People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

University M'Hamed BOUGARA – Boumerdes

Institute of Electrical and Electronic Engineering

Department of Power and Control

Final Year Project Report Presented in Partial Fulfilment of
the Requirements for the Degree of

MASTER

In **Electrical and Electronic Engineering**

Option: **Control**

Title:

Kinect-based collision-free path planning for mobile manipulators

Presented by:

- **BOULAHIA ALA EDDINE**

Supervisor:

BELAIDI HADJIRA

HENTOUT ABDELFETAH

Registration Number: ……../…….

Kinect-based collision-free path planning for mobile manipulators

Abstract:

The objective of this project is to generate, in off-line, a path (trajectory) without collision for a mobile base (mobile base with an arm) in an indoor environment which is complex dynamic and cluttered with obstacles the case of reaching tasks.
The generated free path must joint an initial situation of the robot base, $(XB, YB, ZB)_{init}$, to a predefined final position, $(XB, YB, ZB)_{End}$.
In order to move the form an initial situation to the final goal while avoiding any obstacles (using a technique of soft-computing), the robot will operate a kinect camera.
The implementation and validation will be performed on the mobile manipulator RobuTer/ULM available at the DPR CDTA.

First and foremost, I am thankful to God almighty, for showing heavenly blessing upon us, as without that nothing would have been possible.
I would like to express my felt gratitude to my supervisors BELAIDI HADJIRA from INELEC and Mr. HENTOUT ABDELFETAH from CDTA for their support and guidance during the construction of this work.
A big thanks to CDTA for opening their doors to me and offering their help to make this work happen.
Also, A Special thanks to my family and friends for their support and all INELEC staffs and teachers.

Table of Contents

List of figures

List of Tables

List of acronym

AMCL: Adaptive Monti Carlo Localization.

A/D conversion: Analog to Digital conversion.

Acc_lim: Acceleration limit.

CDTA: Center of Development and technology of Algeria.

CMOS camera: Complementary Metal-Oxide Semiconductor camera.

CPR: Cycles Per Revolution.

CAD software: Computer Aided Design.

Dof: Degree of freedom.

DC: Direct Current.

dB: The Decibel.

2D ,3D: 2 Diamantions, 3 Diamantions.

EKF: Extended Kalman Filter.

IR: Infrared.

ICC: Instantaneous Center of Curvature.

Ir_of_robot: Internal radius of the robot.

GUI: Graphical User Interface.

GML camera: Graphic and Media Lab camera.

Max_vel: Maximum velocity.

MDH: Modified Denavit-Hartenberg.

MRPT: Mobile Robot Programming Toolkit.

Nav_msgs: Navigation messages.

OpenCV API: Open Source Computer Vision Application Programming Interface.

OpenNI: Open Natural Interaction.

ROS: Robot Operating System.

RVIZ: Robot visualisation.

RGB: Red, Green, Blue.

Robot_tf_publisher: Robot transformation publisher.

SLAM: Simultaneous Localization and Mapping.

STDR: Simple Two-Dimensional Robot Simulator.

STAIR : STanford Artificiel Intelligence Robot.

.SDF file: Standard Database Format.

Sensor_msgs: Sensor messages.

TF: Transformation frame.

TCP/IP: Transmission Control Protocol/ Internet Protocol.

Tf_broadcaster: Transformation broadcaster.

Tf_listener: Transformation listener.

USB: Universal Serial Bus.

URDF: Unified Robot Description Format.

ULM: Ultra-Light Arm.

. Xml file: Extensible Markup Language.

YARP: Yet Another Robot Platform.

Through their ability to perform tasks in unstructured environments, robots have made their way into applications like: transportation, geology negotiates terrain, military, agriculture, mining, carries payload, customer support and farming. All mobile robots use locomotion that generates traction. The well-designed robotic locomotion stabilizes the robot's frame, smooth's the motion of sensors and insure the configuration of the working tools.

In our days' mobile robot navigation process became an essential feature that guides autonomous robots. During this process, the robot must have knowledge about its objectives starting from processing the environment to detect its location and destination then use the necessary strategies to reach its specified task. Hence, different steps needed from the robot to step through including: perception to collect information from sensors, localization to estimate its initial position, path planning methods to create the optimal path to reach its final position then execute the process.

Several recent robotic applications are performed by a manipulator mounted on a mobile base. These kinds of robots are called "mobile manipulators". They are systems composed of robotic arm mounted on a mobile base (ex; our case of studies, RobuTER/ULM). This combination gives rise for a new class of robots with flexible properties. However, the many Degrees of freedom (Dof) of this type of robots (for our case RobuTER/ULM, 03 Dof for the mobile base and 06 Dof for the manipulator) present new challenges and the combination between the mobile base and the manipulator becomes very difficult to monitor. In addition, to the other problems like the modeling and perception of their environment using multi-sensors fusion, generation of operation plans from the assigned tasks, path planning, etc...

The project objectives focus on mobile manipulator navigation problems in general; particularly on perception, localization, and cognition. These points are achieved by designing an autonomous mobile robot using visual navigation to analyze the environment map with the Image processing tools, multi-sensors fusion to detect and track the robot position regularly, soft computing techniques to generate the path from the initial position to the final one, then build a closed loop control to keep the robot on the designed trajectory by adjusting the data sets that we will send as commends to the robot manipulator.

Hence, this work is organized as follow:

General introduction

- Chapter 01: Localization and locomotion;
- Chapter 02: Mobile robot manipulator Kinematics;
- Chapter 03: Perception (different sensing devices);
- Chapter 04: Simulation and results;

Conclusion and perspectives.

I. Introduction

A mobile robot needs locomotion mechanisms that enable it to move unbounded throughout its environment. But there is large variety of possible ways to move, and so the selection of a robot's locomotion approach is an important aspect of mobile robot design. Hence this chapter deals with an overall description of the robot locomotion and navigation, and all the terms that have a relationship to both of them; then an overview of the robot control process is given. The chapter ends up with the description of the dynamic and kinematic characteristics of the RobuTER/ULM.

I.1. Locomotion/Manipulation description

Locomotion is the complement of manipulation. In manipulation, the robot arm is fixed but moves objects in the workspace by imparting force to them. In locomotion, the environment is fixed and the robot moves by imparting force to the environment. In both cases, the scientific basis is the study of actuators that generate interaction forces, and mechanisms that implement desired kinematic and dynamic properties. Locomotion and manipulation thus share the same core issues of stability, contact characteristics, and environmental type as summarized in Table 1.1.

Table 1.1. Locomotion and manipulation some core issues.

Stability	Characteristics of contact	Type of environment
• number and geometry of contact points • center of gravity • static/dynamic stability • inclination of terrain	• contact point/path size and shape • angle of contact • friction	• structure • medium, (e.g. water, air, soft or hard ground)

I.2. Mobile robot navigation and Autonomy

Figure 1.1 shows the robot interaction diagram. In general, the robot in its environment and according to its mission needs to answer three main questions:

- Where am I now?
- Where am I going to?
- How do I get there?

To deal with these questions the robot must:

- ✓ Perceive the environment to build a map;

- ✓ Analyze the environment and its elements;
- ✓ Find its initial position;
- ✓ Find its final position and build an optimal path to reach it;
- ✓ Start the execution of the tasks by starting its movement.

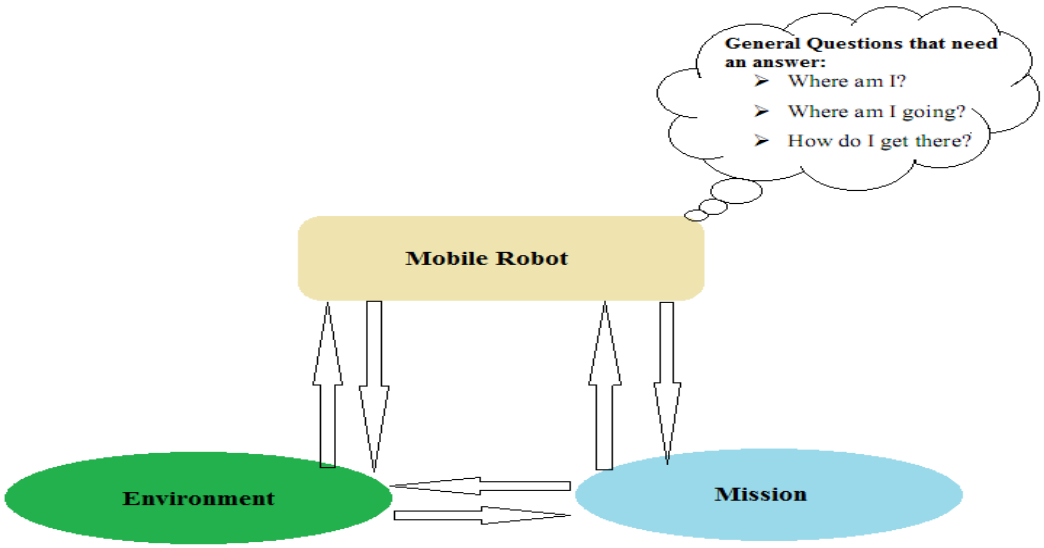


Figure 1.1. Robot interaction diagramme.

I.2.1. Navigation

Navigation is one of the most challenging competences required of a mobile robot. Success in navigation requires success at the basic four building blocks of navigation, which are:

Perception: the robot must interpret its sensors to extract meaningful data;

Localization: the robot must determine its position in the environment;

Cognition: the robot must decide how to act to achieve its goals;

Motion control: the robot must modulate its motor outputs to achieve the desired trajectory.

Navigation is a central capability of mobile robots and substantial progress has been made in the area of autonomous navigation in the past [1].

I.2.2. Autonomy

Autonomy is the quality of being self-controlled. One measure of autonomy is the amount of human control that is required for the robot's operation.

An autonomous robot is capable of detecting objects by means of sensors or cameras and processing this information into movement without a remote control.

I.2.3. Autonomous navigation

There exist two types of autonomous navigation:

I.2.3.1. Indoor navigation

In order to associate behaviors with a place (localization), it is required for the robot to know where it is and be able to navigate point-to-point. Such navigation began with wire-guidance and progressed to beacon-based triangulation. Current commercial robots autonomously navigate based on sensing natural features. At first, autonomous navigation was based on planar sensors, such as laser range-finders, that can only sense at one level. The most advanced systems now fuse information from various sensors for both localization (position) and navigation. Systems such as Motivate can rely on different sensors in different areas, depending upon which provides the most reliable data at the time, and can re-map a building autonomously [3].

I.2.3.2. Outdoor navigation

Outdoor autonomy is most easily achieved in the air, since obstacles are rare. Pilotless drone aircraft are increasingly used for reconnaissance. Some of these unmanned aerial vehicles are capable of flying their entire mission without any human interaction at all. Outdoor autonomy is the most difficult for ground vehicles, due to:

- Three-dimensional terrain.
- Great disparities in surface density.
- The weather changes.
- Instability of the sensed environment.

In details, autonomous navigation includes different interrelated activities such as:

(i) Perception, as obtaining and interpreting sensory information. Autonomous robots must have a range of environmental sensors to perform their task and stay out of trouble. Common exteroceptive sensors (Exteroception is sensing things about the environment) include **Contact Sensors, Range Sensor and Vision Sensors.**

(ii) Exploration, as the strategy that guides the robot to select the next direction to go. Using realistic sensors to carry out a systematic exploration of its environment. The robot is modeled as a single point moving in a two-dimensional configuration space populated with visually opaque and transparent obstacles. The robot is equipped with proximity sensors, a vision-based recognition system, all of which have some uncertainty associated with their measurements.

(iii) Mapping, involving the construction of a spatial representation of the environment by using the perceived sensory information. Mapping is that branch of one, which deals

with the study and application of ability to construct map or floor plan by the autonomous robot and to localize itself in it. The problem of learning maps with mobile robots has received considerable attention over the past years. Most of the approaches assume that the environment is static during the data-acquisition phase [4].

(iv) Localization, as the strategy to denotes the ability of the robot to establish its own position and orientation within the frame of reference.

(v) Path planning, as the strategy to find a path towards a goal location being optimal or not. Path planning is effectively an extension of localization, so that it requires the determination of the robot's current position and a position of a goal location, both within the same frame of reference or coordinates.

(vi) Path execution, where motor actions are determined and adapted to environmental changes and by sending commends to the actuators to turn them with a desired acceleration and angle to move forward the trajectory or path that have been built.

To operate in crowded, dynamic environments, autonomous robots must be able to effectively utilize and coordinate their limited physical and computational resources. As complexity increases, it becomes necessary to impose explicit constraints on the control of planning, perception, and action to ensure that unwanted interactions between behaviors do not occur.

The classical robotic control system is based on three sequence blocks: Sensing -> Thinking -> Acting as it is shown in figure 1.2.

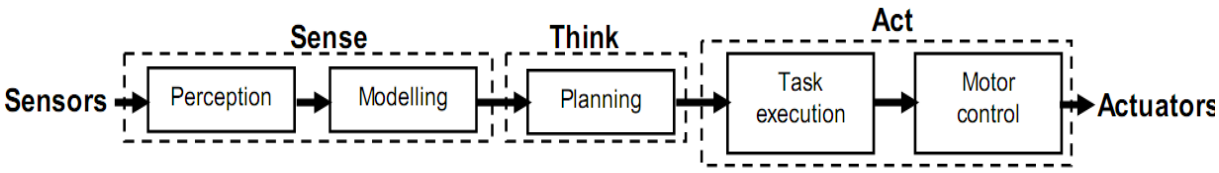


Figure 1.2. Classical mobile robot control system composition

The cycle for the mobile robot control shame and the interaction between these activities are summarized by figure 1.3.

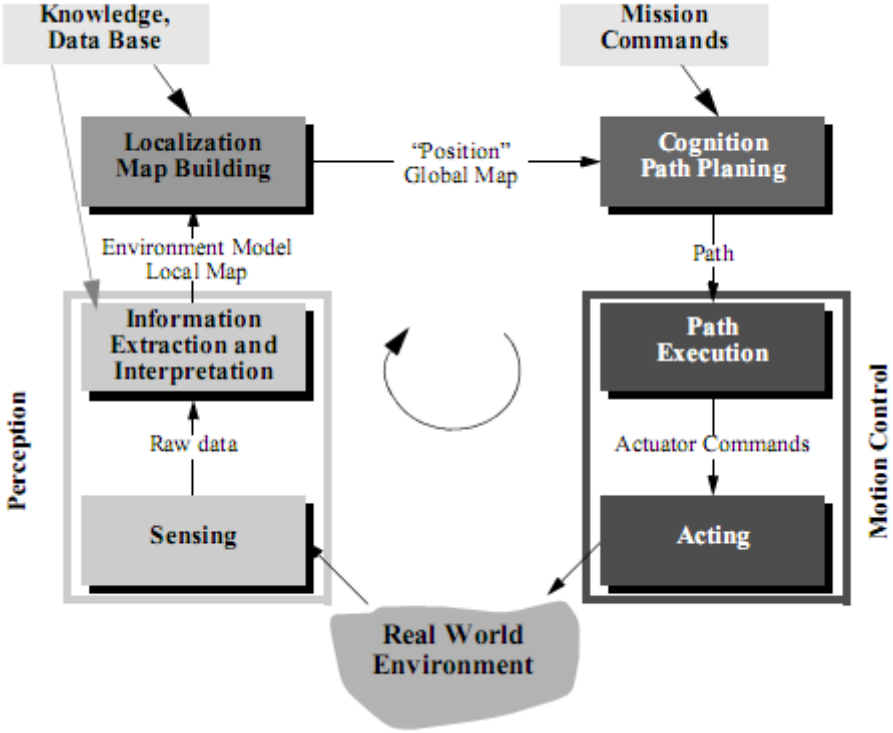


Figure 1.3. Reference control shame for mobile robot systems [1].

I.3. Mobile manipulator an overview

Nowadays, mobile manipulator is a widespread term to refer to robot systems built from a robotic manipulator arm mounted on a mobile base. Such systems combine the advantages of mobile base and robotic manipulator arms and reduce their drawbacks. This system offers a dual advantage of mobility offered by a mobile platform and dexterity offered by the manipulator. However, the operation of such system is challenging because of the many degrees of freedom and the unstructured environment that it performs in. The General compositions of such system are:

- Mobile Platform or base.
- Robot manipulator or arm.
- Vision (cameras, sensors to sense the internal changes in the robot, sensors to sense the external changes in the environment …).
- Tooling (End-of-the-arm design and configuration).

I.3.1. A case of study: RobuTER/ULM

RobuTER/ULM is an autonomous mobile robot manipulator that is available in the Center of Development of Advanced Technologies (CDTA) of Algiers. The locomotion of

this robot is performed via the control of two independent DC motors coupled to each drive wheel, and two additional caster wheels.

The RobuTER/ULM is shown in figure 1.4, and its geometric properties are given in the following table (Table 1.2).

Table 1.2. Geometric properties for the RobuTER/ULM.

Property	Length	Width	Height	Weight	Payload	Max speed
Value	102.5 cm	68.0 cm	44.0 cm	150 kg	120 kg	1.0 m/s

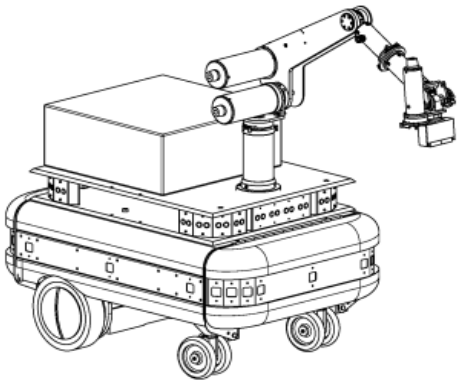


Figure 1.4. RobuTED/ULM available at CDTA [2].

I.3.2. The architecture of the experimental robotic system

The experimental robotic system, shown in figure 1.5, consists of a local (Operator) site and a remote site, connected by wireless communication systems:

- Local site: it includes an off-board PC running under Windows XP, a wireless TCP/IP communication media, a wireless video reception system and input devices.
- Remote site: it includes the RobuTER/ULM mobile manipulator, a wireless TCP/IP Communication media and a wireless video transmission system.

I.3.3. Description of the mobile base of RobuTER

The robot base consists of a platform with two wheels and a load capacity of 15 kg (see figure 1.5). The wheels are 250 mm in diameter, and have a torque of 22 Nm nominal per wheel. They are driven by DC electric motors and enable it to reach a nominal speed of 2.6 m/s. The direction of RobuTER is given by the differential speed of the two wheels. The two wheels are placed at the front of the platform to provide stability.

The nominal robot consumption is *30A* and between *30* to *48VDC*, the peak current is *60A* to *48VDC* for *2s*.

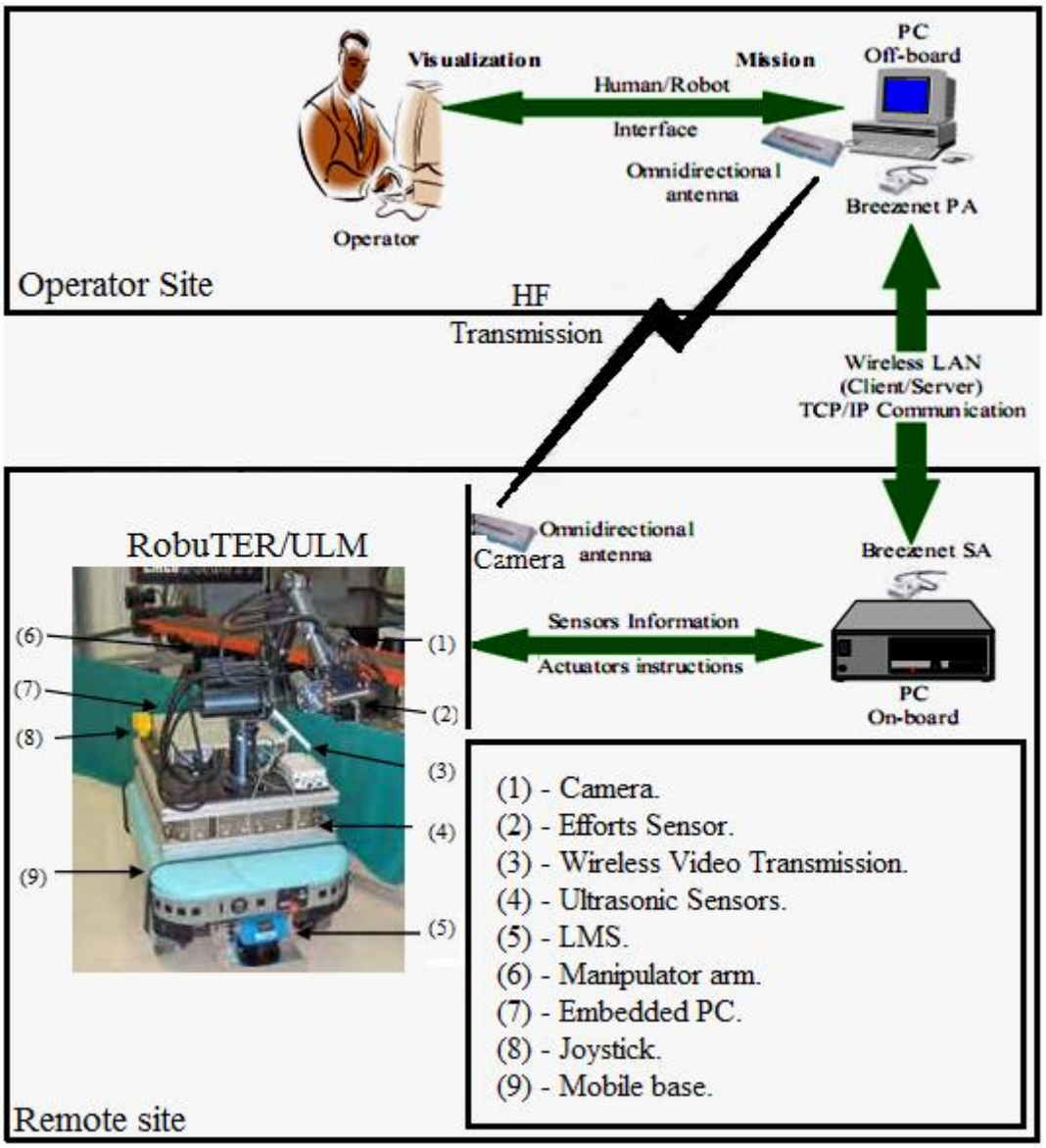


Figure 1.5. The architecture of the experimental robotic system [2].

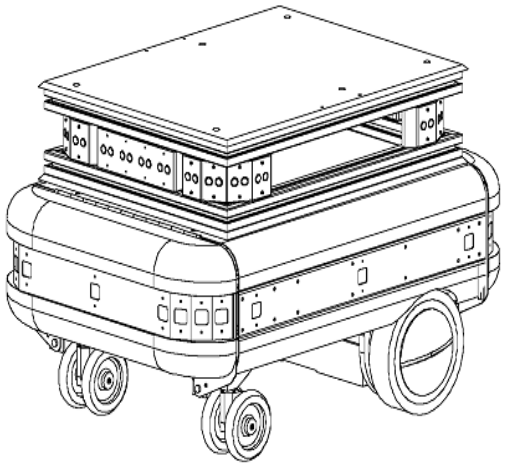


Figure 1.6. RobuTER mobile Base [2].

I.3.4. The Ultra Light Arms (ULM) of RobuTER

The ULM has six axes as they appear in figure 1.6 (provides 6 Dof), when the arm is fully extended the arm reaches 700 mm and has a repeatability of +/- 1 mm and a load capacity of 2kg.

Terms of use: Temperature: 0 ° C to 45 ° C

Humidity: 20-80% non-condensing

Nominal consumption of the arm: 36A when 30 to 48VDC.

Current crest of the arm: 60A (for 2s to 48VDC).

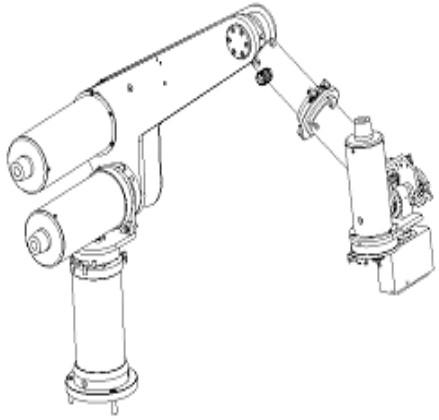


Figure 1.6. Ultra Light Arm (ULM) of RobuTER [2].

I.3.5. The Kinematic model of RobuTER/ULM

Kinematics is the description of motion without regard to the forces that cause it. It is a collection of studies of position, velocity, acceleration, and higher derivatives of the position variables. The mobile robot kinematics is deeply studied in chapter 2. However, a general idea about the RobuTER/ULM kinematic model can be assembled as shown in figure 1.7.

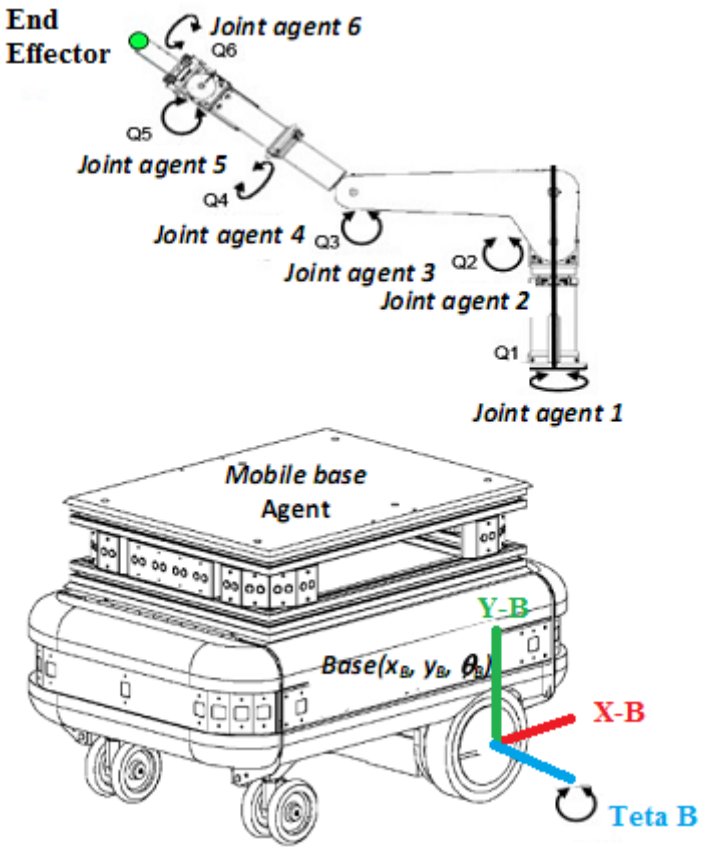


Figure 1.7: The general kinematic model of the RobuTER/ULM

1.4. Conclusion

This chapter provides a brief introduction to locomotion and navigation in general and the RobuTER/ULM as a specific case of studies. All the keywords used throughout this chapter, including the different block paradigms, necessary terms like autonomous navigation and different steps that the robot take in the navigation process starting from perception and map building followed by finding its localization, are discussed within this chapter. Then an overview in mobile manipulators architecture and finally an introduction to the kinematics analysis for RobuTER/ULM are given also.

II. Introduction

Kinematics is the most basic study of how mechanical systems behave. In mobile robotics, it is needed to understand the mechanical behavior of the robot both in order to design appropriate mobile robots for tasks and to understand how to create control software for an instance of mobile robot hardware. Of course, mobile robots are not the first complex mechanical systems which require such analysis. Robot manipulators have been the subject of intensive study for more than thirty years [1].

Hence, this chapter is organized as follow: In the first section, notation that allows the expression of robot motion in a global reference frame and in the robot's local reference frame is introduced. Then, using this notation, the construction of simple forward kinematic model of motion is demonstrated by describing how the entire robot moves as a function of its geometry and individual wheel behavior.

Next, the kinematic constraints of individual wheels are formally described, and then these kinematic constraints are combined to express the whole robot's kinematic constraints. With these tools, one can evaluate the paths and trajectories that define the robot's maneuverability.

In the second section, the arm manipulator kinematics, the forward kinematics and inverse kinematics is studied. Forward kinematics problem is straightforward and there is no complexity in deriving the equations. Hence, there is always a forward kinematics solution of a manipulator. Inverse kinematics is a much more difficult problem than forward kinematics. The solution of the inverse kinematics problem is computationally expansive and generally takes a very long time in the real-time control of manipulators.

Hence, the forward and inverse kinematics transformations for an open kinematics chain are described based on the homogenous transformation. Then, geometric and algebraic approaches are given. Afterward, problems in the inverse kinematics are discussed and explained. Finally, the forward and inverse kinematics transformations are derived based on the quaternion modeling convention.

II.1. Main reference frames

The kinematic analysis of the robot needs to focus on the following main reference frames and transformation matrices:

- $R_A = (O_A, \vec{x}_A, \vec{y}_A, \vec{Z}_A)$: Absolute reference frame.
- $R_B = (O_B, \vec{x}_B, \vec{y}_B, \vec{Z}_B)$: Mobile base reference frame.
- $R_M = (O_M, \vec{x}_M, \vec{y}_M, \vec{Z}_M)$: Manipulator reference frame.

- $R_E = (O_E, \vec{x}_E, \vec{y}_E, \vec{Z}_E)$: End-effector reference frame.
- ${}^{M}_{E}T$: Transformation matrix defining R_E in R_M. It corresponds to the Kinematic Model of the manipulator.
- ${}^{A}_{B}T$: This matrix defines R_B in R_A .
- ${}^{B}_{M}T$: This matrix defines R_M in R_B.
- ${}^{A}_{E}T$: This matrix defining R_E in R_A.

II.2. Kinematic analysis of the mobile base

A relationship between the global reference frame of the plane and the local reference frame of the robot was established in order to specify the position of the robot on the plane.

The axes X_i and Y_i in figure 2.1(a) define an arbitrary inertial basis on the plane as the global reference frame from some origin O: { X_I , Y_I }. To specify the position of the robot, choose a point ***P*** on the robot chassis as its position reference point. The basis { X_R , Y_R } defines two axes relative to ***P*** on the robot chassis and is thus the robot's local reference frame. The position of ***P*** in the global reference frame is specified by coordinates x and y, and the angular difference between the global and local reference frames is given by θ. The position of the robot can be described as a vector with these three elements. Note the use of the subscript I to clarify the basis of this position as the global reference frame (eq. 2.1):

$$\xi_I = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \quad \text{..... (2.1)}$$

The velocity in the global reference frame is: $\dot{\xi}_I = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix}$ ..... (2.2)

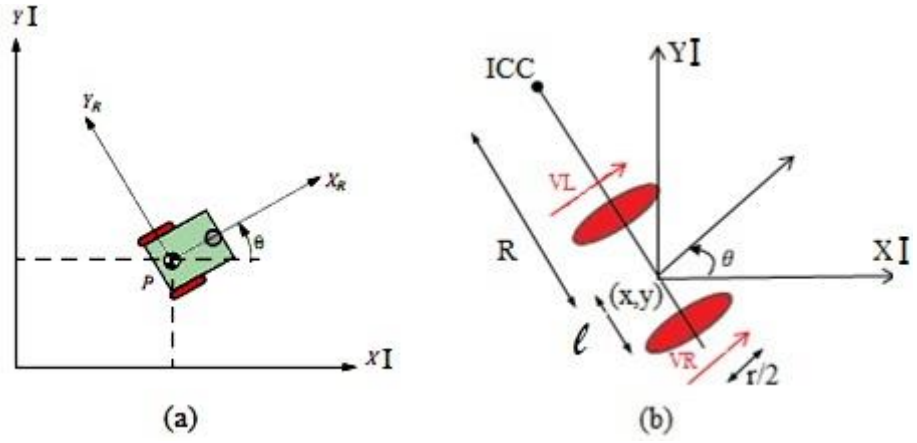


Figure 2.1. (a) The Global reference frame. (b) The robot Wheels Kinematics

To describe the robot motion in terms of component motions, it will be necessary to map motion along the axes of the global reference frame to motion along the axes of the robot's local reference frame. Of course, the mapping is a function of the current position of the robot. The mapping operation is denoted by eq. 2.3:

$$\dot{\xi}_R = R(\theta)\,\dot{\xi}_I \xrightarrow{Inverse} \dot{\xi}_I = R(\theta)^{-1}\,\dot{\xi}_R \quad \text{..... (2.3)}$$

This mapping is accomplished using ***the orthogonal rotation matrix***:

$$R(\theta) = \begin{bmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \xrightarrow{Inverse} R(\theta)^{-1} = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{... (2.4)}$$

II.2.1. Forward kinematic models

The differential drive robot has two wheels (see Figure 2.1 (b)), each with diameter r. P is centered between the two drive wheels; each wheel is a distance l from P. Given r, l, p, θ and the speed of each wheel, $\dot{\varphi}r$ and $\dot{\varphi}l$ a forward kinematic model would predict the robot's overall speed in the global frame given by eq. 2.5.

$$\dot{\xi}_I = f(l, r, \theta, \dot{\varphi}r, \dot{\varphi}l) \quad \text{..... (2.5)}$$

And the Instantaneous Center of Curvature is given by eq. 2.6:

$$\text{ICC} = [x - R\sin\theta,\ y + R\cos\theta] \quad \text{..... (2.6)}$$

Also we have the velocities equations on the two wheels are:

$$Vr = \omega(R + l)\ ;\ Vl = \omega(R - l) \xrightarrow{Hence\ we\ get:} R = l\frac{Vr + Vl}{Vr - Vl}\ ;\ \omega = \frac{Vr - Vl}{2l}\ ;$$

$$V = R\omega = \frac{Vr+Vl}{2} \quad \text{..... (2.7)}$$

Combining these individual formulas yields a kinematic model given by eq. 2.8 [5]:

$$\dot{\xi}_I = R(\theta)^{-1}\,\dot{\xi}_R \xrightarrow{Gives:} \begin{bmatrix} \dot{x}_I \\ \dot{y}_I \\ \dot{\theta}_I \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{x}_R \\ \dot{y}_R \\ \dot{\theta}_R \end{bmatrix} = \begin{bmatrix} V\,cos\theta \\ Vsin\theta \\ \omega \end{bmatrix} = \begin{bmatrix} \frac{Vr+Vl}{2}cos\theta \\ \frac{Vr+Vl}{2}sin\theta \\ \frac{Vr-Vl}{2l} \end{bmatrix} \quad \text{..... (2.8)}$$

Note that neither wheel can contribute to sideways motion in the robot's frame, so $\dot{x}_R = V$, $\dot{\theta}_R = \omega$ and $\dot{y}_R = 0$.

During its motion, the mobile base calculates its position coordinates and orientation angles in real time and to build a near position transformation of the position in the global frame the following approximation are used:

- $\dot{X}_I \approx \frac{\Delta X}{\Delta T}$, $\dot{Y}_I \approx \frac{\Delta Y}{\Delta T}$, $\dot{\theta}_I \approx \frac{\Delta X}{\Delta T}$, while ΔT is very small.
- The curve between two positions is approximated by a line-segment with constant angle: $\theta + \frac{\Delta\theta}{2}$.
- ΔD_r , ΔD_l: are the traveled distances for the right and left wheels respectively.

Hence, eq. 2.9 is gotten for the left and right wheels respectively [5]:

$$\begin{bmatrix}\Delta X_I\\ \Delta Y_I\\ \Delta\theta_I\end{bmatrix} = \begin{bmatrix}\Delta D\ cos(\theta+\frac{\Delta\theta}{2})\\ \Delta D\ sin(\theta+\frac{\Delta\theta}{2})\\ \frac{(\Delta D_r-\Delta D_l)}{2l}\end{bmatrix} = \begin{bmatrix}\frac{\Delta D_r+\Delta D_l}{2}\cos(\theta+\frac{\Delta\theta}{2})\\ \frac{\Delta D_r+\Delta D_l}{2}\sin(\theta+\frac{\Delta\theta}{2})\\ \frac{\Delta D_r-\Delta D_l}{2l}\end{bmatrix} \quad \text{..... (2.9)}$$

And in the Global reference frame, the next position can be found by eq. 2.10 [5]:

$$\xi_I(k+1) = \xi_I(k) + \Delta\xi_I = \begin{bmatrix}X_I\\ Y_I\\ \theta_I\end{bmatrix} + \begin{bmatrix}\frac{\Delta D_r+\Delta D_l}{2}\cos(\theta+\frac{\Delta\theta}{2})\\ \frac{\Delta D_r+\Delta D_l}{2}\sin(\theta+\frac{\Delta\theta}{2})\\ \frac{\Delta D_r-\Delta D_l}{2l}\end{bmatrix} \quad \text{..... (2.10)}$$

II.2.2. The transformation matrix ${}^{A}_{B}T$ that defines the base

Assuming that the non-holonomic mobile base, RobuTER, moves on the plan, its kinematic model can be decided by three parameters $x_B, y_B\ and\ \theta_B$, which represent the Cartesian coordinates of O_B in R_A and the orientation angle of the mobile base. Hence the transformation matrix that defines the base is given by eq. 2.11:

$${}^{A}_{B}T = \begin{bmatrix}\cos\theta_B & -\sin\theta_B & 0 & x_B\\ \sin\theta_B & \cos\theta_B & 0 & y_B\\ 0 & 0 & 1 & z_B\\ 0 & 0 & 0 & 1\end{bmatrix} \quad \text{..... (2.11)}$$

II.2.3. Motion Control (Kinematic Control)

Motion control might not be an easy task for non-holonomic systems. However, different studies have been done on the topic and some adequate solutions for motion control of a mobile robot system are available:

II.2.3.1. Open loop control (trajectory-following)

The objective of a kinematic controller is to follow a trajectory described by its position or velocity profile versus time. This is often done by dividing the trajectory (path) into motion segments of clearly defined shape, for example, straight lines and segments of a circle [1]. The control problem is thus to pre-compute a smooth trajectory based on line and circle segments which drives the robot from the initial position to the final position as shown in figure 2.2. This approach can be regarded as open-loop motion control, because the measured robot position is not fed back for velocity or position control. It has several disadvantages such as:

- It is not at all an easy task to pre-compute a feasible trajectory if all limitations and constraints of the robot's velocities and accelerations have to be considered.
- The robot will not automatically adapt or correct the trajectory if dynamic changes of the environment occur.

- The resulting trajectories are usually not smooth, because the transitions from one trajectory segment to another are not smooth. This means there is a discontinuity in the robot's acceleration.

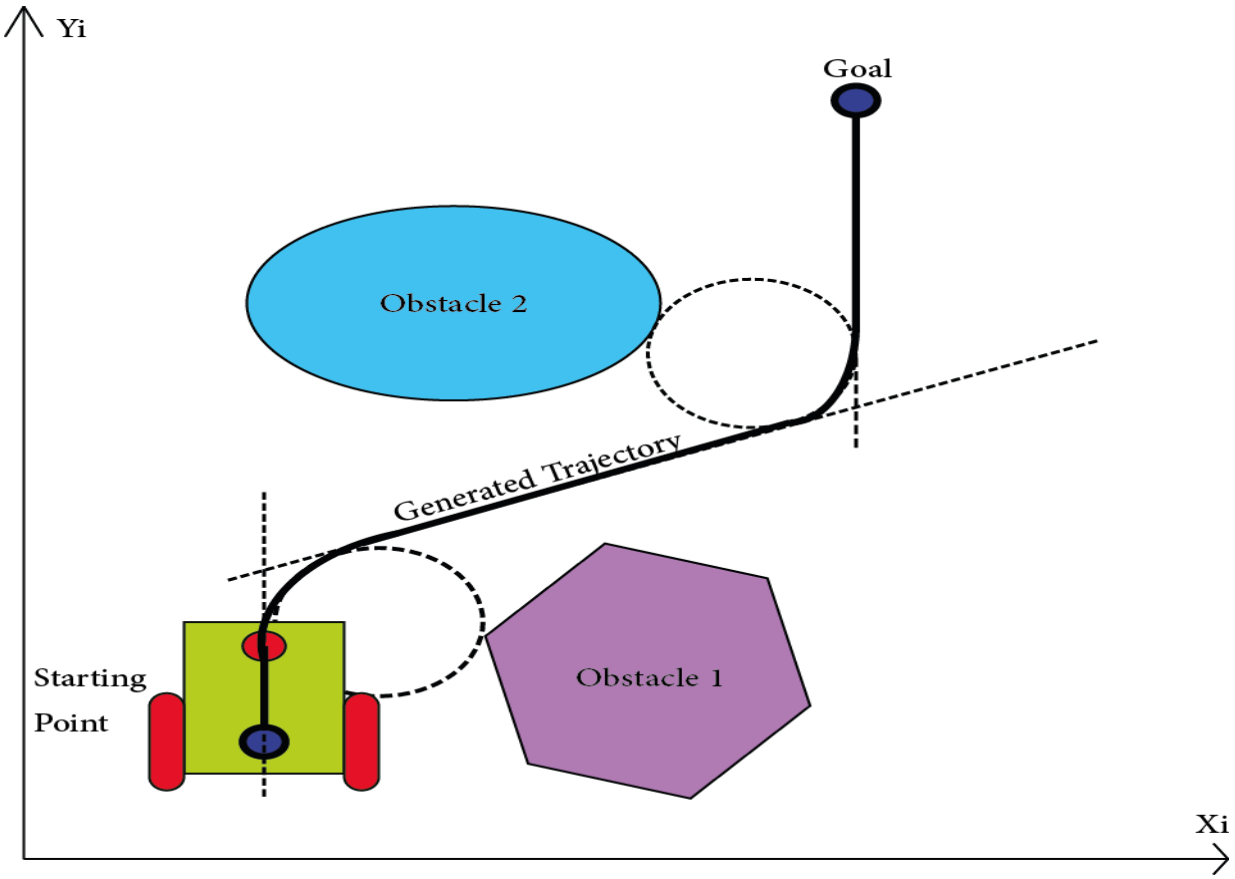


Figure 2.2. Open loop control of a mobile robot based on straight lines and circular trajectory segments.

II.2.3.2. Feedback control

In automatic control, feedback improves system performance by allowing the successful completion of a task even in the presence of external disturbances or initial errors.

A more appropriate approach in motion control of a mobile robot is to use a real-state feedback controller [1]. With such a controller the robot's path-planning task is reduced to setting intermediate positions (sub goals) lying on the requested path.

However, the most common approach to feedback motion planning in the presence of obstacles is based on potential fields [1]. Based on developed navigation functions (potential functions with a unique minimum at the goal and meeting certain other criteria) using potential functions in a generalized sphere world utilized a potential field over the operational space to guide a manipulator or mobile robot to the goal.

II.3. Kinematic analysis of the mobile arm

There are mainly two different spaces used in kinematics modeling of manipulators namely, Cartesian space and Quaternion space. The transformation between two Cartesian

coordinate systems can be decomposed into a rotation and a translation. There are many ways to represent rotation, including the following: Euler angles, Gibbs vector, Cayley-Klein parameters, Pauli spin matrices, axis and angle, orthonormal matrices, and Hamilton's quaternions [6]. However, homogenous transformations based on 4x4 real matrices (orthonormal matrices) have been used most often in robotics.

Although quaternions constitute an elegant representation for rotation, they have not been used as much as homogenous transformations by the robotics community. Dual quaternion can present rotation and translation in a compact form of transformation vector, simultaneously. While the orientation of a body is represented by nine elements in homogenous transformations, the dual quaternions reduce the number of elements to four. It offers considerable advantage in terms of computational robustness and storage efficiency for dealing with the kinematics of robot chains.

The robot kinematics can be divided into forward kinematics and inverse kinematics. The relationship between forward and inverse kinematics is illustrated in Figure 2.3.

Forward kinematics problem is straight forward and there is no complexity in deriving the equations, always there is a forward kinematics solution of a manipulator.

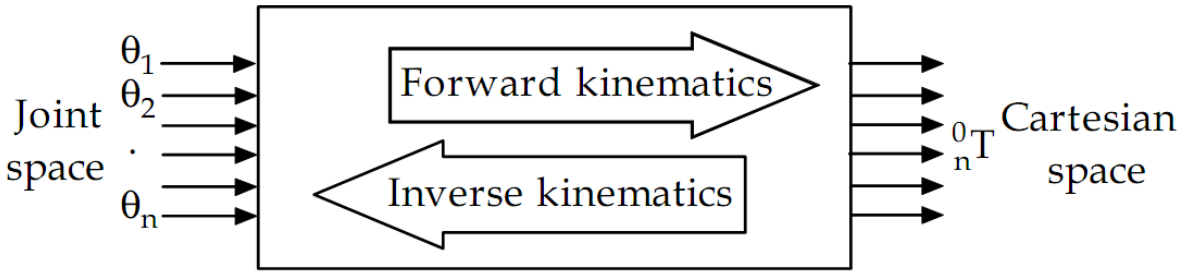


Figure 2.3. The schematic representation of forward and inverse kinematics.

For the inverse kinematics problem there are two main solution techniques: analytical and numerical methods. In the first type, the joint variables are solved analytically according to given configuration data. In the second type of solution, the joint variables are obtained based on the numerical techniques. In this chapter, the analytical solution of the manipulators is examined rather than numerical solution.

II.3.1. Forward Kinematics

Determining the position and orientation of the end-effector in a workspace frame by a known joint variables of a manipulator is the main problem in the forward kinematic. Each joint has a single degree of freedom.

Denavit & Hartenberg (1955) [6] showed that a general transformation between two joints requires four parameters. These parameters are known as the Denavit-Hartenberg (DH)

parameters. This method that uses the four parameters is the most common method for describing the robot kinematics. These parameters are:

a_{i-1}: The link length.

α_{i-1} : The link twist

d_i : The link offset

θ_i : The joint angle

II.3.1.1. The general robot manipulator model analysis

To determine DH parameters a coordinate frame is attached to each joint. Z_i axis of the coordinate frame is pointing along the rotary or sliding direction of the joints. Figure 2.4 shows the coordinate frame assignment for a general manipulator; such that:

- Z_i is a unit vector along the axis in space about which the link *i-1* and *i* are connected.
- The distance from Z_{i-1} to Z_i measured along X_{i-1} is assigned as a_{i-1} .
- The angle between Z_{i-1} and Z_i measured along X_i is assigned as α_{i-1}.
- The distance from X_{i-1} to X_i measured along Z_i is assigned as d_i .
- The angle between X_{i-1} to X_i measured about Z_i is assigned as θ_i [6].

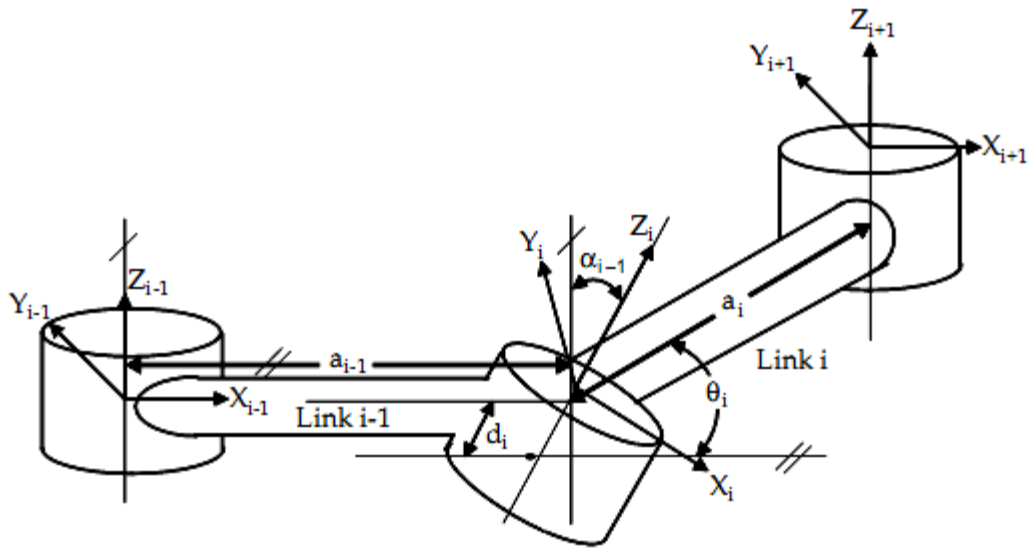


Figure 2.4. Coordinate frame assignment for a general manipulator.

The general transformation matrix ${}^{i-1}_{i}T$ for a single link can be obtained as follows (eq. 2.12):

$${}^{i-1}_{i}T = R_X(\alpha_{i-1}).D_X(a_{i-1}).R_Z(\theta_i).Q_i(d_i) = Rot(X,\alpha_i).Trans(X,a_i).Rot(Z,\theta_i).Trans(Z,d_i)$$

… ..... (2.12)

Where the notation:

- $Rot(X,\alpha_i)$: Stands for rotation around X_i axis by α_i.

- $Trans(X, a_i)$: Is a transition along X_i axis by a distance a_i.
- $Rot(Z, \theta_i)$: Stands for rotation around Z_i axis by θ_i.
- $Trans(Z, d_i)$: The transition along Z_i by a distance d_i.

Hence we have for a single link the transition matrix can be written by eq. 2.13 and eq. 2.14:

$$^{i-1}_{i}T = R_X(\alpha_{i-1}).D_X(a_{i-1}).R_Z(\theta_i).Q_i(d_i)$$

$$^{i-1}_{i}T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c\alpha_{i-1} & -s\alpha_{i-1} & 0 \\ 0 & s\alpha_{i-1} & c\alpha_{i-1} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 1 & 0 & 0 & a_{i-1} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} c\theta_i & -s\theta_i & 0 & 0 \\ s\theta_i & c\theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

.. (2.13)

$$^{i-1}_{i}T = \begin{bmatrix} c\theta_i & -s\theta_i & 0 & a_{i-1} \\ c\alpha_{i-1}.s\theta_i & c\alpha_{i-1}.c\theta_i & -s\alpha_{i-1} & -d_i.s\alpha_{i-1} \\ s\alpha_{i-1}.s\theta_i & s\alpha_{i-1}.c\theta_i & c\alpha_{i-1} & d_i.c\alpha_{i-1} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{..... (2.14)}$$

Where:

- $c\theta_i$: The short hands $cos\theta_i$.
- $s\theta_i$: The short hands of $sin\theta_i$.
- $c\alpha_{i-1}$: The short hands $\cos\alpha_{i-1}$.
- $s\alpha_{i-1}$: The short hands of $\sin\alpha_{i-1}$.

Each homogenous transformation is of the form of the matrix:

$$^{i-1}_{i}T = \begin{bmatrix} ^{i-1}_{i}R & ^{i-1}_{i}P \\ 0 & 1 \end{bmatrix} \quad \text{..... (2.15)}$$

Where:

- $^{i-1}_{i}P$: Three-dimensional vector denoting the position.
- $^{i-1}_{i}R$: 3x3 rotational matrix.

II.3.1.2. Full robot joints transformation

Now, suppose a robot has n-1 Links numbered from zero to n-1 starting from the base of the robot as link 0 to the end-effector as link n-1. The joints are numbered from 1 to n. The i^{th} joint variable is denoted by q_i.

The matrix $^{i-1}_{i}T$ is not constant, but varies according to the change of the robot configuration, however, $^{i-1}_{i}T$ is a function of only a single joint variable, namely q_i , as shown in eq. 2.16:

$$^{i-1}_{i}T = {}^{i-1}_{i}T(q_i) \quad \text{..... (2.16)}$$

Then for n-joint, the position and orientation of the end-effector in the inertial frame is determined by multiplication of all the ${}^{i-1}_{i}T$ matrices:

$${}^{0}_{n}T(q_1, q_2 \ldots . q_n) = {}^{0}_{1}T(q_1).{}^{1}_{2}T(q_2).\ldots\ldots{}^{n-1}_{n}T(q_n)\ldots\ldots\ldots \quad (2.17)$$

Hence,

$${}^{base}_{end\_effector}T = {}^{0}_{1}T(q_1).{}^{1}_{2}T(q_2).\ldots\ldots{}^{n-1}_{n}T(q_n)\ldots\ldots\ldots \quad (2.18)$$

An alternative representation of ${}^{base}_{end\_effector}T$ can be written as eq. 2.19:

$${}^{base}_{end\_effector}T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} {}^{j}_{i}R & {}^{j}_{i}P \\ 0 & 1 \end{bmatrix}\ldots\ldots \quad (2.19)$$

Where:

- r_{kj}: Represent the rotational elements of transformation matrix (k and j = 1, 2 and 3).
- ${}^{j}_{i}R$: Express the orientation of frame i relative to frame j $(i > j)$ and is given as: ${}^{j}_{i}R = {}^{j}_{j+1}R \ldots {}^{i-1}_{i}R$.
- p_x, p_y, p_z: Denote the elements of the position vector.
- ${}^{j}_{i}P$: Express the vector position and is given by for $(i > j)$: ${}^{j}_{i}P = {}^{j}_{j+1}P + {}^{j}_{i-1}R\,.\,{}^{i-1}_{i}P$.

II.3.1.3. ULM arm manipulator model analysis and transformation matrix ${}^{M}_{E}T$

For the RobuTER/ULM arm of six jointed manipulator or six degrees of freedom the position coordinates and orientation angles of the end-effector are calculated in $R_M = (O_M, \vec{x}_M, \vec{y}_M, \vec{Z}_M)$ by using the *Modified Denavit-Hartenberg (MDH)* representation where the transformation matrix linking the base to the end-effector is constructed first as:

$${}^{M}_{E}T = {}^{base}_{end\_effector}T = {}^{0}_{1}T(q_1).{}^{1}_{2}T(q_2).{}^{2}_{3}T(q_3).{}^{3}_{4}T(q_4).{}^{4}_{5}T(q_5).{}^{5}_{6}T(q_6).{}^{6}_{7}T(q_7)\ldots\ldots \quad (2.20)$$

Now, to calculate the different joints transformation from 0 to 6 the matrix given by eq. 2.21 is used:

$${}^{i-1}_{i}T = \begin{bmatrix} c\theta_i & -s\theta_i & 0 & a_{i-1} \\ c\alpha_{i-1}.s\theta_i & c\alpha_{i-1}.c\theta_i & -s\alpha_{i-1} & -d_i.s\alpha_{i-1} \\ s\alpha_{i-1}.s\theta_i & s\alpha_{i-1}.c\theta_i & c\alpha_{i-1} & d_i.c\alpha_{i-1} \\ 0 & 0 & 0 & 1 \end{bmatrix}\ldots\ldots\ldots\ldots\ldots \quad (2.21)$$

However, first the MDH for the ULM manipulator is needed.

The different MDH parameters α_i, d_i, θ_i and a_i and the joints limits of the ULM manipulator are given in the following table (Table 2):

Table 2.1. The MDH parameters and joints limits of the ULM Manipulator.

i	Denavit-Hartenberg (DH) parameters				Joints limits	
	$\alpha_{i-1}(°)$	$d_i(mm)$	θ_i	$a_i(mm)$	$Q_{min}(°)$	$Q_{max}(°)$
1	0	d1=290	θ_1	0	-95	96
2	90	d2=108.49	θ_2	0	-24	88
3	-90	d3=113	0	a_3= 402	--	--
4	90	0	θ_3	0	-2	160
5	90	d4=389	θ_4	0	-50	107
6	-90	0	θ_5	0	-73	40
7	90	deff=220	θ_6	0	-91	91

It is straightforward to compute each of the link transformation matrices using eq. 2.21, as follows:

$$ {}^{0}_{1}T = \begin{bmatrix} c\theta_1 & -s\theta_1 & 0 & a_1 \\ c\alpha_0.s\theta_1 & c\alpha_0.c\theta_1 & -s\alpha_0 & -d_1.s\alpha_0 \\ s\alpha_0.s\theta_1 & s\alpha_0.c\theta_1 & c\alpha_0 & d_1.c\alpha_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} c\theta_1 & -s\theta_1 & 0 & 0 \\ s\theta_1 & c\theta_1 & 0 & 0 \\ 0 & 0 & 1 & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \dots\dots\dots (2.22) $$

$$ {}^{1}_{2}T = \begin{bmatrix} c\theta_2 & -s\theta_2 & 0 & a_2 \\ c\alpha_1.s\theta_2 & c\alpha_1.c\theta_2 & -s\alpha_1 & -d_2.s\alpha_1 \\ s\alpha_1.s\theta_2 & s\alpha_1.c\theta_2 & c\alpha_1 & d_2.c\alpha_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} c\theta_2 & -s\theta_2 & 0 & 0 \\ 0 & 0 & -1 & -d_2 \\ s\theta_2 & c\theta_2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \dots\dots (2.23) $$

$$ {}^{2}_{3}T = \begin{bmatrix} c0 & -s0 & 0 & a_3 \\ c\alpha_2.s0 & c\alpha_2.c0 & -s\alpha_2 & -d_3.s\alpha_2 \\ s\alpha_2.s0 & s\alpha_2.c0 & c\alpha_2 & d_3.c\alpha_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & a_3 \\ 0 & 0 & 1 & d_3 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \dots\dots\dots\dots (2.24) $$

$$ {}^{3}_{4}T = \begin{bmatrix} c\theta_3 & -s\theta_3 & 0 & a_4 \\ c\alpha_3.s\theta_3 & c\alpha_3.c\theta_3 & -s\alpha_3 & -0.s\alpha_3 \\ s\alpha_3.s\theta_3 & s\alpha_3.c\theta_3 & c\alpha_3 & 0.c\alpha_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} c\theta_3 & -s\theta_3 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ s\theta_3 & c\theta_3 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \dots\dots\dots (2.25) $$

$$ {}^{4}_{5}T = \begin{bmatrix} c\theta_4 & -s\theta_4 & 0 & a_5 \\ c\alpha_4.s\theta_4 & c\alpha_4.c\theta_4 & -s\alpha_4 & -d_4.s\alpha_4 \\ s\alpha_4.s\theta_4 & s\alpha_4.c\theta_4 & c\alpha_4 & d_4.c\alpha_4 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} c\theta_4 & -s\theta_4 & 0 & 0 \\ 0 & 0 & -1 & -d_4 \\ s\theta_4 & c\theta_4 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \dots\dots\dots (2.26) $$

$$ {}^{5}_{6}T = \begin{bmatrix} c\theta_5 & -s\theta_5 & 0 & a_6 \\ c\alpha_5.s\theta_5 & c\alpha_5.c\theta_5 & -s\alpha_5 & -0.s\alpha_5 \\ s\alpha_5.s\theta_5 & s\alpha_5.c\theta_5 & c\alpha_5 & 0.c\alpha_5 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} c\theta_5 & -s\theta_5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -s\theta_5 & -c\theta_5 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \dots\dots\dots\dots (2.27) $$

$$ {}^{6}_{7}T = \begin{bmatrix} c\theta_6 & -s\theta_6 & 0 & a_7 \\ c\alpha_6.s\theta_6 & c\alpha_6.c\theta_6 & -s\alpha_6 & -d_{eff}.s\alpha_6 \\ s\alpha_6.s\theta_6 & s\alpha_6.c\theta_6 & c\alpha_6 & d_{eff}.c\alpha_6 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} c\theta_6 & -s\theta_6 & 0 & 0 \\ 0 & 0 & -1 & -d_{eff} \\ s\theta_6 & c\theta_6 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \dots (2.28) $$

Hence, the Transformation matrix from manipulator to the end-effector is given by eq. 2.29 and 2.30 as follow:

$$ {}^{M}_{E}T = {}^{0}_{1}T(q_1).{}^{1}_{2}T(q_2).{}^{2}_{3}T(q_3).{}^{3}_{4}T(q_4).{}^{4}_{5}T(q_5).{}^{5}_{6}T(q_6).{}^{6}_{7}T(q_7) \dots\dots\dots\dots (2.29) $$

$$ {}^{M}_{E}T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \dots\dots\dots\dots (2.30) $$

Where:

- $r_{11} = -\,\mathrm{c6}.(\mathrm{s5}.(\mathrm{c1}.\mathrm{c2}.\mathrm{s3} + \mathrm{c1}.\mathrm{c3}.\mathrm{s2}) - \mathrm{c4}.\mathrm{c5}.(\mathrm{c1}.\mathrm{c2}.\mathrm{c3} - \mathrm{c1}.\mathrm{s2}.\mathrm{s3})) - \mathrm{s4}.\mathrm{s6}.(\mathrm{c1}.\mathrm{c2}.\mathrm{c3} - \mathrm{c1}.\mathrm{s2}.\mathrm{s3})$
- $r_{12} = \mathrm{s6}.\big(\mathrm{s5}.(\mathrm{c1}.\mathrm{c2}.\mathrm{s3} + \mathrm{c1}.\mathrm{c3}.\mathrm{s2}) - \mathrm{c4}.\mathrm{c5}.(\mathrm{c1}.\mathrm{c2}.\mathrm{c3} - \mathrm{c1}.\mathrm{s2}.\mathrm{s3})\big) - \mathrm{c6}.\mathrm{s4}.(\mathrm{c1}.\mathrm{c2}.\mathrm{c3} - \mathrm{c1}.\mathrm{s2}.\mathrm{s3})$
- $r_{13} = c5.(c1.c2.s3 + c1.c3.s2) + c4.s5.(c1.c2.c3 - c1.s2.s3)$
- $r_{21} = -\,\mathrm{c6}.(\mathrm{s5}.(\mathrm{c2}.\mathrm{s1}.\mathrm{s3} + \mathrm{c3}.\mathrm{s1}.\mathrm{s2}) - \mathrm{c4}.\mathrm{c5}.(\mathrm{c2}.\mathrm{c3}.\mathrm{s1} - \mathrm{s1}.\mathrm{s2}.\mathrm{s3})) - \mathrm{s4}.\mathrm{s6}.(\mathrm{c2}.\mathrm{c3}.\mathrm{s1} - \mathrm{s1}.\mathrm{s2}.\mathrm{s3})$
- $r_{22} = s6.\big(s5.(c2.s1.s3 + c3.s1.s2) - c4.c5.(c2.c3.s1 - s1.s2.s3)\big) - c6.s4.(c2.c3.s1 - s1.s2.s3)$
- $r_{23} = c5.(c2.s1.s3 + c3.s1.s2) + c4.s5.(c2.c3.s1 - s1.s2.s3)$
- $r_{31} = \mathrm{c6}.(\mathrm{s5}.(\mathrm{c2}.\mathrm{c3} - \mathrm{s2}.\mathrm{s3}) + \mathrm{c4}.\mathrm{c5}.(\mathrm{c2}.\mathrm{s3} + \mathrm{c3}.\mathrm{s2})) - \mathrm{s4}.\mathrm{s6}.(\mathrm{c2}.\mathrm{s3} + \mathrm{c3}.\mathrm{s2})$
- $r_{32} = -\,s6.(s5.(c2.c3 - s2.s3) + c4.c5.(c2.s3 + c3.s2)) - c6.s4.(c2.s3 + c3.s2)$
- $r_{33} = c4.s5.(c2.s3 + c3.s2) - c5.(c2.c3 - s2.s3)$
- $p_x = d2.s1 + deff.(c5.(c1.c2.s3 + c1.c3.s2) + c4.s5.(c1.c2.c3 - c1.s2.s3)) + d4.(c1.c2.s3 + c1.c3.s2) + a3.c1.c2 - c1.d3.s2$
- $p_y = d4.(c2.s1.s3 + c3.s1.s2) - c1.d2 + deff.(c5.(c2.s1.s3 + c3.s1.s2) + c4.s5.(c2.c3.s1 - s1.s2.s3)) + a3.c2.s1 - d3.s1.s2$
- $p_z = d1 + c2.d3 + a3.s2 - d4.(c2.c3 - s2.s3) - deff.(c5.(c2.c3 - s2.s3) - c4.s5.(c2.s3 + c3.s2))$

Note: Here just for simplicity we used: $s\theta_i = \mathrm{si},\ c\theta_i = \mathrm{ci}, \text{where } (i = 1 \dots 6)$.

And we used trigonometric identities:

$$s(i \pm j) = s_{ij}^{\pm} = si.cj \pm ci.sj$$

$$c(i \pm j) = c_{ij}^{\pm} = ci.cj \mp si.sj$$

where: $(i = 1 \dots 6), (j = 1 \dots 6)$

Hence, the final simplified format is given by:

- $r_{11} = \mathrm{c6.c4.c1.c23.c5} - \mathrm{c6.c1.s23.s5} - \mathrm{s6.s4.c1.c23}$
- $r_{12} = \mathrm{s6.c1.s23.s5} - \mathrm{s6.c4.c1.c23.c5} - \mathrm{c6.s4.c1.c23}$
- $r_{13} = c1.s23.c5 + c4.c1.c23.s5$
- $r_{21} = \mathrm{c6.c4.s1.c23.c5} - \mathrm{c6.s1.s23.s5} - \mathrm{s6.s4.s1.c23}$
- $r_{22} = s6.s1.s23.s5 - s6.c4.s1.c23.c5 - c6.s4.s1.c23$
- $r_{23} = s1.s23.c5 + c4.s1.c23.s5$
- $r_{31} = \mathrm{c6.s5.c23} + \mathrm{c6.c4.c5.s23} - \mathrm{s4.s6.s23}$
- $r_{32} = -s6.s5.c23 - s6.c4.c5.s23 - c6.s4.s23$
- $r_{33} = c4.s23.s5 - c23.c5$
- $p_x = d2.s1 + deff.(c5.c1.s23 + c4.s5.c1.c23) + d4.c1.s23 + a3.c1.c2 - c1.d3.s2$
- $p_y = d4.s1.s23 - c1.d2 + deff.(c5.s1.s23 + c4.s5.s1.c23) + a3.c2.s1 - d3.s1.s2$
- $p_z = d1 + c2.d3 + a3.s2 - d4.c23 - deff.(c5.c23 - c4.s5.s23)$

II.3.2. Inverse Kinematics

The Inverse Kinematics (IK) analyses of the serial manipulators have been putted to studies for many decades. It is needed in the control of manipulators. Solving the inverse kinematics is computationally expensive and generally takes a very long time in the real time control of manipulators. The aim here is to find a solution to the problem of IK using a Geometric Approach and an Algebraic Approach by determining the joint angles for desired position and orientations in Cartesian space. Hence, the ULM arm transformation matrix defined by eq. 2.30 is used to build an inverse kinematic analysis. The IK is more complex to deal with then the forward kinematic.

II.3.2.1. Geometric approach

First we specify the target position of the end-effector by (x, y, z) in the Cartesian space where:

- z is the height relative to the base.
- (x, y) are the 2D Cartesian space position.

The inverse kinematic equations that will be built can be solved in a closed manner.

From figure 2.5, which shows the top view of the ULM manipulator range of rotation in Cartesian space, it can be seen clearly that the distance d and the position estimation x_d and y_d are equal to:

$$d = \sqrt{x_d^2 + y_d^2}$$

$$x_d = \cos\theta_1$$

$$y_d = \sin\theta_1$$

$$\theta_1 = atan2(x, y)$$

The angles $\theta_1, \theta_2, \theta_3, \theta_4, \theta_5$ and θ_6 correspond to the joints q_1, q_2, q_3, q_4, q_5 and q_6 respectively filling the range angles as shown in figures 2.5(a), 2.6(a), between:

$$-135° \leq \theta_1 \leq 135°$$

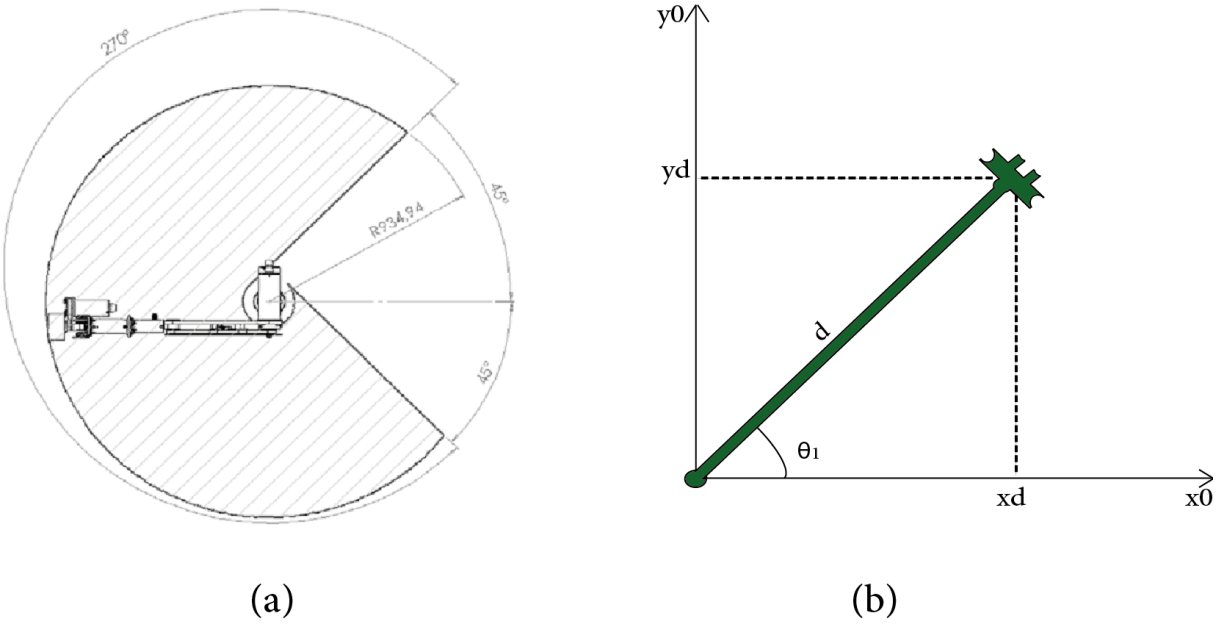
$$-120° \leq \theta_2 \leq 120°$$

$$-90° \leq \theta_3 \leq 90°$$

$$0° \leq \theta_4 \leq 360°$$

$$-55° \leq \theta_5 \leq 55°$$

$$0° \leq \theta_6 \leq 360°$$



(a) (b)

Figure 2.5. (a) Tope View of the ULM manipulator. (b) Top view of the arm in Cartesian space

The lengths d_1, d_2, d_3, a_3, d_4 and d_{eff} as shown in figure 2.6(a) and 2.6(b) are the main parameters that specify our manipulator (the ULM arm), they are essential parameters to use in our geometric analysis.

δ is a small constant that allows the end-effector to pick up objects without changing its Cartesian position or orientation, we look for a solution to the inverse kinematics as a closed form in the case of δ is already fixed and adapted by the manufacturer of the ULM arm.

From figure 2.6 (b), we find a relationship between $\delta, \theta_2, \theta_3 \; and \; \theta_5$ as:

$$\delta \approx (\theta_2 + \theta_3) - \theta_5 \quad (2.31)$$

Looking for the radial distance and height at joint q_5 :

$$r_5 = r_{eff} - d_{eff}\cos(\delta) \quad or \quad r_5 = a_3\cos(\theta_2) + d_4\cos(\theta_2 + \theta_3) \quad (2.32)$$

$$z_5 = z_{eff} - d_{eff}\sin(\delta) \quad or \quad z_5 = a_3\sin(\theta_2) + d_4\sin(\theta_2 + \theta_3) + (d_1 + d_3) \quad (2.33)$$

Before looking for the angles $\theta_2, \theta_3 \; and \; \theta_5$ geometrically, $\beta, a \; and \; s$ must be found first from figure 2.6(b) by eq. 2.34, eq. 2.35 and eq. 2.36:

$$\beta = atan2(s^2 + a_3^2 - d_4^2, 2a_3 s) \quad (2.34)$$

$$a = atan2(z_5 - d_1, r_5) \quad (2.35)$$

$$s = \sqrt{(z_5 - d_1)^2 + {r_5}^2} \quad (2.36)$$

Hence, the desired angels are:

$$\theta_2 = a \pm \beta \quad (2.37)$$

$$\theta_3 = atan2(s^2 - a_3^2 - d_4^2, 2a_3 d_4) \quad (2.38)$$

$$\theta_5 \approx (\theta_2 + \theta_3) - \delta \quad (2.39)$$

Note that: $\theta_4 \; and \; \theta_6$ are both rotating symmetrically around the axis of joints $q_4 \; and \; q_6$ respectively where the range of rotation for both has already been set:

$$0° \leq (\theta_4, \theta_6) \leq 360°$$

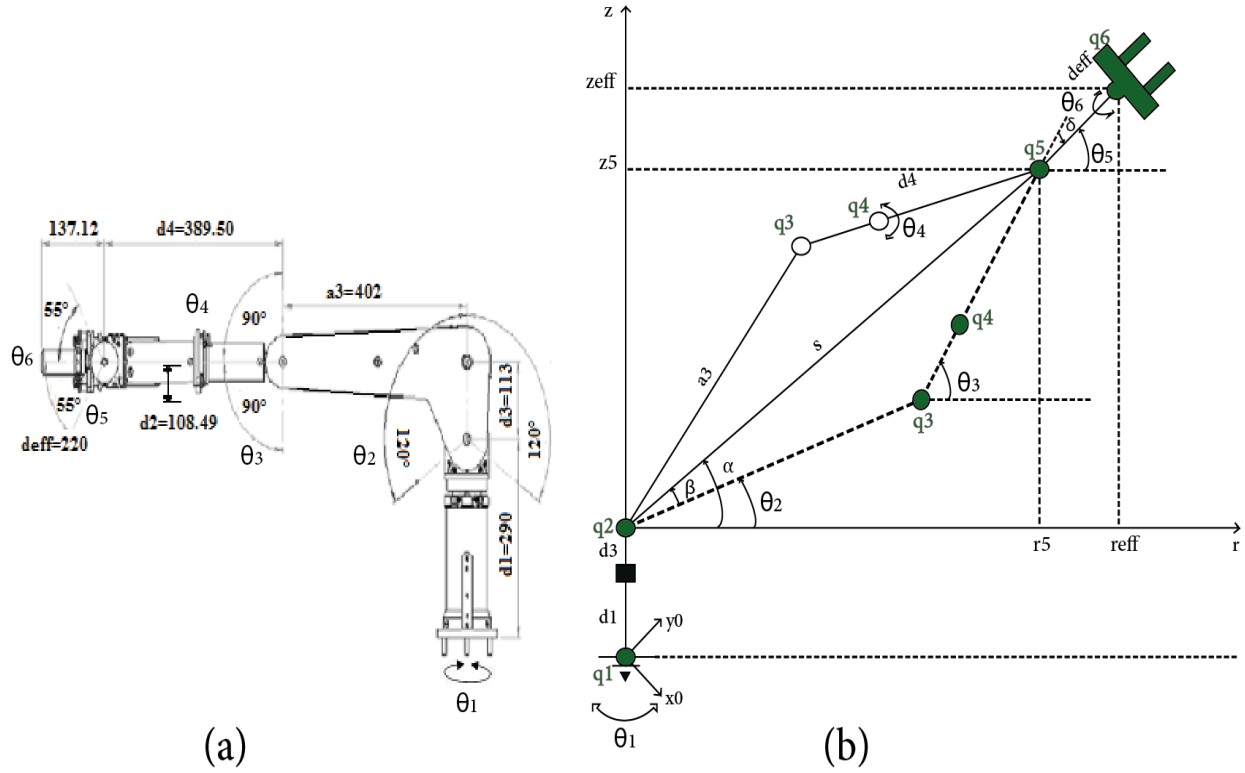


Figure 2.6. (a) Planner view of the ULM. (b) Planner view of the 6 DOF robotic arm.

II.3.2.2. Analytical or algebraic approach

The position vector denoted by the elements(p_x, p_y, p_z), that has been calculated in eq. 2.30, is used to solve the IK.

$$\begin{cases} p_x = d2.s1 + deff.(c5.c1.s23 + c4.s5.c1.c23) + d4.c1.s23 + a3.c1.c2 - c1.d3.s2 \\ p_y = d4.s1.s23 - c1.d2 + deff.(c5.s1.s23 + c4.s5.s1.c23) + a3.c2.s1 - d3.s1.s2 \\ p_z = d1 + c2.d3 + a3.s2 - d4.c23 - deff.(c5.c23 - c4.s5.s23) \end{cases} \quad \text{..... (2.40)}$$

$\rightarrow$ *Finding* θ_1:

We can calculate the angle θ_1 just from the geometric approach as shown in figure 2.5.(b), Hence:

$$\theta_1 = atan2(x, y) \quad \text{..... (2.41)}$$

$\rightarrow$ *Finding* θ_3:

Eq. 2.40 can be written as follow:

$$\begin{cases} p_x - d2.s1 = deff.(c5.c1.s23 + c4.s5.c1.c23) + d4.c1.s23 + a3.c1.c2 - c1.d3.s2 \text{ ..... (2.4} \\ p_y - d4.s1.s23 = -c1.d2 + deff.(c5.s1.s23 + c4.s5.s1.c23) + a3.c2.s1 - d3.s1.s2 \text{ ..... (2.4} \\ p_z = d1 + c2.d3 + a3.s2 - d4.c23 - deff.(c5.c23 - c4.s5.s23) \text{ ..... (2.} \end{cases}$$

Squaring the two sides of eq. 2.40a and 2.40b then sum them gives eq. 42 below:

- $(p_x - d2.s1)^2 + (p_y + c1.d2)^2 =$

$$= c1^2(deff.(c5.s23 + c4.s5.c23) + d4.s23 + a3.c2 - d3.s2)^2$$
$$+ s1^2(deff.(c5.s23 + c4.s5.c23) + d4.s23$$
$$+ a3.c2 - d3.s2)^2$$
$$= (c1^2 + s1^2).(deff.(c5.s23 + c4.s5.c23) + d4.s23 + a3.c2 - d3.s2)^2$$
$$= (deff.(c5.s23 + c4.s5.c23) + d4.s23 +$$
$$a3.c2 - d3.s2)^2 \ldots\ldots\ldots\ldots\ldots\ldots (2.42)$$

Where: $ci^2 + si^2 = 1$

Hence eq. 2.42 and eq. 2.40c can be simplified to get eq. 2.43a and eq. 2.43b:

$$\begin{cases} a3.c2 + d4.s23 = \pm\sqrt{(p_x - d2.s1)^2 + (p_y + c1.d2)^2} - deff.(c5.s23 + c4.s5.c23) + d3.s2 \ldots\ldots (2.43a) \\ a3.s2 - d4.c23 = p_z + deff.(c5.c23 - c4.s5.s23) - d3.c2 - d1 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2.43b) \end{cases}$$

Squaring both sides of eq.2.43 and adding them together leads to:

- $(a3.c2 + d4.s23)^2 + (a3.s2 - d4.c23)^2 =$

$$= \left(\pm\sqrt{(p_x - d2.s1)^2 + (p_y + c1.d2)^2} - deff.(c5.s23 + c4.s5.c23) + d3.s2\right)^2$$
$$+ (p_z + deff.(c5.c23 - c4.s5.s23) - d3.c2 - d1)^2$$

- $a3^2 + d4^2 - 2a3.d4.(s2c23 - c2.s23) =$

$$= a3^2 + d4^2 - 2a3.d4.s3$$

$$= \left(\pm\sqrt{(p_x - d2.s1)^2 + (p_y + c1.d2)^2} - deff.(c5.s23 + c4.s5.c23) + d3.s2\right)^2 +$$
$$(p_z + deff.(c5.c23 - c4.s5.s23) - d3.c2 - d1)^2$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2.44)$$

So, clearly $s3 = \sin\theta_3$ and $c3 = \text{cin}\,\theta_3$ can be found as follow:

$$s3 = -\frac{\left(\pm\sqrt{(p_x - d2.s1)^2 + (p_y + c1.d2)^2} - deff.(c5.s23 + c4.s5.c23) + d3.s2\right)^2 + (p_z + deff.(c5.c23 - c4.s5.s23) - d3.c2 - d1)^2 - a3^2 - d4^2}{2a3.d4}$$

$$c3 = \pm\sqrt{1 - s^2}$$

Hence, θ_3 is given by:

$$\theta_3 = atan2(s3, c3) \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2.45)$$

$\rightarrow Finding\ \theta_2$:

From figure 2.6(b) eq. 2.46 is gotten:

$$\theta_2 = a - \beta \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2.46)$$

And we know that:

- $a = atan2(p_z + deff.(c5.c23 - c4.s5.s23) - d3.c2 - d1 \quad ,$
  $\pm\sqrt{(p_x - d2.s1)^2 + (p_y + c1.d2)^2} - deff.(c5.s23 + c4.s5.c23) + d3.s2)$
- $\beta = atan2(d4.s3 \ , a_3 + d4.c3)$

Hence, $\theta_2$is given by:

$$\theta_2 = atan2\Big(p_z + deff.(c5.c23 - c4.s5.s23) - d3.c2 - d1 \quad ,$$

$$\pm\sqrt{(p_x - d2.s1)^2 + \big(p_y + c1.d2\big)^2} - deff.(c5.s23 + c4.s5.c23) + d3.s2\Big) -$$

$$atan2(d4.s3 \ , a_3 + d4.c3)$$

…… (2.47)

$\rightarrow$ *Finding* θ_5:

From the transformation matrix ${}^{M}_{E}T$ given by eq. 2.30 and after applying trigonometric identities, eq. 2.48 can be written:

$$\begin{cases} r_{11} = \mathrm{c6.c4.c1.c23.c5} - \mathrm{c6.c1.s23.s5} - \mathrm{s6.s4.c1.c23} \\ r_{12} = \mathrm{s6.c1.s23.s5} - \mathrm{s6.c4.c1.c23.c5} - \mathrm{c6.s4.c1.c23} \\ r_{13} = c1.s23.c5 + c4.c1.c23.s5 \\ r_{21} = \mathrm{c6.c4.s1.c23.c5} - \mathrm{c6.s1.s23.s5} - \mathrm{s6.s4.s1.c23} \\ r_{22} = s6.s1.s23.s5 - s6.c4.s1.c23.c5 - c6.s4.s1.c23 \\ r_{23} = s1.s23.c5 + c4.s1.c23.s5 \\ r_{31} = \mathrm{c6.c23.s5} + \mathrm{c6.c4.s23.c5} - \mathrm{s6.s4.s23} \\ r_{32} = -s6.c23.s5 - s6.c4.s23.c5 - c6.s4.s23 \\ r_{33} = c4.s23.s5 - c23.c5 \end{cases} \quad \text{……(2.48)}$$

The following pairs of algebraic equations taken from 2.48 are solved to find $c5$:

$$\begin{cases} s23.r_{23} = s23.(s1.s23.c5 + c4.s1.c23.s5) \\ (-s1.c23).r_{33} = (-s1.c23).(c4.s23.s5 - c23.c5) \end{cases} \quad \text{…… (2.49)}$$

By adding both sides of eq. 2.49 *c5* is gotten:

$$c5 = \frac{s23.r_{23} - s1.c23.r_{33}}{s1} \quad \text{…… (2.50)}$$

The following pairs of algebraic equations taken from 2.48 are also solved to find $s5$:

$$\begin{cases} s23.r_{22} = s23.(s6.s1.s23.s5 - s6.c4.s1.c23.c5 - c6.s4.s1.c23) \\ (-s1.c23).r_{32} = (-s1.c23).(-s6.c23.s5 - s6.c4.s23.c5 - c6.s4.s23) \end{cases} \quad \text{… (2.51)}$$

Adding both sides of eq. 2.51, *s5* is gotten:

$$s5 = \frac{s23.r_{23} - s1.c23.r_{33}}{s6.s1} \quad \text{…… (2.52)}$$

Hence, from eq. 2.51 and eq. 2.52, $\theta_5$is given by:

$$\theta_5 = atan2(s5, c5) \quad \text{…… (2.53)}$$

$\rightarrow Finding\ \theta_4\ and\ \theta_6$:

As it has already mentioned $\theta_4\ and\ \theta_6$ are both rotating symmetrically in joints $q_4 and\ q_6$ respectively around the axis that links the joints $q_3\ to\ q_5\ and\ q_5$ with the end-effector, where the range of rotation for both is already set:

$$0° \leq (\theta_4\ , \theta_6) \leq 360°$$

II.4. Kinematic analysis of the mobile manipulator (RobuTER/ULM)

In this section, the full system from manipulator to base will be analyzed. This involves the interaction between the mobile base and the manipulator. This analysis is based in the direct kinematic transformation matrixes that have been derived. The location of the end-effector is given in $R_A = (O_A, \vec{x}_A\ , \vec{y}_A\ , \vec{Z}_A)$ by:

$$^{A}_{E}T = {}^{A}_{B}T . {}^{B}_{M}T . {}^{M}_{E}T$$

The transformations of $^{A}_{B}T$, $^{M}_{E}T$ of the RobuTER base and the ULM manipulator respectively are already obtained. Now, for the transformation defining the base to the manipulator is given by letting: (x_B, y_B, z_B) are the Cartesian coordinates of O_B in R_A and (x_M, y_M, z_M) are the Cartesian coordinates of O_M in R_B . Hence, the transformation matrix $^{B}_{M}T$ is denoted by:

$$^{B}_{M}T = \begin{bmatrix} 1 & 0 & 0 & x_M \\ 0 & 1 & 0 & y_M \\ 0 & 0 & 1 & z_M \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{.....} \quad (2.54)$$

So, now the total transformation matrix linking base to the end-effector (This analysis involves the interaction between the mobile base and the manipulator) can be constructed as:

$$^{A}_{E}T = {}^{A}_{B}T . {}^{B}_{M}T . {}^{M}_{E}T$$

$$^{A}_{E}T = \begin{bmatrix} \cos\theta_B & -\sin\theta_B & 0 & x_B \\ \sin\theta_B & \cos\theta_B & 0 & y_B \\ 0 & 0 & 1 & z_B \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 1 & 0 & 0 & x_M \\ 0 & 1 & 0 & y_M \\ 0 & 0 & 1 & z_M \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{.....} \quad (2.55)$$

II.5. Conclusion

This chapter provides a deep explanation of the kinematic analysis of the RobuTER/ULM robot. Where both direct kinematic models of the base to arm and base to manipulator are built based on the transformation matrixes. Then, an inverse kinematic analysis based in geometric approach and analytic approach to find the mathematical model of the different joints angels has been constructed. Finally, this chapter ends up by describing the Kinematic analysis of the mobile manipulator by building the final matrix which describes the robot model.

III. Introduction

One of the most important tasks of an autonomous system of any kind is to acquire knowledge about its environment. This is done by taking measurements using various sensors and then extracting meaningful information from those measurements. There are a wide variety of sensors used in mobile robots. Some sensors are used to measure simple values like the rotational speed of the motors. Other, more sophisticated sensors can be used to acquire information about the robot's environment or even to directly measure a robot's global position and in constructing the environments map. This chapter focuses primarily on sensors used to extract information about the robot's environment (Kinect 360, Ultrasonic captures and Motors encoder). Because a mobile robot moves around, it will frequently encounter unforeseen environmental characteristics, and therefore such sensing is particularly critical. A functional classification of sensors is given first. Then, the selected sensors are described in detail.

III.1. Sensors classification for Mobile Robots

Sensors are classified using two important functional axes: proprioceptive/exteroceptive and

Passive/active [1].

III.1.1. Proprioceptive sensors

They measure values internal to the robot system; for example, motor speed, wheel load, robot arm joint angles, battery voltage.

III.1.2. Exteroceptive sensors

They acquire information from the robot's environment; for example, distance measurements, light intensity, sound amplitude. Hence exteroceptive sensor measurements are interpreted by the robot in order to extract meaningful environmental features.

III.1.3. Passive sensors

They measure ambient environmental energy entering the sensor. Examples of passive sensors include temperature probes, microphones and CMOS camera's.

III.1.4. Active sensors

They emit energy into the environment, and then measure the environmental reaction. Because active sensors can manage more controlled interactions with the environment, they often achieve superior performance. However, active sensing introduces several risks: The outbound energy may affect the very characteristics that the sensor is attempting to measure. Furthermore, an active sensor may suffer from interference between its signal and those beyond its control. For example, signals emitted by other nearby robots, or similar sensors on the same robot, may influence the resulting measurements. Examples of active sensors

include wheel quadrature encoders, the Kinect camera, ultrasonic sensors, and laser rangefinders.

II.2. Basic sensor characteristics

A number of sensor characteristics can be rated quantitatively in a laboratory setting. Such performance ratings will necessarily be best-case scenarios when the sensor is placed on a real world robot, but are nevertheless useful. Hence, some important sensors characteristics are discussed below.

II.2.1. Dynamic range

It is the ratio of the maximum input value to the minimum measurable input value. Because this raw ratio can be unwieldy, it is usually measured in decibels, which are computed as ten times the common logarithm of the dynamic range.

$$dB = \alpha \,.\log\left(\frac{a}{b}\right) \; where \; \alpha = 10 \; or \; 20 \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \; (3.1)$$

However, there is potential confusion in the calculation of decibels, which are meant to measure the ratio between powers, such as watts. Range is also an important rating in mobile robot applications because often robot sensors operate in environments where they are frequently exposed to input values beyond their working range. In such cases, it is critical to understand how the sensor will respond. For example, an optical rangefinder will have a minimum operating range and can thus provide spurious data when measurements are taken with the object closer than that minimum.

II.2.2. Resolution

It is the minimum difference between two values that can be detected by a sensor. Usually, the lower limit of the dynamic range of a sensor is equal to its resolution. However, in the case of digital sensors, this is not necessarily. For example, suppose that you have a sensor that measures voltage, and performs an analog-to-digital (A/D) conversion, and outputs the converted value as an 8-bit number linearly corresponding to between 0 and 5 V. If this sensor is truly linear, then it has 2^8-1 total output values, or a resolution of 5V (255) = 20 mV.

II.2.3. Linearity

It is an important measure governing the behaviour of the sensor's output signal as the input signal varies. A linear response indicates that if two inputs X and Y result in the two outputs $f(X)$ and $f(Y)$, then for any values a and b:

$$f(aX + bY) = af(X) + bf(Y) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \; (3.2)$$

This means that a plot of the sensor's input/output response is simply a straight line.

II.2.3. Sensitivity

It is a measure of the degree to which an incremental change in the target input signal changes the output signal. Formally, sensitivity is the ratio of output change to input change. Unfortunately, the sensitivity of exteroceptive sensors is often confounded by undesirable sensitivity and performance coupling to another environmental parameter.

III.3. The main RobuTER/Ulm Integrated sensors

III.3.1. The Kinect Xbox360 V2 sensor

The innovative technology behind Kinect is a combination of hardware and software contained within the Kinect sensor (see figure 3.1). It is a flat black box that sits on a small platform and inside the sensor case contains:

- An RGB camera (Color sensor) that stores three channel data in a 1280x960 resolution. This makes capturing a color image possible.
- An infrared (IR) emitter and an IR depth sensor. The emitter emits infrared light beams and the depth sensor reads the IR beams reflected back to the sensor. The reflected beams are converted into depth information measuring the distance between an object and the sensor. This makes capturing a depth image possible.
- A multi-array microphone, which contains four microphones for capturing sound. Because there are four microphones, it is possible to record audio as well as find the location of the sound source and the direction of the audio wave.
- A 3-axis accelerometer configured for a 2G range, where G is the acceleration due to gravity. It is possible to use the accelerometer to determine the current orientation of the Kinect.

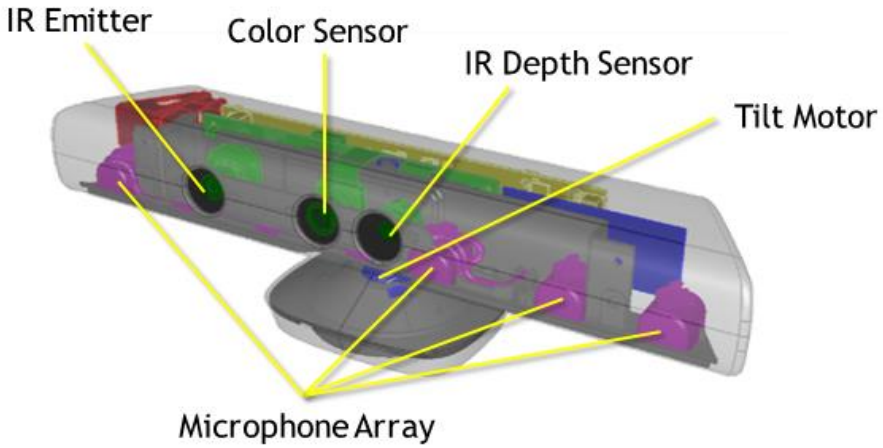


Figure 3.1. The Kinect sensor.

The main feature is that the sensor is made by differential pixels, meaning that each pixel is split in two accumulators and a clock regulates which one of the pixel side is the one

currently active. This permits creating a series of different output images (depth images, grey scale images dependent from ambient lighting and grey scale images independent from ambient lighting). The system measures the phase shift of the modulated signal and computes the depth from phase using Eq. 3.3:

$$2d = \frac{phase}{2\pi} \frac{c}{fmod} \quad \text{..... (3.3)}$$

where *d* is the depth measure, *c* is the speed of light, f_{mod} is the modulation frequency. Table 3.1 gives some specifications of the Kinect sensor.

Table 3.1. Kinect Sensor specifications

Sensor Specifications	**Kinect 1.0**
RGB camera (pixel)	1280 × 1024 or 640 × 480
Depth camera (pixel)	640 × 480
Max depth distance (m)	4.0
Min depth distance (m)	0.8
Tilt motor	Yes
USB	2.0

III.3.1.1. Depth measurement model

Operation of Kinect depth sensor is grounded on structured light analysis approach. The sensor incorporates a laser IR diode for emitting a dotted light pattern and an IR camera for capturing reflected patterns. By using a suitable window size, the sensor compares reflected patterns to reference patterns, obtained for a plane placed at a known distance from the sensor, and uses the position of the best match pattern to infer disparity of reflected bean and further calculate the depth of reflection surface. A supplementary RGB camera is added to provide additional information on the color and the texture of the surface. The relationship between depth of reflection surface and the disparity between images of light beans obtained for a reference and measurement (object) surface may be derived in the following manner (the derivation closely follows Khoshelham and Elberink [7]). Looking at Figure 3.2, where the reference bean is assumed to pass the path P- R-C and the measurement bean passes the path P- M-C ,from similarity of triangles MR*C and M'R'C we obtain:

$$\frac{d}{D} = \frac{f}{Z} \text{..... (3.4)}$$

Where Z is the distance of the measurement (Object) plane from the sensor, d = $\overline{R'M'}$ denotes the disparity between images of reference R' and measurement M' beans, and f is the focal length of IR camera. From similarity of triangles CPR and R''MR , another relation is obtained :

$$\frac{D}{b} = \frac{Z_0 - Z}{Z_0} \quad \text{..... (3.5)}$$

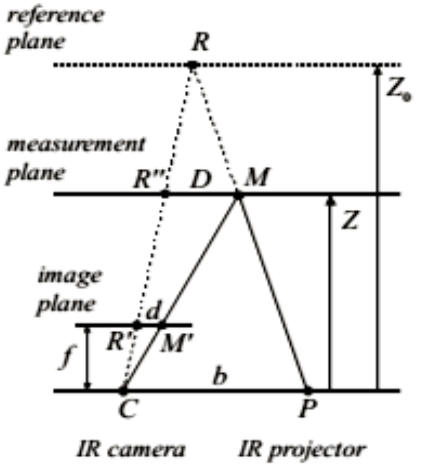


Figure 3.2. Depth measurement geometry.

where Z is the distance of the measurement (object) plane from the sensor, $d = \overline{R\,M}$ denotes the disparity between images of reference R' and measurement M' beans, and f is the focal length of IR camera.

III.3.1.2. Sensor calibration

Kinect-type 3D sensors, considered in this work, operates as structured light sensors. A sensor (Figure 3.1) incorporates a laser infra-red (IR) diode for emitting a dotted light pattern and an IR camera for capturing reflected patterns. Depth is calculated by sensor software on the basis of disparity of reflected patterns with the respect to the reference patterns obtained for a plane placed at a known distance from the sensor. A supplementary RGB camera is added to provide additional information on colour and texture of the surface. Thus, sensor output consists of flowing three data: images from RGB camera, raw images from IR camera and depth maps calculated by sensor firmware. Sensor calibration can be viewed as a refinement of correspondences between 3D object coordinates and coordinates in RGB, IR and depth images [7].

The proposed calibration procedure consists of two steps:

- The first step comprises calibration of sensor's RGB/IR cameras.
- The second is the performance of depth model calibration.

Although real-time depth information is provided by IR camera, the depth map tells how far the IR camera's pixels are and we actually do not know the depth information of the color image because the two cameras have different characteristics. As it is shown in the image below (figure 3.3) the pixels do not match in the two images. The locations of the hand and the arm are completely different in the two images.

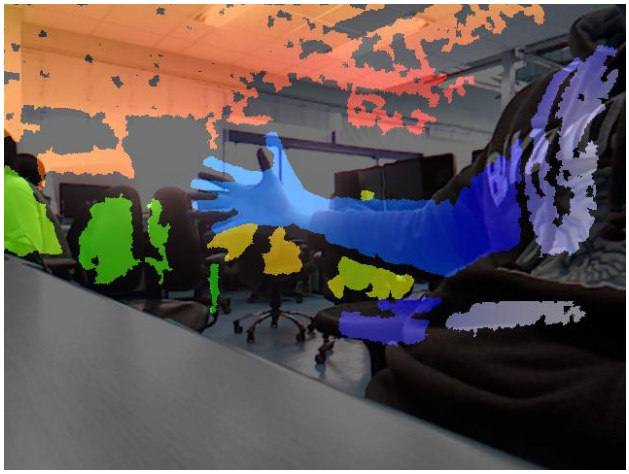


Figure 3.3. IR/RGB camera.

If it would be used for 3D scene capture or it is wanted to relate the RGB and the depth images, it is needed to match the color image's pixels to the depth image's. Thus, the calibration is needed to be performed.

Kinect camera calibration is not different from the general camera calibration. It is just needed to capture images of a chessboard pattern from IR and RGB cameras. Several images of a chessboard pattern are needed to be captured. When capturing images from the IR camera, the emitter with something for good corner detection in chessboard images must be blocked. If not, the captured images will look like below and corner detection will fail.

If the lightings in our environment do not have enough IR rays, a light source that emits IR rays is needed. It might be good to capture the same scenes with two cameras. The images below show two images captured from the IR and RGB cameras, respectively (see Figure 3.4).

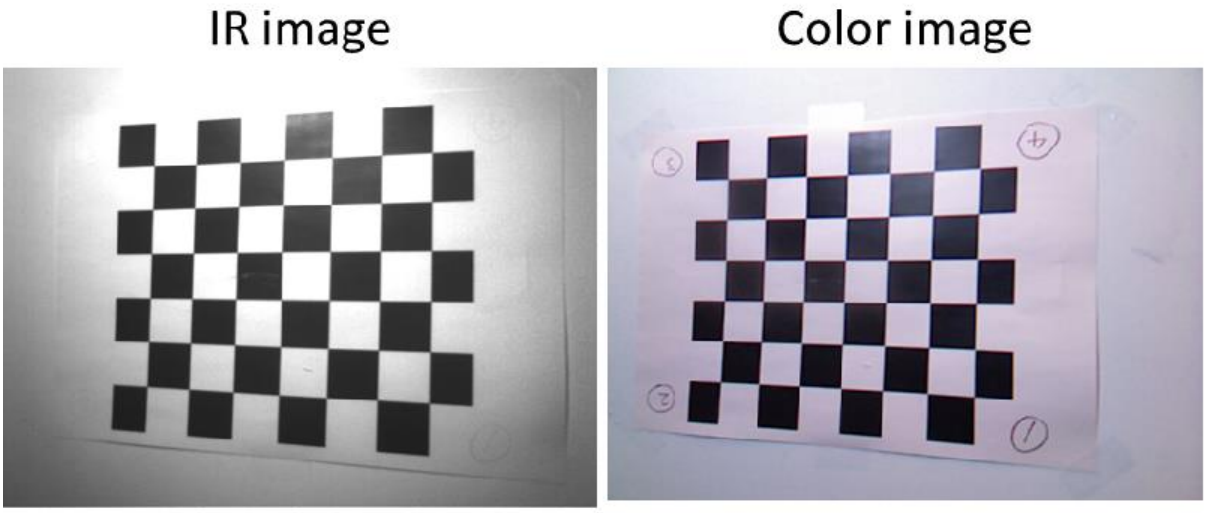


Figure 3.4. The difference between IR image and color image.

Once images are taken, calibration can be performed for each camera by using OpenCV API GML calibration toolbox. After calibration, the intrinsic camera matrices, K_ir and K_rgb, and distortion parameters of the two cameras are obtained (see figure 3.5).

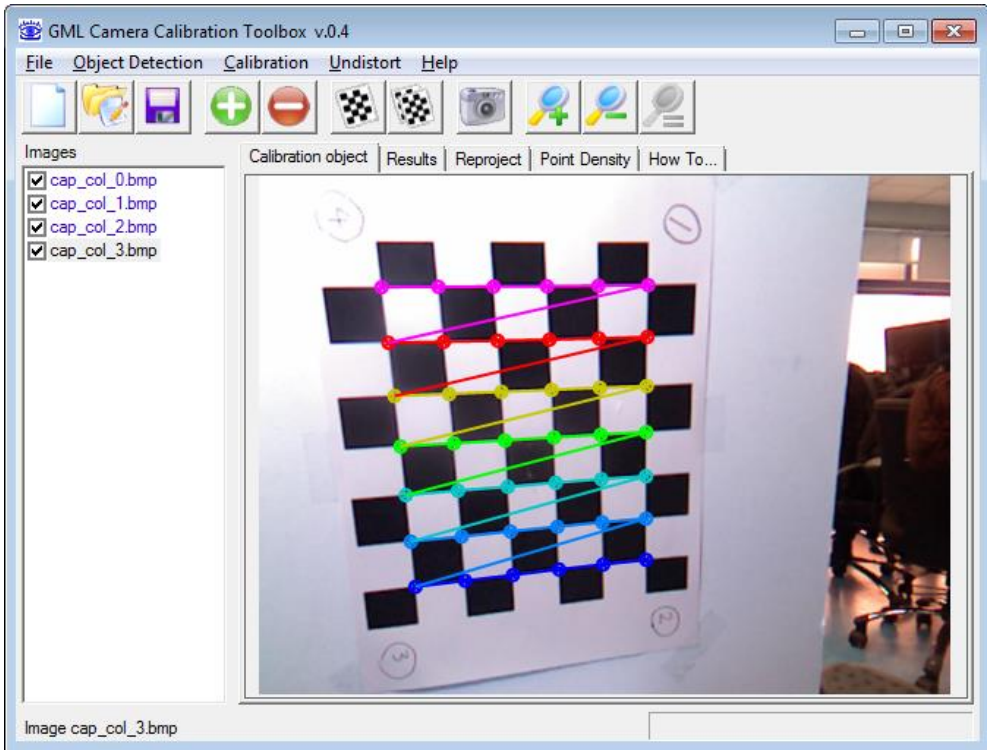


Figure 3.5. Using OpenCV API in GML Camera calibration toolbox.

To achieve our goal, a more information is needed, the geometrical relationship between the two cameras expressed as a rotation matrix R and a translation vector t. To compute them, capture the same scene containing the chessboard pattern with the two cameras and compute extrinsic parameters. From two extrinsic parameters, the relative transformation can be computed easily.

Now, the depth of the colour image can be computed from the depth image provided by the IR camera. Let's consider a pixel p_ir in the IR image. The 3D point P_ir corresponding to the p_ir can be computed by back-projecting p_ir in the IR camera's coordinate system.

$$P_{ir} = inv(K) \times p\_ir \quad \text{..... (3.6)}$$

P_ir can be transformed to the RGB camera's coordinate system through relative transformation R and t.

$$P\_rgb = R \times P\_ir + t \quad \text{..... (3.7)}$$

Then, P_rgb is projected onto the RGB camera image and we obtain a 2D point p_rgb.

$$p_{rgb} = K_{rgb} \times P\_rgb \quad \text{..... (3.8)}$$

Finally, the depth value corresponding to the location p_rgb in RGB image is P_rgb's Z axis value.

depth of p_rgb = Z axis value of P_rgb ……..... (3.9)

P_ir : 3D point in the IR camera's coordinate system.

R, t : Relative transformation between two cameras.

P_rgb : 3D point in the RGB camera's coordinate system.

p_rgb : The projection of P_rgb onto the RGB image.

In the above, conversion to homogeneous coordinates are omitted. When two or more 3D points are projected to the same 2D location in the RGB image, the closest one is chosen. The colour values of the depth map pixels can also be computed in the same way. p_ir's colour corresponds to the colour of p_rgb.

Figure 3.6 illustrate the resulting depth image of the RGB camera. Since the RGB camera sees wider region than the IR camera, not all pixels' depth information are available.

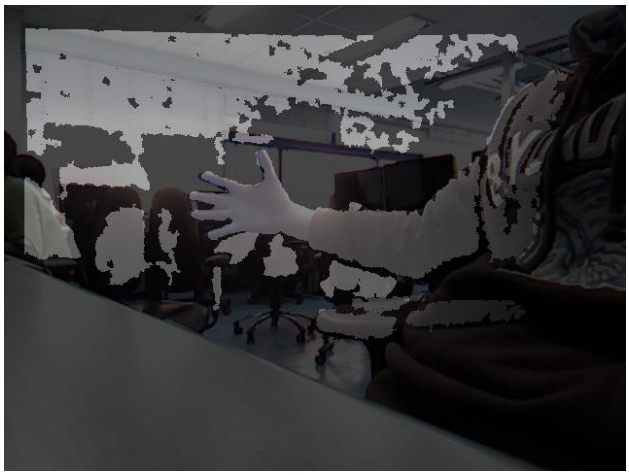


Figure 3.6. Depth Image of the RGB camera.

In Figure 3.7 bellow we can see that the two match well, while they do not before calibration as shown at the beginning of this test.

Figure 3.7. The calibration shows that the pixels match well.

III.3.2. The Ultrasonic Sensor in RobuTER

Unfortunately, the sensing devices which are available for mobile robots often fail in a variety of circumstances. This is especially true for the less expensive devices such as ultrasonic and infrared range sensors. Combining data from several sensors and from a pre-stored model of the domain provides a way to enhance the reliability of a perception system. Such combination may be accomplished by integrating range measurements into a geometric model of the local environment.

The robuTER the case of our studies has an ultrasonic belt. Next, the position (x, y, z, theta) of each sensor is proposed, where the first is the one that is in the left of the belt before starting the direction of robuter. For information, the robuTER is equipped with a ring of 24 ultrasonic range sensors as shown in figure 3.8.

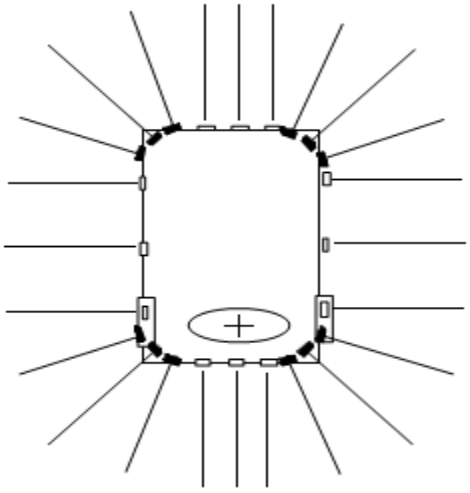


Figure 3.8. Configuration of 24 Ultrasonic Range sensors [7].

The table 3.2 shows the orientation of the landmark; while, the table 3.3 shows the position information of each sensor.

Table 3.2. The orientation of direct landmark.

Axe	Direction	Sense
X	In the axis of robuTER	Forward
Y	Vertical	To the top
Z	In the axis of the drive wheels	To the right
Theta	0 ° along the x axis	Trigonometric

Dimensions are in millimetres and degrees by taking the middle as the origin of the axis of the drive wheels.

Table 3.3. The position information of each sensor.

Sensors	X	Y	Z	Theta
1	285.96	428.50	-278.50	90
2	535.98	428.50	-265.10	60
3	572.60	428.50	-228.48	30
4	586.00	428.50	-178.46	0
5	586.00	428.50	-90.00	0
6	586.00	428.50	-30.00	0
7	586.00	428.50	30.00	0
8	586.00	428.50	60.00	0
9	586.00	428.50	178.46	0
10	572.60	428.50	228.48	330
11	535.98	428.50	265.10	300
12	485.96	428.50	278.50	270
13	-39.96	428.50	278.50	270
14	-89.98	428.50	265.10	240
15	-126.60	428.50	228.48	210
16	-140.00	428.50	178.46	180
17	-140.00	428.50	90.00	180
18	-140.00	428.50	30.00	180
19	-140.00	428.50	-30.00	180
20	-140.00	428.50	-90.00	180
21	-140.00	428.50	-178.46	180
22	-126.60	428.50	-228.48	150
23	-89.98	428.50	-265.10	120
24	-39.96	428.50	-278.50	90

The position and orientation of the sensors with respect to the origin of the robot are defined in a sensor configuration parameter (figure 3.9). So for each sensor, the sensor configuration parameter gives:

- r: The distance from the robot's origin to the sensor.
- γ: The angle from the robot's axis to the sensor
- β: The orientation of the sensor with respect to the robot's axis.

The sensor description algorithm can be made to work with a variety of sensor configurations.

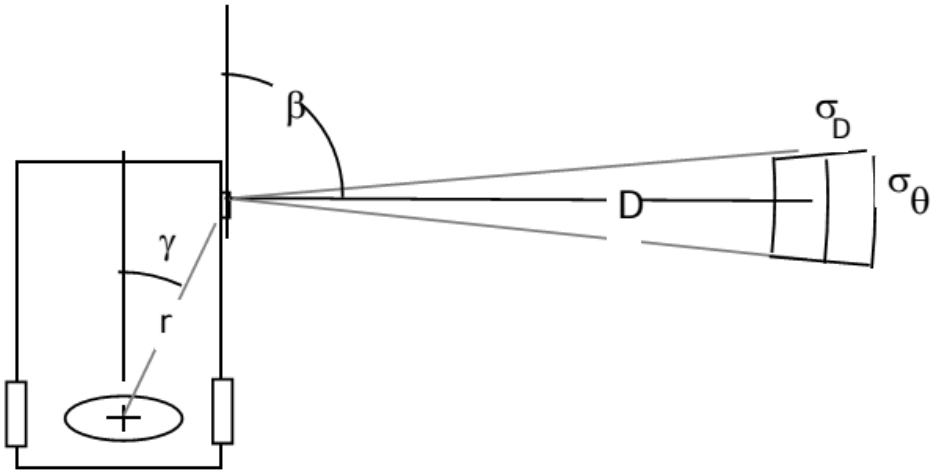


Figure 3.9. Projection of a Range Reading to External Coordinates [7].

A sensor data description process reads range measurements from the sonar table, as well as the estimated position of the robot from the vehicle controller. With this information, the depth measure, d, for each sensor, s, is projected to external coordinates, (Xs, Ys), using the estimated position of the robot, (X, Y, α) , as shown in figure 3.10.

$$Xs = X + r\ Cos(\gamma + \alpha\ ) + d\ Cos(\beta + \alpha)$$

$$Ys = Y + r\ Sin(\gamma + \alpha\ ) + d\ Sin(\beta\ + \alpha)$$

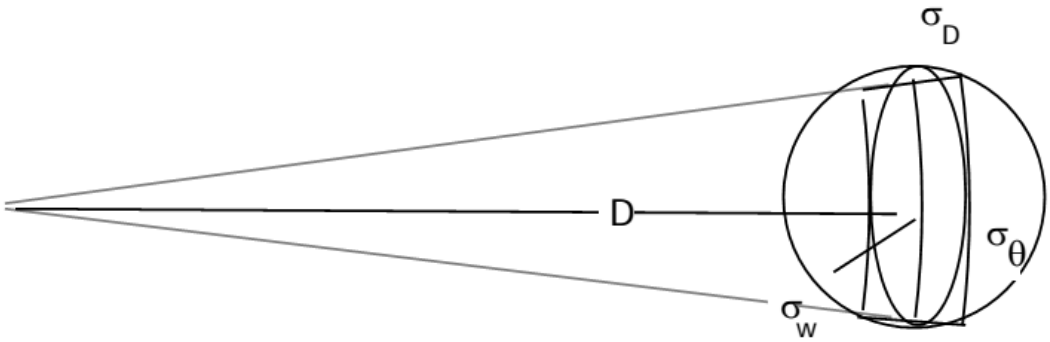


Fig 3.10. Model of the Ultrasonic Range Sensor and its Uncertainties.

In order to combine data from different viewpoints and sensors, the inherent precision of the data must be estimated. A model of an ultrasonic range sensor has been developed, which predicts that an echo comes from an arc shaped region as illustrated in figure 3.10. This region is determined by the composition of an arc blurred in a perpendicular direction. The length of the arc is given by the uncertainty in orientation, σW, while the perpendicular blurring is caused by an uncertainty in depth σD.

III.3.3. The Incremental Encoder in RobuTER

Optical incremental encoders have become the most popular device for measuring angular speed and position within a motor drive or at the shaft of a wheel or steering mechanism. In mobile robotics, encoders are used to control the position or speed of wheels and other motor-driven joints. Because these sensors are proprioceptive, their estimate of position is best in the reference frame of the robot and, when applied to the problem of robot localization, significant corrections are required.

An optical encoder is basically a mechanical light chopper that produces a certain number of sine or square wave pulses for each shaft revolution. It consists of an illumination source, a fixed grating that masks the light, a rotor disc with a fine optical grid that rotates with the shaft, and fixed optical detectors. As the rotor moves, the amount of light striking the optical detectors varies based on the alignment of the fixed and moving gratings. In robotics, the resulting sine wave is transformed into a discrete square wave using a threshold to choose between light and dark states. Resolution is measured in Cycles Per Revolution (CPR). The minimum angular resolution can be readily computed from an encoder's CPR rating. A typical encoder in mobile robotics may have 2000 CPR, while the optical encoder industry can readily manufacture encoders with 10000 CPR. In terms of required bandwidth, it is of course critical that the encoder be sufficiently fast to count at the shaft spin speeds that are expected.

Industrial optical encoders present no bandwidth limitation to mobile robot applications. Usually in mobile robotics the quadrature encoder is used. In this case, a second illumination and detector pair is placed 90 degrees shifted with respect to the original in terms of the rotor disc. The resulting twin square waves, shown in figure 3.11, provide significantly more information. The ordering of which square wave produces a rising edge first identifies the direction of rotation. Furthermore, the four detectably different states improve the resolution by a factor of four with no change to the rotor disc. Thus, a 2000 CPR encoder in quadrature yields 8000 counts. Further improvement is possible by retaining the sinusoidal wave measured by the optical detectors and performing sophisticated interpolation.

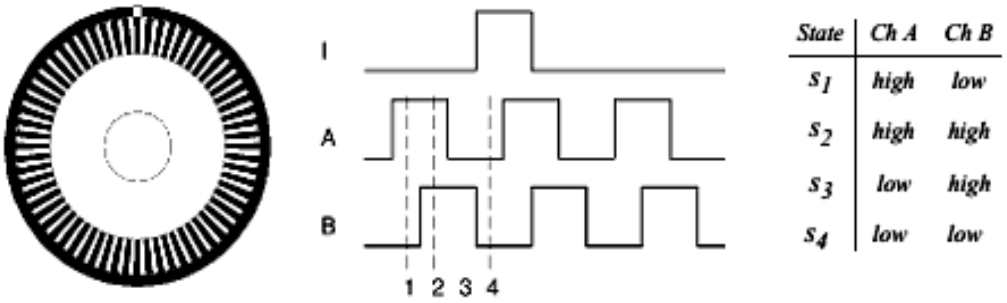


Figure 3.11. Quadrature optical wheel encoder.

In figure 3.11, the observed phase relationship between channel A and B pulse trains are used to determine the direction of the rotation. A single slot in the outer track generates a reference (index) pulse per revolution. The characteristics of the incremental encoder of robuTER are summarized in Table 3.4.

Table 3.4. Incremental Encoder line driver for RobuTER.

Number of points	500
Mass	0,085 kg

Such methods, although rare in mobile robotics, can yield 1000-fold improvements in resolution. As with most proprioceptive sensors, encoders are generally in the controlled environment of a mobile robot's internal structure, and so systematic error and cross-sensitivity can be engineered away. The accuracy of optical encoders is often assumed to be 100% and, although this may not be entirely correct, any errors at the level of an optical encoder are dwarfed by errors downstream of the motor shaft.

III.4. Conclusion

This chapter provides a deep explanation of the different sensors characteristics of the RobuTER. Where, the different classifications of those sensors are given, which can be proprioceptive/exteroceptive and Passive/active. Then, their basic characteristics to the change of environment are stepped over. Finally, the chapter ends up by a deep description of the different sensors integrated in RobuTER from the Kinect to the Ultrasonic sensor and the Incremental Encoder.

IV. Introduction

Simulation is a flexible methodology which can be used to analyze the behavior of a present or proposed scenarios, and by **performing simulation** and **analyzing the results**, an understanding of how a system operates can be gained even if it was a complicated one. This chapter will focus primarily in creating the necessary files that construct our simulation results in simulation and insure making the navigation tasks in a correct manner. The chapter will end up by a step by step demonstration of our simulation results and of the path planning execution in ROS.

IV.1. Robot 3D model

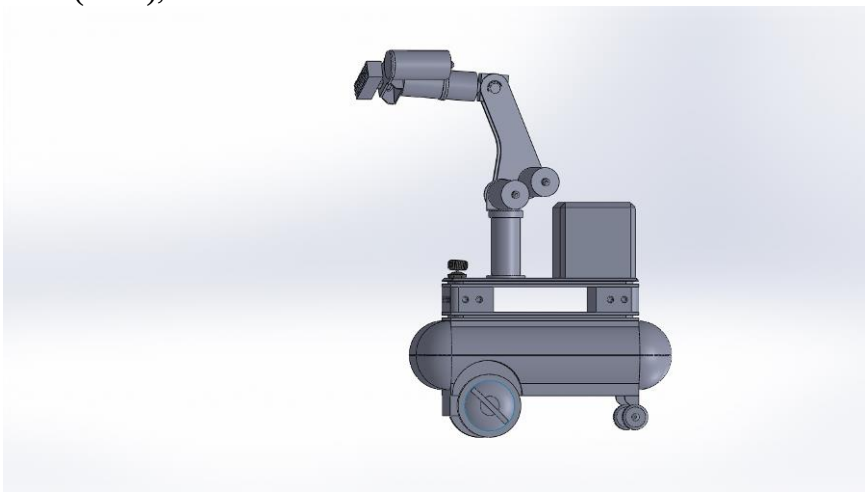
The first phase of robot manufacturing is its design and modeling. The robot can be designed and modeled using CAD tools such as AutoCAD, Solid Works, Blender, and so on. One of the main purposes of modeling robot is simulation.

The virtual robot model must have all the characteristics of real hardware, the shape of robot may or may not look like the actual robot but it must be an abstract, which has all the physical characteristics of the actual robot.

SolidWorks

SolidWorks is modern computer aided design (CAD) software. It enables designers to create a mathematically correct solid model of an object that can be stored in a database. When the mathematical model of a part or assembly is associated with the properties of the materials used, we get a solid model that can be used to simulate and predict the behavior of the part or model with finite element and other simulation software. The same solid model can be used to manufacture the object and also contains the information necessary to inspect and assemble the product. SolidWorks and similar CAD programs have made possible concurrent engineering, where all the groups that contribute to the product development process can share real-time information.

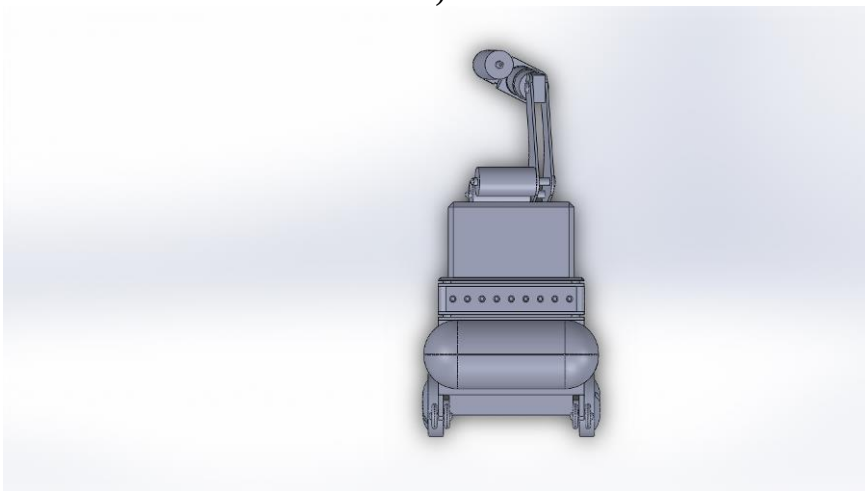
In this part, a 3D model for our RobuTER/ULM will be built; then, the model to an URDF (Unified Robot Description Format), the one that is suitable to be processed in robot operating system (ROS), will be extracted.



a)

b)

c)



d)

Figure 4.1. 360° View of RobuTER/ULM. a) Left view. b) Top view. c) Front view. d) Rear view.

IV.2. Creating a 3D virtual environment in Gazebo

After creating the robot model in solidworks and extract it to an URDF file. We need now to build a 3D virtual environment where we simulate and test the robot navigation tasks.

First, we need to launch gazebo in an empty word by writing the following command line

```
roslaunch gazebo_ros empty_world.launch
```

Then, we need to Build the virtual map by clicking on -- Edit—Building Editor. After that, we sketch the walls to create the virtual map as shown in Figure 4.2.

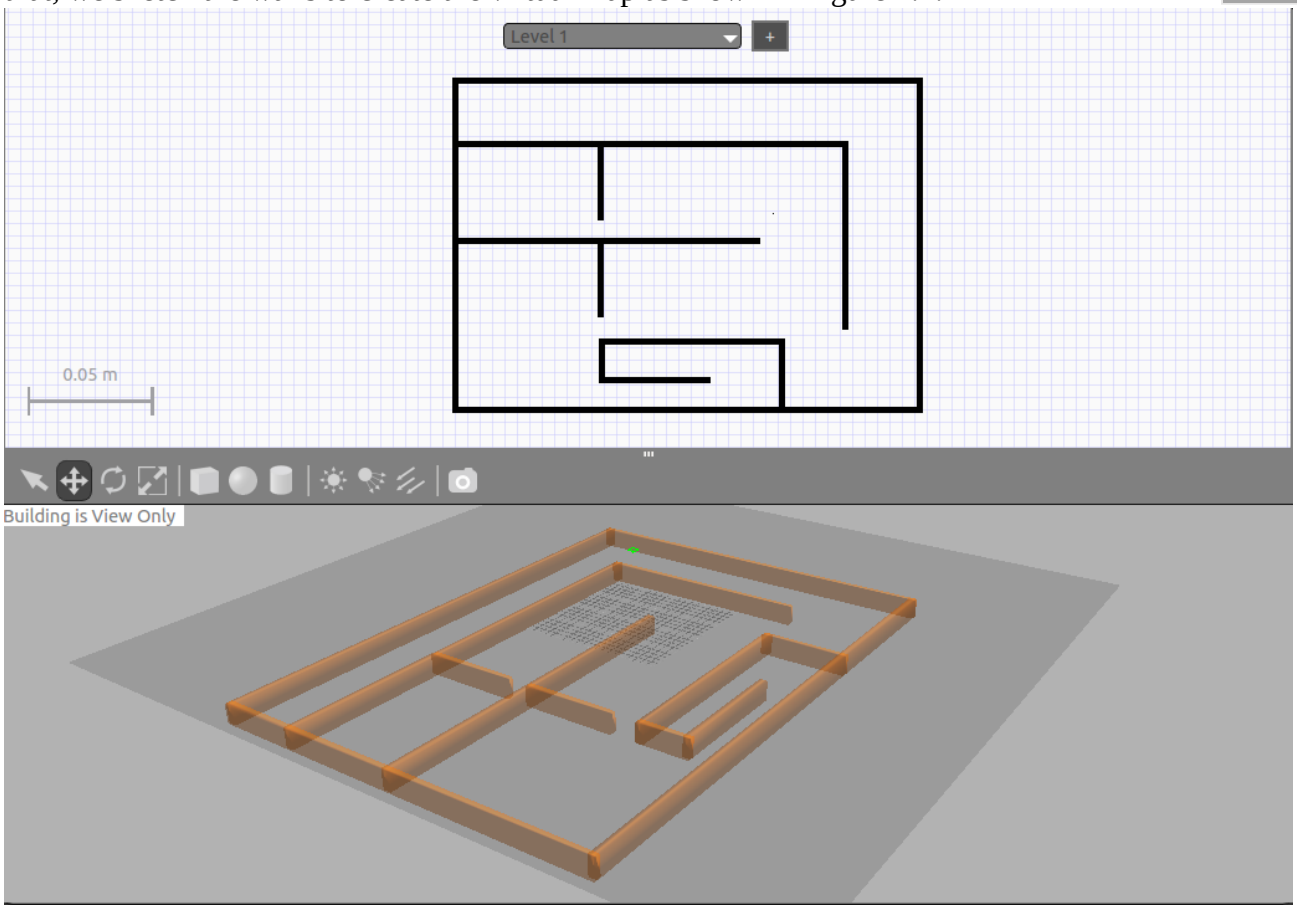


Figure 4.2. Creating a Virtual environment where map exploring will be performed.

After creating the environment, we save the file as a ".SDF" file to use it in other processes and simulators like rviz (ROS visualization check appendix A.3).

IV.3. Loading the 3D robot to the virtual environment in Gazebo

Now, we will test our robot by integrating it in the map and locate it in a position according to the global reference frame. We will execute the following command line to run the robot_world.launch file from the created robot_gazebo package which contains the created map and the robot as shown in figure 4.3.

```
roslaunch robot_gazebo robot_world.launch
```

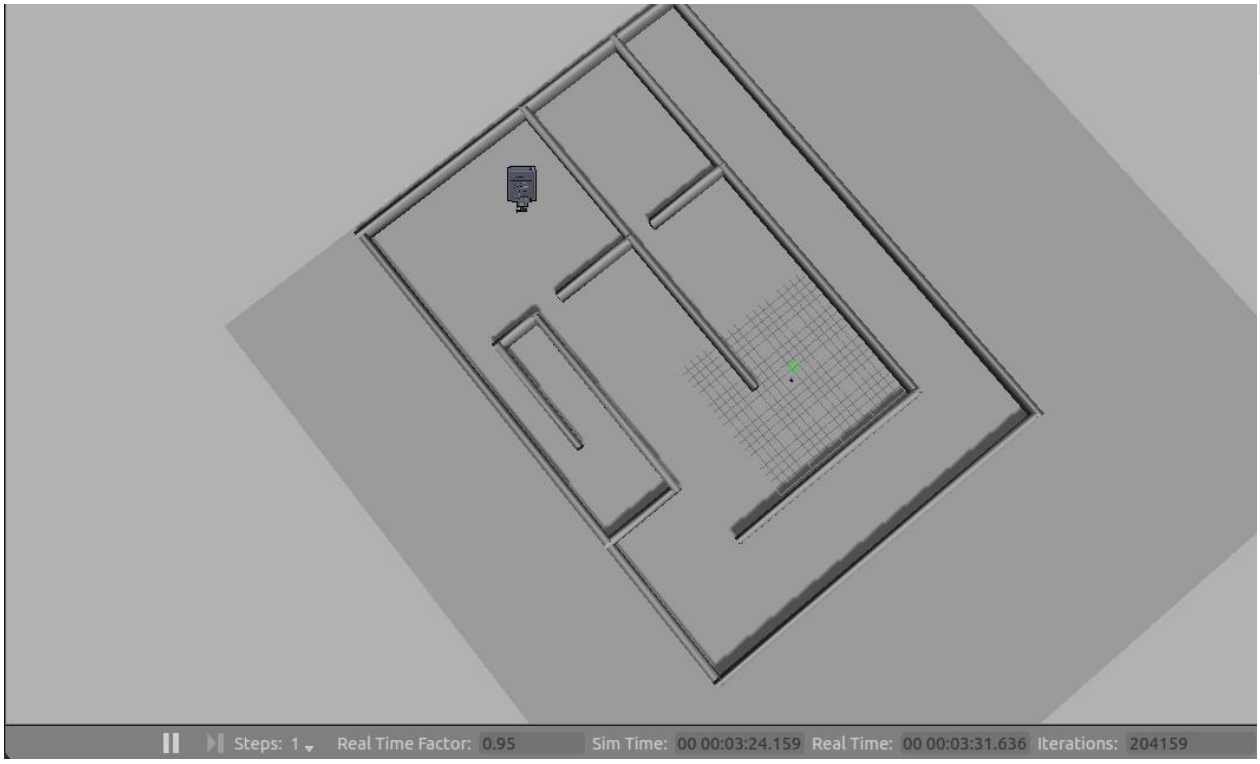


Figure 4.3. Inserting the virtual robot to the created map and test it in Gazebo.

IV.4. Navigation stack

In the previous sections we have seen how to create our robot and mount it through the virtual world in gazebo simulator.

In this section, we will learn something that is probably one of the most powerful features in ROS, something that will let us move our robot autonomously.

ROS has many algorithms that can be used for navigation. First of all, we will learn the necessary ways to configure the navigation stack with our robot model. Then, we will learn how to configure and launch the navigation stack on the simulated robot; by inserting goals and configuring some parameters to get the best results. In particular, we will cover the following items in the first part [11]:

- Introduction to the navigation stacks and their powerful capabilities. Clearly one of the greatest pieces of software that comes with ROS.
- The TF (Transform Frames) is explained in order to show how to transform from the frame of one physical element to another; for example, the data received using a sensor or the command for the desired position of an actuator.
- Create a laser driver and Kinect laser scan (see figure 4.4).
- Explain how the odometry is published, and how integrate it in Rviz.
- A base controller will be presented, including a description of how to create one for our robot.

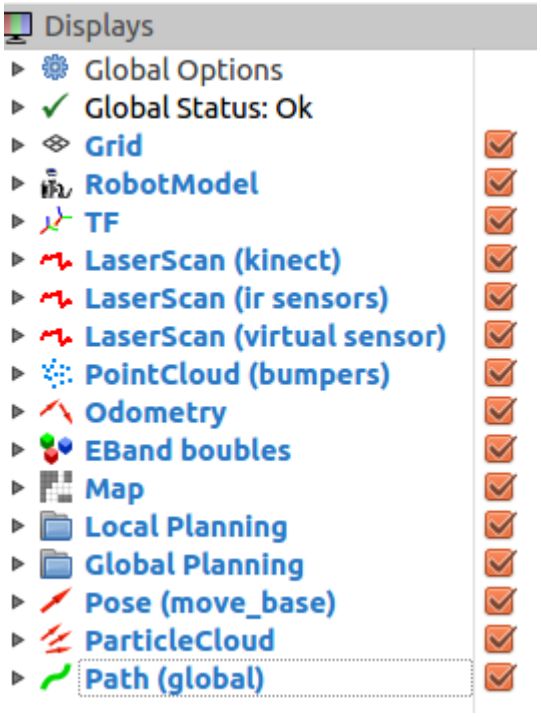


Figure 4.4. Display navigation stack in Rvis.

Figure 4.5 shows how the navigation stacks are linked together to perform the process of map building then path planning and navigation is presented in an organized way.

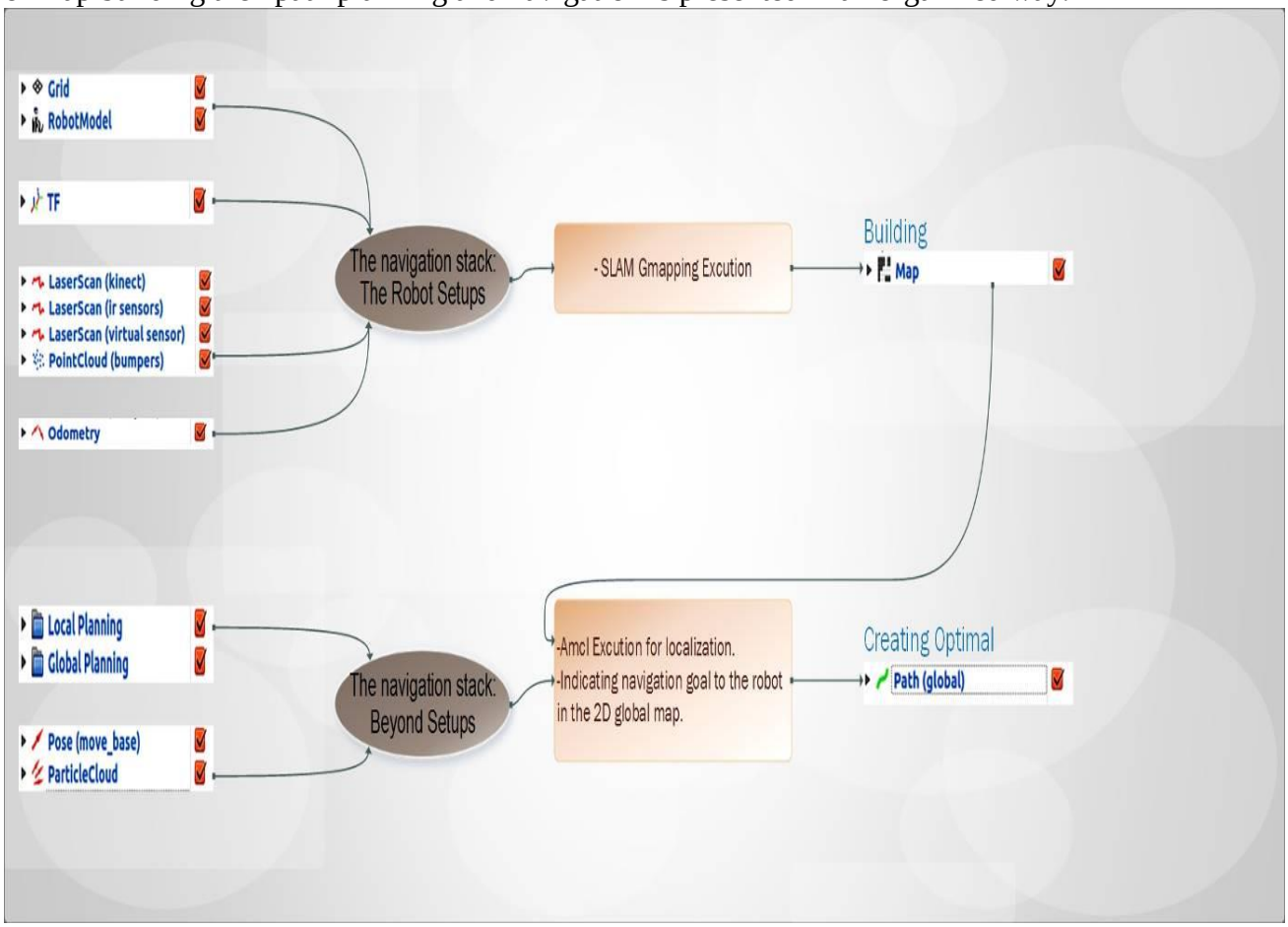


Figure 4.5. The relationship between the navigation stack parts.

IV.4.1. Navigation stack – Robot Setups

In order to understand the navigation stack, we should think of it as a set of algorithms that use the sensors of the robot and the odometry, and the robot can be controlled using a standard message. It can move the robot without problems (for example, without crashing or getting stuck in some location, or getting lost) to another position. We should assume that this stack can be easily used with any robot. This is almost true, but it is necessary to tune some configuration files and write some nodes to use the stack. The robot must satisfy some requirements before it uses the navigation stack [11]:

- The navigation stack can only handle a differential drive and holonomic wheeled robots. The shape of the robot must be either a square or a rectangle. However, it can also do certain things with biped robots, such as robot localization, as long as the robot does not move sideways.

- It requires that the robot publishes information about the relationships between all the joints and sensors' position.

- The robot must send messages with linear and angular velocities.

- A planar laser must be on the robot to create the map for localization.

The navigation stack assumes that the robot is configured in a particular manner in order to run. Figure 4.6 shows an overview of this configuration. The white components are the required ones and are already implemented, the gray components are optional components and are also already implemented, and the blue components must be created for each robot platform. The pre requisites of the navigation stack, along with instructions on how to fulfil each requirement, are provided in the sections below.

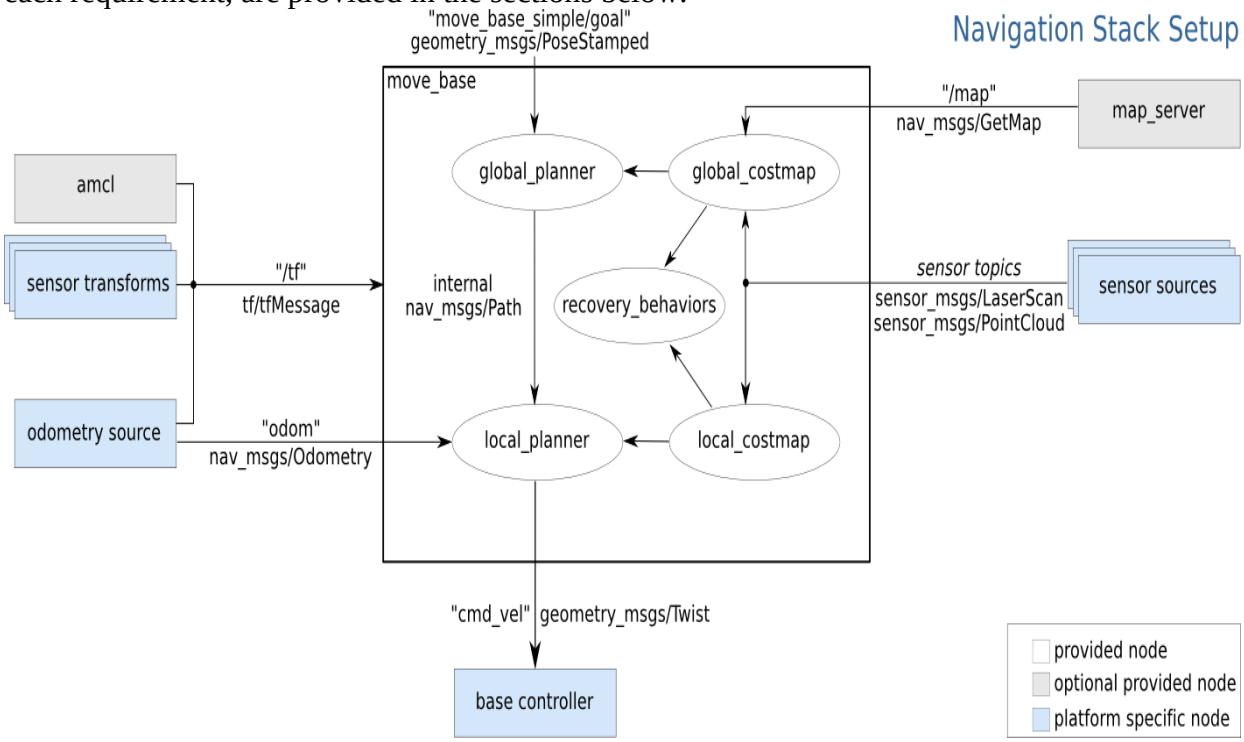


Figure 4.6. Navigation stack setup [11].

IV.4.1.1. Transform Configuration TF

The navigation stack needs to know the position of sensors, wheels; and joints. To do that, we use the TF (which stands for Transform Frames) software library. It manages a transform tree. We could do this with mathematics, but if we have a lot of frames to calculate, it will be a bit complicated and messy. Thanks to TF, we can add more sensors and parts to the robot, and the TF will handle all the relations for us.

At this point, let we assume that we have some data from the laser in the form of distances from the laser's center point. In other words, we have some data in the base_laser coordinate frame. Now suppose we want to take this data and use it to help the mobile base avoid obstacles in the world. To do this successfully, we need a way of transforming the laser scan we have received from the base_laser frame to the base_link frame. In essence, we need to define a relationship between the base_laser and base_link coordinate frames (see figure 4.7).

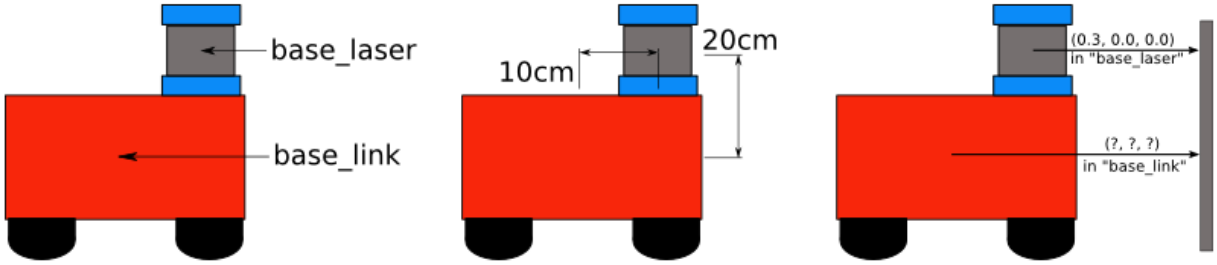


Figure 4.7. Demonstration for the base_laser and base_link position.

If we put the laser 10-cm backwards and 20-cm above with regard to the origin of the coordinates of the base_link, we would need to add a new frame to the transformation tree with these offsets. Once inserted and created, we could easily know the position of the laser with regard to the base_link value or the wheels. The only thing we need to do is call the TF library and get the transformation. Now, we have to take the transform tree and create it with code.

First of all, we need to create a new package in our workspace name it robot_Nav_stack1 and once we have get our package, we need to create the nodes that will do the work of broadcaster and a listener.

Creating a broadcaster

First, we create a robot_Nav_stack1/src/tf_broadcaster.cpp file as described in the following algorithm:

Where first, we call the tow libraries “ros.h” and “transform_broadcaster.h” to run ROS nodes and integrate some broadcaster transformation parameters, then we set the TF broadcaster parameters position.

```
        Call header file ros.h
        Call header file transform_broadcaster.h
        Set variables integer argc, character argv
        initialize roscore with (argc, argv, "robot_tf_publisher")
        initialize ros NodeHandle as n;
        initialize ros Rate as r (100);
        initialize tf TransformBroadcaster broadcaster;

start loop
while(n,r()){
        use functions
        broadcaster.sendTransform(tf_StampedTransform(tf_Transform
        (set tf_Quaternion to (0, 0, 0, 1), set tf_Vector to (0.1, 0.0, 0.2)),"base_link", "base_laser"));
        r.sleep()
        }
```

After that we need to create another node that will use the transform, and it will give us the position of a point from the sensor with regard to the center of base_link (our robot).

Creating a listener

Now, we are going to write a node that will use that transform to take a point in the base_laser frame and transform it to a point in the base_link frame. Once again, open an editor and write the code into the robot_Nav_stack1/src/tf_listener.cpp file described by the following algorithm:

Where we call the necessary libraries at first, to use functions to link the nodes with a specific messages described with initial laser position than collect information about the position transformation from the base laser to the base link.

Call header file ros.h
Call header file PointStamped.h
Call header file transform_listener.h

//we will create a point in the base_laser frame that we would like to transform to the base_link frame:
laser_point.header.frame_id = "base_laser"

//we will just use the most recent transform available for our simple example:
laser_point.header.stamp = ros_Time(

//just an arbitrary point in space
Fix position laser_point.point.x to 1.0
Fix position laser_point.point.y to 0.2
Fix position laser_point.point.z to 0.0

Collect information about Tf position "base_laser: (%.2f, %.2f. %.2f) -----> base_link: (%.2f, %.2f, %.2f) at time “%.2f".

If Receive an excepted error trying to transform a point from "base_laser" to "base_link" type “%s”.

Now that we have written our nodes, we need to build them. Open up the CMakeLists.txt file and add the following lines to the bottom of the file.

```
add_executable(tf_broadcaster src/tf_broadcaster.cpp)
add_executable(tf_listener src/tf_listener.cpp)
target_link_libraries(tf_broadcaster ${catkin_LIBRARIES})
target_link_libraries(tf_listener ${catkin_LIBRARIES})
```

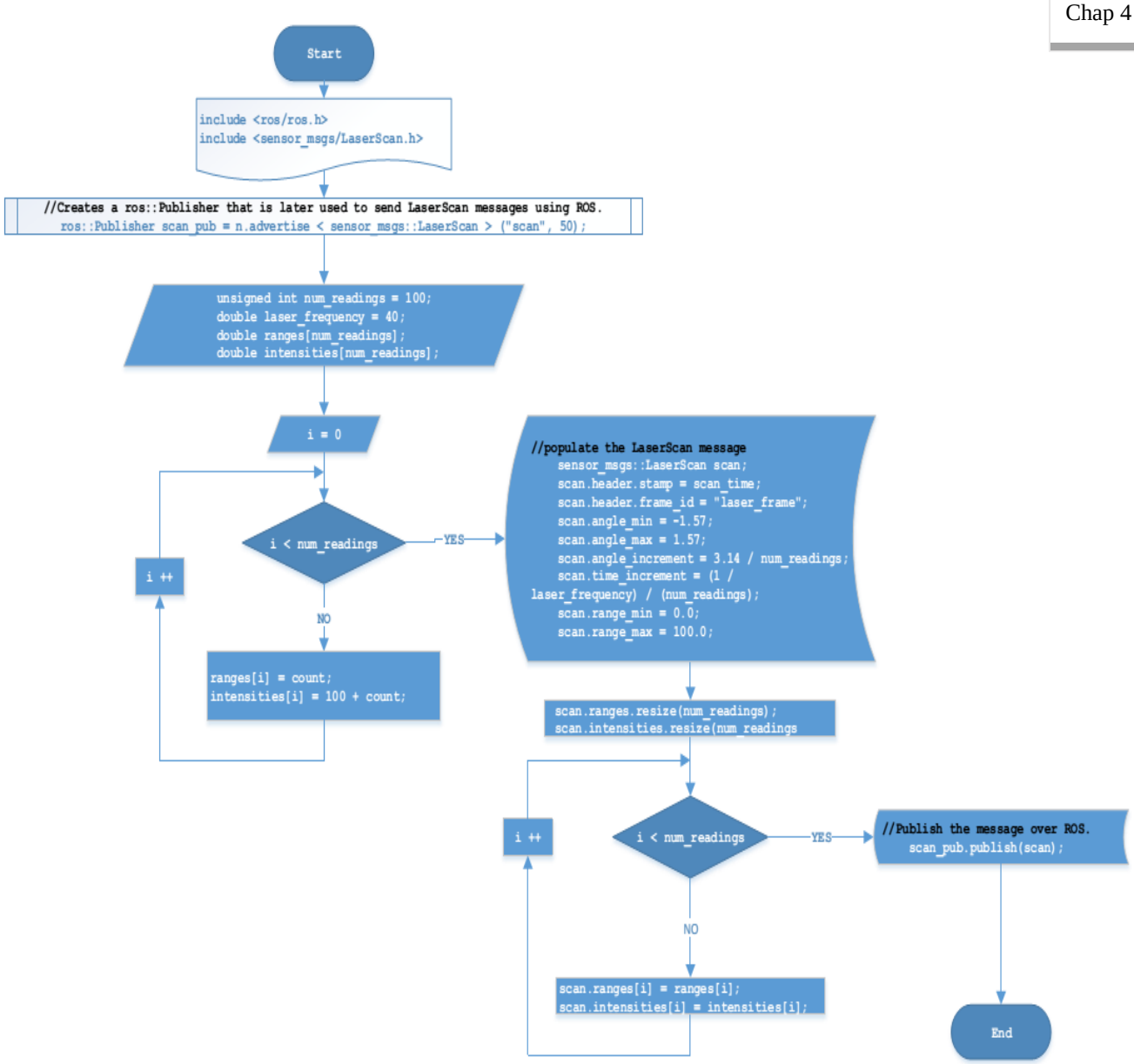
Next, we save the file and build the package using Catkin_Make command line.

IV.4.1.2. Sensor Information

Our robot can have a lot of sensors to see perceive the world. In our case, we use the Kinect V2 as an IR laser scan to provide information; we can program a lot of nodes to take this data and do something, but the navigation stack is prepared only to use the planar laser's sensor. So, the sensor must publish either sensor_msgs/LaserScan or sensor_msgs/PointCloud messages over ROS.

Publishing LaserScans over ROS

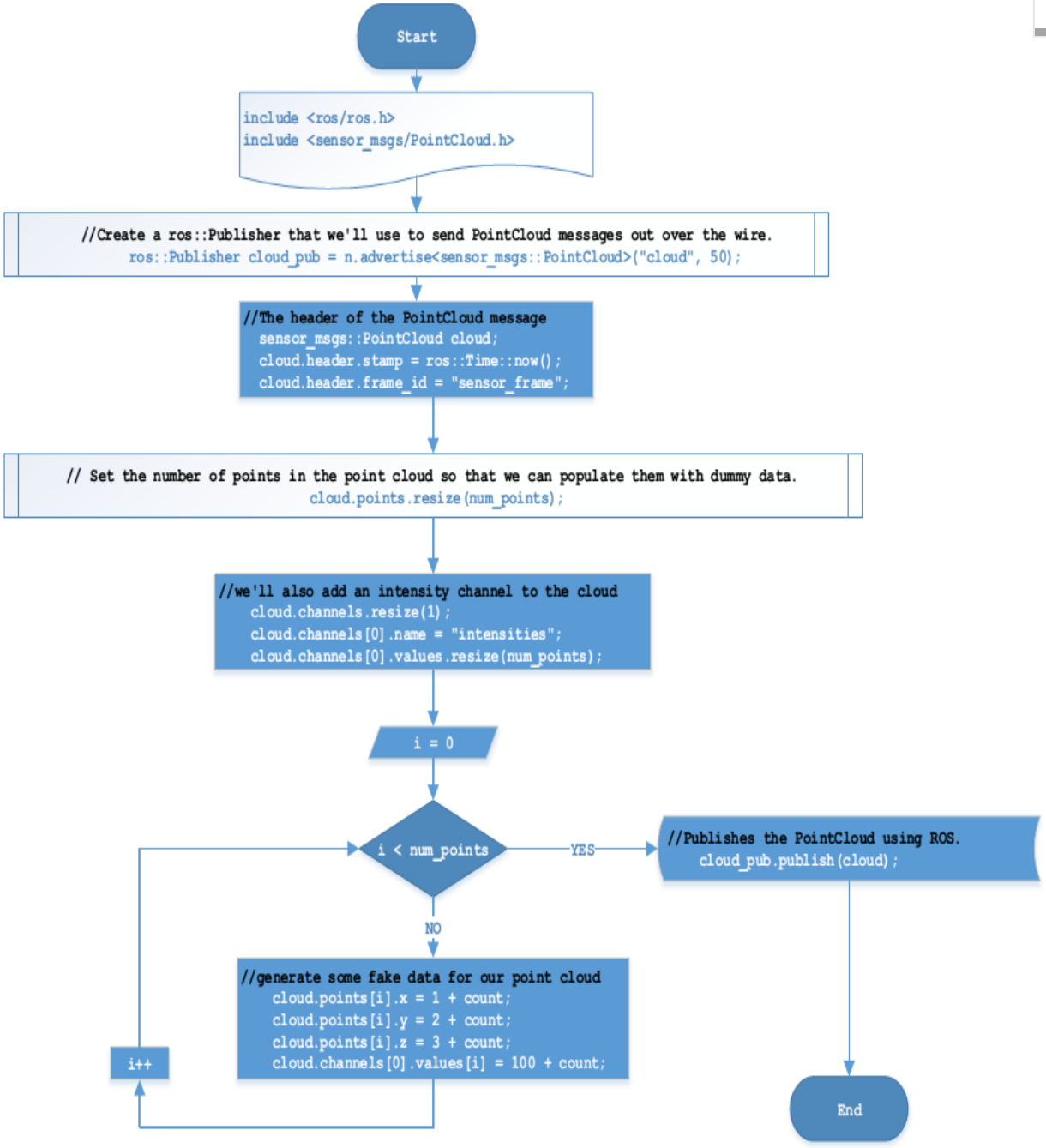
Now we create a new file in the package robot_Nav_stack1/src with the name laser.cpp the following flowchart describe the code in it:



Publishing PointClouds over ROS

For storing and sharing information about a number of points in the virtual world, ROS provides a sensor_msgs/PointCloud message. This message is meant to support arrays of points in three dimensions along with any associated data stored as a channel.

Publishing a PointCloud with ROS is fairly straightforward. We create now a new file in the package robot_Nav_stack1/src name it PointCloud.cpp, the following flowchart describe the C++ code.



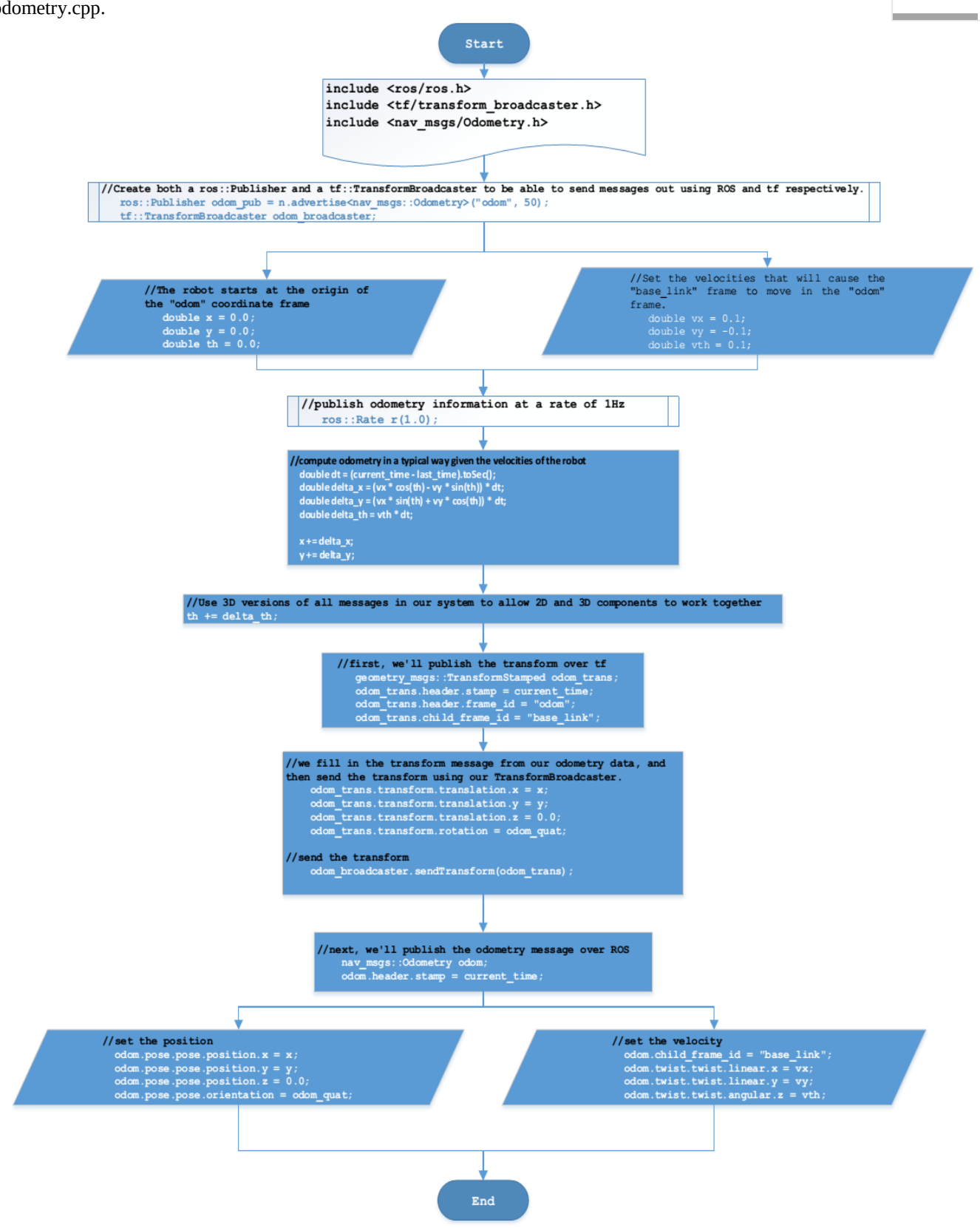
Now that we've written our nodes, we need to build it using Catkin_Make commend line. We note that with the previous nodes codes template, you can use any laser although it has no driver for ROS. You only have to change the fake data with the right data from your laser in our case we have used the data from the Kinect V2 specially IR Emitter information.

IV.4.1.3. Odometry information

The odometry is the distance of something relative to a point. The navigation stack uses Tf to determine the robot's location in the world and relate sensor data to a static map. However, Tf does not provide any information about the velocity of the robot. Because of this, the navigation stack requires that any odometry source publish both a transform and a message over ROS that contains velocity information. The type of message used by the navigation stack is nav_msgs/Odometry. This message stores an estimate of the position and velocity of a robot in free space.

Any odometry source must publish information about the coordinate frame that it manages. In the following we will take a look in the flowchart describing the C++ code for

publishing odometry, after creating a new file in the package robot_Nav_stack1/src name it odometry.cpp.



We need to write down the following dependences in to the Manifest.xml file then build the package using Catkin_Make command line.

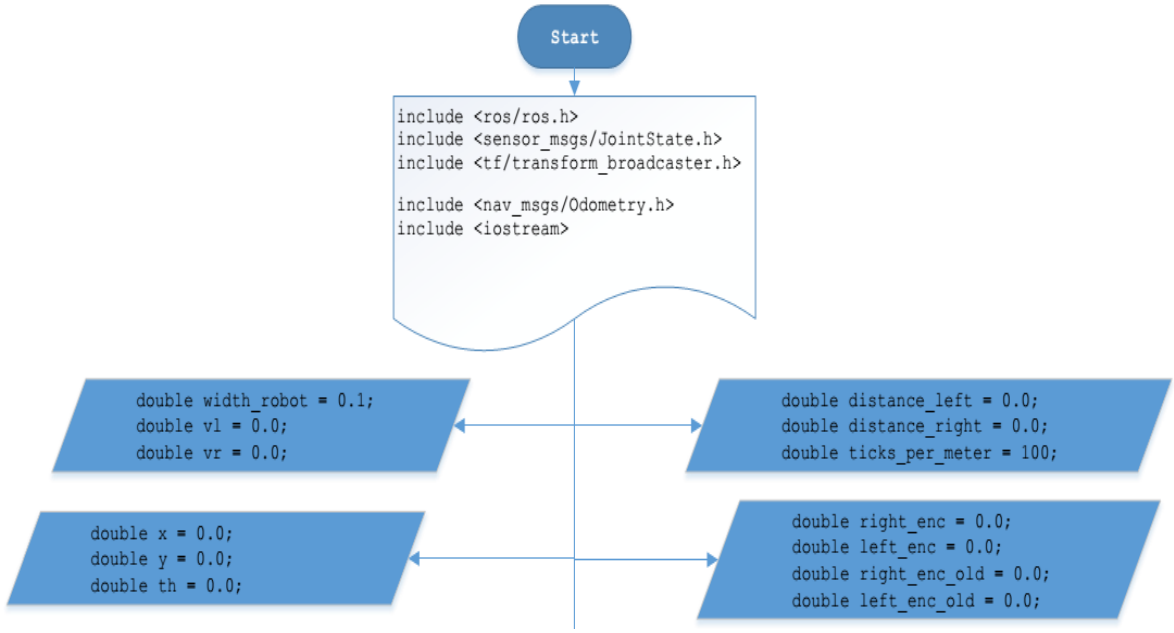
```
<depend package="tf"/>
<depend package="nav_msgs"/>
```

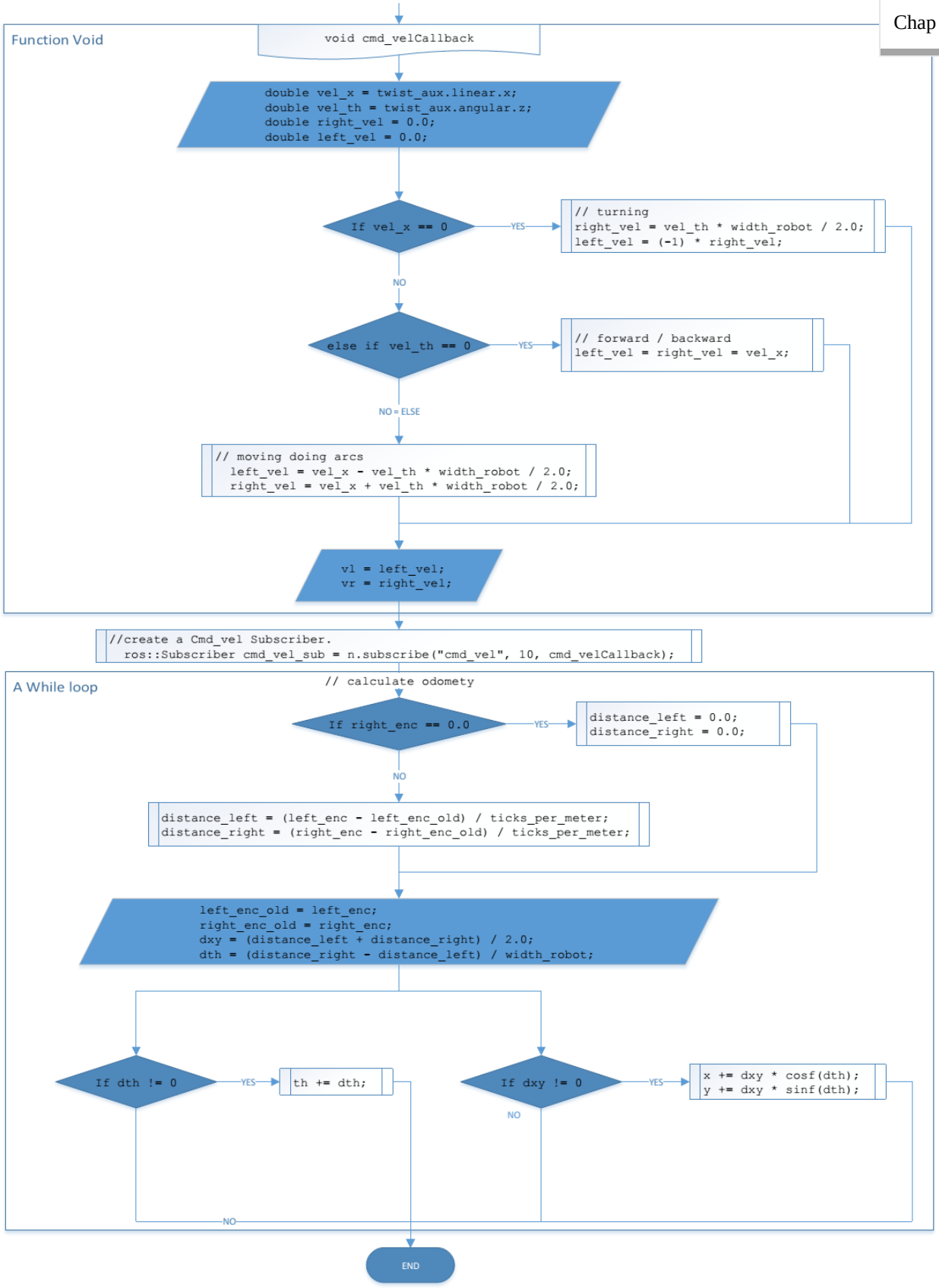
IV.4.1.4. Base Controller (base controller)

A base controller is an important element in the navigation stack because it is the only way to effectively control our robot. It communicates directly with the electronics of our robot. ROS does not provide a standard base controller, so we must write a base controller for our mobile platform.

Our robot has to be controlled with the message type geometry_msgs/Twist. This message is used on the Odometry message that we have seen before. For our robot, we will only use the linear velocity x and the angular velocity z. This is because our robot is on a differential wheeled platform, and it has two motors to move the robot forward and backward and to turn.

Now we create a new file in the package robot_Nav_stack1/src and name it base_controller.cpp. The following flowchart describes the file internal code file:





We should insert the following line in the CMakeListes.txt file to create an executable from this file; then, we run the command line catkin_make to build the package.

```
rosbuild_add_executable(base_controller src/base_controller.cpp)
```

IV.4.1.5. Creating a map in ROS using SLAM (Simultaneous Localization and Mapping)

SLAM is concerned with the problem of building a map of an unknown environment by a mobile robot while at the same time navigating the environment using the map. The term SLAM is an acronym for Simultaneous Localization and Mapping. It was originally developed by Hugh Durrant-Whyte and John J. Leonard [9]. SLAM consists of multiple parts: Landmark extraction, data association, state estimation, state update and landmark update. SLAM is more like a concept than a single algorithm. There are many steps involved in SLAM and these different steps can be implemented using different algorithms. SLAM is applicable for both 2D and 3D motion [9].

The SLAM process consists of several steps. The goal of the process is to use the environment to update the position of the robot. Since the odometry of the robot (which gives the robots position) is often erroneous, we cannot rely directly on the odometry. We can use laser scans of the environment to correct the position of the robot. This is accomplished by extracting features from the environment and re-observing when the robot moves around. An EKF (Extended Kalman Filter) is the heart of the SLAM process. It is responsible for updating where the robot thinks it is based on these features. These features are commonly called landmarks. The EKF keeps track an estimate of the uncertainty in the robot's position and also the uncertainty in these landmarks while seeing the environment. An outline of the SLAM process is given in figure 4.8[10].

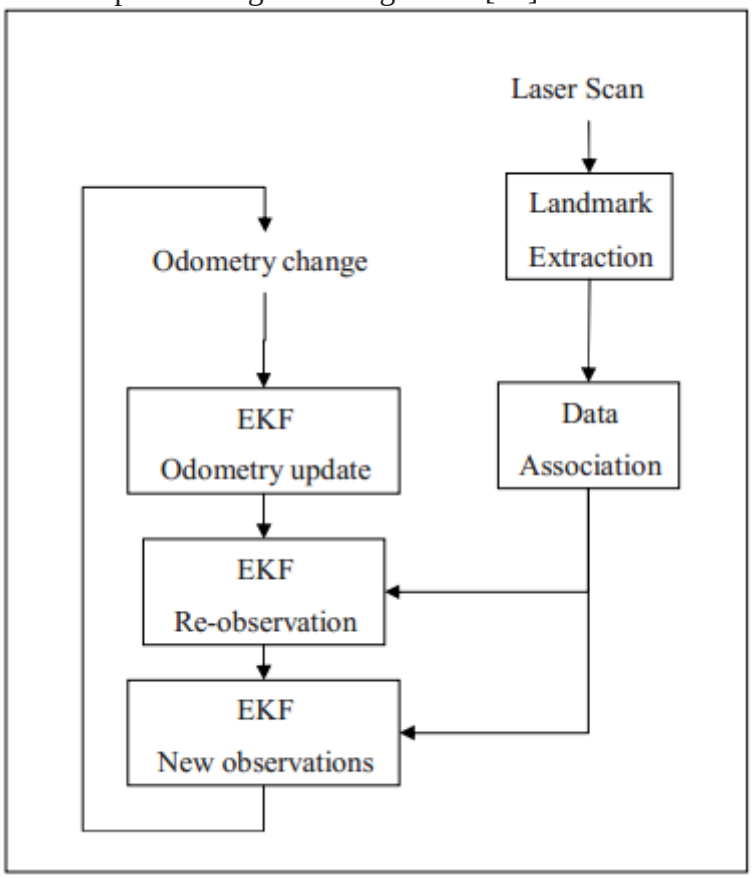


Figure 4.8. Overview of the SLAM process [10].

When the odometry changes as the robot moves the uncertainty pertaining to the robot's new position is updated in the EKF using Odometry update. Landmarks are then extracted from the environment from the robot new position. The robot then attempts to associate these landmarks to the previously seen landmarks observations. Re-observed landmarks are then used to update the robot's position in the EKF. Landmarks which have not been seen previously are added to the EKF as new observations so they can be re-observed later [10].

Creating the map step by step

To start building the map we need to run the following commands one by one in Ubuntu terminal. To run the slam_gmapping package we use.

```
roslaunch robot_navigation gmapping_demo.launch
```

To move the robot in rviz environment using keyboard touch so that we can explore the map and receive data we use the command line:

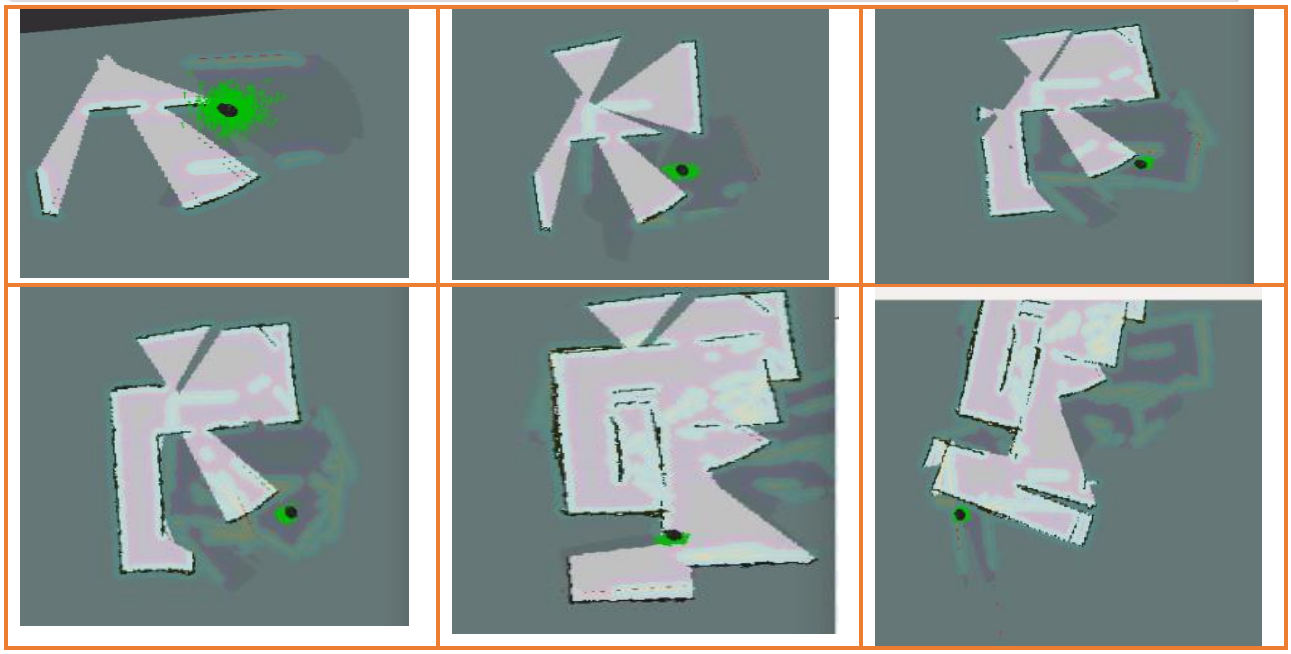
```
roslaunch turtlebot_teleop keyboard_teleop.launch
```

To start the visual scan for the map in rviz we use the command line:

```
roslaunch robot_stage robot_in_stage.launch
```

When we see that we have explored the full closed map we need to run the following commend line to save the map (see figure 4.9). This will be detailed in next part.

```
$ rosrun map server map saver -f map
```



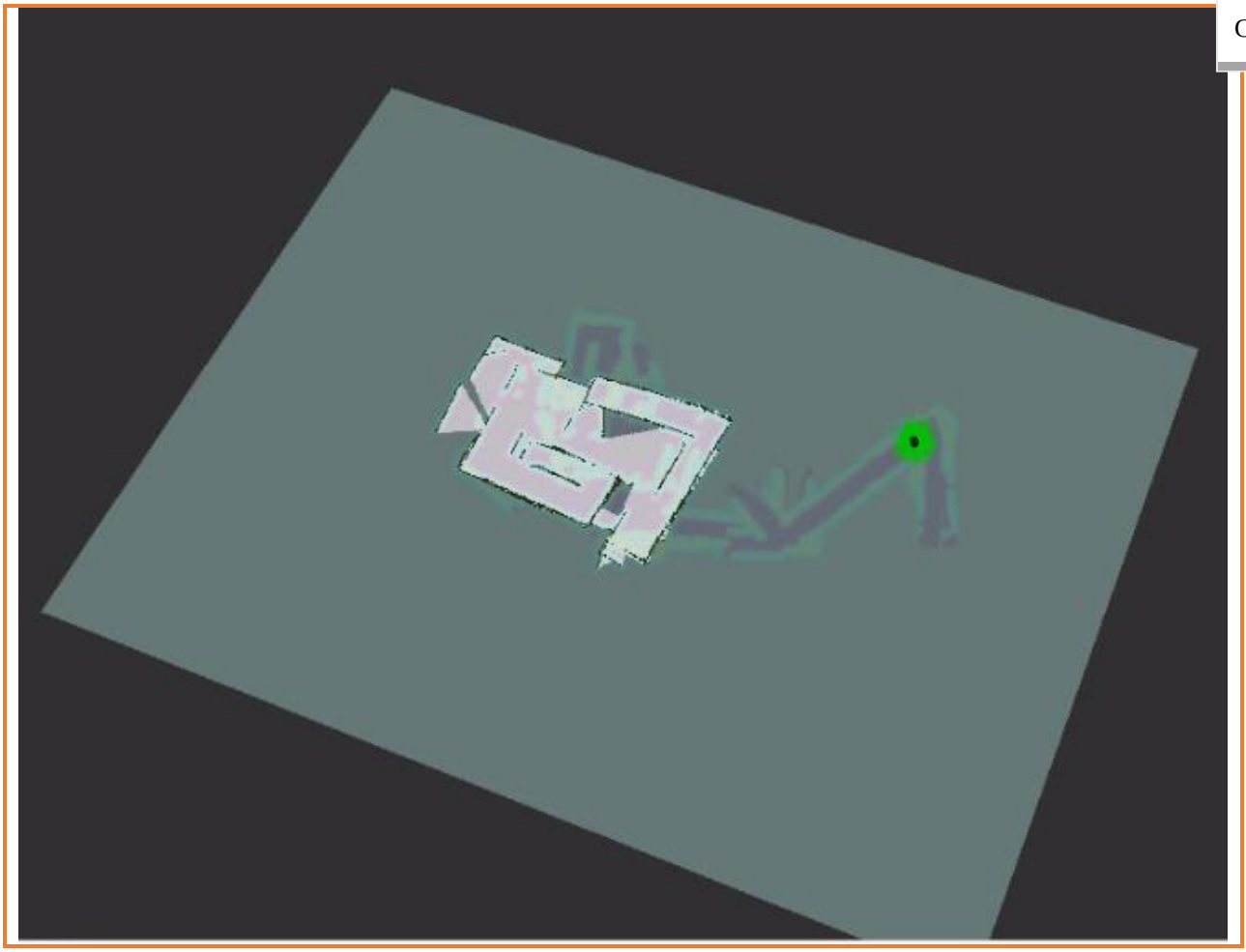


Figure 4.9. Step by step screenshots for the map building process

IV.4.1.6. Saving the map using map_server

To save the map in a specific folder like /tmp as my_map we use the command line as have been shown in the previous section

```
$ rosrun map server map saver -f map
```

```
ala@ala-HP-ProBook-4520s:~/catkin_robot$ rosrun map_server map_saver -f /tmp/my_
map
[ INFO] [1468936354.153794057]: Waiting for the map
[ INFO] [1468936354.434023844]: Received a 200 X 200 map @ 0.050 m/pix
[ INFO] [1468936354.434180229]: Writing map occupancy data to /tmp/my_map.pgm
[ INFO] [1468936354.435773346]: Writing map occupancy data to /tmp/my_map.yaml
[ INFO] [1468936354.495913624]: Done
```

This command will create tow files: map.pgm and map.yaml. The first one is the map in the -.pgm format. The other is the configuration file for the map (see figure 4.10, figure 4.11 and figure 4.12).

Figure 4.10. The files created in the catkin_robot workspace.

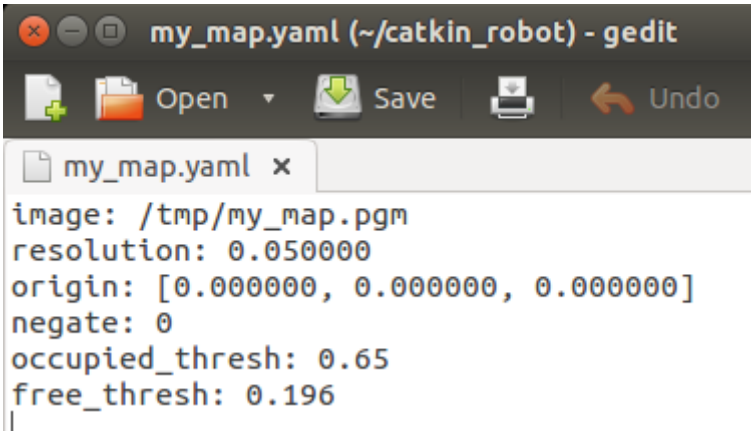


Figure 4.11. my_map.yaml configuration file description.

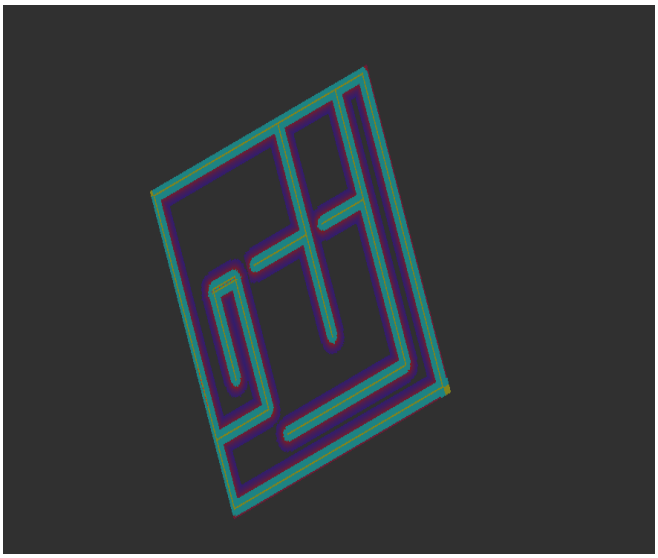


Figure 4.12. The final map build.

IV.4.1.7. Loading the map using map_server

When we want to use the map built with our robot, it is necessary to load it with the map_server package. The following command will load the map:

```
$ rosrun map_server map_server map.yaml
```

But to make it easy, we create another .launch file in the package robot_gazebo/launch with the name gazebo_map_robot.launch and described by the following algorithm:

-- start up gazebo world --
    call file "gazebo_map_robot.launch"
    call node "joint_state_publisher"
-- start robot state publisher --
    call node "robot_state_publisher"
    set parameter name "publish_frequency" to value "50.0"
    call node "map_server" and load file "map.yaml"
    call node to start map in "rviz"

Now launch the file using the next command and we should remember the model of the robot that which will be used; then, we will see rviz with the robot and the map.

```
$ roslaunch robot gazebo gazebo map robot.launch model:="`rospack find Urdf`/robot/.URDF"
```

IV.4.2. Navigation stack – Beyond Setups

The main aim of the ROS navigation stack packages is to move the robot from the start position to the goal position, with avoiding collision with the environment. This packages come with an implementation of several navigation related algorithms which can easily help implement autonomous navigation in the mobile robot. In the following we will take a look in the essential additional packages that we need to create to perform autonomous navigation correctly and create the optimal path when we specify a goal target for our robot navigation.

IV.4.2.1. Costmaps configuration (globalcost map and localcost map)

The robot will move through the map using two types of navigation—global and local.

- The global navigation is used to create paths to a desired goal in the map or a far-off distance as shown in figure 4.13.

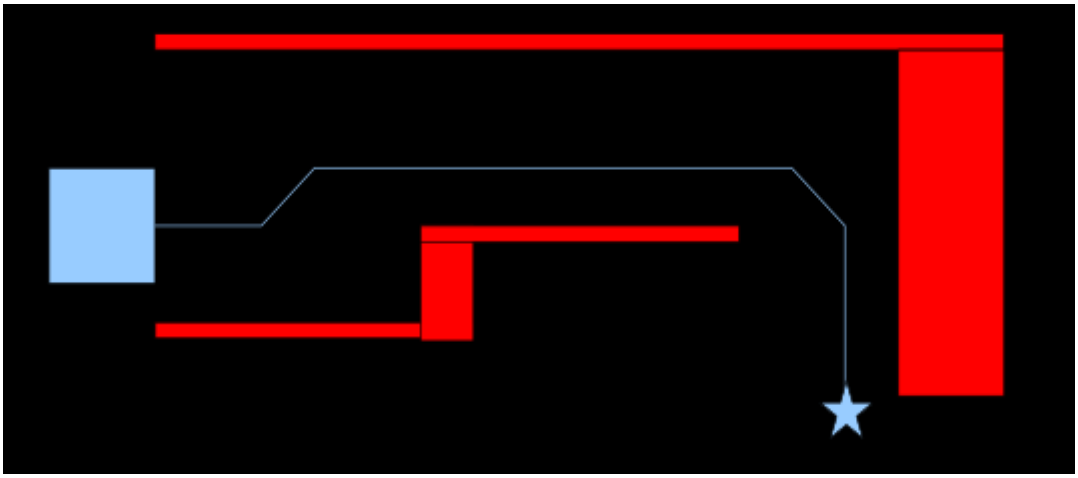


Figure 4.13. Demonstration for the global navigation.

- The local navigation is used to create paths in the nearby distances and avoid obstacles, for example, a square window of 4 x 4 meters around the robot as shown in figure 4.14.

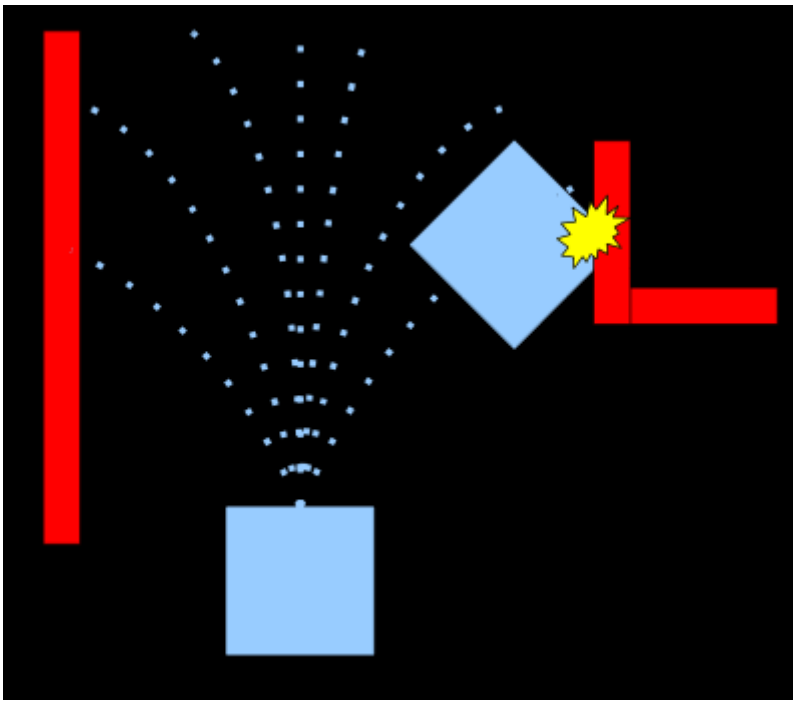


Figure 4.14. Demonstration for the local navigation.

These modules use costmaps to save all the information of our map. The cosmaps have parameters to configure the behaviours, and they have common parameters as well, which

are configured in a shared file. Configuration basically consists of three files where we can set-up different parameters. These files are as follows:

- costmap_common_params.yaml
- global_costmap_params.yaml
- local_costmap_params.yaml

Let's start by creating three different files in the package robot_nav_stack2/launch with the names costmap_common_params.yaml, global_costmap_params.yaml and local_costmap_params.yaml.

The following algorithm describes the costmap_common_params.yaml file:

```
        set obstacle_range to 2.5
        set raytrace_range to 3.0
        Fix footprint position to [[-0.2, -0.2], [-0.2,0.2], [0.2, 0.2], [0.2, -0.2]]
-- set parameter to robot_radius: ir_of_robot --
        set inflation_radius to 0.55
        set observation_sources to the value of laser_scan_sensor
```

This file is used to configure common parameters. The parameters are used in local_costmap and global_costmap.

The algorithm below is describes the global_costmap_params.yaml file:

```
For the global_costmap:
        set global_frame to path /map
        set robot_base_frame to path /base_footprint
        set update_frequency to 1.0
        set static_map to true
```

The following script is present in local_costmap_params.yaml file:

```
For the local_costmap:
        set global_frame to path /map
        set robot_base_frame to path /base_footprint
        set update_frequency to 1.0
        set publish_frequency to 2.0
        set static_map to true
        set rolling_window to false
        set width to 10.0
        set height to 10.0
        set resolution to 0.1
```

Once we have the costmaps configured, it is necessary to configure the base planner. The base planner is used to generate the velocity commands to move the robot. So we need to create a new file in the package robot_nav_stack2/launch and name it base_local_planner_params.yaml; then, we write the following algorithm as a description to the real code:

```
For TrajectoryPlannerROS:
        set max_vel_x to 1
        set min_vel_x to 0.5
        set max_rotational_vel to 1.0
        set min_in_place_rotational_vel to 0.4
        set acc_lim_th to 3.2
        set acc_lim_x to 2.5
        set acc_lim_y to 2.5
        set holonomic_robot to false
```

After creating those files and save them we need to build them using the command line catkin_make in ubuntu terminal.

Creating a launch file for the previous configuration

Now, we have all the files created and the navigation stack configured. To run everything, we need to create a launch file in the package robot_nav_stack2/launch with the name move_base.launch. The following algorithm describes the launch program:

```
-- Run the map server --
         call node "map_server" find “map.yaml"
```

find amcl "amcl_demo.launch"
call node "move_base"

-- Run the configuration files --
call file "costmap_common_params.yaml" command "load global_costmap"
call file "costmap_common_params.yaml" command "load local_costmap"
call file "local_costmap_params.yaml" command "load"
call file "global_costmap_params.yaml" command "load"
call file "base_local_planner_params.yaml" command "load"

With this file we will launch all the files that we have created in the above section at the same time.

IV.4.2.2. Adaptive Monte Carlo Localization (AMCL) for localization

This package provides probabilistic localization system for a robot moving on 2D. It implements the adaptive Monte Carlo localization approach, which uses a particle filter to track the pose of a robot against a known map. To run this package manually we us the following command line in Ubuntu terminal:

```
$   roslaunch robot_gazebo amcl_demo.launch
```

IV.4.2.3. Path planning and obstacles avoidance

Previously, we have created the necessary files needed to perform the navigation process in a proper way. So, now we deal with the last step in this process by generating the last .launch file in the package robot_stdr/launch and name it robot_in_stdr with the following algorithm to describe the launch file code and start performing path planning, obstacle avoidance and reaching the needed goals.

Where here we just start the necessary nodes and packages listed at first to start the navigation process.

-- The Robot navigation simulation: --
- stdr
- move_base
- amcl
- map_server
- rviz view

set names to "base", "stacks", "3d_sensor", "laser_topic", "odom_topic", "odom_frame_id", "global_frame_id".

-- Name of the map to use (without path nor extension) and initial position --
Set name as "map_file" value is "find map file"
Set name as "initial_pose_x" value is "2.0"
Set name as "initial_pose_y" value is "2.0"
Set name as "initial_pose_a" value is "0.0"
Set name as "min_obstacle_height" value is "0.0"
Set name as "max_obstacle_height" value is "5.0"

-- ********** Stdr********** --
call file “robot_manager.launch"
-- Run STDR server with a prefedined map--
call node "stdr_server" with screen output" map_file"
-- Run Gui --
call file "stdr_gui.launch"
-- Run the relay to remap topics --
call file “relays.launch.xml"

-- ********** Robot Model ********** --
call file "robot.launch.xml"
call "base"

```
        call "stacks"
        call "3d_sensor"
        call node "joint_state_publisher"
        call node "mobile_base_nodelet_manager"
        call node "cmd_vel_mux"

-- ****** Maps ***** --
        call node "map_server"

-- ********** Navigation ********** --
        call file "move_base.launch.xml"
        call "odom_topic"
        call "laser_topic"
        call "odom_frame_id"
        call "base_frame_id"
        call "global_frame_id"

-- ********** Manually setting some parameters ********** --
        set parameter value to "min_obstacle_height"
        set parameter value to "max_obstacle_height"
        set parameter value to "min_obstacle_height"
        set parameter value to "max_obstacle_height"

-- ********** AMCL ********** --
        call file "amcl.launch.xml"
        check name "scan_topic" with value "laser_topic"
        check name "use_map_topic" with value "true"
        check name "odom_frame_id" with value "odom_frame_id"
        check name "base_frame_id" with value "base_frame_id"
        check name "global_frame_id" with value "global_frame_id"
        check name "initial_pose_x" with value "initial_pose_x"
        check name "initial_pose_y" with value "initial_pose_y"
        check name "initial_pose_a" with value "initial_pose_a"

-- ******* Small tf tree connector between robot0 and base_footprint******* --
        call node "tf_connector" and load file "tf_connector.py"

 -- ********** Visualisation ********** --
        Call node to start "rviz" and load the file "robot_navigation.rviz"
```

Finally, we should build the package using catkin_make command line to make the files executable.

When we run the previous .launch file using the following command line, the Rvis simulator popup as shown in Figure 4.15. The map and the robot will be loaded to the simulator.

```
$ roslaunch robot stdr robot in stdr.launch
```

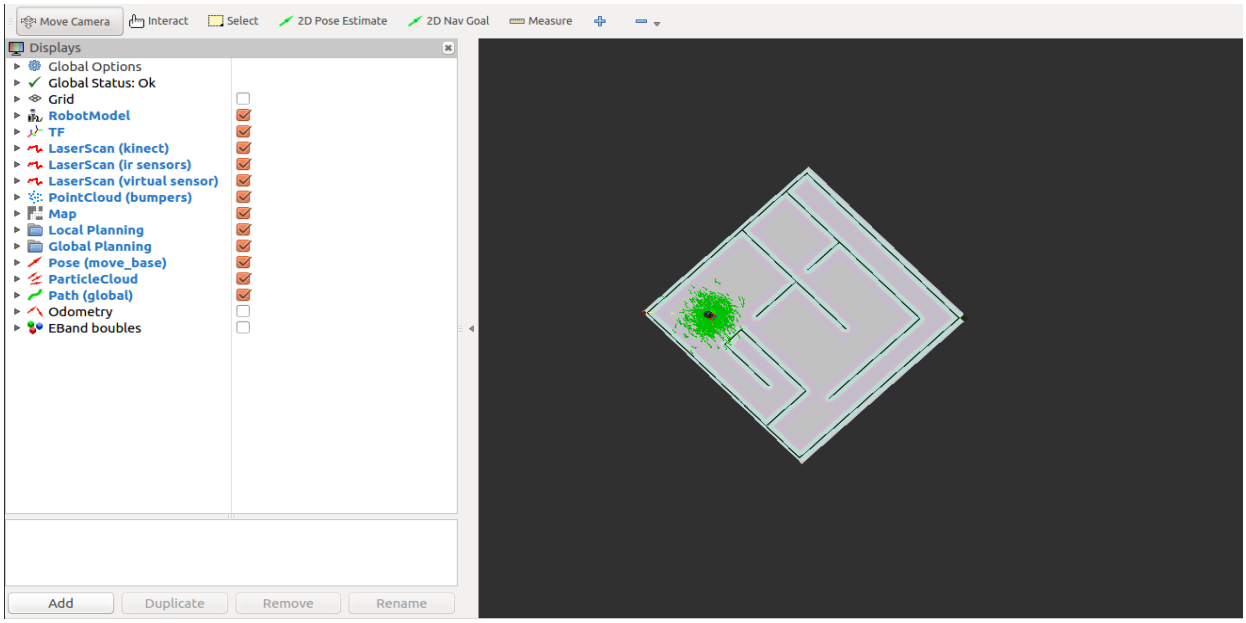
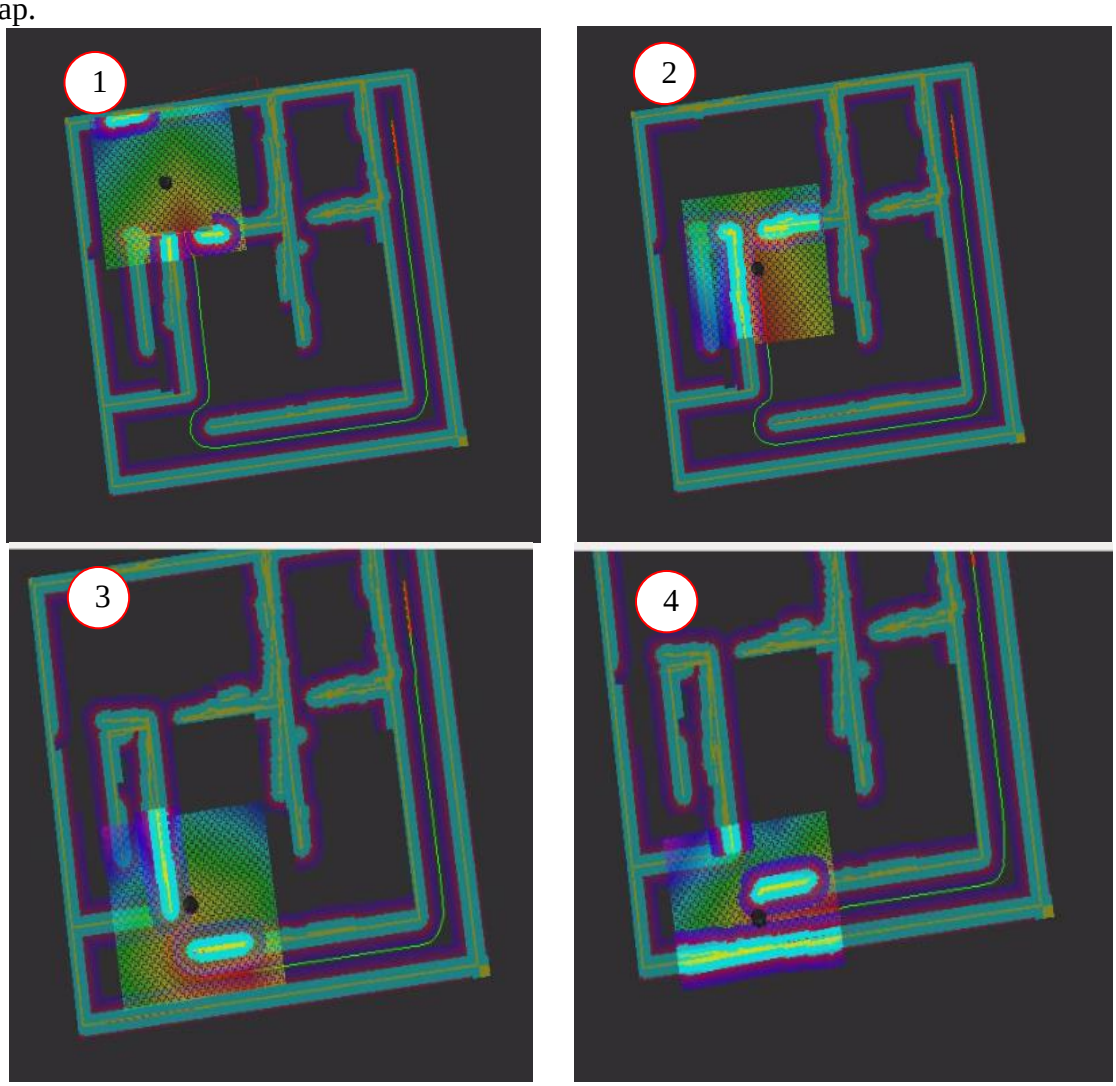


Figure 4.15. Rviz popup screen to start the robot navigation.

Now, we just perform navigation in the map that we have created using the rviz parameter "2D Nav Goal" button is used to create an optimum path for the robot to reach the needed goal. Figure 4.16 shows the step by step path execution for a desired goal in known generated map.



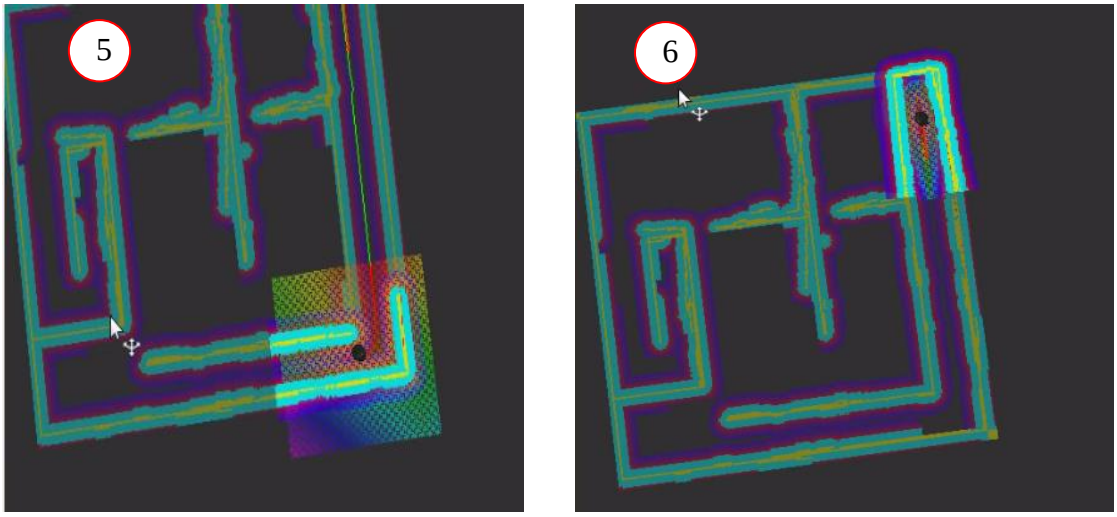


Figure 4.16. Step by step path planning execution.

IV.5. Conclusion

This chapter provides a depth explanation of the simulation of the RobuTER/ULM navigation process. Where, we have seen how to construct the 3D model of the robot in Solidworks and how to simulate the robot inside ROS indigo. Then, we showed how to build a map using SLAM package. Finally, we close the chapter by providing the path planning simulation procedure in 2D global map from one place to another. Hence, obstacles avoidance is done automatically in the local and global plane and adapting the path for new goals actualized regularly.

The aim of this project was to simulate and test an artificial intelligent navigation for differential drive autonomous mobile robot based on the Kinect camera. Hence, the first purpose was to make the mobile robot able to achieve any desired position by itself, including sensing the surrounding environment and building a virtual map. The second purpose was determining its position as a localization process, and finding and executing its path.

During this work, we have stated some aspects about autonomous navigation for mobile robot, technical robot terms, sensors and present the robot data. In the simulation, we have created a virtual environment for the robot to explore and build a map; then, we have used this 2D map to perform navigation and path planning.

Initially, the robot 3D model and the virtual map were designed. Then, we have started by building files for navigation stack. The first part was for the robot setup where we have configured the transformation frame, Odometry information, sensor information and the base controller; so that this will allow the mobile robot to explore the virtual environment and construct the 2D map through the execution of the SLAM algorithm. The second part consist of building the navigation stack which concerns the path planning in local and global map by configuring the local costmap and global costmap and running the AMCL algorithm for localization. Finally, we have reached the needed simulation and received the desired results.

Future Works

- Implementing this study in the real RobuTER/ULM was kind-off impossible since the robot does not have a driver that is compatible with ROS so that all the algorithms will be installed inside the robot. Hence, for now the robot just receives a set of data for the velocities of the tow wheels and the angles of the arm joints. In addition, once they, at CDTA, integrate a driver compatible with ROS; it is needed to add files extra to the ones that we have created, files that will detect and communicate with the robot hardware and read and adapt information from the real world.

- The creation of a 3D map is possible using the Kinect camera and **RTAB-Map** (Real-Time Appearance-Based Mapping) package in ROS. It is an RGB-D SLAM approach based on a global loop closure detector with real-time constraints. So, navigation in 3D will be more accurate for obstacle avoidance than 2D map.

References and Bibliography

[1] Roland Siegwart and Illah R. Nourbakhsh *"Introduction to autonomous mobile robots"* MIT press. 2004.

[2] A. HENTOUT, B. BOUZOUIA, I. AKLI and R. TOUMI "*Mobile Manipulation: A Case Study"* InTech April, 2010.

[3] Jonathan Dixon, Oliver Henlich "*Mobile Robot Navigation*'', 10 June 1997.

[4] Ulrich Nehmzow *"Mobile Robotics: A Practical Introduction"* Second Edition. Springer. 2003

[5] BAKDI Azzeddine, BOUTAMAI Hakim *"Visual path planning for autonomous*
mobile robots: Case of RobuTER" Final Year Project Report for the Degree of MASTER, 2015.

[6] Carl D. Crane III , Joseph Duffy *"Kinematic Analysis of Robot Manipulators"* Cambridge University Press, 2008.

[7] Editado por Margarita N. Favorskaya,Lakhmi C. Jain *"Computer Vision in Control Systems-2: Innovations in Practice",* Spring.

[8] Arbnor Pajaziti, Petrit Avdullahu *"SLAM – Map Building and Navigation via ROS"* International Journal of Intelligent Systems and Applications in Engineering, Received 05th September 2014, Accepted 18th December 2014.

[9] Søren Riisgaard and Morten Rufus Blas *"SLAM for Dummies"* A Tutorial Approach to Simultaneous Localization and Mapping, *By* the dummies'.

[10] Aaron Martinez, Enrique Fernández *"Learning ROS for Robotics Programming"* Published by Packt Publishing Ltd, September 2013.

[11] Lentin Joseph *"Mastering ROS for Robotics Programming*" Published by Packt Publishing Ltd, December 2015.

Appendix
-A-

A.1. An Introduction to Robot Operating System ROS

Robot Operating System (ROS) is a trending robot application development platform that provides various features such as message passing, distributed computing, code reusing, and so on.

The ROS project was started in 2007 with the name *Switchyard* by Morgan Quigley as part of the Stanford STAIR robot project. The main development of ROS happened at Willow Garage.

Here are some of the reasons why people choose ROS over other robotic platforms such as Player,YARP, Orocos, MRPT, and so on [12]:

• **High-end capabilities**: ROS comes with ready to use capabilities, for example, **SLAM** (**Simultaneous Localization and Mapping**) and **AMCL** (**Adaptive Monte Carlo Localization**) packages in ROS can be used for performing autonomous navigation in mobile robots and the MoveIt package for motion planning of robot manipulators.

• **Tons of tools**: ROS is packed with tons of tools for debugging, visualizing, and performing simulation. The tools such as rqt_gui, RViz and Gazebo are some of the strong open source tools for debugging, visualization, and simulation. The software framework that has these many tools is very rare.

• **Support high-end sensors and actuators**: ROS is packed with device drivers and interface packages of various sensors and actuators in robotics. The high-end sensors include Velodyne-LIDAR, Laser scanners, Kinect, and so on and actuators such as Dynamixel servos. We can interface these components to ROS without any hassle.

• **Inter-platform operability**: The ROS message-passing middleware allows communicating between different nodes. These nodes can be programmed in any language that has ROS client libraries. We can write high performance nodes in C++ or C and other nodes in Python or Java. This kind of flexibility is not available in other frameworks.

• **Modularity**: One of the issues that can occur in most of the standalone robotic applications are, if any of the threads of main code crash, the entire robot application can stop. In ROS, the situation is different, we are writing different nodes for each process and if one node crashes, the system can still work. Also, ROS provides robust methods to resume operation even if any sensors or motors are dead.

• **Concurrent resource handling**: Handling a hardware resource by more than two processes is always a headache. Imagine, we want to process an image from a camera for face detection and motion detection, we can either write the code as a single entity that can do both, or we can write a single threaded code for concurrency. If we want to add more than two features in threads, the application behavior will get complex and will be difficult to debug. But in ROS, we can access the devices using ROS topics from the ROS drivers. Any number of ROS nodes can subscribe to the image message from the ROS camera driver and each node can perform different functionalities. It can reduce the complexity in computation and also increase the debug-ability of the entire system.

• **Active community**: When we choose a library or software framework, especially from an open source community, one of the main factors that needs to be checked before using it is its software support and developer community. There is no guarantee of support from an

open source tool. Some tools provide good support and some tools don't. In ROS, the support community is active. The ROS community has a steady growth in developers worldwide.

A.2. Understanding the ROS file system level

Similar to an operating system, ROS files are also organized on the hard disk in a particular fashion. In this level, we can see how these files are organized on the disk. The following graph shows how ROS files and folder are organized on the disk:

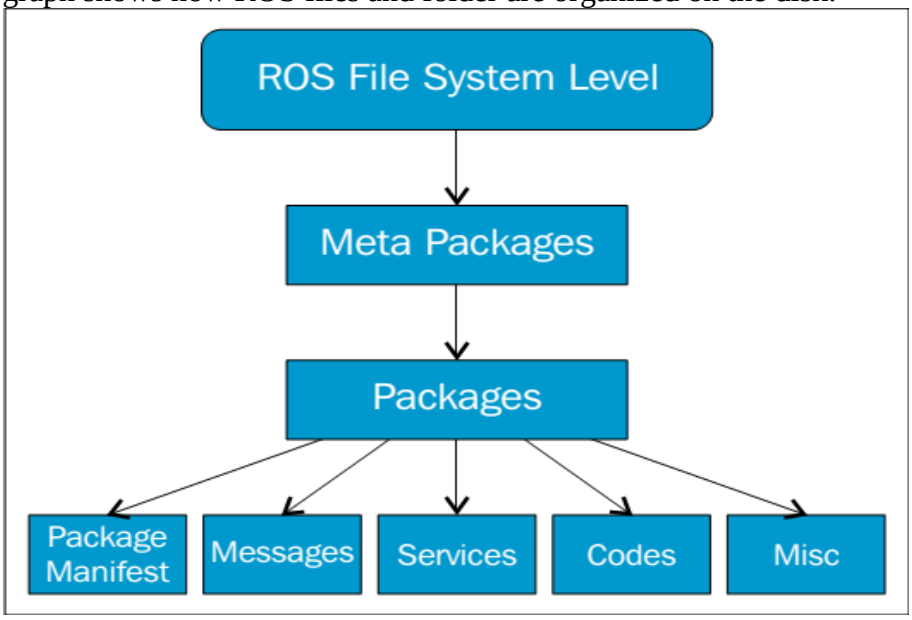


Figure A.1. ROS File system level [12].

A typical structure of a ROS package is shown here:

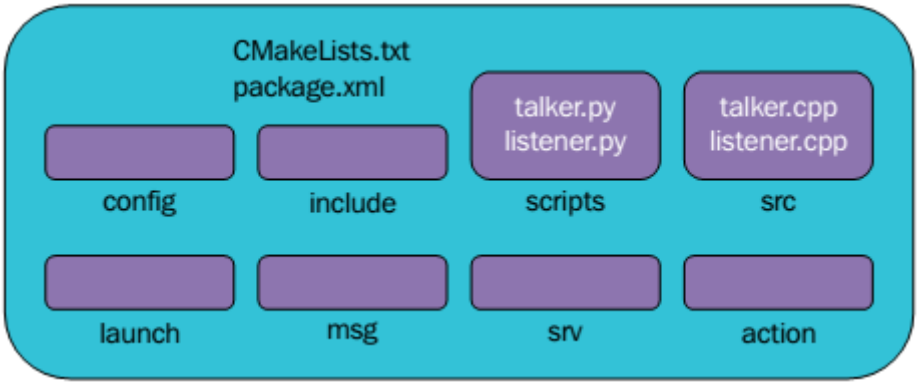


Figure A.2. Structure of a typical ROS package [12].

Let us now introduce some of ROS's architecture keywords. ROS uses the concept of nodes, messages, topics, stacks, and packages, below a quick described of this concepts:

- Node: A process that performs computation; nodes communicate with each other through messages.
- Message: A strictly type of data structure; a node sends a message by publishing it to a topic.
- Topic: Channel between tow or more nodes; nodes communicate by publishing and/or subscribing to the appropriate topics.
- Package: Compilation of nodes that can easily be compiled and ported to other computers, necessary to build a complete ROS-based controller system.
- Stack: Groups of ROS packages making easier the process of sharing code with the community.

A.3. Visualization and Simulation in ROS

The ROS framework comes with a great number of powerful tools to help the user and developer in the process of debugging the code, and detecting problems with both the hardware and software. In order to make simulations with our robots on ROS, we are going to use Gazebo, Rvize and Moveit.

Gazebo

Robot simulation is an essential tool in every roboticist's toolbox. A well-designed simulator makes it possible to rapidly test algorithms, design robots, and perform regression testing using realistic scenarios. Gazebo offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. At your fingertips is a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces.

Installing and Launching Gazebo

Install ROS indigo and get the simulator_gazebo package by the commend line:

```
sudo apt-get install ros-indigo-simulators
```

Setup ros environnement variables :

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

The standard Gazebo launch file is started using:

```
roslaunch gazebo_worlds empty_world.launch
```

Except in the case of Indigo, where the launch file is started using:

```
roslaunch gazebo_ros empty_world.launch
```

This should start the simulator and open up a GUI window that looks like this:

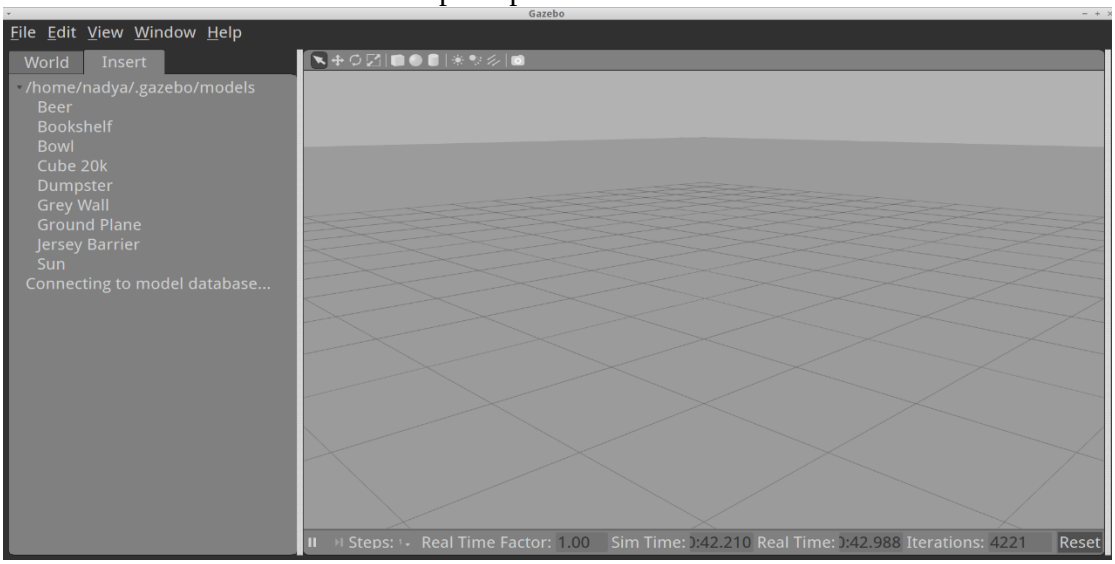


Figure A.3. The Gazebo GUI.

Rviz

Rviz stands for ROS visualization. It is a general-purpose 3D visualization environment for robots, sensors, and algorithms. Like most ROS tools, it can be used for any robot and rapidly configured for a particular application.

Rviz can plot a variety of data types streaming through a typical ROS system, with heavy emphasis on the three-dimensional nature of the data. In ROS, all forms of data are attached to a frame of reference.

Installing and launching Rviz

Download the rviz sources into your ros_workspace. First to satisfy any system dependencies.

```
rosdep install rviz
```

Now build the visualiser:

```
rosmake rviz
```

You might have to run a line such as

```
source /opt/ros/indigo/setup.bash
roscore &
```

Then start the simulator :

```
rosrun rviz rviz
```

When rviz starts for the first time, you will see an empty window:

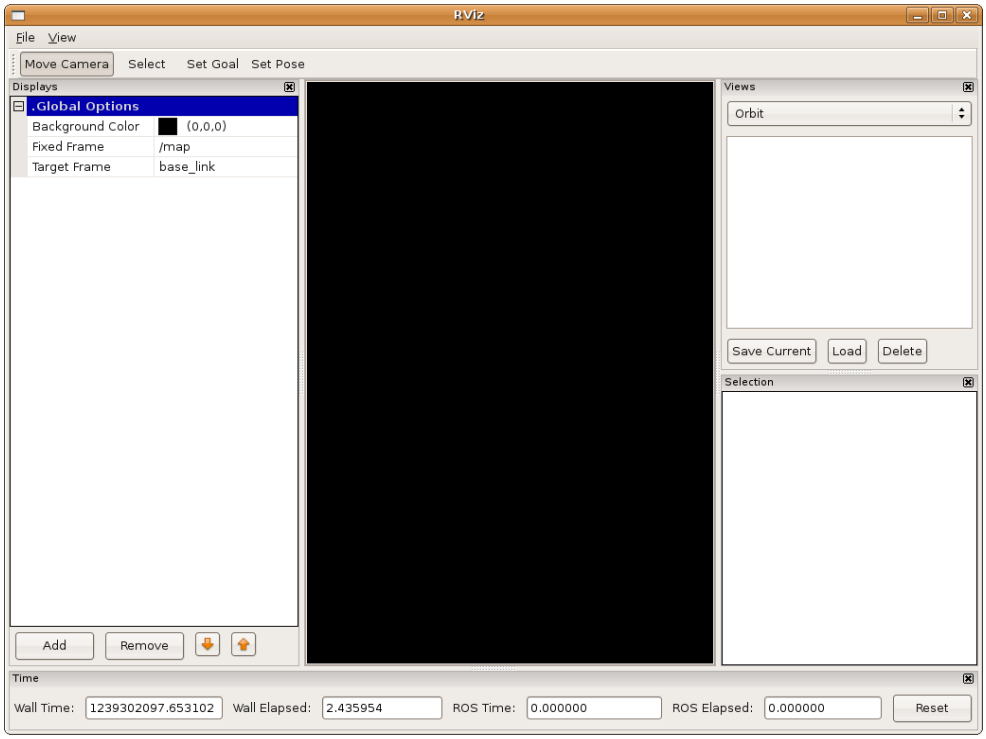


Figure A.4. The Rviz GUI.

STDR Simulator

Simple Two-Dimensional Robot Simulator (STDR Simulator) is a 2-D multi-robot Unix simulator. Its goals are:

Easy multi-robot 2-D simulation

STDR Simulator's goal is not to be the most realistic simulator, or the one with the most functionalities. Our intention is to make a single robot's, or a swarm's simulation as simple as possible, by minimizing the needed actions the researcher has to perform to start his/hers experiment. In addition, STDR can function with or without a graphical environment, which allows for experiments to take place even using ssh connections.

To be ROS compliant

STDR Simulator is created in way that makes it totally ROS compliant. Every robot and sensor emits a ROS transformation (Tf) and all the measurements are published in ROS topics. In that way, STDR uses all ROS advantages, aiming at easy usage with the world's most state-of-the-art robotic framework. The ROS compliance also suggests that the Graphical User Interface and the STDR Server can be executed in different machines, as well as that STDR can work together with the Rviz simulator in ROS.

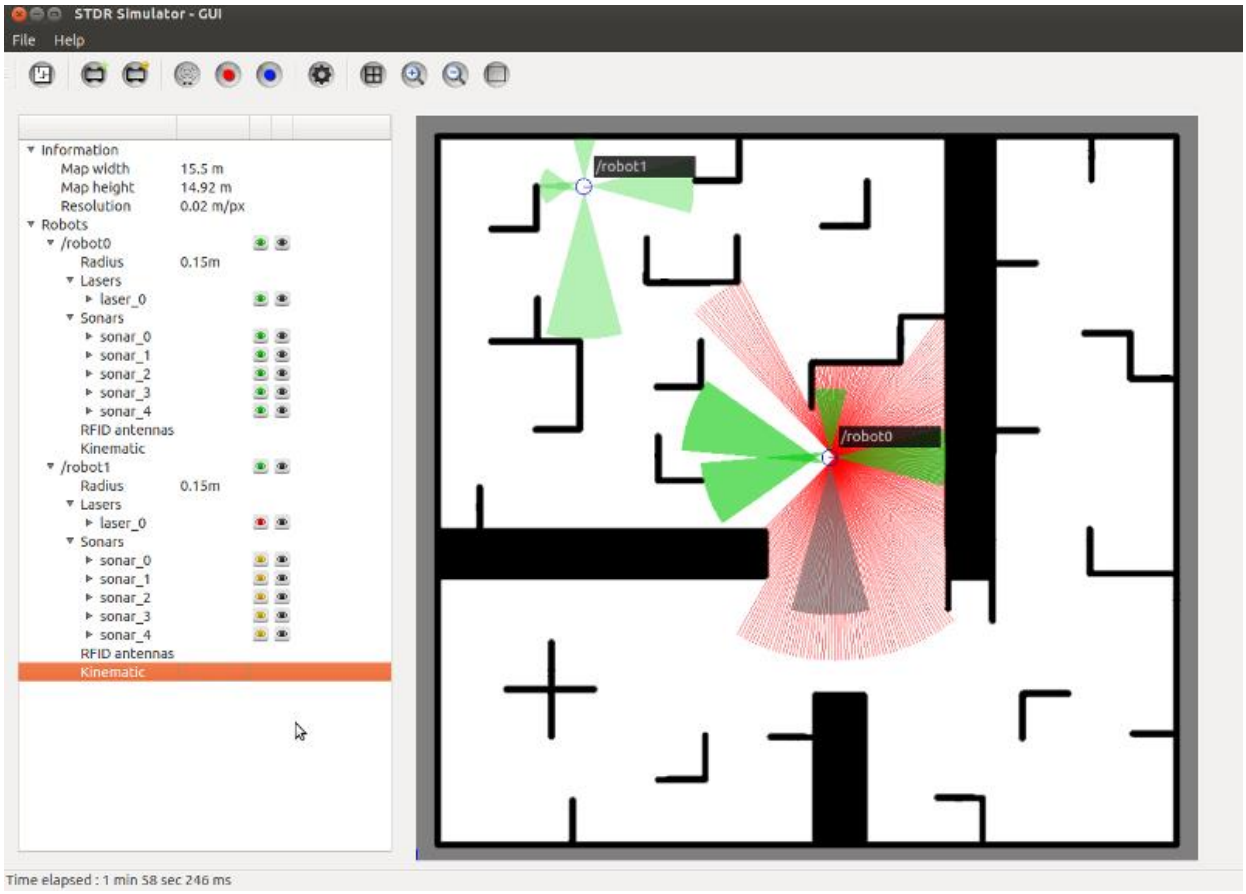


Figure A.5. The STDR GUI.

STDR Simulator ROS packages

- stdr_server, Implements synchronization and coordination functionalities of STDR Simulator.
- stdr_robot, Provides robot, sensor implementation, using nodelets for stdr_server to load them.
- stdr_parser, Provides a library to STDR Simulator, to parse yaml and xml description files.
- stdr_gui, A gui in Qt for visualizing purposes in STDR Simulator.
- stdr_msgs, Provides msgs, services and actions for STDR Simulator.
- stdr_launchers, Launch files, to easily bringup server, robots, guis.
- stdr_resources, Provides robot and sensor descripiton files for STDR Simulator.
- stdr_samples, Provides sample codes to demonstrate STDR simulator functionalities.

Appendix
-B-

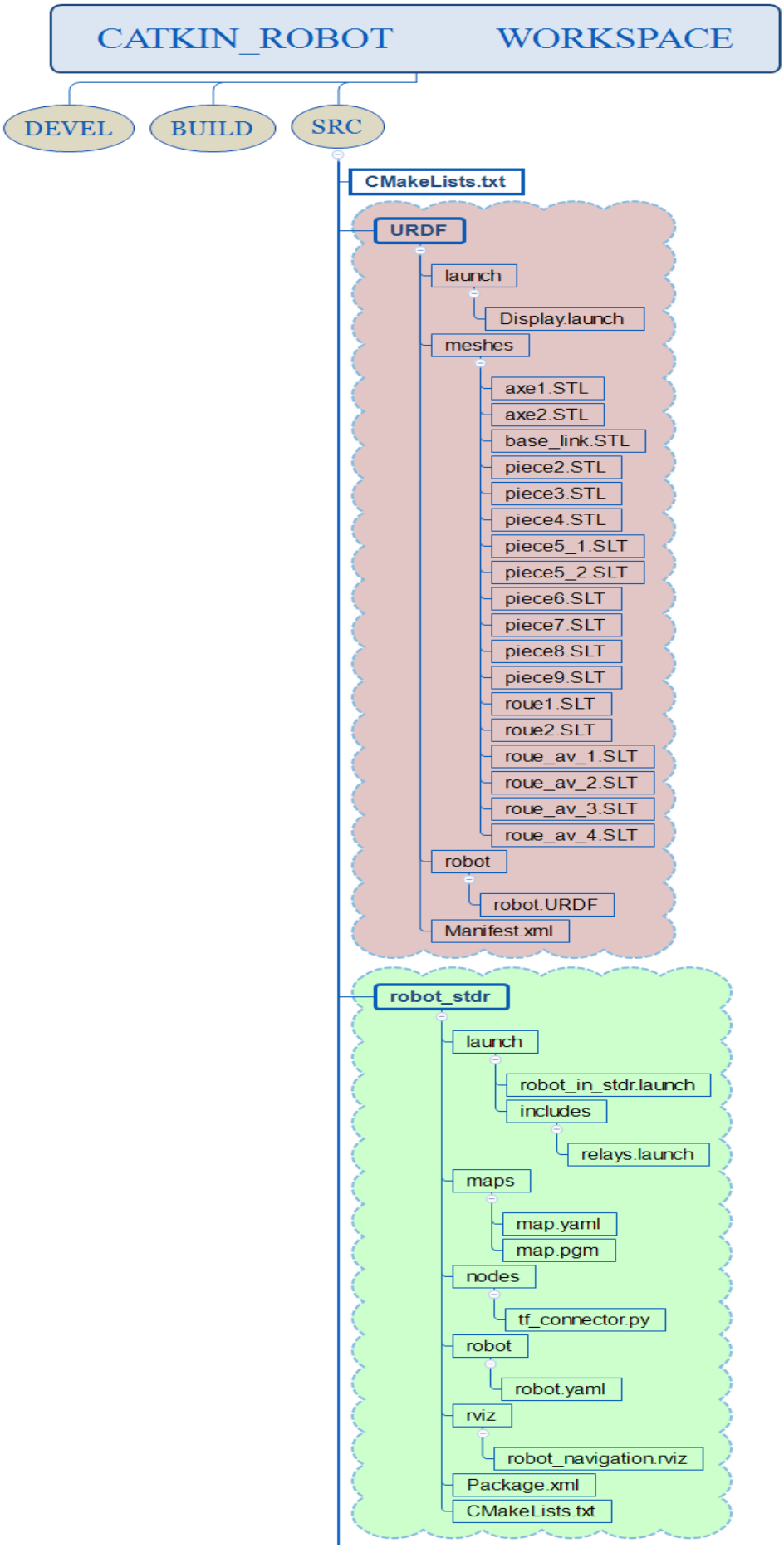
The Catkin workspace package architecture

In this part we will take a look on the packages that we have created in the **catkin_robot** workspace that we had made as a place to save our simulation files.

As we can see in the Figure C.1 below we do have three main files that are the basic architecture of our **catkin_robot** workspace where the **src** folder saves the packages that we create then to make this packages readable or executable we build them using the commend line **catkin_make** so that we generate system files in the **build** and **devel** folders of our **catkin_robot** workspace.

As we can see also we do have seven packages that describe the robot and the process of navigation with its files, each file contains codes either in C++, Python or Xml programing language.

For the description of the robot modal we do have the URDF package that we have created starting from the simulation in Solidworks 2015 and extracting the files to an URDF format by describing the joints and likes that build a transformation frame to each join point in the robot.


CATKIN_ROBOT WORKSPACE
DEVEL
BUILD
SRC
CMakeLists.txt
URDF
launch
Display.launch
meshes
axe1.STL
axe2.STL
base_link.STL
piece2.STL
piece3.STL
piece4.STL
piece5_1.SLT
piece5_2.SLT
piece6.SLT
piece7.SLT
piece8.SLT
piece9.SLT
roue1.SLT
roue2.SLT
roue_av_1.SLT
roue_av_2.SLT
roue_av_3.SLT
roue_av_4.SLT
robot
robot.URDF
Manifest.xml
robot_stdr
launch
robot_in_stdr.launch
includes
relays.launch
maps
map.yaml
map.pgm
nodes
tf_connector.py
robot
robot.yaml
rviz
robot_navigation.rviz
Package.xml
CMakeLists.txt

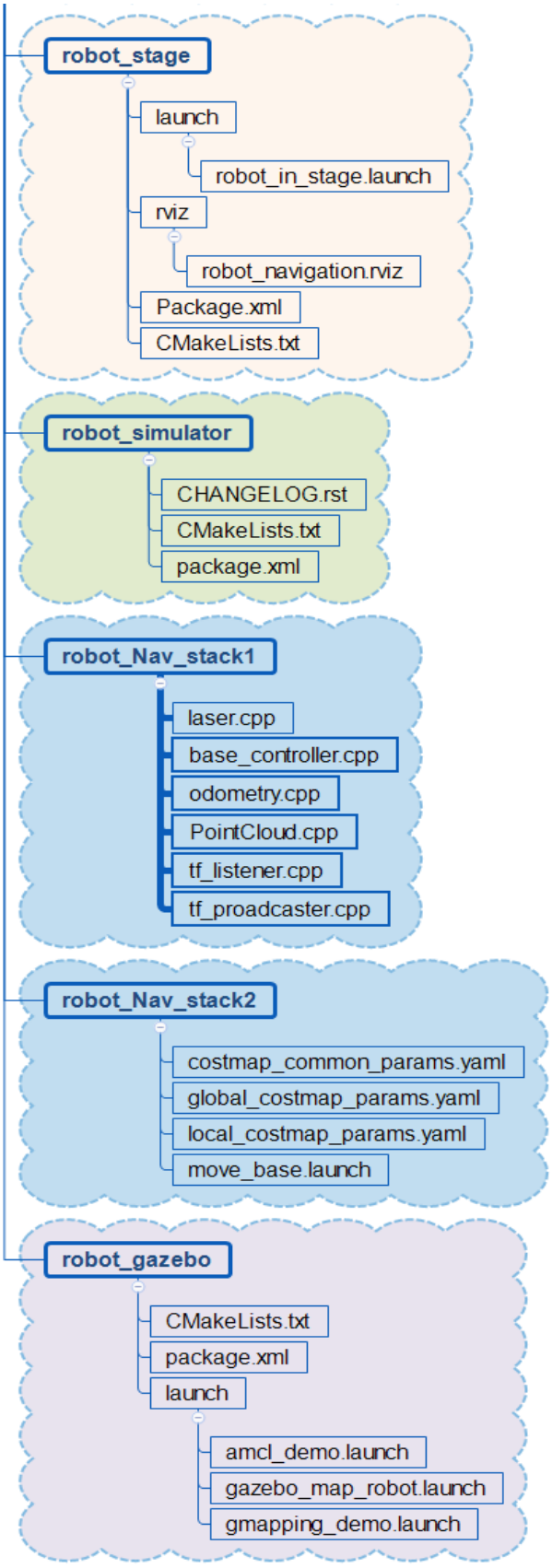


Figure B.1. The architecture of the Catkin_robot workspace files.

Appendix
-C-

Steps to Test the Kinect in ROS

In the following we will take a look on the process of installing and running the Kinect Xbox360-V2 driver in ROS indigo. Here with the Kinect V2 we use the **OpenNI** drivers instead of **Libfreenect** drivers that are compatible with the Kinect V1.

- sudo apt-get install libopenni0 libopenni-dev.
- sudo apt-get install ros-indigo-openni-camera.
- sudo apt-get install ros-indigo-openni-launch.
    - cd ~/Downloads
      unzip avin2-SensorKinect-v0.93-5.1.2.1-0-g15f1975.zip
      cd avin2-SensorKinect-15f1975/Bin
      tar -xjf SensorKinect093-Bin-Linux-x64-v5.1.2.1.tar.bz2
      cd Sensor-Bin-Linux-x64-v5.1.2.1
      sudo ./install.sh
- To test in Rviz simulator the kinect.
  roscore
  roslaunch openni_launch openni.launch
  rosrun image_view image_view image:=/camera/rgb/image_color
  rosrun rviz rviz

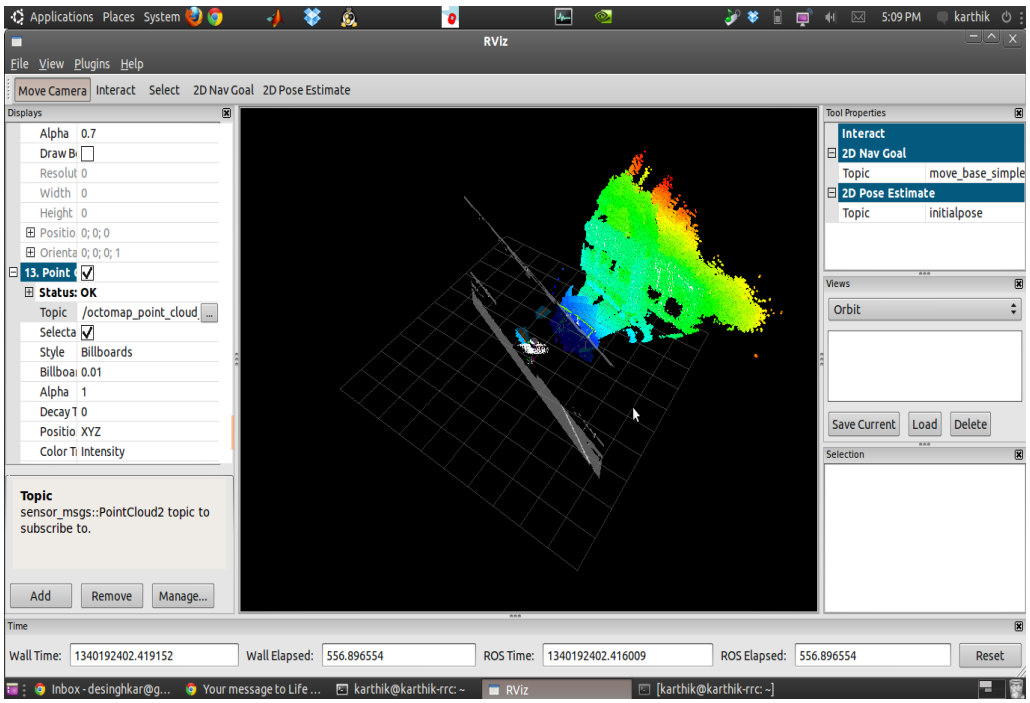


Figure C.1. The Kinect test in the Rviz simulator.

Appendix
-D-

Flowcharts shapes explanation

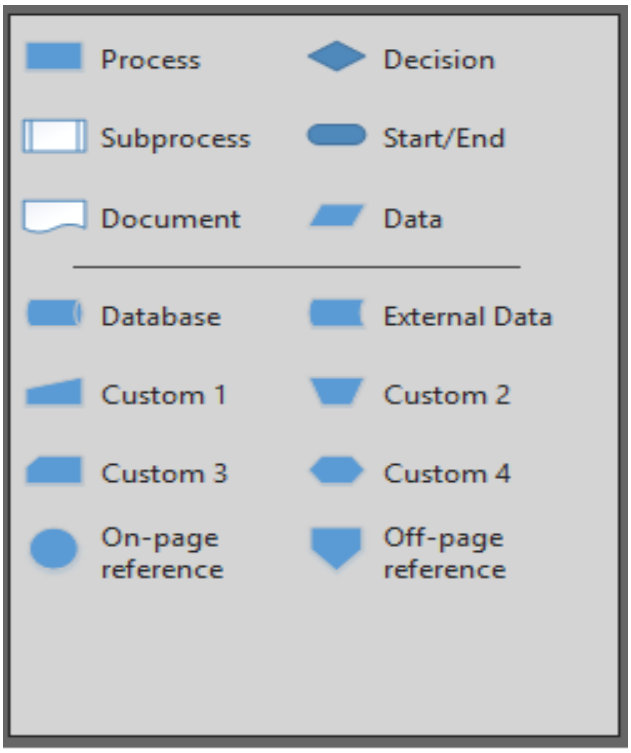


Figure D.1. Flowcharts shapes explanation