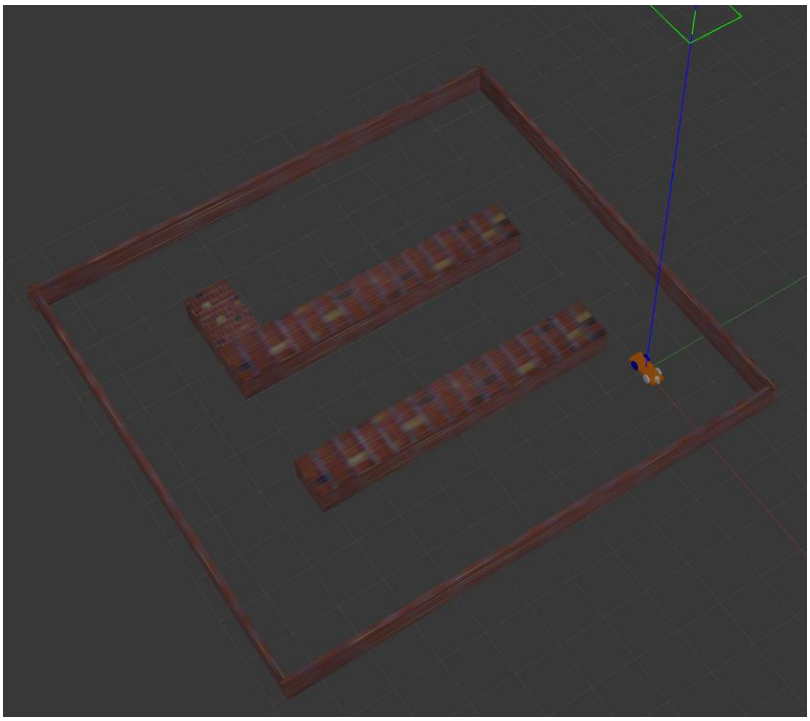


Figure 5.37 Gazebo world used

First, Let's launch our ROS package and see how the map would look like in RVIZ simulation ROS simulator:

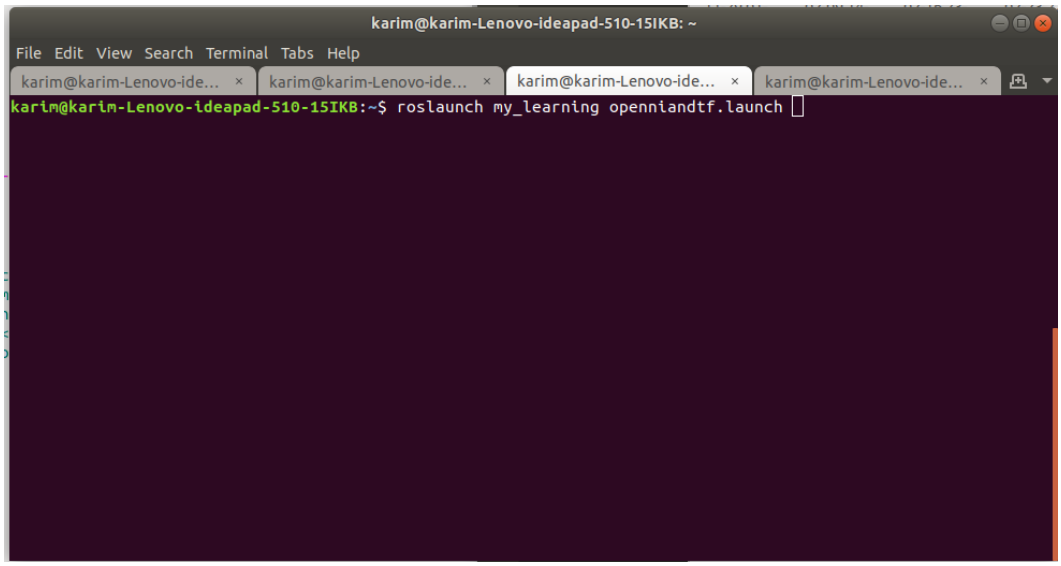


Figure 5.38 Launching the Package which is responsible for building a map using kinect camera

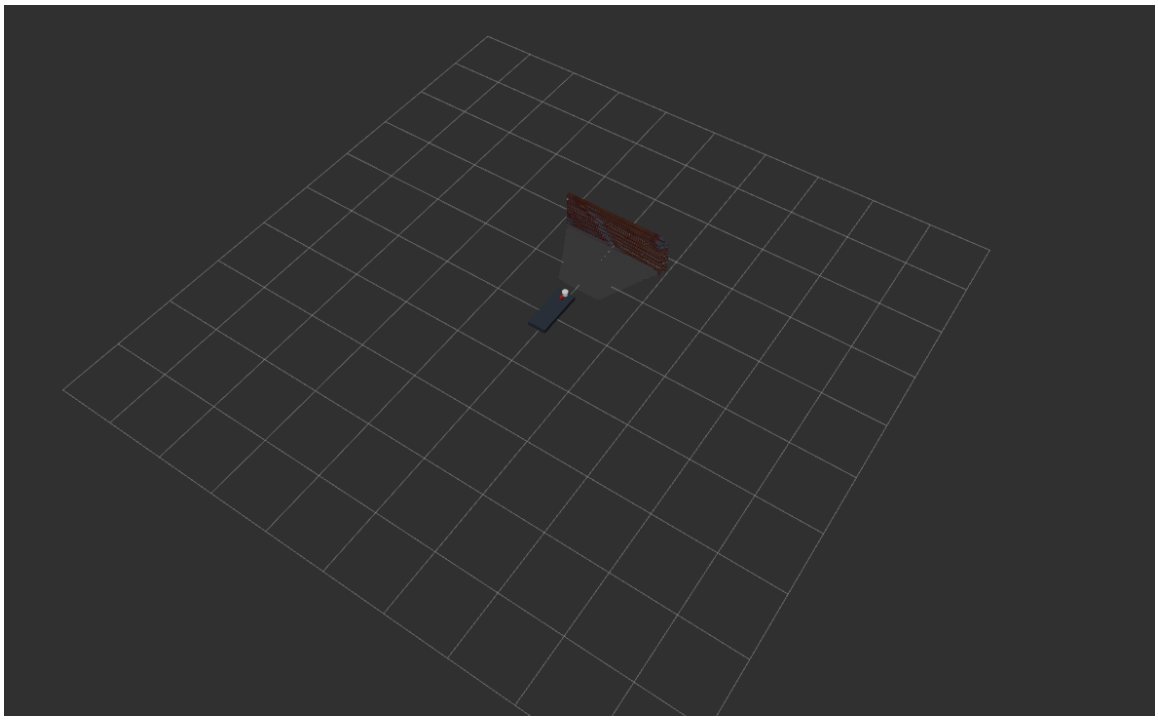


Figure 5.39 Initial 3D pointcloud map of the simulated world

After some exploration the map would look like:

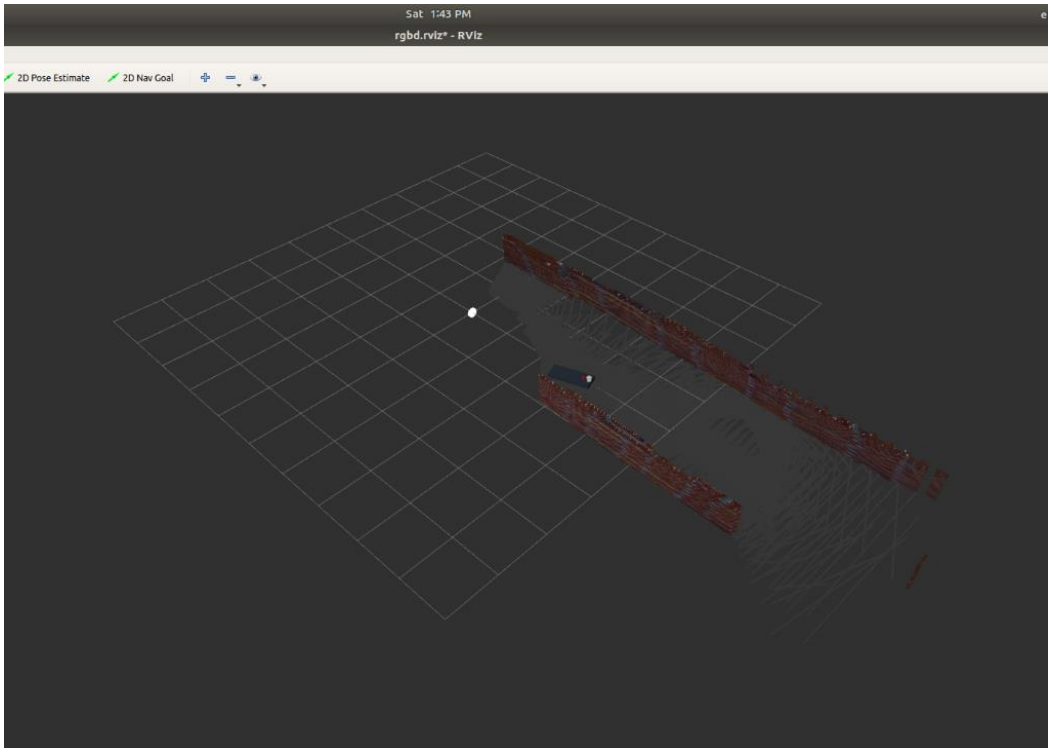


Figure 5.40 3D map after some exploration

5.5 Comparison between 2D and 3D maps

1- 2D LIDAR sensor is more accurate than Kinect sensor, covers more distance and covers an angle of 360^0 of the surroundings. The Kinect sensor covers only about 70^0.
2- Kinect camera is better for obstacle avoidance as it detects the height of the obstacle, so the rover could decide if he could pass beneath it or not.
3- The Kinect camera has much noises and larger errors than LIDAR sensor.
4- Kinect camera needs to move much slower to avoid loop closure failure on creating the map.
5- You must track the camera readings using RTAB-mapviz simulation platform. (RTAB-mapviz is a simulation platform shows pointcloud maps, the current image frame and loop closure status detection).

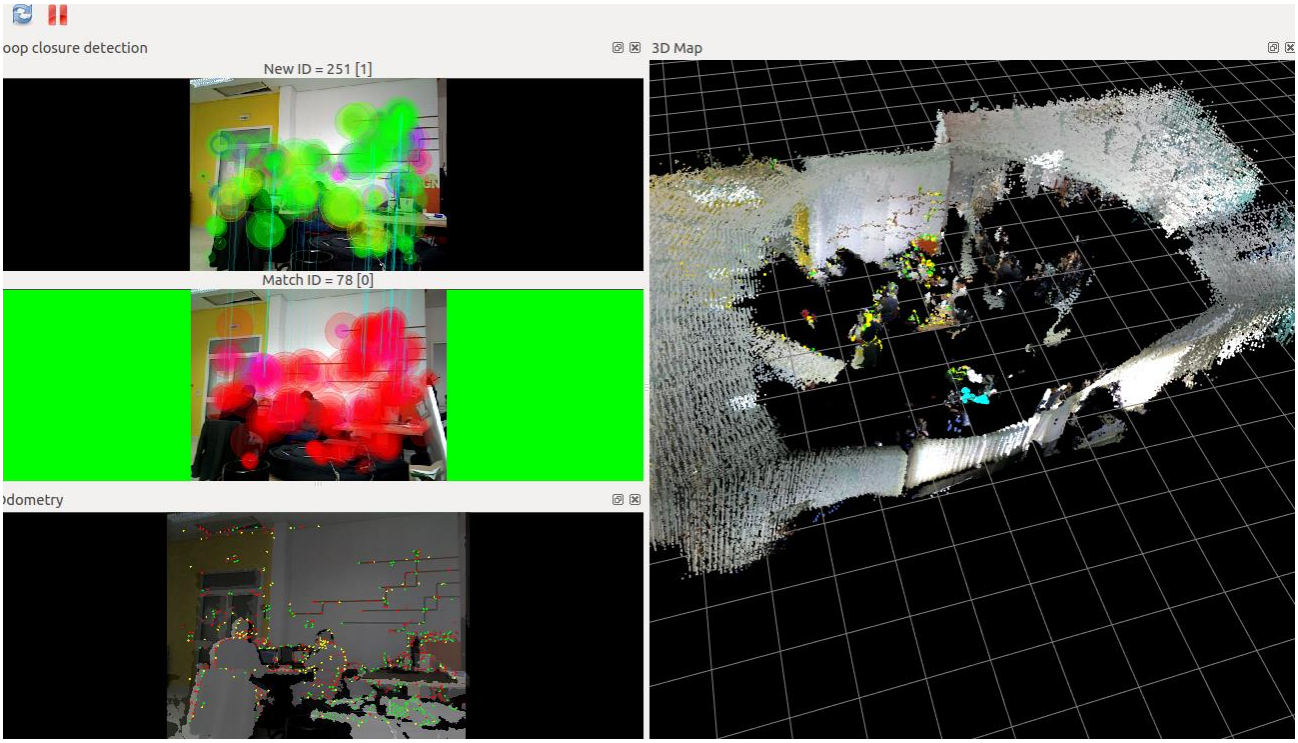


Figure 5.41 RTAPmapviz

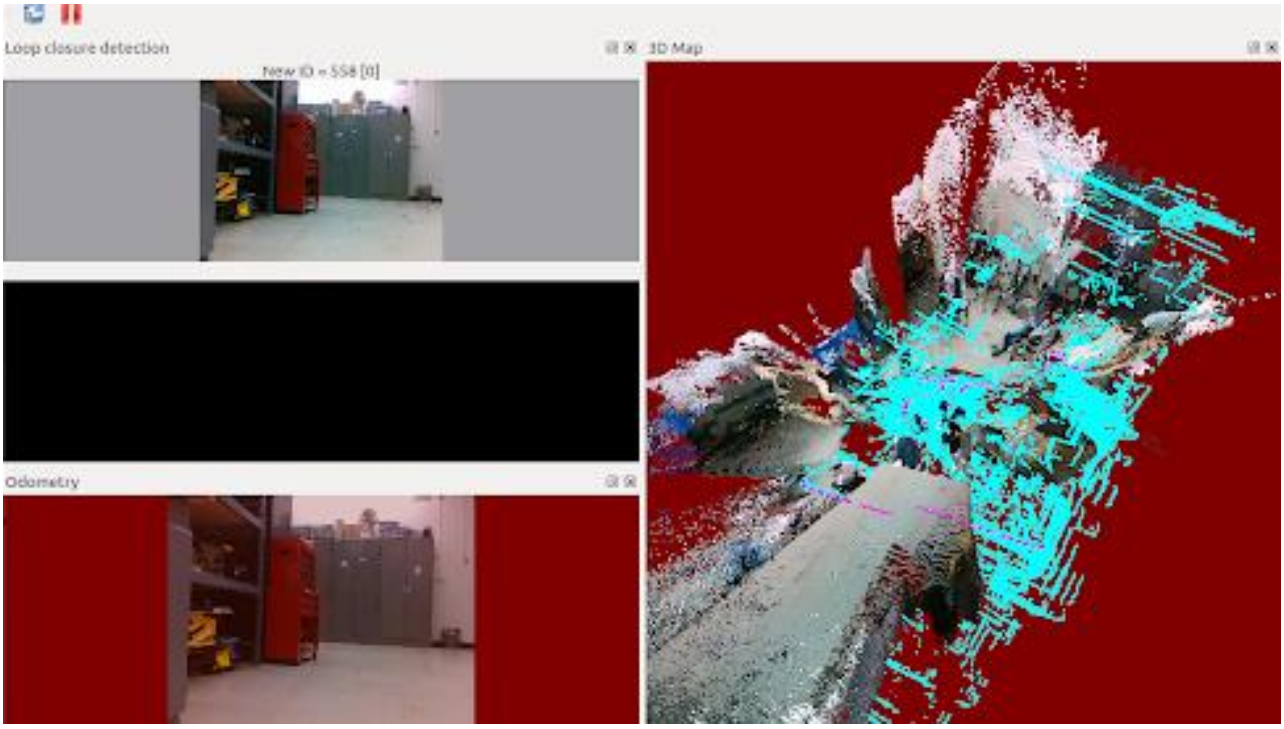


Figure 5.42 RTABmapviz loop closure failure

5.6 Conclusion

It's better to use a 2D laser scanner as LIDAR in 2D mapping and navigation. We can use a non-accurate image depth sensor as Xbox360 Kinect in short range obstacle avoidance as it can detect the dimensions of the obstacle. So, using the data which we get from a Kinect sensor, we can use it to pass beneath an obstacle which presents at a height more than the rover height. Also, a laser scanner can't detect obstacles at another level from that of the 2D LIDAR, so if we depend on only the laser readings, the rover may collide with a high obstacle but shorter than the rover (example: a short table). So, the presence of such two types of sensors is essential for doing SLAM and path planning algorithms.

Chapter 6

6. Motion Planning

6.1 Introduction

Moving from one place to another is a trivial task, for humans. One decides how to move in a split second. For a robot, such an elementary and basic task is a major challenge. In autonomous robotics, path planning is a central problem in robotics. The typical problem is to find a path for a robot, whether it is a vacuum cleaning robot, a robotic arm, or a magically flying object and surly mars rover vehicle, from a starting position to a goal position safely. The problem consists in finding a path from a start position to a target position. This problem was addressed in multiple ways in the literature depending on the environment model, the type of robots, the nature of the application, etc.

Safe and effective mobile robot navigation needs an efficient path planning algorithm since the quality of the generated path affects enormously the robotic application. Typically, the minimization of the traveled distance is the principal objective of the navigation process as it influences the other metrics such as the processing time and the energy consumption.

to solve the robot navigation problem, we need to find answers to the three following questions:

Where am I?

Where am I going?

How do I get there?

These three questions are answered by the three fundamental navigation functions:

localization,

mapping,

motion planning, respectively.

- Localization: means that robot knows its location W.R.T. the environment around it and for this mission sensors are used in our mars rover we used two main sensors for this mission Depth camera (Kinect camera) and Lidar where both provide us with good odometry which get more better with the useful math in filtering these data like BAYES filters (KALMEN filter)

- Mapping: The robot requires a map of its environment in order to identify where he has been moving around so far. The map helps the robot to know the directions and locations. The map can be placed manually into the robot memory (i.e., graph representation, matrix representation) or can be gradually built while the robot discovers the new environment. Mapping is an overlooked topic in robotic navigation.

- Motion planning: To find a path for the mobile robot, the goal position must be known in advance by the robot, which requires an appropriate addressing scheme that the robot can follow. The addressing scheme serves to indicate to the robot where it will go starting from its starting position. For example, a robot may be requested to go to a certain room in an office environment with simply giving the room number as address. In other scenarios, addresses can be given in absolute or relative coordinates

6.2 Path planning algorithms

There are different types of path planning algorithms could used to perform the need of get a feasible path to go through. Now we will distinguish between two main type of methods used global path planning and local path planning

Global path planning: the global planner requires a map of the environment to calculate the best route. Its main job is to draw a feasible path from the source(start) point to the goal(end) point discarding the accuracy of obstacles existing in the way

Local path planning: In order to transform the global path into suitable waypoints, the local planner creates new waypoints taking into consideration the dynamic obstacles and the vehicle constraints. So, to recalculate the path at a specific rate, the map is reduced to the surroundings of the vehicle and is updated as the vehicle is moving around. It is not possible to use the whole map because the sensors are unable to update the map in all regions and a large number of cells would raise the computational cost. Therefore, with the updated local map and the global waypoints, the local planning generates avoidance strategies for dynamic obstacles and tries to match the trajectory as much as possible to the provided waypoints from the global planner

Now we will study our case and its needs to specify the needed algorithms, we started from the basic motion planning algorithms which it doesn't required a map to start with these algorithms called bug algorithms which depends mainly on two main scripts go to point and follow walls

And there are three types of bug algorithms bug0, bug1 and bug2.

Then we go up for more complicated algorithms which require maps like A* (A star) algorithm. The following part we will illustrate the theory of each algorithm and how to do it and the codes we built

6.2.1 Follow wall

We depend on lidar sensor to get laser scan topic which is published by lidar node then we performed some modification on it

Lidar has a property called sensitivity which describes how many readings that published per one degree angle out lidar publish 720 readings per 180 deg

So we divide these readings to 5 parts each part has 144 readings from 720/5 = 144

Then we have 5 parts front, front left, front right, left and right such like the following figure

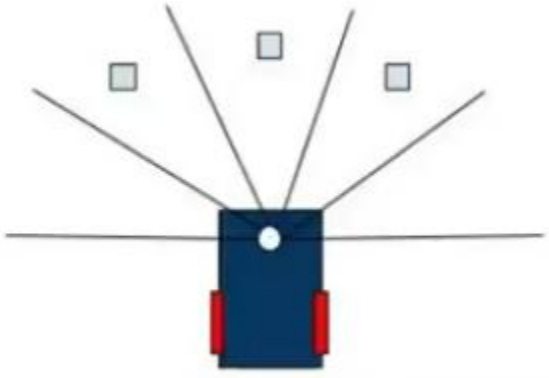


Figure 6.1 laser scan parts

Then we consider the following three parts to depend on sensing obstacles front, front left and front right

So, we have eight different cases of obstacle existence specified in the following table :

Case	Front left	Front	Front right	Task
1				Find wall
2		X		Turn left
3			X	Go parallel
4	X			Find wall
5		X	X	Turn left
6	X	X		Turn left
7	X	X	X	Turn left
8	X		X	Find wall

'X' defines that there is obstacle in this side

In our case we determined that if obstacle is far less than 1.5 m that means it's considered as an obstacle

Then the algorithm of follow the wall consists of three main functions find wall, make the wall parallel and follow the wall

First if the robot senses no obstacle just like case #1 the robot will drive forward and have a little yaw angle until it senses an obstacle

Then it will go through one of the rest cases if there just one obstacle on front only just like case #2 or case #3 or case #4 it will turns make wall parallel in which the robot turns around until the wall get in right or left side

Then the robot turns to third part follow the wall until it reaches obstacle end

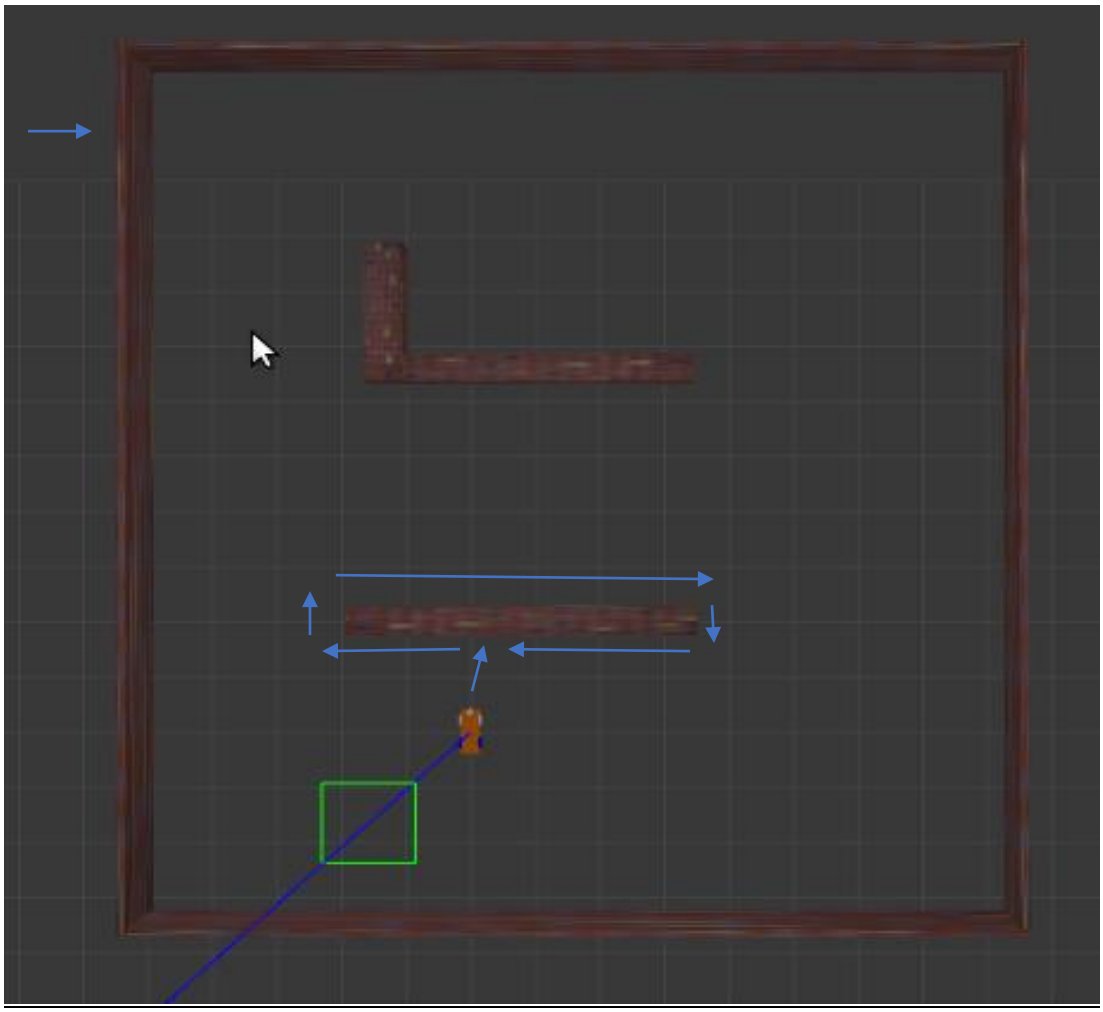


Figure 6.2 follow wall test

6.2.2 Go to point

This algorithm tends to make robot go to goal (end point) from source (start point) using just geometry of source and goal points

The algorithm consists of three main functions fix heading, go straight and Done

- **Fix Heading**: Denotes the state when robot heading differs from the desired heading by more than a threshold
- **Go Straight**: Denotes the state when robot has corrected heading but is away from the desired point by a distance greater than some threshold
- **Done**: Denotes the state when robot has corrected heading and has reached the destination.

The following flow chart represents the algorithm

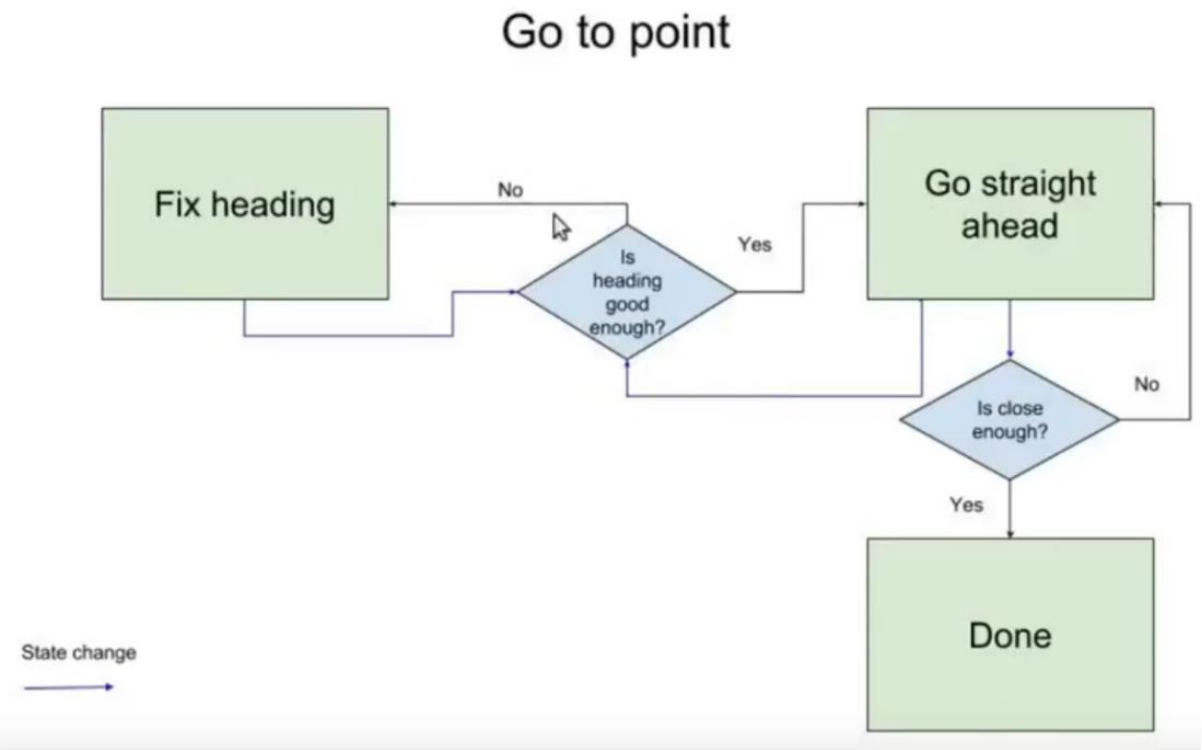


Figure 6.3 go to point flow chart

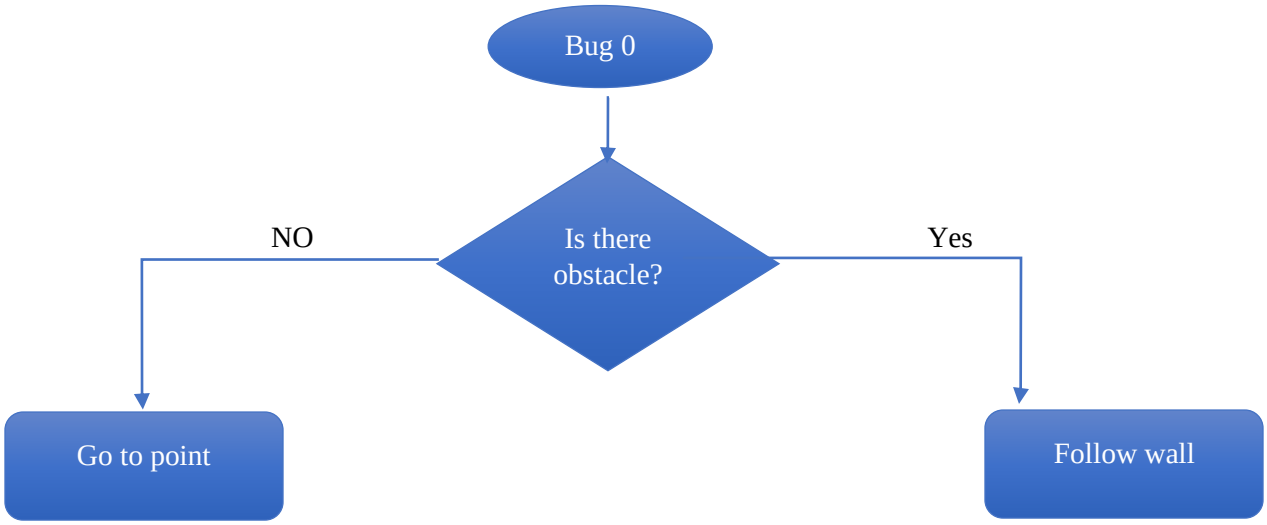
The previous simple algorithms could generate more complex algorithms that what we doing next with bug algorithms

6.2.3 Bug 0 algorithm

Bug 0 algorithm depends on the previous two algorithms follow wall and go to point it just follow the following two steps

- Head to goal (go to point)
- Follow wall of obstacle (follow wall)

It first set heading to goal point until it found a wall then robot follows that wall and detect if the path to goal is free of obstacles then it returns to first algorithm go to point
The following chart shows that



Algorithm overview:
It's simple algorithm that satisfy basic need of navigation and the following case shows how its simple to navigate to goal as it starts from source with go to point algorithm until it finds a wall of

obstacle then the robot follows it until it finds the pathway to goal is clear from obstacles then it turns to go to point algorithm and repeat the steps to reach the goal point
The following figure shows how rover be able to go from its state to the goal point (red point) with bug 0 algorithm first it uses go to point algorithm shown with bule arrows until it finds an obstacle then uses the wall follower algorithm shown with orange arrows until the wall has ended then returns to go to point algorithm and start fixing heading to goal point and repeats the same sequence when it faces the second obstacle

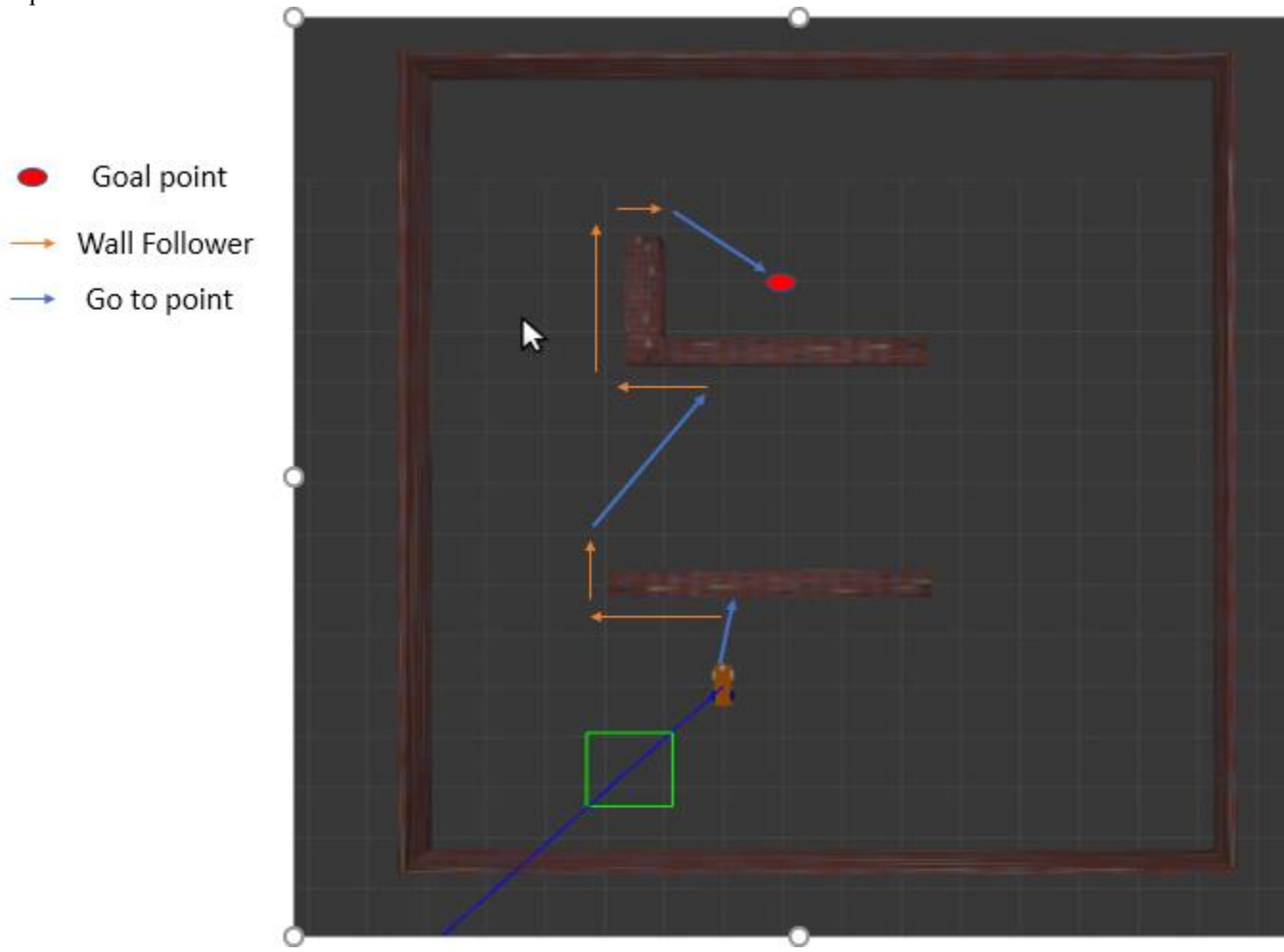


Figure 6.4 bug 0 test

but it has enormous cases that it fails to reach its goal just like the following world case

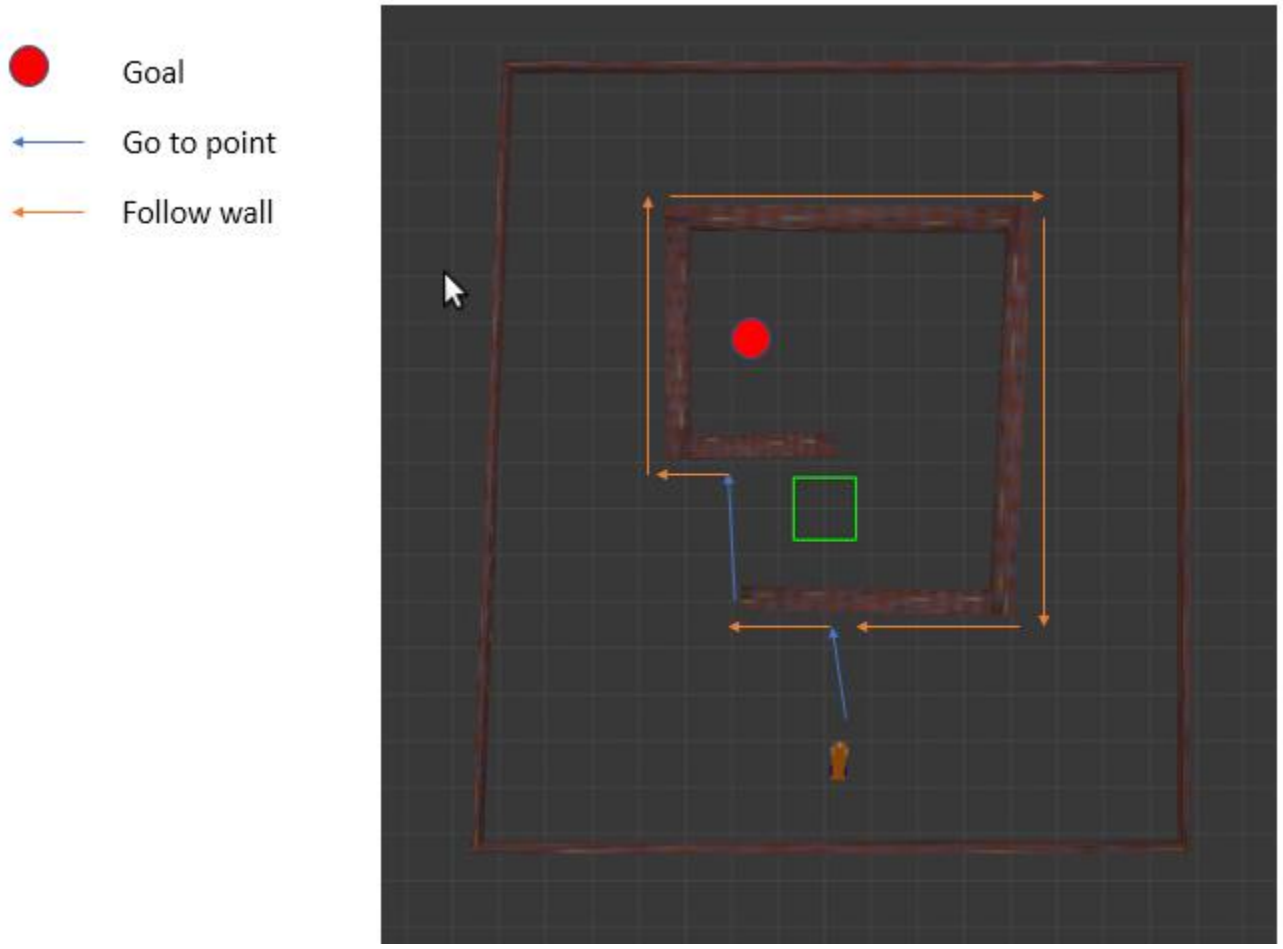


Figure 6.5 bug 0 fail case

The main reason for this failure is the follow wall algorithm always follow walls from the same side in our case is the left side and the arrangement of obstacle is closed from all side but one so the robot is stuck in follow wall algorithm and can't get out of it

So that bug 0 algorithm is not reliable algorithm that you could depend on it, so next bug 1 algorithm that has more improvements to solve bug 0 problems

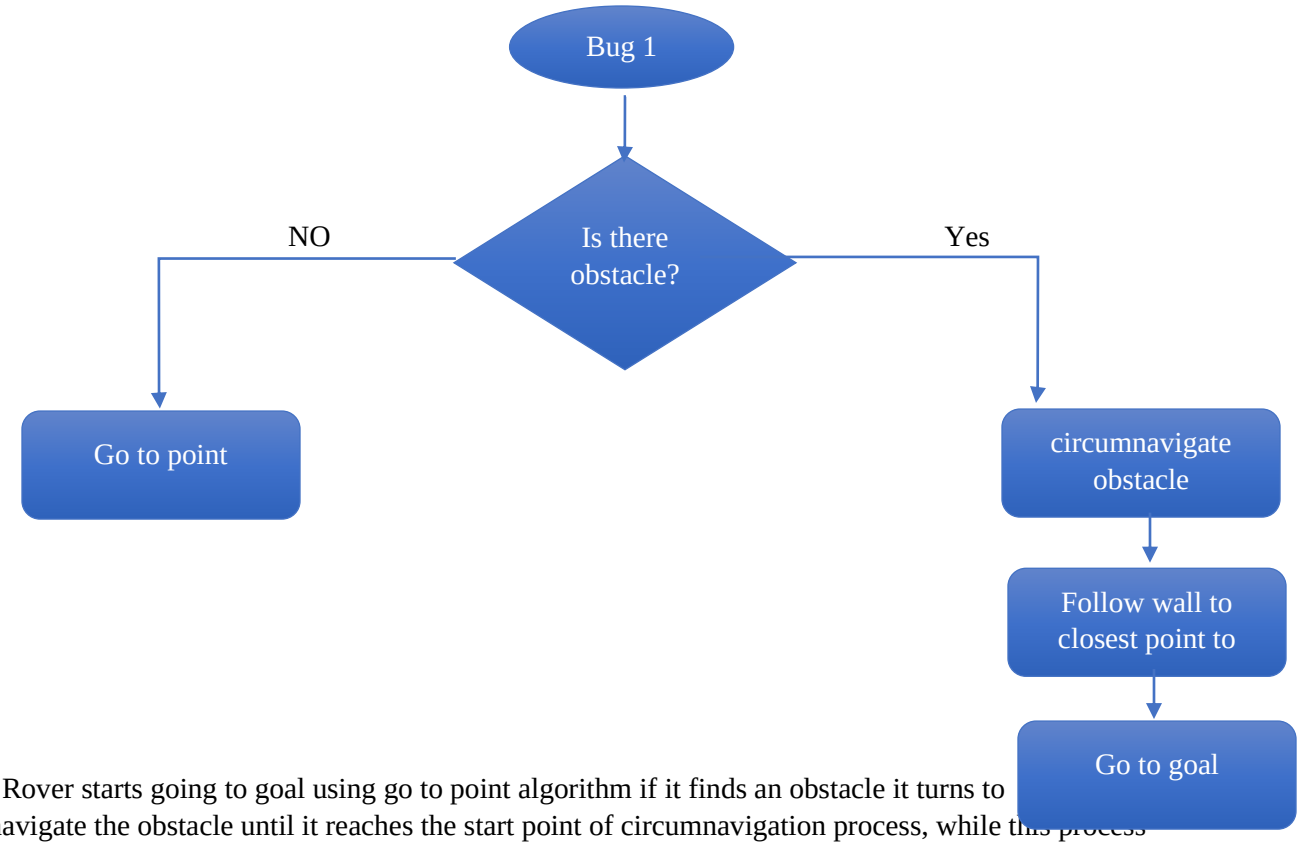
6.2.4 Bug 1 algorithm

Bug 1 algorithm is more advanced from bug 0 algorithm but also slower to reach final goal but it is reliable specially in world looks like puzzles

It depends on three main functions:

- Go to point
- Circumnavigation
- Follow wall to closest point to goal

The following flow chart show the algorithm



Rover starts going to goal using go to point algorithm if it finds an obstacle it turns to circumnavigate the obstacle until it reaches the start point of circumnavigation process, while this process the rover collects data about how far the point where it exists from the goal point

When rover reaches the start point of circumnavigation process started it turns to go to point that is the closes point to the goal point then it turns to go to goal point

The pervious process grantee that the rover will pass the obstacle even if the pathway is over estimated so it's very slow algorithm to use also but it solves problems that faces bug 0 algorithm when the map was somehow puzzle

The following figure shows the case which bug 0 fails how bug 1 solves it

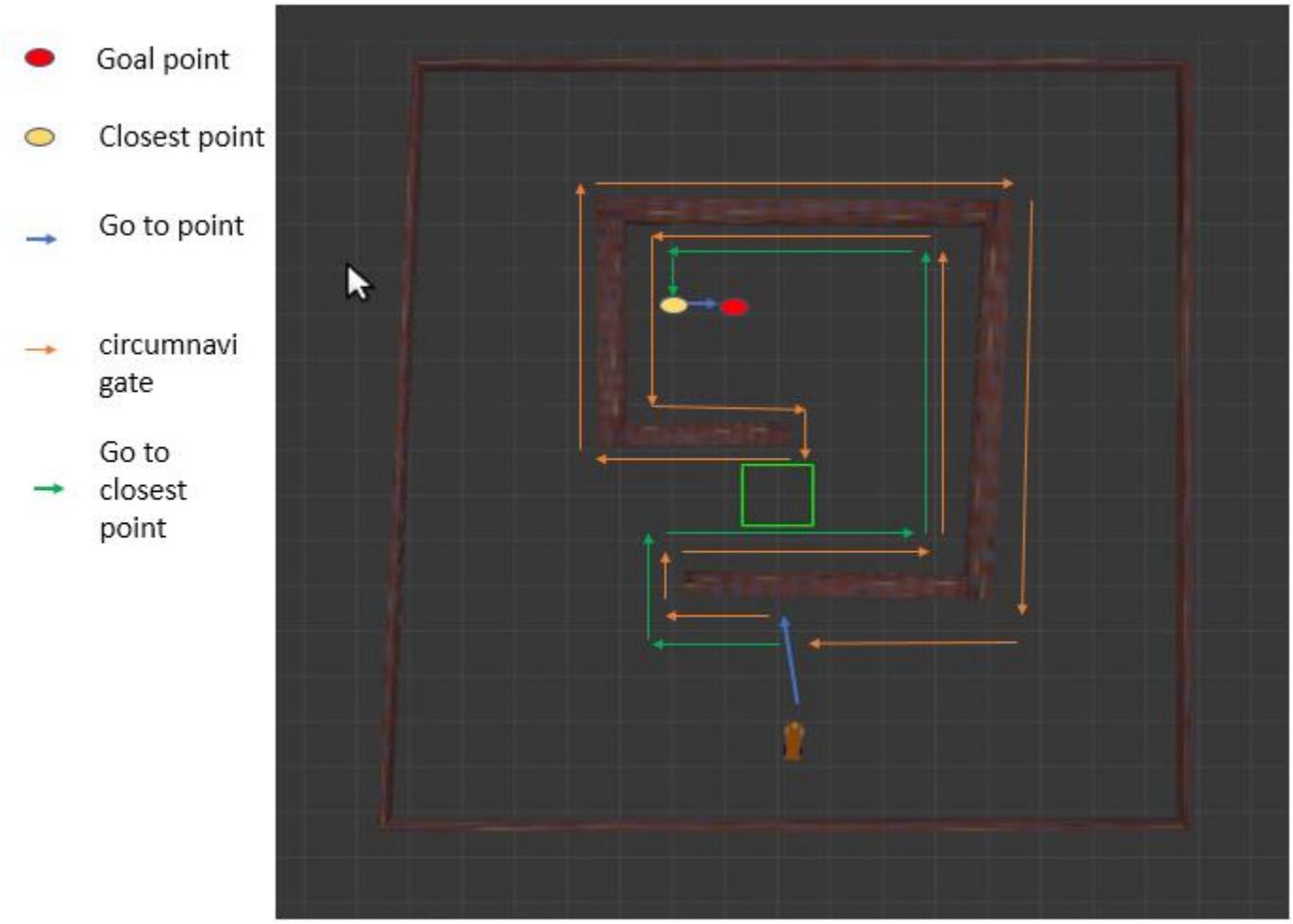


Figure 6.6 bug 1 test case

The previous figure shows how far the pathway needed to solve this problem, but it all because of independency of maps that all bugs algorithms independent of maps so it has no long term vision of how to generate paths that be more efficient

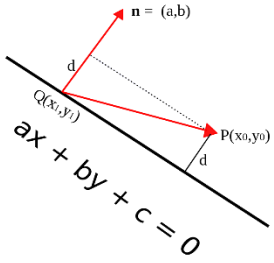
6.2.5 Bug 2 algorithm

Bug 2 algorithm is the final algorithm that developed for bug algorithms in motion planning and it depends mainly of three part:

- Go to point algorithm
- Wall following
- Go to global pathway

This algorithm starts by drawing a global path from start point to goal point which is a straight line discarding any obstacles on this pathway then rover has two main tasks in order first go to this global path through measure distance from its position to the line using the following formula

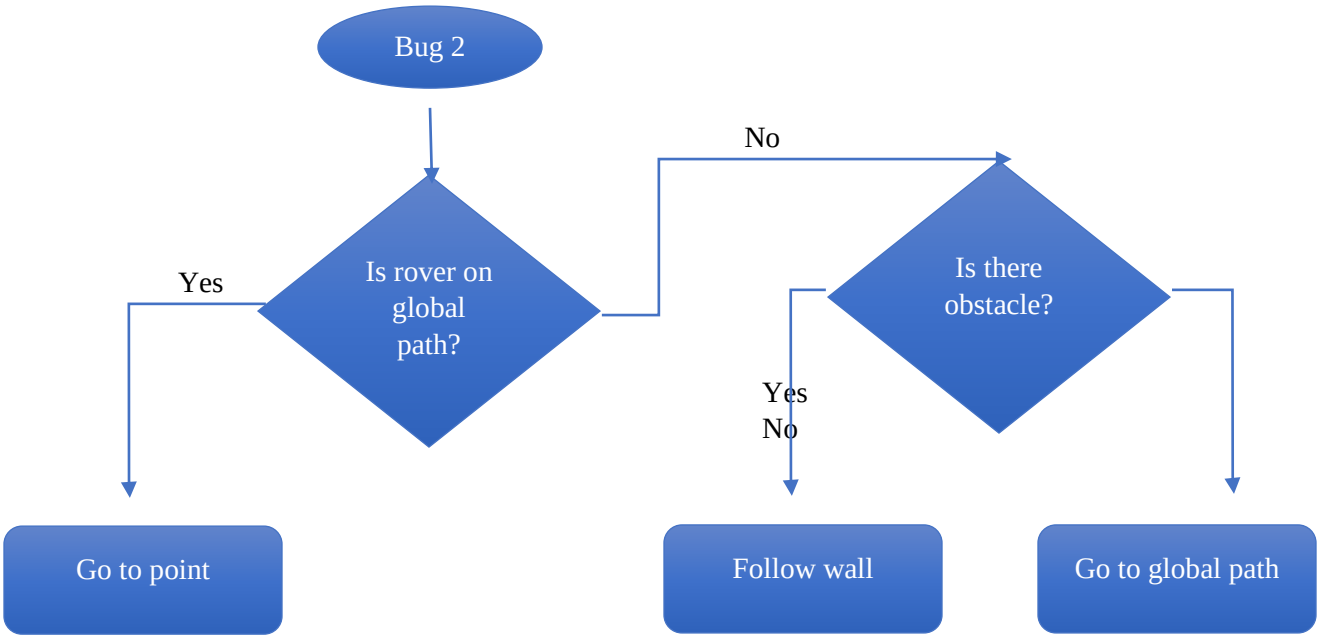
$$d = \frac{|ax_o + by_0 + c|}{\sqrt{a^2 + b^2}}$$



Then if rover be on this line it turns to go to point algorithm so it fixes heading towards the goal point

Then if rover finds an obstacle it turns to wall follower algorithm until the obstacle end it turns to go to the global pathway then go to goal point and repeat until reach the goal

The following flow chart shows the algorithm



The following test case shows the sequences of bug 2 algorithm

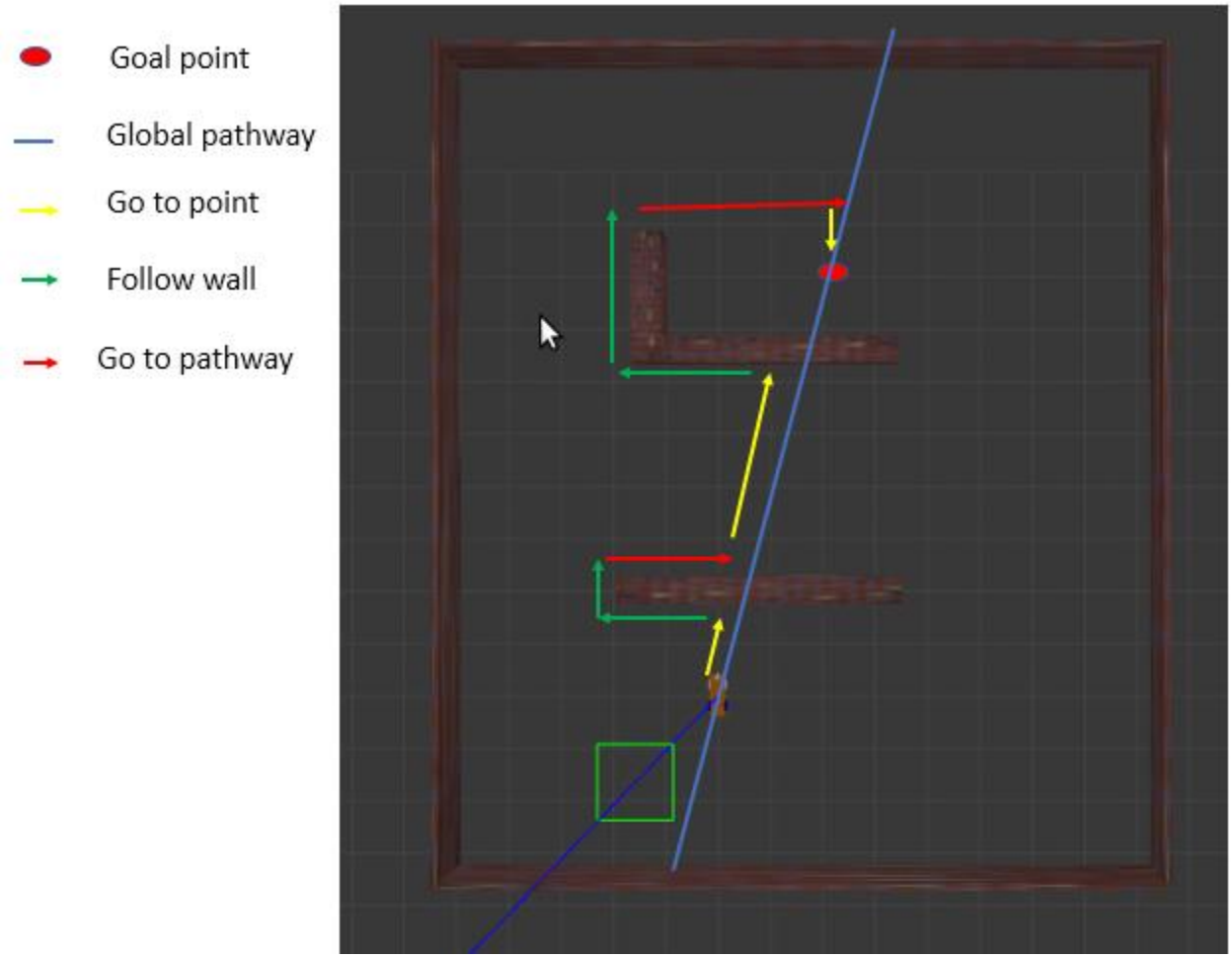


Figure 6.7 bug 2 test case

Conclusion on bugs algorithms

Bugs algorithms are not efficient to use for autonomous vehicles like rovers, as it shown before from test cases, so we need for more advanced algorithms to depend on it in automation of our rover but also bugs algorithms are not useless

Because the advanced algorithm we used as main algorithm of motion planning and navigation is A* algorithm which is dependent on cost maps which is advanced map from the original maps which is generated from SLAM part in SLAM chapters

So to run A* algorithm we need a map, for map generation in unknown world that is not discovered yet so we need way to explore the world autonomously to generate the maps need for navigation with A*

So, we used previous bugs algorithms in exploring the unknown worlds that doesn't discovered yet then give these maps to A* to have a reliable navigation algorithms

6.3 A* (A star) algorithm

what exactly is the A* algorithm? It is an advanced searching algorithm that searches for shorter paths first rather than the longer paths. A* is optimal as well as a complete algorithm.

What do I mean by Optimal and Complete? Optimal meaning that A* is sure to find the least cost from the source to the destination and Complete meaning that it is going to find all the paths that are available to us from the source to the destination.

So that makes A* the best algorithm in most cases, But A* is slow and also the space it requires is a lot as it saves all the possible paths that are available to us. This makes other faster algorithms have an upper hand over A* but it is nevertheless, one of the best algorithms out there.

A* is used to find the most optimal path from a source to a destination. It optimizes the path by calculating the least distance from one node to the other

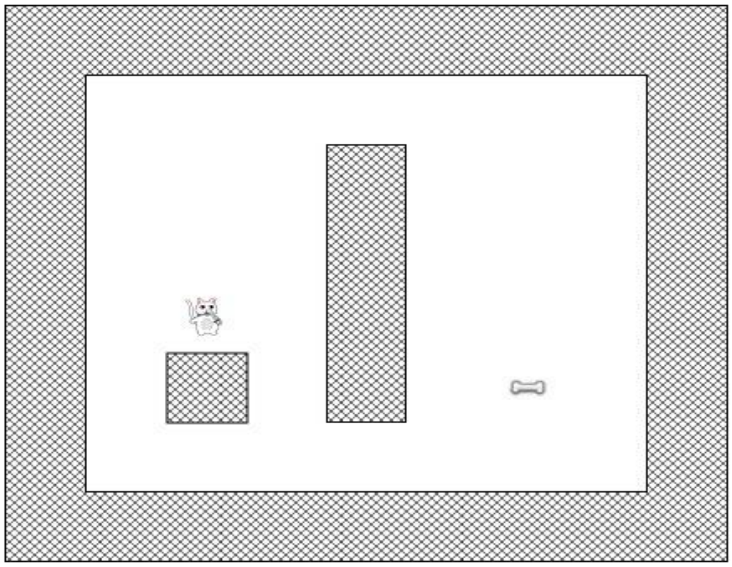
$$F = G + H$$

- F: is the parameter of A* which is the sum of the other parameters G and H and is the least cost from one node to the next node. This parameter is responsible for helping us find the most optimal path from our source to destination.

- G: is the cost of moving from one node to the other node. This parameter changes for every node as we move up to find the most optimal path.

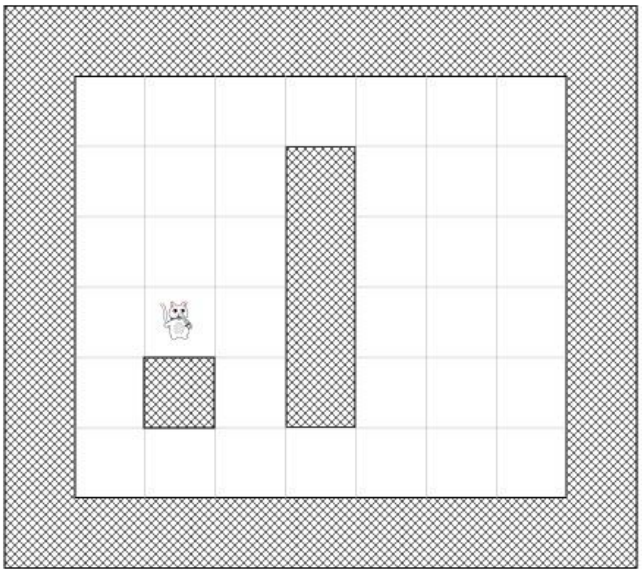
- H: is the heuristic/estimated path between the current code to the destination node. This cost is not actual but is, in reality, a guess cost that we use to find which could be the most optimal path between
  our source and destination.

Let’s imagine that we have a game where a cat wants to find a way to get a bone.



We need to simplify the searching area to save computational power so we first divide the map to grid with suitable dimensions grid these dimensions are related to the footprint of our rover or in this tutorial to our cat size

This shown in the following figure

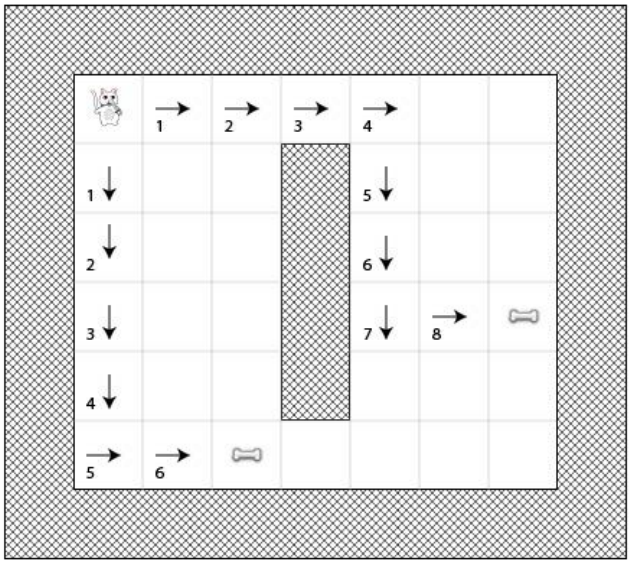


6.3.1 illustration how to calculate G:

Recall that G is the movement cost from the start point A to the current square.

In order to calculate G, we need to take the G of its parent (the square where we came from) and to add 1 to it. Therefore, the G of each square will represent the total cost of the generated path from point A until the square.

For example, this diagram shows two paths to two different bones, with the G score of each square listed on the square:



Note: G is describing length value so u could assign it as real length between the grids

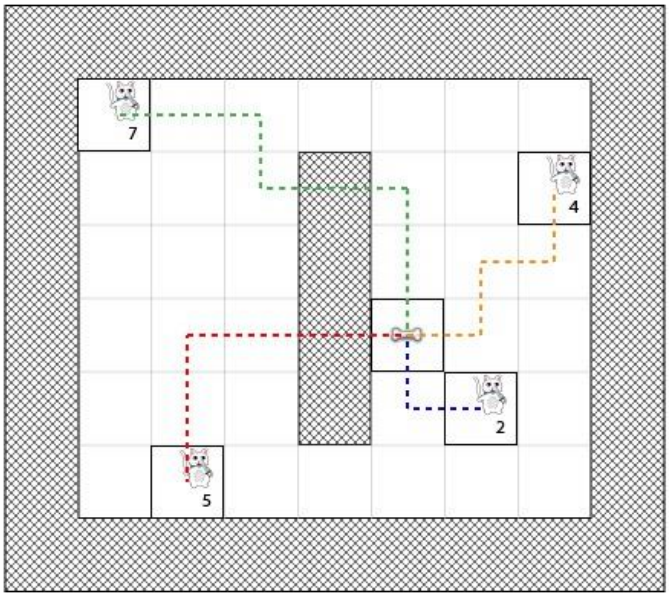
6.3.2 illustration how to calculate H:

Recall that H is the estimated movement cost from the current square to the destination point.

The closer the estimated movement cost is to the actual cost, the more accurate the final path will be. If the estimate is off, it is possible the path generated will not be the shortest (but it will probably be close). This topic is quite complex so will not be covered in this tutorial series, but I have provided a link explaining it very well at the end of the article.

To put it simply, we will use the “Manhattan distance method” (Also called “Manhattan length” or “city block distance”) that just counts the number of horizontal and vertical square remaining to reach point B without taking into account of any obstacles or differences of land.

For example, here's a diagram that shows using the "city block distance" to estimate H from various starts and destinations:



6.3.3 A* Algorithm review

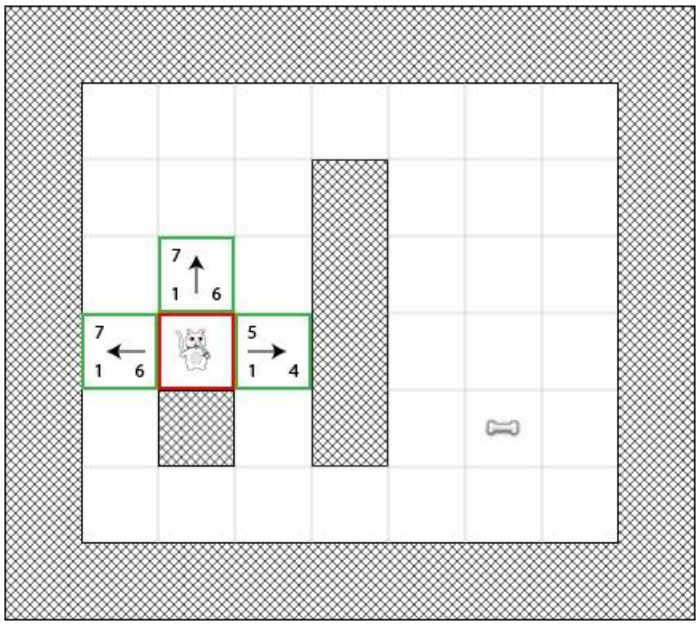
First, we need to declare two lists for saving memory of algorithm the two lists are:

- **Open list**: write down all the squares that are being considered to find the shortest path
- **Closed list**: write down the square that does not have to consider it again

The following figure shows how the algorithm works in the previous case of the cat and a bone and it's a illustration to the values shown in figures

F	.
G	H

Step 1



First step algorithm starts to calculate the F,G and H of the neighborhood nodes of the start node

For example, for the right node

G = 1 because it is far away from the start node with just one node

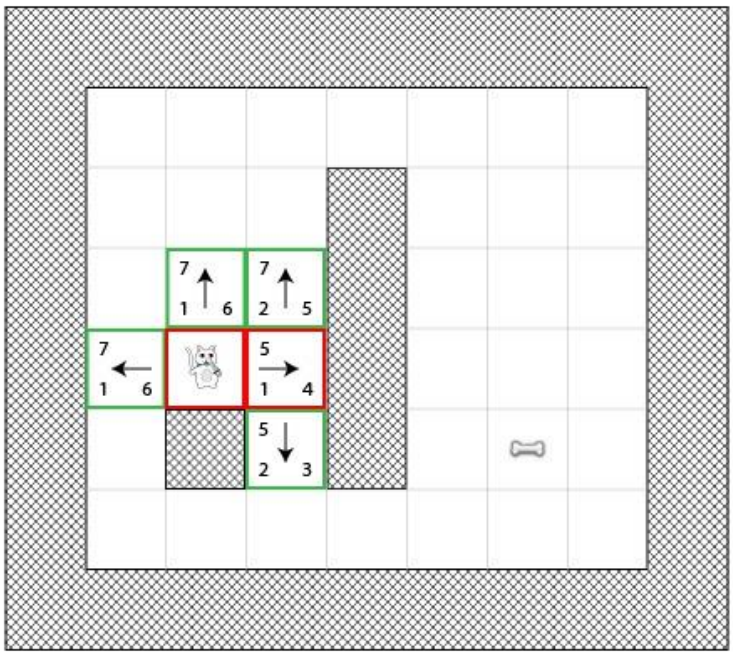
H = 4 because it is far away from the goal node with 4 nodes discarding the obstacles

F = G + H = 5

And so on,

Then the robot(cat) will navigate to the lowest node which is the right node and repeat the same steps of step 1

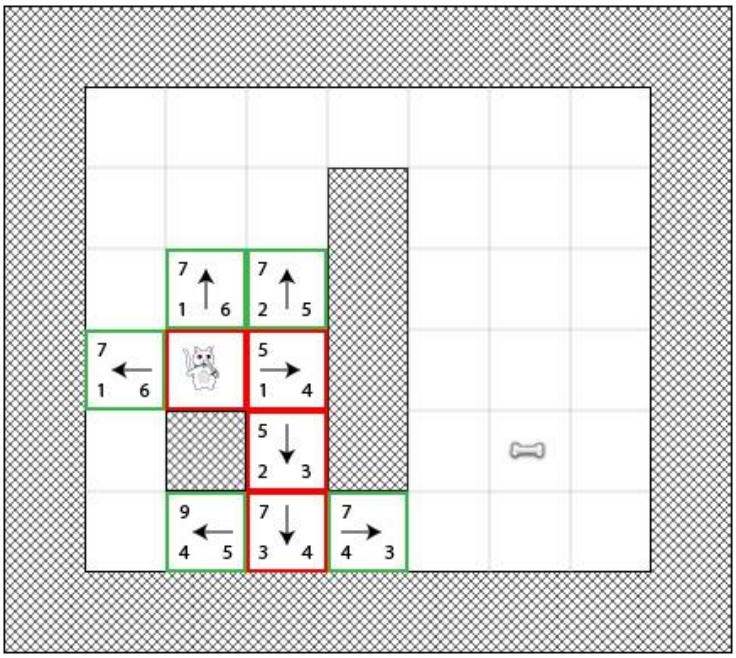
Step 2



As default the robot will navigate to the lowest value of F is to the down node

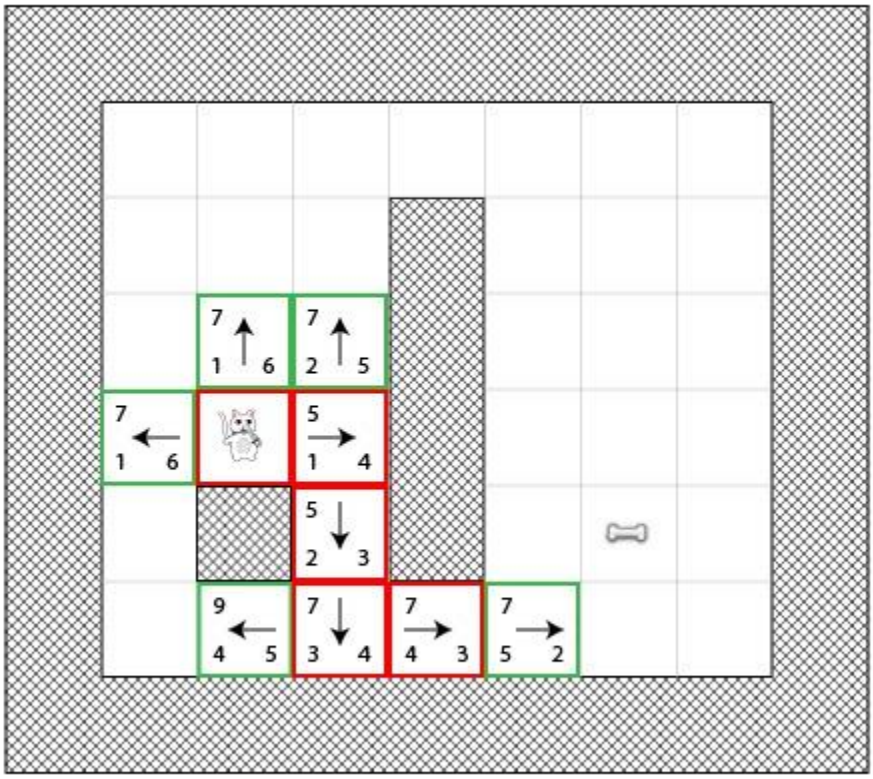
Note that G here is 2 cause it far with 2 nodes from the start node and H is 3 cause it far with 3 nodes discarding the obstacle

Step 3



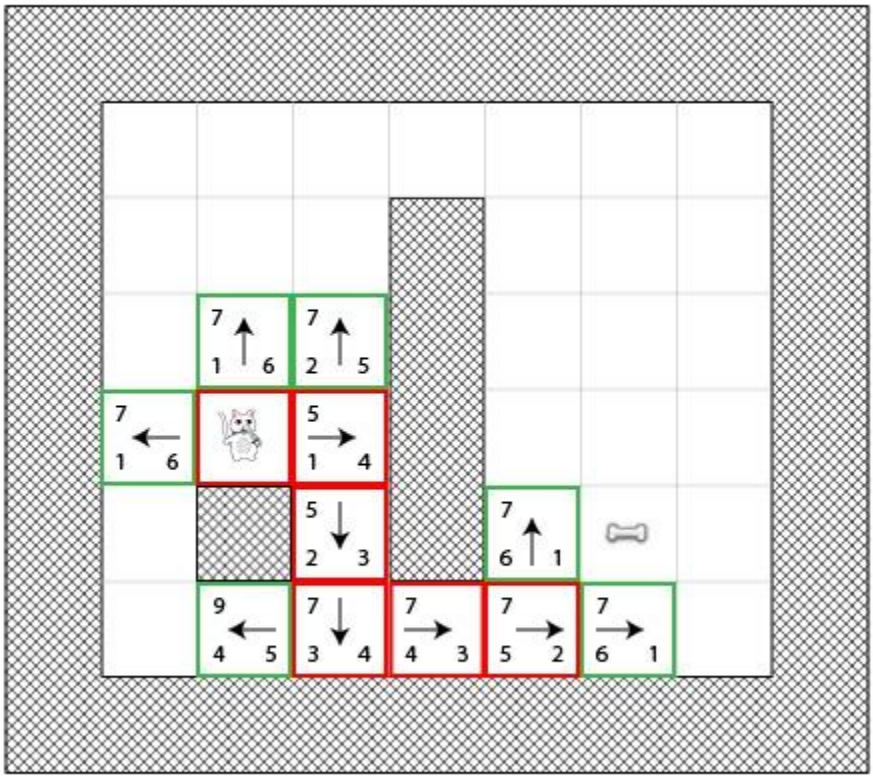
It goes with the same steps

Step 4

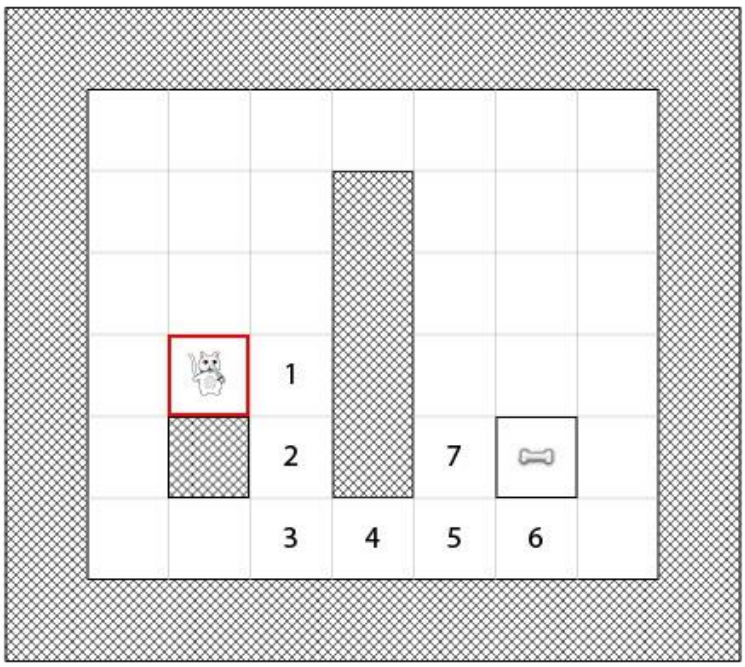


It also goes with the same steps

Step 5



Here we have the neighborhood node has the same values of cost F so we have to equal paths with the same cost so it is up to our choice with no different

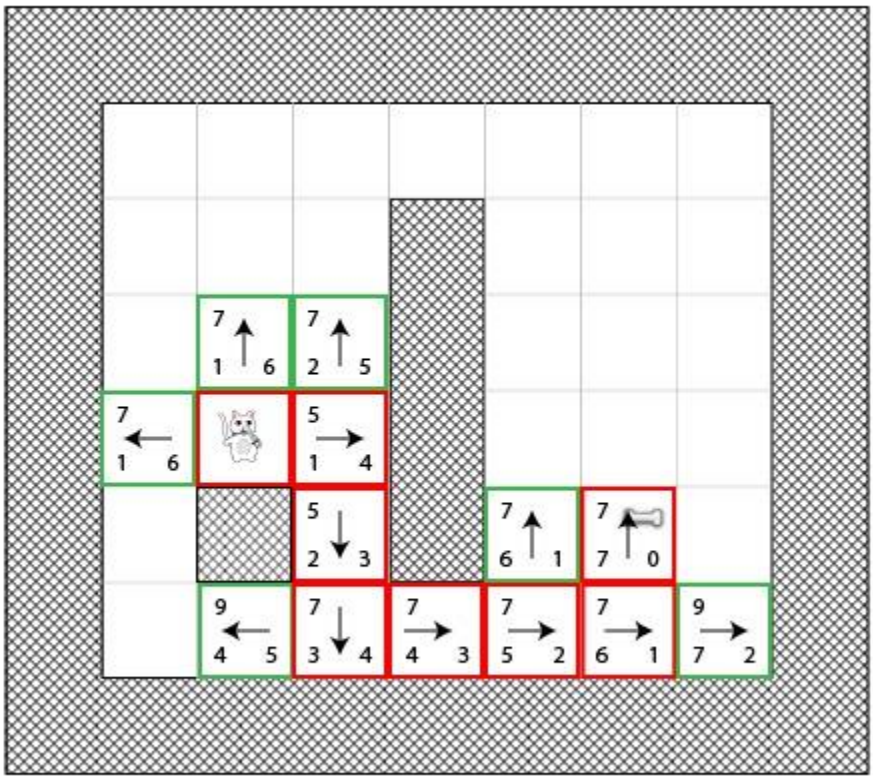


The two paths:

- 1-2-3-4-5-6-Bone
- 1-2-3-4-5-7-Bone

It doesn't really matter which of these we choose, it comes down to the actual implementation in code.

Step 6



Now the cat reaches to the bone with fast low and cost path and the mission accomplish

6.4 Rover tests

The following figure shows the world(environment) that we prepared to test rover on it

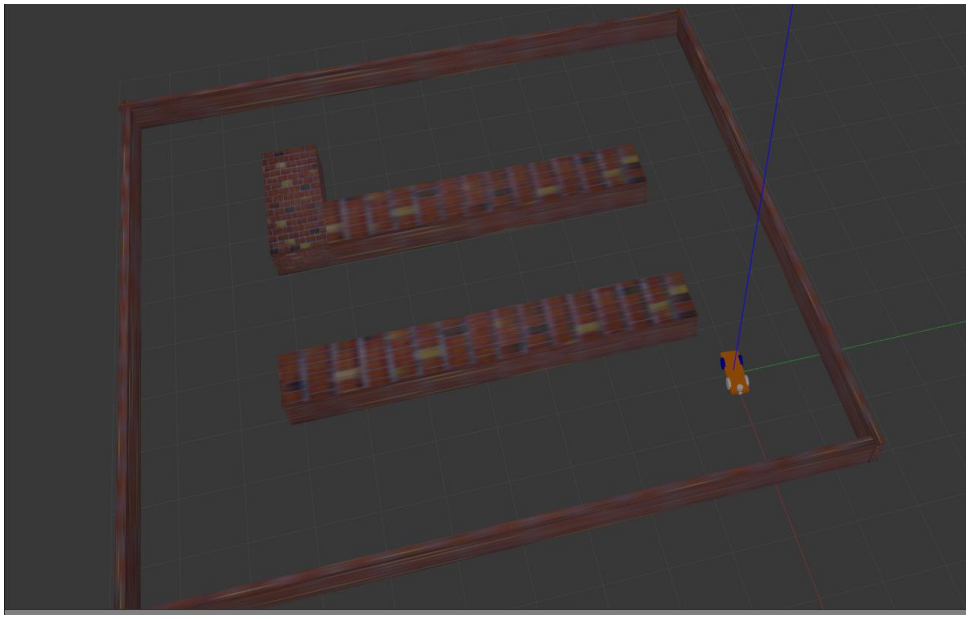


Figure 6.8 gazebo world

Then the following group of figures show the global path and its updates with the cost maps from RVIZ

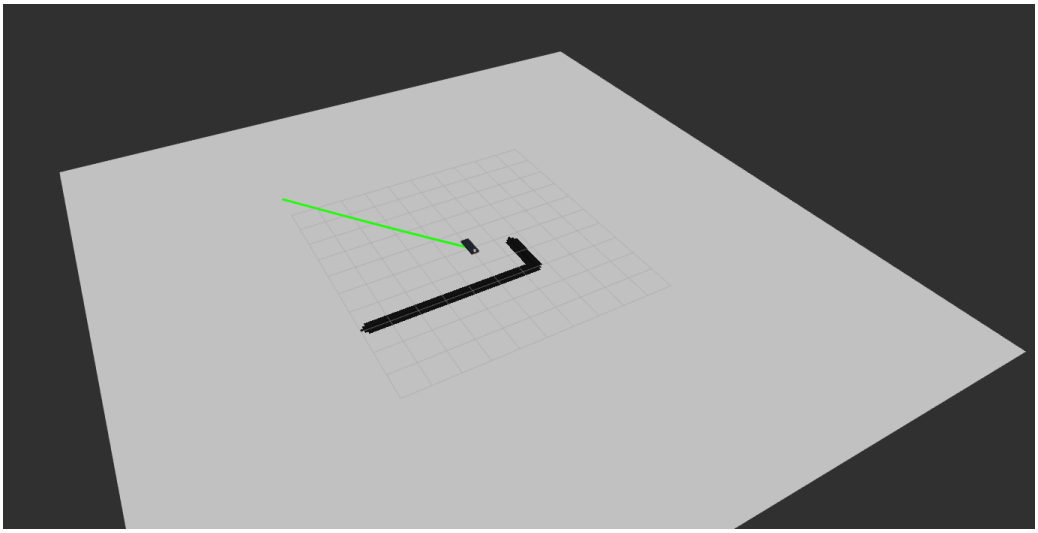


Figure 6.9 step 1

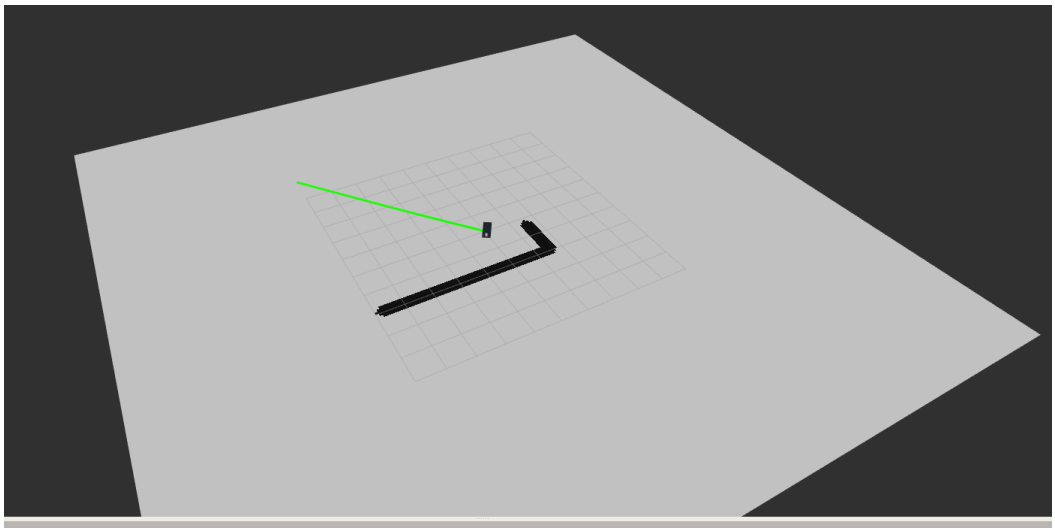


Figure 6.10 step 2

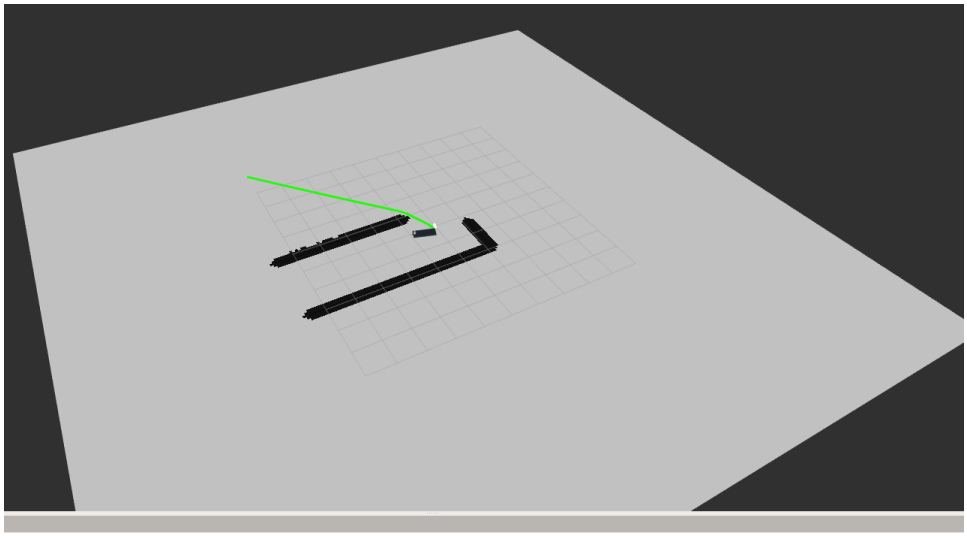


Figure 6.11step 3

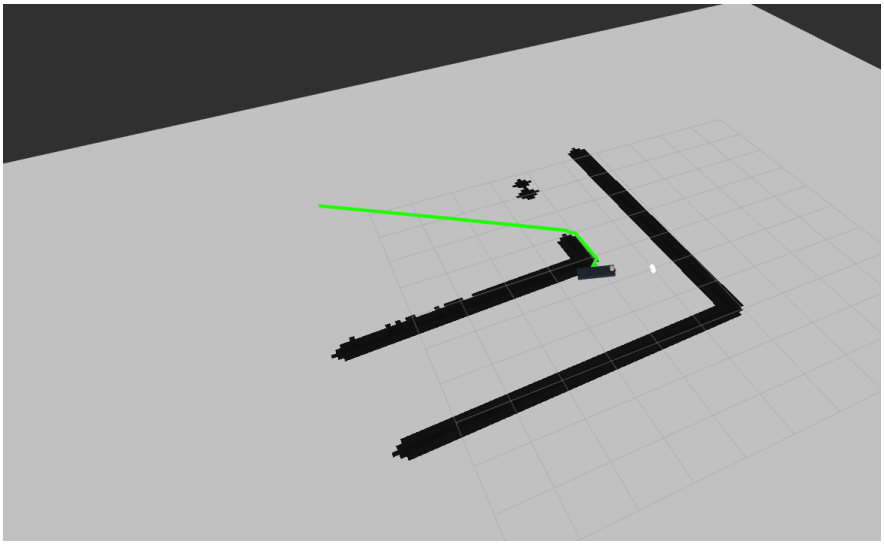


Figure 6.12step 4

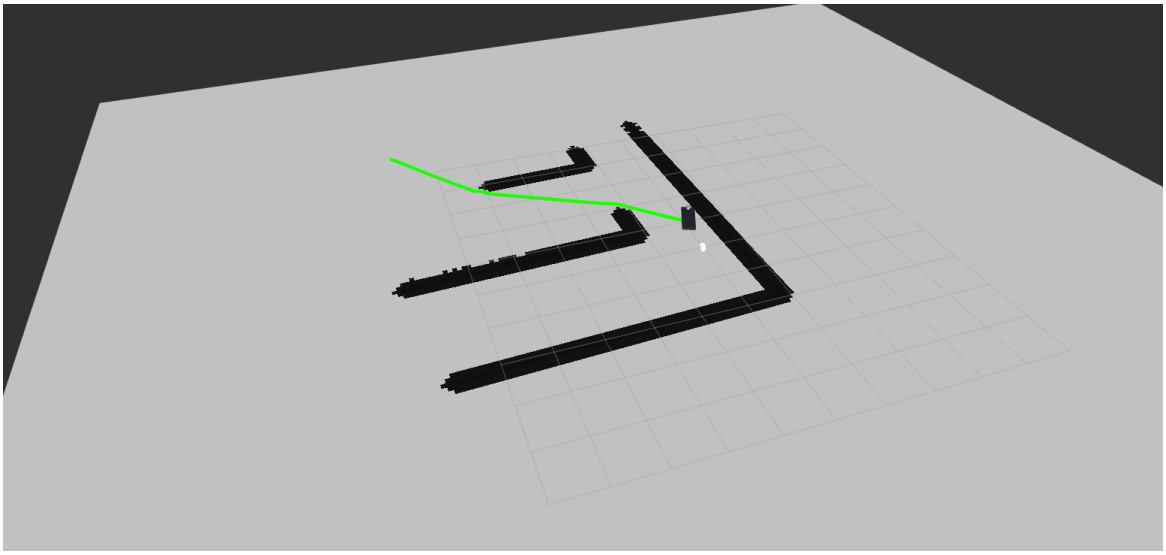


Figure 6.13 step 5

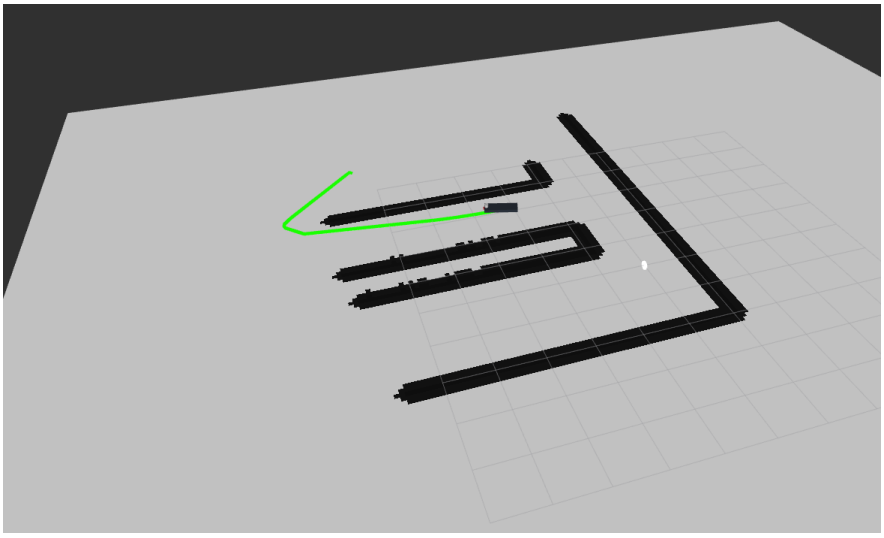


Figure 6.14 step 6

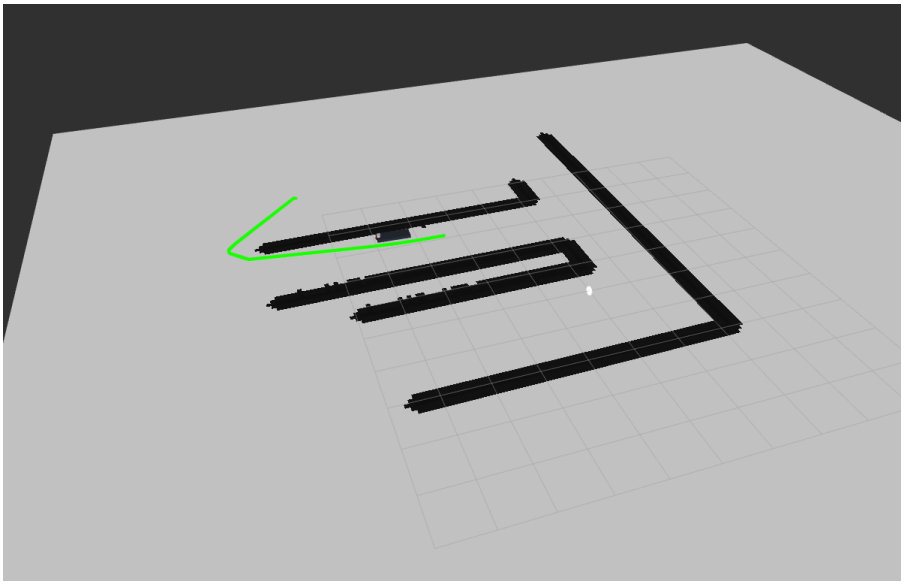


Figure 6.15 step 7

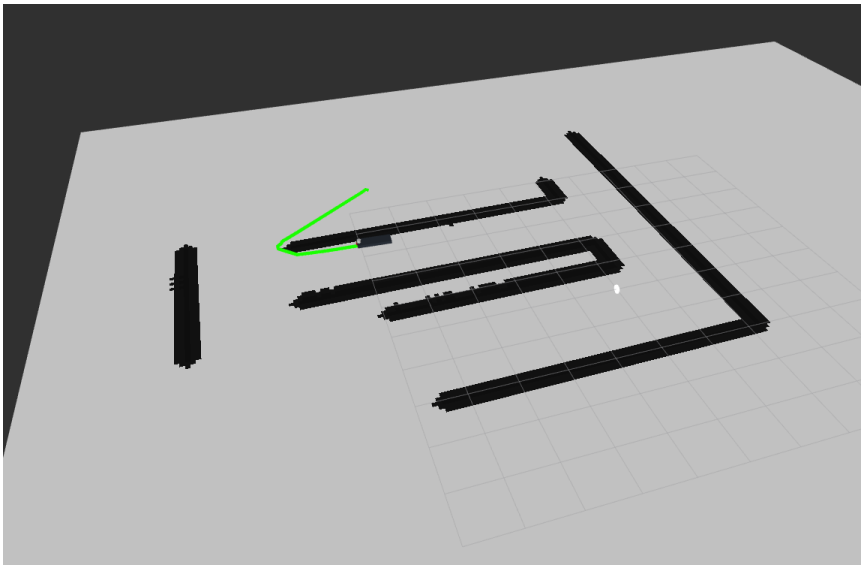


Figure 6.16 step 8

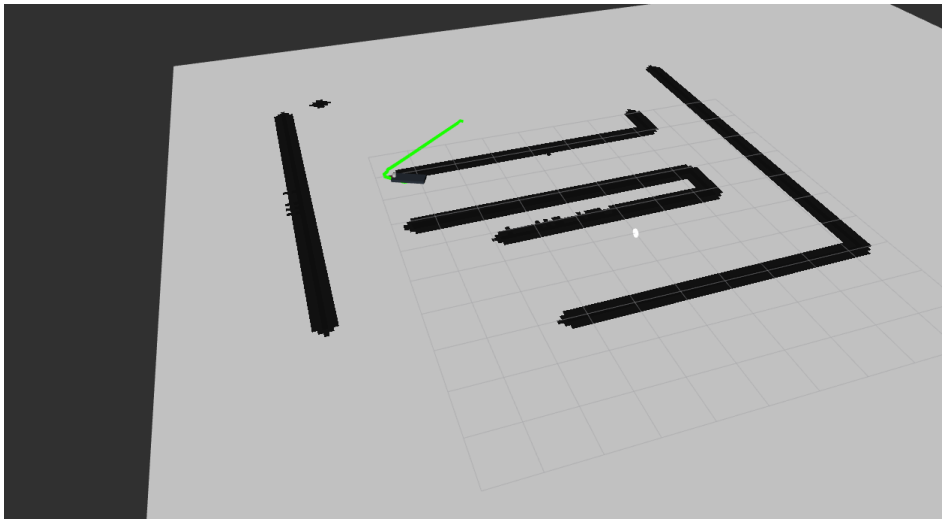


Figure 6.17 step 9

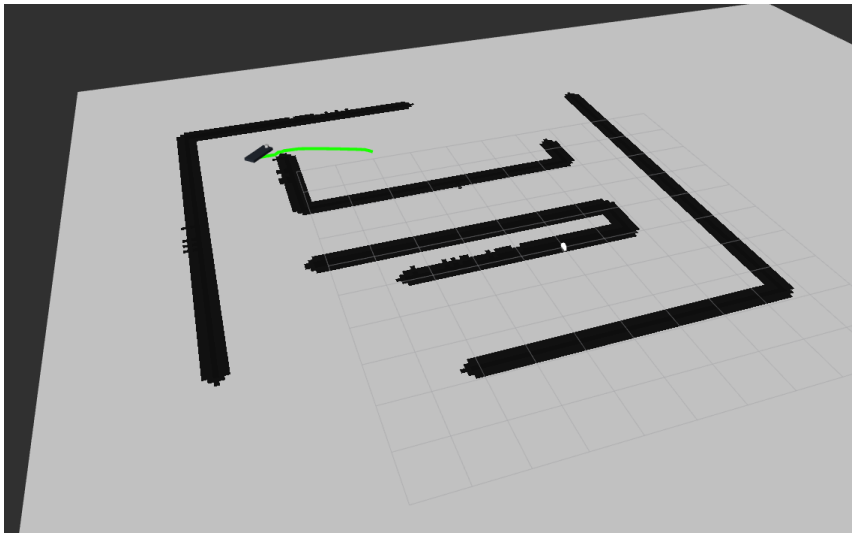


Figure 6.18 step 10

6.5 Conclusion on A* results

If we notice the subsequences of navigating to goal it starts with draw a pathway with respect to it vision with the limits of vision map which converted to cost-map then it always update the path and get a full map when exploring more of the map and that what make A* very useful algorithm

7. References

[1] Morgan Quigley, Brian Gerkey & William D. Smart, "Programming Robots with ROS A PRACTICAL INTRODUCTION TO THE ROBOT OPERATING SYSTEM", O'Reilly Media, Inc, December 2015.

[2] YoonSeok Pyo, HanCheol Cho, RyuWoon Jung, TaeHoon Lim, "ROS Robot Programming", ROBOTIS Co.,Ltd, Dec 22, 2017

[3] D.B.Reister, M.A.Unseren, "Position and Constraint force Control of a Vehicle with Two or More Steerable Drive Wheels", IEEE Transaction on Robotics and Automation, page 723- 731, Volume 9, Dec 1993.

[4] H.Hacot, "Analysis and Traction Control of a Rocker-Bogie Planetary Rover", M.S.Thesis, Massachusetts Institute of Technology, Cambridge, MA, 1998.

[5] Thomas Bräunl, "EMBEDDED ROBOTICS Mobile Robot Design and Applications with Embedded Systems third edition", springer.com, 2008.

[6] ROS Gazebo tutorials - Arabic (https://www.youtube.com/playlist?list=PL0cxiXoTD1yprQ-KsUF50xGoGqx1strJ3)

[7] ROS official website (http://wiki.ros.org/)

[8] Anis Koubaa course, "ROS for Beginners: Basics, Motion, and OpenCV ", https://www.udemy.com/course/ros-essentials/

[9] Anis Koubaa course, "ROS for Beginners II: Localization, Navigation and SLAM ", https://www.udemy.com/course/ros-navigation/

[10] Madhur Behl lecture, "Localization Using Particle Filters", https://www.youtube.com/watch?v=l_nWiL3mkzQ