Acknowledgement

We would like to thank our guide, Prof. Sunil Surve, Head of Department of Computer Engineering, for encouraging us to work in this domain (Robotics) and helping us throughout the duration of the project.

We would also like to thank Prof. Kavi Arya (CSE Department, IIT Bombay) and the entire RA staff at the Embedded Real Time Systems Lab (IIT, Bombay), for assisting us and more importantly, for conducting the E-yantra Project – an initiative that encourages students to explore the exciting field of Robotics.

Abstract

This project has multiple facets - not just in implementation - but also in its purview of Engineering. It encompasses several sub domains of Electrical and Computer Engineering.
Therefore, multiple approaches for development and implementation exist. It is therefore of statutory importance to note that, a lot of energy (which culminates into a lot of time), is expended in lesser tangible activity such as reading up underlying concepts, going through fundamentals of a relatively new field (with regard to personal experience) and learning completely new frameworks, tools and other necessities, which more often than not, go unnoticed in the final scheme of things. A lot of high-level, pen-and-paper concepts need to be realized. These go down with varying limits of error and randomness.

We started off by constructing a robotic platform, one designed to suit our approach, specifically, that of making a robot on which we could mount a Kinect sensor device as well as a Laptop computer and keeping the entire set up mobile.
Therefore, the first phase of the project was devoted in its entirety to the putting together of hardware: Designing and modeling the robot, planning and making a list of requirements for its realization, along with the price estimates of each component, keeping in mind a modest financial credit.

This was followed up by an active phase of purchasing the equipment and studying the manuals of all the components. After this, came the relatively long period of integrating the hardware.

Next, we studied the Robot Operating System, the brain metaphor for our robot. ROS, a sophisticated robot framework, is as beneficial as it is complex. However, a lot of driver level programming and hardware abstraction, which seemed beyond the scope of our project was achieved by ROS.

We carried the process forward by studying various topics in probabilistic robotics such as Particle Filters, Kalman Filters and Monte Carlo Localization.

Table of Contents

Table of Notation:

slam	*simultaneous localization and mapping*
$\mathbf{s}_t$	*pose of the robot at time t*
θ_n	*position of the n-th landmark*
Θ	*set of all n landmark positions*
$\mathbf{z}_t$	*sensor observation at time t*
$\mathbf{z}$	*set of all observations* $\{z_1,\ldots,z_t\}$
$\mathbf{u}_t$	*robot control at time t*
$\mathbf{u}$	*set of all controls* $\{u_1,\ldots,u_t\}$
$\mathbf{w}^{[m]}_t$	*Importance weight of the m-th particle*

1. Project Overview

1.1 Introduction and Motivation

The post-industrialized world is already in a labor crisis, where a very low percentage of its population regards labor intensive tasks as acceptable professions. Much of the developing world will be at a similar stage in a few decades.
Case in point, Sweden: with an end-of-history type economy, all of Sweden's unskilled workforce (construction, Metropolis maintenance and cleaning, domestic help, etc.) comprise of foreign Diaspora. This, however, as evident, results in a complex Socio-cultural anthropological state. The same is true for the rest of Scandinavia, parts of Europe, North America and Japan.

With a flattening growth in population and rise in mid-income figures in developing countries, it is about time that Robotics and Artificial Intelligence technology is actively dispatched in everyday, human inhabited environments.

The truth is, however, that AI (although successfully implemented in Industrial automation), is faced with near-intractable challenges in the world at large.
Probabilistic Robotics, Computer Vision, Machine Learning and a versatile Microprocessor and Microcontroller industry (fuelled by Moore's Law) has shown promise in solving an eclectic set of real-world problems. However, a standard, economical solution for everyday use is yet to be realized.

1.2 SLAM - An informal introduction

A Robot capable of carrying out Simultaneous Localization and Mapping (SLAM), is essential in generating a view (example, a map) of an unknown (lacking a priori knowledge) environment.

The process would enable the robot to learn about its surroundings, in terms of path (the movable areas in the environment), Landmarks (any object that exists in the environment, for example, walls, equipment, furniture, etc.).
This would, in turn, make the robot ready to carry out tasks, in an environment, which it is now relatively familiar to.
The tasks could range from simple observation of the existing Landmarks to more complex feature exploration, semantic object recognition and detection, path planning, object fetching, etc. The complexity of application proportionately necessitates a capable robot.

1.3 Problem Statement

1.3.1 Simultaneous Localization and Mapping - Explained

Consider a mobile robot moving through an unknown, static environment. The robot executes controls and collects observations of features in the world. Both the controls and the observations are corrupted by noise. Simultaneous Localization and Mapping, also known as SLAM is the process of recovering a map of the environment and the path of the robot from a set of noisy controls and observations. If the path of the robot were known with certainty, (using GPS for example), then mapping would be a straightforward problem. The positions of objects in the robot's environment could all be estimated using independent filters. However, when the path of the robot is unknown, error in the robot's path correlates errors in the map. As a result, the state of the robot and the map must be estimated simultaneously.

The problem of Simultaneous Localization and Mapping has attracted immense attention in the robotics literature. SLAM addresses the problem of a mobile robot moving through an environment of which no map is available a priori. The robot makes relative observations of its ego-motion and of objects in its environment, both corrupted by noise. The goal of SLAM

is to reconstruct a map of the world and the path taken by the robot. SLAM is considered by many to be a key prerequisite to truly autonomous robots.

If the true map of the environment were available, estimating the path of the robot would be a straightforward localization problem. Similarly, if the true path of the robot were known, building a map would be a relatively simple task. However, when both the path of the robot and the map are unknown, localization and mapping must be considered concurrently—hence the name Simultaneous Localization and Mapping.

If the path of the robot were known with certainty, (using GPS for example), then mapping would be a straightforward problem. The positions of objects in the robot's environment could all be estimated using independent filters. However, when the path of the robot is unknown, error in the robot's path correlates errors in the map. As a result, the state of the robot and the map must be estimated *simultaneously*.

The correlation between robot pose error and map error can be seen graphically:

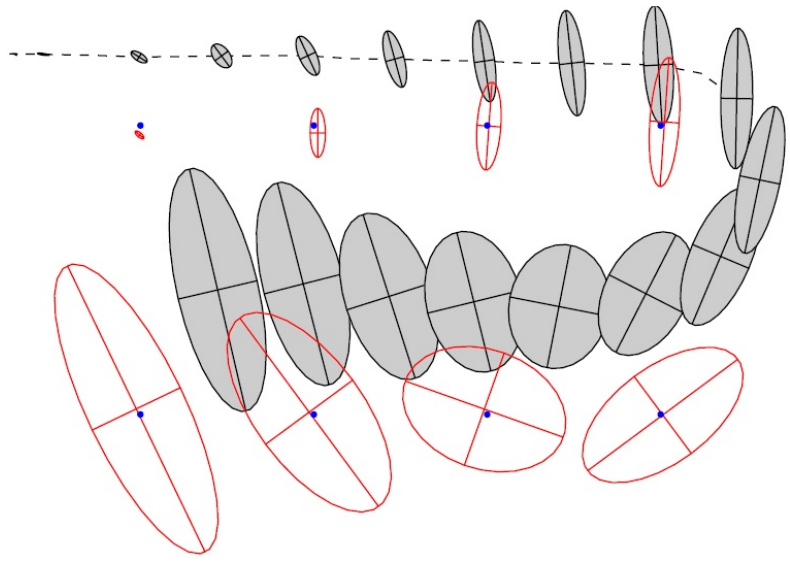


Figure 1.1: Before closing the loop: landmark uncertainty increases as robot pose uncertainty increases. Robot pose estimates over time are shown as shaded ellipses. Landmark estimates are shown as un-shaded ellipses.

A robot is moving along the path specified by the dashed line, observing nearby Landmarks, drawn as circles. The shaded ellipses represent the uncertainty in the pose of the robot, drawn over time. As a result of control error, the robot's pose becomes more uncertain as the robot moves. The estimates of the landmark positions are shown as un-shaded ellipses. Clearly, as the robot's pose becomes more uncertain, the uncertainty in the estimated positions of newly observed landmarks also increases.

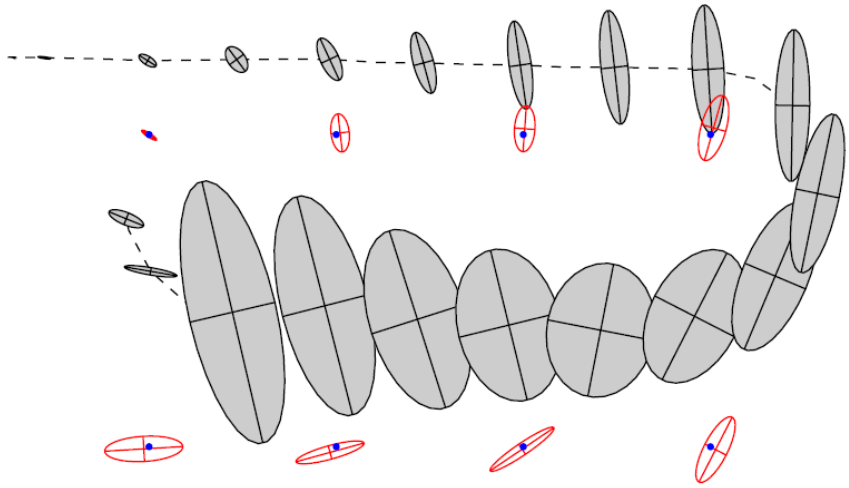


Figure 1.2: After closing the loop: Revisiting a known landmark decreases not only the robot pose uncertainty, but also the uncertainty of landmarks previously observed.

The robot completes the loop and revisits a previously observed landmark. Since the position of this first landmark is known with high accuracy, the uncertainty in the robot's pose estimate will decrease significantly. This newly discovered information about the robot's pose increases the certainty with which past poses of the robot are known robot. Again, this is because of the correlated nature of the SLAM problem. Errors in the map are correlated through errors in the robot's path. Any observation that provides information about the pose of the robot will necessarily provide information about all previously observed landmarks [2].

1.3.2 SLAM - A joint estimation

The chicken-or-egg relationship between localization and mapping is a consequence of how errors in the robot's sensor readings are corrupted by error in the robot's motion. As the robot moves, its pose estimate is corrupted by motion noise. The perceived locations of objects in the world are, in turn, corrupted by both measurement noise and the error in the estimated pose of the robot. Unlike measurement noise, however, error in the robot's pose will have a systematic effect on the error in the map. In general, this effect can be stated more plainly; error in the robot's path correlates errors in the map. As a result, the true map cannot be estimated without also estimating the true path of the robot.

Figure: 1.3 Map built without correcting robot path

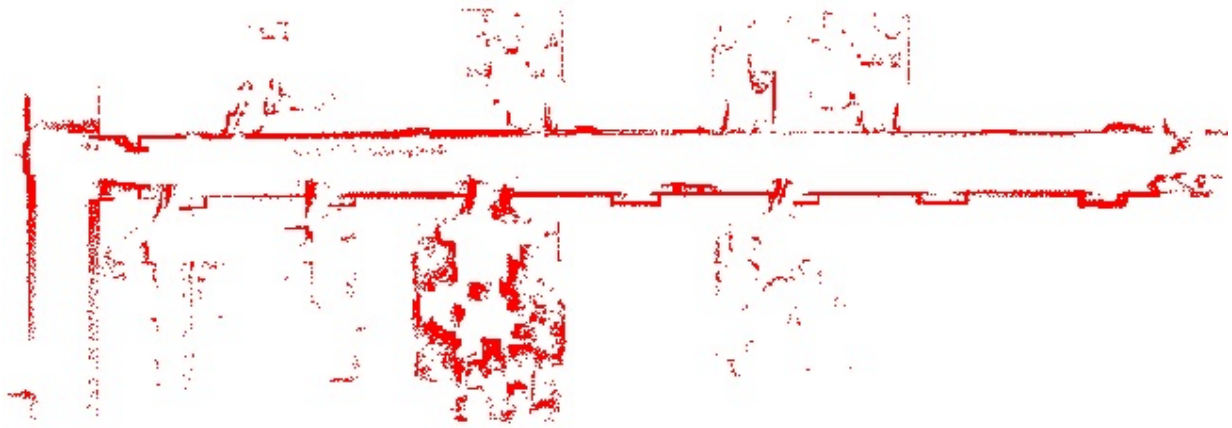


Figure 1.4: Map built using SLAM

The above figures (1.3 and 1.4) show a set of laser range scans collected by a mobile robot moving through a typical office environment. The robot generates estimates of its motion using wheel encoders. In Figure 1.3, the laser scans are plotted with respect to the path of the robot as measured by the encoders. Clearly, as error accumulates in the estimated pose of the robot, the overall map becomes increasingly inaccurate. Figure 1.4 shows the laser readings plotted according to the path of the robot as reconstructed by a SLAM algorithm [2].

1.3.3 Posterior Estimation

According to the standard formulation of the SLAM problem, a robot executes controls and accumulates observations of its environment, both corrupted by noise. Each control or observation, coupled with an appropriate noise model, can be thought of as a probabilistic constraint. For example, each control probabilistically constrains two successive poses of the robot. Observations, on the other hand, constrain the relative positions of the robot and objects in its environment. As the network of constraints expands, new observations can be used to update not only the current map feature and robot pose, but also map features that were observed in the past. An example of the constraints imposed by observations and controls is shown in Figure 1.3.

Initially, these constraints may be very uncertain. However, as objects in the map are observed repeatedly, the constraints become increasingly rigid. In the limit of infinite observations and controls, the positions of all map features will become fully correlated. The primary goal of SLAM is to estimate this true map and the true pose of the robot, given the set of observations and controls currently available.

The pose of the robot at time t will be denoted s^t. For robots operating in a planar environment, this pose consists of the robot's x-y position in the plane and its heading direction. The complete trajectory of the robot, consisting of the robot's pose at every time step, will be written as s^t:

$s^t = \{s_1, s_2 \ldots s_t\}$

Another assumption is that the robot's environment can be modeled as a set of N immobile, point landmarks. Point landmarks are commonly used to represent the locations of features extracted from sensor data, such as geometric features in a laser scan or distinctive visual features in a camera image. The set of N landmark locations will be written $\{\theta_1 \ldots \theta_N\}$. For notational simplicity, the entire map will be written as Θ.

As the robot moves through the environment, it collects relative information about its own motion. This information can be generated using odometers attached to the wheels of the robot, inertial navigation units, or simply by observing the control commands executed by the robot. Regardless of origin, any measurement of the robot's motion will be referred to generically as a control. The control at time t will be written u_t. The set of all controls executed by the robot till time t are written as $u_t = \{u_1, u_2 \ldots u_t\}$

As the robot moves through its environment, it observes nearby landmarks. In the most common formulation of the planar SLAM problem, the robot observes both the range and bearing to nearby obstacles. The observation at time t will be written z_t. The set of all observations collected by the robot will be written z_t. Set of all observations written as:

$z_t = \{ z_1, z_2 \ldots z_t\}$.

It is commonly assumed in the SLAM literature that sensor measurements can be decomposed into information about individual landmarks, such that each landmark observation can be incorporated independently from the other measurements. This is a realistic assumption in virtually all successful SLAM implementations, where landmark

features are extracted one-by-one from raw sensor data. Thus, we will assume that each observation provides information about the location of exactly one landmark θ*n* relative to the robot's current pose s_t.

The variable *n* represents the identity of the landmark being observed. In practice, the identities of landmarks usually cannot be observed, as many landmarks may look alike. The identity of the landmark corresponding to the observation z_t will be written as n_t, where $n_t = \{1 \ldots \ldots N\}$. The set of all data associations can be written as $n_t = \{n_1, n_2 \ldots n_t\}$.

Again for simplicity, I will assume that the robot receives exactly one measurement *zt* and executes exactly one control *ut* per time step. Multiple observations per time step can be processed sequentially, but this leads to a more cumbersome notation. Using the notation defined above, the primary goal of SLAM is to recover the best estimate of the robot pose *st* and the map Q, given the set of noisy observations *zt* and controls *ut.* In probabilistic terms, this is expressed by the following posterior, referred to in the future as the SLAM posterior:
$p(s_t, \Theta \mid z^t, u^t)$.

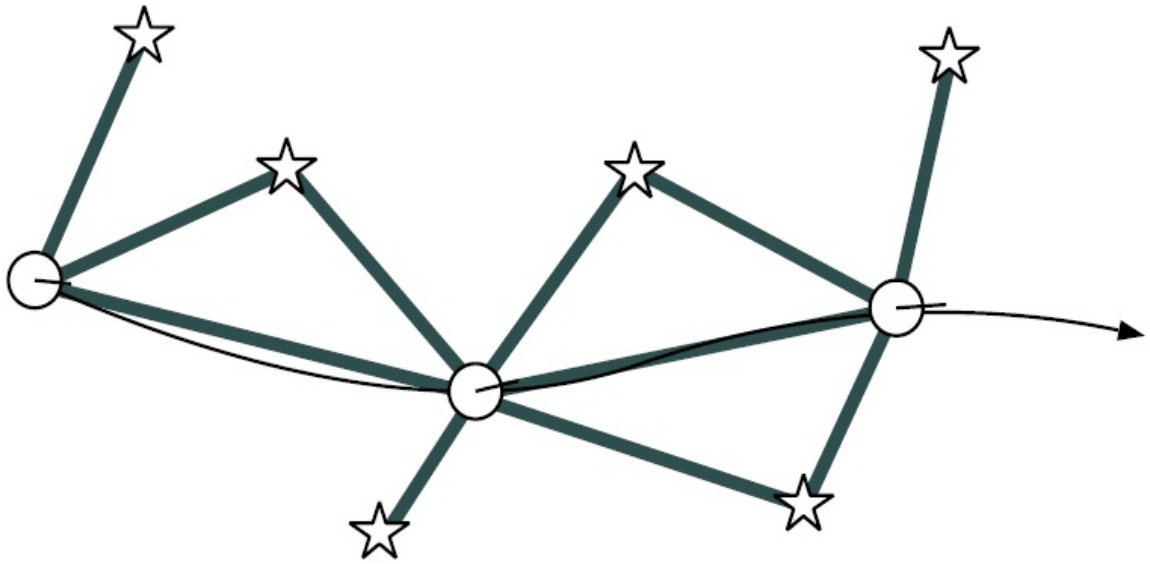


Figure 1.5: Observations and controls form a network of probabilistic constraints on the pose of the robot and the relative positions of features in the robot's environment. These constraints are shown in the figure as dark lines.

Observations and controls form a network of probabilistic constraints on the pose of the robot and the relative positions of features in the robot's environment. These constraints are shown in Figure 1.5 as dark lines.

The most popular online solutions to the SLAM problem attempt to estimate a posterior probability distribution over all possible maps and all possible robot poses, given the sensor readings accumulated by the robot. This distribution, called the SLAM posterior, can be written as: $p(s_t, \theta \mid z^t, u^t, n^t)$ where s_t is the current pose of the robot and is the map. The posterior is conditioned on the set of all sensor readings z^t, controls u^t and data associations n^t. The data associations n^t describe the mapping of observations z^t to features in θ.

Any parameterized model can be chosen to represent the map θ, however it is commonly assumed to be a collection of point features. This model assumes that the robot's environment can be represented as a collection of points, also known as "landmarks," relative to some external coordinate system. In a real implementation, these point landmarks may correspond to the locations of features extracted from sensors, such as cameras, sonar, and laser range-finders. Throughout this document I will assume the point landmark representation, though other representations can be used. Higher order geometric features, such as line segments, have also been used to represent maps in SLAM.

Posterior estimation is desirable because, in addition to returning the most probable map and robot path, it also estimates the uncertainty with which each quantity is known. Posterior estimation has several advantages over solutions that consider only the most likely state of the world. First, considering a distribution of possible solutions, leads to more robust algorithms in noisy environments. Second, uncertainty can be used to evaluate the relative information conveyed by different components of the solution. One section of the map may be very uncertain, while other parts of the map are well known.

The following recursive formula, known as the Bayesian Filter, can be used to compute the SLAM posterior at time t, given the posterior at time t −1:

$$p(s_t ; \Theta \mid z^t, \quad u^t, \quad n^t) = \eta p(z_t \mid s_t, \Theta, n_t) \int p(s_t \mid s_{t-1}, u_t) p(s_{t-1}; \Theta \mid z^{t-1}, u^{t-1}, n^{t-1}) ds_{t-1}$$

In general, the integral above cannot be evaluated in closed form. However, this function can be computed by assuming a particular form for the posterior distribution. Many statistical estimation techniques, including the Kalman filter and the particle filter, are simply approximations of the general Bayesian Filter [2].

2. Literature Surveyed

We started off by getting ourselves acquainted to the problem of Localization and Mapping. From our reading of *SLAM for dummies by Søren Riisgaard and Morten Rufus Blas,* we got a good introduction to SLAM.

Localization of robot is the problem of finding making the robot realize where it is in a given environment. 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. SLAM consists of multiple parts; Landmark extraction, data association, state estimation, state update and landmark update. There are many ways to solve each of the smaller parts. SLAM is applicable for both 2D and 3D motion. The paper considered only 2D SLAM. The SLAM process consists of a number of 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. The paper goes on the explain all the mathematics behind implementing SLAM using extended Kalman Filer.

FastSLAM: A factored solution to SLAM by Montemerlo, et al. helped us widen our understanding of the SLAM problem. It mentioned several applications for SLAM and gave us impetus to carry on.

SLAM papers are very theoretic and primarily focus on innovations in small areas of SLAM, which of course is their purpose. Therefore, we had to look at more practical sources to learn about implementation. *Prof. Sebastian Thrun's* online course, *CS373: Programming a robotic car (udacity.com)*, gave us a good exposure to a practical approach of SLAM. Albeit they were simulations, we programmed different aspects to the Localization process including MCL, EKF and particle filters. This is when we got better understanding of how Bayes and Total probability are used to evaluate state of robot in a map. We also learnt about graphSLAM through this very innovative online course.

Now we had to bridge the gap between simulation and actual implementation. We developed the robotic platform by looking at various models. *Development of a Modular Mobile Robot Platform by Rached Dhaouadi and Mohamad Sleiman, IEEE Industrial Electronics, December 2011.* Their differential drive robot, along with their Kinematic analysis of the robot became our basis.

We derived our circuits by referring to various schematics online, including *Arduino.cc*.

Our early exposure to the *Robot Operating System* (*ros.org)*, gave us a proper implementation level view of SLAM. This is an implementation framework, and it helps in various practical aspects such as interfacing sensors, microcontroller programming and basically realizing high level concepts. Since our robot closely resembles *Turtlebot by Willow Garage,* we looked at their extensive online documentation on *ros.org.* It gave us access to reusable stacks such as *pointcloud_to_laserscan, slam_gmapping and rgbd slam, visualizations, etc.*

3. Requirement Analysis

3.1 Hardware Requirements

The hardware of the robot is quite important. To do SLAM there is the need for a mobile robot and a range measurement device. The mobile robots we consider are wheeled indoor robots. This documents focus is mainly on software implementation of SLAM and does not explore robots with complicated motion models (models of how the robot moves) such as humanoid robots, autonomous underwater vehicles, autonomous planes, robots with weird wheel configurations etc. We here present some basic measurement devices commonly used for SLAM on mobile robots [1].

3.1.1 Robot

Important parameters to consider are ease of use, odometry performance and price. The odometry performance measures how well the robot can estimate its own position, just from the rotation of the wheels. The robot should not have an error of more than 2 cm per meter moved and 2° per 45° degrees turned. Typical robot drivers allow the robot to report its (x, y) position in some Cartesian coordinate system and also to report the robots current bearing/heading.

3.1.2 A Range measurement device

The range measurement device used is usually a laser scanner nowadays. They are very precise, efficient and the output does not require much computation to process. On the downside they are also very expensive. A SICK scanner costs about 5000USD. Problems with laser scanners are looking at certain surfaces including glass, where they can give very bad readings (data output). Also laser scanners cannot be used underwater since the water disrupts the light and the range is drastically reduced. Second there is the option of sonar. Sonar was used intensively some years ago. They are very cheap compared to laser scanners. Their measurements are not very good compared to laser scanners and they often give bad readings. Where laser scanners have a single straight line of measurement emitted from the scanner with a width of as little as 0.25 degrees a sonar can easily have beams up to 30 degrees in width. Underwater, though, they are the best choice and resemble the way dolphins navigate. The type used is often Polaroid sonar. It was originally developed to

measure the distance when taking pictures in Polaroid cameras. Sonar has been successfully used in. The third option is to use vision. Traditionally it has been very computationally intensive to use vision and also error prone due to changes in light. Given a room without light a vision system will most certainly not work. In the recent years, though, there have been some interesting advances within this field. Often the systems use a stereo or triclops system to measure the distance. Using vision resembles the way humans look at the world and thus may be more intuitively appealing than laser or sonar. Also there is a lot more information in a picture compared to laser and sonar scans. This used to be the bottleneck, since all this data needed to be processed, but with advances in algorithms and computation power this is becoming less of a problem. Vision based range measurement has been successfully used.

Given the expensive nature of ranging devices such as LIDARs and the error proneness of cheaper devices like SONARs and stereo vision, we decided to choose the XBOX 360 Kinect sensor as a ranging device. It strikes a perfect balance between price and performance. It comes at a cost of US $180 and has a suitable working range of 180-200 cm[1].

3.2 Mobile Robots – A Kinematic analysis

Figure 3.1: An assortment of Mobile robots

The two mobile robot types are Hilare-type and car-like mobile robots. Hilare-type robots have two independently driven wheels as the drive mechanism and are usually balanced by a

passive caster wheel. They have good maneuvering abilities, e.g., a zero minimum turn radius, and are easier to control. They are also easier to build due to their simple drive mechanism.

Car-like mobile robots, as their name implies, have a drive mechanism similar to cars. They are driven by a single motor that powers a differential, which in turn distributes the motor's torque to the rear wheels. They have a steering mechanism at the front wheel(s), which is driven by a motor to generate steering angles to steer the robot. Car-like robots have a nonzero minimum turn radius. The nonzero minimum turn radius limits the maneuvering ability of carlike robots. Care must be taken when defining a desired path such that the minimum radius of curvature of the path is not less than the minimum turn radius of the car-like robot. Otherwise, the robot will not be able to follow that path correctly independent of the controller that is being used for the trajectory tracking.

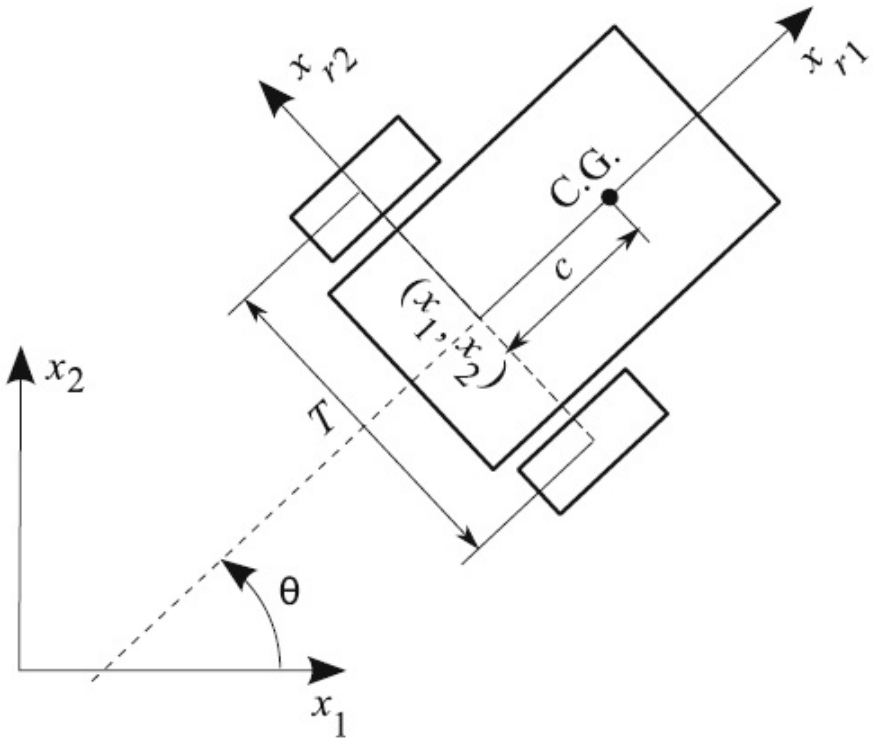


Figure 3.2:A Hilare-type Mobile robot

A schematic figure of a Hilare mobile robot is shown in Figure 3.2. This type of robot is mostly used for indoor applications. The drive mechanism of a Hilare-type robot has two independent motors. Each of these motors power one of the robot's wheels. Thus, the actual kinematic inputs that drive the robot and affect its speed and direction of motion are the two wheel speeds. With this in mind, at first glance it seems intuitive to write the kinematic equations of motion of a Hilare mobile robot in terms of these speeds. However, on most commercial mobile robots, there exists a low-level controller that controls the linear and angular velocity of the robot. Therefore, for application purposes, it is more convenient to choose the linear and angular velocity of the mobile robot as the inputs of the kinematic model. When a control law is found later based on this model, it can be more easily applied using the development packages available for commercial robots. Now, consider the Hilare-type mobile robot as shown. Assume that the robot motion is reasonably slow such that the longitudinal traction and lateral force exerted on the robot's tires do not exceed the maximum static friction between the tires and the floor in the longitudinal and lateral directions. In other words, assume that no-slip happens between the robot's tire and the floor during the whole motion of the robot. The first direct result of this assumption is that the velocities of the center of the robot's wheels do not have any lateral components. As a consequence, one can assume that the velocity of point (x1, x2), the midpoint of the line attaching the center of the wheels, does not have any lateral component and is parallel with the wheel planes. The second result of the no-slip assumption is that one can relate the velocity of point (x1, x2), the midpoint of the line attaching the center of the wheels, to the rotational velocity of the wheels. Before writing the kinematic equations of motion for the robot, one has to define the configuration variables of the robot. Let the coordinates of point (x1, x2) define the global position of the robot with respect to the inertial coordinate system x1−x2. Consider a line that is perpendicular to the wheel axis and goes through the point (x1, x2) as an orientation reference for the robot. The angle that this line makes with the positive x1 axis, θ, represents the orientation of the robot. The three variables that define the geometrical configuration of the robot at any given time are:

$$q = \begin{bmatrix} x1 \\ x2 \\ \theta \end{bmatrix}$$

Assume that the point (x1, x2) on the robot moves with a linear speed of v, while the robot has an angular velocity of ω. Now, one can use the first direct result of the no-slip assumption and write the velocity components of the point (x1, x2) in the inertial frame as:

$$vx = vcos\theta$$

$$vy = vsin\theta$$

Also, the rate of change of the robot's orientation is:

$$\omega = \frac{d\theta}{dt} = (vr - vl)/b$$

where vr and vl are right and left velocities (obtained from the respective optical sensors) and b is the distance between the centers of the right and left wheels.

$$\dot{\mathbf{q}} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} u$$

Where,

$$u = [v \quad \omega]$$

Once the input vector u is known as a function of time, it can be numerically integrated to predict the motion of the robot. Note that one can choose inputs different than the ones that are used here. Example of other sets of inputs are the rotational velocity of the wheels, the linear velocity of the point (x1, x2) and the difference of the linear velocity of the wheels, etc [3].

3.3 Coordinate Transformations and Frames

A Transform Tree defines offsets in terms of both translation and rotation between different coordinate frames. We used tf-a tranform library provided by ROS for coordinate transformations

To define and store the relationship between two frames using tf, we need to add them to a transform tree. Conceptually, each node in the transform tree corresponds to a coordinate frame and each edge corresponds to the transform that needs to be applied to move from the current node to its child. Tf uses a tree structure to guarantee that there is only a single

traversal that links any two coordinate frames together, and assumes that all edges in the tree are directed from parent to child nodes.

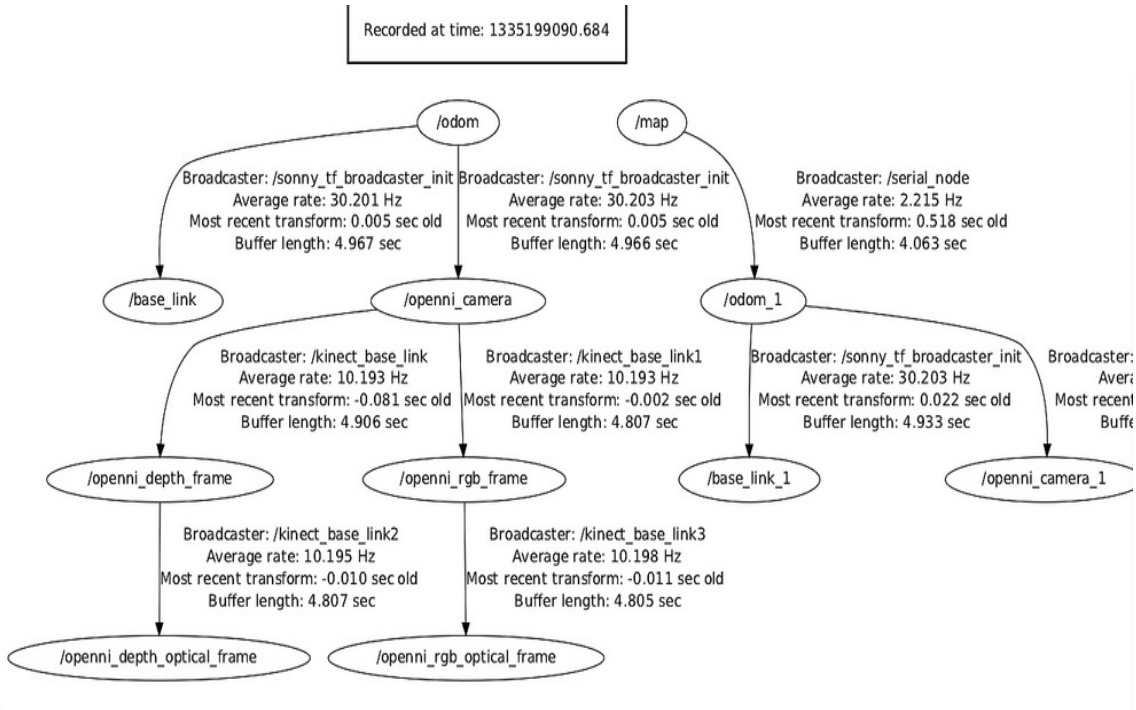


Figure 3.3: Transform tree generated using tf API.

4. Project Design

4.1 Designing the Robot

In making the design of the robot, we had to consider the following constraints which greatly influenced the final outcome of the robot model:

- Mobile
- Laptop mountable
- Kinect mountable
- Differential Drive for simplicity

We created the model on Google Sketch Up.

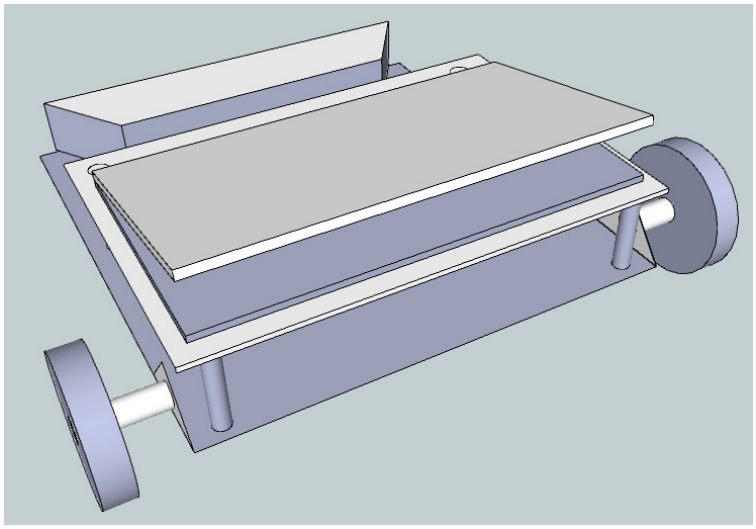


Figure 4.1: Auxiliary view: Back

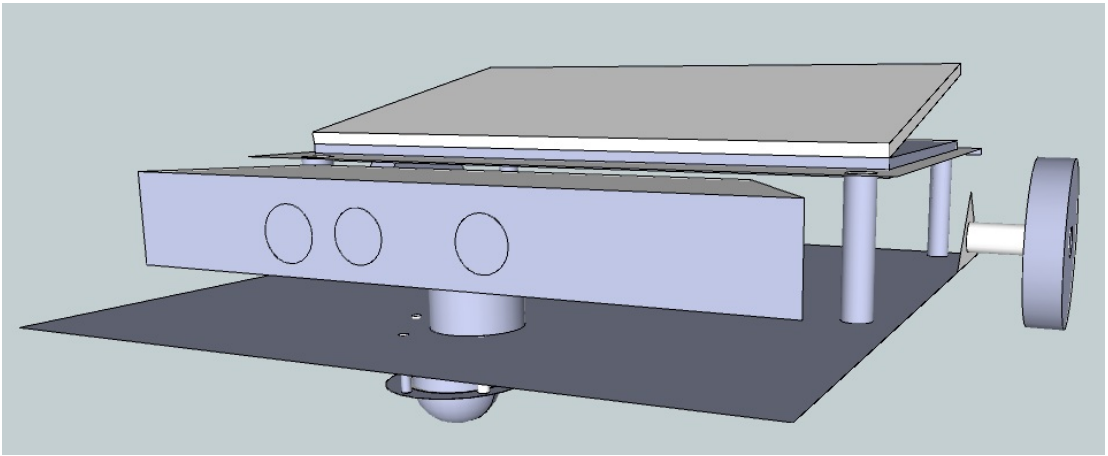


Figure 4.2: Auxiliary view: Front

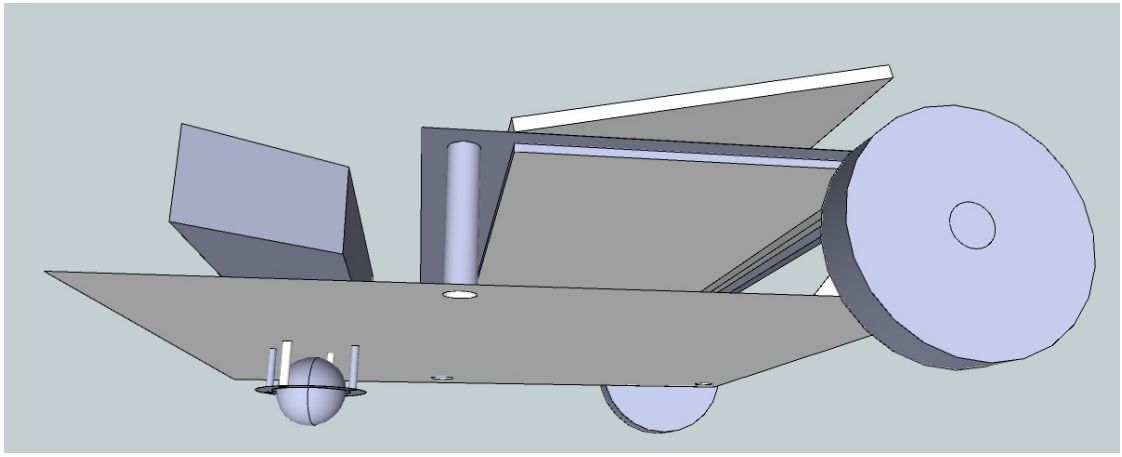


Figure 4.3: Auxiliary view: Side

Figures 4.1-4.3 show screen shots of the Google Sketch Up robot model.

4.2 Robot components architecture

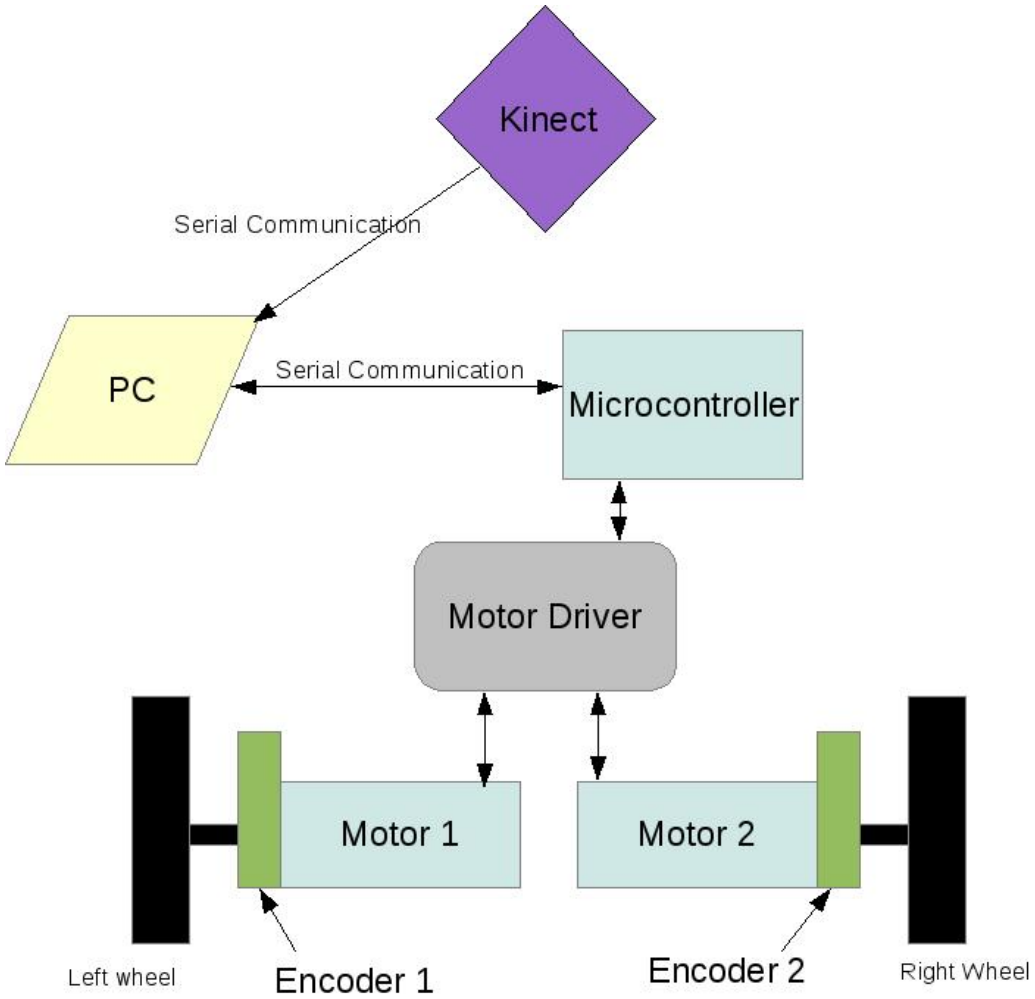


Figure 4.4: Logical architecture of the robot

The robot is a collection of highly heterogeneous components, connected to each other with a set of heterogeneous interfaces. Understanding how each of these components are connected and their overall "cog in the wheel" understanding is essential. Figure 4.4 is a simple representation of an otherwise complicated set of components and interfaces, and elicits greater understanding.

The left and right wheels and encoders are physically connected to the left and right motors respectively. The motors are connected to the motor driver using electric connectors. Further, the motor driver is interfaced with the microcontroller (mounting). The Kinect and the microcontroller are interfaced with the PC (which is a Laptop) via a USB serial interface.

4.3 Robot software architecture

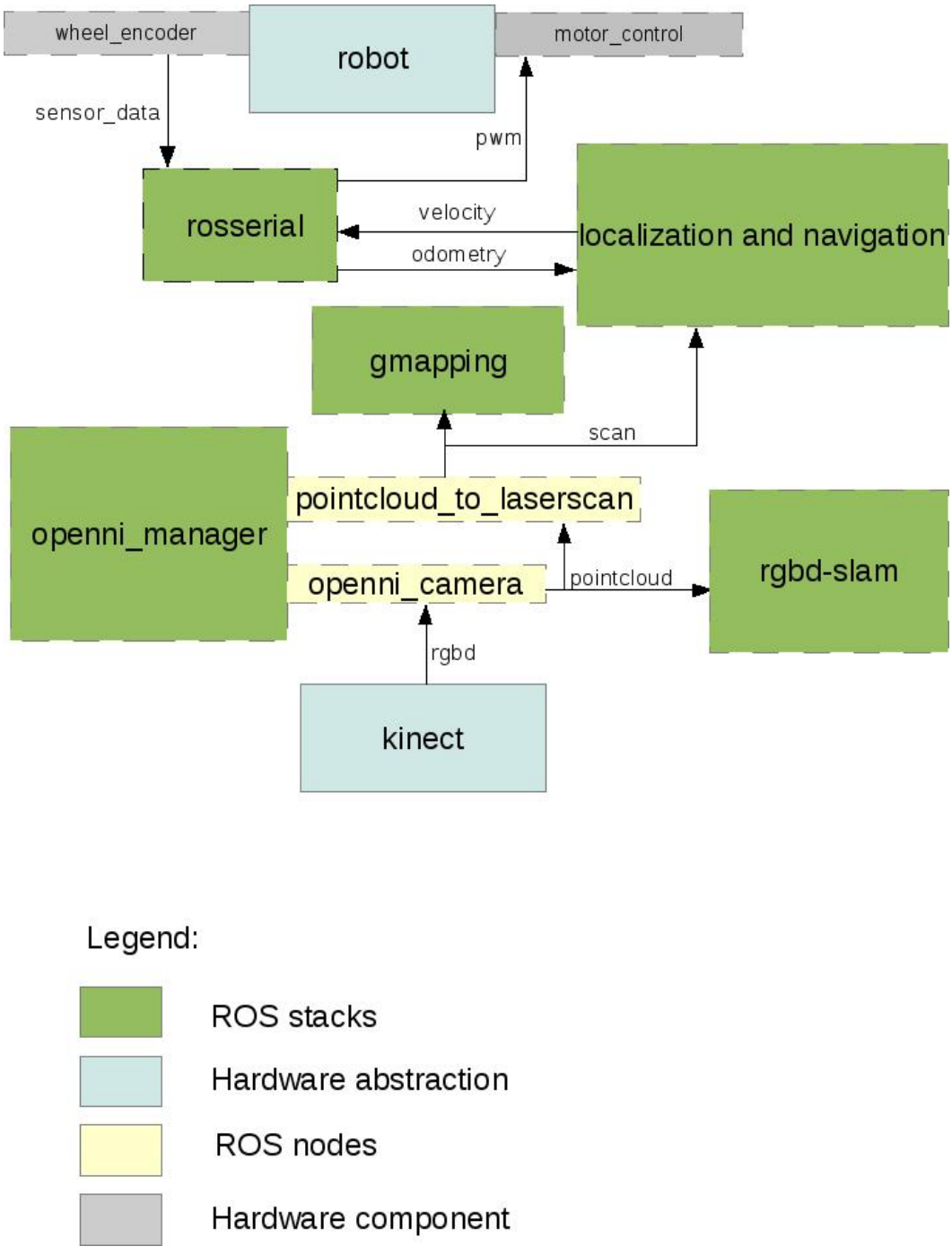


Figure 4.5: ROS nodes, stacks and topics

An early decision to use the ROS framework in the project enabled us to design a ROS based software/package architecture. This helps in understanding of the passage and computation of

data streamed by the sensor devices as well as the control signals passed to the actuators, as shown in Figure 4.5. It represents Hardware components (such as the Kinect) in an abstracted way and only specifies the type of data published or subscribed by the components.

The ROS stacks interface with the Hardware using special libraries.

5. Implementation Details

5.1 Robot Components with cost

Table 5.1: Components list, quantity and pricing

Component	**Quantity**	**Sub-total (Rs.)**
Arduino Uno	1	1510 (1360+150)
Xbox 360 Kinect	1	9990
Platform I, II	2	390 (140+250)
60 rpm DC Motors	2	1100 (2*550)
Wheels	2	60 (2*30)
Castor Wheel (large)	1	20
Sensor (encoders)	2	300 (2*150)
Motor-driver (L298)	1	550
Digital compass	1	1600
Battery Li-ion (12V)	1	713 (563+150)
Battery Pb-acid (12V)	1	350
Clamps	2	60 (2*30)
	Total	**16643**

5.2 Robot assembly

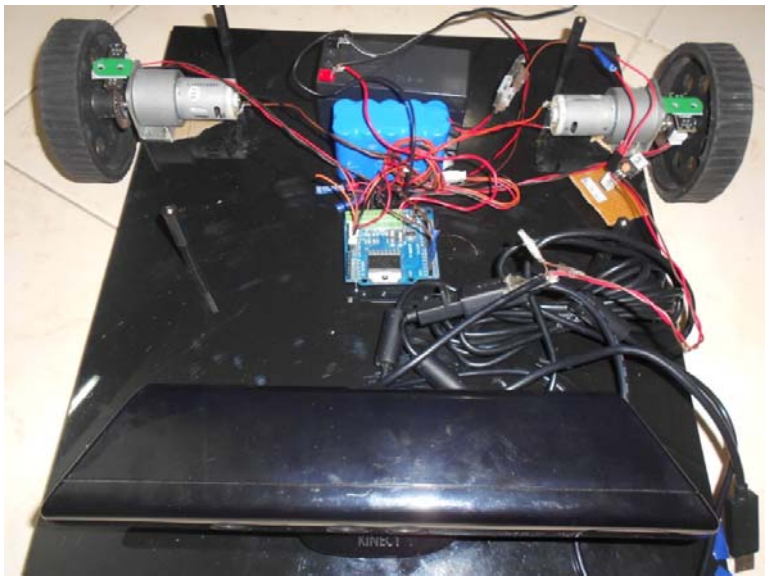


Figure 5.1: Top-view

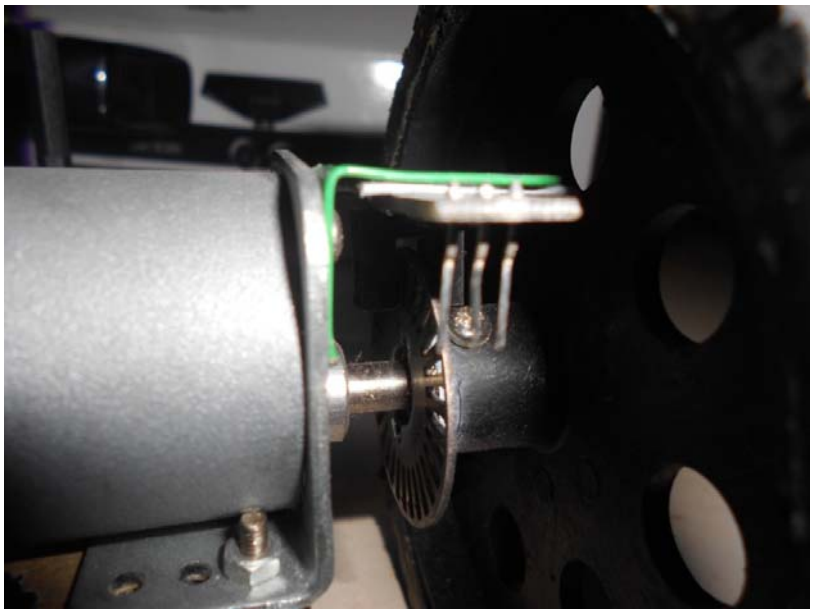


Figure 5.2: Motors and Wheel encoder

5.3 Overall circuit diagram of robot

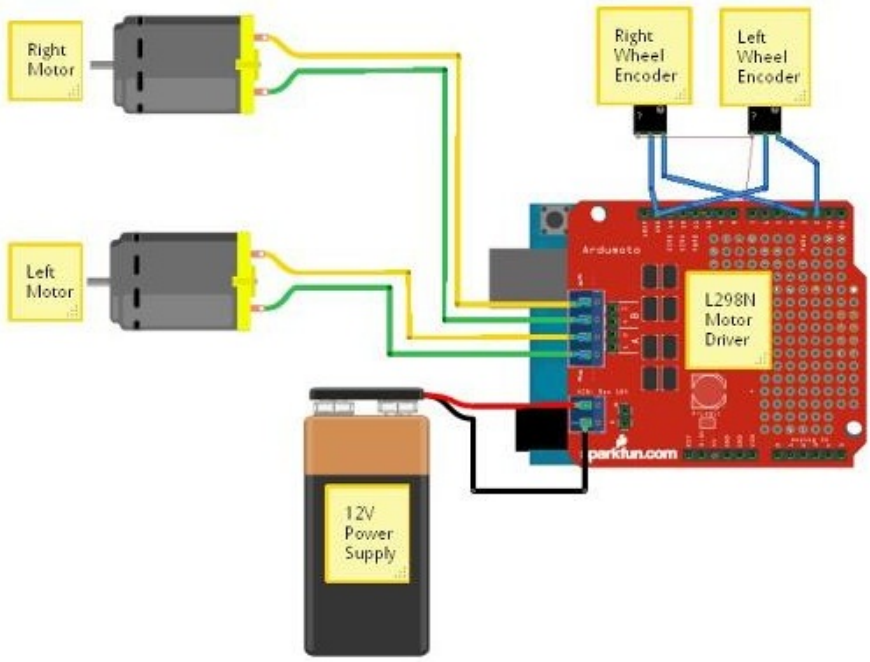


Figure 5.4: Overall circuit diagram of robot developed using Frizting (*http://fritzing.org/*)

The robot circuit is physically separated over two distinct components. Figure 5.4 shows the circuitry of the Driving mechanism, including odometry. The motors are connected to the motor driver L298N using insulated copper wires (soldered with the motor pins), on both the terminals, positive and negative. The L298N is powered using a 12V Li-ion battery. The L298N is mounted on the Arduino Uno Microcontroller. Further, the Left and Right encoders are inputs to the Uno, powered by Vcc from the Uno itself.

The other circuit is simple, and consists of a Kinect device customized to work with a 12V regulated power supply from a battery.

5.4 Adding the Kinect to a Robot

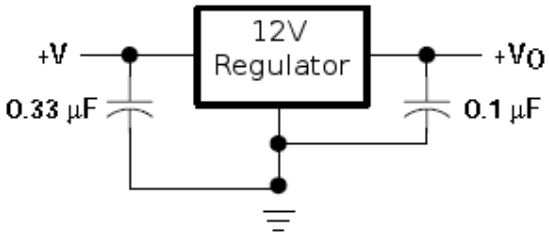


Figure 5.4 : Voltage Regulator Circuit Diagram

Solder the components onto the prototype board.

Connecting to the Kinect

Take the Kinect USB and power splitter cable and cut the cable connected to the power adapter about 10 inches from the splitter. Inside the cable there is a brown and white wire, the brown is +12V and the white GND. Confirm Voltages by plugging the wall adapter into a wall and using a digital multimeter.

Figure 5.5: Kinect wire

5.5 Algorithms:

Algorithm1: slam_process

---

required*:*

frames **-**

o: *odometry frame, frame of odometry data*

b: *base link frame, frame of a central, representative point*

m: *map frame, represents the global frame*

k: *kinect frame, frame of Kinect sensor*

transforms **-**

$^{o}\mathbf{T_b}$: odometry to base link transform

$^{b}\mathbf{T_k}$: Kinect frame to base link transform

$\mathbf{S_t}$: *pose at t, given by odometry publisher*

$\mathbf{Z_t}$: *most recent laser scan*

while (true **) do**

slam_gmapping(S_t, Z_t)

update map

// *map to base link transformation*

publish $^{m}T_b$

end while

Algorithm 2: slam_gmapping

Require:

$\mathbf{S_{t-1}}$, *the sample set of the previous time step*

$\mathbf{z_t}$, *the most recent laser scan*

$\mathbf{u_{t-1}}$, *the most recent odometry measurement*

Ensure:

$\mathbf{S_t}$, *the new sample set*

$S_t = \{\}$

for all $s^{(i)}_{t-1} \in S_{t-1}$ **do**

$< x^{(i)}_{t-1}, w^{(i)}_{t-1}, m^{(i)}_{t-1} >= s^{(i)}_{t-1}$

// *scan-matching*

$x'^{(i)}_t = x^{(i)}_{t-1} \oplus u_{t-1}$

$\dot{X}^{(i)}_t = \text{argmax}_x\, p(x \mid m^{(i)}_{t-1}, z_t, x'^{(i)}_t)$

if $\dot{X}^{(i)}t$ = failure **then**

$x^{(i)}_t \sim p(x_t \mid x^{(i)}_{t-1}, u_{t-1})$

$w^{(i)}_t = w^{(i)}_{t-1} \cdot p(z_t \mid m^{(i)}_{t-1}, x^{(i)}_t)$

else

// *sample around the mode*

for $k = 1, \ldots, K$ **do**

$x_k \sim \{x_j \mid |x_j - \dot{X}^{(i)}| < \Delta\}$

end for

// compute Gaussian proposal

$\mu^{(i)}_t = (0, 0, 0)^T$

$\eta^{(i)} = 0$

for all $x_j \in \{x_1, \ldots, x_K\}$ **do**

$\mu^{(i)}_t = \mu^{(i)}_t + x_j \cdot p(z_t \mid m^{(i)}_{t-1}, x_j) \cdot p(x_t \mid x^{(i)}_{t-1}, u_{t-1})$

$\eta^{(i)} = \eta^{(i)} + p(z_t \mid m^{(i)}_{t-1}, x_j) \cdot p(x_t \mid x^{(i)}_{t-1}, u_{t-1})$

end for

$\mu^{(i)}_t = \mu^{(i)}_t / \eta^{(i)}$

$\Sigma^{(i)}_t = 0$

for all $x_j \in \{x_1, \ldots, x_K\}$ **do**

$$\Sigma^{(i)}_t = \Sigma^{(i)}_t + (x_j - \mu^{(i)})(x_j - \mu^{(i)})^T \cdot p(z_t \mid m^{(i)}_{t-1}, x_j) \cdot p(xj \mid x^{(i)}_{t-1}, u_{t-1})$$

end for

$\Sigma^{(i)}_t = \Sigma^{(i)}_t / \eta^{(i)}$

// sample new pose

$x^{(i)}_t \sim \mathcal{N}(\mu^{(i)}_t, \Sigma^{(i)}_t)$

// update importance weights

$w^{(i)}_t = w^{(i)}_{t-1} \cdot \eta^{(i)}$

end if

// update map

$m^{(i)}_t = \text{integrateScan}(m^{(i)}_{t-1}, x^{(i)}_t, z_t)$

// update sample set

$S_t = S_t \cup \{< x^{(i)}_t, w^{(i)}_t, m^{(i)}_t >\}$

end for

$N_{eff} = 1/(\Sigma_{i=1}^{N} (\tilde{w}^{(i)})^2)$

if $N_{eff} < T$ then

$S_t = \text{resample}(S_t)$

end if

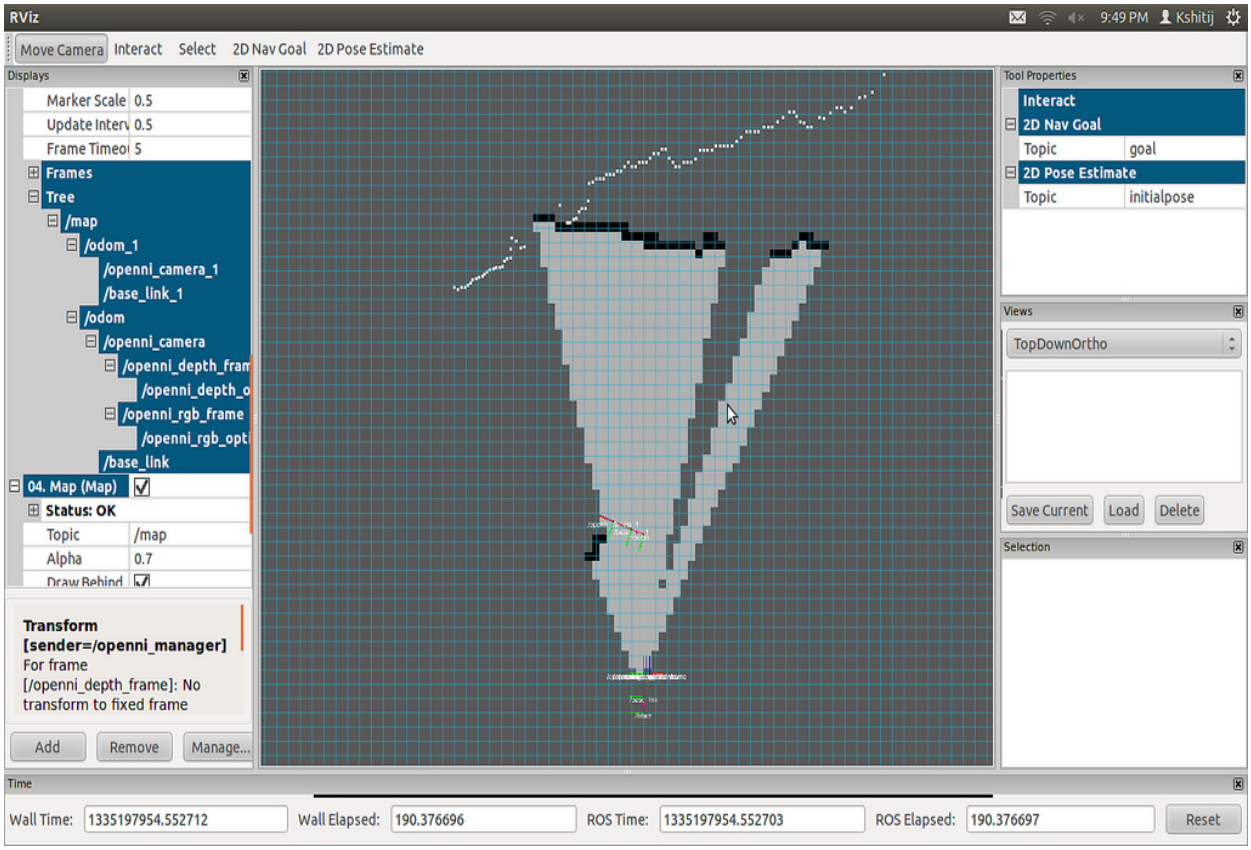


Figure 5.5: Map created using slam_gmapping

6. Technologies Used

6.1 Operating System

Ubuntu 11.10

Robot Operating System

6.2 Programming

C++

Arduino programming

6.3 Packages, Libraries and drivers

6.3.1 Drivers

OpenNI – Driver for Kinect

6.3.2 ROS Stacks/Packages

1. rgbd slam – Developed by Felix Endres, Juergen Hess, Nikolas Engelhard.

This package can be used to register the point clouds from RGBD sensors such as the Kinect or stereo cameras.

2. rosserial – Developed by Michael Ferguson, Adam Stambler.

ROS Serial is a point-to-point version of ROS communications over serial, primarily for integrating low-cost microcontrollers (Arduino) into ROS. ROS serial consists of a general p2p protocol, libraries for use with Arduino, and nodes for the PC/Tablet side (currently in both Python and Java).

3. pointcloud_to_laserscan – Developed by Tully Foote

Converts a 3D Point Cloud into a 2D laser scan. This is useful for making devices like the Kinect appear like a laser scanner for 2D-based algorithms (e.g. laser-based SLAM).

6.4 Hardware technology

Kinect – A device which gives RGB and range (distance) values.

Arduino Open Source Hardware – Microcontroller

6.5 Details

6.5.1 Robot Operating System

Overview

ROS is an open-source, meta-operating system for robots. It provides the services one would expect from an operating system, including **hardware abstraction**, **low-level device control**, implementation of commonly-used functionality, **message-passing between processes**, and **package management**. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. ROS is not a real-time framework, though it is possible to integrate ROS with real-time code.

Concepts

ROS has three levels of concepts: the File system level, the Computation Graph level, and the Community level.

ROS File system:

The file system level concepts are ROS resources that you encounter on disk, such as:

- **Packages**: Packages are the main unit for organizing software in ROS. A package may contain ROS runtime processes (*nodes*), a ROS-dependent library, datasets, configuration files, or anything else that is usefully organized together.

- **Manifests**: Manifests (manifest.xml) provide metadata about a package, including its license information and dependencies, as well as language-specific information such as compiler flags.
- **Stacks**: Stacks are collections of packages that provide aggregate functionality, such as a "navigation stack." Stacks are also how ROS software is released and have associated version numbers
- **Stack Manifests**: Stack manifests (stack.xml) provide data about a stack, including its license information and its dependencies on other stacks.
- **Message (msg) types**: Message descriptions, stored in my_package/msg/MyMessageType.msg, define the data structures for messages sent in ROS.
- **Service (srv) types**: Service descriptions, stored in my_package/srv/MyServiceType.srv, define the request and response data structures for services in ROS.

ROS Computation Graph Level

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

- **Nodes**: Nodes are processes that perform computation. ROS is designed to be modular at a fine-grained scale; a robot control system will usually comprise many nodes. For example, one node controls a laser range-finder, one node controls the wheel motors, one node performs localization, one node performs path planning, one Node provides a graphical view of the system, and so on. A ROS node is written with the use of a ROS client library, such as roscpp or rospy.
- **Master**: The ROS Master provides name registration and lookup to the rest of the Computation Graph. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services.
- **Parameter Server**: The Parameter Server allows data to be stored by key in a central location. It is currently part of the Master.
- **Messages**: Nodes communicate with each other by passing messages. A message is simply a data structure, comprising typed fields. Standard primitive types (integer,

floating point, Boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays (much like C structs).

- **Topics**: Messages are routed via a transport system with publish / subscribe semantics. A node sends out a message by *publishing* it to a given topic. The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will *subscribe* to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each others' existence. The idea is to decouple the production of information from its consumption. Logically, one can think of a topic as a strongly typed message bus. Each bus has a name, and anyone can connect to the bus to send or receive messages as long as they are the right type.
- **Services**: The publish/subscribe model is a very flexible communication paradigm, but its many-to-many, one-way transport is not appropriate for request / reply interactions, which are often required in a distributed system. Request / reply is done via services, which are defined by a pair of message structures: one for the request and one for the reply. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply. ROS client libraries generally present this interaction to the programmer as if it were a remote procedure call.
- **Bags**: Bags are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data, such as sensor data, that can be difficult to collect but is necessary for developing and testing algorithms.

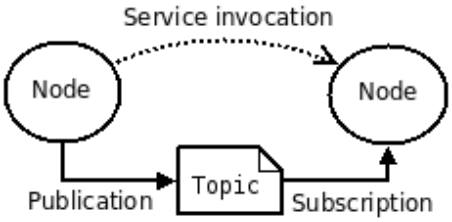


Figure 6.1:Publisher-Subscriber in ROS

Nodes connect to other nodes directly; the Master only provides lookup information, much like a DNS server. Nodes that subscribe to a topic will request connections from nodes that

publish that topic, and will establish that connection over an agreed upon connection protocol.

For example, to control a Hokuyo laser range-finder, we can start the hokuyo_node driver, which talks to the laser and publishes sensor_msgs/LaserScan messages on the scan topic. To process that data, we might write a node using laser_filters that subscribes to messages on the scan topic. After subscription, our filter would automatically start receiving messages from the laser. Note how the two sides are decoupled. All the hokuyo_node node does is publishing scans, without knowledge of whether anyone is subscribed. All the filter does is subscribing to scans, without knowledge of whether anyone is publishing them. The two nodes can be started, killed, and restarted, in any order, without inducing any error conditions [4].

6.5.2 Xbox 360 Kinect sensor

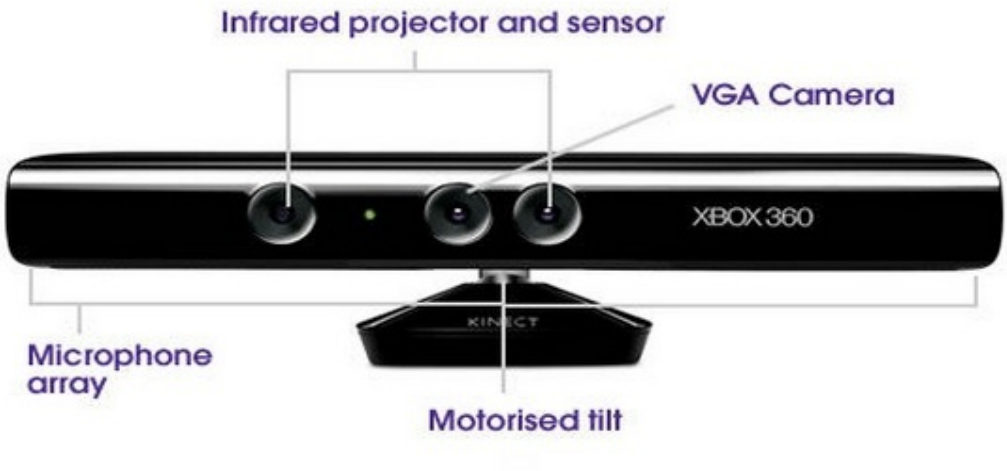


Figure 6.2: The Xbox 360 Kinect Sensor

It is a piece of hardware which incorporates a structured light projector, an infrared (IR) camera, a color camera, and various other peripherals like an array of microphones and an accelerometer meant for adjustment of a built in pivot motor. In this report, the two cameras and the structured light projector are put to use. The infrared camera, when paired with the structured light projector, is able to estimate the depth of a given point in an image relative to

its viewpoint. Given that the x and y position of any point in the image plane is implicitly given, having its depth gives us its three-dimensional projective coordinates. On such coordinates one can perform a reverse perspective projection in order to extract the 3 dimensional Cartesian coordinate of the point relative to the viewpoint of the camera. Performing such a transformation on all points in the image plane will result in a set of points, called a point cloud. A point cloud can be re-projected and explored from all viewpoints independently of how it was generated. Given that the Kinect also has a normal color camera, this can be used to give color to point clouds captured by the structured light depth estimation system[5].

6.5.3 Arduino Uno (Open source Hardware) Microcontroller

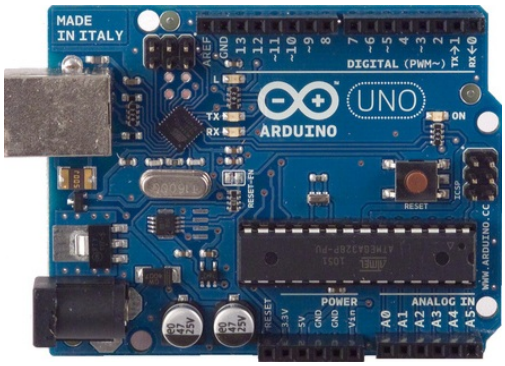


Figure 6.3: Arduino Uno R3 microcontroller board

Arduino is a popular open-source single-board microcontroller, descendant of the open-source Wiring platform, designed to make the process of using electronics in multidisciplinary projects more accessible. The hardware consists of a simple open hardware design for the Arduino board with an Atmel AVR processor and on board input/output support. The software consists of a standard programming language compiler and the boot loader that runs on the board. Arduino hardware is programmed using a Wiring-based language (syntax and libraries), similar to C++ with some slight simplifications and modifications, and a Processing-based integrated development environment[6].

8. Project time-line

Week 1, 2: August 15 - August 24, August 29 - September 4

- Attended Robotics Workshop by Robosoft Systems at IITB on August 20, 21
    - Got a perspective of Robotics as a science and industry in India
    - Worked on the Arduino (Open Source Hardware) Microcontroller
- Introduction to Simultaneous Localization and Mapping
- Referred and studied paper titled “SLAM for Dummies” by Søren Riisgaard and Morten Rufus Blas, MIT

Week 3: September 5 - September 11

- More online research on SLAM algorithms and implementation
- Attended Robotics workshop on FireBird V at ERTS Lab on September 9

Week 4, 5: September 12 - September 18, September 19 - September 25

- Referred to the thesis “FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem With Unknown Data Association” by Michael Montemerlo, CMU

Week 6: October 3 - October 9

- Acquired quotations on list of Hardware specs

Week 7: October 10 - October 16

- Preliminary research on ROS

Week 8: 26 December - 1 January

- created robot model on Google Sketch Up
- made a list of all equipment required
- went to purchase robot equipment

Week 9: 2 January - 8 January

- started robot building process
    - interface of compass
    - fixed the clamp
    - fixed the castor wheel

Week 10: 9 January - 15 January

- robot building continuation
    - fixed motors and interfaced it with motor driver L298N.
    - tested out microcontroller

Week 11: 16 January - 22 January

- robot building continuation
    - interfaced the wheel encoders

    - modded the Kinect

Week 12: 23 January - 29 January

- studied ROS
    - ROS File systems
    - ROS commands

Week 13: 30 January - 5 February

- studied ROS
    - nodes, topics and messages
    - publishers and subscribers

Week 14: 6 February - 12 February

- studied ROS
    - services and parameters
    - used rosserial stack to interface Arduino

Week 15: 13 February – 19 February

- studied ROS
    - visualized point clouds from Kinect using rviz
    - designed ROS architecture for project

Week 16: 20 February – 26 February

- studied and simulated Monte Carlo localization

Week 17: 27 February – 4 March

- studied and simulated Kalman Filter

Week 18: 5 March – 11 March

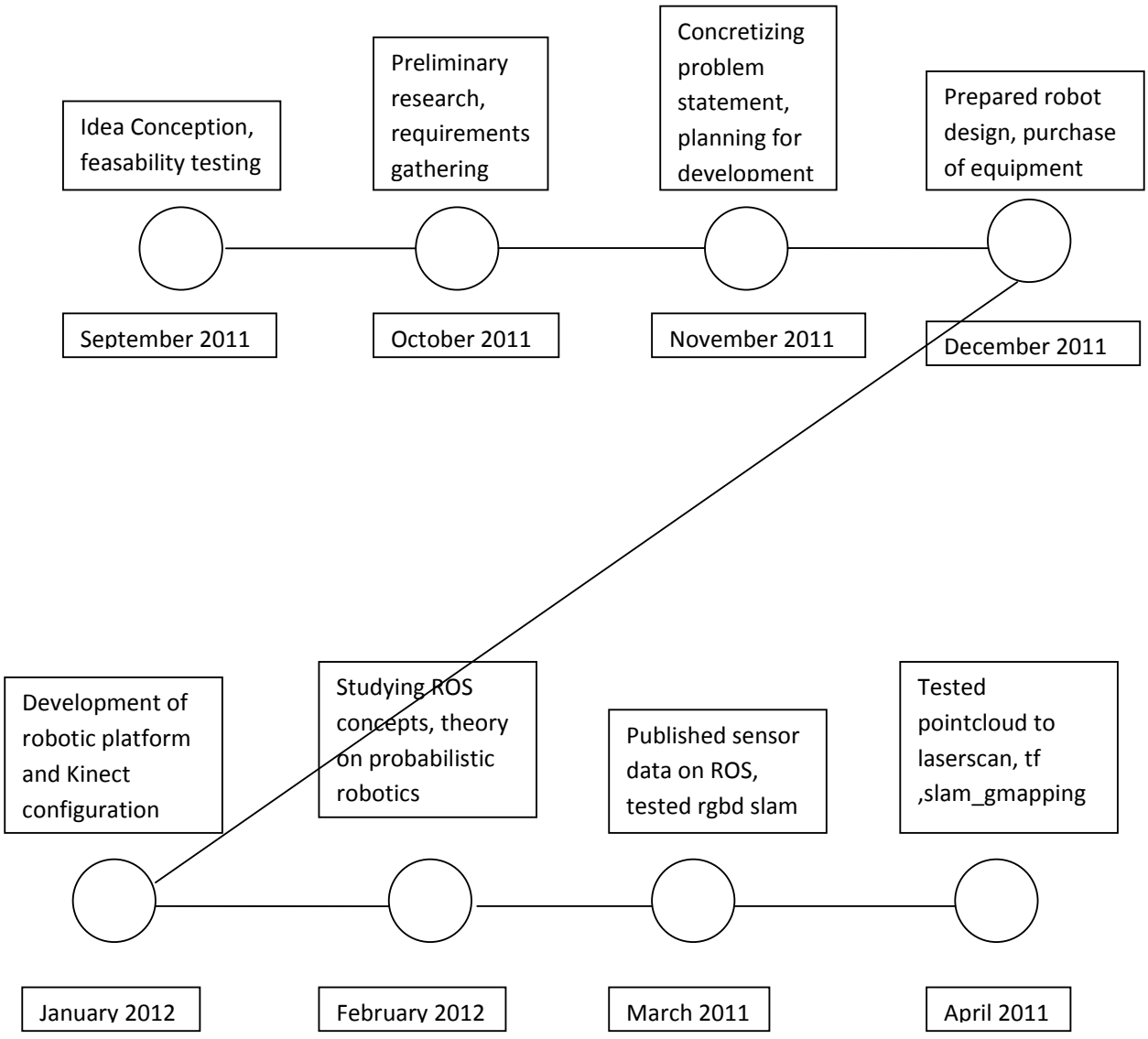
- studied and simulated particle filter

Week 19 – Week 25 12 March – 16 April

- studied graph SLAM
- streamed odometry data on ROS
- published control commands via ROS to robot
- tested RGBD slam
- interfaced second battery to run Kinect
- calibrated odometry data

- initialize geometry_msgs/PoseStamped as message type for odometry data
- implemented tf(transform library) for coordinate frame transformations

8.1 Time-line Summary



9. Conclusion and Future Work

We developed a mobile robot platform, suitable for indoor use. It's equipped with wheel encoders for pose estimates as well as a Kinect sensor, a depth sensing device. This set up is well suited for carrying out mobile robotic experimentation.

Further, we studied the Robot Operating System, an implementation framework for robots such as the one we developed – a tool that provides hardware abstraction and message passing. Using this, we could easily publish the robot pose estimates, a prerequisite for robot localization.

We could interface the Kinect, a device capable of producing point clouds, with which we experimented with rgb-d slam, a package that uses visual odometry to create 3D maps of environments.

Next, we went on to study and implement coordinate transforms for our robot. We could successfully use the odometry data, and similarly the Kinect depth/scan data and transform it to the base link frame of reference, a representational point on the centre of the robot.

Due to the robust nature of our platform, a lot of potential for research still exists.

We could use the 3D maps generated by rgb-d slam, and easily make 3D occupancy grids. These octomaps, as they are known, can be used for path planning activities.

Yet another scope for refinement would be to extract landmarks from the environment, and make the robot capable of identifying them. This paves the way to activities such as autonomous pick and place and learning such as classification and clustering.

People and Object tracking is another area that could well be explored using the platform we developed.

10. References:

[1]. *SLAM for dummies - A Tutorial Approach to Simultaneous Localization and Mapping* by Søren Riisgaard and Morten Rufus Blas, MIT

[2]. *FastSLAM – A factored approach to SLAM* by Montemerlo, CMU

[3]. *Autonomous Robots: Modelling, Planning and Control by* Farbod Fahimi

[4]. *Robot Operating System – http:// www.ros.org, Willow Garage*

[5]. *A Quantitative analysis of Two Automated Registration Algorithms In a Real World Scenario using Point Clouds from the Kinect by* Jacob Kjaer

[6]. *Arduino Open Source(*http://arduino.cc*)*

[7]. *RGBD Slam*(http://www.ros.org/wiki/rgbdslam*)*