Vision Powered

Pick and Place Robot

Supervised by

Dr. Khalid Wassif

Implemented by

|  |  |
| --- | --- |
| **20156001** | **Abanob Effat Farouk** |
| **20156015** | **Hisham Mohamed Salah** |

Graduation project

Academic Year 2018-2019

[**Chapter 1: Introduction**](#_heading=h.c2tzeemo71ce) **[4](#_heading=h.c2tzeemo71ce)**

[**1.1 Abstract**](#_heading=h.ue8ldlwz6tw2) **[4](#_heading=h.ue8ldlwz6tw2)**

[**1.2 Introduction**](#_heading=h.bvmehopt5brj) **[5](#_heading=h.bvmehopt5brj)**

[1.3 Motivation and Beneficiary](#_heading=h.2pjp0frf5oun) [5](#_heading=h.2pjp0frf5oun)

[1.4 Techniques](#_heading=h.c3hn5qv8b3ws) [6](#_heading=h.c3hn5qv8b3ws)

[1.5 Applications](#_heading=h.bxbpgccsiaem) [7](#_heading=h.bxbpgccsiaem)

[**1.5 Problem definition**](#_heading=h.npvze3v58xro) **[7](#_heading=h.npvze3v58xro)**

[**1.7 Work Plan**](#_heading=h.s0ds9rycaofk) **[8](#_heading=h.s0ds9rycaofk)**

[1.8 Report Organization](#_heading=h.mq414128l211) [9](#_heading=h.mq414128l211)

[**Chapter 2: Related Work**](#_heading=h.7txnkpur0ytk) **[10](#_heading=h.7txnkpur0ytk)**

[2.1 Vision based robot systems](#_heading=h.uljk5vx9247f) [10](#_heading=h.uljk5vx9247f)

[2.2 onVision Guided Pick and Place Robotic Arm](#_heading=h.tuid1yfvlkam) [14](#_heading=h.tuid1yfvlkam)

[**Chapter 3: System Specification**](#_heading=h.30zyttzcjp42) **[15](#_heading=h.30zyttzcjp42)**

[3.1 System Architecture](#_heading=h.4lw7yiuoge7p) [15](#_heading=h.4lw7yiuoge7p)

[3.2 System flow and techniques](#_heading=h.eun49o5mgxw1) [17](#_heading=h.eun49o5mgxw1)

[3.3 System Input/Output](#_heading=h.2h1z0ra7sw4f) [19](#_heading=h.2h1z0ra7sw4f)

[3.4 Obstacles](#_heading=h.5qhd36npnnrh) [19](#_heading=h.5qhd36npnnrh)

[**Chapter 4: Technologies and tools used**](#_heading=h.u54h1phxdv2) **[20](#_heading=h.u54h1phxdv2)**

[4.1 ROS](#_heading=h.ui8bbxki6siu) [20](#_heading=h.ui8bbxki6siu)

[4.1.1 A Brief History](#_heading=h.kb5i6t5k97b4) [21](#_heading=h.kb5i6t5k97b4)

[4.1.2 Philosophy behind ROS](#_heading=h.95xqbq9x61f5) [22](#_heading=h.95xqbq9x61f5)

[4.1.3 Peer to peer Architecture](#_heading=h.ytm4h2zeirz0) [22](#_heading=h.ytm4h2zeirz0)

[4.1.4 Tools-based](#_heading=h.yzyqn1dy3zsq) [22](#_heading=h.yzyqn1dy3zsq)

[4.1.5 Multilingual](#_heading=h.ntfqiuq3ivhm) [23](#_heading=h.ntfqiuq3ivhm)

[4.1.6 Free and open source](#_heading=h.8hie80ou85bn) [23](#_heading=h.8hie80ou85bn)

[4.1.7 ReUsability](#_heading=h.e2kwfl2jzcid) [24](#_heading=h.e2kwfl2jzcid)

[4.1.8 The ROS Graph](#_heading=h.8irquiqvxwct) [24](#_heading=h.8irquiqvxwct)

[4.1.9 roscore](#_heading=h.nx7ob8p1qk3l) [26](#_heading=h.nx7ob8p1qk3l)

[4.2 Gazebo](#_heading=h.ma39ro9n2a6w) [28](#_heading=h.ma39ro9n2a6w)

[4.3 Rviz](#_heading=h.ull0jis3tmrb) [30](#_heading=h.ull0jis3tmrb)

[**Chapter 5: Implementation**](#_heading=h.gkkpohciolia) **[31](#_heading=h.gkkpohciolia)**

[5.1 Computer Vision](#_heading=h.pmggilwyvui8) [31](#_heading=h.pmggilwyvui8)

[5.1.1 Statistical Outlier Filter](#_heading=h.9pkrwyjqf88) [32](#_heading=h.9pkrwyjqf88)

[5.1.2 Voxel Grid Downsampling](#_heading=h.9pr7reg6nvxr) [33](#_heading=h.9pr7reg6nvxr)

[5.1.3 Pass Through Filtering](#_heading=h.s3fwmg2g8pdh) [35](#_heading=h.s3fwmg2g8pdh)

[5.1.4 RANSAC](#_heading=h.2tnyn9ozvlu4) [37](#_heading=h.2tnyn9ozvlu4)

[5.1.5 Clustering with PCL](#_heading=h.gghpjqqqyclr) [39](#_heading=h.gghpjqqqyclr)

[5.2 Machine Learning](#_heading=h.amdrzevojory) [40](#_heading=h.amdrzevojory)

[5.2.1 Support Vector Machine](#_heading=h.vqlvvtj57a53) [40](#_heading=h.vqlvvtj57a53)

[5.2.2 Generating Features for the dataset](#_heading=h.ghhcfp7kwcaq) [43](#_heading=h.ghhcfp7kwcaq)

[5.3 Path Planning](#_heading=h.awavuyq37nhw) [47](#_heading=h.awavuyq37nhw)

[5.3.1 Sampling-based Motion Planning](#_heading=h.ru5mtmickhws) [47](#_heading=h.ru5mtmickhws)

[5.3.1.1 Probabilistic Roadmap](#_heading=h.rmc680ypjnt) [49](#_heading=h.rmc680ypjnt)

[5.3.1.2 Tree-based Planners](#_heading=h.yun7s6bcb6p6) [51](#_heading=h.yun7s6bcb6p6)

[5.3.2 Primitives of Sampling-based Planning](#_heading=h.z3suxssug91e) [54](#_heading=h.z3suxssug91e)

[5.4 Kinematics](#_heading=h.evupt06egl6k) [55](#_heading=h.evupt06egl6k)

[**Chapter 6: Debugging and Testing**](#_heading=h.coat5u5zhp8p) **[56](#_heading=h.coat5u5zhp8p)**

[6.1 Common problems](#_heading=h.thsqwj226a6w) [56](#_heading=h.thsqwj226a6w)

[6.2 Future work](#_heading=h.wz0swffa61yu) [59](#_heading=h.wz0swffa61yu)

[**References**](#_heading=h.p3nkon9et3wl) **60**

# 

# Chapter 1: Introduction

## 1.1 Abstract

The 21st century is a century for robotics. Robots have long borne the potential to bridge the gap between the cybernetic world (the internet of things) and the physical world. As the most promising candidate to theme the next major industrial revolution succeeding the present third (digital) industrial revolution, robotics is set to play an ever increasingly important role in society for its influence in every aspect of life, including medicine and healthcare, building services, manufacturing, food production, logistics and transportation.

Many companies started to research to provide a solution so that the robot can function in a dynamic environment, The main solution was that the robot should have a mind of his own and perceive the environment and decides what actions would be done without any kind of human assistant. Companies like amazon started to activate some warehouses that have robot arms, others made the robot arm to work in various jobs even making the arm cook by providing only the recipes.

Our robot is basically targeted for warehouses or other environments that have similar tasks that our robot will provide which are classifying different objects, ordering products and packaging different kinds of products together for shipment and operating other tasks with other robots to handle all the jobs in the warehouse without a need for humans or at least under human supervision only.

## 1.2 Introduction

Robots are the future of industries and any labor job even housework, Robots will clean, cook, organize things, maybe fetch something when it’s asked to and many more tasks.

Obviously robots aren’t intended to solve one problem or do one task and robots vary in their shape, components and the tasks that they are intended to do, However there are some tasks that most of the robots are doing not to solve the main task but to solve some sub-tasks to reach the final goal. In this project these sub-tasks are what we are intended to solve which are how to recognize different objects, build a collision map, path planning and finally to be able to pick and place objects.

## 1.3 Motivation and Beneficiary

As there are tasks that are done by human labor that is tedious and repetitive, but must be done to complete a certain product in the supply line of the organization. Due to humans doing those tasks repetitively they are bound to make mistakes which is human error or might look a bit different. And other tasks need high precision so it should be obvious that robots should do it.

Secondly, there are also risky tasks that we can replace the human factor with robots to decrease the risk and keeping people from getting hurt.

Thirdly, there are tasks that cannot be done by humans such as exploring and examining rocks at MARS for years. As humans can’t live on MARS right now we just send robots instead.

With all the previous mentioned reasons we need robotic arms to replace humans and provide precise and accurate interaction with the objects around them.

## 1.4 Techniques

We are using 7 degree of freedom robot with two arms in our project, we simulate the robot and environment on Gazebo simulation. We’re using Rviz which is a visualization software, that will allow us to view the gazebo data in case of a simulation is used, and also can be used for real world data in case we used a real robot.

We are using ROS (robot operating system) framework which allows us to tackle each sub-problem of the pick and place problem separately and also make it easier to develop on real robot as ROS architecture is built on topics and publish subscribe, which will allow us to easily deploy our system to a real robot arm in the feature.

Every robot should perform three main tasks which are perception, decision making and action

Our robot will perceive the surrounding environment through a camera using computer vision algorithms to collect some aspects from the environment and based on the information processed from the camera the robot will decide the action whether to keep looking for an object or if the robot found an object.

If the robot found an object, then the robot will start to plan its action which is three paths the arm will follow to perform a correct and precise pick and place action.

The first path is from the robot arm to the object, the second path is from the object to the desired place and the third path is to make the robot arm return to its initial pose.

After that the robot will calculate the forward and inverse kinematics to perform accurate movements to follow the planned paths.

### 

## 1.5 Applications

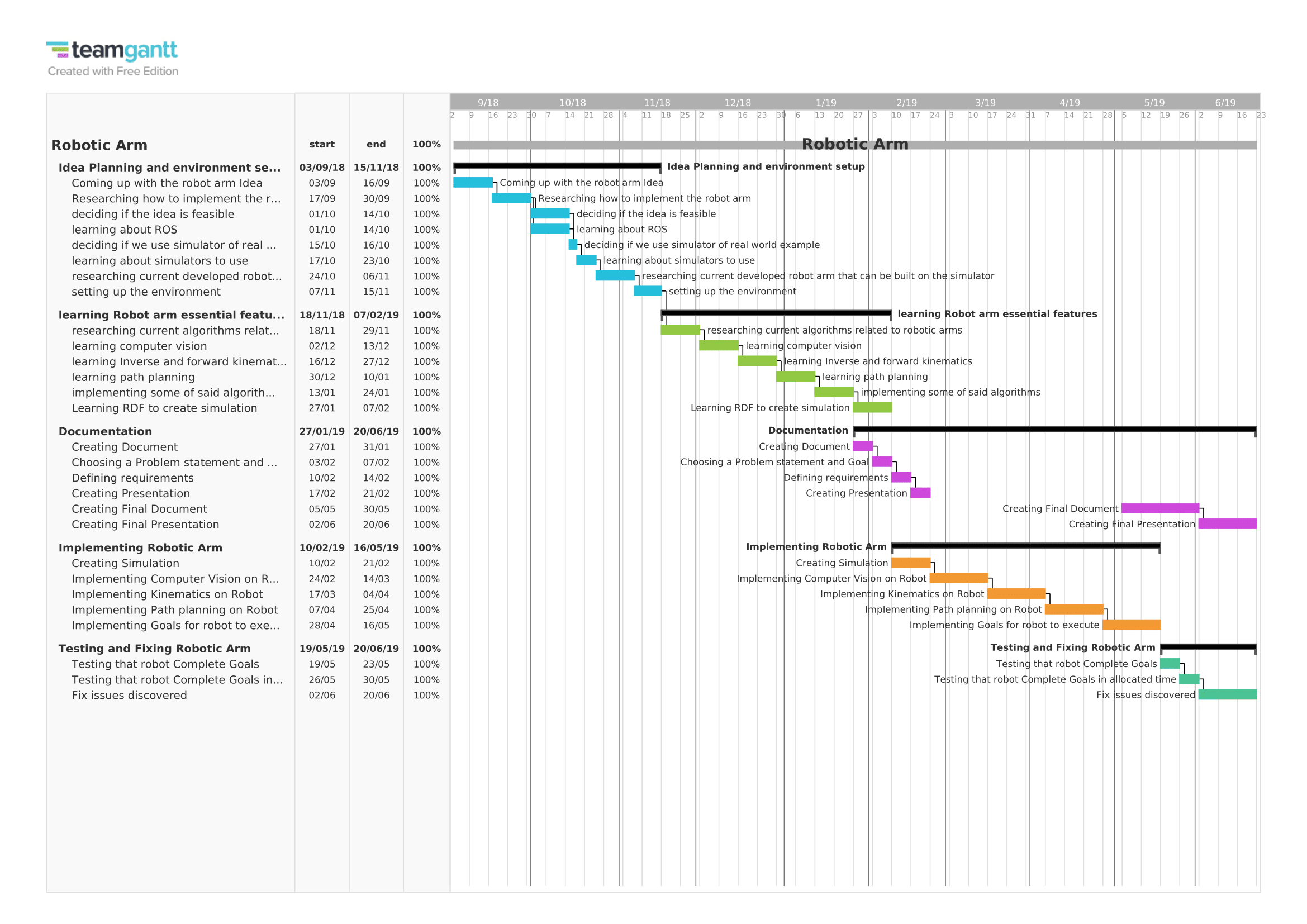
* Painting vehicles
* Soldering
* Pick and place
* Act in a human-designed environment: send the arm on a mobile base to a damaged/radioactive building and use the arm to open the door and manipulate the tools by itself or remotely controlled

## 1.5 Problem definition

The problem that we are tackling is one of the main problems in the robotics field which are the pick and place problem. While it may seem like an easy task, many companies supervise a lot of competitions to solve the pick and place problem accurately and efficiently such as Amazon picking challenge.

The robot’s objective in this project is to identify some objects in the environment and pick them then place them in the destination box and then return to its initial position.

## 1.7 Work Plan



## 1.8 Report Organization

The material presented at the document is organized into 5 chapters. After this introductory chapter, chapter 2 talks about the related work which includes, The closest examples of the project and the main differences between them and my project and contains references to the related work.

Chapter3, System specification which includes system architecture, the reason behind choosing this specific architecture, system flow, system input and output and obstacles.

Chapter4, Technologies and tools In this chapter we will give a brief description about ROS(Robot Operating System) framework and its architecture while mentioning the key elements in said architecture. Secondly, we will be talking about Gazebo which is a simulation to the both the robot and the environment, while also mentioning why we are using a simulator instead of a real robot and the advantages of using a simulator while developing using ROS. Thirdly, we will talk about rviz, our visualization tool for the robot sensors and what the robot is perceiving in real time from the environment created by Gazebo.

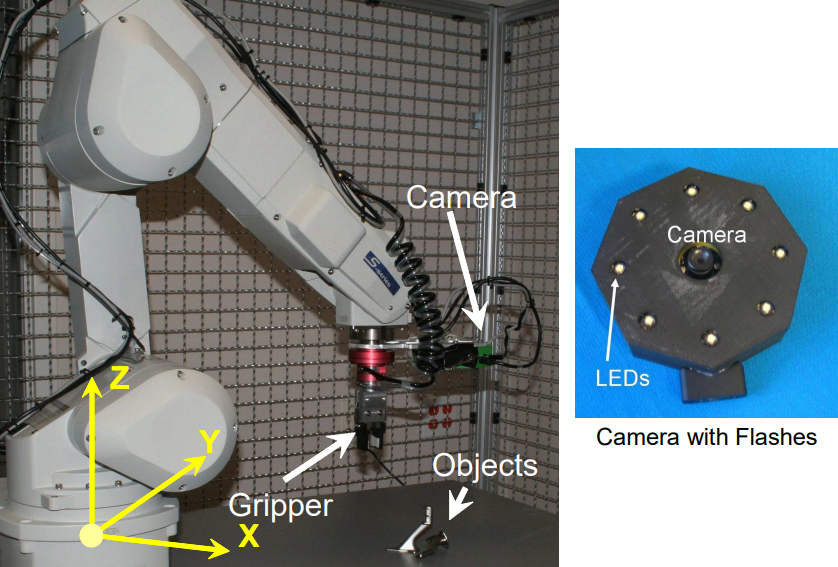
Chapter5, Implementation in this chapter we will discuss our algorithms in detail and provide also our calculations that were used in computer vision, kinematics and path planning.

Chapter6, Debugging and Testing in this chapter we will discuss the errors and how it was fixed, Tuning of some hyper-parameters and the accuracy result. And what to do for further improvement and future work.

# Chapter 2: Related Work

## 2.1 Vision based robot systems

Vision based robot systems have been the focus of significant research in both academia and industry. These systems typically employ single/multiple cameras and illumination devices to analyze the scene, locate the part and provide feedback to the robot arm for subsequent operations. To successfully grip and pick up parts, the vision system needs to recognize the position and the orientation of the objects. The vision sensor can be mounted on the end-effector of the robot arm, or located at a position near the robot. Vision based robot systems can be broadly classified into (a) 2D, (b) 2.5D, and (c) 3D vision systems. 2D vision systems have been successfully employed in several industrial applications. Most of the current vision systems fall into this category. These systems have been used for several tasks such as inspection and limited part acquisition. Typically such systems can recognize the in-plane orientation and location of the part but cannot determine the out of plane rotation and the distance of the part precisely. They require parts to be non-overlapping and placed on a flat surface. A model based approach can be used for 3D pose estimation. Edges are extracted in captured 2D images, and the contours of the object are detected by connecting the edges. The detected contours are then matched with a stored CAD model and the location and orientation of the object is estimated. However, these systems are highly susceptible to background color and illumination variations. In contrast, our approach based on depth edges can easily handle challenging backgrounds and nonuniform illumination. 3 Objects Gripper Camera Y Camera with Flashes X Z



*Figure 1: Experimental setup. A six degree of freedom (DOF) robot arm equipped with a pneumatic gripper is used for experiments. The camera attached to the robot arm is housed inside a plastic case and is surrounded by eight light emitting diodes (LED’s) controlled by a microcontroller. Objects to be picked are placed on a table as shown. The world coordinate system is attached to the base of the robot with the axis directions as shown.*

**2.5D vision systems** augment the 2D vision system by also calculating the distance of the object from the change in the size of the image of the object or by finding depths at few points. However, they cannot estimate the exact out of plane rotation and are often unreliable in their depth estimates. Such systems often misleadingly claim to estimate 3D pose but can only handle a few degrees of out of plane rotation for simple objects.

**3D vision systems** use sensors for estimating the 3D geometry of the scene. The object is recognized and localized by comparing the estimated range image with the standard orientated CAD models in a database. 3D range data can either be obtained with shape-from-texture, laser triangulation or edge-based binocular stereo. Some of the popular approaches are described below. Our system does not require a 3D sensor but it can estimate the three dimensional pose of the object using depth edges.

• **Stereo Vision**: Stereo systems use two cameras looking at the object to estimate the stereo has a high degree of sensitivity of the depth estimates with the noise in feature localization. Another problem with stereo is that the depths are recovered only at the feature points and not on the entire object. The reduced accuracy can be tolerated for certain applications such as un-racking body panels in body shops, but is not sufficient for accurate picking of the object.

• **Laser triangulation**: These systems use structure light [Sic] to create a pattern on the surface of the object, which is viewed from a camera. The laser triangulation can recover the 3D point cloud on the object surface. This technology has been used for applications involving edge tracking for welding, 4 sealing, glue deposition, grinding, waterjet cutting and deburring of flexible and dimensionally unstable parts. Use of lasers for part pose determination requires registration and accounting for shadows/occlusions. Laser triangulation does not work well on specular shiny objects due to laser light being reflected from the object surface. In addition, the use of lasers also leads to safety issues when deployed in close proximity to human operators.

**Active Illumination**: Controlling illumination is important for vision algorithms. Back-light illumination is used to segment objects by illuminating them from behind. In bright-field illumination, the light comes in approximately perpendicular to the object surface. The whole object appears bright, with features displayed as a continuum of gray levels. This sort of illumination is used for most general-vision applications. In dark-field illumination, the object is illuminated at a low angle from a point parallel to its surface. Texture and other angular features appear bright while most of the object appears dark. Dark-field illumination is useful for imaging surface contamination, scratches, and other small raised features. In coaxial illumination, the object is illuminated from precisely the direction of the imaging lens using a beam-splitter. Coaxial illumination is used to inspect features on flat, specular surfaces, to image within deep features, and to eliminate shadows. Shadows are usually considered a nuisance, but in our approach the illumination source is intentionally placed close to the camera and cast shadows are utilized to estimate depth edges. Several vision approaches use active illumination to simplify the underlying problem. Nayar et al. Recover shape of textured and textureless surfaces by projecting an illumination pattern on the scene. Shape from structured light has been an active area of research for 3D capture. Raskar et al. Proposed the multi-flash camera (MFC) by attaching four flashes to a conventional digital camera to capture depth edges in a scene. Crispell et al. Exploited the depth discontinuity information captured by the MFC for a 3D scanning system which can reconstruct the position and orientation of points located deep inside concavities. The depth discontinuities obtained by the MFC have also been utilized for robust stereo matching, recognition of finger-spelling gestures, automated particle size analysis with applications in mining and quarrying industry and for 3D segmentation. Our approach also uses a variant of MFC (with 8 flashes) to extract depth discontinuities, which are then used to segment objects and estimate their 3D pose.

**Model based Pose Estimation** has been a topic of significant research in computer vision. Initial work on using a CAD model and features in intensity images for 3D pose estimation was shown in [Low87, Low91]. An algorithm is proposed for pose estimation from given 2D/3D correspondences. Silhouettes have also been used for model based human pose estimation and object recognition/classification. In these approaches, background segmentation to obtain the silhouettes is typically a difficult problem. We show that by using cast shadows, obtaining silhouettes is easy even for challenging environments. Several techniques on classifying objects based on silhouettes assume complete closed contours, but our approach can work with incomplete silhouettes and missing depth edges.

## 2.2 onVision Guided Pick and Place Robotic Arm

Based on “onVision Guided Pick and Place Robotic Arm System Based on SIFT” paper

Their robot work is submodule to 4 phases

1. In the first phase the user initiates the system through user interface to choose the target object by selecting the object image from database.
2. In the second phase the snapshot of the current scene is taken in real time, and then the objects in the current scene image are detected and localized
3. In the third phase object selection is done.
4. The fourth phase object Localization is done and coordinates are provided to the microcontroller which directs robotic arm manipulator for precise movements for pick and place operation.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Commanded action | Live Actions | Development | Camera perceived Angle |
| onVision | Work based on user commands | Doesn’t do any work without user command | Developed without any framework | Only the front-view |
| Our robot | Work based on user commands | Continue working if any work appeared in the environment | Developed using ros framework | The front-view  And both the side-views |

# 

# Chapter 3: System Specification

This project is a research project applied on simulations.

## 

## 3.1 System Architecture

This robot is developed using ros framework which provide the overall system architecture which consist of Topics and subscribers.

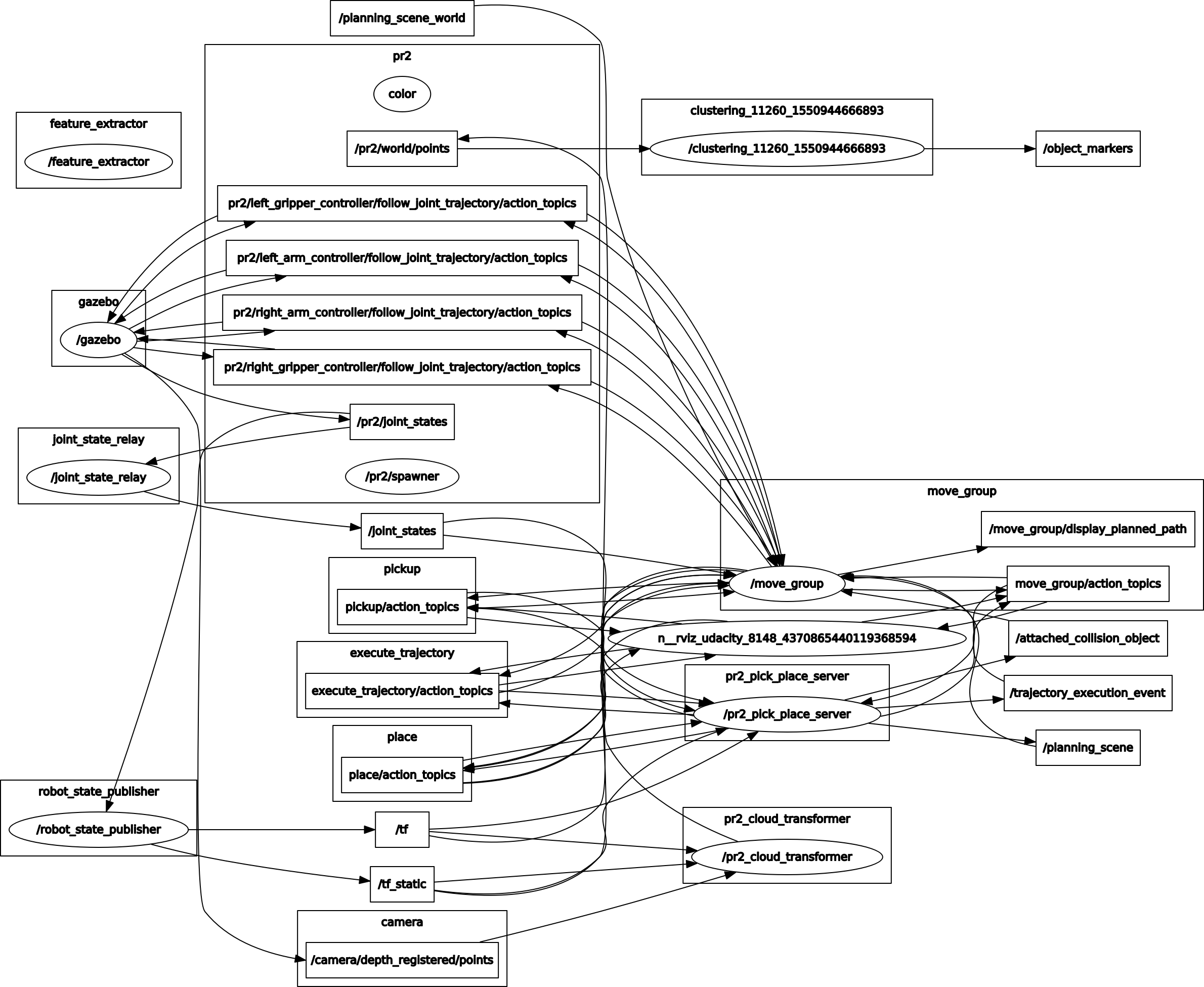
As shown in the next.

In feature \_extractor topic nothing is publishing or subscribing on it as it was only used in the initial phase to create and train the machine learning model.

The Camera topic Gazebo publish on that topic the video stream while the pr2\_cloud\_transformation topic subscribe on it and my code gets the stream and apply some computer vision algorithm to get all the information needed then pass them to other topics to initiate more action!

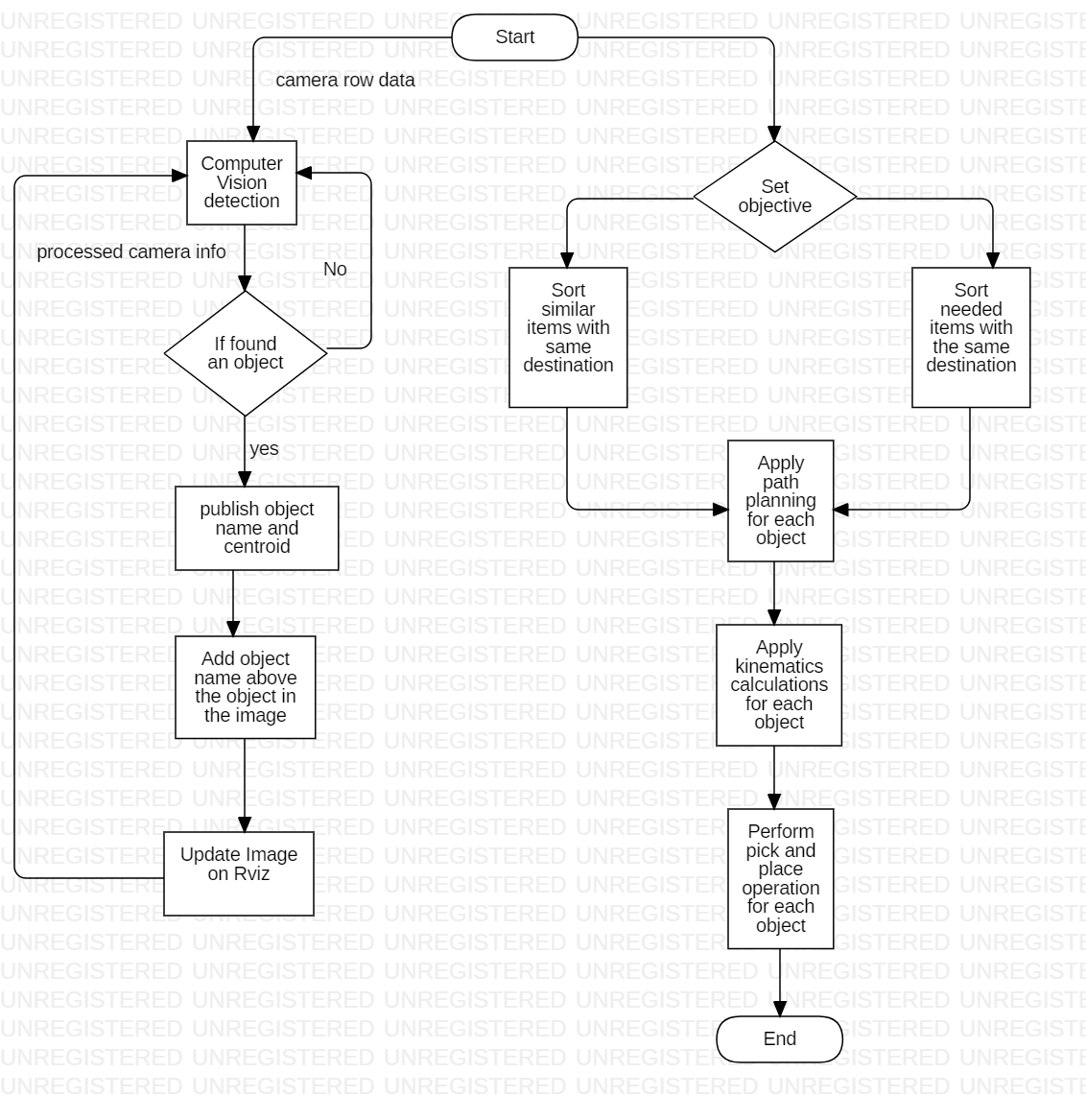
In pr2 it includes many topics which get the goal pose and initial pose and other topics for joints states, trajectories, paths and actions which Gazebo subscribe on it to apply any action by the robot.

In Move group it contains all the heavy work topics that calculate the kinematics and path planning and then publish them on Pr2 group so that Gazebo would be able to get those command and perform them on the simulator.



System Architecture in term of Topics.

## 3.2 System flow and techniques



Flowchart that represents system flow.

Our system is solving a problem in the robotics domain which is a very abstract domain

Which contains several domain and technologies inside.

1. In Machine learning domain a creation of a model is done using a support vector machine (SVM) that can classify different objects.
   1. First we will extract features from objects by loading each object in front of the RGB-D camera in the Gazebo simulation with different poses and same theses images.
   2. Then use the extracted features from simulation and use it in the SVM to create a model.
2. In Computer vision domain several filters and techniques are used to detect and identify different objects then calculate each object centroid
   1. First converte ros message to pcl image type, then implement different filter techniques.
   2. First filter is Statistical outlier filter to reduce the noise and have a clean image.
   3. Next Voxel grid filter to downsample the image and have faster processing.
   4. After that apply PassThrough filter to crop the image from the z-axis to locate the table only, x-axis to decrease the table width.
   5. After that Table and Objects Segmentation by apply RANSAC to identify the table without objects (inliers) and from it we can extract the objects only from the image (outliers).
   6. Then Clustering using Euclidean Clustering to create clusters of the objects
   7. After that identify each cluster in the image using trained model.
   8. Then at last calculate each object centroid.
3. Finally Path planning and applying kinematics to calculate each joint angle to follow that path.

## 

## 

## 

## 

## 3.3 System Input/Output

Input of the system

1. Pickup list that specifies the items that the robot should pick and the designated box for each item
2. The data perceived from the RGB-D camera.

Outputs expected from the system are

1. Robot action to pick and place each object to the desired place.

## 3.4 Obstacles

1. Financial obstacle as the robot arm is very expensive but luckily, simulations like Gazebo helps to simulate the real world and robot and Rviz helps to control over the robot whether the robot is in the real world or simulated world.
2. Processing power issue as not only the robot processing happens on the same machine but also Gazebo the robot and environment simulator runs on the same machine which needs a lot of processing to calculate the physics calculations including (gravity, collision, force applied on each object, etc ..) which sometimes affect the performance of the robot.
3. There is an issue in Gazebo were the robot gripper slips the objects from it’s gripper while applying the needed force on the items but Gazebo fails to identify the object as attached which result in making the robot not picking the objects.

# 

# Chapter 4: Technologies and tools used

## 4.1 ROS

The Robot Operating System (ROS) is a framework for writing robot software. It is a

collection of tools, libraries, and conventions that aim to simplify the task of creating

complex and robust robot behavior across a wide variety of robotic platforms.

Why? Because creating truly robust, general-purpose robot software is hard. From

the robot’s perspective, many problems that seem trivial to humans can actually

encompass wild variations between instances of tasks and environments.

Consider a simple “fetch an item” task, where an office-assistant robot is instructed to

retrieve a stapler. First, the robot must understand the request, either verbally or

through some other modalities, such as a web interface, email, or even SMS. Then, the

robot must start some sort of planner to coordinate the search for the item, which

will likely require navigating through various rooms in a building, perhaps including

elevators and doors. Once arriving in a room, the robot must search desks cluttered

with similarly sized objects (since all handheld objects are roughly the same size) and

find a stapler. The robot must then retrace its steps and deliver the stapler to the

desired location. Each of those sub problems can have arbitrary numbers of complicating

factors. And this was a relatively simple task!

Dealing with real-world variations in complex tasks and environments is so difficult

that no single individual, laboratory, or institution can hope to build a complete system

from scratch. As a result, ROS was built from the ground up to encourage collaborative

robotics software development. For example, in the “fetch a stapler” problem,

one organization might have experts in mapping indoor environments and could

contribute a complex yet easy-to-use system for producing indoor maps. Another

group might have expertise in using maps to robustly navigate indoor environments.

Yet another group might have discovered a particular computer vision approach that

works well for recognizing small objects in clutter. ROS includes many features

specifically designed to simplify this type of large-scale collaboration.

### 4.1.1 A Brief History

open collaboration framework was felt by many people in the robotics research community.

Various projects at Stanford University in the mid-2000s involving integrative,

embodied AI, such as the STanford AI Robot (STAIR) and the Personal Robots

(PR) program, created in-house prototypes of the types of flexible, dynamic software

systems described in this book. In 2007, Willow Garage, Inc., a nearby robotics incubator,

provided significant resources to extend these concepts much further and create

well-tested implementations. The effort was boosted by countless researchers who

contributed their time and expertise to the core of ROS and its fundamental software

packages. Throughout, the software was developed in the open using the permissive

BSD open source license, and it gradually became widely used in the robotics research community.

From the start, ROS was being developed at multiple institutions and for multiple

robots. At first, this seemed like a headache, since it would have been far simpler for

all contributors to place their code on the same servers. Ironically, over the years, this

has emerged as one of the great strengths of the ROS ecosystem: any group can start

their own ROS code repository on their own servers, and they will maintain full ownership

and control of it. They don’t need anyone’s permission. If they choose to make

their repository publicly visible, they can receive the recognition and credit they

deserve for their achievements and benefit from specific technical feedback and

improvements like all open source software projects.

The ROS ecosystem now consists of tens of thousands of users worldwide, working

in domains ranging from tabletop hobby projects to large industrial automation

systems.

### 4.1.2 Philosophy behind ROS

All software frameworks impose their development philosophies on their contributors

directly or indirectly, through their idioms and common practices. Broadly

speaking, ROS follows the Unix philosophy of software development in several key

aspects. This tends to make ROS feel “natural” for developers coming from a Unix

background but somewhat “cryptic” at first for those who have primarily used graphical

development environments on Windows or Mac OS X. The following paragraphs

describe several philosophical aspects of ROS:

### 4.1.3 Peer to peer Architecture

ROS systems consist of numerous small computer programs that connect to one

another and continuously exchange messages. These messages travel directly

from one program to another; there is no central routing service. Although this

makes the underlying “plumbing” more complex, the result is a system that scales

better as the amount of data increases.

### 

### 4.1.4 Tools-based

As demonstrated by the enduring architecture of Unix, complex software systems can be created from many small, generic programs. Unlike many other robotics

software frameworks, ROS does not have a canonical integrated development

and runtime environment. Tasks such as navigating the source code tree, visualizing

the system interconnections, graphically plotting data streams, generating

documentation, logging data, etc. are all performed by separate programs. This

encourages the creation of new, improved implementations, since (ideally) they

can be exchanged for implementations better suited for a particular task domain.

Recent versions of ROS allow many of these tools to be composed into single

processes for efficiency or to create coherent interfaces for operators or debugging,

but the principle remains the same: the individual tools themselves are relatively

small and generic.

### 4.1.5 Multilingual

Many software tasks are easier to accomplish in “high-productivity” scripting

languages such as Python or Ruby. However, there are times when performance

requirements dictate the use of faster languages, such as C++. There are also various

reasons that some programmers prefer languages such as Lisp or MATLAB.

Endless email flame wars have been waged, are currently being waged, and will

doubtless continue to be waged over which language is best suited for a particular

task. Acknowledging that all of these opinions have merit, that languages have

different utilities in different contexts, and that each programmer’s unique background

is hugely important when choosing a language, ROS chose a multilingual

approach. ROS software modules can be written in any language for which a client

library has been written. At the time of writing, client libraries exist for C++,

Python, LISP, Java, JavaScript, MATLAB, Ruby, Haskell, R, Julia, and others. ROS

client libraries communicate with one another by following a convention that

describes how messages are “flattened” or “serialized” before being transmitted

over the network.

### 4.1.6 Free and open source

The core of ROS is released under the permissive BSD license, which allows commercial and noncommercial use. ROS passes data between modules using interprocess

communication (IPC), which means that systems built using ROS can

have fine-grained licensing of their various components. Commercial systems,

for example, often have several closed source modules communicating with a

large number of open source modules. Academic and hobby projects are often

fully open source. Commercial product development is often done completely

behind a firewall. All of these use cases, and more, are common and perfectly

valid under the ROS license.

Moving on from the philosophy comes the Architecture of the ROS framework, where we will introduce the ROS Graph, nodes, edges and roscore.

### 

### 4.1.7 Reusability

The ROS conventions encourage contributors to create standalone libraries and

then wrap those libraries so they can send and receive messages to and from

other ROS modules. This extra layer is intended to allow the reuse of software

outside of ROS for other applications, and it greatly simplifies the creation of

automated tests using standard continuous integration tools.

### 

### 4.1.8 The ROS Graph

One of the original “challenge problems” that motivated the design of ROS was

fondly referred to as the “fetch an item” problem. Imagine a relatively large and complex

robot with several cameras and laser scanners, a manipulator arm, and a wheeled

base. In the “fetch an item” problem, the robot’s task is to navigate a typical home or

office environment, find the requested item, and deliver it to the requested location.

This task, like many robotics tasks, led to several observations about many robotics

software applications, which became some of the design goals of ROS:

* The application task can be decomposed into many independent subsystems,

such as navigation, computer vision, grasping, and so on.

* These subsystems can be used for other tasks, such as doing security patrols,

cleaning, delivering mail, and so on.

* With proper hardware and geometry abstraction layers, the vast majority of the

application software can run on any robot.

These goals can be illustrated by the fundamental rendering of a ROS system: its

graph. A ROS system is made up of many different programs running simultaneously

and communicating with one another by passing messages. It is convenient to use a

mathematical graph to represent this collection of programs and messages: the programs

are the graph nodes, and programs that communicate with one another are connected by edges? A sample ROS graph appears in Figure 2-1, which represents one

of the earliest implementations of the “fetch an item” application using ROS. The

details of this graph are not particularly important; it is just provided to illustrate

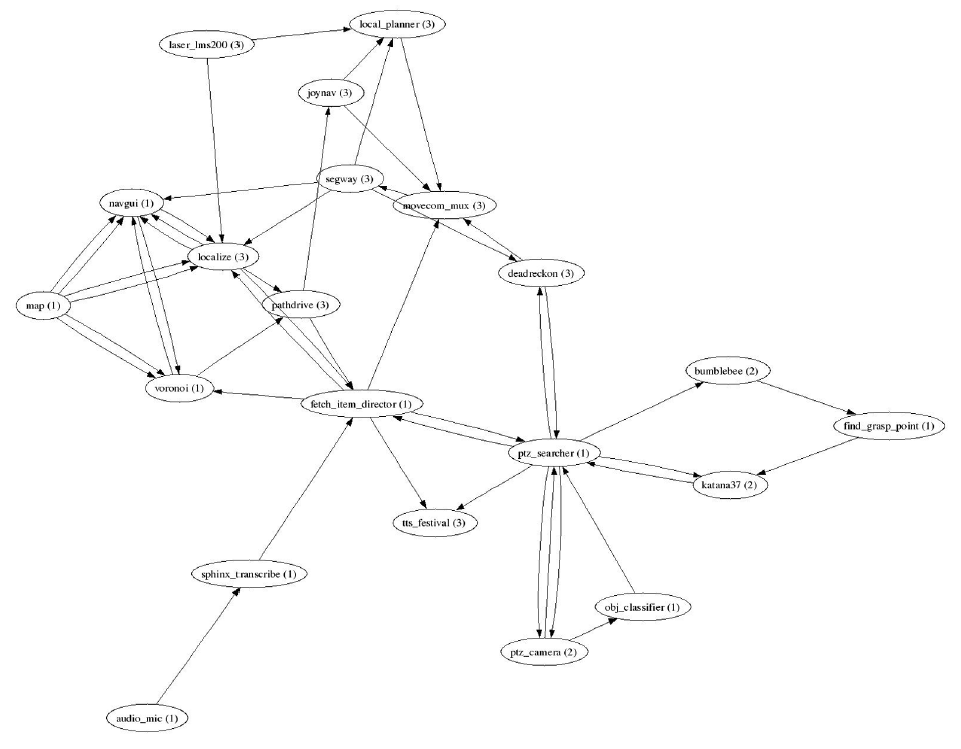
the general concept of a ROS system as a collection of nodes passing messages to one

another. We can represent any ROS system, large or small, in this way. In fact, this

representation is so useful for software development that we actually refer to ROS

programs as nodes, to help us remember that each program is just one piece of a

much larger system.



*Figure 2-1. ROS graph of a fetch-an-item robot—nodes in the graph represent individual*

*programs; edges represent message streams communicating sensor data, actuator commands,*

*planner states, intermediate representations, and so on*

To reiterate: a ROS graph node represents a software module that is sending or receiving

messages, and a ROS graph edge represents a stream of messages between two

nodes. Although things can get more complex, typically nodes are POSIX processes,

and edges are TCP connections. This offers additional fault tolerance: a software

crash will typically only take down its own process. The rest of the graph will stay up,

passing messages and functioning as normal. The circumstances leading up to the

crash can often be recreated by logging the messages entering a node and simply

playing them back at a later time inside a debugger.

However, perhaps the greatest benefit of a loosely coupled, graph-based architecture

is the ability to rapid-prototype complex systems with little or no software “glue”

required for experimentation. Single nodes, such as the object recognition node in a

“fetch an item” system, can be trivially swapped by simply launching an entirely different

process that accepts images and outputs labeled objects. Not only can a single

node be swapped, but entire chunks of the graph (subgraphs) can be torn down and

replaced, at runtime, with other subgraphs. Real-robot hardware drivers can be

replaced with simulators, navigation subsystems can be swapped, algorithms can be

tweaked and recompiled, and so on. Since ROS is creating all of the required network

backend on the fly, the entire system is interactive and designed to encourage

experimentation.

Among all the traffic that is moving around on this busy network, there must be an entity that allows nodes to find each other, so they can start passing messages. That entity is called roscore.

### 4.1.9 roscore

roscore is a service that provides connection information to nodes so that they can

transmit messages to one another. Every node connects to roscore at startup to register

details of the message stream it publishes and the streams to which it wishes to

subscribe. When a new node appears, roscore provides it with the information that it

needs to form a direct peer-to-peer connection with another nodes publishing and subscribing

to the same message topics. Every ROS system needs a running roscore, since without it, nodes cannot find other nodes.

However, a key aspect of ROS is that the messages between nodes are transmitted

peer-to-peer. The roscore is only used by nodes to know where to find their peers.

This is a bit subtle, and can lead to some misunderstandings, as programmers coming

from web-based backgrounds are often familiar with client/server systems, such as

web-browsers talking to web servers, where the roles of clients and servers are clearly

defined. The ROS architecture is a hybrid between a classical client/server system and

a fully distributed one, due to the presence of a central roscore that provides a name

service for the peer-to-peer message streams.

When a ROS node starts up, it expects its process to have an environment variable

named ROS\_MASTER\_URI. This variable is expected to contain a string of the form

http://hostname:11311/, which in this case would imply that there is a running

instance of roscore accessible on port 11311 somewhere on a host called hostname

that can be accessed over the network.

With knowledge of the location of roscore on the network, nodes register themselves

at startup with roscore and then query roscore to find other nodes and data streams

by name. Each ROS node tells roscore which messages it provides and which it

would like to subscribe to. roscore then provides the addresses of the relevant message

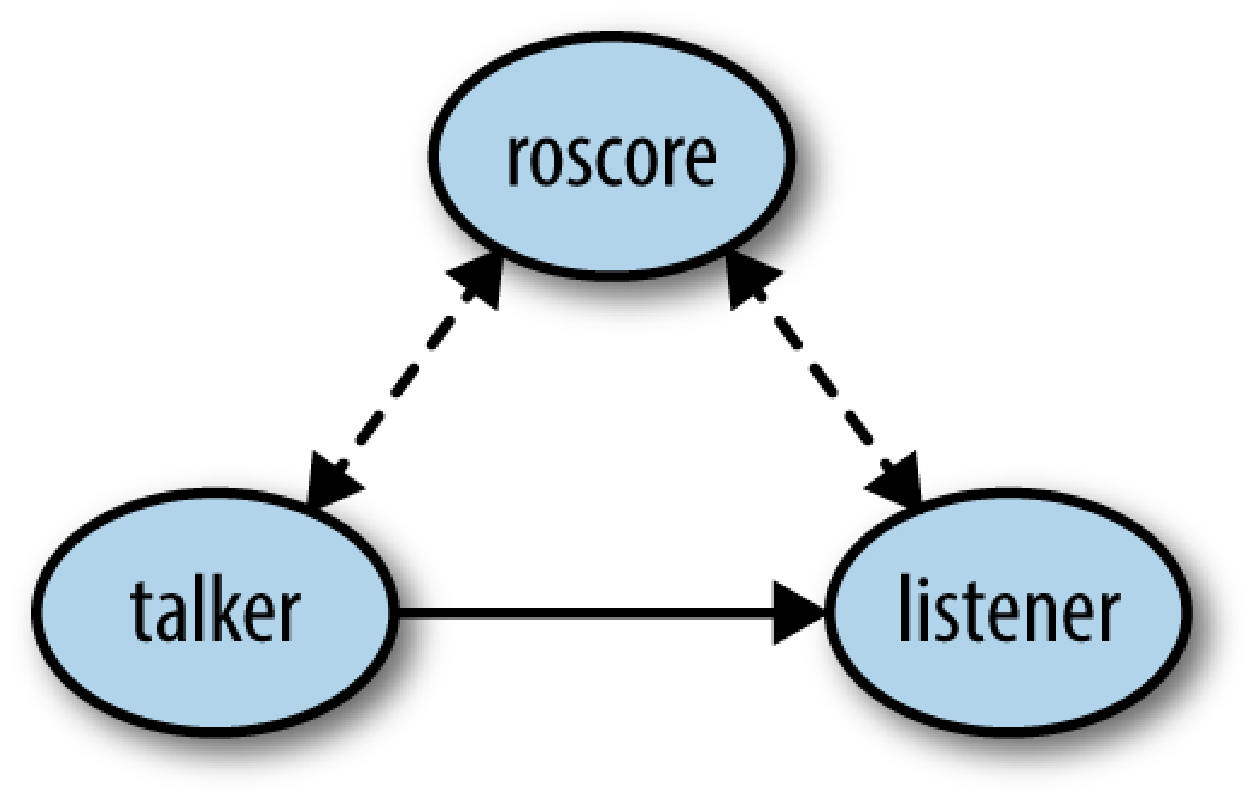
producers and consumers. Viewed in a graph form, every node in the graph can

periodically call on services provided by roscore to find its peers. This is represented

by the dashed lines shown in Figure 2-2, which show that in this minimalist two node

system, the talker and listener nodes can periodically make calls to roscore

while exchanging peer-to-peer messages directly themselves.



*Figure 2-2. roscore connects only ephemerally to the other nodes in the system*

roscore also provides a parameter server, which is used extensively by ROS nodes for

configuration. The parameter server allows nodes to store and retrieve arbitrary data

structures, such as descriptions of robots, parameters for algorithms, and so on.

## 

## 4.2 Gazebo

The second technology we are going to introduce is Gazebo which is a 3D simulator for our robot, PR2.

Although there are low-cost robots that can be built or bought then customized to our liking, we decided on using a simulator like Gazebo, because it is out of our area of expertise to handle the mechanical aspects of building robots and building a 7 degree of freedom robot arm from scratch is no easy feat as real robots come with a lot of challenges that would distract us from our main task which is controlling.

In addition, real robots require logistics including lab space,

recharging of batteries, and operational quirks that often become part of the institutional

knowledge of the organization operating the robot. Sadly, even the best robots

break periodically due to various combinations of operator error, environmental conditions,

manufacturing or design defects, and so on.

Many of these headaches can be avoided by using simulated robots. At first glance,

this seems to defeat the whole purpose of robotics; after all, the very definition of a

robot involves perceiving and/or manipulating the environment. Software robots,

however, are extraordinarily useful. In simulation, we can model as much or as little of reality as we desire. Sensors and actuators can be modeled as ideal devices, or they

can incorporate various levels of distortion, errors, and unexpected faults. Although

data logs can be used in automated test suites to verify that sensing algorithms produce

expected results, automated testing of control algorithms typically requires

simulated robots, since the algorithms under test need to be able to experience the

consequences of their actions.

Due to the isolation provided by the messaging interfaces of ROS, a vast majority of

the robot’s software graph can be run identically whether it is controlling a real robot

or a simulated robot. At runtime, as the various nodes are launched, they simply find

one another and connect. Simulation input and output streams connect to the graph

in the place of the device drivers of the real robot. Although some parameter tuning

is often required, ideally the *structure* of the software will be the same, and often the

simulation can be modified to reduce the amount of parameter tweaks required when

transitioning between simulation and reality.

As alluded to in the previous paragraphs, there are many use cases for simulated

robots, ranging from algorithm development to automated software verification. This

has led to the creation of a large number of robot simulators, many of which integrate

nicely with ROS like Gazebo.

Like all simulators, Gazebo (Figure 2-3) is the product of a variety of trade-offs in its

design and implementation. Historically, Gazebo has used the Open Dynamics

Engine for rigid-body physics, but recently it has gained the ability to choose between

physics engines at startup.

*Figure 2-3. Typical screenshot of the Gazebo simulator*

ROS integrates closely with Gazebo through the gazebo\_ros package. This package

provides a Gazebo plugin module that allows bidirectional communication between

Gazebo and ROS. Simulated sensor and physics data can stream from Gazebo to

ROS, and actuator commands can stream from ROS back to Gazebo. In fact, by

choosing consistent names and data types for these data streams, it is possible for

Gazebo to exactly match the ROS API of a robot. When this is achieved, all of the

robot software above the device-driver level can be run identically both on the real

robot, and (after parameter tuning) in the simulator.

## 4.3 Rviz

Rviz is the third technology we are going to introduce. Rviz stands for ROS visualization.

It is a general-purpose 3D visualization environment for robots, sensors, and algorithms.

Like most ROS tools, it can be used for any robot and rapidly configured for a particular application. For example, our pick and place problem, we want to be able to see the camera feed of the of the camera on the head of the robot.

Rviz can plot a variety of data types streaming through a typical ROS system, with

heavy emphasis on the three-dimensional nature of the data. In ROS, all forms of data

are attached to a frame of reference. For example, the camera on a PR2 robot is

attached to a reference frame defined relative to the center of the PR2’s mobile base.

The following figure 2-4 will show what the PR2’s camera is capturing.

So by using the gazebo simulate the environment and the robot for us, then using Rviz to visualize what the robot is sensing in the environment, we now have the technologies to start developing our 7 degree of freedom pick and place robot arm.

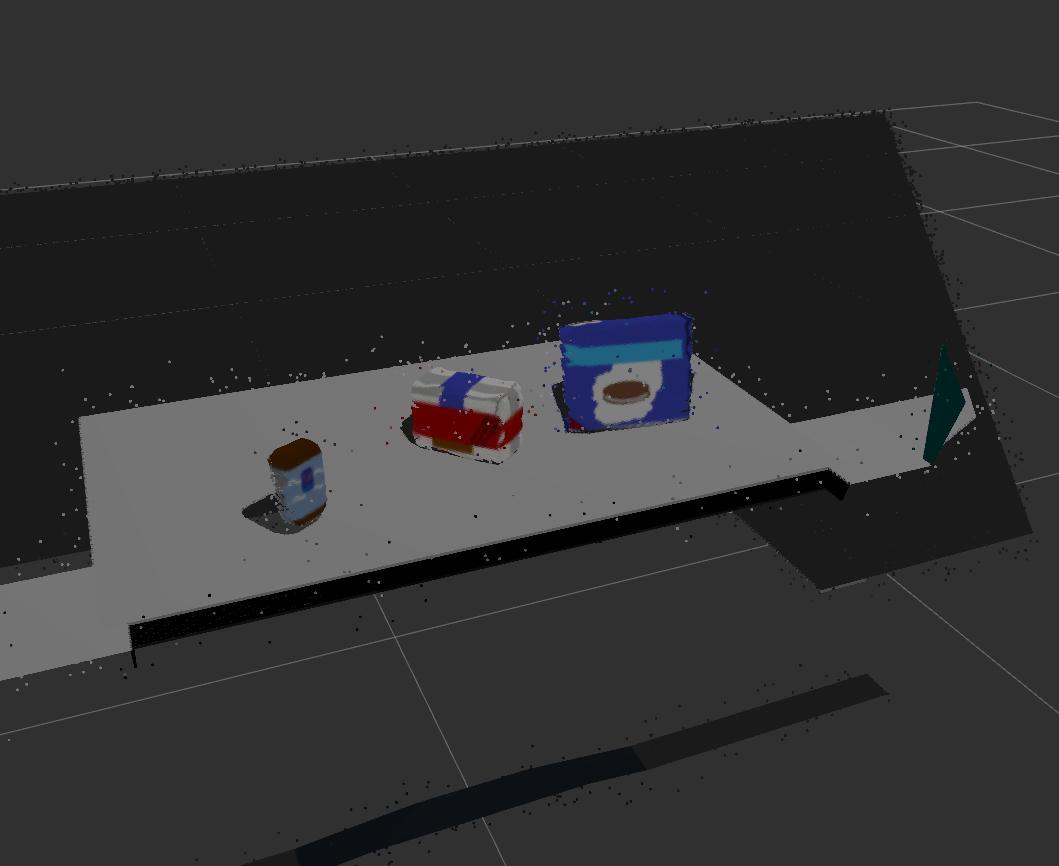
# Chapter 5: Implementation

This Chapter discuss in details all the algorithms that were used in this project.

## 5.1 Computer Vision

Pr2 robot is using RGB-D camera which not only get the RGB (red, green, blue) data but also get the depth which helps in the calibration process, it will allow you to combine the RGB camera data with the accompanying depth camera data to generate 3D point clouds.

At first the raw data contain noise which we will target to remove to improve the accuracy of the output.

Images from the environment that contain noise.

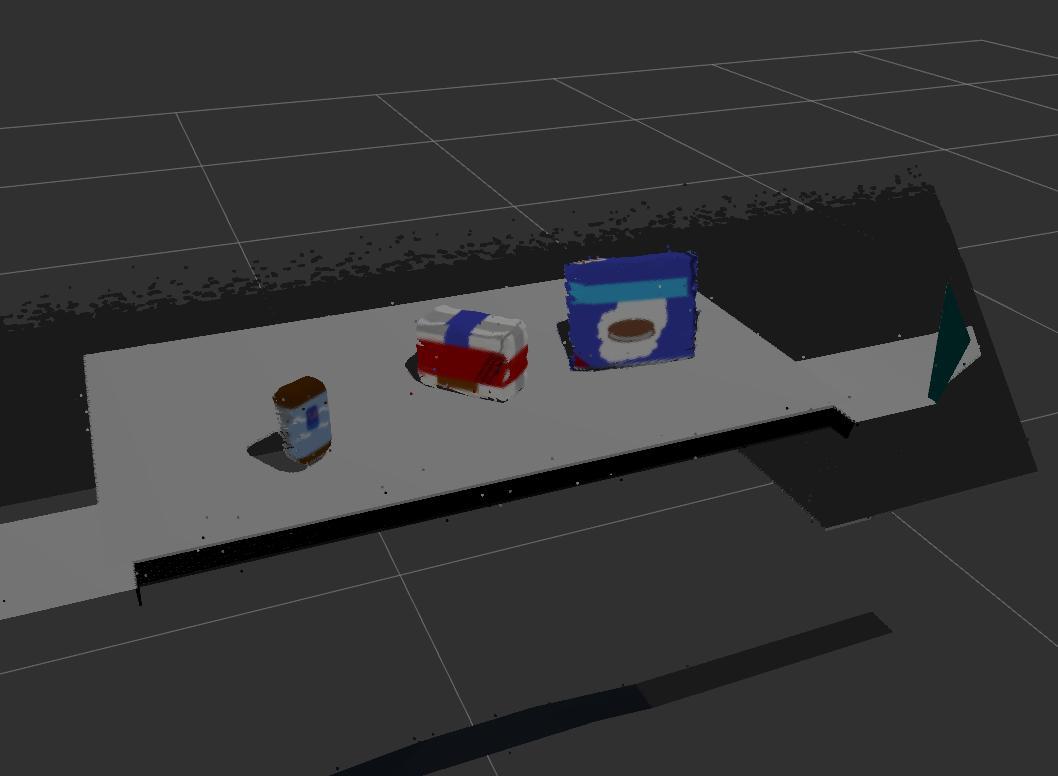
### 5.1.1 Statistical Outlier Filter

The algorithm iterates through the entire input twice: During the first iteration it will compute the average distance that each point has to its nearest k neighbors. The value of k can be set using **setMeanK()**. Next, the mean and standard deviation of all these distances are computed in order to determine a distance threshold. The distance threshold will be equal to: mean + stddev\_mult. The multiplier for the standard deviation can be set using **setStddevMulThresh()**. During the next iteration the points will be classified as inlier or outlier if their average neighbor distance is below or above this threshold respectively.

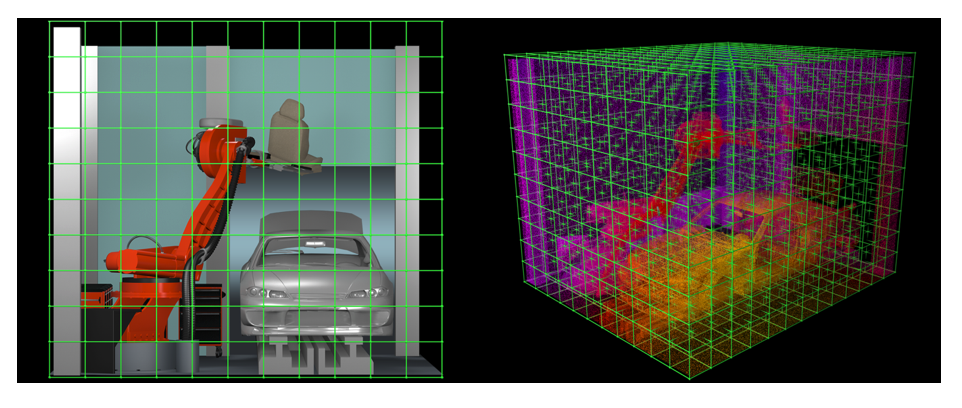
The neighbors found for each query point will be found amongst ALL points of **setInputCloud()**, not just those indexed by **setIndices()**. The **setIndices()** method only indexes the points that will be iterated through as search query points.

Here in fig 5.1.1 the result after removing the noise by applying

statistical outlier filter:

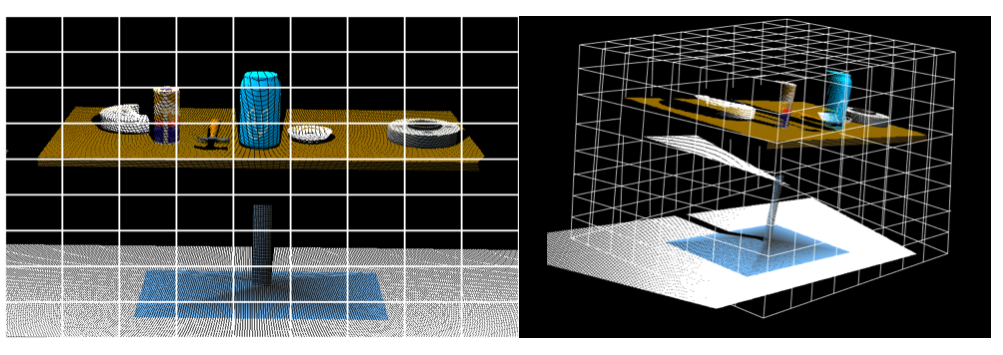
**5.1.1** Statistical outlier filter

### 5.1.2 Voxel Grid Downsampling



RGB-D cameras provide feature rich and particularly dense point clouds, meaning, the more points are packed in per unit volume than, for example, a Lidar point cloud. Running computation on a full resolution point cloud can be slow and may not yield any improvement on results obtained using a more sparsely sampled point cloud.

So, in many cases, it is advantageous to downsample the data. In particular, we are going to use a VoxelGrid Downsampling Filter to derive a point cloud that has fewer points but should still do a good job of representing the input point cloud as a whole.



A voxel grid filter allows you to downsample the data by taking a spatial average of the points in the cloud confined by each voxel. You can adjust the sampling size by setting the voxel size along each dimension. The set of points which lie within the bounds of a voxel are assigned to that voxel and statistically combined into one output point.

The real result from my project is shown in fig 5.1.2

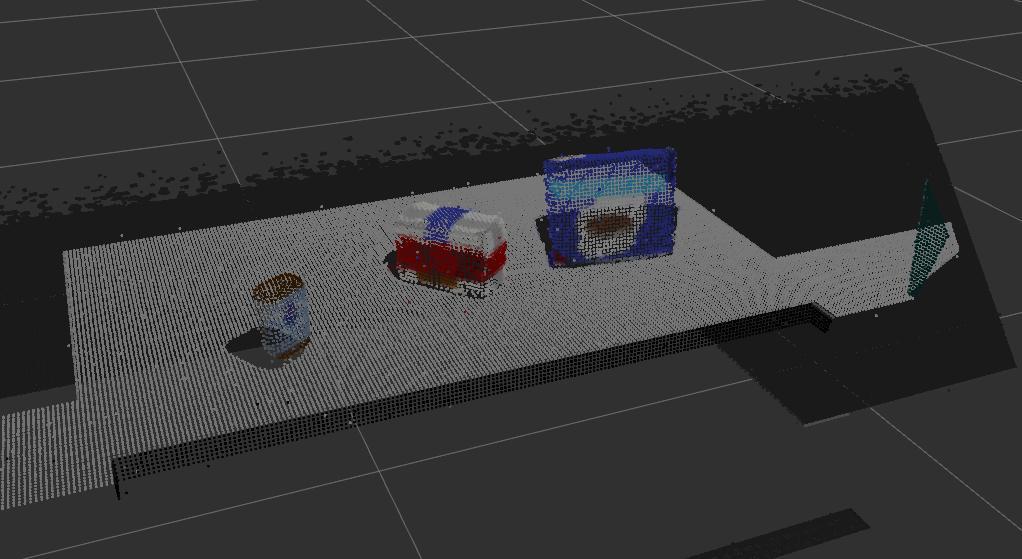


fig 5.1.2 Voxel Grid filter

### 5.1.3 Pass Through Filtering

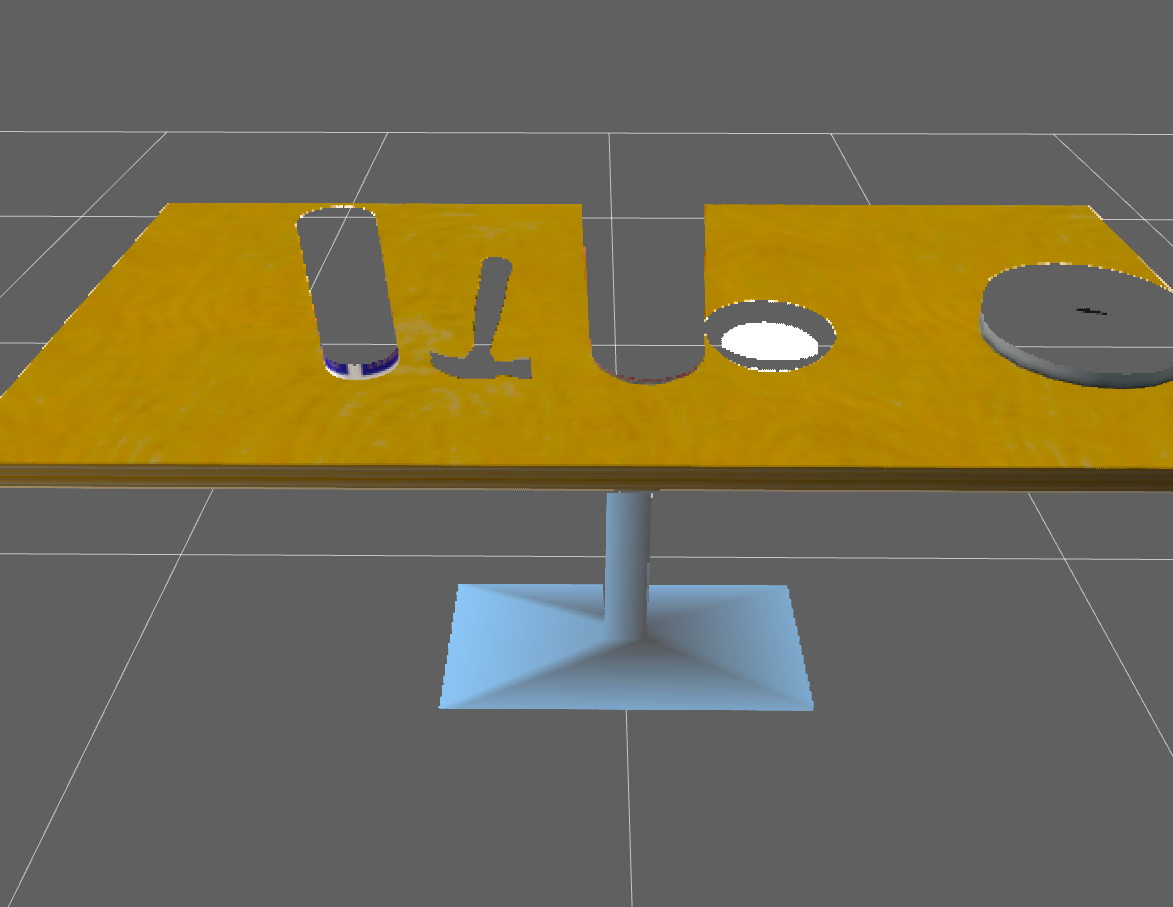


If you have some prior information about the location of your target in the scene, you can apply a Pass Through Filter to remove useless data from your point cloud. Which luckily we have.

The Pass Through Filter works much like a cropping tool, which allows you to crop any given 3D point cloud by specifying an axis with cut-off values along that axis. The region you allow to *pass through*, is often referred to as ***region of interest***.

For instance, in our tabletop scene we know that the table is roughly in the center of our robot’s field of view. Hence by using a Pass Through Filter we can select a region of interest to remove some of the excess data.

Applying a Pass Through filter along the z axis (the height with respect to the ground) to our tabletop scene in the range 0.1 to 0.8 gives the following result:



Oops! We've retained the table but filtered out all of the objects!

We can solve this problem by trying to increase the range of the z-axis so we could retain only the tabletop and the objects sitting on the table.

and fig 5.1.3 shows my result

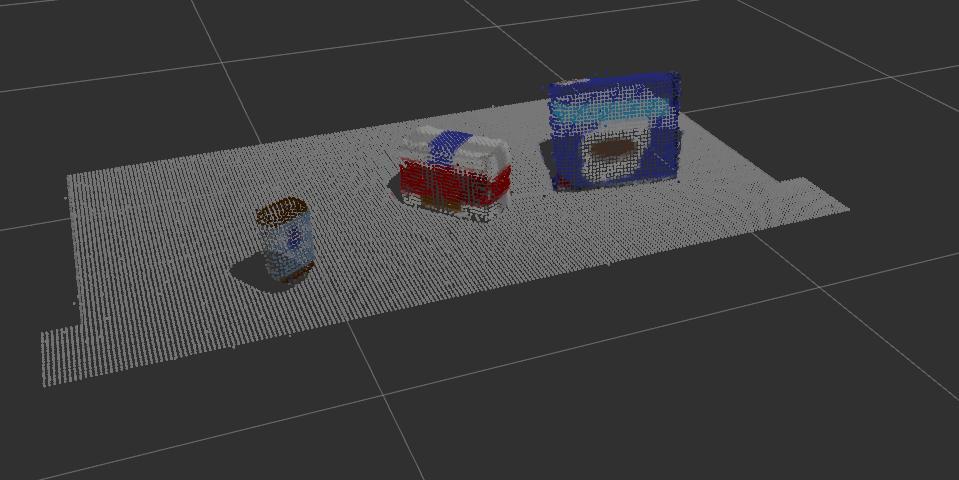


fig 5.1.3 PassThrough filter

### 5.1.4 RANSAC

In our tabletop scenario, combining some prior knowledge about the scene with a few point cloud filters, we were able to reduce the point cloud down to just the table and objects on top of it.

Next in our perception pipeline, we needed to remove the table itself from the scene. To do this we used a popular technique known as [Random Sample Consensus](https://en.wikipedia.org/wiki/Random_sample_consensus) or "RANSAC". RANSAC is an algorithm, that is used to identify points in dataset that belong to a particular model. In the case of the 3D scene we are working with here, the model we chose could be a plane, a cylinder, a box, or any other common shape.

The RANSAC algorithm assumes that all of the data in a dataset is composed of both inliers and outliers, where inliers can be defined by a particular model with a specific set of parameters, while outliers do not fit that model and hence can be discarded. Like in the example below, we can extract the outliers that are not good fits for the model.

like in fig 5.1.4

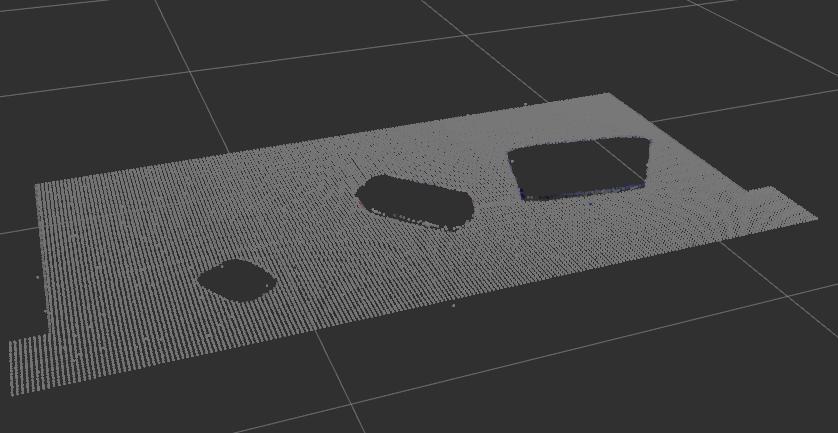


fig 5.1.4 RANSAC filter outlier

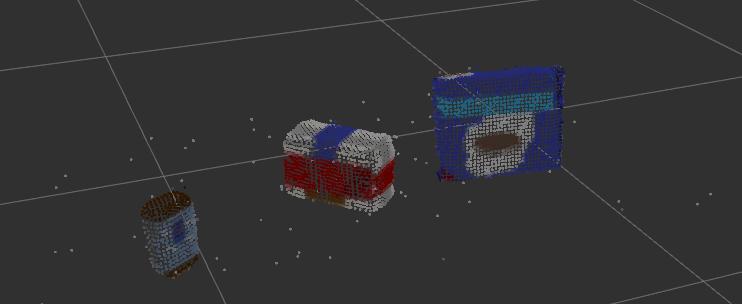
On the other hand, a disadvantage of RANSAC is that there is no upper limit on the time it can take to compute the model parameters. This is somewhat alleviated by choosing a fixed number of iterations but that has its own demerits.

If we chose a lower number of iterations, the solution obtained may not be optimal. In this way RANSAC offers a trade-off between compute time versus model detection accuracy.

Since the top of the table in the scene is the single most prominent plane, after ground removal, we can effectively use RANSAC to identify points that belong to the table and discard/filter out those points using a relatively low number of iterations.

Another popular use case of such plane segmentation appears in mobile robot autonomous navigation. For collision avoidance with objects and to determine traversable terrain, ground plane segmentation is an important part of a mobile robot’s perception toolkit.

Here is the result of the inlier filter (fig 5.1.5)

fig 5.1.5 RANSAC filter inlier

### 5.1.5 Clustering with PCL

In order to perform Euclidean Clustering, we must first construct a [k-d tree](http://pointclouds.org/documentation/tutorials/kdtree_search.php) from the cloud\_objects point cloud.

The k-d tree data structure is used in the Euclidean Clustering algorithm to decrease the computational burden of searching for neighboring points. While other efficient algorithms/data structures for nearest neighbor search exist, PCL's Euclidean Clustering algorithm only supports k-d trees.

To construct a k-d tree, we first convert the XYZRGB point cloud to XYZ, because PCL's Euclidean Clustering algorithm requires a point cloud with only spatial information.

So we now have a list of indices for each cluster (a list of lists). In the next step, we create a new point cloud to visualize the clusters by assigning a unique color for each segmented Object as fig 5.1.6 shows

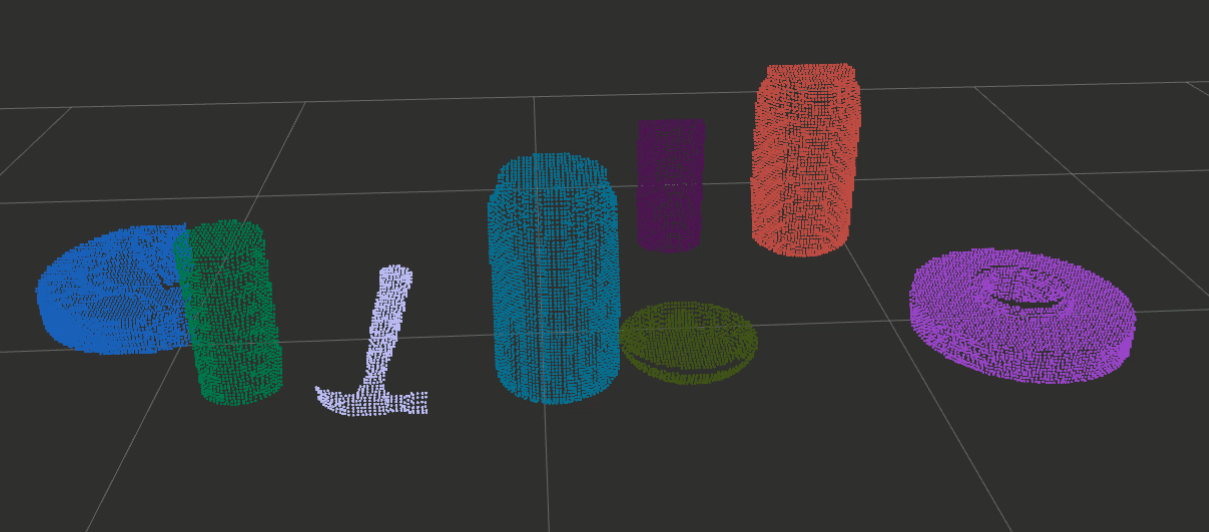
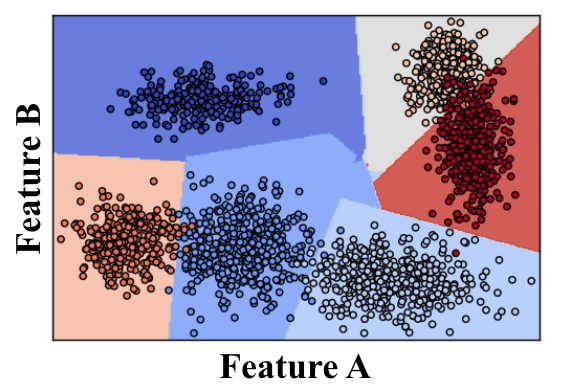


fig 5.1.6 Euclidean Clustering

## 5.2 Machine Learning

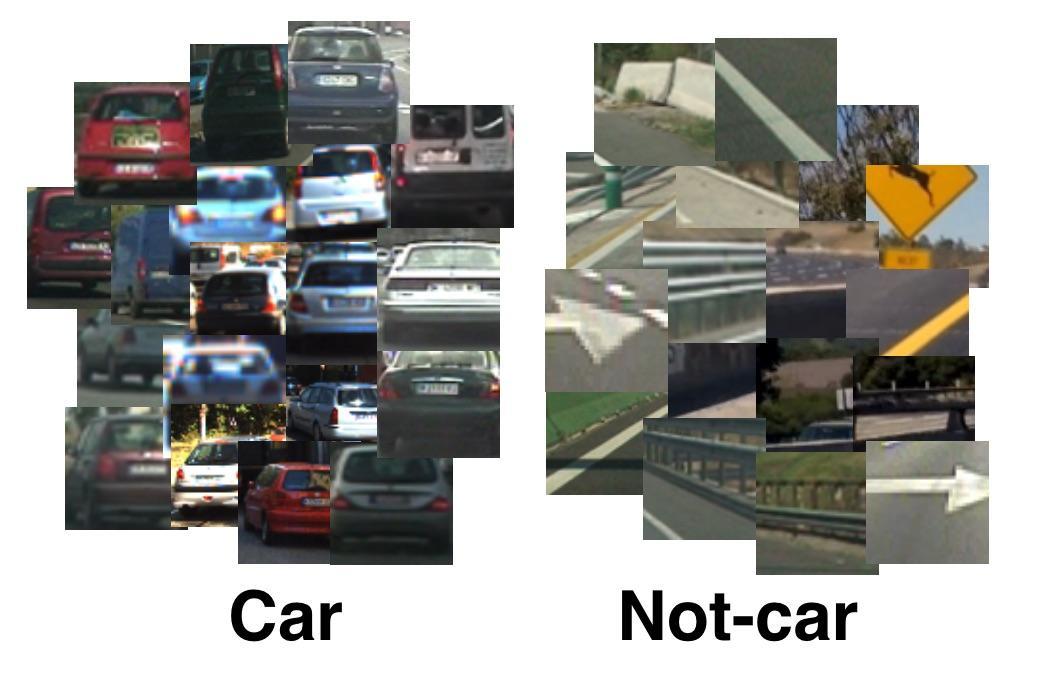
Since Our goal is to make the robot independent from the environment and user, Pr2 robot must be able to identify on his own different items and which designated box are set for each item, So a machine learning model must be developed with good accuracy so that the robot is reliable in identifying the different items.

### 5.2.1 Support Vector Machine



Support Vector Machine or "SVM" is a name for a particular supervised machine learning algorithm that allows you to characterize the parameter space of your dataset into discrete classes. SVMs work by applying an iterative method to a training dataset, where each item in the training set is characterized by a feature vector and a label. In the image above, each point is characterized by just two features, A and B. The color of each point corresponds to its label, or which class of object it represents in the dataset.

Applying an SVM to this training set allows you to characterize the entire parameter space into discrete classes. The divisions between classes in parameter space are known as "decision boundaries", shown here by the colored polygons overlaid on the data. Having created decision boundaries means that when you're considering a new object for which you have features but no label, you can immediately assign it to a specific class. In other words, once you have trained your SVM, you can use it for the task of object recognition!



We have now looked at how an SVM can be used to classify multi-class datasets, but only with two features describing each element. With our point cloud data, we’ll have a rich feature set containing color and surface normal histograms. Classification with a rich feature set works just the same as with two features, but it's harder to visualize, so we'll just learn through the example of image classification using color histograms.

To demonstrate image classification, we'll borrow an exercise from Udacity Self-Driving Car Nanodegree Program! In this exercise, the dataset is composed of hundreds of images of cars and other images which are found in the scene with a car, but are something else. The goal is to train an SVM to recognize whether an image contains a car or not based on an input feature vector composed of color histograms. Here we'll introduce a few more concepts related to preparing our training data and evaluating the performance of our classifier.

To begin with, we’ll read in our car and non-car images, extract the color features for each, then scale the feature vectors to zero mean and unit variance. After that we'll define a labels vector, shuffle and split the data into training and testing sets, and finally, define a classifier and train it!

At last I will state some of the advantages and disadvantages of the svm

1. The advantages of support vector machines are:

* Effective in high dimensional spaces.
* Still effective in cases where the number of dimensions is greater than the number of samples.
* Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
* Versatile: different [Kernel functions](https://scikit-learn.org/stable/modules/svm.html#svm-kernels) can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

2. The disadvantages of support vector machines include:

* If the number of features is much greater than the number of samples, avoid over-fitting in choosing [Kernel functions](https://scikit-learn.org/stable/modules/svm.html#svm-kernels) and regularization term is crucial.
* SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation (see [Scores and probabilities](https://scikit-learn.org/stable/modules/svm.html#scores-probabilities), below).

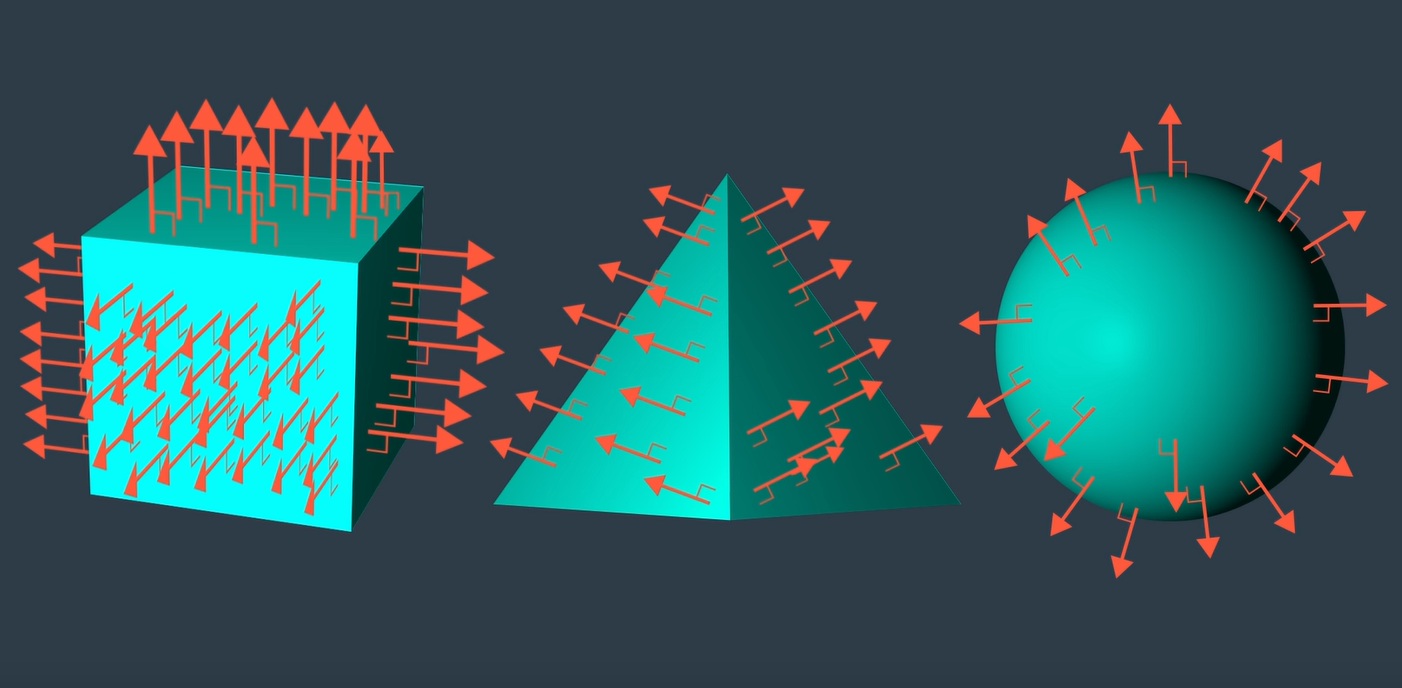
### 5.2.2 Generating Features for the dataset

### 

I Made a stick on the stick with the same height of the robot and put the camera above the stick and started generating the dataset by putting every object in front of the camera with different orientations and by that method we can get any dataset size!

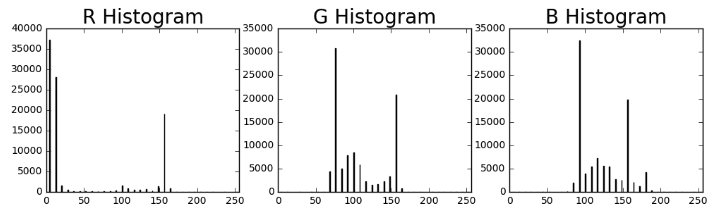
Now let's talk about the features we want to extract from the data set, We chose two features surface normal and color histogram

1. Surface normal is a representation of the normal of any shape like in fig 5.2.2.1

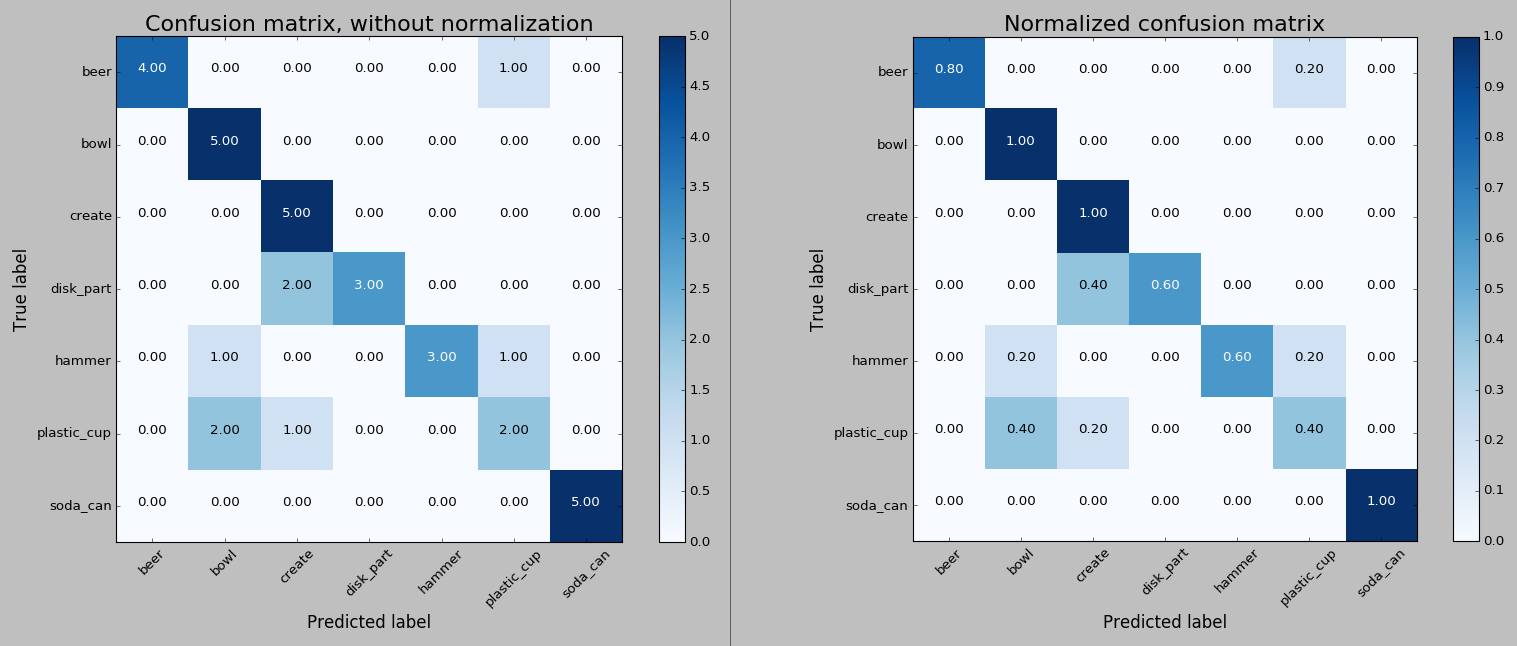


We can see how the normal surface is different for each object then we built a histogram of the result for faster processing

2. Color histogram is a representation of the distribution of colors in an image. For digital images, a color histogram represents the number of pixels that have colors in each of a fixed list of color ranges, that span the image's color space, the set of all possible colors.



Now what we have done is we trained a model using svm and after training the sklearn module that provide the svm functionality in python also produce the result of training in a confusion matrix as shown in fig 5.2.2.2



**fig** 5.2.2.2

At last we can now make the robot use the model to identify different set of objects by taking the output from the computer vision algorithm and use it as an input for the model, Now the next thing is to determine the pose and orientation of each object and determine which box to put the item into, Luckily the designated box for each object is fixed in our environment and user is the one who determine the boxes for the robot, But for the item pose and orientation (Centroid).

The **centroid** of a volume can be thought of as the geometric center of that area. It is the average position (x, y, and z coordinates) of all the points in the area. If this volume represents a part with a uniform density (like most single material parts) then the centroid will be the same as the center of mass. In our case we are not dealing with heavy weight objects so center of mass is not a problem so we won’t talk in our account.

So till now the robot could recognize and identify different set of objects and calculate the objects pose and orientation. In Fig 5.2.2.3 it shows the result from the real simulator.



Fig 5.2.2.3

## 5.3 Path Planning

A short introduction to the principles of sampling-based motion planning is first given. Robotic motion planning seeks to find a solution to the problem of “Go from the start to the goal while respecting all of the robot’s constraints." From a computational point of view, however, such an inquiry can be very difficult when the robot has a large number of degrees of freedom. For simplicity, consider the classical motion planning problem known as the piano mover’s problem. In this formulation, there exists a rigid object in 3D (the piano), as well as a set of known obstacles. The goal of the piano mover’s problem is to find a collision-free path for the piano that begins at its starting position and ends at a prescribed goal configuration. Computing the exact solution to this problem is very difficult. In this setup, the piano has six degrees of freedom: three for movement in the coordinate planes (x,y,z), and three more to represent rotation along the axes of these coordinate planes (roll, pitch, yaw). To solve the piano mover’s problem we must compute a set of continuous changes in all six of these values in order to navigate the piano from its starting configuration to the goal configuration while avoiding obstacles in the environment. It has been shown that finding a solution for the piano mover’s problem is PSPACE-hard, indicating computational intractability in the degrees of freedom of the robot [1, 2, 3].

### 5.3.1 Sampling-based Motion Planning

Sampling-based motion planning is a powerful concept that employs sampling of the state space of the robot in order to quickly and effectively answer planning queries, especially for systems with differential constraints or those with many degrees of freedom. Traditional approaches to solving these particular problems may take a very long time due to motion constraints or the size of the state space. Sampling arises out of the need to quickly cover a potentially large and complex state space to connect a start and goal configuration along a feasible and valid path. The need to reason over the entire continuous state space causes traditional approaches to breakdown in high-dimensional spaces.

In contrast, sampling-based motion planning reasons over a finite set of configurations in the state space. The sampling process itself computes in the simplest case a generally uniform set of random robot configurations, and connects these samples via collision free paths that respect the motion constraints of the robot to answer the query. Most sampling-based methods provide probabilistic completeness. This means that if a solution exists, the probability of finding a solution converges to one as the number of samples reasoned over increases to infinity. Sampling-based approaches cannot recognize a problem with no solution.

The problem to be solved by the motion query using a sampling-based method, and define some useful terminology that will be used throughout the remainder of the primer.

**Workspace**: The physical space that the robot operates in. It is assumed that the boundary of the workspace represents an obstacle for the robot.

**State space**: The parameter space for the robot. This space represents all possible configurations of the robot in the workspace. A single point in the state space is a state.

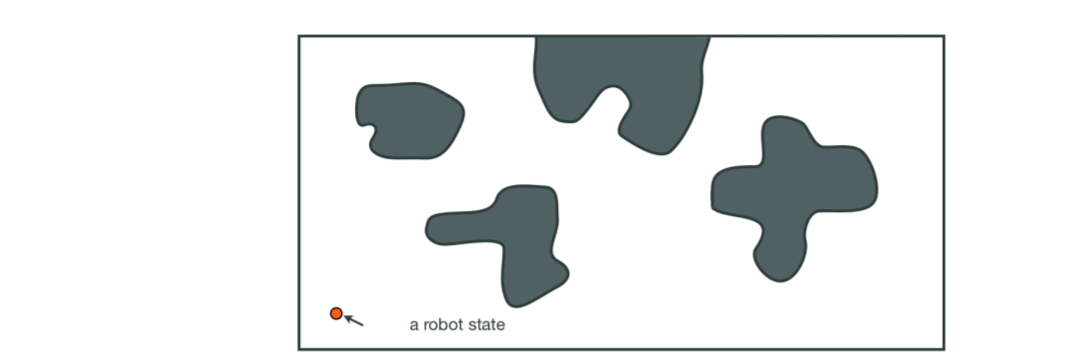
**Free state space**: A subset of the state space in which each state corresponds to an obstacle free configuration of the robot embedded in the workspace.

**Path**: A continuous mapping of states in the state space. A path is collision free if each element of the path is an element of the free state space. From these definitions, the goal of a sampling-based motion planning query can be formalized as the task of finding a collision path in the state space of the robot from a distinct start state to a specific goal state, utilizing a path composed of configurations connected by collision free paths.

In the remainder of this section will discuss two types of sampling-based planners, from which many of the state-of-the-art techniques can be derived from. It should be noted that many sampling strategies exists, but their presentation is beyond the scope of this document.

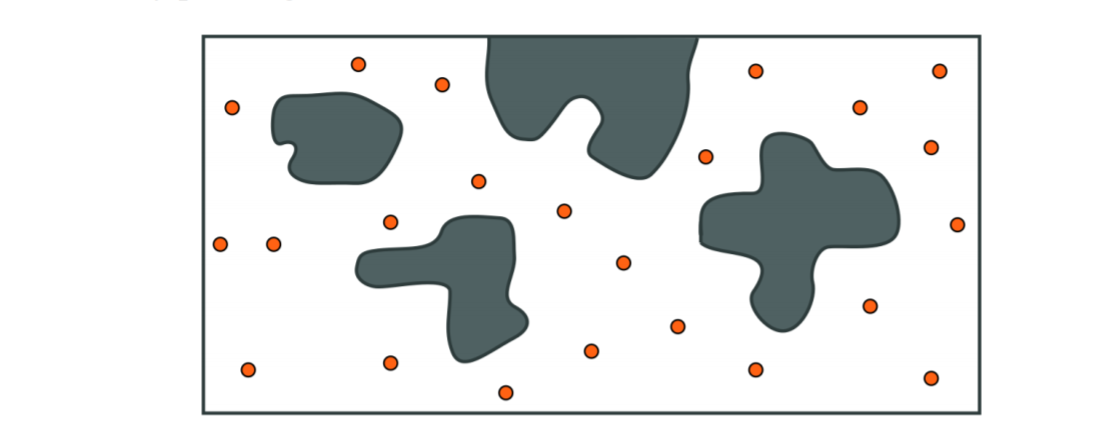
#### 5.3.1.1 Probabilistic Roadmap

The probabilistic roadmap (PRM) is among the first sampling-based motion planners [4]. This approach utilizes random sampling of the state space to build a roadmap of the free state space. This roadmap is analogous to a street map of a city. To illustrate the fundamentals of the PRM, a simple example of a 2D workspace and freely moving point robot will be used.

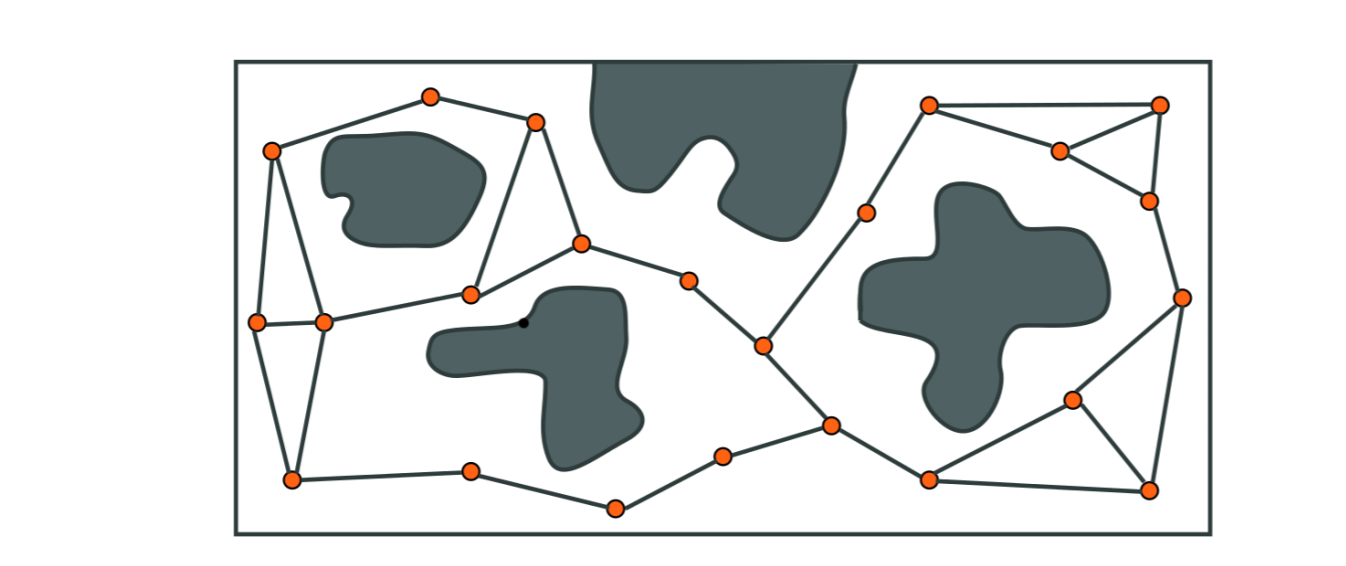


**Fig** 5.3.1

Consider the workspace in Figure 5.3.1 This image shows a bounded workspace for the point robot in which the shaded regions are obstacles. One particular state for the robot is highlighted. The PRM works by uniformly sampling the free state space and making connections between the samples to form a roadmap of the free state space. The roadmap can be stored efficiently as a graph data structure where the random samples compose the vertices, as in Figure 5.3.2 It should be noted that the free state space is almost never explicitly known in sampling-based methods. Each sample that is generated is checked for collision, and only collision free samples are retained. Additionally, there are many different ways to sample the free state space, and changing the sampling strategy is beneficial in many planning instances.

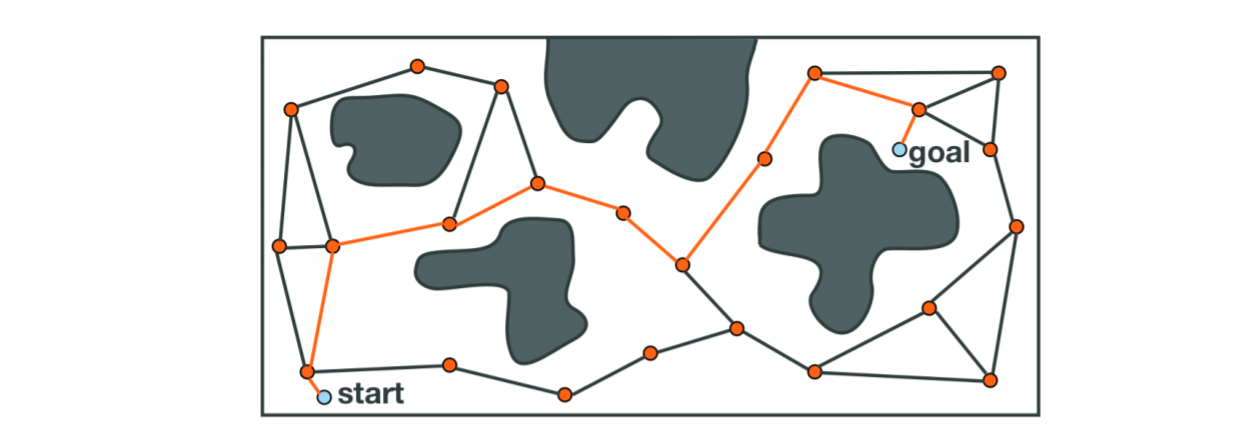


**Fig** 5.3.2

Once the desired number of free samples have been found, the roadmap itself can be constructed by connecting the random samples to form edges. The canonical PRM attempts to connect each sample to the k samples nearest to it by using a local planner that is tasked with finding short collision free paths. The local planner finds this path by interpolating the motion of the robot between the two samples, checking for collisions at some prescribed resolution. If no configuration of the robot between the samples collides with an obstacle, then an edge is inserted to the roadmap. Figure 5.3.3 shows a complete probabilistic roadmap in the 2D workspace example. Once the roadmap is complete, it can be used to answer motion planning queries by connecting the start and goal states to the roadmap using the local planner, and performing a graph search to

**Fig** 5.3.3

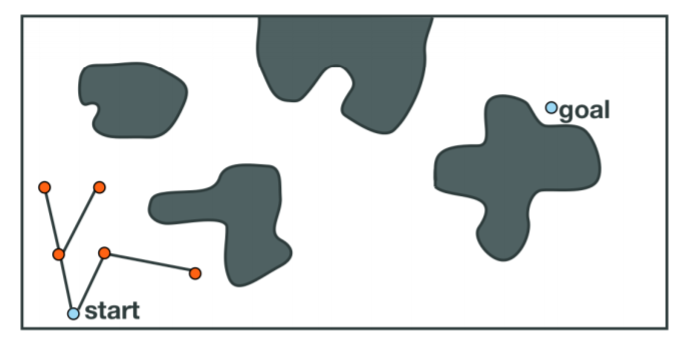
The find the shortest path in the roadmap. This is seen in Figure 5.3.4.



**Fig** 5.3.4

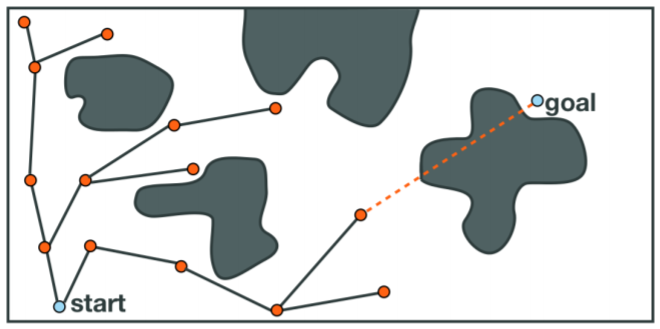
#### 5.3.1.2 Tree-based Planners

There exist many types of sampling-based planners that create tree structures of the free state space. The trees generated by these methods are analogous to the probabilistic roadmap, except that the structure contains no cycles. Due to the wide variety of tree-based planners (e.g., RRT[5], EST[6], SBL[7], KPIECE[8]), one specific type will not be discussed in detail here. However, a general framework will be described. These methods begin by rooting a tree at the starting configuration of the robot. With the first node of the tree intact, random sampling of the free space then occurs. The planner employs an expansion heuristic, which typically gives the method its name, from which the sample is connected to the tree along a collision free path. Figure 5.3..5 shows an example in the 2D workspace scenario where the first few valid samples are connected to the tree. Since it is highly improbable that the sampling process will ever sample the goal state exactly, the methods often bias the expansion of the tree toward the goal state. If it is possible to connect the



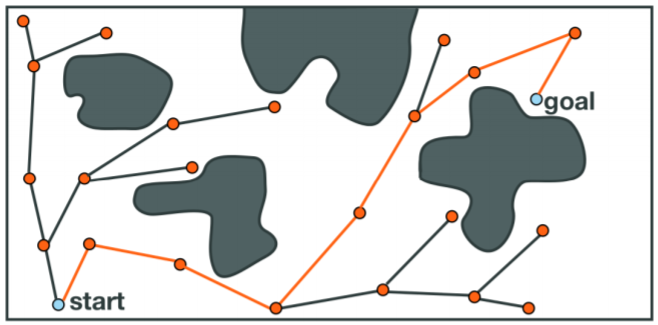
**Fig** 5.3.5

goal to the existing tree, then the search is complete; a path through the free state space has been found from the start to the goal. Figure 5.3.6 shows a case where the goal cannot be connected to the tree, and Figure 5.3.7 shows the case where the goal is connected, terminating the search.



**Fig** 5.3.6

It is important to highlight the difference between the roadmap-based planners and the tree-based planners. The tree-based techniques are most suitable for single-query planning. These trees do not normally cover the free space in the same manner that a roadmap would. However, when planning with differential constraints, it is not easy to encode control information into an undirected edge. Controls are usually directed commands, and require a specific pre-condition in order for a particular control to be valid. Tree-based methods, on the other hand, excel at planning with complex dynamics because of the directed, acyclic nature of the underlying data structure. Control information can be encoded for each edge of the tree, with the vertices of the tree satisfying the prerequisites for the valid controls. Finally, it should be noted that many sampling-based approaches require a smaller memory footprint than other motion planners. The compactness of these planners stems from the sampling process itself, as well as the fact that no explicit representation of the state space is needed in order to solve the problem. Storage and search of the underlying data structure (e.g., graph, tree) should



**Fig** 5.3.7

be efficient to fully maximize the quality of these methods. However, as tree planners are used for increasingly difficult problems, memory requirements may increase in order to keep information that guides the expansion of the search.

### 5.3.2 Primitives of Sampling-based Planning

Sampling-based planning is a very powerful tool for planning in high-dimensional spaces or for system with complex dynamics. There exists many kinds of sampling-based motion planners, with many commonalities, but the method in which one samples the state space is key to computing a solution.

Collision checking is a very important part of sampling-based planning. It is used not only in the local planner when attempting to find collision free paths between samples, but also during the sampling process itself. In a complex or high-dimensional system it may not be easy to explicitly represent the free state space, but in sampling-based methods it is not necessary to create this space. It is the job of the collision checker to accept a configuration of the robot and quickly determine whether or not this state is in collision.

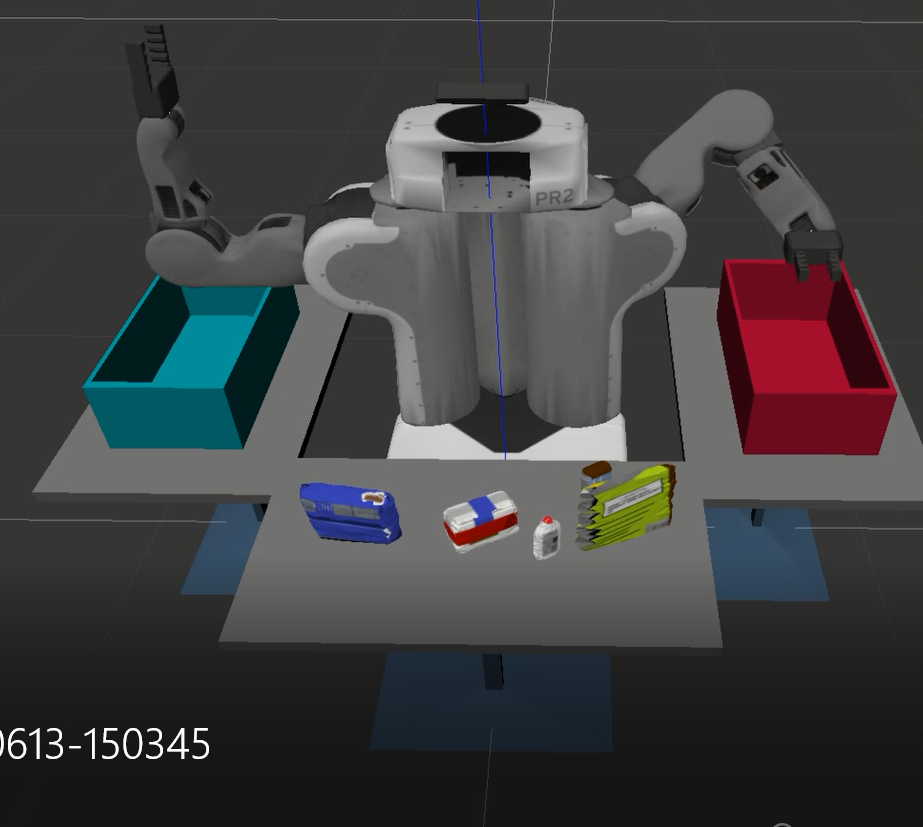
Nearest neighbor searching is another cornerstone of sampling-based methods. It is from the ability of determining whether two states of the robot are close that many of the common approaches are able to effectively find paths through a high-dimensional space. Distances, however, are not easy to compute in non-Euclidean spaces where many of the interesting problems reside. Kd-trees offer one way to perform this search, but the optimal connection strategy for samples remains elusive.

## 5.4 Kinematics

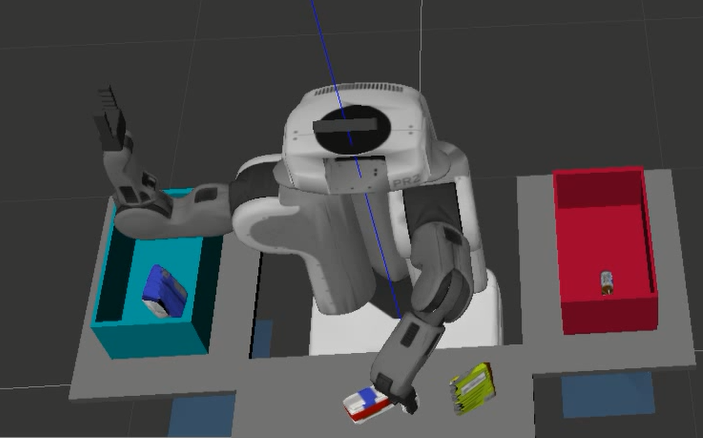
# Chapter 6: Debugging and Testing

## 6.1 Common problems

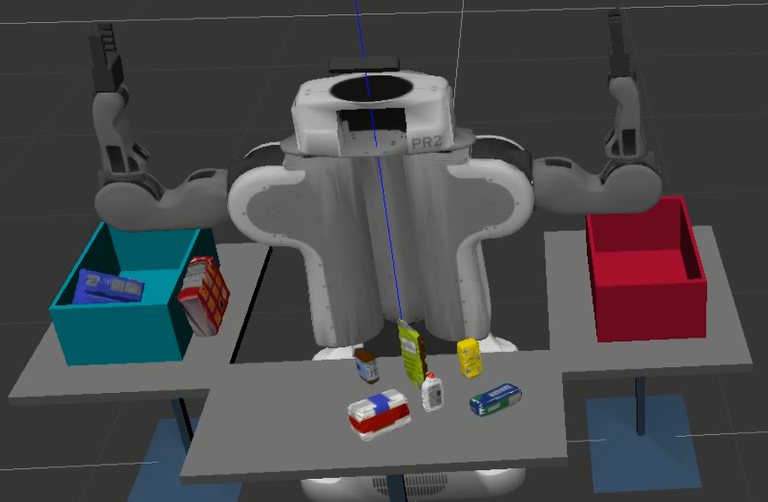
In this section i am going to discuss the problems that affected the Pr2 accuracy and the common problems that we can’t fix but could decrease its occurrence, how we fixed them and what we did plan to do as future work.



This was the first problem that our robot have which was that when the robot pick an item the gripper fail to attach the item despite all the right calculations and actions the items didn’t get picked, After searching we fixed this issue by adding friction value for the items. And That error got fixed

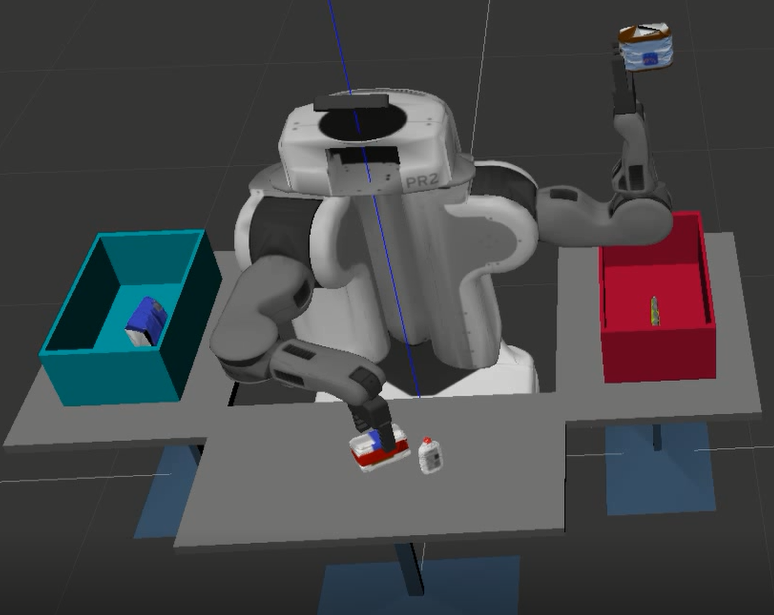


The second problem was that the robot sometimes hit the object while gripping it despite that the centroid of the item was calculated correctly, After searching for the causes of this problem we found that this was error accumulated by the robot and we tried to fix it by lowering the speed of the robot which helped to decrease the accumulated error.



Here the problem was that sometimes the box turn transparent making objects fall from it. This problem after searching was a performance issue from Gazebo that sometimes Gazebo drops the collision value of an object for a split of a second making any object pass through it.

we couldn’t find any proper solution for that problem so we couldn’t really mark that problem as fixed but what we did was to apply some optimizations and to not make all tasks calculated at ones to lower the computations so that the environment don’t go wild!



We noticed that some objects when the robot tries to drop it in the box it sticks to the gripper even after opening, fixing that problem was to add the appropriate slipping value to each object so that it would drop after opening the gripper and at the same time the object doesn’t slip while moving the arm.

## 6.2 Future work

For performance issues we needed more computational power but as it costs a lot we planned to use several laptops as nodes to help in improving the performance and it was marked as future work as we need to search for more papers that talks about this task before going for implementing this idea.

# References

* Vision Guided Robot System for Picking Objects By Casting Shadows
* “onVision Guided Pick and Place Robotic Arm System Based on SIFT” paper
* Programming Robots with ROS, Morgan Quigley, Brian Gerkey & William D. Smart
* <https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC>
* <https://en.wikipedia.org/wiki/Random_sample_consensus>
* <https://github.com/strawlab/python-pcl/blob/master/examples/statistical_outlier_fiter.py>
* <https://www.naftaliharris.com/blog/visualizing-k-means-clustering/>
* <http://pointclouds.org/documentation/tutorials/kdtree_search.php>
* <https://www.udacity.com/course/robotics-software-engineer--nd209>
* Open Motion Planning Library: A Primer Kavraki Lab Rice University