
REAL-TIME ROSBERRYPI SLAM ROBOT

A Design Project Report

Presented to the School of Electrical & Computer Engineering of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering, Electrical and Computer Engineering

Submitted by

Gautham Ponnu, Jacob George

MEng Field Advisor : Dr. Joseph Skovira

Degree Date : May 2016

ABSTRACT

MASTER OF ENGINEERING PROGRAM

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

CORNELL UNIVERSITY

DESIGN PROJECT REPORT

Project Title : Real-time ROSberryPi SLAM Robot

Authors : Gautham Ponnu (gp348) , Jacob George (jg965)

Abstract : A realtime monocular (single camera) visual Simultaneous Localization And Mapping (SLAM) robot is built utilizing a server-node based computational approach. The core of the robot is a Raspberry Pi 2 with a Robot Operating System (ROS) wrapper over the Raspbian Wheezy Linux kernel. Different nodes from the robot communicate with the server to map its location based on its surroundings using ORB-SLAM, with loop detection and re-localization capabilities.

EXECUTIVE SUMMARY

The Simultaneous Localization and Mapping (SLAM) problem is concerned with building an intelligent robot that can identify its position when kept at an unknown location and in an unknown environment, while incrementally building a map of the environment it is placed in. SLAM has been formulated and solved by the Robotics community and several algorithms (for example, self-driving cars) already exist. However, the cost of sensors involved and computational intensity of the algorithms are pretty high.

The aim of this project is to build a self-aware, mobile 3D-motion and depth sensing robot using the Raspberry Pi Hardware platform. With the advent of cloud computing, the barriers of high-performance computing have been greatly reduced. Utilizing this ability to offload signal processing, the project intends to produce a single robot implementation with minimal cost on the bill of materials and computational cost. This requires using an architecture which can effectively decouple the robot from the server with minimal effect on the robot's processing capability.

We have demonstrated an affordable implementation platform using low-cost computational hardware and open-source algorithms. Utilizing the wrappers offered by Robot Operating System [ROS] a server-node based model, utilizing a linux-based laptop as a server and a Raspberry-Pi based robot as the client, has been successfully built. A real-time monocular Visual SLAM system on the above described architecture has been successfully demonstrated and results compared.

CONTRIBUTIONS

GAUTHAM PONNU

- Algorithm Identification
 - ROS Server Implementation
 - ORB SLAM Implementation
-

JACOB GEORGE

- Hardware Development
- ROS Raspberry-Pi Implementation
- Calibration & Testing

Contents

1	Introduction	8
2	Design Problem	10
2.1	Simultaneous Localization and Mapping Problem	11
2.2	Problem Statement	13
3	Related Work	14
3.1	Feature Descriptors and Detectors	15
3.1.1	Harris Corner Detector	15
3.1.2	SIFT	15
3.1.3	SURF	15
3.1.4	BRIEF	16
3.1.5	FAST	16
3.1.6	ORB	16
3.1.7	Keyframes	17
3.1.8	Bundle Adjustment	17
3.1.9	Bag of Words	17
3.2	ORBSLAM	18
3.2.1	Tracking	18
3.2.2	Mapping	19
3.2.3	Loop Closing	20
4	Implementation	21
4.1	Design Decisions	21
4.1.1	Communication Protocol	21
4.1.2	Components	22
4.2	Architecture	23
4.3	Simulation Results	26
4.4	Hardware Implementation	26

4.5 Software Implementation	28
4.5.1 Node Setup	28
4.5.2 Server Setup	31
5 Design Verification	37
5.1 Calibration	37
5.2 Experimental Trials	42
5.2.1 Trial 1	42
5.2.2 Trial 2	42
5.2.3 Trial 3	45
5.2.4 Trial 4	45
5.2.5 Trial 5	45
5.3 Bill Of Materials (BoM)	47
6 Conclusion	48
A Code Listing	49
B User Manual	50
C Camera Calibration	51
D References	52
D.1 Hardware	52
D.2 Software	53

List of Figures

2.1	A Robot in an unknown environment	12
4.1	System Architecture	24
4.2	IMU initial test	27
4.3	Robot Hardware	27
5.1	Pi Camera Calibration	38
5.2	Trial 1 : Results	43
5.3	Trial 2 : Results	43
5.4	Trial 3 : Results - Part 1	44
5.5	Trial 3 : Results - Part 2	44
5.6	Trial 4 : Results	45
5.7	Trial 5 : Loop Closure	46

Chapter 1

Introduction

Perception is an important attribute for building any sort of intelligent robot. This aspect is required in a range of robots starting from simple automated vacuum-cleaning robots, self-driving cars to rovers that perform deep space exploration. The perception task can be split primarily into two problems: knowing the environment - area mapping and knowing where the robot is present in the environment - robot localization. To solve the localization problem using any form of on-board sensors a map must be available in the robot from which it can locate it's relative position. However, in order to be able to map the surrounding the robot must know its current relative position. This fundamental **Simultaneous Localisation And Mapping (SLAM)** problem has been at the center of decades of robotics research.

Since its introduction in the nineties, Internet has slowly been seeping into everyday life to the point where it is almost taken for granted today. With this increase in connectivity, several core assumptions that formed the pillars of robot-building need to be rethought. When a robot can transmit and receive data at a sufficient rate from a more powerful computing platform, do we really need to have the intelligence to process the data in the robot itself. Several tools have exploited this new paradigm of robot programming. These tools utilize a server and node based programming approach with multiple independent nodes communicating to each other and the server.

A prominent tool among these being **Robot Operating System (ROS)**, maintained by the Open Source Robotics Foundation. ROS (Robot Operating System) provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more. ROS is licensed under an open source, BSD license.

The Raspberry Pi Foundation, established in 2008 as a UK-based charity through their trading subsidiary, Raspberry Pi Trading Limited, invents and sells low-cost, high-performance computers. Since launching their first product in February 2012, they have sold eight million Raspberry Pi computers and have helped to establish a global community of digital makers and educators.

The Raspberry Pi 2 Model B is the second generation Raspberry Pi. It replaced the original Raspberry Pi 1 Model B+ in February 2015. Running a 900MHz quad-core ARM Cortex-A7 CPU with 1GB RAM, it boasts of several features like 40 GPIO pins, Camera interface (CSI) etc. Since, it has an ARMv7 processor, it can run the full range of ARM GNU/Linux distributions. With an open-design, popularity among hobbyists, backed by a education charity it offers an ideal product to base a low-cost hardware platform on.

Through the course of the following chapters, the design decisions and implementations of an attempt to build a low-cost distributed SLAM robotics platform are presented.

Chapter 2

Design Problem

A rigid body in a 3-dimensional space generally has six degrees of freedom - namely three rotational (commonly called pitch, roll & yaw) and three translational (commonly referred by x, y & z axes). This combination of the position and orientation of the body is commonly referred to as the **pose**.

The concept of State Estimation from sensor data forms the cornerstone of modern day probabilistic robotics. At a surface level, the assumption that the differential displacement can be obtained by double integrating the acceleration values (acquired from an accelerometer or gyroscope - called **odometry data**) seems straightforward. However, this leaves the critical problem of estimating the right value of the integration constant. The integration constant c is a random value that is distributed in a *bayesian* normal distribution. The problem of simultaneous Localization and Mapping was first introduced and solved in a seminal paper by Smith, Self and Cheeseman. [1]

Using filters called *Kalman Filters*, these values can be estimated to obtain a more accurate value of the pose. A more computationally efficient version of this was built and called *Extended Kalman Filters*. This was first developed into a system by Moutarlier and Chatila [2]. The Extended Kalman Filter is used to estimate the state (position) of the robot from odometry data and landmark observations.

2.1 Simultaneous Localization and Mapping Problem

SLAM is essentially a process used to describe the surroundings at a given location without any priori knowledge of the location. We typically estimate the trajectory of the bot as well as map the relative location of surrounding objects (landmarks). The robot tends to produce errors on its own location and the location of landmarks. However, the relative distance between the landmarks are recorded accurately and are constantly updated as the bot traverses. Based on the relative locations of the landmarks, the trajectory and location of the bot is also updated over time.

Consider a robot moving in an unknown environment [3] as depicted in Fig. 2.1 with

- x_k : The state vector describing the location and orientation of the vehicle.
- u_k : The control vector, applied at time $k - 1$ to drive the vehicle to a state x_k at time k .
- m_i : The vector describing the location of the i^{th} landmark which is time invariant.
- z_{ik} : An observation taken from the robot of the location of the i^{th} landmark at time k .

The following sets are also defined:

- $X_{0:k} = x_0, x_1, \dots, x_k = X_{0:k-1}, x_k$: History of robot locations.
- $U_{0:k} = u_1, u_2, \dots, u_k = U_{0:k-1}, u_k$: History of control inputs.
- $m = m_1, m_2, \dots, m_n$: Set of all landmarks.
- $Z_{0:k} = z_1, z_2, \dots, z_k = Z_{0:k-1}, z_k$: Set of all landmark observations.

The **observation model** describes the probability of making an observation z_k when the robot location and landmark locations are known, and is generally described in the form

$$P(z_k | x_k, m) \quad (2.1)$$

Once the robot location and map are defined, observations are conditionally independent given the map and the current vehicle state. The motion model for the robot can be described in terms of a probability distribution on state transitions in the form

$$P(x_k | x_{k-1}, u_k) \quad (2.2)$$

2.1. SIMULTANEOUS LOCALIZATION AND MAPPING PROBLEM DESIGN PROBLEM

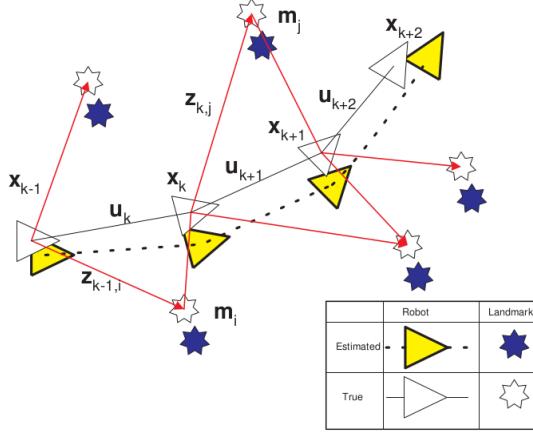


Figure 2.1: A Robot in an unknown environment

The SLAM algorithm is implemented using a two step recursive prediction (time-update) correction (measurement-update) form:

Time-Update

$$P(x_k, m | Z_{0:k-1}, U_{0:k}, x_0) = \int P(x_k | x_{k-1}, u_k) \times P(x_{k-1}, m | Z_{0:k-1}, U_{0:k-1}, x_0) dx_{k-1} \quad (2.3)$$

Measurement-Update

$$P(x_k, m | Z_{0:k}, U_{0:k}, x_0) = \frac{P(z_k | x_k, m) P(x_k, m | Z_{0:k-1}, U_{0:k}, x_0)}{P(z_k | Z_{0:k-1}, U_{0:k})} \quad (2.4)$$

The implementation is based on the assumption that the location of the robot is deterministic at every point. Based on the location, a map is constructed by combining its observations from the surroundings. Conversely, it may be such that the landmark locations are known and the location of the bot has to be computed relative to these locations.

2.2 Problem Statement

The main objective is to build a system to achieve **real-time SLAM** with minimal cost.

It can be divided into the following components :

1. Perform an accurate pose estimation of the robot
2. Offload processing to the server
3. Develop a local map that is capable of reacting to loop closure
4. Keep the overall cost less than \$ 100
5. Minimize the computational load on the robot as much as possible

Chapter 3

Related Work

Today, when one looks at the developments that have occurred in different fields, Computer Vision is one among the disciplines where a lot of development has occurred. Several advanced algorithms have been developed to extract meaningful data from image data. The field of using visual data to perform SLAM is called Visual SLAM.

The objective of Visual SLAM algorithms is to estimate the pose of the camera which is moving and map the geometric structure and appearance of the environment the camera is moving in. The most vital problem in Visual SLAM is obtaining correspondence throughout the image stream. This results in them having a bad response when there are abrupt changes in the motion of the camera. Modern Visual SLAM algorithms obtain this correspondence throughout the images captured using Sparse feature matching techniques and construct maps using a composition of simple geometric constructs like points. The resulting map provides a sparsely furnished, incomplete representation of the environs.

The computer vision research tries to understand a very challenging problem of trying to invert the image formation process that occurs when a real 3D-scene is projected into the lens of camera to create its projected 2D image. Several decades of computer vision research has gone in to try and understand how to recover the structure of the original 3D-scene given only the 2D-passive images.

Going back to the problem of reconstructing the 3D-structure from 2D-images, the first step would be to obtain a reliable method of detecting features. Such methods to extract distinctive invariant features from images, are loosely described together as *Feature descriptors*.

3.1 Feature Descriptors and Detectors

Any Visual SLAM algorithm requires a feature descriptor to describe the feature and a feature detector to detect the features. In almost all Visual SLAM algorithms, *corners* are typically detected and matched into a database. Corners are points that have a different dimension that may arise as the result of geometric discontinuities, such as the corners of real world objects, or from small patches of texture changes.

3.1.1 Harris Corner Detector

The most widely used detector is the Harris corner detector , proposed in 1988 [4]. This was a combined corner and edge detector based on local auto-correlation function. However, Harris corners are not scale-invariant.

3.1.2 SIFT

In 2004, a now popular descriptor called SIFT (Scale Invariant Feature Transform) [5] was introduced. This computes a histogram of local oriented gradients around the interest point and stores the bins in a 28-dimensional vector (8 orientation bins for each of the 4×4 location bins). Relatively fast and reliable, the SIFT descriptor is widely used. However, it imposes a large computational burden, especially for real-time systems such as visual odometry.

3.1.3 SURF

Less computationally intensive algorithms were being researched as a replacement for SIFT. In 2006, a new descriptor called SURF (Speeded-Up Robust Features) [6], based on the Hessian matrix was introduced.

Relying on integral images for image convolution, the SURF operator outperformed SIFT in most tests. The SIFT descriptor was a 128 - vector and therefore was relatively slow to compute and match. While SURF addressed the issue of speed, its descriptor was a 64-vector of floating points values requiring 256 bytes. This caused memory issues when million of descriptors had to be stored.

3.1.4 BRIEF

In 2010, a new feature descriptor called BRIEF (Binary Robust Independent Elementary Features) [7] was developed. This used binary strings as a feature point descriptor. It offered the advantage of being small in size and being highly discriminative. Moreover, the similarity tests could be done by simple evaluation of Hamming distance, instead of the usual l_2 norm. BRIEF performed a relatively small number of intensity difference tests to represent an image patch as a binary string.

However, BRIEF was not designed to be rotationally invariant.

3.1.5 FAST

FAST is a keypoint detector. The intensity threshold between the center pixel and those in a circular ring of distance (eg. in FAST9 - the value of pixels at a distance of 9 pixels about the center) is calculated.

3.1.6 ORB

In 2011, another descriptor based on the FAST keypoint detector and the BRIEF descriptor called ORB (Oriented FAST and Rotated BRIEF) was developed [8]. This used a modified BRIEF descriptor called rBRIEF and a oriented FAST detector.

The authors

1. Added a orientation component to FAST.
2. Computed oriented BRIEF features.
3. Analysed the variance and correlation of oriented BRIEF features.

4. Implemented a learning method for de-correlating BRIEF features under rotational invariance, leading to better performance in nearest-neighbor applications.

The authors also contributed a BSD licensed implementation of ORB to the community, via OpenCV 2.3.

3.1.7 Keyframes

Monocular SLAM was initially performed by filtering all the frames to jointly estimate the map and camera position. However, this wastes a lot of computational effort on processing consecutive frames with no new information. Selecting certain frames (keyframes) and performing costly but more accurate bundle adjustment operations produces better results. This way the mapping is also not related to the frame rate. [9]

3.1.8 Bundle Adjustment

Bundle adjustment refers to methods that are used for refining a visual reconstruction to produce jointly optimal 3D structure and viewing parameter (camera pose and/or calibration) estimates. [10]

3.1.9 Bag of Words

Loop closure refers to the detecting that the current path has already been visited and cognizantly updating the map with the correctly associated data information to build consistent maps. While for small environments, map-to-image methods achieve nice performance, for large environments, image-to-image (or appearance-based) methods scale better. Discrete bag of words technique builds a database from the frames collected by the robot so that the most similar one can be retrieved when a new frame is captured. If they are similar enough, a loop closure is detected. In this work [11], a bag of words was shown to be used to discretize a binary space, and augment it with a direct index, in addition to the usual inverse index. This allowed for faster retrieval of images and efficiently obtain point correspondences between images.

3.2 ORBSLAM

ORBSLAM is a SLAM algorithm implementation developed by Raul Mur et. al [12] at the University of Zaragoza. The system utilizes ORB descriptors and has several novel features, including

- Uses the same ORB features for all tasks
- real-time loop closing based on optimization of the pose graph
- recovery from tracking failure
- automatic map initialization procedure
- survival of fittest approach to map point and key frame selection

The system works in three parallel threads which are described below:

3.2.1 Tracking

The tracking thread localizes the camera pose with every frame and selects when to insert a new keyframe. First, an initial feature matching with the previous frame is done. Then, the pose is optimized using motion-only bundle adjustment techniques.

This thread also handles tracking reinitialization, in case tracking is lost (eg. due to abrupt movement). This is done using the place recognition module which performs a global relocalization.

At a high level, the following are the operations that occur to perform pose estimation

- Extract FAST corners at eight-scale levels
- Divide each scale level in a grid to obtain at least 5 corners per cell
- Detect corners in each cell, adjusting threshold to find at least 5 corners
- Compute orientation and ORB descriptor
- Assume a constant velocity model and check whether guide points can be tracked

- If not, perform wider search and optimize with the found global correspondence

In case tracking is lost, the frame is converted into bag of words and the recognition database is queried for keyframe candidates for global relocalization using a PnP algorithm [13].

To insert a new keyframe, all of the following conditions must be met.

- More than 20 frames must have passed from the last global relocalization.
- Local mapping is idle, or more than 20 frames have passed from last keyframe insertion.
- Current frame tracks at least 50 points.
- Current frame tracks less than 90% points than reference keyframe

3.2.2 Mapping

This thread runs on every new keyframe.

Upon every new keyframe insertion, this thread updates the *covisibility graph*, adds a new node for that keyframe and updates all shared points. It then updates the spanning tree, linking this frame with the keyframe that has the most points in common. Finally, it computes bag of words representation for each keyframe.

In order for a point to be retained in the map it must fulfill two conditions within the next three keyframes. This is so as to avoid spurious noise signals from becoming part of the map and ensure that they are trackable and not wrongly triangulated.

1. The tracking must find the point in more than the 25% of the frames in which it is predicted to be visible.
2. If more than one keyframe have passed from map point creation, it must be observed from at least three keyframes.

By triangulating ORBs from connected keyframes in the covisibility graph new map points are created.

This thread also tries to detect redundant keyframes and delete them . All keyframes whose 90% of the map points have been seen in at least other three keyframes in the same or finer scale are deleted.

3.2.3 Loop Closing

The loop closing thread takes the last processed keyframe and tries to determine whether loop closure has occurred. This is done by calculating a similarity transformation between ORB in the map points of the current keyframe and the loop candidate keyframes. If loop closure has been discovered, then duplicated map points are fused and new edges are inserted in the covisibility graph that will attach the loop closure.

Chapter 4

Implementation

This section pertains to the of the overall architecture, design and implementation of SLAM on a low-cost computing platform. Each portion was carefully chosen to serve specific needs of the system, attempting to maximize performance and efficiency.

4.1 Design Decisions

4.1.1 Communication Protocol

The overall design is based on a distributed system, in which the robot (slave) is proposed to acquire data while the computation is performed on the server (master). Possible and appropriate communication between the robot and server would be either via Bluetooth or Wi-Fi. Weighing the cons and pros of each, we concluded using Wi-Fi was best suited for the project. Wi-Fi provided access to a larger distance, allowing the user to remotely communicate with robot and server. The tradeoff here was the power utilization which was outweighed by the range achieved on using Wi-Fi. Wi-Fi also provided a greater bandwidth for communication which was key in sending images across the robot and server. Adhering to a low cost platform, the price of Wi-Fi and Bluetooth modules for the Raspberry Pi 2 were almost the same. Hence, in almost every aspect Wi-Fi better suited the project.

4.1.2 Components

Inertial Measurement Unit (IMU)

The IMU served to debug and test the initial architecture of the robot. Based off various reviews and surveys on the possible accuracy achievable, the project uses Invensense MPU 9250. This IMU served as a 9 axis accelerometer, gyroscope and magnetometer.

Range Finder

Popular SLAM techniques generally involve using a laser range finder for computation and generating a map. A basic laser range finder would cost approximately \$200. However, the cost of a laser range finder of moderately decent accuracy (required for the project) cost over \$1000 and this deviated from our cost constraints. ORB Slam, chosen for our implementation, is able to map the surroundings without the need for a range finder (rather it works based off camera image features) and hence the range finder was avoided.

Battery

The batteries were meant to run the servos on the Robot as well as power the Raspberry Pi 2. Since we needed to reuse the system and constantly replacing the battery seemed expensive and inefficient, the batteries chosen for the robot were rechargeable. Based off the voltage and current requirements as well as maintaining a relatively low payload on the robot, we chose five Ni Metal Hydride batteries. Each provided 1.2V and 4200mAH and each weighed 60.33g. Total payload of batteries accumulated to 301.65 grams.

Camera

Cameras for SLAM implementation could be single (monocular) or dual (stereo). Our implementation uses a monocular camera which is the Raspberry Pi camera module of 5 megapixel resolution. It was operated at a frame rate of 25 frames/second and provided a resolution of 320x200.

Motor

Motors used were parallax continuous rotation motors. They served the necessary requirements to move the payload of the robot after careful calibration and testing.

4.2 Architecture

Broadly, the architecture can be divided into the robot (Raspberry Pi 2 based) and server (Laptop) communicating via the internet. The robot, also known as the slave in the system, is built using Raspberry Pi 2 along with key components (nodes) such as parallax continuous rotation servos, the Inertial Measurement Unit (IMU) MPU 9250 and the Raspberry Pi camera module.

The Inertial Measurement Unit (IMU) was initially part of the architecture for the implementation of SLAM. On further research, along with the IMU, a range finder of certain accuracy was required for its implementation. As described earlier, the range finder is quite expensive and it defeats the purpose of our project which was developing a low cost platform.

This inspired further research on SLAM and Visual SLAM was selected as best suited our need. Visual SLAM used visual images acquired to compute the map based off the difference of features in frames. Specifically, a visual SLAM algorithm known as ORB-SLAM 2,[12] developed by Raul Mur-Artal, Juan D. Tardos, J. M. M. Montiel and Dorian Galvez-Lopez was used.

The Raspberry Pi 2 is running Linux Kernel 4.1.6 of the Raspbian Wheezy distribution. Although a newer release Jessie was available, the stability of the Wheezy release was deemed vital. Hence, Wheezy was chosen to be the kernel version. On top of this kernel, the Robot Operating System (ROS) wrapper is placed for communication. The ROS compiled over the Raspbian Kernel is ROS Indigo Igloo - Common. This compilation was intensive and over 120 packages were installed. Several challenges of missing files, dependencies and deprecations had to be resolved.

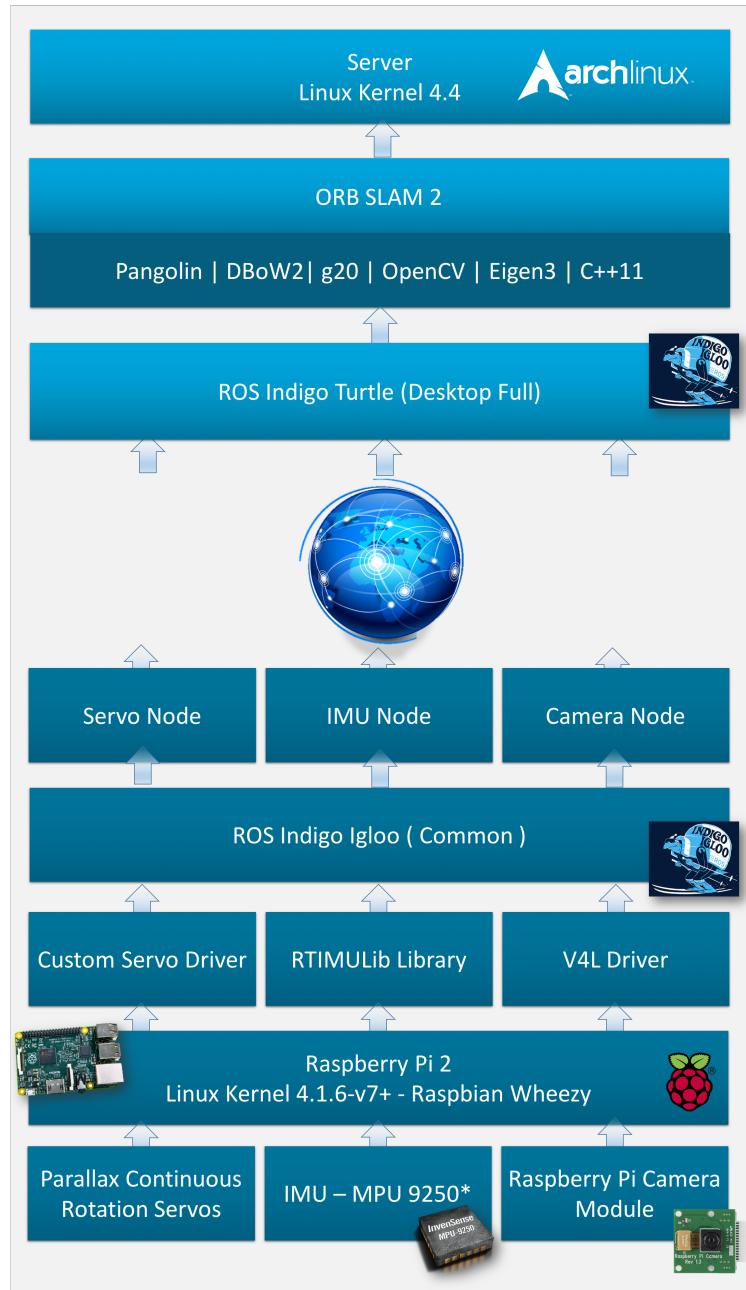


Figure 4.1: System Architecture

Various components are interfaced with the Raspberry Pi 2 using drivers and libraries. The Servo motors are interfaced using a custom built driver, the IMU library is courtesy RTIMULib using i2c for communication and finally the camera module is interfaced using V4L driver. The custom built driver for the servos operated such that, the user could remotely specify the direction (forward, reverse, left or right) for a time step of 250 ms. The motors were run on half the maximum speed to accurately map the environment.

Over the Linux Kernel, the Robot Operating System is used to establish a server-node system. The mobile robot being the node is used for acquiring the images necessary for computation on the server end.

The Robot Operating System basically involves utilization of *nodes*. These ROS-nodes are established for communication across different Robot Operating Systems. This architecture involves three nodes on the slave end, each being for the servo motors, the Inertial Measurement Unit and the camera. Each ROS-node publishes *messages* to a particular *topic*. For instance, the Raspberry Pi camera node publishes raw, uncompressed image data (messages) using a Raspberry Pi camera topic.

These nodes are tuned to communicate with the Robot Operating System on the Master end. The Master here is the server / laptop where the Robot Operating System collects the messages published by the nodes.

Communication between the Camera node and Master node is over the internet using Wi-Fi. Finally on the server / master end, we have an Arch-Linux Kernel 4.4 over which the Robot Operating System (ROS) wrapper has been compiled. Here, the ROS is Full Desktop version of ROS Indigo Turtle, which involved intensive compilation. The ROS here is used for communication and collection of the messages published by the nodes on the slave / robot end.

The key data received is the raw (uncompressed) image data acquired from the Raspberry Pi Camera module. This in turn is used for computation in mapping the environment, that is, implementation of SLAM.

As mentioned previously, we have implemented ORB-SLAM 2 (a version of Visual SLAM) on the server / master end. ORB SLAM include several details such as Pangolin, Discrete Bag of Words (DBoW2), g2o, OpenCV, Eigen 3, C++11. In brief, Pangolin is used for OpenGL to visualize and generate a user interface. DBoW2 is used for image classification, it converts images in terms of bag of words representation to provide a visual vocabulary. This vocabulary file had over a million lines of data. DBoW2 is used for in-place recognition while g2o is used for optimization of non-linear functions. OpenCL is used for acceleration and OpenCV is used for feature and image manipulations. Eigen 3 is used for matrix computations that are involved in g2o.

ROS and ORB-SLAM 2 provided us the desired distributed system at a low cost with a server-node approach. This enables room for expansion by adding new nodes and pchapter more slaves to map a region quicker.

4.3 Simulation Results

Initial debugging and testing was performed using the Inertial Measurement Unit (IMU) MPU 9250 on the Raspberry Pi 2. This was interfaced using the driver provided by RTIMULib. The GUI for this is shown in Fig.4.2, this is after the calibration of the accelerometer.

4.4 Hardware Implementation

A simple and rugged structure for the robot was required. The complete robot is shown in Fig. 4.3 and the various components are listed below.

Chassis

The chassis of the robot is courtesy of Prof. Carl Poitras's (Cornell ECE) course which is 3D printed model. This served as the base for mounting the Raspberry Pi 2, the camera module and the Inertial Measurement Unit (IMU).

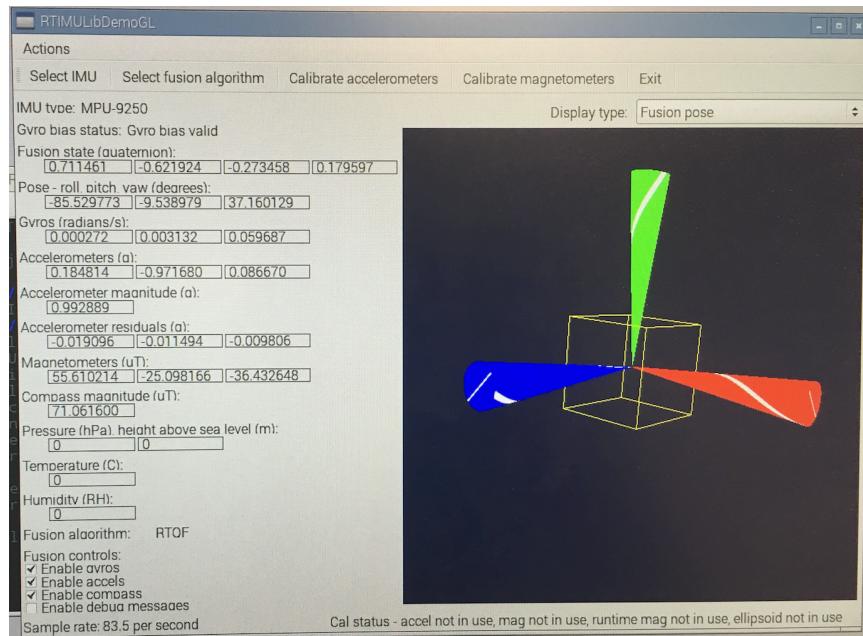


Figure 4.2: IMU initial test

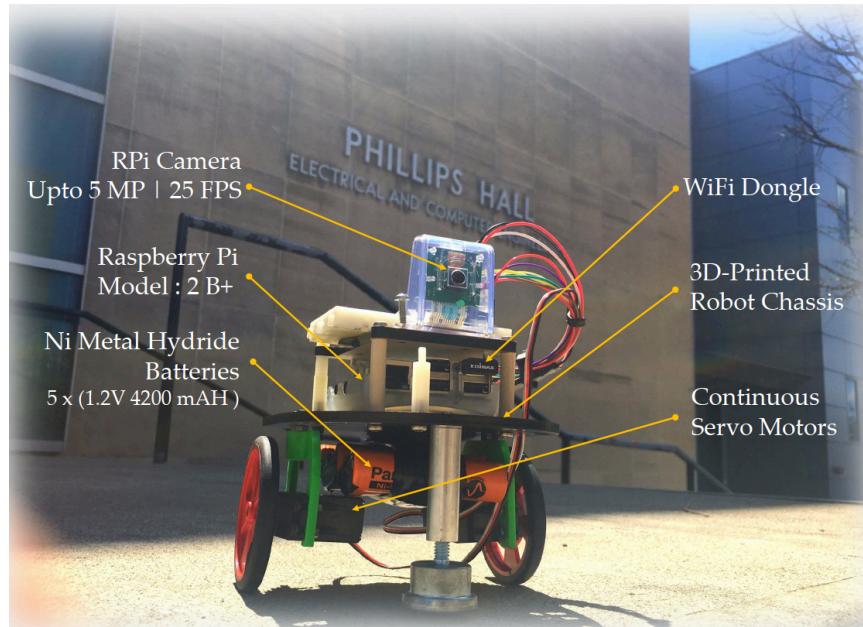


Figure 4.3: Robot Hardware

Battery

The four Nickel Metal Hydride batteries are placed below the chassis using velcro and electrical tape to pad them all together.

Motor

The two parallax continuous rotation motors are attached to the wheels at the bottom below the chassis.

Sensor

Camera Module was module on top and the IMU was placed at the center tip of the robot. Both are placed as accurately parallel to the ground.

4.5 Software Implementation

The system has a complex, multi-levelled software implementation. The following sub-sections will list different components of the same.

4.5.1 Node Setup

The Node setup comprises of the software on the Raspberry Pi.

Core System

The core linux system of the Raspberry Pi is based on the Raspbian Wheezy Kernel. Detailed instructions to build this can be found at <https://www.raspberrypi.org/downloads/raspbian/>

ROS Installation on Raspbian

The basic bootstrap dependencies for installing

```

1 $ sudo apt-get install python-pip python-setuptools
  ↳ python-yaml python-argparse python-distribute
  ↳ python-docutils python-dateutil python-six
2 $ sudo pip install rosdep rosinstall_generator wstool
  ↳ rosinstall

```

The `rosdep` tool is a ROS program that allows the user to install all the dependencies for different packages.

```

1 $ sudo rosdep init
2 $ rosdep update

```

ROS Dependencies

In order to build the catkin packages for ROS, a Catkin workspace needs to be initialized and built. Then using `wstool` the particular variant of ROS packages can be downloaded to the `src` directory. The list of files is stored in the `indigo-ros_comm-wet.rosinstall` file. The tool can then be programmed to download the required packages. [14]

```

1 $ mkdir ~/ros_catkin_ws
2 $ cd ~/ros_catkin_ws
3 $ rosinstall_generator ros_comm --rosdistro indigo --deps
  ↳ --wet-only --exclude roslib --tar >
  ↳ indigo-ros_comm-wet.rosinstall
4 $ wstool init src indigo-ros_comm-wet.rosinstall

```

Certain dependencies are not available in the Raspbian platform. These packages need to be compiled manually.

The system needs to have gcc version 4.7+. `libconsole-bridge-dev` can be installed with the following commands.

```

1 $ cd ~/ros_catkin_ws/external_src
2 $ sudo apt-get build-dep console-bridge
3 $ apt-get source -b console-bridge
4 $ sudo dpkg -i libconsole-bridge0.2*.deb
   ↳ libconsole-bridge-dev_*.deb

```

`liblz4-dev` can be installed with the following commands.

```

1 $ cd ~/ros_catkin_ws/external_src
2 $ apt-get source -b lz4
3 $ sudo dpkg -i liblz4-*.deb

```

With the dependencies installed the actual ROS packages can now be installed. Begin by getting the various dependencies.

```

1 $ cd ~/ros_catkin_ws
2 $ rosdep install --from-paths src --ignore-src --rosdistro
   ↳ indigo -y -r --os=debian:wheezy

```

A 8 GB SD card was used as the main non-volatile memory in the Raspberry Pi. This by default does not have a swap space enabled to conserve disk usage. The ROS package compilation will fail in such a scenario with an "internal compiler error" because the Pi does not have enough memory. A fix for this is to add swap space on a USB pendrive that is connected to the Pi.

To manage swap space, Raspian uses a script `dphys-swapfile`. The standard location for the configuration files is at `/etc/dphys-swapfile`. The file at `/etc/dphys-swapfile` is modified to the below.

```

1 CONF_SWAPSIZE=512 # Size of the Swap Space
2 CONF_SWAPFILE=/mnt/sda1/swap.file # Location, with /mnt/sda1
   ↳ being a USB pendrive that we connected to the Pi

```

Care should be taken to always deactivate the swap space before unmounting the pendrive.

```
1 $ sudo swapoff /mnt/sda1/swap.file
```

Now, the ROS package compilation can be run.

```
1 $ sudo ./src/catkin/bin/catkin_make_isolated --install
  ↳ -DCMAKE_BUILD_TYPE=Release --install-space /opt/ros/indigo
```

ROS will be installed in the system. In order to use ROS, the `setup.bash` file needs to be sourced. This can be added to the `~/.bashrc` file so that it automatically gets sourced every time the bash shell is started.

```
1 $ echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
2 $ source ~/.bashrc
```

4.5.2 Server Setup

The Server setup comprises of the software on the Linux Server.

Install arch linux base package

Begin by installing the base system of Arch Linux. The following link offers a good set of instructions https://wiki.archlinux.org/index.php/Installation_guide. The project was tested on a kernel of 4.5.4-1-ARCH #1 SMP PREEMPT. A good desktop environment, like GNOME is recommended.

A tool to utilize the Arch Build System like `pacaur` <https://github.com/rmarquis/pacaur> is required. The Arch Build System (ABS) is a ports-like

system for building and packaging software from source code. While pacman is the specialized Arch tool for binary package management (including packages built with the ABS), ABS is a collection of tools for compiling source into installable .pkg.tar.xz packages.

Install Pangolin Dependencies

The following are vital pangolin dependencies that need to be installed.

FFMPEG

FFMPEG is required for video decoding and image rescaling. Install by
(arch/ABS) pacman -S ffmpeg

libuvc

LibUVC is required for cross-platform webcam video input
(arch/ABS) pacman -S libuvc

Image Libraries

The following image libraries, viz. libjpeg, libpng, libtiff, libopenexr are required for reading still-image sequences.
(arch/ABS) pacman -S libjpeg libpng12 libtiff5 libopenexr

Install Pangolin

Now that all the dependencies have been installed proceed to install Pangolin.

```

1 $ git clone https://github.com/stevenlovegrove/Pangolin.git
2 $ cd Pangolin
3 $ mkdir build
4 $ cd build
5 $ cmake -DCPP11_NO_BOOST=1 ..
6 $ make -j

```

As a norm, going ahead, a detailed explanation for each of the fixes implemented is provided. These will be explained in the comments.

```

1 $ vim Pangolin/include/pangolin/video/drivers/ffmpeg.h
2 // add "#define PixelFormat AVPixelFormat" after AVCodec
   ↳ includes
3 $ vim Pangolin/src/pangolin/video/drivers/ffmpeg.cpp
4 // replace all "PIX_FMT_*" strings to "AV_PIX_FMT_*"

```

Install OpenCV2

OpenCV needs to be installed in a particular order. The authors strongly feel that this probably due to the hard-coded links towards OpenCV 2.4 in ROS *Indigo Turtle*. So, an ideal base install would be OpenCV2 but the arch system due to the fact it uses a rolling update mechanism has already migrated towards OpenCV 3. However, we can utilize the Arch Build System [ABS] to install the required packages in the right order

Begin by installing OpenCL. OpenCL is an implementation to install parallel programming using the GPU. Based on the GPU installed select among the right headers for OpenCL. For an integrated Intel HD Graphics card, use

```
(arch/ABS) $ pacman -S intel-opencl-runtime intel-opencl-sdks
```

Then install OpenCV

```
(arch/ABS) $ pacman -S opencv2
```

Once, the main OpenCV is installed, you will now have to compile the ROS OpenCV modules manually to ensure OpenCL headers are rightly linked in the OpenCV libraries. This will be done during the install of ROS.

Install Eigen, BLAS and LAPACK

Also required are libraries for Eigen (Matrix Manipulation) and BLAS LAPACK algorithms.

```
(arch/ABS) $ pacaur -S eigen blas lapack
```

Install ROS Desktop

Robot Operating System is a set of open source software libraries and tools that help you build robot applications provided by Willow Garage. In our particular implementation, we used *ROS Indigo Turtle*.

```
(arch/ABS) pacaur -S python2-rosdep python2-rosdistro  
(arch/ABS) pacaur -S python2-rosinstall-generator  
(arch/ABS) pacaur -S python2-rosPKG ros-build-tools ros-desktop-full
```

Once, that is done edit .bashrc to add the following lines.

```
1 # ROS
2 indigo() {
3     source /opt/ros/indigo/setup.bash
4     export
5         PYTHONPATH=/opt/ros/indigo/lib/python2.7/site-packages:$PYTHONPATH
6     export
7         PKG_CONFIG_PATH="/opt/ros/indigo/lib/pkgconfig:$PKG_CONFIG_PATH"
8 }
```

Install ROS Vision OpenCV

Now, compile and install the ROS Vision OpenCV module.

```
1 # Call the function we wrote in .bashrc to source the
   ↵ environment files and commands.
2 $ indigo
3 $ mkdir -p ~/ros_catkin_ws/src
4 $ cd ~/ros_catkin_ws/src
5 $ catkin_init_workspace
6 $ git clone
   ↵ https://github.com/ros-perception/vision_opencv.git
7 $ cd ~/catkin_ws/
8 # Arch has Python3 as default, so pass the right flags to
   ↵ ensure python2.7 is used
9 $ catkin_make -DPYTHON_EXECUTABLE=/usr/bin/python2
   ↵ -DPYTHON_INCLUDE_DIR=/usr/include/python2.7
   ↵ -DPYTHON_LIBRARY=/usr/lib/libpython2.7.so
10 $ source devel/setup.bash
```

Install ORBSLAM2

Now, proceed ahead to the install of ORBSLAM2 algorithm.

```
1 $ git clone https://github.com/raulmur/ORB_SLAM2.git ORB_SLAM2
2 $ cd ORB_SLAM2
3 $ chmod +x build.sh
4 $ ./build.sh
```

ORBSLAM source code has certain issues, fix them by manually editing the files as shown below.

```

1 ORBSLAM2 has errors in usleep.
2 Inserted #include <unistd.h> in all these files.
3
4 Actual error was:
5 /home/gapo/meng/deps/ORB_SLAM2/src/LocalMapping.cc:94:28:
   ↳ error: usleep was not declared in this scope
6           usleep(3000);
7 /home/gapo/meng/deps/ORB_SLAM2/src/LoopClosing.cc:85:20:
   ↳ error: usleep was not declared in this scope
8 /home/gapo/meng/deps/ORB_SLAM2/src/System.cc:133:28: error:
   ↳ usleep was not declared in this scope
9 /home/gapo/meng/deps/ORB_SLAM2/src/Tracking.cc:1523:20: error:
   ↳ usleep was not declared in this scope
10 /home/gapo/meng/deps/ORB_SLAM2/src/Viewer.cc:159:28: error:
    ↳ usleep was not declared in this scope

```

Install ORBSLAM2 ROS NODE

Add the path including Examples/ROS/ORB_SLAM2 to the ROS_PACKAGE_PATH environment variable. Open .bashrc file and add at the end the following line. Replace PATH by the folder where you cloned ORB_SLAM2:

```

export
→ ROS_PACKAGE_PATH=${ROS_PACKAGE_PATH}:PATH/ORB_SLAM2/Examples/ROS}
# Now source .bashrc
$ source ~/.bashrc

```

Now, go to the Examples/ROS/ORB_SLAM2 folder and execute:

```

1 $ mkdir build
2 $ cd build
3 $ cmake .. -DROS_BUILD_TYPE=Release
4 $ make -j

```

Chapter 5

Design Verification

The basic setup involves running the *roscore* on the master / server end. The Raspberry Pi Camera node is initialized and raw or uncompressed data is published from the slave / robot end. Further, ORB-SLAM 2 is run on the master end and a real time map is generated based off the surroundings, post initialization.

5.1 Calibration

The calibration of the Raspberry Pi camera module is done using OpenCV as show in fig. 5.1. This is performed using Monocular calibration by calibrating the camera against a 8x6 checkerboard of 108mm squares. The calibration file generated is then modified to suit needs for ORB-SLAM 2 implementation.

The following file is obtained from OpenCV after the completion of calibration with over 110 readings taken.

```
1 [image]
2
3 width
4 320
5
6 height
```

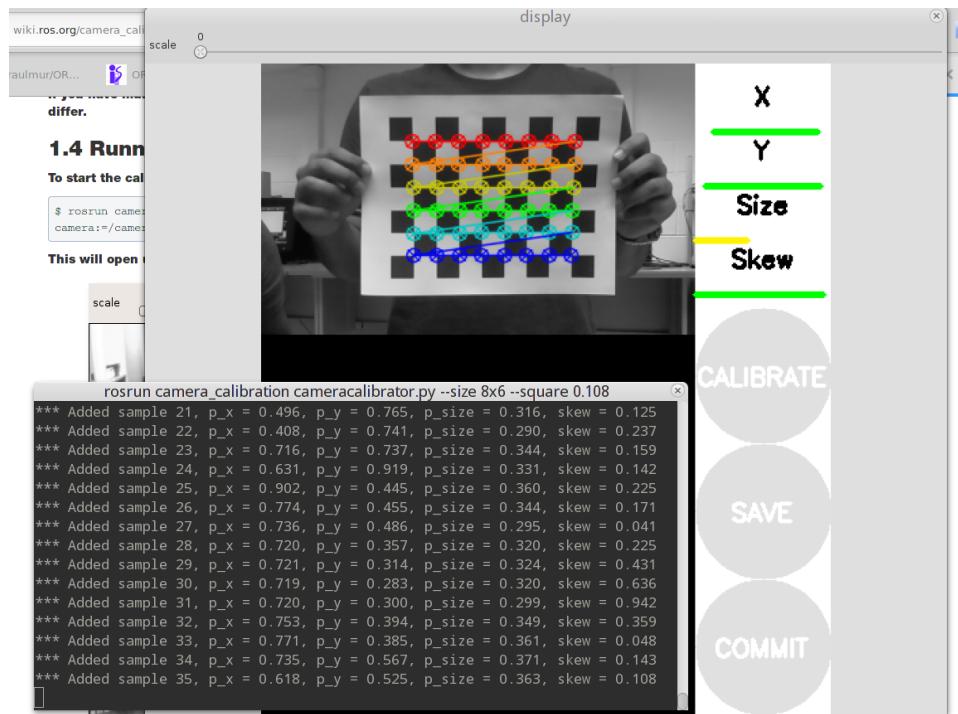


Figure 5.1: Pi Camera Calibration

```

7 200
8
9 [narrow_stereo]
10
11 # Camera Matrix is of the form
12 # | fx 0 cx |
13 # | 0 fy cy |
14 # | 0 0 1 |
15
16 camera matrix
17 324.509473 0.000000 158.846334
18 0.000000 331.040207 60.245944
19 0.000000 0.000000 1.000000
20
21 # Distortion Matrix is of the form = (k1 k2 p1 p2 k3)
22
23 distortion
24 -0.083907 -0.017098 -0.060973 -0.006870 0.000000
25
26 rectification
27 1.000000 0.000000 0.000000
28 0.000000 1.000000 0.000000
29 0.000000 0.000000 1.000000
30
31 projection
32 322.097412 0.000000 156.528006 0.000000
33 0.000000 312.132385 51.607250 0.000000
34 0.000000 0.000000 1.000000 0.000000
35
36
37 ('D = ', [-0.08390708306380001, -0.01709849449502445,
    ↳ -0.06097293595759911, -0.006869946930348199, 0.0])
38 ('K = ', [324.50947291890566, 0.0, 158.84633408648983, 0.0,
    ↳ 331.0402072169732, 60.24594431450795, 0.0, 0.0, 1.0])
39 ('R = ', [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0])
40 ('P = ', [322.097412109375, 0.0, 156.52800603058859, 0.0, 0.0,
    ↳ 312.13238525390625, 51.60724963445773, 0.0, 0.0, 0.0, 1.0,
    ↳ 0.0])
41 # oST version 5.0 parameters
42

```

```

43
44 [image]
45
46 width
47 320
48
49 height
50 200
51
52 [narrow_stereo]
53
54 camera matrix
55 324.509473 0.000000 158.846334
56 0.000000 331.040207 60.245944
57 0.000000 0.000000 1.000000
58
59 distortion
60 -0.083907 -0.017098 -0.060973 -0.006870 0.000000
61
62 rectification
63 1.000000 0.000000 0.000000
64 0.000000 1.000000 0.000000
65 0.000000 0.000000 1.000000
66
67 projection
68 322.097412 0.000000 156.528006 0.000000
69 0.000000 312.132385 51.607250 0.000000
70 0.000000 0.000000 1.000000 0.000000

```

As mentioned in the comments above, the values from the `Camera matrix` and `distortion matrix` are extracted. These calibration parameters are parsed to create the calibration `.yaml` file for ORB SLAM.

```

1 %YAML:1.0
2
3 #-----
4 # Camera Parameters.
5 #-----
6

```

```
7 Camera.fx: 347.055763
8 Camera.fy: 346.578125
9 Camera.cx: 149.511257
10 Camera.cy: 123.117774
11
12 Camera.k1: 0.167090
13 Camera.k2: -0.506412
14 Camera.p1: -0.008420
15 Camera.p2: -0.014372
16 Camera.k3: 0
17
18 # Camera frames per second
19 Camera.fps: 25.0
20
21 # Color order of the images (0: BGR, 1: RGB. It is ignored if
   ↳ images are grayscale)
22 Camera.RGB: 1
23
24 #-----
25 # ORB Parameters
26 #-----
27
28 # ORB Extractor: Number of features per image
29 ORBExtractor.nFeatures: 1000
30
31 # ORB Extractor: Scale factor between levels in the scale
   ↳ pyramid
32 ORBExtractor.scaleFactor: 1.2
33
34 # ORB Extractor: Number of levels in the scale pyramid
35 ORBExtractor.nLevels: 8
36
37 # ORB Extractor: Fast threshold
38 # Image is divided in a grid. At each cell FAST are extracted
   ↳ imposing a minimum response.
39 # Firstly we impose initThFAST. If no corners are detected we
   ↳ impose a lower value minThFAST
40 # You can lower these values if your images have low
   ↳ contrast
41 ORBExtractor.initThFAST: 20
```

```
42 ORBExtractor.minThFAST: 7
43 -----
44 #-----#
45 # Viewer Parameters
46 #-----#
47 Viewer.KeyFrameSize: 0.05 # 0.05
48 Viewer.KeyFrameLineWidth: 1
49 Viewer.GraphLineWidth: 0.9
50 Viewer.PointSize: 2
51 Viewer.CameraSize: 0.08 # 0.08
52 Viewer.CameraLineWidth: 3
53 Viewer.ViewpointX: 0
54 Viewer.ViewpointY: -0.7
55 Viewer.ViewpointZ: -1.8
56 Viewer.ViewpointF: 500
```

5.2 Experimental Trials

5.2.1 Trial 1

Fig. 5.2 depicts the initial run of ORB-SLAM. The system examines for sufficient features for tracking in the environment in order to initialize. When the necessary threshold of features detected is achieved, it completes the initialization.

5.2.2 Trial 2

Post initialization, the keyframes are generated (in blue) and the green line depicts the path traversed. The green-colored rectangle shows the current position of the robot. The red dots denote the points in the surrounding that have not yet been correlated. All mapping takes place in real time across the master and slave. Currently it is operating in SLAM Mode.

5.2. EXPERIMENTAL TRIALS

DESIGN VERIFICATION

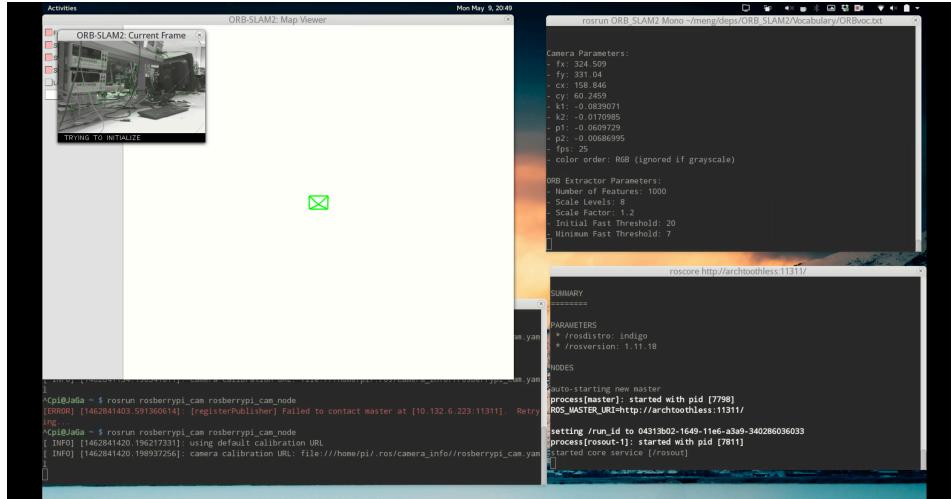


Figure 5.2: Trial 1 : Results

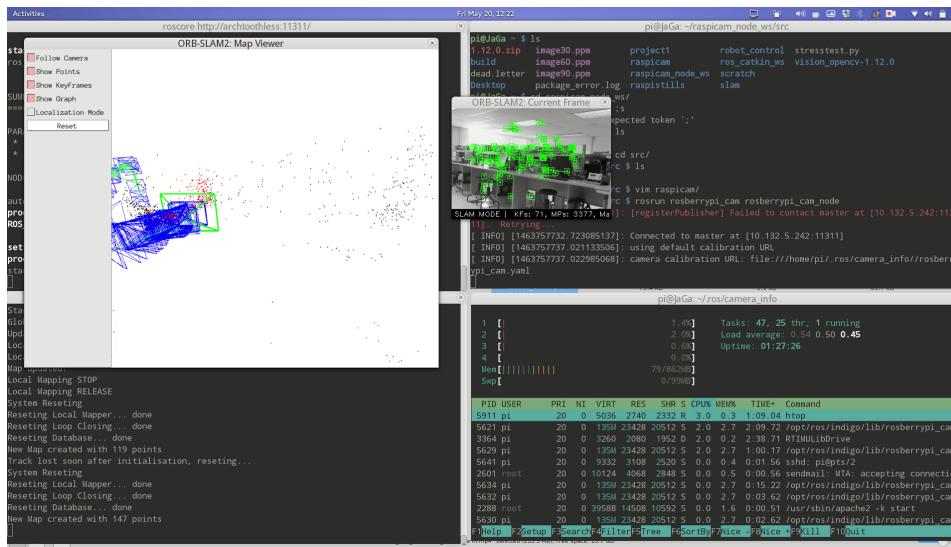


Figure 5.3: Trial 2 : Results

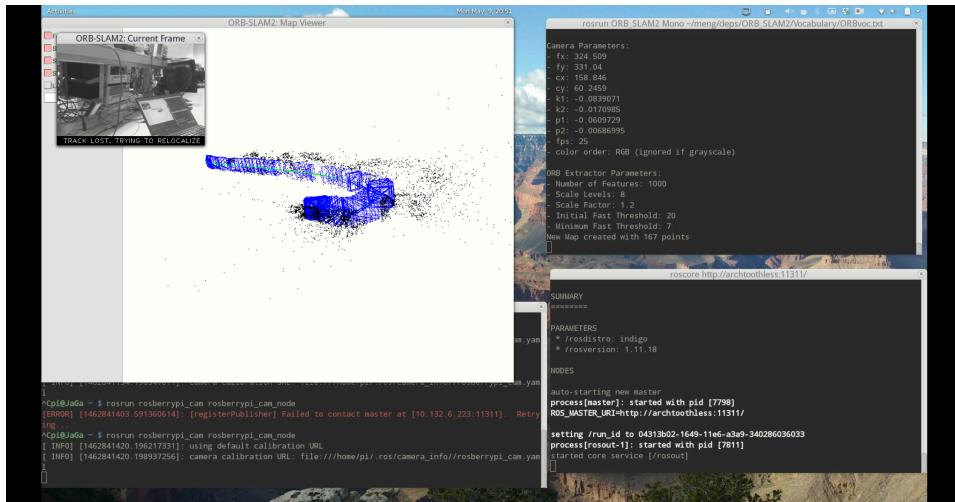


Figure 5.4: Trial 3 : Results - Part 1

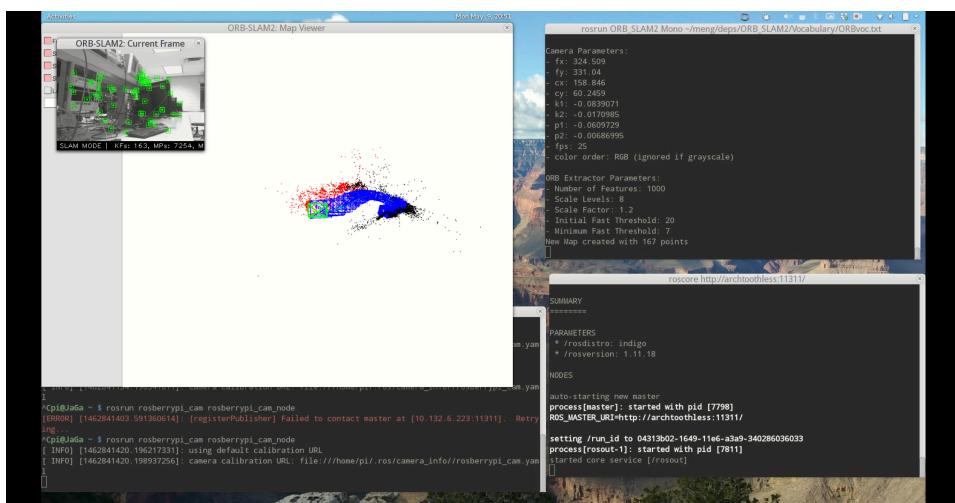


Figure 5.5: Trial 3 : Results - Part 2

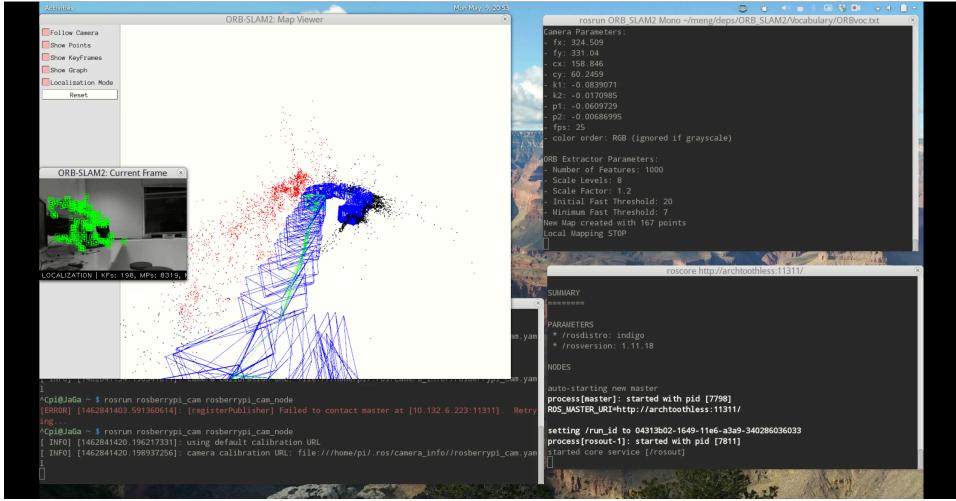


Figure 5.6: Trial 4 : Results

5.2.3 Trial 3

Fig. 5.4 and Fig. 5.5 show the robot losing track of the map while traversing and re-establishing the track. On retracing to a familiar region already successfully mapped, the system is able to reinitialize its position correctly based off the features mapped earlier. This shows application in the real world where there are constant changes and reinitialization is key.

5.2.4 Trial 4

Until Trial 4, the mode of operation of ORB-SLAM 2 was SLAM mode which was used in mapping the environment surrounding. Shown in Fig. 5.6 is on operation of Localization Mode where based off the surroundings already mapped, it recognizes its location.

5.2.5 Trial 5

The loop closure feature of the SLAM implementation was verified and is shown in Fig. 5.7

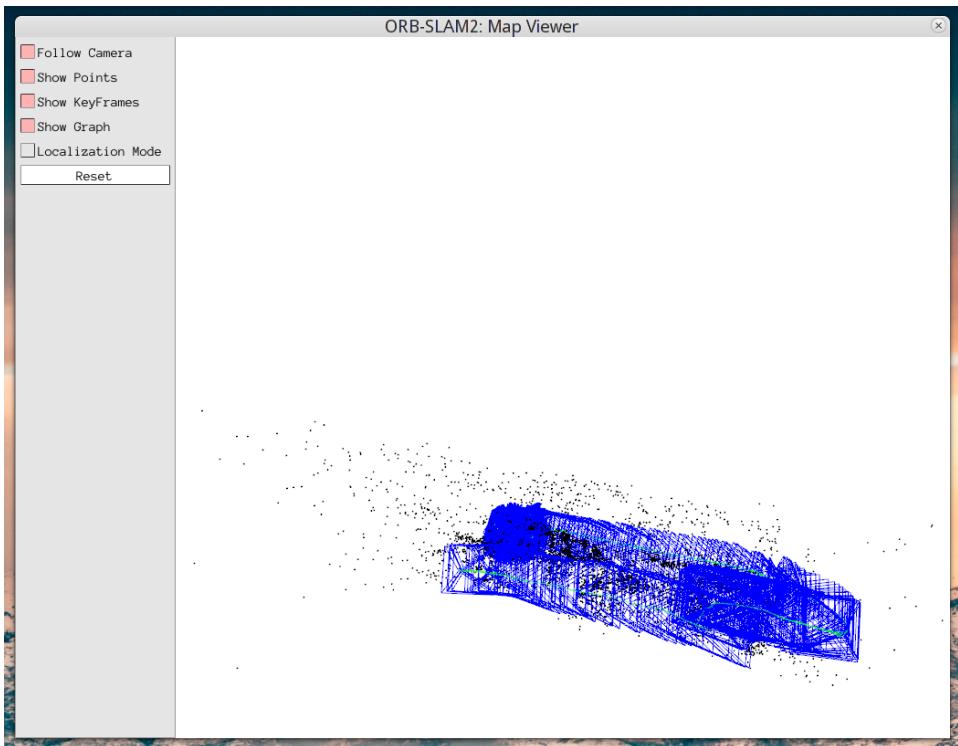


Figure 5.7: Trial 5 : Loop Closure

5.3 Bill Of Materials (BoM)

Adhering to the aim of building a low cost platform, the final cost for the system is approximately around \$100. The price split up is shown below.

<i>Item</i>	<i>Cost</i>
Raspberry Pi 2 & Accessories	\$ 40
Wi-Fi Dongle	\$ 10
Robot Body	\$ 25
Raspberry Pi Camera	\$ 24

Chapter 6

Conclusion

The system was successfully built. It worked to our expectations. The system transmitted monocular images at frame rate of 25 fps at 320x200 resolution and the trajectory was tracked in the server successfully.

The performance was as expected in scenarios where the environment had a lot of different types of object or textures. In scenarios with very similar or same texture, the system was not able to detect ORBs and failed to initialize. This was an expected result. As real world generally has a lot of objects and varied texture, this should not prove to be a problem.

Appendix A

Code Listing

Due to the high volume of code involved in the project, the code is hosted online at the following location :

<https://github.com/gapo/SLAMberry>

Appendix B

User Manual

Get IP by :

```
1 $ ip addr show
```

Now set Master IP in ROS Slave by editing the .bashrc file and setting the IP obtained from this step to the value in ROS_MASTER_URI. Now run ROS CAMERA Node on Pi:

```
1 $ rosrun rosberryypi_cam rosberryypi_cam_node
```

After running roscore and starting the camera node,

```
1 $ rosrun ORB_SLAM2 Mono
    ↳ ~/meng/deps/ORB_SLAM2/Vocabulary/ORBvoc.txt
    ↳ ~/meng/deps/ORB_SLAM2/pi_cam.yaml
```

Appendix C

Camera Calibration

To calibrate the camera, after running roscore and starting the camera node by

```
1 $ rosrun camera_calibration cameracalibrator.py --size 8x6
  ↳ --square 0.108 image:=~/rosberrypi_cam/image_raw
  ↳ camera:=/rosberrypi_cam
```

Appendix D

References

D.1 Hardware

1. Raspberry Pi Hardware : <https://www.raspberrypi.org/documentation/hardware/raspberrypi/README.md>
2. General Purpose Input/Output pins on the Raspberry Pi : <https://www.raspberrypi.org/documentation/hardware/raspberrypi/gpio/README.md>
3. CortexTM-A7 MPCoreTMTechnical Reference Manual : <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0464f/index.html>
4. Raspberry Pi Camera : <https://www.raspberrypi.org/documentation/hardware/camera/README.md>
5. Invensense 9250 IMU : <http://43zrtwysvxb2gf29r5o0athu.wpengine.netdna-cdn.com/wp-content/uploads/2015/02/MPU-9250-Datasheet.pdf>
6. Parallax Continous Rotation Servo : <https://www.parallax.com/sites/default/files/downloads/900-00008-Continuous-Rotation-Servo-Documentation-v2.2.pdf>

D.2 Software

1. ORB SLAM2 : https://github.com/raulmur/ORB_SLAM2
2. Pangolin : <https://github.com/stevenlovegrove/Pangolin>
3. DBow2 : <https://github.com/dorian3d/DBoW2>
4. g2o : <https://github.com/RainerKuemmerle/g2o>
5. Eigen : http://eigen.tuxfamily.org/index.php?title=Main_Page
6. ROS : <http://www.ros.org/>

Bibliography

- [1] R.C.Smith and P. Cheeseman., "On the representation and estimation of spatial uncertainty," *International Journal of Robotics Research*, vol. 5, no. 4, p. 5668, 1986.
- [2] P. Moutarlier and R.Chatila, "An experimental system for incremental environment modeling by an autonomous mobile robot," in *1st International Symposium on Experimental Robotics*, (Montreal), jun 1989.
- [3] H. Durrant-Whyte and T. Bailey, "Simultaneous localisation and mapping (slam): Part i the essential algorithms," 2006.
- [4] C. Harris and M. Stephens, "A combined corner and edge detector," in *Alvey vision conference*, vol. 15, p. 50, Citeseer, 1988.
- [5] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.
- [6] H. Bay, T. Tuytelaars, and L. Van Gool, *Computer Vision – ECCV 2006: 9th European Conference on Computer Vision, Graz, Austria, May 7-13, 2006. Proceedings, Part I*, ch. SURF: Speeded Up Robust Features, pp. 404–417. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006.
- [7] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, *Computer Vision – ECCV 2010: 11th European Conference on Computer Vision, Heraklion, Crete, Greece, September 5-11, 2010, Proceedings, Part IV*, ch. BRIEF: Binary Robust Independent Elementary Features, pp. 778–792. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.
- [8] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: an efficient alternative to sift or surf," in *Computer Vision (ICCV), 2011 IEEE International Conference on*, pp. 2564–2571, IEEE, 2011.

- [9] H. Strasdat, J. M. Montiel, and A. J. Davison, "Visual slam: why filter?," *Image and Vision Computing*, vol. 30, no. 2, pp. 65–77, 2012.
- [10] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon, "Bundle adjustment: modern synthesis," in *Vision algorithms: theory and practice*, pp. 298–372, Springer, 1999.
- [11] D. Gálvez-López and J. D. Tardós, "Bags of binary words for fast place recognition in image sequences," *IEEE Transactions on Robotics*, vol. 28, pp. 1188–1197, October 2012.
- [12] R. Mur-Artal, J. Montiel, and J. D. Tardos, "Orb-slam: a versatile and accurate monocular slam system," *Robotics, IEEE Transactions on*, vol. 31, no. 5, pp. 1147–1163, 2015.
- [13] V. Lepetit, F. Moreno-Noguer, and P. Fua, "Epnp: An accurate $O(n)$ solution to the pnp problem," *International journal of computer vision*, vol. 81, no. 2, pp. 155–166, 2009.
- [14] W. Garage, "Ros raspberry pi installataion instructions," 2016.