ABSTRACT

PANDYA, HARSH VIRENDRA. Mobile Manipulator based Framework for Dataset Generation and Algorithm Testing. (Under the direction of Dr. Edgar Lobaton).

Annually, around 50 million people get injured in automobile accidents and around 1.25 million die around the world [48]. Most of these casualties are due to preventable driver-borne causes like distracted driving, reckless driving, speeding, disobeying traffic signs, etc. Autonomous driving, let it be fully autonomous or assistive, can significantly reduce such human errors and hence reducing number of traffic accidents. In addition, such technology also reduces human effort, stress, and leads to better traffic management.

The core idea behind autonomous driving is to acquire sufficient amount of data from the surroundings of the vehicle using various sensor suites and then using data analysis and computer vision techniques to perceive, understand the environment, and make driving decisions based on that. The general development procedure for autonomous driving systems mainly consists of devising new algorithms/techniques and then analyzing them using benchmark datasets. While there are quite a few databases available in the research community consisting of depth images, stereo images, and LiDAR data, most of these are recorded from a single perspective. This thesis outlines a simple yet effective framework to automatically produce data streams for testing vision algorithms for autonomous driving applications based on depth images captured from multiple perspectives. We accomplish this by capturing depth images from multiple perspectives and location on a plane using the Microsoft Kinect sensor and a 6DOF robotic arm. Random trajectories on the plane are created and then datastreams are generated by selecting specific views from the dataset of depth images. These trajectories can then be used to perform statistical analysis of the performance of visual algorithms such as obstacle detection, vehicle tracking, etc. We illustrate the pipeline with a ground segmentation and tracking application.

Mobile Manipulator based Framework for Dataset Generation and Algorithm Testing

by
Harsh Virendra Pandya

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Electrical Engineering

Raleigh, North Carolina

2015

APPROVED BY:

_____	_____
Dr. Wesley Snyder	Dr. Edward Grant

_____
Dr. Edgar Lobaton
Committee Chair

DEDICATION

To my parents.

BIOGRAPHY

Harsh Pandya was born in the year 1991 in Vadodara, India. He completed his undergraduate degree (Bachelor of Engineering in Electronics) from The Maharaja Sayajirao University of Baroda in 2012.

Harsh began his graduate study at North Carolina State University in August 2013 working towards a Master of Science in Electrical Engineering with a focus on Robotics and Computer Vision.

ACKNOWLEDGMENTS

I would like to thank Dr. Edgar Lobaton, my advisor, for his guidance and support throughout my thesis research. Starting from the inception of this idea, nailing down the milestones to finishing up the writing, he has played a significant role in the advancement of my research.

Special thanks to Dr. Wesley Snyder for inspiring me to further my research. If it were not for you, I might have never embraced Computer Vision so much. I would also like to thank Dr. Edward Grant for teaching me the very bare bone essentials of robotics viz. Kinematics and Dynamics. I cannot imagine my research without the knowledge that I gained from both of you.

A special thanks to Jeremy Cole for clearing my doubts and helping me with hardware issues; thanks to Qian for her contribution to the implementation of RANSAC in MATLAB. I also thank Sahil, Somrita, Alireza, Daniel, and Namita for their cooperation in the lab.

Lastly and most importantly I thank my family: my parents, my wife, and my sister. Without their efforts and support I would not have reached this point in my life. This one is for you!

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

1.1 Motivation and problem overview

Traffic accidents retain to be a major source of disability and mortality worldwide. Every year, 1.25 million people die and up to 50 million people are injured [48, 45]. Autonomous or driver-assistive cars have the potential to reduce these numbers dramatically by the virtue of reducing human errors. Apart from accidental concerns, Urbanization is such a substantial demographic trend [41] that affects millions of people worldwide, which is particularly observable for numerous growing megacities [28]. This results in chronic track congestion, air pollution, slow speed in rush hours, and the struggle for finding empty parking spots, which has become overly annoying [47]. This has also lead general public, a majority of which includes commuters, to increasingly demand alternatives for their personal mobility, culminating in about one third of drivers being interested in buying an affordable self-driving car [5, 47]. A substantial aspect of tomorrows urban mobility is most likely going to be personal self-driving cars and autonomous car fleets that provide solutions for these challenges [35].

However, building a self-driving car that exceeds human driving performance is not easy. The distinct aspect of autonomous vehicles from normal intelligent vehicles is to be controlled by motion planning algorithms based on the information sensed by a machine not a human. Since decision making and actuator control are initiated by the sensing information, the quality and quantity of data about environments play a key role for fast and safe vehicle control [31, 24]. Thanks to the recent technology advances, the performance, convenience, and safety of modern vehicles has been greatly

improved [25]. Moreover, the maturity of this technology has reached a level enough to enable a fully autonomous vehicle to drive complying with urban traffic laws [10].

[21, 25] demonstrate that autonomous vehicles can be successful in challenging environments. The DARPA Grand and Urban Challenge competitions (2005, 2007) [23, 11] offer a modern, uniform testing opportunity to examine the state-of-the-art in autonomous cars.

As mentioned before, work on self-driving cars spans several decades [15, 37, 22, 9]. Especially in the past decade considerable amount of work has been done in the area of autonomous driving. The basic idea revolves around using a sensor suite for collecting data about the surroundings of the car and then classifying/tracking/detecting various elements in the environment. The said sensor suite ranges from expensive LiDAR systems to inexpensive stereo camera systems based on the level of autonomy and accuracy demonstrated by the overall system. For good and sufficient sensing information, there are two primary approaches: long range and wide angle sensor equipment, and distributed perception [32, 40]. The high performance sensors provide an immediate response sensing time or large area sensing capability, but whose price is prohibitively expensive for economic viability, and sensing area is limited by line-of-sight [24].

The general development procedure for autonomous driving systems mainly consists of devising new algorithms/techniques and then analyzing them using benchmark datasets. While there are quite a few databases available in the research community consisting of depth images, stereo color images, and LiDAR data, most of these are recorded from a single perspective. We present a simple yet effective framework to automatically produce datastreams for testing vision algorithms for autonomous driving applications based on depth images captured from multiple perspectives.

A dataset consisting of multiple perspective data streams serves two major purposes when used for algorithmic testing:

1. **Evaluating the sensitivity of an algorithm/technique to trajectory perturbations/disturbances.** It is quite common for vehicles to not follow a smooth trajectory on the roads due to several factors like unfavorable weather conditions, ill-conditioned roads, spills, obstacles, other vehicles, change in overall traffic speed, etc. It is such situations that, to some extent, contribute to human errors, resulting into an accident. The same can be applied to autonomous vehicles; when they experience a perturbation in the trajectory of their motion, the data streams captured by their sensor suite lose continuity. This might lead to the failure of one or more algorithms (e.g. obstacle tracking failure as an obstacle fled the field of view of the camera). A dataset with multiple perspectives can be used to simulate such perturbations and test any given algorithm/technique to find out whether it fails in such situations.
2. **Simulating a multi-camera sensor suite.** A viable alternative to the expensive state-of-the-art sensor suites (e.g. LiDAR, RADAR) is a sensor setup consisting of multiple cameras (in most cases mounted on top of the vehicle). The idea here is to maximize the amount of data collected from the surroundings while minimizing the data to be processed to bare minimum. Data stream from all cameras is pre-processed to figure out which ones contain maximum vital information required to make important decisions for autonomous driving e.g. a camera capturing multiple obstacles or cars at any given frame is providing more information compared to the one which shows an empty road. The system then needs to analyze image stream only from a smaller subset of cameras at any given time, reducing the amount of data to be processed in real time. Our multi perspective dataset can be used to simulate such a multi-camera system where each single perspective data stream acts as an individual camera.

Techniques/algorithms for such system can be easily tested using our multi-perspective dataset.

In order to generate our dataset, we use a Mobile Manipulator equipped with a 6 DOF arm for manipulating the position and orientation of a Microsoft Kinect for capturing depth images from multiple perspectives. ROS (Robot Operating System) forms the backbone of our software framework providing necessary communication between various hardware components. Further the registered depth and brightness images captured are processed using Kinect View – a Matlab based GUI program for data extraction, path selection, and testing techniques used in autonomous driving. Kinect View is capable of creating random trajectories on the plane and generating datastreams by selecting specific views from the dataset of depth images.

We make use of a RANSAC-based plane fitting ground segmentation technique to demonstrate the capabilities of this framework. The program can be similarly used for testing other techniques involved in autonomous driving such as obstacle detection, vehicle tracking, etc.

1.2 System Overview and Contribution

The block diagram below provides a bird's eye view of our framework and the dataflow within it:

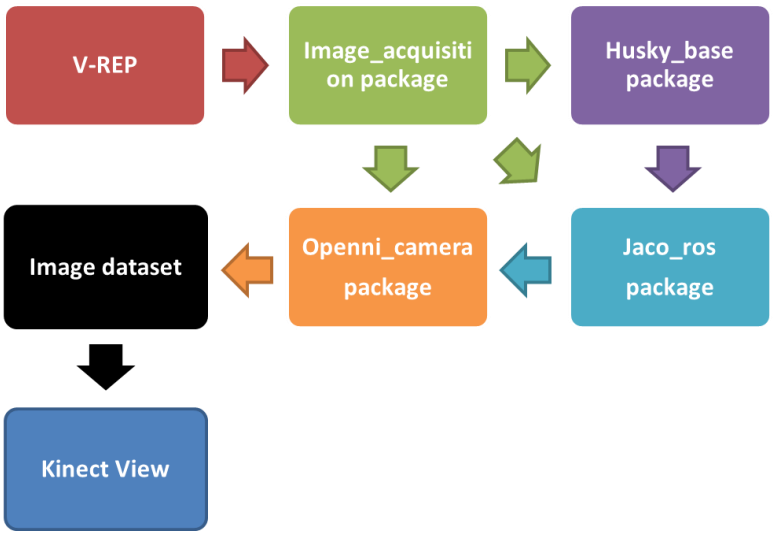


Figure 1.1: System Overview

As shown in Figure 1.1, we start with simulating our hardware platform on V-REP. Various Kinematic experiments were performed to establish proper working of this step in the pipeline. The hardware platform for specific experimental setup was simulated in V-REP and motion planning data pertaining to Jaco arm and Husky base was collected. This data is then sent to ROS for actual hardware control. Image_acquisition package in ROS manages this data and activates nodes in other packages to control individual hardware components including Husky robot, Jaco arm, and Kinect. Image_acquisiton package also stores depth, RGB, and disparity data collected. This data is then sent to our MATLAB based Kinect View program that

performs analysis and tests algorithmic performance. Original work done during the course of this thesis includes:

- V-REP development: Created various simulation experiments on V-REP, wrote scripts for individual objects in the scenes, motion planning, and data extraction.
- Created Image_acquisition package in ROS to facilitate the communication between hardware driver packages like openni_camera, husky_base, and jaco_ros.
- New nodes to facilitate image capturing using openni.
- Kinect camera calibration.
- MATLAB based GUI for Kinect View.
- Backend scripts to control the Kinect View behavior dynamically.
- Random path generation using biased graph walk with data stream extraction.
- RANSAC plane fitting for ground segmentation.

Subsequent chapters cover the above procedures, analysis and results in detail.

1.3 Thesis Organization

Chapter 1 introduced the autonomous driving problem, general approach to resolve it, and our platform for data collection and testing various techniques. Upcoming chapters are organized as follows:

Chapter 2 describes the hardware platform used in our framework. It describes the basic properties and features expected from the hardware platform, introduces individual elements in the system, and fleshes out how they fit the role.

Chapter 3 introduces the software platforms used for our framework. It goes into explaining the GUI based simulation software V-REP used for scene setup and motion planning. It also introduces ROS, the backbone of our framework, and various packages used for interacting with individual hardware components.

Chapter 4 goes into motion planning and data collection with several experimental setups. Moreover, it also details out the flow of data, and the underlying pipeline of operation, how individual pieces interact with each other and how final dataset is acquired.

Chapter 5 introduces ground segmentation as a means of exploring the proof of concept for the said framework. We use a RANSAC based plane fitting approach for ground plane fitting and eventual segmentation.

Chapter 6 discusses the MATLAB based GUI program called Kinect_View, developed for analyzing the data captured using our framework. This chapter explains the program architecture and various functionalities in detail. Kinect_View is highly modular, in that one can plug in any image processing algorithm/technique for testing and analysis.

Finally Chapter 7 concludes the thesis by analyzing the results obtained, enumerating the salient features, expansion capabilities, and future work for this system.

Chapter 2

Hardware Platform

This Chapter describes the Hardware Platform created for the said framework. The platform was created with following qualities in mind:

- Mobility – The platform needs to emulate a vehicle to certain extent in order to collect data sets for testing autonomous driving techniques. Hence it should be fairly mobile.
- Agility – The platform has to be agile with precise movement control as the orientation and position of camera while recording frames has to be accurate. High level of controllability is desirable.
- Ease-of-interface – Each individual component of the hardware platform should be able to communicate with other components directly or indirectly using some common platform.
- Remote and local control – Finally, It is desirable to have an onboard computing unit on the platform for local control and analysis. It should also be possible to control and collect data remotely.

Figure 2.1 shows the assembled hardware platform used for our framework. Various individual components labelled in figure are discussed in further sections in detail.

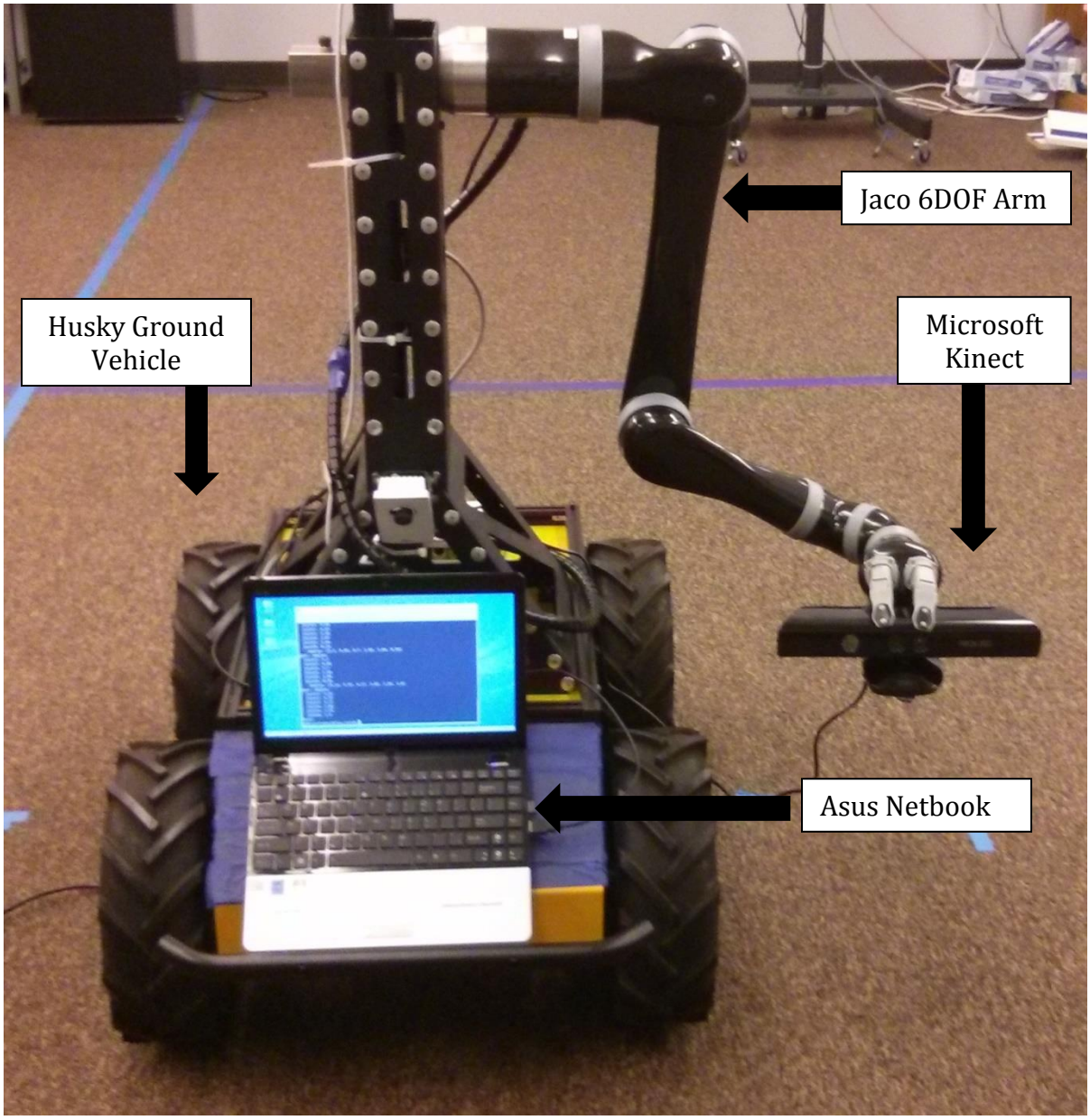


Figure 2.1: Hardware Platform

2.1 Husky Unmanned Ground Vehicle

Husky A200 robot from Clearpath Robotics is a rugged and easy-to-use unmanned ground vehicle for rapid prototyping and research applications. Figure 2.2 provides an isometric view of Husky. The robot is extremely basic, in that it comes with only the base: wheels, motors, chassis, battery, and a simple lockdown mechanism. Yet its rugged nature, payload capacity and agility make it a perfect base for our manipulator platform.

Figure 2.2: Husky A200 Isometric view and reference frame

We follow a Cartesian reference frame based on ISO 8855 for non-holonomic motion, and is shown in Figure 2.2. In terms of dimension it is 990mm long, 670mm wide and 390mm high. The Husky A200 weighs about 50kg and has a payload capacity of 75kg. It can go up to 1m/s maximum speed and climb up to 45°.

Figure 2.3 gives a tour of key Husky components. The Husky A200 has a RS-232 serial port, which is the only means of out of the box communication. We added a generic RS-232 to USB convertor for facilitating easier communication. While it does not feature any kind of sensor to aid our application, it does have internal sensors for battery status, odometry, motor currents, and motor encoders.

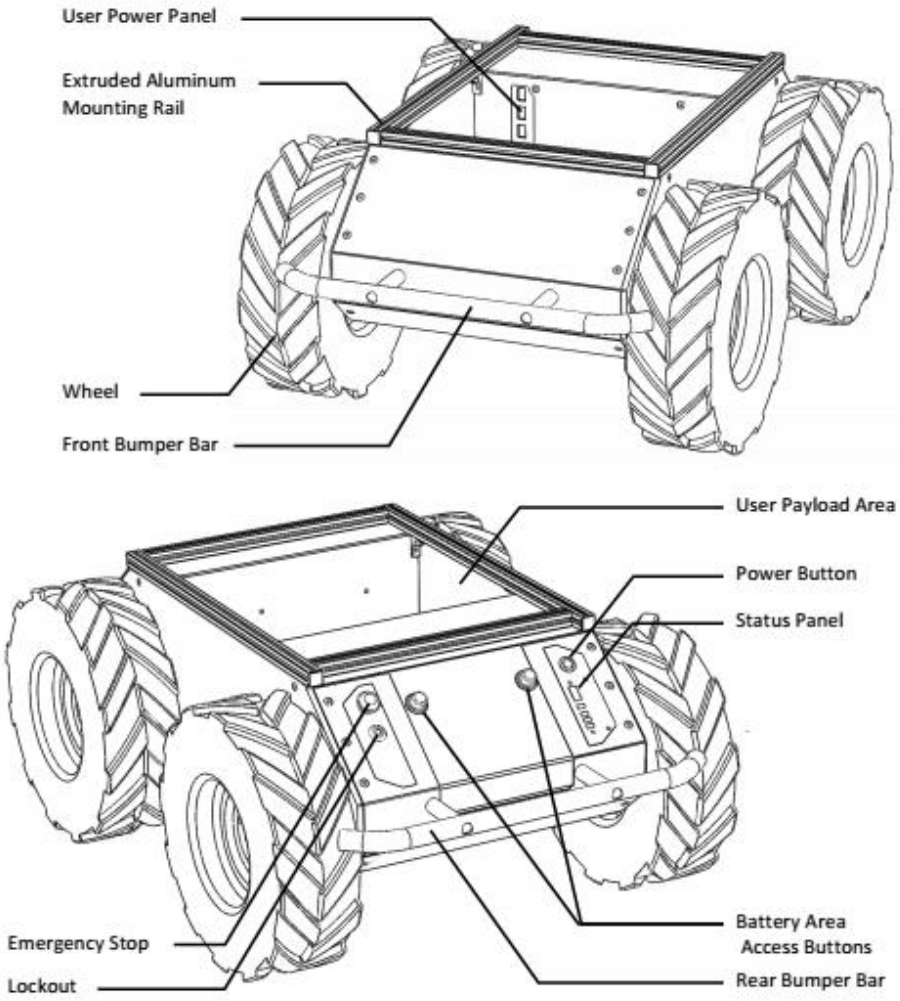


Figure 2.3: Husky A200 front and rear with component labels

The relationship between wheel velocity and platform velocity is governed by:

$$v = \frac{v_l + v_r}{2},\ \omega = \frac{v_r - v_l}{w}$$

where v represents the instantaneous translational speed of the platform and ω the instantaneous rotational speed. v_r and v_l are the right and left wheel velocities, respectively. w is the effective track of the vehicle (0.555m).

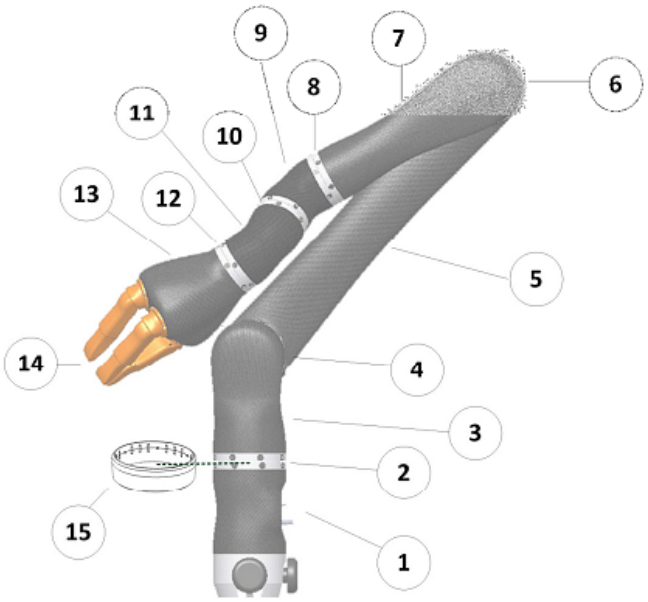
Husky supports four distinct control modes:

- Kinematic Control, the default, uses a speed control feedback loop, and allows specifying the desired linear and angular speed of Husky.
- Torque Control, the linear and angular values specified are multiplied by parameterized scaling factors and used for a current control feedback loop.
- Raw Control works similarly to torque control, but disables feedback control altogether, instead using the scaling values to specify open-loop voltage for the motor drivers. This mode may be useful for novel control algorithms on the PC, or for debugging a suspected wheel encoder failure.

2.2 Kinova Jaco Robotic Arm

The JACO arm is a light-weight, technical assistance robot designed to mimic mobility of the upper limbs. It is composed of six inter-linked segments, the last of which is a three-fingered hand. The robot's hand is capable of movement in three dimensional space and grasp or release objects with the hand. The JACO arm has a weight of 5.6kg, can reach approximately 90cm in all directions and can lift objects of up to 1.5kg.

The six joints of the JACO, as shown in Figure 2.4, are individually driven by six DC geared servomotors located in each joint. Each motor module includes a planetary gearhead. Two types of motor modules are used depending on joint location in the kinematic chain. Joints 1–3, where joint 1 the nearest to the base, use large motor modules while joints 4–6 use small motor modules [13].



Part ID	Name
1	Fixed Base
2	Joint #1
3	Base
4	Joint #2
5	Arm
6	Joint #3
7	Forearm
8	Joint #4
9	Wrist #1
10	Joint #5
11	Wrist #2
12	Joint #6
13	Hand
14	Fingers
15	Plastic Ring

Figure 2.4: Jaco Arm with various parts labelled

Within a single manipulator, each motor module is interchangeable in terms of its position based on its class. That is to say, a small motor module can replace any motor module of joints 4–6 without any effect on the performance of the arm. The carbon-fiber links house the series of motor modules at the links while wiring is routed through the hollow members. Each finger in the end effector is driven by an individual motor bringing the sum of motors to 9 [13].

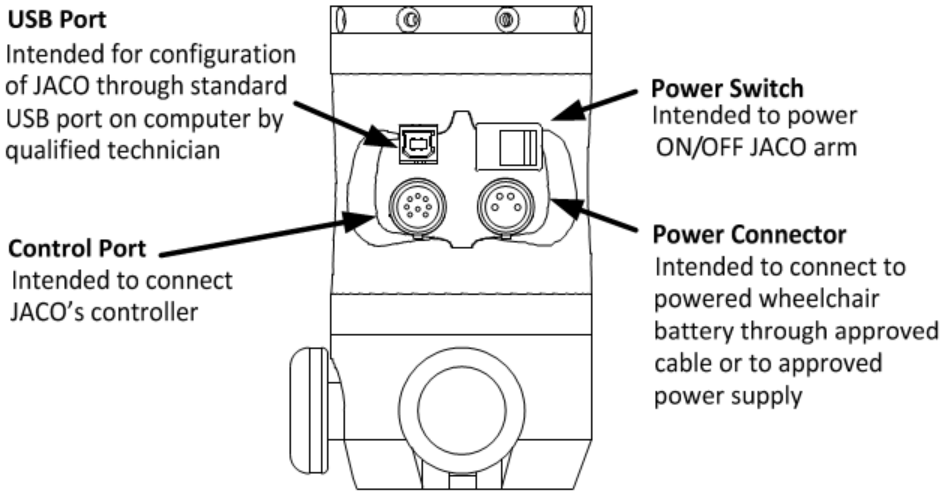


Figure 2.5: Jaco Arm base with connectors

The base of the JACO houses the digital signal processor (DSP) which sends information to each motor joint and finger motor based on user input. Power and input devices are plugged into the base of the arm, the configuration of which is then recognized by the control electronics. Figure 2.5 shows the external connectors located on the JACO arm fixed base.

2.3 Microsoft Kinect

Originally intended to be used as an input device for Microsoft's Xbox 360 gaming console, the Kinect (Figure 2.6) contains a diverse set of sensors. Most notable of those sensors being a depth camera based on infrared structured light technology [44]. With a proper calibration of its color and depth cameras, the Kinect can capture detailed color point clouds at up to 30 frames per second. This capability uniquely positions the Kinect for use in fields other than entertainment, such as robotics, natural user interfaces, and three-dimensional mapping.

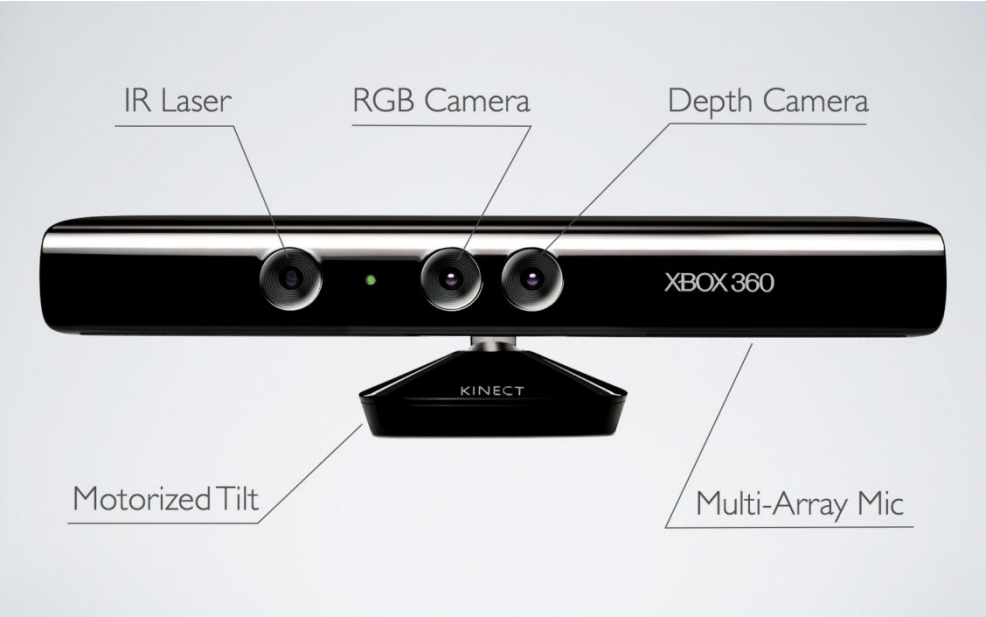


Figure 2.6: Microsoft Kinect

Microsoft aptly describes the Kinect as a "sensor array" since it contains more than a depth camera [16]. A teardown of the Kinect reveals that its full hardware complement includes a structured light depth camera, a color camera, a microphone array, a three-axis accelerometer, and a tilt motor as shown in Fig. 2.6. The Kinect's

camera system consists of paired CMOS infrared and color cameras and an infrared structured light projector. Both cameras sit in the Kinect's middle on either side of its standoff whereas the structured light projector is displaced towards the Kinect's left side (as viewed from the front). Using stereo calibration, the cameras are measured as roughly 2.5cm apart, and a similar method measures the separation between the structured light projector and infrared camera as roughly 7.5cm [12, 26].

Microsoft's official Kinect programming guide specifies the system's field of view as 43° vertical by 57° horizontal, the color and depth image sizes as VGA and QVGA respectively, the "playable range" as 1.2m to 3.5m, and the frame rate as 30Hz [33]. Based upon the PrimeSensor Reference Design specifications, the Kinect's expected depth resolution at 2m is 1cm [38]. The Kinect's infrared structured light projector emits the pattern resembles the illumination scheme described by [18], one of PrimeSense's depth imaging patents. According to the patent, the pattern consists of pseudo-random spots such that any particular local collection of spots (a "speckle feature") is uncorrelated with the remainder of the pattern [16]. Thus, a known speckle feature is uniquely identifiable and locatable in an image of the structured light pattern.

In addition to the color and depth camera system, the Kinect includes an array of four microphones, a three-axis accelerometer, and a tilt motor. The microphone array is capable of beamforming, source localization, and speech recognition with the assistance of a host computer, although the Kinect can perform on-board echo cancellation and noise suppression [33, 16]. The tilt motor can angle the Kinect ±31° from horizontal and works in conjunction with the accelerometer to level the Kinect on an angled surface [36].

The Kinect's hardware interface consists of a proprietary type-A USB connector which provides a +12V supply in addition to USB connectivity. We use a converter to the standard type-A USB connector that draws extra power from a wall adapter. The Kinect thus effectively requires separate connections for data and power.

2.4 Asus Eee 1215N Netbook

As a local computing unit for our hardware platform, we use a lightweight Asus 1215N netbook. The Asus 1215N netbook runs on a 1.8GHz Intel® Atom™ D525 (Dual Core) Processor. It has 1GB DDR1 RAM with 250GB SATA hard drive. Wireless connectivity in the form of WLAN 802.11 a/b/g/n and Bluetooth V3.0+HS is available. It also provides various interface options like 1 x VGA Connector, 1 x USB 2.0, 2 x USB 3.0, 1 x LAN RJ-45, and 1 x HDMI.

Figure 2.7: Asus Eee 1215N

We utilize 2 x USB 3.0 ports for connections with Kinect and Jaco arm, as they require more frequent communication to and from ROS, while the USB 2.0 port is used for the Husky base which receives less frequent commands from ROS.

All the preinstalled software packages, including the Operating System, were removed and a fresh Ubuntu 12.04.5 LTS (Precise Pangolin) was installed. To support the said framework development, following software packages were installed:

- ROS Hydro Medusa (Robot Operating System 7)
- V-REP 3.2.0 (Virtual Robot Experimentation Platform)
- OpenCV 2.4.4

Above mentioned packages and their utilization in this project are discussed in detail in entailing chapters.

Chapter 3

Software Platform

As mentioned in previous chapter, ease-of-interface among various components was one of the basic requirements for this platform. We chose Robot Operating System (ROS) Hydro as the backbone for our software framework, while Virtual Robot Experimentation Platform (V-REP) was used extensively for simulations and motion planning.

3.1 Virtual Robot Experimentation Platform (V-REP)

V-REP is like a Swiss army knife among robot simulator platforms, it is equipped with a plethora of functions, features, and elaborate APIs. V-REP PRO EDU license is free to use, fully functional simulation platform available to use for non-commercial, educational, and research purposes.

The robot simulator V-REP, with integrated development environment, is based on a distributed control architecture: each object/model can be individually controlled via an embedded script, a plugin, a ROS node, a remote API client, or a custom solution [19]. This makes V-REP very versatile and ideal for multi-robot applications. Controllers can be written in C/C++, Python, Java, Lua, Matlab, Octave or Urbi.

Following are a few of V-REP's applications:

- Simulation of factory automation systems
- Remote monitoring
- Hardware control
- Fast prototyping and verification
- Safety monitoring
- Fast algorithm development
- Robotics related education
- Product presentation

V-REP can be used as a stand-alone application or can easily be embedded into a main client application: its small footprint and elaborate API makes V-REP an ideal candidate to embed into higher-level applications. An integrated Lua script interpreter makes V-REP an extremely versatile application, leaving the freedom to the user to combine the low/high-level functionalities to obtain new high-level functionalities.

[19] V-REP offers various means for controlling simulations or even to customizing the simulator itself (Figure 3.1). V-REP is wrapped in a function library, and requires a client application to run. The V-REP default client application is quite simple and takes care of loading extension modules, registers event callbacks (or message callbacks), relays them to the loaded extension modules, initializes the simulator, and handles the application and simulation loop.

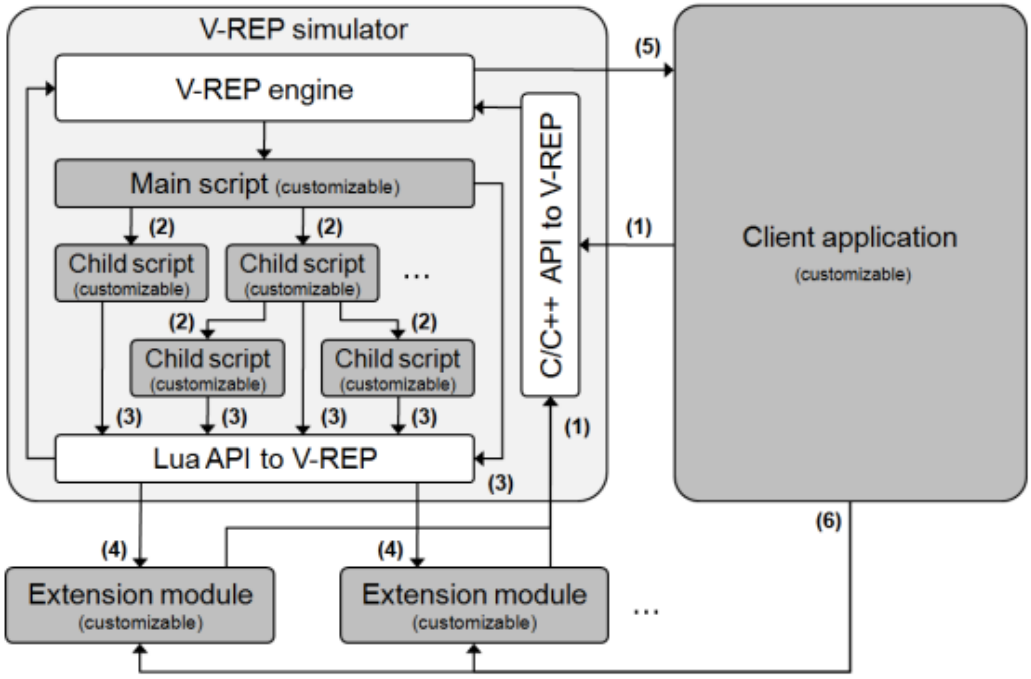


Figure 3.1: Control architecture in V-REP. Greyed areas are user customizable

In addition, custom simulation functions can be added via:

- **Scripts in the Lua language.** Lua [1] is a lightweight extension programming language designed to support procedural programming. The Lua script interpreter is embedded in V-REP, and extended with several hundreds of VREP specific commands. Scripts in V-REP are the main control mechanism for a simulation.
- **Extension modules to V-REP (plugins).** Extension modules allow for registering and handling custom commands. An extension module can handle this high-level custom command by executing the corresponding logic and low-level API function calls in a fast and hidden fashion.

3.1.1 Scene creation and model setup

Each new simulation in V-REP begins with a scene. Simulations in V-REP are hierarchical where a Scene can be considered as the root of the hierarchy. This scene in turn may contain various objects, models, tasks, and scripts. Figure 3.2 shows the structure and components of a typical scene in V-REP.

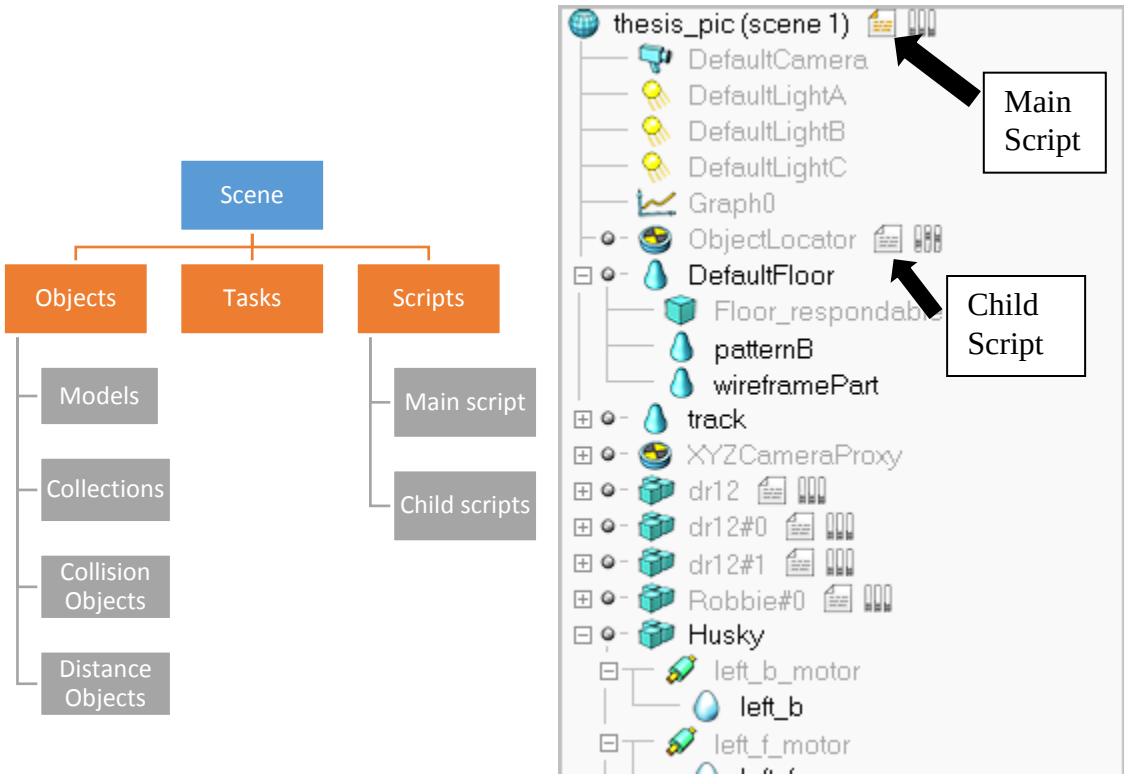
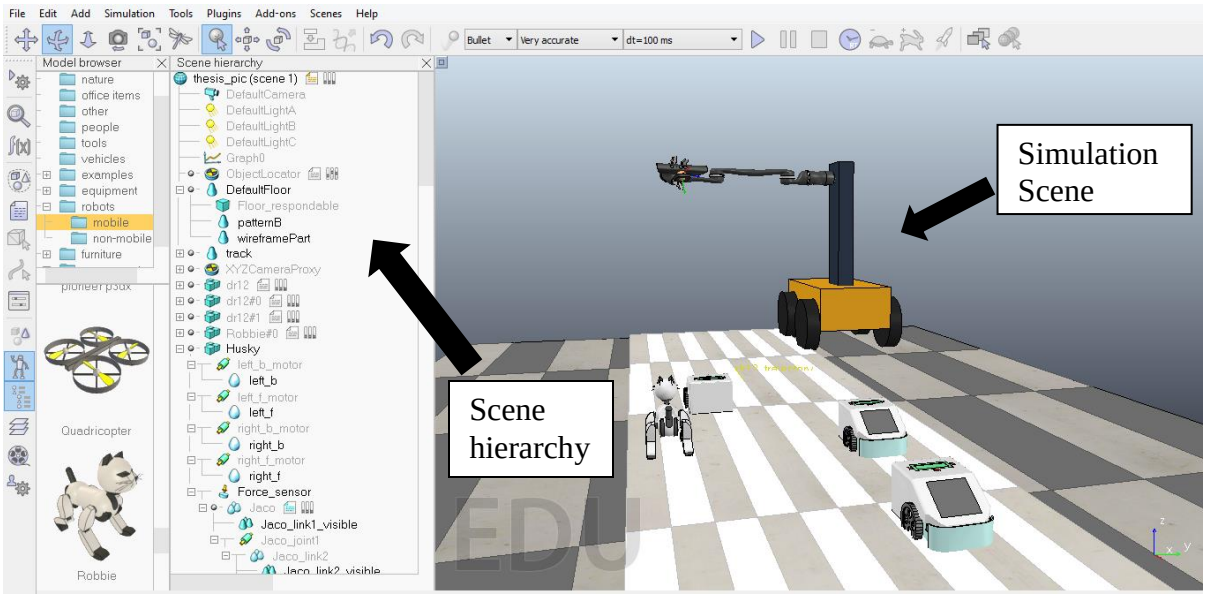


Figure 3.2: Scene hierarchy in V-REP



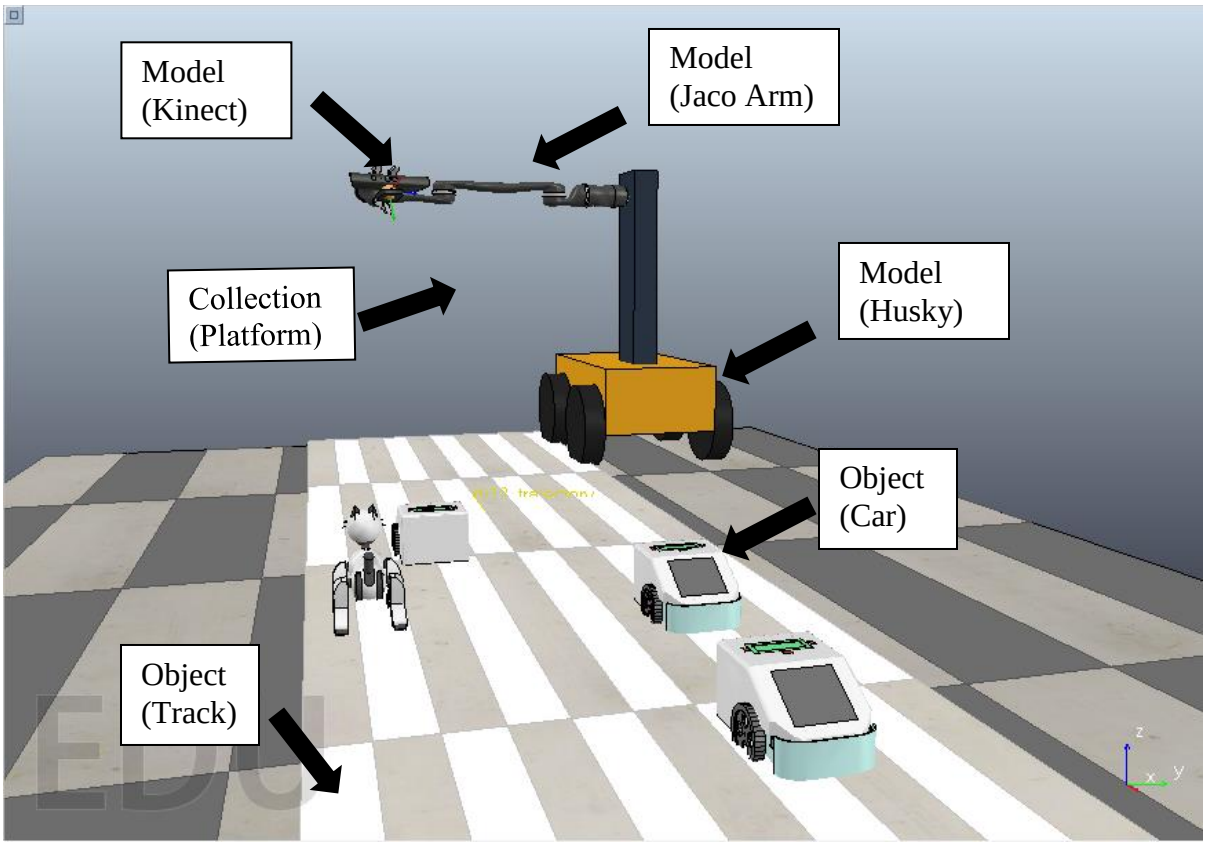


Figure 3.3: Top: V-REP main screen; Bottom: A scene in V-REP with various components

One of the scenes created for the framework is shown in figure 3.3. This scene consists of several objects, collections, scripts, and models. Some of the models were readily available for use (e.g. Jaco Arm), while others were custom created from basic shapes, assigning physical properties like density, material type, center of mass, etc. (e.g. Husky robot and tower assembly). Models or objects can be dragged and dropped from Model browser. The scene can be rotated, tilted, or dragged using camera navigation toolbar. Individual objects can be accessed by single clicking in either the scene or the hierarchy. Object manipulation tool can be used to rotate or translate selected object in 3D space.

A simulation is handled when the client application calls a main script, which in turn can call child scripts. Each simulation scene has exactly one main script that handles all default behaviour of a simulation, allowing simple simulations to run without even writing a single line of code. The main script is called at every simulation pass and is non-threaded.

Child scripts on the other hand are not limited in number, and are associated with (or attached to) scene objects. As such, they are automatically duplicated if the associated scene object is duplicated. In addition to that, duplicated scripts do not need any code adjustment and will automatically fetch correct object handles when accessing them. Child scripts can be non-threaded or threaded (i.e. launch a new thread).

The default child script calling methodology is hierarchical; each script is in charge of calling all first encountered child scripts in the current hierarchy (since scene objects are built in a tree-like hierarchy, scripts automatically inherit the same hierarchy). This is achieved with a single function call: simHandleChildScript (sim handle all).

3.1.2 Calculation Modules

Scene objects are rarely used on their own, they rather operate on (or in conjunction with) other scene objects (e.g. a proximity sensor will detect shapes or dummies that intersect with its detection volume). In addition, V-REP has several calculation modules that can directly operate on one or several scene objects. Following are V-REP's main calculation modules [19]:

- **Forward and inverse kinematics module:** allows kinematics calculations for any type of mechanism (branched, closed, redundant, containing nested loops, etc.). The module is based on calculation of the damped least squares pseudoinverse [46]. It supports conditional solving, damped and weighted resolution, and obstacle avoidance based contraints.
- **Dynamics or physics module:** allows handling rigid body dynamics calculation and interaction (collision response, grasping, etc.) via the Bullet Physics Library [2].
- **Path planning module:** allows holonomic path planning tasks and non-holonomic path planning tasks (for car-like vehicles) via an approach derived from the Rapidly-exploring Random Tree (RRT) algorithm [30].
- **Collision detection module:** allows fast interference checking between any shape or collection of shapes. Optionally, the collision contour can also be calculated. The module uses data structures based on a binary tree of Oriented Bounding Boxes [20] for accelerations. Additional optimization is achieved with a temporal coherency caching technique.
- **Minimum distance calculation module:** allows fast minimum distance calculations between any shape (convex, concave, open, closed, etc.) or collection of shapes. The module uses the same data structures as the collision detection module. Additional optimization is also achieved with a temporal coherency caching technique.

Except for the dynamics or physics modules that directly operate on all dynamically enabled scene objects, other calculation modules require the definition of a calculation task or calculation object, that specifies on which scene objects the module should operate and how. If for example the user wishes to have the minimum distance between shape A and shape B automatically calculated and maybe also recorded, then a minimum distance object has to be defined, having as parameters shape A and shape B. Figure 3.4 shows V-REP's typical simulation loop, including main scene objects and calculation modules.

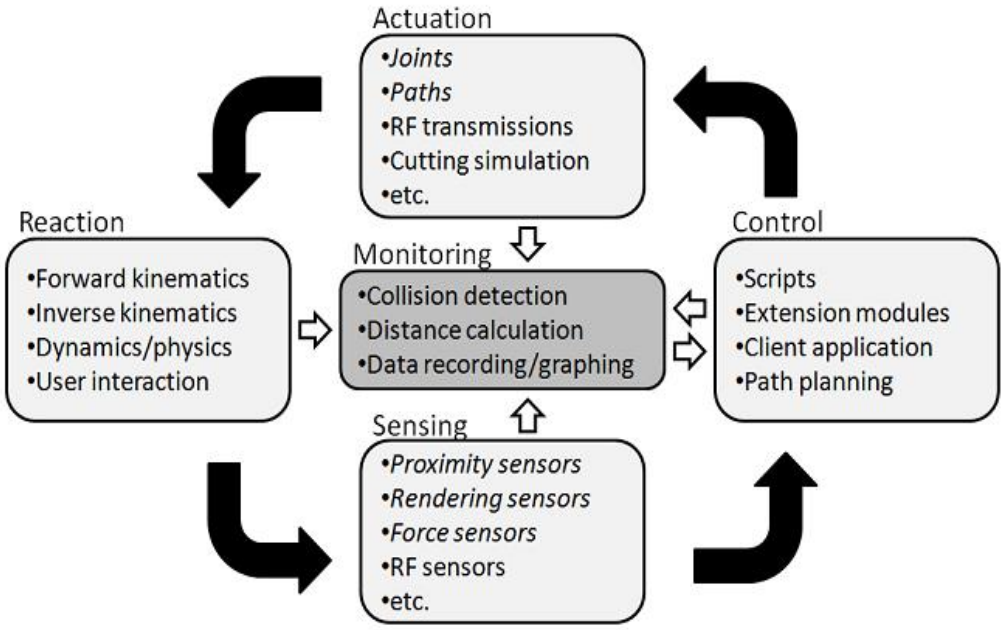


Figure 3.4: Simulation Loop in V-REP

3.2 Robot Operating System (ROS)

In spite of rapid progress in the field of robotics in the past few decades, robots still present significant challenges for software developers. From the robot's perspective, problems that seem trivial to humans often vary wildly between instances of tasks and environments. Often robotic systems, tackling real life problems, require complex integration of sensor suites, one or more robotic platforms, and various actuators. A common framework/platform built upon features like:

- Solid communication Infrastructure
- Robot-Specific features
- Elimination of programming language barrier
- Diagnostic tools
- Advance Simulation capabilities

can eliminate most, if not all, aforementioned challenges in robotic software development.

With a mission of creating truly robust, general-purpose robot software, Willow Garage came up with a software platform called Robot Operating System, or ROS that is intended to ease some of these difficulties. The official description of ROS is:

> *"ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers."*

This description is accurate — and it correctly emphasizes that ROS does not replace, but instead works alongside a traditional operating system. The core strength of ROS lies in its communication network.

Before we describe the communication protocol and how data passes along various aspects of ROS, here are a few key concepts one must know [14]:

- **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. Packages are the most atomic build item and release item in ROS.
- **Nodes:** Nodes are process that can perform computation, execute some tasks and communicate thanks to the ROS network. Each node registers to the network with a unique id and a list of topics and services that it wants to send or receive messages from and some additional connection parameters. ROS provides libraries to write the nodes with the C++ or the Python languages.
- **Master:** The master is a special node that is launched every time ROS is started. It handles the registration, subscriptions and disconnection of every node to the network and links for each topic or service so that messages will be able to reach its target successfully. 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.
- **Messages:** Packets sent on the network are defined in ROS 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 (Figure 3.5). In

general, publishers and subscribers are not aware of each other's 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 (Figure 3.5), 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.

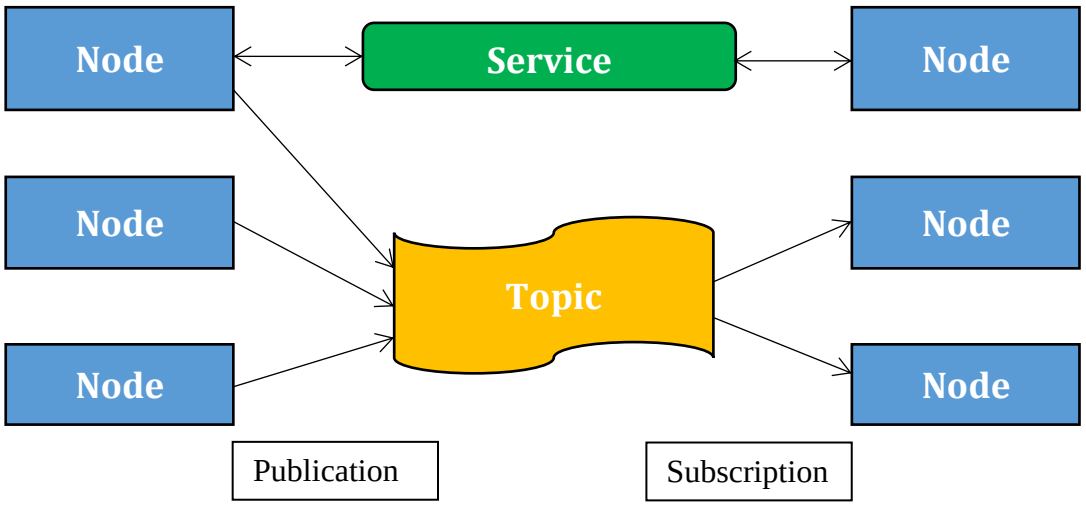


Figure 3.5: Communication in ROS among nodes

[14] The ROS Master acts as a nameservice in the ROS Computation Graph. It stores topics and services registration information for ROS nodes. Nodes communicate with the Master to report their registration information. As these nodes communicate with the Master, they can receive information about other registered nodes and make connections as appropriate (Figure 3.6). The Master will also make callbacks to these nodes when the registration information changes. This allows nodes to dynamically create connections as new nodes are run.

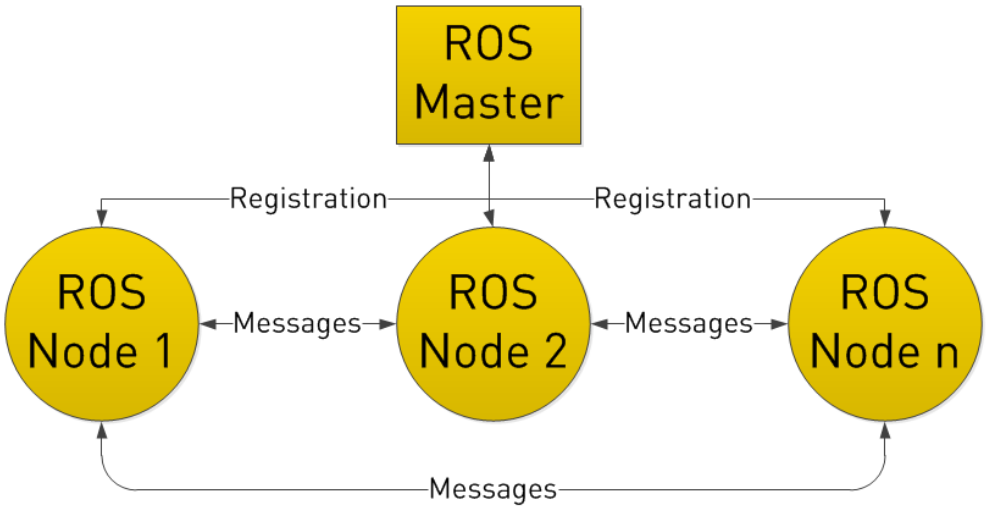


Figure 3.6: Role of Master in ROS communication

Nodes connect to other nodes directly; the Master only provides lookup information, much like a DNS server. Nodes that subscribe to a topic will request connections from nodes that publish that topic, and will establish that connection over an agreed upon connection protocol. The most common protocol used in a ROS is called TCPROS, which uses standard TCP/IP sockets.

This architecture allows for decoupled operation, where the names are the primary means by which larger and more complex systems can be built. Names have a very important role in ROS: nodes, topics, services, and parameters all have names. Every ROS client library supports command-line remapping of names, which means a compiled program can be reconfigured at runtime to operate in a different Computation Graph topology.

The rest of this chapter focusses on individual ROS packages used to control individual hardware components of our platform. These packages are described in details pertaining to various nodes, topics, and services they offer.

3.2.1 jaco_ros package

The ROS JACO Arm metapackage provides a ROS interface for the Kinova Robotics JACO robotic manipulator arm. This metapackage provides access to the Kinova JACO C++ hardware API through ROS.

The JACO API is exposed to ROS using a combination of actionlib (for sending trajectory commands to the arm), services (for instant control such as homing the arm or e-stop) and published topics (joint feedback). The arm may be commanded using either angular commands or cartesian co-ordinates. In addition, a transform publisher enables visualization of the arm via rviz.

There are three actionlib modules available: arm_pose/arm_pose, joint_angles/arm_joint_angles, and fingers/finger_position. These server modules accept coordinates which are passed on to the Kinova JACO API for controlling the arm.

The services available are: in/home_arm, in/stop, and in/start. These services require no input goals, and are intended for quick control of basic arm functions. When called, home_arm will halt all other movement and return the arm to its "home" position. The stop service is a software e-stop, which instantly stops the arm and prevents any further movement until the start service is called.

Published topics are: out/cartesian_velocity, out/joint_velocity, out/finger_position, out/joint_angles, out/joint_state, and out/tool_position. The cartesian_velocity and joint_velocity are both subscribers which may be used to set the joint velocity of the arm. The finger_position and joint_angles topics publish the raw angular position of the fingers and joints, respectively, in degrees. The joint_state topic publishes via sensor_msgs the transformed joint angles in radians. The tool_position topic publishes the Cartesian co-ordinates of the arm and end effector via geometry_msgs.

The jaco_arm_driver node acts as an interface between the Kinova JACO C++ API and the various actionservers, message services and topics used to interface with the arm. The jaco_tf_updater node subscribes to the jaco_arm_driver node to obtain current joint angle information from the node. It then publishes a transform which may be used in visualization programs such as rviz.

In our case none of the inverse kinematic topic subscribers seemed to work, hence we decided to use V-REP as our inverse kinematics and motion planning tool. Details about message structure and parameters for each of the above mentioned topics and services can be found at [7]. Some of the basic services and topics used most frequently are:

- To “home” the arm
  rosservice call /jaco_arm_driver/in/home_arm
- To activate the emergency stop function
  rosservice call /jaco_arm_driver/in/stop
- To restore control of the arm
  rosservice call /jaco_arm_driver/in/start
- To obtain the raw joint angles in degrees
  rostopic echo jaco/joint_angles
- To obtain the finger angles in degrees
  rostopic echo jaco/finger_position

- To obtain the arm's position in Cartesian units
  rostopic echo jaco/tool_position

3.2.2 openni_camera package

A ROS package that contains OpenNI driver for RGB+D cameras that includes Microsoft Kinect, PrimeSense PSDK, ASUS Xtion Pro and Pro Live. This package publishes raw depth, RGB, and IR image streams using the ROS message-passing system. OpenNI, in itself, is the largest 3-D sensing development framework and community that provides an open source SDK.

There are two formats of the depth channel output — disparity map or the real depth in millimetres. The sensor returns the disparity value and the driver is by default configured to convert this value to the real depth. The disparity of the pixel represents the shift of a point in two stereoscopic images [39] (analogically for the pattern projected by the IR projector and captured by the IR camera). In the PrimeSense terminology, the disparity is called a shift value.

This package also contains launch files for using OpenNI-compliant devices such as the Microsoft Kinect in ROS. It creates a nodelet graph to transform raw data from the device driver into point clouds, disparity images, and other products suitable for processing and visualization. Openni_launch launches in one process the device driver and many processing nodelets for turning the raw RGB and depth images into useful products, such as point clouds. Provides default tf tree linking the RGB and depth cameras [4].

image_view is a simple viewer for ROS image topics. It can be used to visualize RGB, depth, and disparity image topics.

RGB image:

rosrun image_view image_view image:=/camera/rgb/image_color

Figure 3.7: image_view RGB

Depth image:

rosrun image_view image_view image:=/camera/depth/image

Figure 3.8: image_view Depth

image_view also contains the disparity_view viewer for sensor_msgs/DisparityImage messages:

rosrun image_view disparity_view image:=/camera/depth/disparity

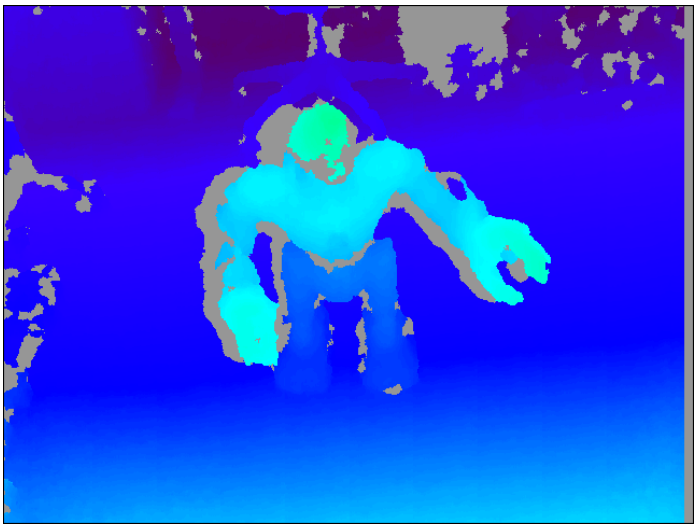


Figure 3.9: image_view Disparity

Topics published by openni_camera are:

- camera/depth/image_raw (sensor_msgs/Image)
  uint16 depth image in mm, the native OpenNI format.
- camera/rgb/image_raw (sensor_msgs/Image)
  Raw image stream from the camera driver, the native OpenNI format.
- camera/rgb/camera_info (sensor_msgs/CameraInfo)
  Camera calibration and metadata.
- camera/depth/camera_info (sensor_msgs/CameraInfo)
  Camera calibration and metadata.

These topics are then used by support packages like image_proc and depth_image_proc which publish new topics with registered images, rectified images, converting data type of encoded data streams, creating a point cloud stream for visualization in rviz, etc.

3.2.3 husky_base package

This ROS package contains Clearpath's driver for husky. The husky_base package contains a node for communicating with the Husky MCU, and launch files to bring up the Husky control and diagnostic systems. The husky_node hardware interface is configured through the husky_control package.
It is subscribed to a single topic:

- husky_velocity_controller/cmd_vel (geometry_msgs/Twist)
  Twist velocity command for Husky MCU, muxed in husky_control. External clients should publish to bare cmd_vel topic.

While it publishes a couple of topics:

- husky_velocity_controller/odom (nav_msgs/Odometry)
  Odometry information from Husky MCU.
- status (husky_msgs/HuskyStatus)
  Status information from Husky MCU and husky_node. Also exposed over standard diagnostics interface.

rviz instances of husky are also available via husky_viz, which can be used to simulate Husky in rviz prior to actual hardware implementation.

3.2.4 image_acquisition package

We created this ROS package to establish flow of data between V-REP and other ROS packages. It contains two nodes:

- messenger_node: Communication bridge between V-REP and ROS. Converts motion planning data obtained from V-REP into a custom message format for other nodes to work with.
- reader_node: Manages activation and suspension of multiple nodes in oppeni_camera, jaco_ros, and husky_base. It basically controls the activation of those nodes by waiting for specific events to occur like arm reaching desired configuration or Kinect finishing up saving images.

This package does not create any new topics or services but subscribes and publishes to many topics from the prior mentioned packages. Details about these nodes and this package in general are provided in chapter 4.

Chapter 4

Motion Planning and Image Acquisition

In order to generate a desirable image set, precise positioning of the camera is of utmost importance. Not only positioning but its traversal through the 3D workspace has to be compliant, collision free, and optimal (in terms of uniformity of motion). The hardware platform described in chapter 3 was initially simulated in a physics engine based simulator called Virtual Robot Experimentation Platform (V-REP), before actual implementation.

We use this simulation platform extensively for testing various circumstances and scenarios. It is also our primary means for motion planning and path planning, the results of which are used on the real world platform. This chapter describes various facets of the motion planning stage starting with scene creation, platform modelling, kinematics, and constrained motion planning.

4.1 V-REP

As shown in section 3.1, V-REP is capable of replicating/simulating the entire setup for our platform. We sample the angular positions for each joint in Jaco Arm, as well as the motors in Husky robot wheels, based on simulation time. This provides us a profile of joint space configurations for the Jaco arm to transition through forward kinematically and achieve the required trajectory for capturing the data set.

But before using the raw position values for Jaco joints, they have to be processed in order to account for the differences between physical joints and simulated joints. The physical joint motors are limted to a range of -180 to +180, while there is no such limitation on the simulated joints, hence a simple conversion needs to take place before the joint positions can be used. For the Right Jaco arm, following conversions were performed:

- Joint 1: P_ang= - (S_ang+90)
- Joint 2: P_ang= S_ang+90
- Joint 3: P_ang= - (S_ang-90)
- Joint 4: P_ang= - (S_ang-180)
- Joint 5: P_ang= - (S_ang-180)
- Joint 6: P_ang= - (S_ang+270)

The child script attached to the Jaco Arm performs the aforementioned conversion before publishing the joint configurations. The conversion takes place during the simulation time. There are two choices in terms of publishing the joint space configuration:

1. Real time, each sample configuration (which contains 6 joint angles for Jaco arm and 2 motor position for Husky) is published on a rostopic. V-REP creates a few publisher and subscriber nodes on ROS which in turn can be used by other ROS nodes. This method is slower among the two as V-REP simulation has to pause

until the actual hardware platform has finished the actions for the published frame. Also requires more computing power as ROS and V-REP are actively consuming computing resources. Running the simulation in parallel to actual platform, as fancy as it might sound, is a little hazardous. If something goes wrong in any step during the simulation, the same erroneous behavior will carry on to the actual platform. To avoid this one has to constantly observe the simulation until the end and pause or stop it if required.

2. Batch transfer. In this method, we let the simulation to get complete while recording all the frames/configurations based on our sampling time in a text or csv file. This csv/text file can later on be read by a ROS node to drive the actual platform at any time. This method is quite reliable and we first go through the simulation, making sure everything worked out fine, and then use the file at any time to perform the actual experiment. The only downside is lack of real time factor. We designed our pipeline such that the actual platform movement can be queued to begin right after the simulation finishes up by simply activating a node on one of the terminal windows. This allows for the system to reallocate resources like RAM, processing power, etc. between V-REP and ROS.

4.1.1 Kinematic Simulations

V-REP uses IK groups and IK elements to solve inverse and forward kinematics tasks. It is important to understand how an IK task is solved. An IK group contains one or more IK elements:

- IK groups: IK groups group one or more IK elements. To solve the kinematics of a simple kinematic chain, one IK group containing one IK element is needed. The IK group defines the overall solving properties (like what solving algorithm to use, etc.) for one or more IK elements.
- IK elements: IK elements specify simple kinematic chains. One IK element represents one kinematic chain. A kinematic chain is a linkage containing at least one joint object. In short, an IK element is made up by:
    - **A base** (any type of object, even a joint. In that case however the joint is considered as rigid). It represents the start of the kinematic chain.
    - **Several links** (any type of object except joints). Joints which are not in IK mode are however also considered as links (in that case they behave as rigid joints (fixed value)).
    - **Several joints**. A joint which is not in IK mode is however not considered as a joint, but as a link. Refer also to the joint properties.
    - **A tip**. The tip is always a dummy and is the last object in the considered kinematic chain (when going from the base to the tip). The tip dummy should be linked to a target dummy and the link should be an IK, tip-target link type. Refer also to the dummy properties.
    - **A target.** The target is always a dummy and represents the position and/or orientation the tip should adopt (or follow) during simulation. The target dummy should be linked to a tip dummy and the link should be an IK, tip-target link type.

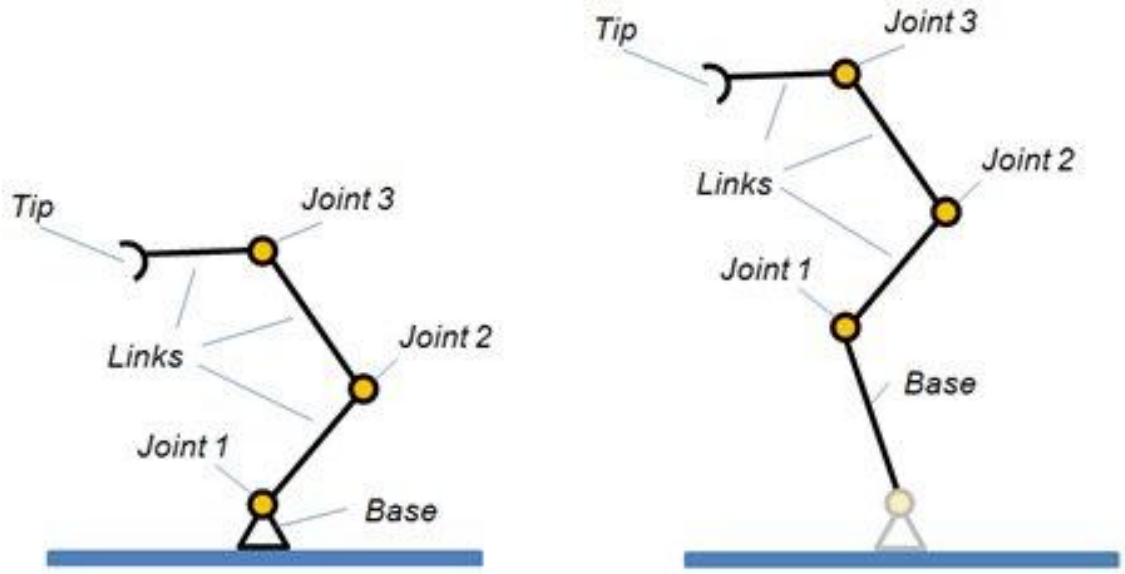


Figure 4.1: Two kinematic chains, each describing an IK element

IK Elements are specified by a kinematic chain (Figure 4.1), and a target to follow. The kinematic chain itself is specified by a tooltip (or end effector, or tip in short), indicating the last object in the chain, and a base, indicating the base object (or first object) in the chain. Above figure shows two kinematic chains as specified for an IK element. The IK element perceives the two chains in a similar way (the very first joint of the second example is ignored by the IK element).

In above example, the kinematic chains as specified by the tip/base pair, both have 3 Degrees of Freedom (DoF) because 3 1-DoF joints are involved. Should one of the joints however be a spherical joint, then the chain would have 5 DoF since a spherical joint has 3 DoF by itself.

Now we have to tell the IK element how the specified kinematic chain should behave during simulation. Typically, we want the tip of the kinematic chain to follow a target (referred to as dummy target in rest of this paper). When the simulation is running and some additional parameters have been defined correctly, then the

mechanism (the specified kinematic chain) should move towards the target. This is the most basic case of IK task.

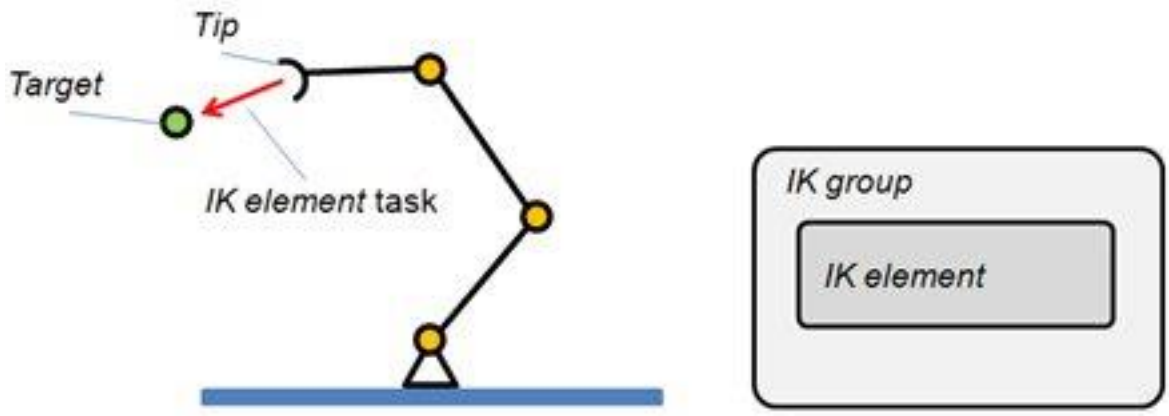


Figure 4.2: IK element and corresponding model of the IK solving task

Following are the steps to successfully set-up IK or FK calculations:

- Specify individual kinematic chains by providing a base and a tip.
- Specify a target to follow (simply link the tip dummy to a target dummy).
- Group IK elements in a single IK Group if they share common joints.
- Order individual IK Groups in order to obtain the wanted behavior.
- Verify that joints in a kinematic chain have the correct properties enabled or disabled.
- Verify that individual IK Elements are not overconstrained (X, Y, Z, Alpha-Beta, Gamma).

The last point is quite important to understand: a kinematic chain's tip that has all constraints turned on will follow its associated target in the x-, y-, z-directions, while trying to keep the same orientation as the target. This however only works well when the kinematic chain has at least 6 non-redundant Degrees of Freedom (DoF). The tip should always be appropriately constrained (i.e. never indicate more constraints than

there are DoFs in the mechanism). Positional constraints are most of the time specified relative to the base's orientation as can be seen from following figure:

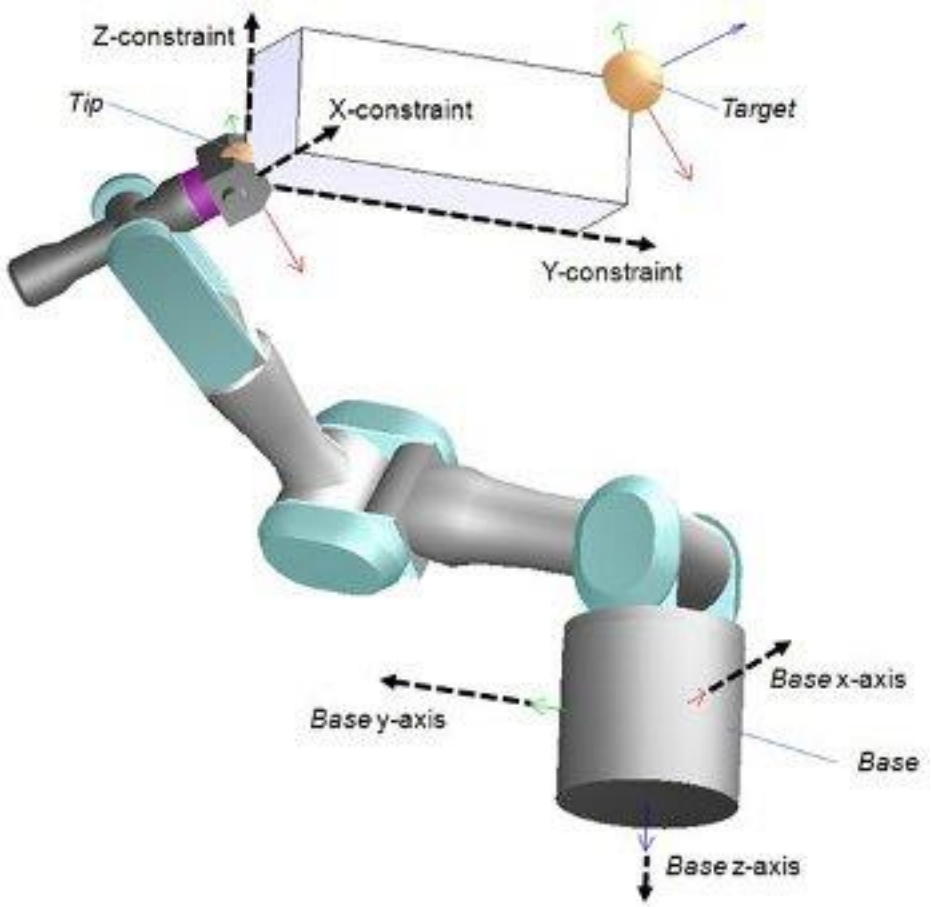


Figure 4.3: IK chain and constraints

Sometimes it is however not possible to correctly specify the constraints for a tip and in that case the IK group's calculation method should be a damped method (e.g. DLS method) with an appropriately selected damping factor. A damped resolution method should also be selected when a target can't possibly be reached (out of reach, or close to a singular configuration). Damping can result in more stable calculations, but keep in

mind that damping will always slow down the IK calculations (more iterations will be needed to put the tip into place).

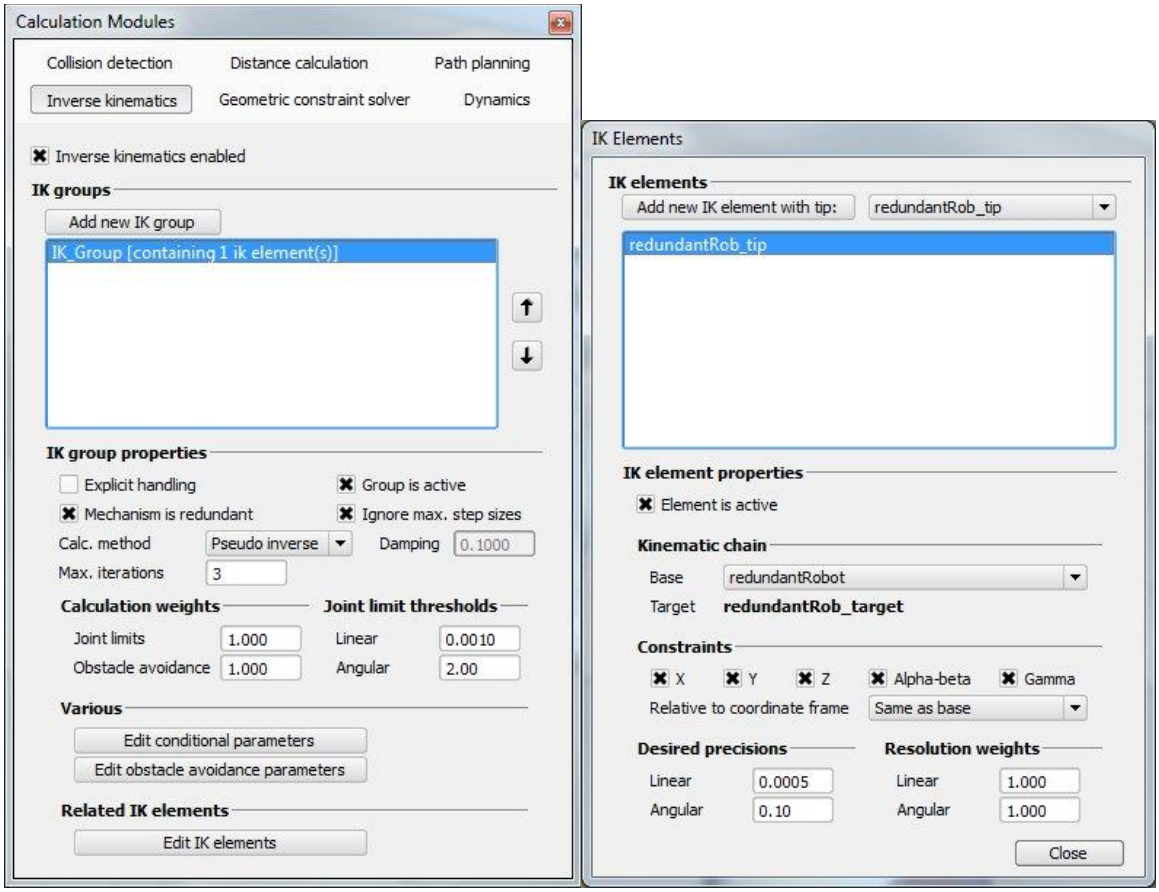


Figure 4.4: IK task menu and options

(Figure 4.3) Turning Alpha-Beta constraint on, will match the tip's z-axis orientation with the target's z-axis orientation, while keeping a free rotation around the z-axis if Gamma constraint is off. When Alpha-Beta constraint and Gamma constraint are turned on, then the tip will try to adopt exactly the same orientation as its associated target.

Solving the FK problem of simple kinematic chains is trivial (just apply the desired joint values to all joints in the chain to obtain the position and orientation of the tip or end effector). It is less trivial to solve the IK and FK problem for closed mechanisms.

For our platform, we start with determining the collision free workspace. We attach both left and right jaco arms to a dummy base of about same height as actual platform and cycle through all possible joint space configurations excluding the ones leading to self-collision. The end effector position and orientation for these candidate configurations are stored and also displayed on the simulation screen. Doing this provides us with a much required constraint of collision free end effector work space.

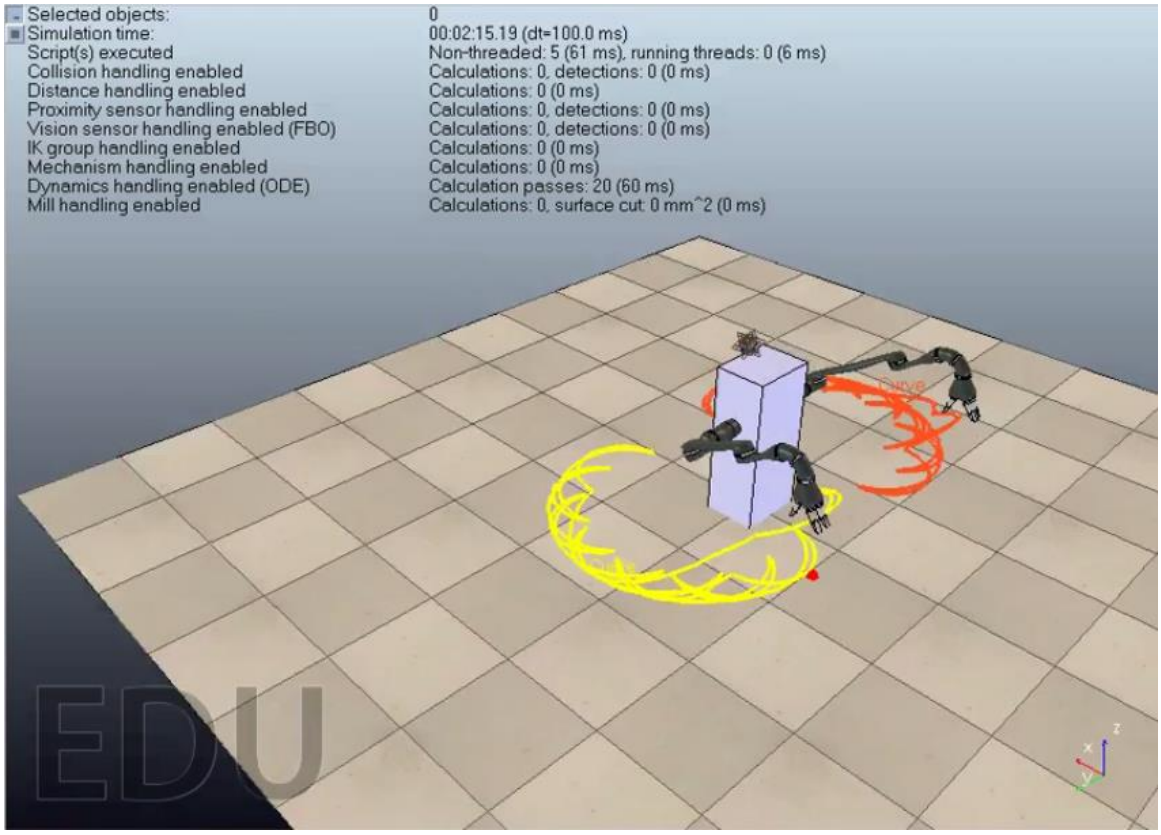


Figure 4.5: Jaco workspace

Once this is done, we now have a workspace for motion planning. Next step is to create an IK group for our robot arm and perform simple inverse Kinematics. We use stationary base of jaco arm as our IK chain base while we create a Dummy object in the gripper to act as tip of our IK chain.

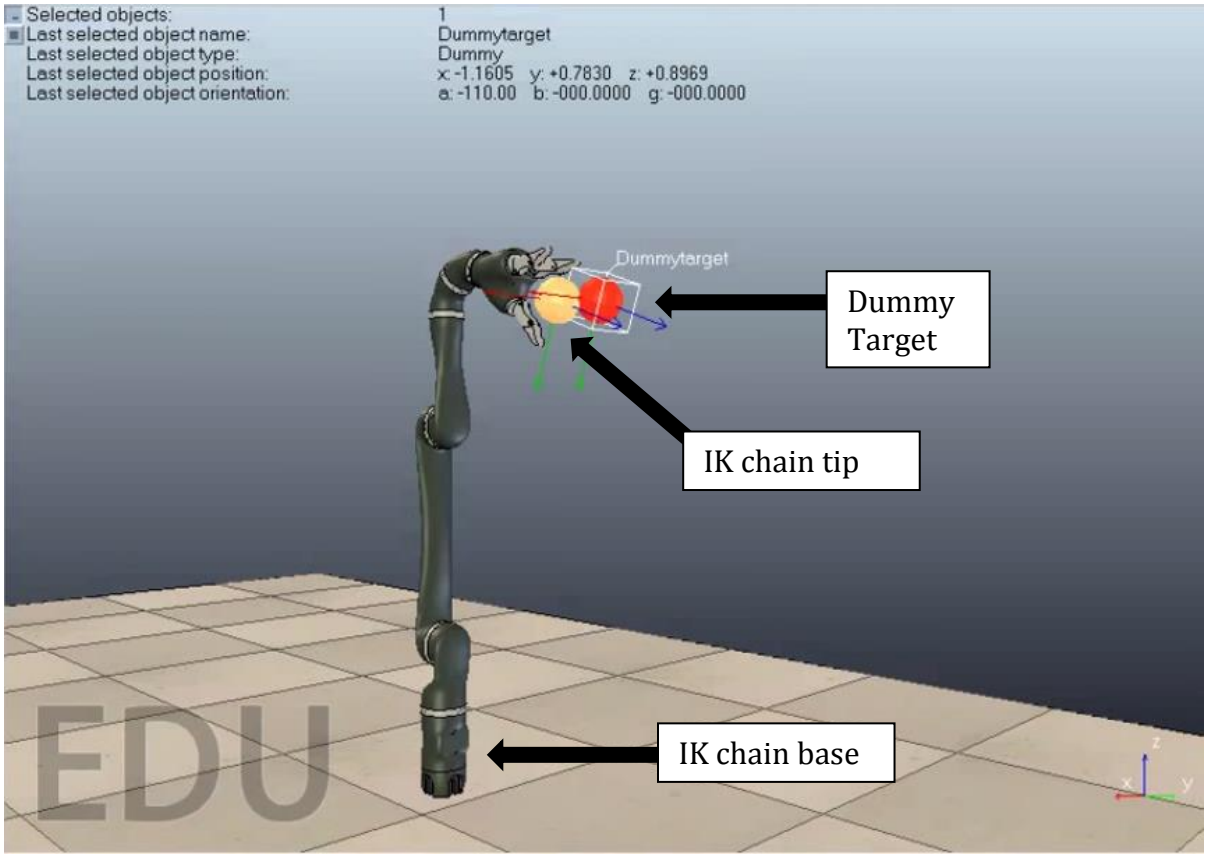


Figure 4.6: IK chain in Jaco arm

As seen in figure 4.6, we have a red dummy target along with our jaco arm IK chain. This target has a child script associated with it, connecting it to the tip of jaco IK chain. Hence one may either manually move the target (by clicking and dragging with mouse) or define a path for its travel in the child script, the end result will be Jaco arm will inverse kinematically follow the target, as long as it is in its workspace. Hence,

while defining its motion path we use the data from previous experiment about workspace determination, to avoid singularity.

4.1.2 Precise Object Tracking

For precise control of camera movement, we carried out an object tracking experiment in V-REP motion planning. As described in the last section, we had an IK chain setup for the right jaco arm and a target dummy object for the arm to follow. For the purpose of proper visibility and effectiveness of our tracking, we added a laser pointer on the tip of jaco arm which should point towards object of interest if the tracking algorithm works successfully.

Our object of interest in this specific case is a 7cm x 7cm x 7cm cube located at some point in 3D space. Notice how this object is outside the workspace of the jaco arm, hence trying to grasp it is not possible. Hence, this object of interest cannot be used as a target dummy for the jaco to follow. Instead we make the jaco arm follow the target dummy located inside its workspace. The script associated with the target dummy defines various trajectory for the dummy to move in 3d space inside jaco arm's workspace. Now for the arm to point towards the object of interest, we use simple 3D line geometry to determine the euler angles for jaco end effector, such that it points in the correct direction.

These angles are determined using following relationship between the end points of line segment between target dummy and object of interest:

$$\gamma = atan2\left(\frac{Object(y) - Target(y)}{Object(x) - Target(x)}\right)$$

$$\beta = atan2\left(\frac{Object(x) - Target(x)}{Object(z) - Target(z)}\right)$$

$$\alpha = atan2\left(\frac{Object(z) - Target(z)}{Object(y) - Target(y)}\right)$$

Where α, β, and γ are euler angles that represent a sequence of three elemental rotations, i.e. rotations about the axes of a coordinate system. First rotation about z by an angle α, second rotation about x by an angle β, and last rotation again about z, by an angle γ.

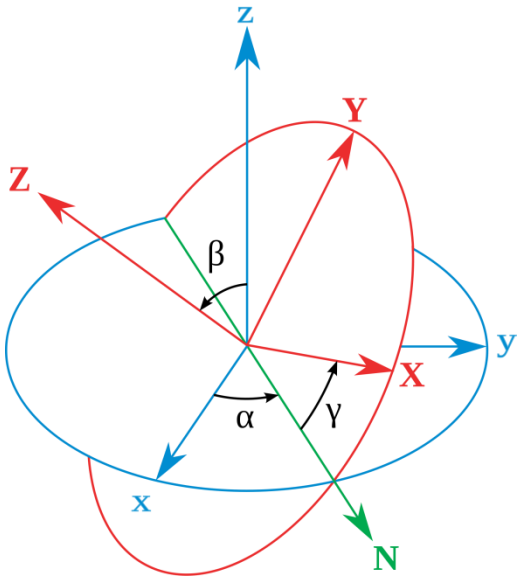


Figure 4.7: Euler angles

For ease of implementation we convert these euler angles into quaternions. The reason for this is, V-REP API contains functions that utilize quaternions more often.

$$q1= \sin(\beta/2)* \sin(\gamma/2)* \cos(\alpha/2)+ \cos(\beta/2)* \cos(\gamma/2)* \sin(\alpha/2)$$

$$q2= \sin(\beta/2)*\cos(\gamma/2)*\cos(\alpha/2)+ \cos(\beta/2)*\sin(\gamma/2)* \sin(\alpha/2)$$

$$q3= \cos(\beta/2)*\sin(\gamma/2)*\cos(\alpha/2)- \sin(\beta/2)*\cos(\gamma/2)*\sin(\alpha/2)$$

$$q4= \cos(\beta/2)*\cos(\gamma/2)*\cos(\alpha/2)- \sin(\beta/2)*\sin(\gamma/2)*\sin(\alpha/2)$$

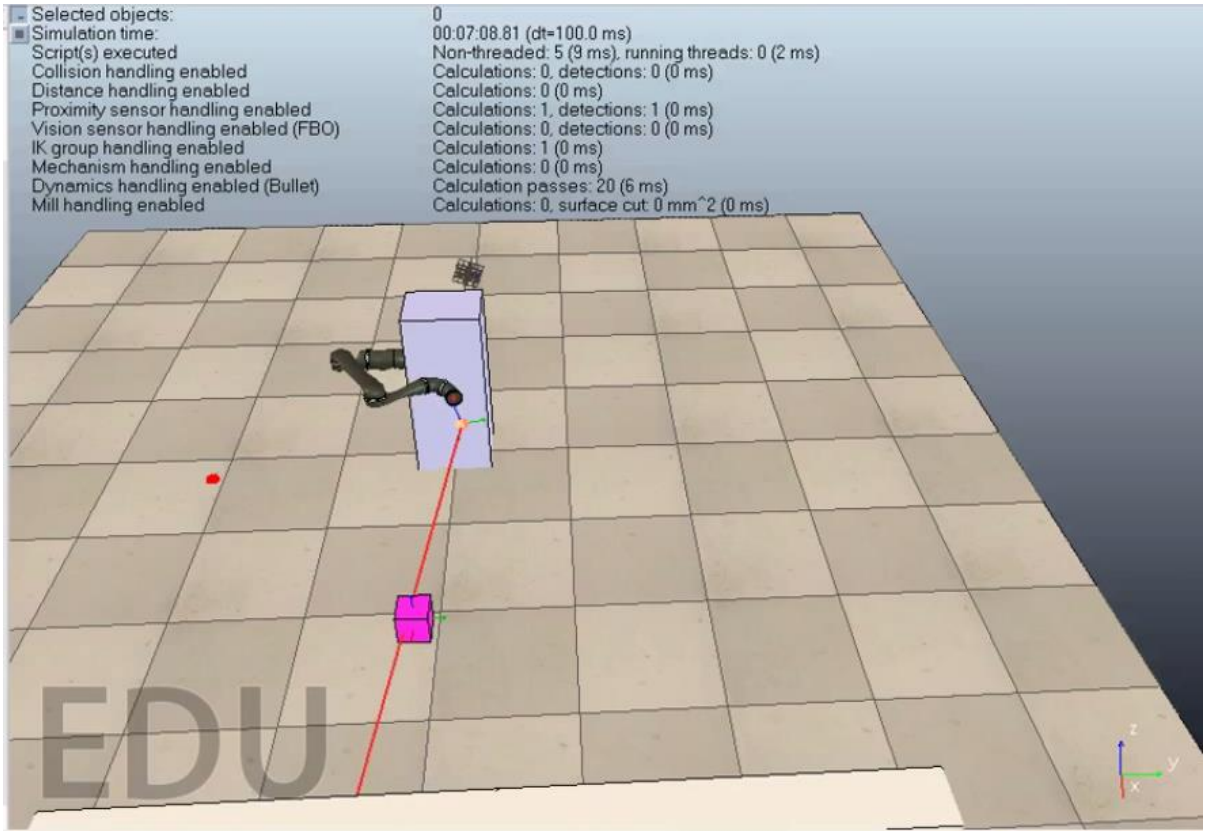


Figure 4.8: Object tracking using Jaco arm

Every iteration, for a given position of the target dummy (based on path planning script), its orientation is calculated as above and then using that position and orientation data, jaco arm inverse kinematically tracks the object of interest. This is true even if the object of interest moves in the 3D space, the only constraint is the speed of object. If it is moving faster than specific threshold, the jaco arm lags behind in terms of tracking. This is due to the fact that for the simulation it takes a finite amount of time to execute each iteration, this depends on number of scripts being executed, the type of functions used in those scripts (some functions take more simulation time compared to the others), and amount of rendering. Other than that it also depends on the amount of change in jaco joint configuration space. For instance, if the arm has to change 3 joint angles from position 1 to 2 it will take more time compared to say changing 1 joint angle from position 2 to 3. After taking all these factors into account, the system still works great for our application as we are not tracking an object in 3D space but simply following a predefined trajectory for moving the camera from one configuration to another.

4.1.3 Final Framework Simulation

Finally we take all these simulation results into account and recreate a scene for our actual hardware platform with all its components in place.

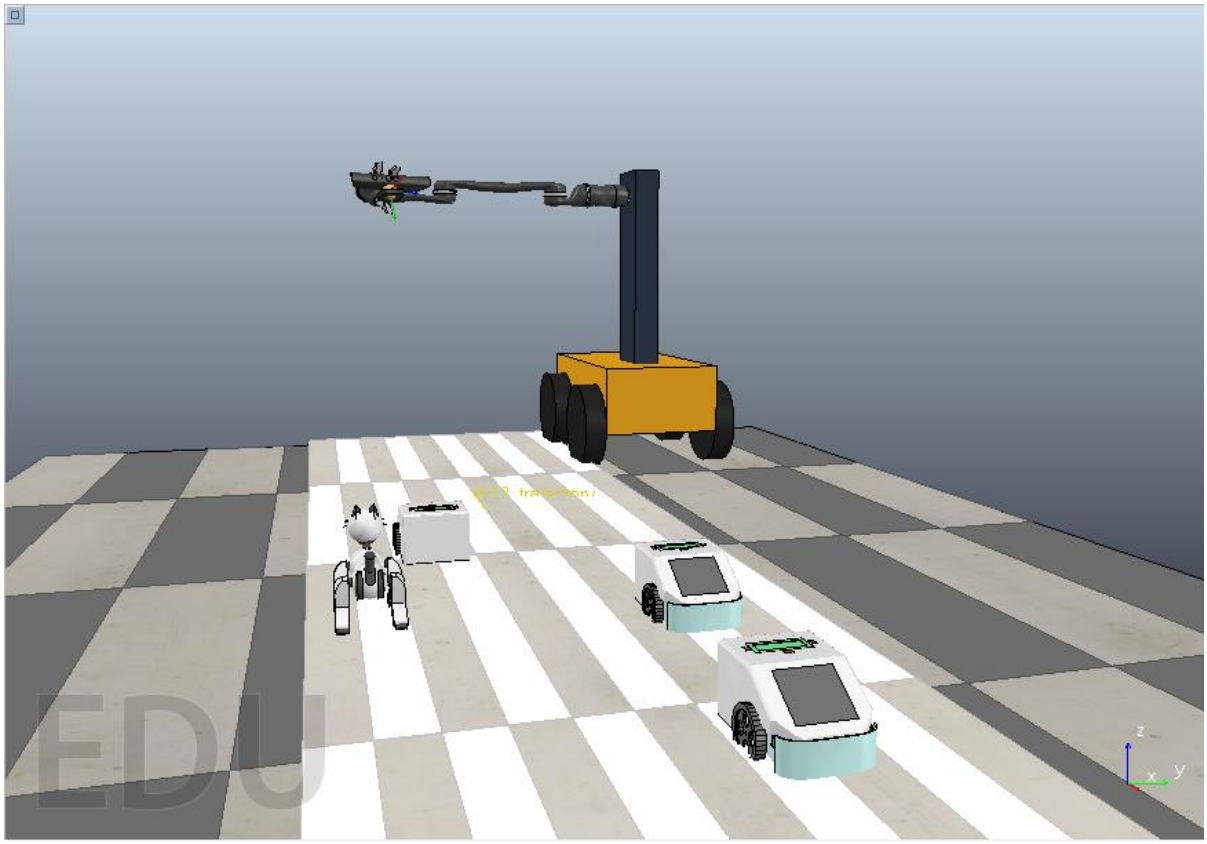


Figure 4.9: Final platform simulation

Our scene now consists of:

- Entire hardware platform model with exact same dimensions as the actual hardware.
- Miniature mobile robots disguised as cars.
- And a giant cat travelling along for some reason.

There are scripts associated with each vehicle that make them move at slightly different speeds. The sequence of events goes like this:

1. We start the environment as shown in figure 4.9 and the jaco arm moves to its intial position (Figure 4.10).
2. Then the target dummy moves across the track at five different equidistant positions, pausing at each for a small duration (to allow the time for capturing images).
3. Once a time frame has been recorded, the system moves forward, the cars move a little and so does the husky base.
4. Again the system as a whole pauses and jaco arm goes to step 2.
5. This continues for specific number of iterations or until the platform reaches end of the track.

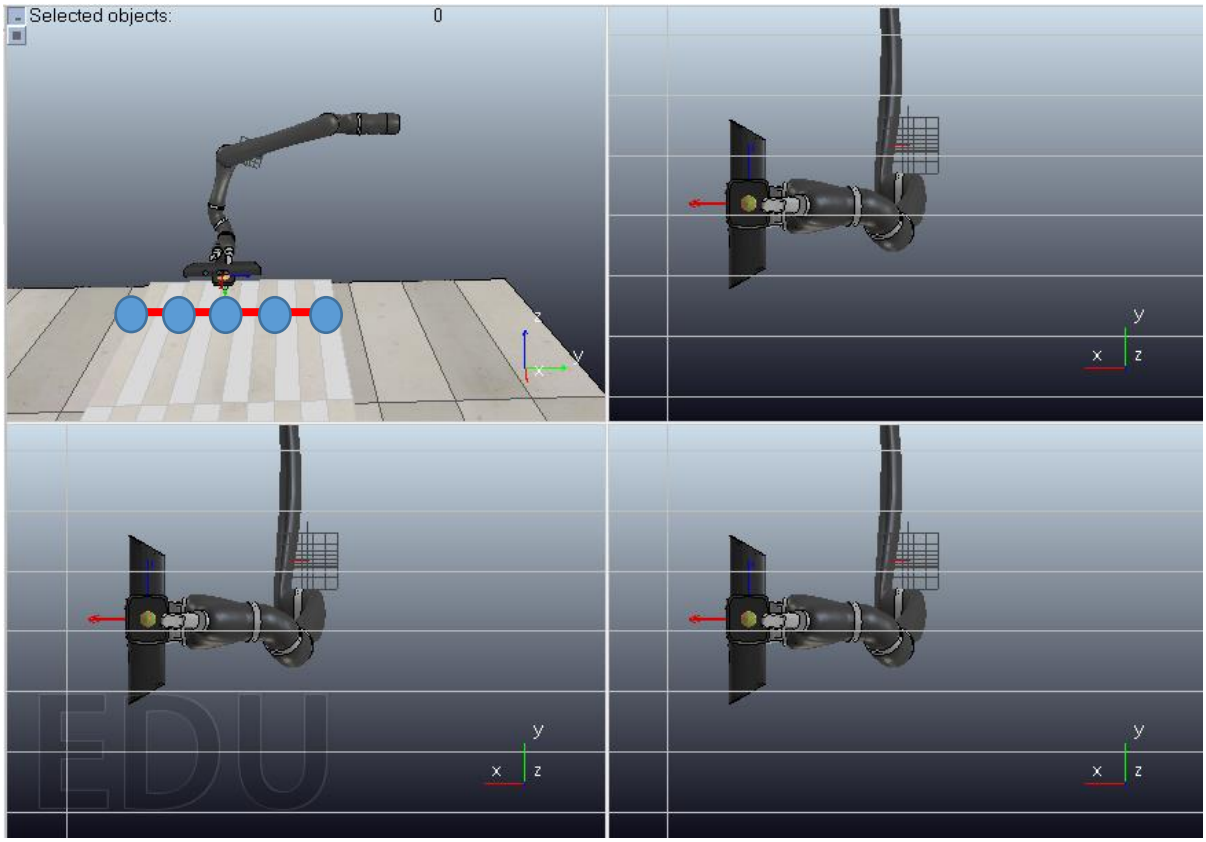


Figure 4.10: Jaco single time frame configurations

Figure 4.9 has the visibility layer for husky platform removed for showing clear trajectory. The arm moves through 5 positions across the track marked by blue dots, following the straight line trajectory defined by red line.

At each end effector configuration and for each time frame, the joint space configuration is recorded in a csv or text file, along with Husky wheel motor positions, for batch transfer to ROS which will further use this motion planning data to drive the actual hardware platform and acquire images.

4.2 ROS Framework

Once V-REP provides motion planning data for husky and jaco, ROS takes over for execution of actual platform. Figure 4.11 provides a flow chart representation of the process of image acquisition using ROS. The packages described in chapter 3 along with a custom package named image_acquisition created by us are used for robot control and data acquisition. The Kinect depth camera is registered to the RGB camera prior to usage by following the method described at [34]. As shown in the figure 4.11, once we have motion planning data in text/csv format following steps are taken:

- messenger_node in image_acquisition package reads the data and creates a new text file where each time frame has following format:
  * *Left motor position Right motor position*
  jaco_config_1 jaco_config_2 jaco_config_3 jaco_config_4 jaco_config_5
- reader_node in image_acquisition package reads this file and activates other nodes from packages described in chapter 3.
- husky_base node is activated with “Left motor position” and “Right motor position” driving the husky base, meanwhile reader_node constantly compares actual husky wheel position to check whether desired motor positions are reached.

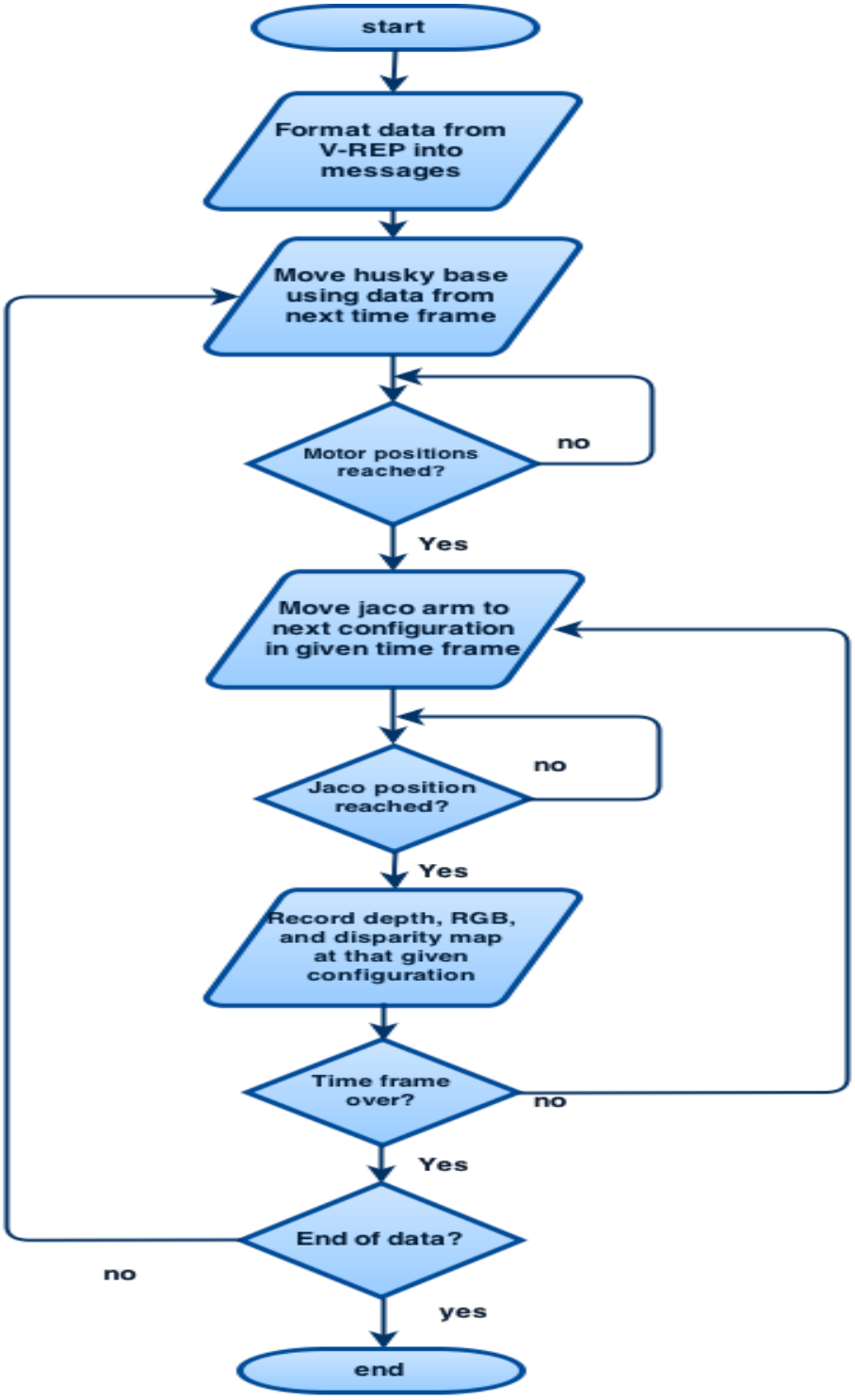


Figure 4.11: Software framework workflow

- Once desired motor positions are reached, reader_node activates jaco_demo/joint_angle_workout node in jaco_arm_driver package. This node takes in joint angle configuration and moves the jaco arm accordingly.
- Again reader_node checks whether a configuration has been reached by jaco_arm or not. Once it reaches that configuration, reader_node activates depth_node, disparity_node, and rgb_node under image_acquisition package. These nodes subscribe to respective topics published by openni_launch, read in image data and then convert them to png images using OpenCV (cv_bridge nodelet).
- Images from same time frame are stored in the same folder. Once reader_node determines that all joint configurations have been reached (time frame over), it checks whether data file has ended, if there is further data, the control goes back to moving husky base and the loop continues until the data file ends upon which we have our required data set.

Chapter 5

Ground Segmentation

Segmentation is the process of partitioning a digital image into multiple segments (sets of pixels). The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyze. [43, 8] Image segmentation is typically used to locate objects and boundaries (lines, curves, etc.) in images. More precisely, image segmentation is the process of assigning a label to every pixel in an image such that pixels with the same label share certain characteristics.

The idea of segmentation is crucial to autonomous driving. Ideally, we want to clearly separate out and identify cars, pedestrians, other vehicles, or obstacles in order to guarantee a reliable motion planning implementation. The first step towards this goal is to identify and isolate the ground/road from the image which in most driving situations is a major portion of the scene. An algorithm that can segment the ground in (ideally) real time is desirable to obtain location data between vehicles simultaneously executing autonomous drive.

Disparity refers to the difference in location of an object in corresponding two (left and right) images as seen by the left and right camera which is created due to parallax (cameras' horizontal separation). The disparity of a pixel is equal to the shift value that leads to minimum sum-of-squared-differences for that pixel. The term disparity was originally used to describe the 2D vector between the positions of corresponding features seen by the left and right eyes. It is inversely proportional to depth. Disparity (Figure 5.1) maps provide us with means to calculate 3D depths of points within an image, relying on cameras set up in a stereo pair for its generation. Before disparity generation can be performed, a camera system needs to be calibrated

to calculate its intrinsic and extrinsic parameters that define its internal geometry, and helps provide a relation between 3D world coordinates and 3D camera coordinates.

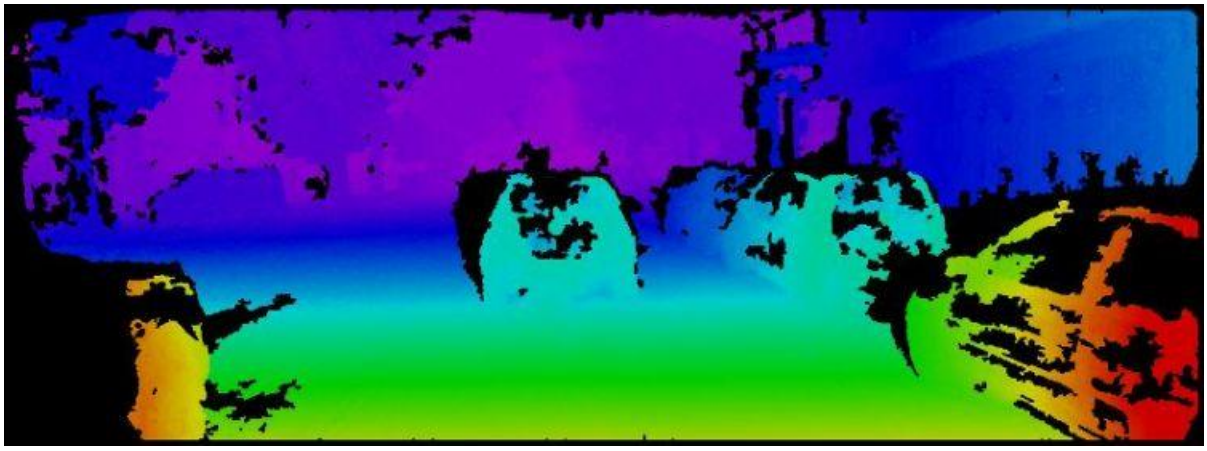
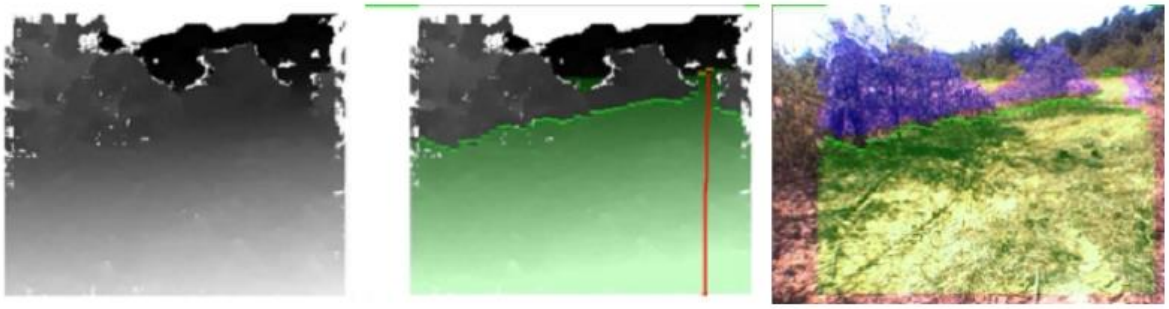


Figure 5.1: Pseudo colored disparity map [3]

We explore one such ground segmentation algorithm based on plane fitting in order to demonstrate the capabilities of our framework. We will use Random Sample Consensus (RANSAC) to estimate ground plane within any given scene. As discussed before, Image segmentation provides a method to split an image up into segments based upon their color or texture, and can be helpful in locating a specific plane. RANSAC is an iterative method used to estimate parameters of a mathematical model from a set of observed data that contains outliers. This technique is often used in mobility devices or for mapping outdoor area, and we will discuss how it can be used in conjunction with our disparity data to automate the process of finding a ground plane.

5.1 RANSAC for Plane fitting

Originally proposed by Fischler et. al [17] in 1981 as a method for best fitting a model to a set of points, such as finding a 2D line that best fits through a set of x and y data points, RANSAC has become a vital technique for the Computer Vision field, owing to its accuracy and performance [6]. This technique can be easily extended to take a data set of 3D points and calculate the plane model that best fits the selection. Research [42] on a mobility aid for the partially sighted uses RANSAC to great effect – enabling best fit of the visible ground plane from disparity values, even when there are obstacles in the way. In a similar manner, research by Konolige et. al [27] into mapping and navigation of outdoor environments relied on RANSAC for finding the ground plane in a challenging terrain filled with obstacles (see Figure 5.2).



(a) Disparity Image *(b) Extracted Ground Plane* *(c) Ground Plane Overlayed on Original Image*

Figure 5.2: Ground Plane Detection Using RANSAC [27]

This robustness to obstacles is useful to our proposed method, as roads and highways generally contain obstacles like cars, pedestrians, etc. At the same time a considerable amount of image area is ground, facilitating RANSAC. The limitations of this technique are in its trade-off between accuracy and performance, based off how much of the ground plane is visible within the image frame when the calculation was processed. If very little of the ground plane is visible during calculation, then RANSAC will be unable to find a suitable plane that fits best with the data available. To overcome this, we select a small region closer to vehicle with a lower disparity value as candidate ground pixels. If it remains unable to find a sufficient plane after a few retries, it could then take the best model it was able to get from those attempts [29].

This approach allows us to combine the plane detection and calculation into a single step, providing a ground plane that can be related to the autonomous driving data.

5.2 Ground segmentation using RANSAC

The basic idea, as described in previous section, is to estimate the model parameters using minimum possible data and then to check which of the remaining data points fit the model estimated. Further, we can apply region growing to merge the neighboring planes within certain local range.

Following are the implementation steps for our RANSAC ground plane fitting algorithm:

1. Select a small region on the disparity map, closer to the vehicle as candidate ground pixels (Figure 5.3). To reduce the time complexity of the fitting algorithm, we just use pixels with lower disparity values compared to a threshold. This is reasonable because we care only about the area close to the camera and most of this area is ground.

2. To further reduce the time complexity, we uniformly sample these candidate pixels by a specific interval. This interval is decided by training the algorithm on any given environment over a few images by increasing the sampling interval until we strike a balance between execution time and performance. One such example of the points used for fitting is shown in Figure 5.3.
3. Create a test plane from three randomly chosen points of the initial pool.

   $$Ax+By+Cz+D = 0$$

   Where, (A, B, C) is the nonzero normal vector of the plane through the point (x, y, z).
4. Test the remaining points in the pool against this fitted plane and count number of inliers based on minimum threshold distance. Keep a track of the plane estimation with maximum inliers.
5. Repeat steps 2 and 3 for a given number of iteration (we use 1000 iterations). If the plane with highest number of inliers crosses a threshold (70% of selected pixels) that plane is considered the ground plane.
6. If no plane has more inliers than the threshold, the process is continued until a plane with more inliers than the threshold is found.
7. Once a plane has been deemed as the ground plane, a least square fit is performed on the inliers for best approximation.

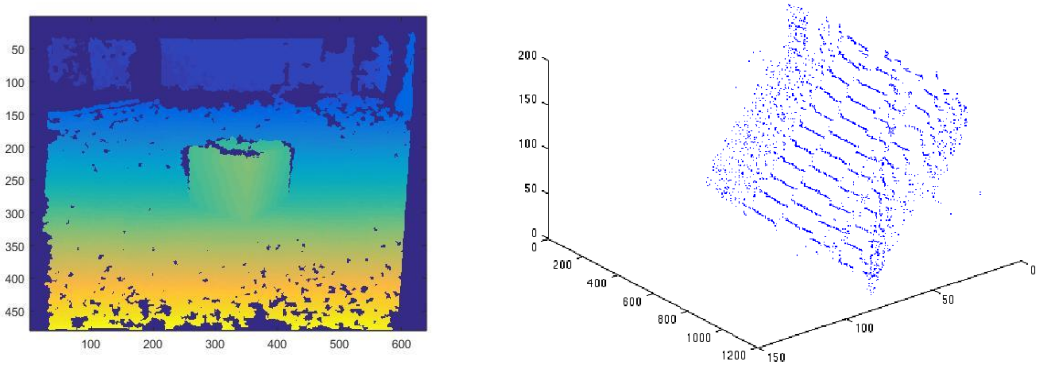


Figure 5.3: Left: Disparity map; Right: Ground pixels used for fitting

After computation of the ground plane, the obstacle pixels are computed as the pixels above the ground plane. The segmentation result of Figure 5.3 is shown in Figure 5.4 (Bottom), while Figure 5.4 (Top) shows the ground plane obtained by using RANSAC.

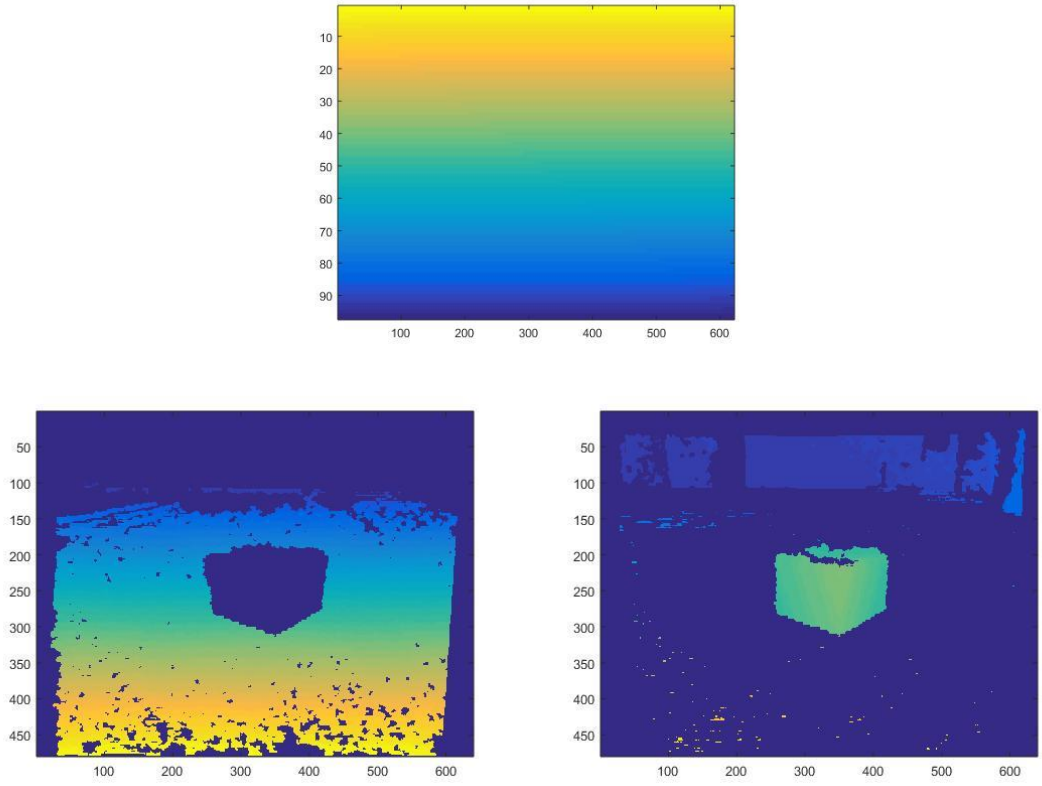


Figure 5.4: Top: Ground plane obtained by RANSAC; Bottom Left: Ground segmentation using plane fitting; Bottom Right: Obstacle segmentation/separation

Chapter 6

Kinect View Validation Framework (KVVF)

Kinect View Validation Framework (KVVF) is a MATLAB based framework with GUI for processing and analyzing image sets acquired using the technique described in chapter 4. In addition to this, the framework also integrates direct means of testing techniques and algorithms for autonomous driving or image processing in general. We will use the plane fitting based ground segmentation technique to show the capabilities of this framework. We used MATLAB to create a graphical user interface using GUIDE. Later we wrote callback functions to interact with the GUI, these callback functions would in turn call other mathematical and image processing functions to achieve desired tasks. Figure 6.1 shows blank GUI with classification.

6.1 High Level Operation

To better understand the user level working of the application, a brief description is provided followed by the appropriate screen-shots. Every time Kinect View begins with a blank environment (Figure 6.1). The GUI is divided into various sections:

1. Frame/path selection

   This section is a matrix of toggle buttons representing camera positions/frames in X-Y plane. The size of this matrix is determined by the dataset size. As discussed in a previous chapter, ROS framework stores resulting dataset in an organized hierarchy of directories. For every time frame (Husky robot standby position) there is a folder named after the sequential position of that time frame.

Inside each such folder lies a folder for each jaco configuration which in turn contains RGB, depth, and disparity images. For instance, if our platform moved to 10 different locations and at each location the arm moves to 5 different positions then we will have 10 parent directories containing 5 child directories each. And hence Kinect View will have a 10x5 Frame/path selection section (Figure 6.2).

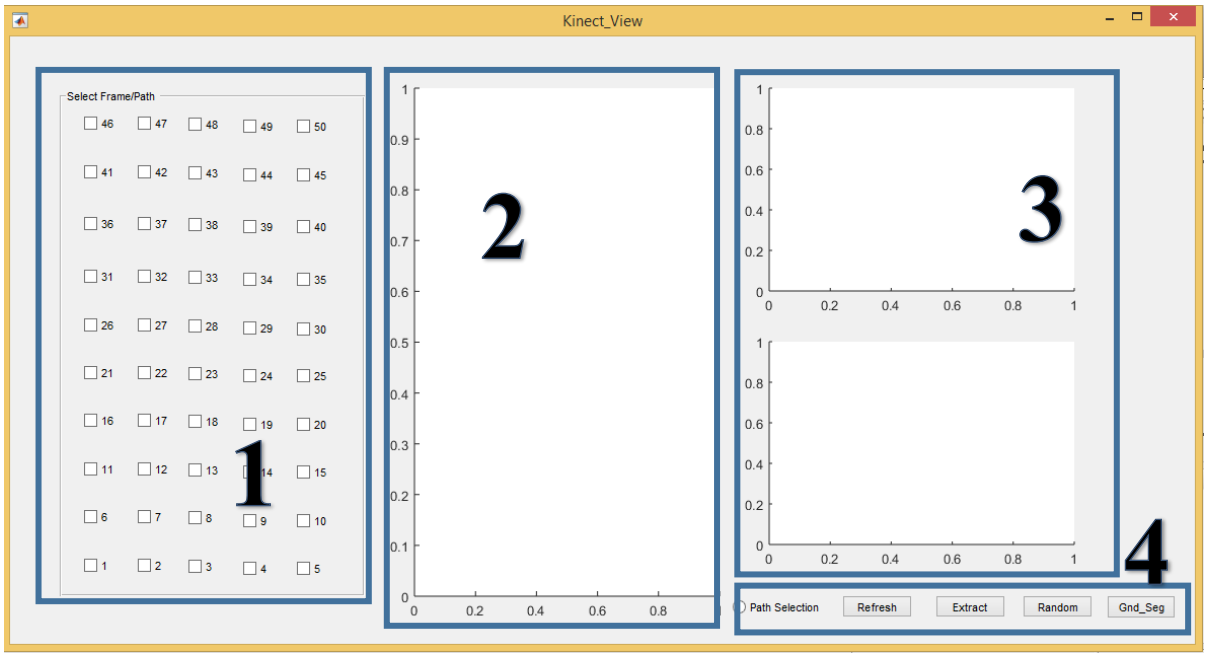


Figure 6.1: Kinect View front end; 1: Frame/path selection; 2: Path display; 3: image view window; 4: options

2. Path display

This section displays the user interaction with previous section. The matrix size is equal to that of the previous section. If the application is being used in Single View mode, active frame will be displayed with a large red dot while other inactive frames will be displayed as black dots. Similarly in Path Selection mode,

selected path will be displayed with red dots connected via a line, while inactive frames will be displayed as black dots (Figure 6.2).

3. Image view window

   This is a matlab implementation of image_view node from openni_launch package of ROS. It basically displays RGB and depth images for selected frame from section 1 in case of single view mode. For path selection mode, it shows depth and RGB images corresponding to last frame in selected path (Figure 6.3).

4. Options

   This is the section where user makes selections about the mode of execution and controls the analysis. Currently this section provides a radio button to toggle between Single View mode and Path Selection mode (Figure 6.3). Refresh button is a global reset for this application and necessarily brings it back to initial blank stage. Random button selects a random path from time frame 1 to last time frame, this is extremely useful for quickly testing any algorithm on a subset of data from a large dataset. Details of implementation are in next section. Extract button extracts and loads data pertaining to selected frame/selected path (based on mode of operation selected) for further analysis and testing of techniques. Finally Gnd_Seg button performs RANSAC based ground segmentation discussed in chapter 5 on the extracted data set and stores result in a specific folder called processed_data. These results can be loaded into the image viewer window for quick analysis.

The Gnd_Seg button is modular and replicable. It can be easily replaced by a drop down menu to allow multiple algorithms/techniques to be a part of this framework. As discussed before there are two basic modes of data analysis:

1. Single View mode
2. Path Selection mode

By default the application loads in Single View mode.

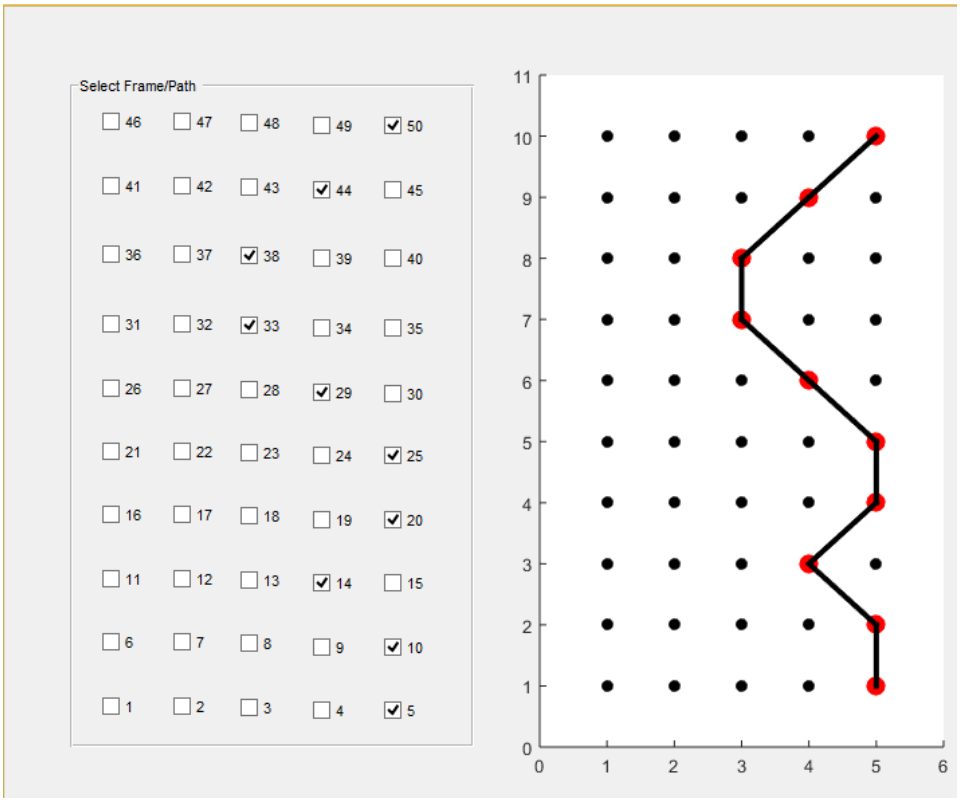


Figure 6.2: Kinect View front end; 1: Frame/path selection; 2: Path display; Path Selection mode ON

In this mode, we can select a single frame from any time frame and the image view window will display corresponding RGB and depth images. After selecting any frame of interest, we can hit extract to load the data and then push Gnd_Seg to perform ground segmentation on that specific frame. Results are displayed in the image view window, top window shows ground portion, while bottom window shows obstacles.

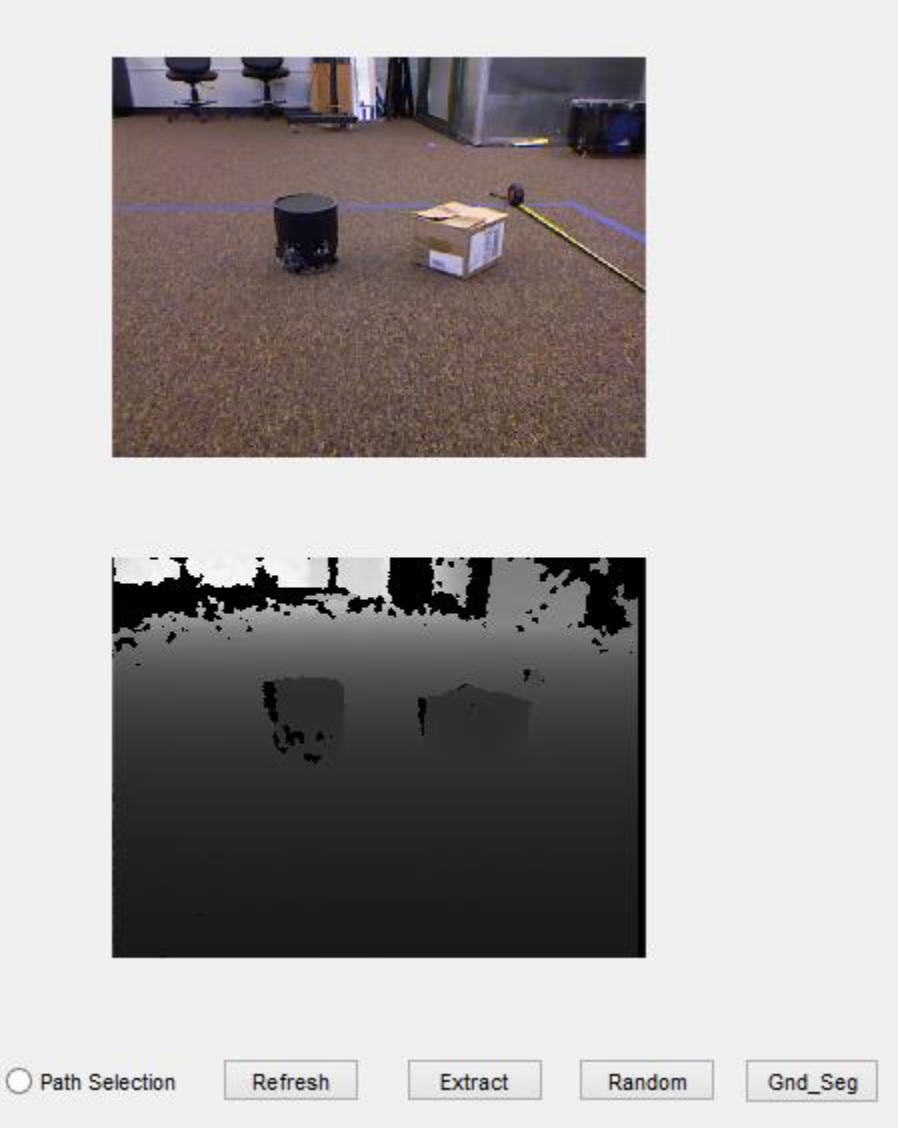


Figure 6.3: Kinect View front end; 3: image view window; 4: options with Single View mode On

In path selection mode, we can create a custom path by selecting a single frame from time frame one to last time frame or hit random button to generate a random path for us. Again, the path display window will display the selected path. Clicking on extract loads all the selected frames sequentially and hitting Gnd_Seg, performs ground

segmentation on the selected subset. The image view window displays segmented ground and obstacles for the last selected frame.

At any point during this operation, one may hit the refresh button to restart the entire pipeline from scratch. The results of ground segmentation are stored in a directory named processed_data in sequential order based on respective time frame.

6.2 Low Level Operation

After getting a look at the overall working of Kinect_View, this section goes down to the low level back end operations of the application. There are three basic components:

- MATLAB GUI
- Callback functions for GUI elements
- Objective MATLAB scripts

The GUI has been created using MATLAB GUIDE, which is a pick and place type GUI wizard. Using GUIDE facilitates quick front end creation with fake buttons and elements. GUIDE automatically creates a matlab file containing callback function declarations for every interactive element in the front end scene.

Using those declarations we can either create function definitions in the same file or separate files. The callback function definitions contain program snippets which define and regulate the behavior of the front end upon interacting with respective elements. The properties of each element like name, value, enable state, etc. can be read and modified by accessing its handle. Hence in order to modify an element from within callback function of another element, one has to pass the handle of the element of interest to that callback function.

As soon as a user interacts with any element, say toggles a switch in single view mode, a call is made to the callback function, which in this case modifies the path

display window by updating the red and black dots and also changes the images loaded in the image view window.

Mode selection radio button has been assigned a global signal in its callback function, thus affecting all the other callback functions to make them behave according to the mode selected. The Random button is implemented by a two dimensional constrained random walk. Consider the frames as vertices of a graph, first vertex is chosen at random from the first time frame. After this we random walk across the graph until we reach the final time frame. The constraints include: no backtracking and not to traverse more than one vertex in any given time frame.

Gnd_Seg button calls a function ground_segment, which is declared and defined in ground_segment.m, iteratively based on the number of frames in selected path (one in case of single view mode). Thus it is highly modular, we can have any number of algorithms or techniques stored as separate MATLAB scripts and based on which technique is selected, proper script is called. The framework in this aspect works like a plugin system where one can write any number of plugins and use interchangeably without having to make any major changes to the framework as a whole.

Chapter 7

Results

Summarizing all the previous chapters, this thesis provides a framework to record depth, RGB, and disparity data from multiple perspectives using a Microsoft Kinect and a mobile manipulator platform. This framework also provides means of analyzing computer vision techniques like object tracking, ground segmentation, etc. by generating data streams using random trajectories from the dataset previously recorded. This chapter presents the results of an experimental setup wherein we use our ground segmentation algorithm discussed in chapter 5 to illustrate the utility of this framework.

7.1 Experimental setup

Following the methodology from chapter 4, a scene was created in V-REP with the hardware platform as shown in Figure 4.9. A raster scan like trajectory was selected for the Kinect to follow (Figure 7.1). In order to record the data set corresponding to this pattern, following steps were taken:

- The scene started at the initial position, corresponding to the top-left black dot in Figure 7.1. The Jaco arm moved 10cm in the right hand direction until all five configurations were achieved.
- The Husky base motor positions and Jaco joint values for each configuration were recorded.
- Next, the Husky base model moved 20cm forward and the entire process was repeated until data for all 50 points was recorded.

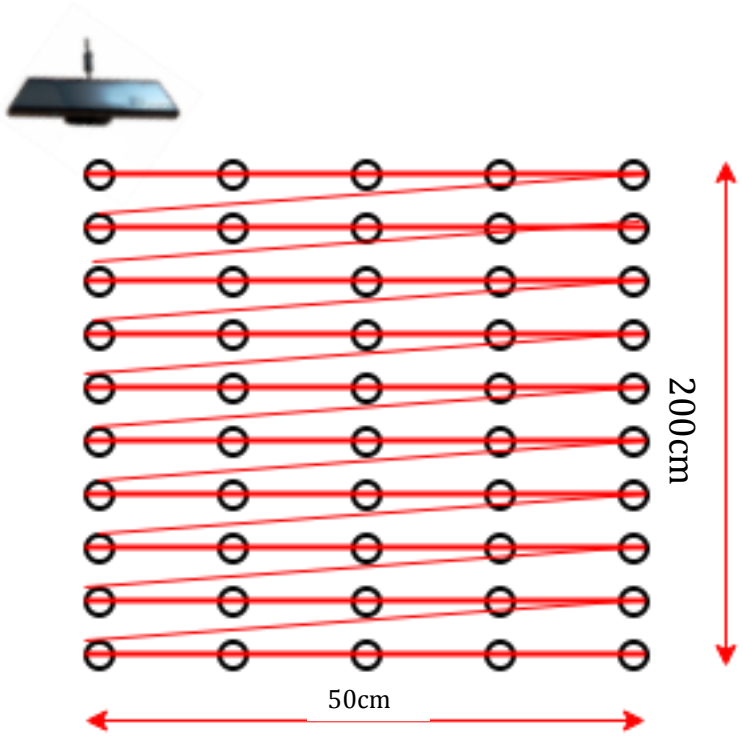
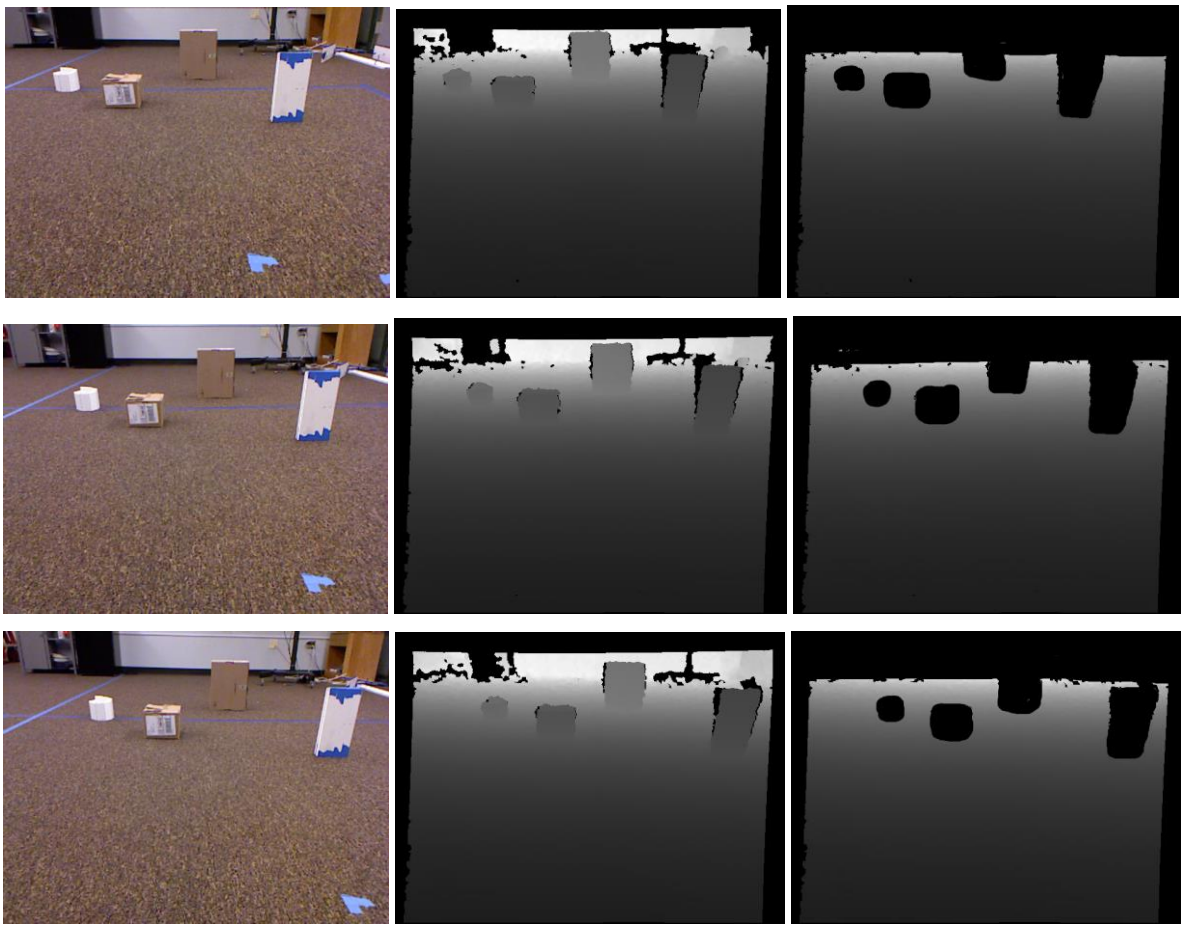


Figure 7.1: Kinect motion trajectory

- All the motion planning data, including Jaco joint angle sets and Husky motor positions were saved to a text file.
- messenger_node from image_acquisition package was activated to convert the text file into proper message format.
- The actual hardware was turned on and reader_node activated.
- Next, the reader_node activated Husky_base node and jaco_arm_driver based on messages passed by messenger node.
- At every point in the lattice, Jaco arm would pause for the reader_node to record the depth, RGB, and disparity data using the Kinect by subscribing to respective topics under openni_camera package.

- The recorded image and disparity data (Figure 7.2) was stored in a hierarchical directory.
- The ground truth shown in Figure 7.2 was obtained by manually editing out obstacles and other occlusions from the depth images.



Color Image Depth Image Ground Truth

Figure 7.2: Recorded data at different time frames

7.2 Evaluation

Now that we had the dataset, we used Kinect View to test our RANSAC based ground segmentation technique. As mentioned in section 5.2, our plane fitting algorithm requires selection of ground pixel candidates. We do this by selecting a small region closer to the vehicle. Figure 7.3 shows selection of ground pixel candidates.

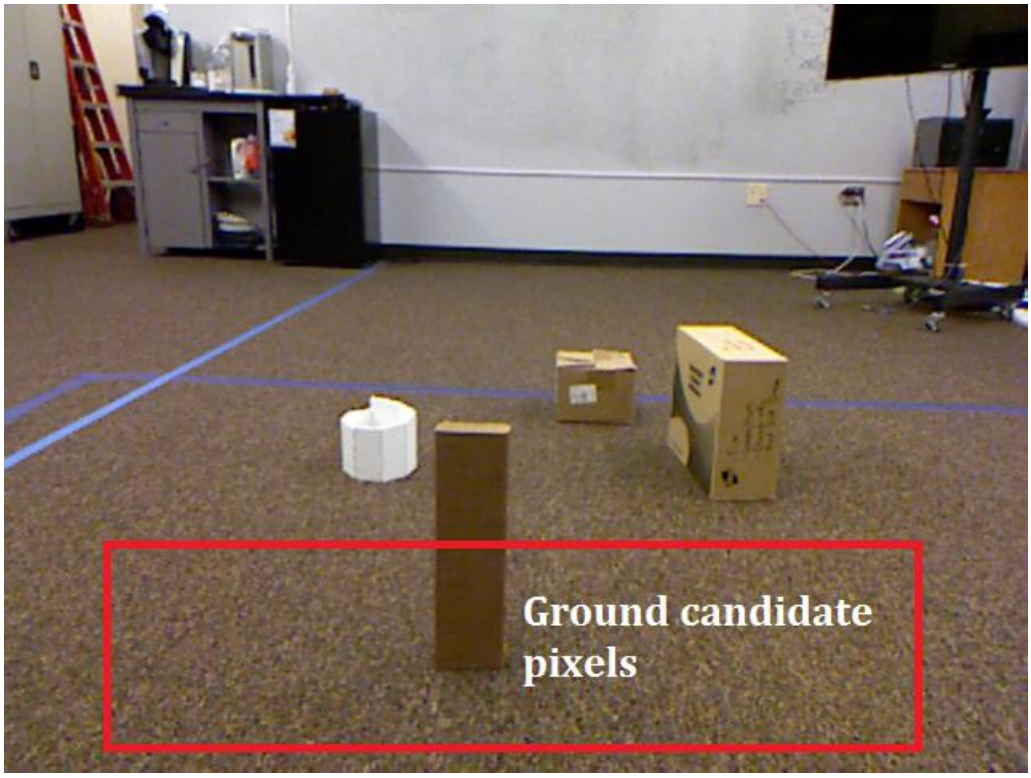


Figure 7.3: Selection of candidate pixels for ground plane

As seen in the above figure, this bounding box contains both obstacle pixels and ground pixels but for most of the cases, the number of ground pixels is much higher than the obstacle pixels and hence RANSAC should have no problem segmenting the two.

Random trajectories were generated using the Random function on Kinect View GUI and data streams were extracted using Extract function. These data streams were then used for ground segmentation. Results from one such trajectory are shown in Figure 7.4. For each frame, the left plot represents the depth data and the right plot is the output of our ground segmentation algorithm.

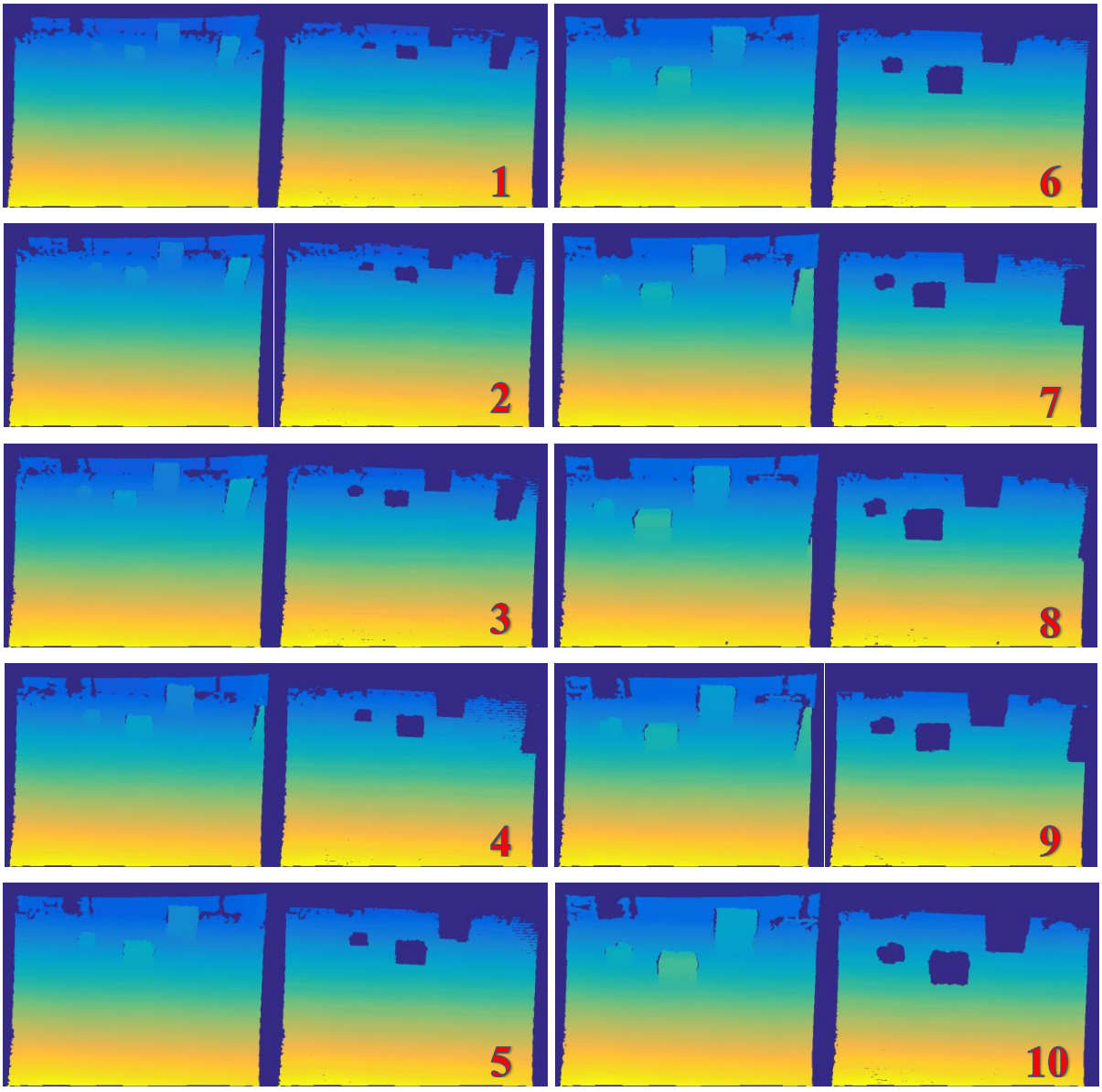


Figure 7.4 Random trajectory data stream (Trajectory from Figure 6.2)

We use Overlap ratio to gauge the performance of ground segmentation. Overlap ratio is defined as follows:

$$\text{Overlap Ratio} = \frac{Number\ of\ Ground\ pixels\ identified\ by\ the\ algorithm}{Number\ of\ Ground\ pixels\ from\ the\ ground\ truth}$$

We create five random trajectories and run our ground segmentation algorithm with a built in Kalman filter for ground plane tracking over subsequent images. Figure 7.5 shows a set of five such random trajectories while Figure 7.6 shows their mean and standard deviation.

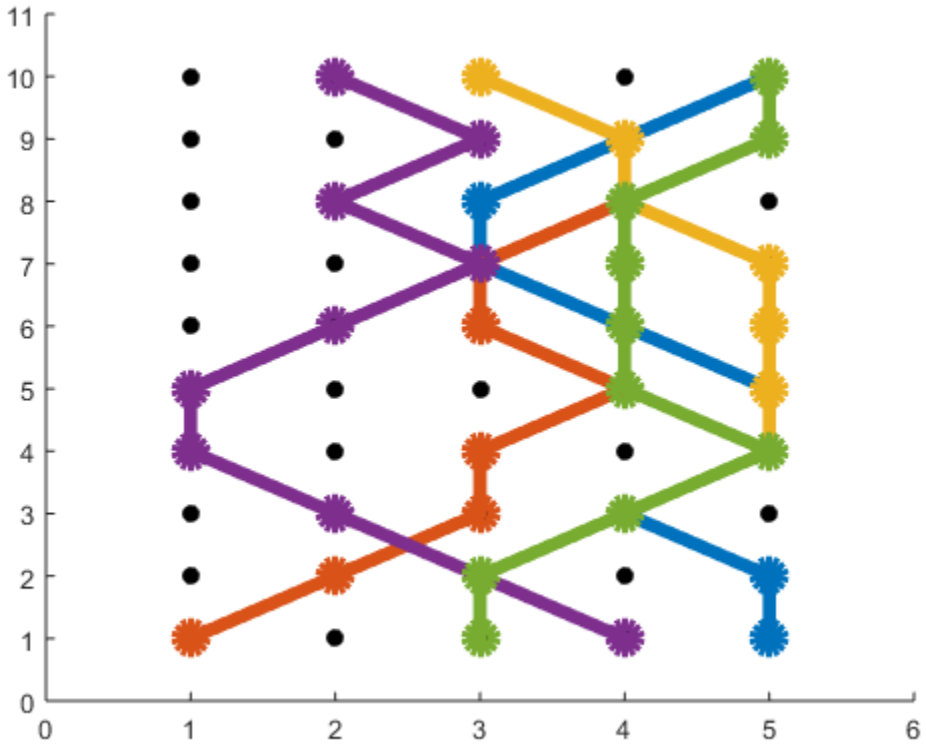


Figure 7.5: Five random trajectories

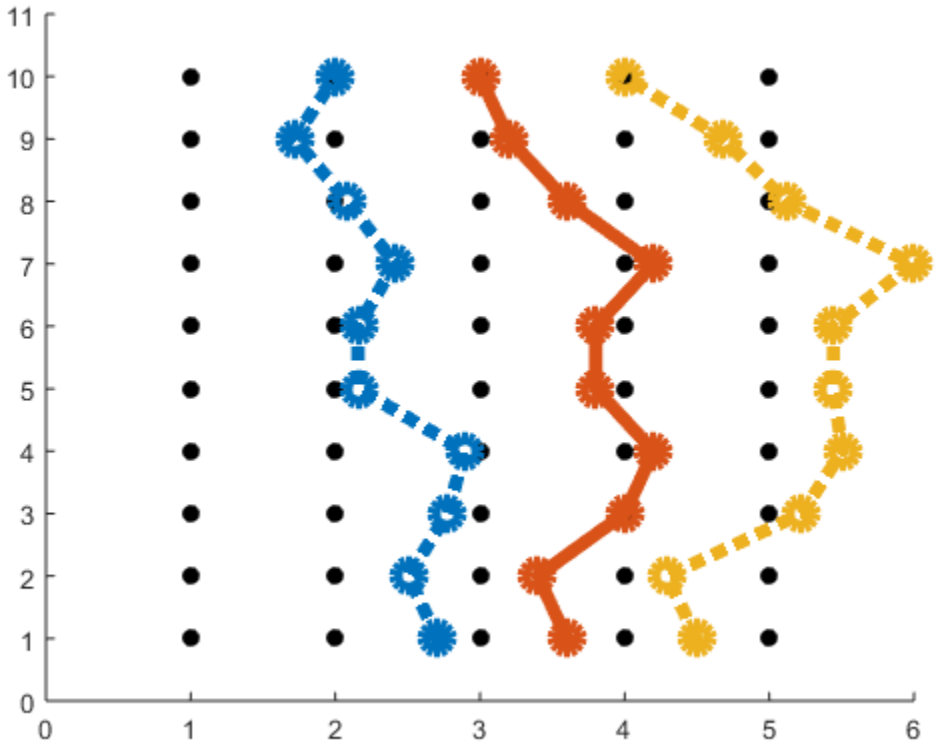


Figure 7.6: Red: Mean trajectory; Blue: (Mean-stdv) trajectory;
Yellow: (Mean+stdv) trajectory

The overlap ratio for each individual path can be seen in Figure 7.7 and the algorithm demonstrates a fairly accurate outcome with the worst overlap ratio being 1.044. Figure 7.8 indicates average and standard deviation of overlap ratios for five given paths. The recorded data contains various moving obstacles resembling different traffic conditions, yet the RANSAC based approach provided excellent and robust results.

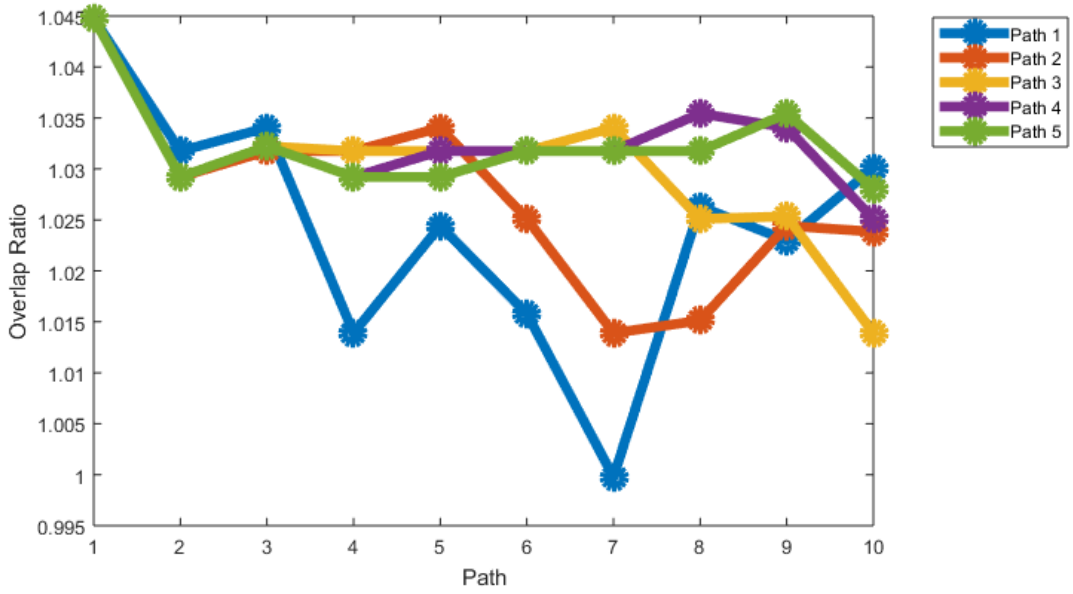


Figure 7.7: Overlap ratio for individual trajectories

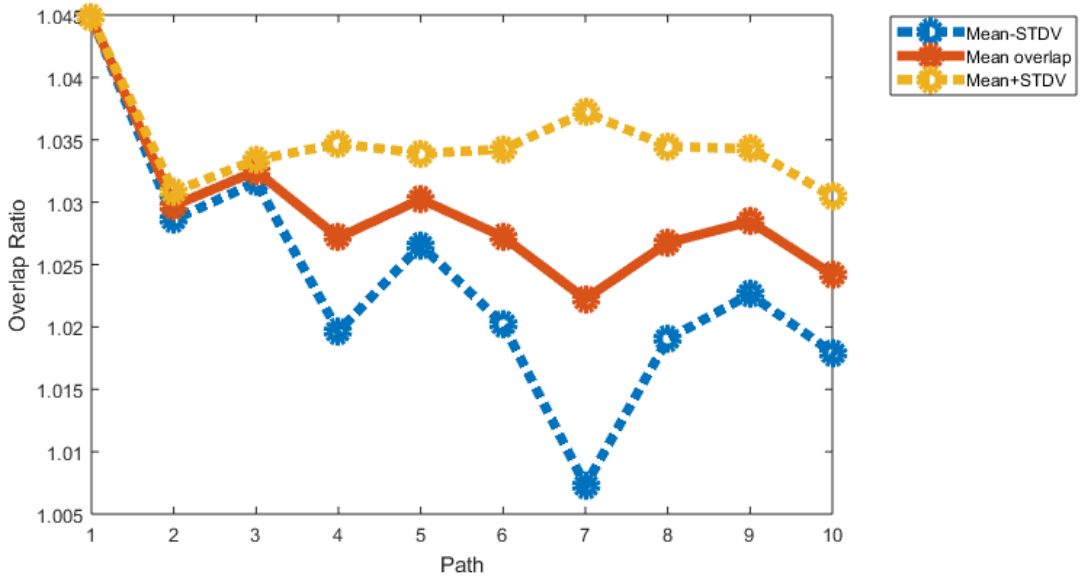


Figure 7.8: Mean and standard deviation of overlap ratio

Chapter 8

Conclusion and Future work

This thesis outlined a unique integration of hardware testbed and software framework for recording dynamic multi-perspective data stream of depth, RGB, and disparity data and then using the same to test the performance of visual algorithms like ground segmentation presented here.

We were able to present results to verify the working of this framework for ground segmentation, and showed how the framework is modular enough to add as many techniques as required for testing. The data acquisition platform can also be used independently to record novel data for use with other software platforms.

The framework being highly modular and capable of expansion, many extensions to the project are possible. The most obvious being, adding more vision based algorithms for testing. Going one step further, for any given position of camera, multiple images at multiple orientations can be obtained quite trivially by modifying the motion planning code; this will add a whole new dimension to the multi-perspective nature of this framework.

In addition to this, track and predict feature might be added to reduce the amount of computation for large dataset. In this, consider a dataset of 5000 camera positions, each having 3 orientations, leading to 15000 images. Such cases are extremely realistic when thinking about autonomous driving. For such cases, sampling the data for processing and filling in the gaps using machine learning and estimation is inevitable. We are currently pursuing this. Even though the Jaco arm is quite precise in positioning the camera, it still suffers from a drifting effect over a considerable period of time. Adding some mechanics for dynamic recalibration of the arm will ensure good reproducibility of poses.

REFERENCES

[1] (2015). *Lua, the programming language*. Available: http://www.lua.org/.

[2] (2015). *Bullet physics library*. Available: http://www.bulletphysics.org.

[3] (2012). *Environment Perception: Probabilistic Models for Urban Intersection Understanding*. Available: http://www.mrt.kit.edu/res2_2655.php.

[4] (2014). *ROS package: openni_camera*. Available: http://wiki.ros.org/openni_camera.

[5] (2012). *Do Consumers Want Autonomous Cars?*. Available: http://www.alpineautotrans.com/?p=326.

[6] G. Baguley. Stereo tracking of objects with respect to a ground plane. 2009.

[7] (2014). *Kinova Jaco Arm ROS package documentation*. Available: http://wiki.ros.org/jaco_ros.

[8] L. Barghout and L. Lee. *Perceptual Information Processing System* 2003.

[9] A. Broggi, M. Bertozzi, A. Fascioli, C. G. L. Bianco and A. Piazzi. The ARGO autonomous vehicle's vision and control systems. *Int. J. Intell. Control Syst. 3(4),* pp. 409-441. 1999.

[10] M. Buehler, K. Iagnemma and S. Singh. *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic* 200956.

[11] M. Buehler, K. Iagnemma and S. Singh. Editorial. *Journal of Field Robotics 25(9),* pp. 567-568. 2008.

[12] (2010). *Kinect calibration*. Available: http://nicolas.burrus.name/index.php/Research/KinectCalibration.

[13] J. W. Capille Jr. Kinematic and experimental evaluation of commercial wheelchair-mounted robotic arms. 2010.

[14] (2014). *ROS Concepts*. Available: http://wiki.ros.org/ROS/Concepts.

[15] E. D. Dickmanns, R. Behringer, D. Dickmanns, T. Hildebrandt, M. Maurer, F. Thomanek and J. Schiehlen. The seeing passenger car'VaMoRs-P'. Presented at Intelligent Vehicles' 94 Symposium, Proceedings of the. 1994, .

[16] M. Draelos. The kinect up close: Modifications for short-range depth imaging. 2012.

[17] M. A. Fischler and R. C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun ACM 24(6),* pp. 381-395. 1981.

[18] B. Freedman, A. Shpunt, M. Machline and Y. Arieli. *Depth Mapping using Projected Patterns* 2012.

[19] M. Freese, S. Singh, F. Ozaki and N. Matsuhira. "Virtual robot experimentation platform v-rep: A versatile 3d robot simulator," in *Simulation, Modeling, and Programming for Autonomous Robots*Anonymous 2010, .

[20] S. Gottschalk, M. C. Lin and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. Presented at Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques. 1996, .

[21] E. Guizzo. How google's self-driving car works. *IEEE Spectrum 18*2011.

[22] M. H. Hebert. *Intelligent Unmanned Ground Vehicles: Autonomous Navigation Research at Carnegie Mellon* 1997.

[23] K. Iagnemma and M. Buehler. Editorial for journal of field Robotics—special issue on the DARPA grand challenge. *Journal of Field Robotics 23(9),* pp. 655-656. 2006.

[24] S. Kim, Z. J. Chong, B. Qin, X. Shen, Z. Cheng, W. Liu and M. H. Ang. Cooperative perception for autonomous vehicle control on the road: Motivation and experimental results. Presented at Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on. 2013, .

[25] S. Kim, E. Lee, M. Choi, H. Jeong and S. Seo. Design optimization of vehicle control networks. *Vehicular Technology, IEEE Transactions on 60(7),* pp. 3002-3016. 2011.

[26] (2010). *kinect_calibration/technical. ROS.org. Discussion of the Kinect's software workings*. Available: http://www.ros.org/wiki/kinect_calibration/technical.

[27] K. Konolige, M. Agrawal, R. C. Bolles, C. Cowan, M. Fischler and B. Gerkey. Outdoor mapping and navigation using stereo vision. Presented at Experimental Robotics. 2008, .

[28] J. Kotkin and W. Cox. The World's fastest-growing megacities. *Forbes.Com* 2013.

[29] P. Kovesi, *Ransacfitplane-a Robust Matlab Function for Fitting Planes to 3d Data Points,* 2009.

[30] J. J. Kuffner and S. M. LaValle. RRT-connect: An efficient approach to single-query path planning. Presented at Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on. 2000, .

[31] J. Leonard, J. How, S. Teller, M. Berger, S. Campbell, G. Fiore, L. Fletcher, E. Frazzoli, A. Huang and S. Karaman. A perception-driven autonomous urban vehicle. *Journal of Field Robotics 25(10),* pp. 727-774. 2008.

[32] L. Merino, F. Caballero, J. R. Martínez-de Dios, J. Ferruz and A. Ollero. A cooperative perception system for multiple UAVs: Application to automatic detection of forest fires. *Journal of Field Robotics 23(3-4),* pp. 165-184. 2006.

[33] (2015). *Kinect for Windows Programming Guide*. Available: https://msdn.microsoft.com/en-us/library/hh855348.aspx?f=255&MSPPError=-2147217396.

[34] (2015). *Intrinsic calibration of the Kinect cameras*. Available: http://wiki.ros.org/openni_launch/Tutorials/IntrinsicCalibration.

[35] (2012). *University of Oxford Press, "Robot cars: more time, fewer prangs, less congestion"*. Available: http://www.ox.ac.uk/media/news_releases_for_journalists/111010.html.

[36] (2013). *Reverse-engineered USB protocol for the Kinect*. Available: http://openkinect.org/wiki/Protocol_Documentation.

[37] D. Pomerleau and T. Jochem. Rapidly adapting machine vision for automated vehicle steering. *IEEE Intelligent Systems 11(2),* pp. 19-27. 1996.

[38] (2010). *The PrimeSensor Reference Design 1.08*. Available: http://www.souvr.com/Soft/UploadSoft/201005/20100506172950 50.pdf.

[39] N. Qian. Binocular disparity and the perception of depth. *Neuron 18(3),* pp. 359-368. 1997.

[40] Rauch, F. Klanner, R. Rasshofer and K. Dietmayer. Car2x-based perception in a high-level fusion architecture for cooperative perception systems. Presented at Intelligent Vehicles Symposium (IV), 2012 IEEE. 2012, .

[41] (2012). *Rede von Dr. Norbert Reithofer, Vorsitzender des Vorstands der BMW AG*. Available: https://www.press.bmwgroup.com/pressclub/p/de/pressDetail.html?id=T0128046DE.

[42] S. Se and M. Brady. Ground plane estimation, error analysis and applications. *Robotics and Autonomous Systems 39(2),* pp. 59-71. 2002.

[43] L. Shapiro and G. C. Stockman. Computer vision. 2001. *Ed: Prentice Hall* 2001.

[44] (2010). *Microsoft Kinect teardown. iFixit.com* . Available: http://www.ifixit.com/Teardown/Microsoft-Kinect-Teardown/4066.

[45] D. M. Stavens. *Learning to Drive: Perception for Autonomous Cars* 2011.

[46] C. W. Wampler. Manipulator inverse kinematic solutions based on vector formulations and damped least-squares methods. *Systems, Man and Cybernetics, IEEE Transactions on 16(1),* pp. 93-101. 1986.

[47] T. Weigl, "Development Process for Autonomous Vehicles," 2014.

[48] World Health Organization. *Global Status Report on Road Safety: Time for Action* 2009.