Copyright

by

Brian Erick O'Neil

2013

The Dissertation Committee for Brian Erick O'Neil Certifies that this is the approved version of the following dissertation:

Object Recognition and Pose Estimation for Manipulation in Nuclear Materials Handling Applications

Committee:

Sheldon Landsberger, Supervisor

Mitchell Pryor, Co-Supervisor

J.K. Aggarwal

Ashish Deshpande

Luis Sentis

Object Recognition and Pose Estimation for Nuclear Manipulation in Nuclear Materials Handling Applicaions

by

Brian Erick O'Neil, B.S.; B.S.E.; M.S.E.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2013

Dedication

To my family: Emily, Henry, and Ellen

Acknowledgements

I would like to thank my wife, Emily whose patience and support made this possible. I would also like to thank Dr. Mitch Pryor for his guidance and wisdom during the writing process as well as Dr. Sheldon Landsberger, who works tirelessly to ensure the success of his students.

This material is based upon work supported under a Department of Energy Nuclear Energy University Programs Graduate Fellowship.

Object Recognition and Pose Estimation for Manipulation in Nuclear Materials Handling Applications

Brian Erick O'Neil, Ph.D.

The University of Texas at Austin, 2013

Supervisor: Sheldon Landsberger

Co-supervisor: Mitchell Pryor

This dissertation advances the capability of autonomous or semiautonomous robotic manipulation systems by providing the tools required to turn depth sensor measurements into a meaningful representation of the objects present in the robot's environment. This process happens in two steps. First, the points from depth imagery are separated into clusters representing individual objects by a Euclidean clustering scheme. Each cluster is then passed to a recognition algorithm that determines what it is, and where it is. This information allows the robot to determine a pose of the object for grasp planning or obstacle avoidance.

To accomplish this, the recognition system must extract mathematical representation of each point cluster. To this end, this dissertation presents a new feature descriptor, the Cylindrical Projection Histogram which captures the shape, size, and viewpoint of the object while maintaining invariance to image scale. These features are used to train a classifier which can then determine the label and pose of each cluster identified in a scene. The results are used to inform a probabilistic model of the object, that quantifies uncertainty and allows Bayesian update of the object's label and position.

Experimental results on live data show a 97.2% correct recognition rate for a classifier based on the Cylindrical Projection Histogram. This is a significant improvement over another state-of-the art feature that gives an 89.6% recognition rate on the same object set. With statistical filtering over 10 frames, the raw recognition rate improve to 100% and 92.3% respectively. For pose estimation, both features offe rrotational pose estimation performance from 12° to 30°, and pose errors below 1 cm.

This work supports deployment of robotic manipulation systems in unstructured glovebox environments in US Department of Energy facilities. The recognition performance of the CPH classifier is adequate for this purpose. The pose estimation performance is sufficient for gross pick-and-place tasks of simple objects, but not sufficient for dexterous manipulation. However, the pose estimation, along with the probabilistic model, support post-recognition pose refinement techniques.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Since the dawn of the industrial revolution, machines have increasingly replaced or augmented human labor in almost every conceivable physical task. Humans put down their shovels and pick axes and mounted steam shovels and excavators. Needles and thread gave way to sewing machines. Engineers designed and built automated mechanical systems to mass produce everything from lug nuts to light bulbs. Advances in mechanical system design and control engineering have made product manufacturing cheaper, faster, and of higher quality than ever before. While machines rapidly replaced human legs, arms, hands, and fingers, none of them could replace the human brain. Each machine required a human operator to do the thinking.

By the time the first industrial robot began service in 1962, the idea of machines that think for themselves had already become a staple of science fiction. But for the first time, advances in science and engineering held the promise that thinking machines were just beyond the horizon. In the 1960's, computer scientists started actively pursing the idea of Artificial Intelligence (AI). Computers learned to play (and win) chess games against humans. HAL debuted in *2001: A Space Odyssey*. Rosie the robot did all the cooking and cleaning for the Jetsons. The coupling of machine intelligence to mechanical systems held the promise of relieving human beings of the tedium of work that was dull, dirty or dangerous.

Now, decades later, that promise has gone unfulfilled. We don't have robot maids cleaning up after us. We still have to drive our own cars, do our own laundry, and (thankfully) open our own pod bay doors. In fact, despite continual strides in machine learning, machine perception, and machine intelligence, the promise of a machine that can reason and move about in its environment like a human seems as near to the horizon

now as it did five decades ago. While the promise of fully autonomous robots remains unfulfilled, we can leverage advanced autonomous technologies to improve robotic systems if we allow the intelligence of a human operator to remain in the loop. But to do so we must accept that our robots will not be like what science fiction has suggested. We must think about how to integrate machine intelligence with human intelligence to make the take advantage of the relative strengths of each.

1.1 BACKGROUND AND MOTIVATION

Improving robotic system capability is not just about bringing science fiction to life or avoiding house chores. It is about protecting and preserving human life by relieving humans of hazardous tasks. Robots can respond to disaster situations by collecting data and locating survivors in locations that humans cannot access either because of physical limitations or severe hazards. Robots can be used to inspect and disarm bombs and improvised explosive devices. Robots can handle toxic or radiologically hazardous materials. In fact, robots are used in many of these scenarios. But there were no robots on hand to inspect the Fukushima reactors after a tsunami disabled cooling systems causing a fuel meltdown. The emergency response was delayed by the delivery of foreign robots and subsequent training on their operation. This is in the same country that brought us ASIMO, a bipedal robot that can autonomously perceive and navigate its environment. Soldiers deploy robots to disarm IED's, but they teleoperate the robot through the entire procedure, hoping they can finish before someone starts shooting at them. Undoubtedly they've seen videos on the internet of robots in research labs perceiving and manipulating objects not unlike roadside bombs with better speed and dexterity than their teleoperated systems provide. Workers at Los Alamos National Laboratory handle toxic and radioactive materials through rubber gloves inside

gloveboxes. Gloves do occasionally tear in these facilities, unnecessarily exposing workers to significant radiological and toxic hazards. There are also ergonomic injuries due to the awkwardness of trying to work though fixed gloveports. Meanwhile, in another facility at the same lab, robots are doing all of the work in gloveboxes.

This disconnect between what robots *can* do and what they *actually* do is that despite all of the technological advances, machine intelligence alone is not robust enough to be adopted in most robotic applications outside of the research lab. Robots do well on the factory floor where all of the uncertainty in the environment and the task can be designed away through sound mechanical and industrial engineering. But post-accident reactor investigation, rendering safe an IED, and working in a glovebox designed for humans are applications where the uncertainty cannot be avoided, and this prevents robots from acting autonomously in these environments. However, this document rejects the notion that autonomy is an either/or proposition. Rather, there exists a spectrum of autonomous robot behavior between pure teleoperation and complete autonomy.

1.1.1 Transitional levels of autonomy

The idea of robots exhibiting behavior somewhere on an autonomy spectrum is not new. This behavior is often called semi-autonomous behavior. The level of autonomy at which a robot can operate is closely correlated with the level of uncertainty in the environment and the task specification. Environmental uncertainty refers to uncertainty in the position of the robot within its environment or uncertainty of the positions of other objects and agents in the environment. Task uncertainty describes the level of uncertainty in the motions that the robot must execute. As the level of structure in the task or the environment decreases, autonomous behaviors become more difficult and less robust. But uncertainty is not a static quantity. As the state of the environment changes, so does the

uncertainty, and the level of autonomous initiative that the robot exhibits must change as well. Figure 1-1 shows a qualitative depiction of where certain tasks lie in environment/task uncertainty space.

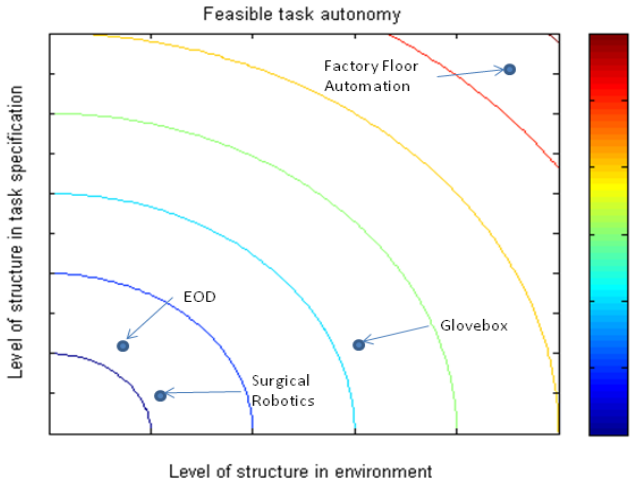


Figure 1-1. Effect of environmental and task uncertainty on feasible task autonomy.

Idaho National Laboratories built the idea of an autonomy spectrum into their mobile robotics suite, the Robot Intelligence Kernel (RIK) [Bruemmer, 2002]. They describe five regimes of dynamic autonomy shown in Table 1-1. The separation of autonomous behaviors into 5 regimes clearly illustrates the different levels of autonomy as well as the advantages of using partial autonomy even when full autonomy is not feasible. At the lowest level, the operator is responsible for motivating all robot motions. The robot's control system does not augment the human's intelligence in any way. In safe mode, the robot monitors its proximity to other objects and stops the motion if a collision becomes imminent. In Shared mode, the robot will execute its own motions to assist the operator in teleoperated tasks. Examples would be obstacle avoidance or reactive path planning. In collaborative tasking mode, high-level tasks are still commanded and supervised by the operator. For example, the user requests that an object be moved, and

the robot moves it without further input, assuming all of the collision detection, motion planning, and safety functions.

Table 1-1. INL's dynamic autonomy modes [Bruemmer, 2002].

Mode	Defines Task Goals	Supervises Direction	Motivates Motion	Prevents Collision
Teleoperation	Operator	Operator	Operator	Operator
Safe Mode	Operator	Operator	Operator	Robot
Shared Mode	Operator	Operator	Robot	Robot
Collaborative Tasking Mode	Operator	Robot	Robot	Robot
Autonomous Mode	Robot	Robot	Robot	Robot

At full autonomy, the robot will determine what tasks need to be completed and execute them. This is only practical in very limited applications. An important aspect of the RIK autonomy levels is that the robot does not operate at a fixed level, but can move between autonomy regimes as changes occur in the task or the environment (hence *dynamic* autonomy).

RIK is first and foremost a framework for mobile robotics. The dynamic autonomy levels in RIK concisely demonstrate the advantages of multiple levels of autonomy, but are tailored to the needs of mobile robotics. The University of Texas Nuclear Robotics Group (UT-NRG) uses its own dynamic autonomy scheme that is a bit more esoteric, but better meets the needs of unstructured manipulation. Like the RIK levels, UT-NRG's transitional levels of autonomy are dynamic. The manipulator can move between levels during operation. Table 1-2 shows the seven levels of the UT-NRG transitional levels of autonomy framework. The levels are defined without regard to the current state of technology. In practice, the maximum level of autonomy is dictated by the current state of the art in machine perception and intelligence. The current state of the art permits operation at level 5 if the uncertainty in the task and environment can be kept

low. Levels above 5 require that the robot react in uncertain situations, so robust operation above 5 is much more difficult to achieve.

Table 1-2. UT-NRG's transitional levels of autonomy.

Autonomy Level	System behavior
0	Joint space teleoperation
1	End effector frame teleoperation
2	Teleoperation with collision detection and/or avoidance
3	Plans and executes motions to locations of interest.
4	Plans and execute object grasps.
5	Completes high-level tasks involving multiple sub-tasks
6	Responds appropriately to non-operator external events. (oven timer, abnormal system operation, low battery)
7	Anticipates and completes future tasks.

The continuum of behaviors between zero autonomy and full autonomy provides the foundation for shared initiative between human and robot. Human/robot collaboration under this system does not deal with the manner in which a human interacts with a robotic system. Rather, it considers the human and robot part of the same system. On a factory floor, you may see humans performing some physical tasks while robots perform others. The physical labor is divided between human and robot according to what each does well. Dynamic autonomy does the same at the supervisory level. Humans are very good at planning and reacting in uncertain environments whereas robot controllers have more difficulty. As uncertainty increases, dynamic autonomy allows the robot control system to transfer more control functions to the human.

1.1.2 Robotics in nuclear materials handling

An integral part of the safety culture among nuclear professionals is the ALARA principle. ALARA states that radiation dose to humans should be kept As Low As

Reasonably Achievable (ALARA). In fact, in addition to hard annual dose limits on radiation workers, ALARA is a Nuclear Regulatory Comission (NRC) requirement under 10 CFR Part 20 which defines ALARA as "making every reasonable effort to maintain exposures to radiation as far below the dose limits in this part as is practical consistent with the purpose for which the licensed activity is undertaken, taking into account the state of technology, the economics of improvements in relation to state of technology, the economics of improvements in relation to benefits to the public health and safety, and other societal and socioeconomic considerations, and in relation to utilization of nuclear energy and licensed materials in the public interest." An important consequence of this requirement is that if a robotic system can feasibly and economically reduce human dose, its use becomes compulsory under NRC regulation.

It comes as no surprise then, that the nuclear industry has made extensive use of robotics throughout its history. The first teleoperated robots were developed for nuclear materials handling when the first nuclear weapons were being developed. They were directly coupled mechanical master/slave systems that bear little resemblance to modern industrial manipulators or electromechanical teleoperated systems. Figure 1-2 (a) shows an operator using one of these systems. These systems allowed operators to safely handle materials from an adjacent room, shielded from the radiation hazard. More advanced teleoperated systems like the one shown in Figure 1-2 (b) are used in highly radioactive "hot cell" environments for unstructured manipulation tasks.

One of the most ambitious robotic systems in the Department Of Energy (DOE) comples is the Advanced Recovery and Integrated Extraction System (ARIES) line at Los Alamos National Laboratory (LANL). . The ARIES system was created to demonstrate the feasibility of an integrated process for dismantling nuclear weapons, reclaiming the plutonium and converting it to a form suitable to disposition and long-term storage. A

key aspect of the technology developed under ARIES has been significant automation of many glovebox functions traditionally performed by humans. ARIES is therefore a unique example of robotic glovebox automation. Figure 1-2 (c) shows the Robotic Integrated Packaging System (RIPS) component of ARIES [Coonley, 2008].

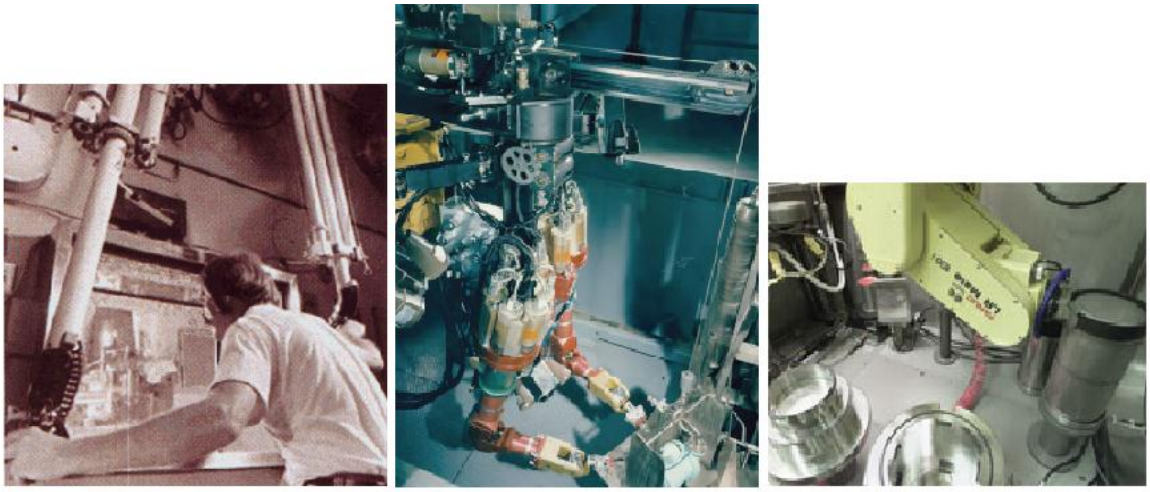


Figure 1-2. Robots for nuclear materials handling. (a) Mechanical master/slave robot. (b) Hot cell teleoperation system (c) RIPS on ARIES line.

A noticeable trend in robotic systems for nuclear materials handling is that they take an either/or approach to autonomy. For unstructured manipulation, telerobotic systems are preferred. Autonomous robotic systems are only used when the process can be tightly controlled thorough process design and appropriate fixturing as seen in Figure 1-2 (c). All three robotic components of the ARIES line are fully autonomous systems in which the uncertainty usually encountered in a glovebox has been designed out of the system. Because most glovebox facilities have not been designed and built with automation in mind, the environment cannot be controlled tightly enough to deploy ARIES-like systems. Most gloveboxes were designed solely for human use. For robots to

augment human performance in these environments, they must be able to operate in the presence of dynamic and uncertain environments.

As the state of the art in robot perception and machine intelligence advances, deployment of robot manipulators in gloveboxes designed for humans becomes more feasible both technically and economically. At some point, technical advances in robotics move them into the domain of ALARA where they can and must be used to reduce radiation dose to workers. Additional benefits from the use of robots to augment human performance in nuclear facilities include reduced worker fatigue and reduced incidence of ergonomic injury. The work presented in this document supports operation of manipulators in *uncertain* environments under *transitional autonomy* in order to deploy robots to reduce *radiological and ergonomic injury* to glovebox workers.

1.1.3 Object recognition and pose estimation

For any dynamic autonomy framework to be useful, the system must be able to switch between autonomy modes when appropriate. A primary motivation of the current work is that the main criterion that determines the appropriate level of autonomy is uncertainty. When the task and environment are deterministic, full autonomy is possible. But as the positions of important objects and collision hazards become less certain, a human must provide more guidance. For the system to choose an autonomy level, it must therefore be able to quantify the uncertainty in the environment. Therefore as sensors update the state of the environment, the control system must also track how much uncertainty is introduced to the model through noisy and imperfect sensing.

In a nuclear materials glovebox, a large source of uncertainty is in the location and types of objects present. Because the gloveboxes are used by humans, different objects may be in different locations any time that the robot is given a task to perform.

The robot's control system must therefore be able to identify the objects present in the glovebox through a sensing system. Figure 1-3 shows a generic control system framework for a manipulator control system.

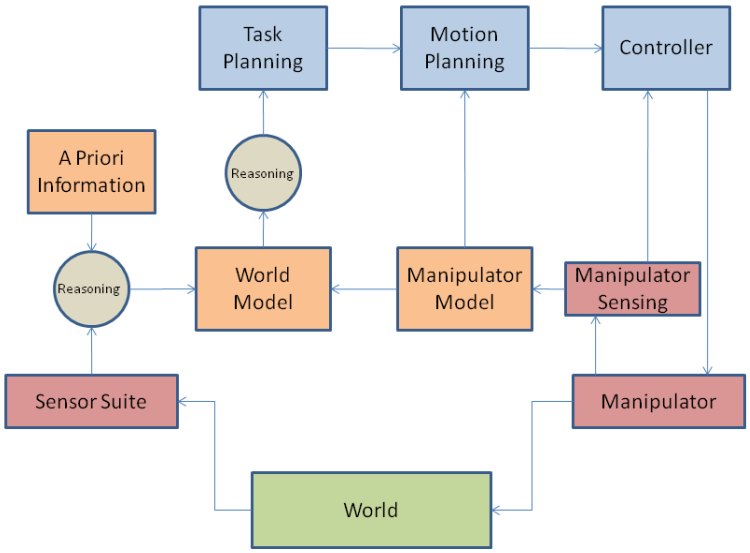


Figure 1-3. Manipulator control system diagram.

The current work involves the Sensor Suite block, the World Model block, and the reasoning block between them. The sensor used is a depth camera that provides a 3D point cloud representing the entire scene. The world model is a list of objects in the environment described by a discrete probability distribution over possible object classes and 4D Gaussian pose estimates. The reasoning block includes an object recognition and pose estimation pipeline. These three components provide key pieces of information that allow the system to run under the dynamic autonomy paradigm. They are described in detail in Chapters 3 and 4.

1.2 ROBOT OPERATING SYSTEM

Robots are incredibly complex systems. Software development for a robotic system spans several abstraction layers from low-level device driver software to high-level decision making and everything in between. This is a difficult problem in robotics research because of the sheer amount of code that must be written just to get a system to function properly before a researcher can even begin to think about making an original contribution. Robotic systems frequently consist of several pieces of hardware assembled to address a specific problem or application area. For example, in the past two years UT-NRG's laboratory has had three types of depth imager, four different grippers, four different manipulators from three different manufacurers, two force/torque sensors, and an Arudino-controlled servo table. In the near future, UT NRG plans to add prismatic actuators to the manipulator bases. These components have been assembled in various configurations and code reuse has been difficult because the system integration details change slightly over time, and inevitably the code gets entangled with the specific application. This process becomes exhausting for researchers who feel like they spend more time porting old code to new systems than actually doing research. This also makes research collaboration difficult because every lab uses its own software architecture custom built for whatever system is in their lab.

1.2.1 ROS overview

In order to meet these challenges, a group of robotics researchers at Stanford and Willow Garage developed an open-source framework for complex robotic system development which they dubbed the Robot Operating System (ROS) [Quigley, 2009]. ROS is distributed under a BSD license as free and open-source software. Since its release, ROS has rapidly secured a place as the *de facto* standard framework for advanced robotic system development. Research groups, hobbyists and enthusiasts have

contributed to the project with such enthusiasm that there is now an enormous code base that provides a convenient springboard for robotic system development.

ROS is a peer-to-peer system in which system components run as nodes on a distributed network. These nodes interact with each other through a standardized messaging system. This provides a strong separation between the interface and implementation that simplifies code reuse and collaborative system development. While well-written device APIs also separate interface from implementation, the system developer must learn the API for every individual device he wishes to use in the system. In ROS, the interface is standardized so that integration of new devices is seamless.

1.2.2 ROS concepts and nomenclature

The work presented in this document has been developed under the ROS framework, so it is appropriate to cover some basic ROS concepts and nomenclature for the unfamiliar reader. More information can be found at www.ros.org

ROS Filesystem

ROS requires that code be organized in a specific way so that various ROS tools can find files and assure that dependencies are met. ROS code is organized into *stacks* and *packages*. Stacks are collections of functionally related packages. Packages are the basic unit of code organization. A package may consist of any set of related nodes, libraries, or other data. Typically a package contains some executables that are used to spawn related ROS *nodes*. Each package contains a package *manifest* (`manifest.xml`). The manifest contains meta-data for the package such as authorship and license information, language specific complier flags, and dependency information. When a package is installed in ROS, the manifest assures that all package dependencies are installed recursively as well. A package may also contain message description

(`*.msg`) and service description (`*.srv`) files that define ROS data structures to contain messages that will be passed between nodes.

ROS Runtime environment

The ROS Runtime environment is the graph of nodes running on the system. Each node is a separate process running on the ROS system. Nodes may run individual devices such as motors and sensors or they may perform higher-level functions like perception or path planning. ROS systems are inherently distributed. The ROS graph can span any number of machines and devices on the same network. There is one special node called the ROS *Master*. The ROS Master provides name registration and a lookup server so that other nodes in the system can find each other. The Master provides only a name lookup service. Connections between nodes are made directly. Communication between nodes is facilitated by the TCPROS protocol which uses standard TCP/IP sockets.

Nodes interact with each other through *messages* and *services*. The basic communication protocol is a publisher/subscriber model. A node sends information out by publishing to a *topic*. Other nodes can then subscribe to the topic to receive this information. For example, a node that runs a depth imager will publish image information on a topic. A perception node will subscribe to that topic to receive image data to process. Any number of nodes can publish on and subscribe to the same topic. Publishing and subscribing nodes are unaware of each other's existence. Communication flows only one way: from publisher to subscriber. This model is ideal for nodes such as sensors that are continuously producing data. However, it is not well suited to request/response communication. There are some interactions in which one node makes a request of another and then expects some sort of response. For example, a node in a control system may want to retrieve a motion plan from a path planner. This would involve sending a

goal position to the motion planner node and receiving a trajectory in response. For these types of interactions, ROS uses services. A service is a synchronous interaction that has two distinct parts, a *request*, and a *response*.

Perception in ROS

To provide object recognition and pose estimation capability to a robot manipulator requires integration of a perception system and a robot control system. The ROS developer community has provided ROS compatible device drivers for many common cameras and imaging systems including the Microsoft Kinect for Xbox 360 used in the current work. Also, developers of some powerful image processing libraries provide ROS interfaces. The basic ROS installation includes both the OpenCV image processing libraries for computer vision [Bradski, 2000], as well as the Point Cloud Library (PCL) libraries for point cloud image processing [Rusu, 2011]. This makes integration of perception convenient because image acquisition and processing are done directly in ROS. Chapter 4 discusses in more detail how ROS and PCL features are leveraged in the current work.

PCL is used extensively in this work. PCL is an open-source project that provides several state-of-the-art algorithms for processing point clouds [Rusu, 2011]. PCL is comprised of several modular libraries that do low-level operations like outlier removal, all the way through more complicated operations like model fitting and segmentation. Like ROS, PCL has a large and active developer community, so it stays current with the state of the art in point cloud processing. Also like ROS, from its release it has quickly become the standard framework for algorithm development in its field. It's tight integration and distribution with ROS has established it as the first choice for point cloud processing in robotics.

ROS and PCL supports the current work by providing a large portion of the foundational software required. PCL provides many of the individual algorithmic componenets of an object recognition pipeline. ROS provides filesystem and runtime tools that greatly simplify creation of a modular pipeline that integrates easily with other robot control system components. However, neither ROS nor PCL provide a way to model objects that supports contact interaction (grasping.). Nor do they provide a complete recognition pipeline. This work uses the strengths of ROS and PCL to create object recognition and modeling packages that support unstructured manipulation.

1.3 MICROSOFT KINECT

Prior to 2010, depth image and point cloud processing research was confined to those willing to spend large sums of money to acquire the images. Integrated stereo systems were available for around $1200 [Point Grey, 2012]. 2D LIDAR systems were several thousand dollars [SICK, 2010], 3D LIDAR systems can cost as much as $75,000 [Velodyne, 2012]. The best infrared time-of-flight cameras cost around $9000 [Mesa Imaging, 2012]. The high cost restricted these systems to applications where 3D imagery was absolutely essential. The LIDAR systems, for example were well represented in the DARPA Grand Challenge and Urban Challenge vehicles. Stereo vision systems were preferred as a low-cost option for most robotics applications.

Then, in November 2010, Microsoft released the Kinect for XBOX 360 Video game controller, shown in Figure 1-5. The Kinect has both an rgb digital video camera and a structured infrared depth camera. Best of all, the Kinect's retail price is $149.99.

Figure 1-4. Microsoft Kinect for XBOX 360.

The device was developed to track human motion so that game players could interact with game software without a traditional hardware controller. However, the robotics community immediately seized upon the device's potential as a vision sensor for robotics. AdaFruit industries offered a bounty of $3000 to the first person to release open-source drivers for the Kinect. Kinect was launched on November 4th, 2010, and the winner of the bounty was announced on November 10th, 2010. In December, PrimeSense technologies, the developer behind Kinect's depth sensing technology released their own open-sournce drivers through OpenNI. At $150, and with open-source drivers readily available, a vibrant user and developer community sprouted almost overnight [AdaFruit, 2012].

ROS was released in 2009. Kinect was released in 2010, and within days was accessible to the robotics research and hobbyist communities. PCL was released early in 2011. These three technologies together transformed robot perception in a very short time frame. All of these technologies provided open-source software and actively nurtured an enthusiastic developer community. The result is a fertile development environment for robotic perception and intelligence. Through a framework for transitional autonomy, the current research brings the fruits of this rapidly advancing technology into applications that require reliable and robust operation.

1.4 SCOPE AND OBJECTIVE OF RESEARCH

This research provides a complete object recognition and pose estimation system that supports unstructured robotic manipulation. The target application is robotic manipulation in gloveboxes for nuclear materials handling. This application provides unique challenges to machine perception that motivate this research. These challenges are a result of the nature of the objects found in gloveboxes, a few examples of which are shown in Figure 1-5.

Figure 1-5. Examples of objects found in a glovebox

The objects in Figure 1-5 have very little visual or geometric texture to use for recognition. Furthermore, they are moderately reflective, which presents challenges for depth imagers. In addition, several objects are of very similar shape, but slightly different dimensions. Therefore, the current research approaches the perception problem under the following assumptions and limitations:

- The objects have little surface or visual texture.
- The objects may be lightly reflective such as matte-finished stainless steel.
- We wish to differentiate between objects of similar shape but different size.
- The environment is uncluttered; objects can be readily segmented by Euclidean clustering or temporal back-ground subtraction.
- We desire real time performance.

The assumption of light clutter is reasonable in a glovebox because the number and type of objects in the glovebox is well controlled. The desire for real time performance is dictated by the need to operate under transitional autonomy. The robot must operate at human processing speeds to assure fast and safe transitions when environmental conditions prevent the robot from operating at higher levels of autonomy.

The objective of this research is the development of a comprehensive visual recognition and modeling system robust enough for use in unstructured industrial environments. Part of this robustness is achieved through shared initiative. That is, if the system is likely to fail under autonomy, more control initiative is transferred to a human operator. While this research is directed at the problem of manipulation in gloveboxes, the methods presented are applicable in any application that meets the limitations and assumptions described above. This includes many household and service robotics tasks, as well as unstructured industrial tasks such as sorting and palletizing.

1.4.3 Original contributions

This work makes three original contributions to the field of robot vision, enumerated here, and discussed in detail in the remainder of this document.

The Cylindrical Projection Histogram

The most significant contribution is a novel point cloud feature descriptor for 3D cluster recognition. This feature, the Cylindrical Projection Histogram (CPH), provides state-of-the-art recognition performance even under challenging conditions of low texture and moderate specularity. In addition, the feature provides substantial reduction in computational cost, and improved robustness to image noise over similar feature descriptors that rely on surface normal estimation.

Probabilistic object model

A probabilistic model of an object's label and pose is presented that allows multiple recognition results to be filtered to improve performance. In addition, this allows the uncertainty to be quantified and used to inform a dynamic shared human/robot initiative framework.

LANL glovebox object dataset

As part of this work, a new pose-annotated point cluster dataset has been produced and made available for testing of visual recognition and pose estimation algorithms. The objects in the dataset provide a challenging benchmark, and this is one of very few available multi-view datasets that includes the relative pose of the object to the camera.

1.5 ORGANIZATION OF THIS DOCUMENT

This chapter introduced the problem of visual object recognition and pose estimation, and discussed the motivating application behind the current research. In addition, it has briefly discussed some of the current technologies behind the recent rapid proliferation of work in the area of robot perception from 3D depth data. The remainder of this document is organized as follows:

Chapter 2 provides a review of previous work in modeling and recognition. This review serves two purposes:

- To provide a high-level overview of the current state of the art in modeling and recognition. This allows the reader to understand where the current work fits in with respect to the taxonomy of recognition systems.
- To provide detailed descriptions of work that is directly related to the current research presented in this document. This allows the reader to understand work

upon which the presented methods rely, and also provides a basis of comparison that allows the reader to understand the similarities and differences between the current work and other recent work in the field.

Chapter 3 presents the methods used in this work. This chapter includes complete mathematical descriptions of the CPH feature, and probabilistic object model. This chapter contains the bulk of the original contributions discussed above.

Chapter 4 discusses the system implementation details. This chapter describes how the methods presented in Chapter 3 are used in conjunction with other hardware and software tools to provide a functional glovebox recognition system. This includes ROS/PCL system integration.

Chapter 5 presents the experiments performed to validate the methods presented in Chapter 3. These experiments are performed on two different datasets using both pre-collected data as well as noisy "live" data. Identical results are provided for the nearest analogous feature descriptor implemented in the most recent stable version of PCL.

Chapter 6 closes the proverbial loop, and demonstrates the use of the CPH-based recognition system in relevant applications. One application demonstration is an automated object sorting demonstration presented live at Automate 2013 in Chicago, IL. The other shows a user interface that displays relevant information in real time to the user.

Finally, **Chapter 7** provides a concise summary and review of this document, and suggests several avenues for continuing research.

Chapter 2: Related Work

This document proposes methods that allow a robot to perceive and model objects in its environment. This chapter broadly reviews literature relevant to both world modeling and visual perception and also describes previous approaches to modeling and perception within the Nuclear Robotics Group (NRG) at UT. The literature review and discussion of related work at NRG provide context for how the contributions of the current work advance the state of the art in perception within NRG and the broader research community.

2.1 MODELING

The current state of the art in world modeling for robots is the result of several decades of research. This chapter reviews relevant literature and separates modeling techniques into two classes that are discussed in the following subsections.

1. Deterministic models are those where the state of the environment is described by some set of finite values. Such models are common in robot manipulator control.
2. Probabilistic models are those where the state of the environment is represented as some set of probability distributions. Such models are more common in mobile robotics, and have been a very active research area in recent years.

2.1.1 Deterministic modeling

The deterministic approach to modeling is natural and intuitive, and as such, early approaches to modeling for robotics have favored this approach. These approaches are motivated by the generalized mover's problem: given an arbitrary robot arm and obstacle set, compute a collision-free trajectory between two poses. [Reif, 1979, Canny, 1987] In

these early approaches, solid bodies were modeled as discrete polyhedra in space. Several seminal works in manipulator motion planning use the deterministic modeling paradigm. It is interesting to note the solution to the generalized mover's problem proposed by Reif acknowledged the importance of uncertainty by inclusion of a factor τ that held a discrete uncertainty value. At each intermediate transformation, no vertex of the mover is permitted to move more than this factor, and the mover is considered in collision if would collide after expanding its dimensions by τ.

Another approach to modeling is to dispense with an internal model altogether. Under this approach, sensor measurements and control laws interact to produce robot behaviors without reference to a model. [Brooks, 1985] This approach is often called behavior-based robotics as it attempts to map sensor measurements and relatively simple control laws directly to actuator commands in order to produce apparently complex behaviors. This approach led to the commercial success of iRobot's Roomba robot vacuum cleaners.

Brooks [1985] demonstrated how behavior-based robotics can be used to produce a layered control scheme that can operate at several layers of competence. These layers of competence start with very low-level tasks such as obstacle avoidance and increase in complexity to include object identification and task planning. This idea is similar to our notion of shared human/robot initiative, and reflects the idea that a robot control system should be able to operate across a continuum of autonomous behaviors. However, the network of sensors and control laws capable of producing high-level behaviors gets increasingly complex, and the behavior-based paradigm has remained confined to relatively simple robot tasks.

Workcell modeling at the University of Texas has traditionally been deterministic. Figure 2-1 shows the structure of a typical Workcell model as originally implemented in

UT's Operational Software Components for Advanced Robotics (OSCAR). [Knoll, 2007] OSCAR has since been repackaged for commercial distribution as a software package called Kinematix, available from Agile Planet, Inc.

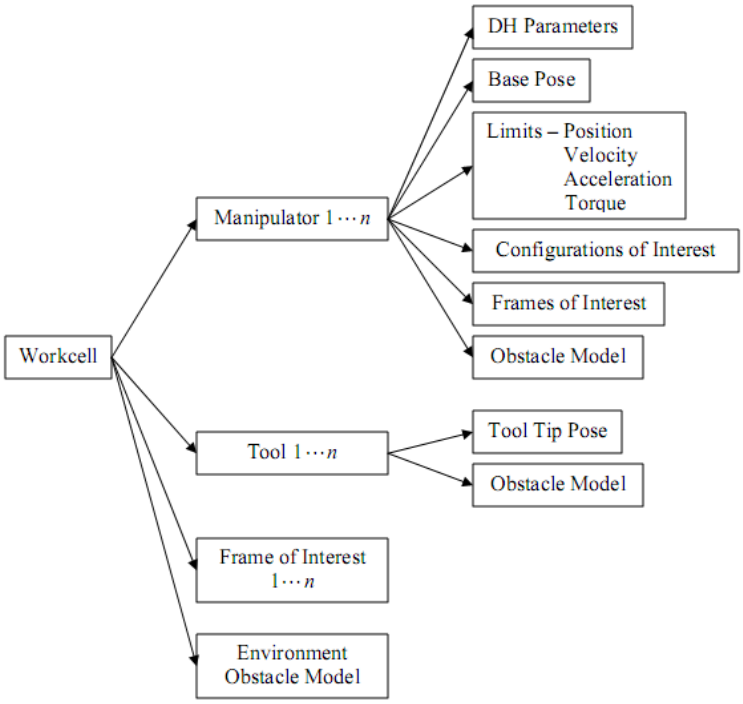


Figure 2-1. OSCAR/Kinematix Workcell model

The OSCAR/Kinematix Workcell model is implemented as an XML schema that contains a rich representation of the properties of the environment that are relevant to robot manipulator control. The XML file points to external files that contain the obstacle models for objects in the environment. These may be simplified geometric primitive representations, or polygonal models derived from CAD data. All of the information contained in the model and associated files is discrete. There is no provision for modeling uncertainty.

The OSCAR/Kinematix Workcell model was later reformulated into a generalized graph in which the links between nodes carried their own information about the relationships between entities in the environment. [O'Neil, 2010, O'Neil 2011] The graphical model is a much richer topological representation of the environment than the tree-based approach. As implemented, both the OSCAR model and the graphical model contain only discrete deterministic values. However, the high-level model is independent of the low-level representation, and there is no reason why either of these could not be adapted to probabilistic modeling and control.

2.1.2 Probabilistic modeling

The idea of probabilistic modeling in robotics can be traced to the Kalman filter. [Kalman, 1960] The Kalman filter is a recursive state estimation technique using a known state transition function and a measurement from a well-characterized sensor. The Kalman filter does not model states as deterministic values, but as multivariate normal probability distributions. The idea that spatial information cannot be known deterministically is important. Reif's technique of incorporating a discrete uncertainty parameter is insufficient because its meaning is unclear. It could be that it is physically impossible to make a larger error, in which case the parameter would be unnecessarily restrictive. Instead it could be a threshold that the robot is very unlikely to exceed. In that case, the threshold becomes arbitrary and begs the question of the level of uncertainty in the uncertainty parameter. The Kalman filter models and propagates uncertainty explicitly, and has proven the value of probabilistic modeling in countless applications.

Smith and Cheeseman [1986] emphasize the importance of estimating and modeling spatial uncertainty in robotics. They present a method for combining serial and parallel coordinate transformations between coordinate frames modeled as random

Gaussian variables. Parallel transformations, such as a transformation derived from external sensors and another from odometry, are merged through Kalman filtering, whereas serial transformations add and accumulate uncertainty. This method can reduce most networks of estimated transformations to a single transformation that, under certain assumptions, preserves the uncertainty information in the network. The result is that the uncertainty in a position estimate is stored and updated by sensing. This is a much more sound approach than choosing a static margin of uncertainty and also allows position estimates from multiple sensors to be fused to obtain a better position estimate than a single sensor alone could achieve.

The Kalman filter is a specific type of a general class of methods called Bayes filters. All Bayes filters recursively update the probabilistic belief about the state in a two-step process.

1. Prediction step: A state transition function is used to predict the state at time t given the state at time t-1.
2. Update step: A measurement is incorporated and the predicted distribution is updated according to Bayes' rule.

Bayes filters require probabilistic state-transition and measurement models, regardless of how the Bayes filter is implemented. The Kalman filter is the most common example of a class of Bayes filters called parametric filters. The Kalman filter assumes a Gaussian probability distribution and maintains the belief about the state by recording the parameters that describe this distribution. These parameters are frequently the mean vector and covariance matrix, a representation known as the moments parameterization. Other Gaussian parameterizations exist that carry the same information but lend themselves to better computational efficiency in some algorithms. [Maybeck, 1990]

Bayesian filtering methods require modeling an object's state as a multivariate Gaussian distribution. In reality, this distribution is not always a valid model. The underlying distribution may be multimodal in which case a single Gaussian would be a poor approximation. Austin and Jensfelt [2000] present a method that uses multiple Gaussians to represent the probabilities of several hypotheses about a mobile robot's location. When measurements are taken from the environment, they are associated and intersected with matching hypotheses. This generates a new set of hypotheses, the least likely of which is pruned. Another example of using multiple Gaussians to track multiple hypotheses comes from the computer vision literature. [Stauffer and Grimson, 1998] A Gaussian Mixture Model (GMM) can be used to model several image pixel states that represent the background in a static image. In this model, a fixed number of Gaussian models the underlying probability distribution. Each Gaussian has an associated weight, ω_i. A new observation is matched to the closest Gaussian by Mahalanobis distance, and the mean and variance of that distribution are updated by alpha blending. The weights of all distributions are then updated by alpha blending to increase the weight of the matched distribution and decrease the weight of the remaining distributions.

A further departure from Gaussian modeling is to use nonparametric techniques to capture underlying probability distributions. In various ways, these techniques avoid forcing posterior distributions to fit a parameterized model such a Gaussian. A histogram filter divides the state space into bins and represents each bin with a uniform distribution. [Thrun, 2006] Update of the belief can then be accomplished by applying a discrete Bayes filter. A drawback of this approach is the accuracy of the modeled distribution is highly dependent on the decomposition of the state space into bins. If there are too few bins or if they are too wide, the histogram representation will be too coarse to capture the underlying distribution. The tradeoff is that using many bins involves higher

computational cost. To address this, dynamic decomposition techniques such as density trees vary the bin size according to the posterior. In less likely regions, the resolution is coarser because no information about the distribution is captured by maintaining several bins in areas of near-zero probability. Selective update techniques finely discretize the bins, but only update bins with relatively high probability, avoiding the high computation cost of the fine grid.

Monte Carlo techniques are another class of nonparametric modeling methods. Monte Carlo methods have been used to model the location of mobile robots in stochastic environments, providing improvements in global localization performance and computation time. [Dallaert, 1999] These are sampling-based methods that represent the underlying density function as a set of random samples or particles. The belief update follows the Bayesian two-step predict/update paradigm for each particle in the sample. In the prediction step, a new particle is drawn randomly from the state-transition distribution. The update step weights each particle by the probability of achieving the current measurement given the hypothesized state. If the algorithm stopped there, particles would eventually be "lost" to regions of low probability. This phenomenon is addressed by importance sampling. In importance sampling, a new set of particles is sampled from the original set with replacement. The probability of selecting each particle is proportional to the weight assigned in the update step. This resampling forces the particle density back to the posterior distribution.

Probabilistic modeling can also reduce computation time for certain tasks. For example, the generalized mover's problem would be solved if the free configuration space were finely discretized with a motion plan generated between configurations. However, the computation of a complete map is prohibitively expensive. A more efficient technique is to randomly generate configurations from a probability distribution, and then

connect them with plans generated from a weak but fast motion planner. [Kavraki, 1996] The result is a graph in configuration space with robot configurations represented by nodes and possible paths represented by edges. Such an algorithm can continue adding nodes until the graph approaches a complete connectivity map of the free configuration space, but computation is reduced by stopping this learning phase when the graph connectivity is sufficient for practical motion planning. Such random methods do not actually represent the environment probabilistically and thus do not capture any information about uncertainty.

Robots operating in uncertain environments must make use of sensors to perceive and estimate the environment state, and then execute a control action based upon this state estimation. However all sensor measurements and control actions are subject to stochastic effects and thus introduce uncertainty. Therefore the control policy that makes decisions for a robot in an uncertain environment must consider this stochastic nature. A common approach that considers the probabilistic state of the environment in planning and control is the Markov Decision Process (MDP). [Kaebling, 1998] The MDP framework considers the stochastic nature of action effects, but not perceptual effects. In other words, the model assumes that the state of the environment can be fully and deterministically sensed at all times. In the MDP model, the state of the robot after taking an action is not known deterministically, and so its control policy must be able to select an action for a range of possible states. One way to accomplish this is to generate a universal plan that directly maps the robot's state to a control action and encompasses the entire finite state space of the robot.

In most applications, the assumption of the MDP model that the environment state is fully observable does not hold. Sensor measurements are noisy and not reliable enough to justify such an assumption. The case where both action results and perception results

are considered probabilistically is known as the Partially Observable Markov Decision Process (POMDP). Dropping the complete observation assumption significantly complicates the problem of determining a control policy as the robot must consider the state of its knowledge about the environment in making control decisions. If there is large uncertainty in the robot's belief about the environment state, the optimal control action is very likely that which results in the greatest disambiguation of the environment model. The control policy for a POMDP cannot be computed by mapping the robot's state to an action as this assumes a deterministic environment state. Instead, the control policy must map the robot's belief state to an action. This adds considerable computational complexity because the dimensionality of the belief space is of the order of the number of possible states. (i.e. for a continuous state space, the belief space is of infinite dimension.) Nonetheless, practical methods exist for approximating optimal control policies for POMDP's. Kaebling [1996] essentially treats the POMDP as a fully observable MDP, treating the most probable state as deterministic. However, the probabilistic belief about the state is monitored. If the entropy in the distribution gets above some threshold, the controller will select a control action that most reduces the entropy.

A glance at the papers cited above reveals that the MDP and POMDP solution techniques focus heavily on mobile robot navigation and control. Robotic manipulation in stochastic domains presents challenges that are difficult to address with these techniques. Most of the techniques for solving POMDPs require that the observation space be finite. This requirement is easy to meet by discretizing the continuous state space for a mobile robot that may only need to localize itself within a square meter. Hoey and Poupart [2005] observe that a control policy need only differentiate between states if they will affect the control action. They present a method for solving POMDPs with continuous observation spaces. They apply their method in an application that guides the cognitively

impaired through daily tasks. This type of guidance is not far related from task planning for robots. In the end, their method merely provides a more intelligent discretization of the observation space where observations are lumped together into clusters that identically affect the choice of control action. Consider the problem of grasping an object with position determined from sensing. For a successful grasp, the relative position of the gripper and object must be sufficiently accurate to permit the gripper to move into a pre-grasp position without collision, and the fingers must be placed accurately enough to assure a force-closed grasp Keeping in mind that the dimensionality of the belief space is on the order of the number of possible states, It is impractical to discretize the state space to such a fine resolution, particularly in the 6-dimensional spatial case. Therefore to frame the task of grasping as a POMDP results in a belief space that is too complex to be practical.

Another difference between manipulation and mobile robotics in uncertain domains is the source of uncertainty. The primary uncertainty of concern for a mobile robot is the uncertainty in its own position. Therefore the modeling and planning techniques focus on modeling and propagating uncertainty through transitions of the robot between states. In manipulation, hardware encoders on individual joints make the uncertainty in the robot's state very small compared with that of the objects in its environment. Particularly in the case of a glovebox, the primary concern of a probabilistic modeling scheme is not to localize the robot, but to recognize and localize the objects for collision avoidance or manipulation. Glover, Rus, and Roy [2008] propose a probabilistic object modeling framework specifically directed at manipulation. Their approach is to model 2D contour shape similarity by the Procrustean metric, dp. This is a measure of shape similarity after removing the relative translation, rotation and scale transformations between two contours. The object model is a Gaussian that describes the

shape variation within an object class. The authors demonstrate that this model can be used for object recognition and grasp planning in the presence of clutter and occlusions.

2.1.3 Previous modeling work at NRG

Previously, NRG has developed graphical world modeling techniques intended for use in unstructured environments [O'Neil 2011]. The nodes in the graph represent specific types of entities in the environment and the links represent specific types of relationships between nodes. A simple example is given in Figure 2-2.

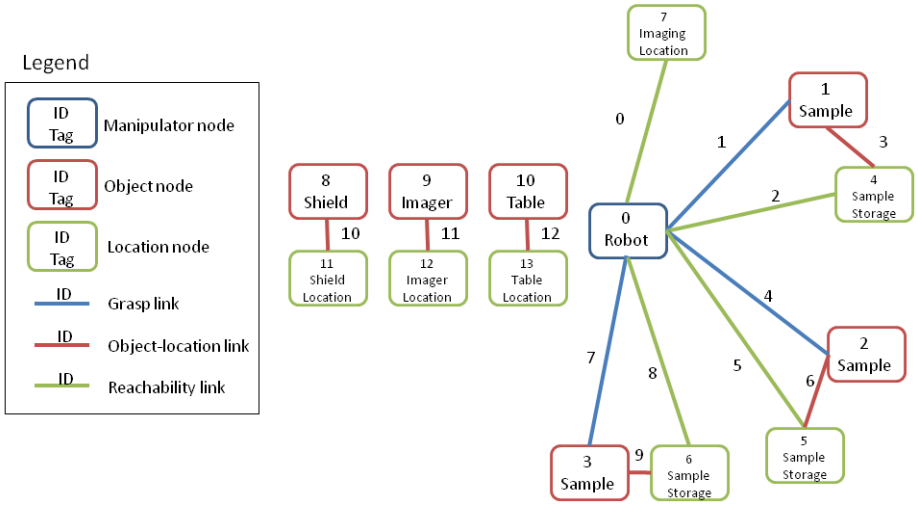


Figure 2-2. Graphical world model

All entities in the environment are represented as one of three node types. A *Robot* node represents a robot and contains information about the kinematic configuration of the robot as well as any geometric models required for collision detection. An *Object* node represents any physical entity in the environment that is not a robot. The object node may contain any data about the object that might be useful to the control system. A *Location* node describes a specific position and orientation in the environment.

Links describe the relationships between the nodes listed above. There are three types of link. A *Locator* link may exist between any two nodes and describes their relative position to each other. The existence of a locator link describes some kind of spatial connectivity. A *Grasp* link exists between a robot node and an object node. Its presence indicates that a valid grasp exists. The grasp link object may contain the grasp parameters required to make the grasp. A *Reachability* link can exist between a robot and a location. Its presence indicates that the robot can reach the location and may contain a joint configuration that optimizes the robot's motion potential at that location.

An important feature of the graph-based model is that the links and nodes described here can be created and added to the model with missing or incomplete information. This allows the control system to, in effect, know what it doesn't know. If the control system requires information about what objects are in a particular fixture, the model can provide this information even if the exact positions of the objects in the fixture are unknown. However, in the current implementation, if the position is known, it is modeled deterministically. Therefore the uncertainty model only permits two states: known with full certainty or completely unknown.

The graph-based model described here is used to support collision detection and prevention as well as motion planning and grasping. The model enables high-level control commands which are executed by semantic search of the graph. In the presence of missing information, the control system gives control to a human who can decide how to proceed or complete the task via tele-operation.

2.1.4 Summary and analysis of modeling techniques

This section has examined some deterministic approaches to world modeling. These models are suitable for manipulator operation in tightly controlled environments,

but do not handle uncertainty well. Some deterministic models attempt to deal with uncertainty by providing a safety margin or uncertainty factor to assure correct operation under the worst-case scenario. These methods are unnecessarily limiting and do not capture the underlying stochastic nature of the real world. Several probabilistic methods are presented that approach the problem of modeling probabilistically. The overwhelming majority of these methods were developed to solve the problem of mobile robot localization and navigation in uncertain domains. MDP and POMDP approaches have enjoyed success in this field, but the high dimensionality of the state space and the need for fine scale control make them impractical for most manipulation applications. However, in the continuous state space of an object's location, Bayesian filtering techniques can be used to maintain a probabilistic belief about an object's state. The belief can be modeled parametrically as a Gaussian or mixture of Gaussians, or nonparametrically as a sample of particles. A Gaussian can be easily updated from measurements by Kalman filtering. A mixture of Gaussians can be updated with an alpha blending scheme, and a particle representation can be updated with Monte Carlo techniques with importance sampling.

All of these techniques update the probabilistic belief from sensor measurements assuming a good perception model is in place**. Previous world modeling efforts at NRG are agnostic to the image representation and are therefore fully compatible with any of the deterministic or probabilistic techniques described in this section. To make the most of these techniques, a sensing system should be able to provide data that can populate and update the model.** The next section addresses this by reviewing work related to visual recognition.

2.2 VISUAL RECOGNITION

This section discusses literature relevant to object recognition from visual sensors. Traditionally, this sensor is a digital camera where each pixel has a single channel (grayscale) or three channels (color). The visual recognition process typically involves two separate phases, a training phase and a test phase. Each phase follows a similar pipeline:

1. Interest point selection
2. Feature extraction
3. Training (training phase)/Classification (test phase).

Design of a visual recognition system therefore requires three broad considerations:

1. Where to sample image features?
2. How to collect features into a single image representation?
3. What type of classifier?

The problem can be decomposed this way because the decisions are largely independent, i.e. the choice of image feature does not usually require a particular classifier. This section will review literature relevant to each of these three considerations.

2.2.1 Interest point selection

Here, interest point selection is defined as deciding where to extract features from the image. This process frequently includes foreground/background segmentation to assure that features are only extracted on objects of interest. This process is fundamentally different for static images and video sequences.

Foreground/background segmentation is somewhat easier in video because the temporal nature of the data provides many cues for background segmentation that aren't

available in static images. A naïve approach to segmentation in video would be to take the mean or median value of a pixel over time to be the background. Any time the pixel varies significantly from a central measure, it is considered foreground. In practice, this method fails if the background is not perfectly static. A swaying tree branch, for example results in a pixel having multiple background states (sky and leaf). The previous section mentioned the Gaussian Mixture Model (GMM) of Stauffer and Grimson [1998]. Their method represents multiple hypotheses of the background with a mixture of several Gaussians. This results in a foreground/background segmentation with considerably less noise. The approach is limited since the number of Gaussians is fixed, and the algorithm relies heavily on tuning the learning rates. An approach that allows an arbitrary number of hypotheses is presented by Kim. [2004] This method represents background hypotheses for each pixel as entries in a codebook. In addition to modeling an arbitrary number of background states, the method uses less memory, runs faster, and is not as sensitive to tuning of parameters.

In static images, the background cannot be observed over time, and foreground segmentation becomes more difficult. Segmentation of a static image is usually considered the process of dividing the image into coherent sub-regions or superpixels that contain homogeneous visual content. The resulting superpixels must still be processed to locate and classify interesting objects. If the processing required is modest, it may help with the task of interest point selection. Otherwise, recognition techniques that start with a static segmentation will likely sample features on the entire image to classify each region independently. If this is the case, the segmentation is only to separate coherent regions rather than to select regions on which to extract interest points.

Graph-cutting techniques are among the most common static image segmentation methods. Wu and Leahy [1993] present a technique for modeling an image as a weighted,

undirected graph whose edge weights describe the similarity between regions. This graph is recursively bisected by minimizing the total weight of the edges removed to make the cut. This continues until there are k remaining regions which each enclose a visually coherent image region. This minimization favors cuts that remove few edges and thus artificially selects small sets of isolated regions. Shi and Malik [2000] normalize the cut cost to the fraction of total edges removed, and therefore produce an unbiased grouping. They pose the optimization of the cut criterion as an eigenvalue problem, yielding good results but poor computational efficiency. Felzenswalb [2004] presents a method that adaptively adjusts the segmentation criterion, and makes greedy decisions about what edges to cut. The method preserves the global coherence of image regions, but is computationally much faster than eigenvector-based min-cuts. The Felzenswalb method has been widely implemented in computer vision problems that extend beyond simple object recognition. [Hoiem 2005a, Hoiem, 2005b, Gould, 2008] Carreira and Sminchisescu [2010a] use a graph-cutting technique and a classifier to automatically identify object-like regions in order to automate the foreground segmentation process. This allows downstream processing of the foreground segments for classification. [Carreira 2010b] Another approach to segmentation is to define a measure of visual saliency that approximates human intuition about the most important parts of a scene. [Liu, 2007] This process involves feature extraction across the entire image and training on user annotated data. Use of saliency has been demonstrated to improve object detection speeds by focusing an algorithm's attention on the most salient image regions. [Navalpakkam, 2006]

If depth imagery is available, the image can be segmented by finding clusters of points that belong to the same object. This can be done by examining how the points in the cloud relate to their neighbors in Euclidean space. Rusu provides a method for

performing Euclidean clustering by starting at some point and searching for nearby points. For input point cloud *P*, the algorithm proceeds as follows:

- Set up an empty list of clusters, *C* and a queue of point to be processed, *Q*.
- For every point $\boldsymbol{p_i} \in P$
- add $\boldsymbol{p_i}$ to the current queue, Q.
- for every point $\boldsymbol{p_i} \in \boldsymbol{Q}$
    - Search for the set of points P_i^k that fall within a sphere of radius d_{th} of $\boldsymbol{p_i}$
    - For every point $\boldsymbol{p_i^k} \in P_i^k$
        - Check to see if the point has been processed previously, add if not, add it to *Q*.
- When all points in *Q* have been processed, add *Q* to *C* and remove all points from *Q*.
- Terminate when all points $\boldsymbol{p_i}$ have been assigned to a cluster.

The result of the algorithm is a set of clusters, *C*, that satisfies:

$$min\|\boldsymbol{p}_i - \boldsymbol{p}_j\| > d_{th} \qquad 2\text{-}1$$

where $\boldsymbol{p_i} \in C_i$ and $\boldsymbol{p_j} \in C_j$. Each cluster represents a tight group of points that can reasonably be assumed to belong to the same object.

However, in the case of several objects sitting on a table, all objects would be assigned to the same cluster because there is a continuous surface of points separated by distances less than d_{th}. To account for this, Rusu locates the largest planar component in the dataset by RANSAC fitting, and removes all planar inliners from *P* prior to Euclidean clustering.

The video segmentation methods discussed previously function on each individual pixel independently and as such, extract very little information about local or global features of the image. The static segmentation and Euclidean clustering

techniques, on the other hand, must extract some visual features in order to perform the segmentation, and may obviate the need for additional interest point selection. In fact, superpixel segmentation and Euclidean clustering are, in a sense, region-based interest point selectors as they identify portions of the image that appear to be visually related and therefore interesting for classification. In the case of video segmentation, or static segmentation where further processing is to be performed within a region, the region must be described by local visual features extracted at interest points.

A widely implemented interest point detector is the Harris corner detector. [Harris 1988] The Harris detector locates areas in an image of high curvature (corners) which are intuitively interesting features in an image. At each pixel, the Harris detector observes the change in intensity that occurs with a shift in any direction. If shifts in different directions result in significant intensity changes, then the point is likely a corner point. The Harris corner detector has the attractive property of being partially invariant to rotation, translation, and scale. This invariance is important for object recognition where the features extracted should be the same regardless of the orientation of the object in the image. David Lowe [1999] formulated a scale invariant interest point detector. This detector takes the Difference of Gaussian (DoG) filter responses between image pyramid levels, and identifies extrema of this function in scale space. An image pyramid is constructed by recursively filtering and subsampling the original image, resulting in a pyramid of progressively lower-resolution images. This method is among the most common interest point detectors currently implemented.

2.2.2 Feature selection

The previous section examines literature relevant to the question of where image features should be sampled. This section will address the question of what low-level

image features to extract. Image descriptors serve to extract visual information in an image into a mathematical representation that is convenient for use in visual recognition. Most local feature descriptors capture image gradients in an image patch surrounding the interest point.

Appearance Features

David Lowe's [1999] Scale Invariant Feature Transform (SIFT) descriptor is a commonly implemented descriptor used in both instance and category-level classification. The SIFT descriptor computes histograms of gradients in each cell of a 4x4 grid surrounding the interest point. Each of these histograms is binned into 8 directions. The SIFT descriptor is therefore a 4 by 4 by 8 or 128-dimensional vector that describes a local image patch. These are typically extracted at the same scale at which the interest point was detected making the SIFT descriptor invariant to change in scale. The SIFT descriptor provides the basis for very impressive classification results. [Lowe, 2004, Mikolajczyk, 2005]

Another descriptor based on local image gradients is the Histogram of Oriented Gradients (HOG) descriptor. [Dalal, 2005] The HOG descriptor is extracted by dividing an image into subregions or cells and binning the orientation of image gradients within the cell into a histogram. This histogram constitutes the local image descriptor, and one such histogram is extracted from each cell in a densely sampled (or even overlapping) set that encompasses the entire image. In spirit, the HOG descriptor is very similar to the SIFT descriptor. They both represent image content with histograms of local image gradients. However, whereas SIFT features are extracted at DoG interest points at the interest point scale, HOG features are densely sampled on the entire image at uniform scale. As such, the HOG descriptor is not invariant to scale. Also SIFT are sampled on a

grid that aligns with the interest point's dominant gradient orientation, giving it invariance to rotation. HOG are sampled at uniform orientation and are therefore not invariant to rotation. Despite the lack of invariance to scale and rotation, the descriptor yields excellent performance in human detection if the humans are generally in an upright orientation.

SIFT and HOG both describe the texture of an image by its local gradients. However, sometimes texture descriptors alone do not describe the image content well. They work very well on commercial packaging, album covers, and other objects that have considerable texture. However, many objects are of relatively uniform color, and do not have many strong local gradients that can be captured by texture-based descriptors. For such objects, the shape is more important than the texture. One way to describe shapes in an image with Maximally Stable Extremal Regions (MSER). [Matas, 2002] MSERs are regions that are significantly brighter (or darker) in intensity than the surrounding image content. Stability refers to the property that the region does not change with respect to changes in intensity threshold. Because the regions are defined only by their intensity, they are invariant to affine transformations, and partially invariant to changes in scale. Once MSERs are detected in an image, a descriptor must be extracted. Forssen and Lowe [2007] use a set of SIFT descriptors on MSERs to extract a shape descriptor.

Depth Features

Recently, the proliferation of inexpensive, consumer-grade depth imaging devices such as the Microsoft Kinect™ have ignited a surge of activity in object recognition from depth imagery. These devices produce an image that consists of a depth value for each pixel. Using the camera's intrinsic matrix, the depth values can be transformed into 3D

world coordinates. Many of the same features used for traditional 2D images can be directly applied to these images as if they were grayscale images. [Lai, 2011] However, gradient based features like SIFT and HOG perform best in the presence of high surface texture. Most objects have only weak gradients in their 3D shape, so texture-based features are weaker in depth imagery than in grayscale intensity images. The 3D spin image feature [Johnson, 1999] is probably the most common descriptor of 3D shape used presently. [Lai, 2010, Lai, 2011] The spin image is a local descriptor generated from a set of 3D points and associated surface normals. It uses a binned 2D accumulator that can be thought of as a sheet that is spun around the surface normal of a point. As points pass through a bin in the spinning sheet, the bin value is incremented. The spin image descriptor is then a fixed-size histogram that represents the local shape information around a point. In this way it can be used in a manner analogous with SIFT descriptors for shape-based recognition.

A spin image is characterized by three parameters: The histogram bin size, the support angle, and the image width. Selection of bin size is straightforward: the histogram bin size should be similar to the image spatial resolution. The support angle is the maximum angle between of the surface normal of the oriented point and that of a point that contributes to the histogram. A large angle will result in an accumulation of more points, but will be more subject to clutter if the image is not segmented. The image width refers to the maximum distance from the oriented point that points are permitted to contribute to the histogram. It determines the global/local nature of the spin image: the larger the width, the more global the shape description. Figures 2-3 and 2-4 show how the spin image is computed, and also how the image width and support angle vary the global/local nature of the feature. [Johnson, 1999]

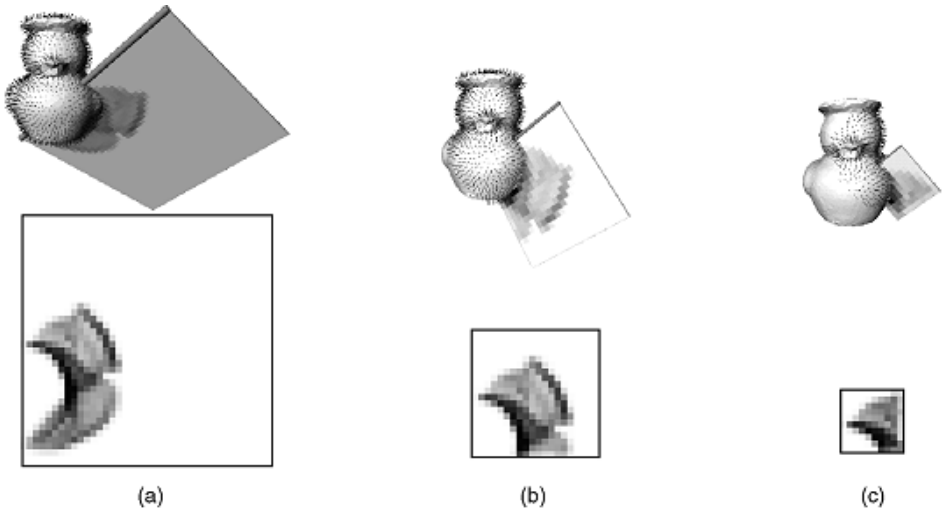


Figure 2-3. Effect of image width. As the image width decreases, the histogram is limited to an increasingly local neighborhood. (a) 40 pixel width. (b) 20 pixel width. (c) 10 pixel width. [Johnson, 1999]

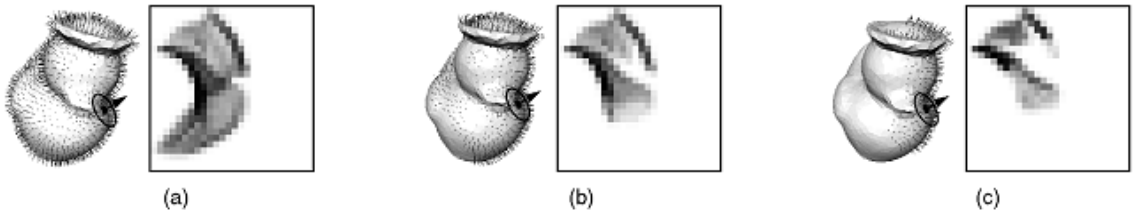


Figure 2-4. Effect of support angle. As the support angle decreases, points from the other side of the image are excluded. (a) 180 degrees (b) 90 degrees (c) 60 degrees [Johnson, 1999]

The spin image is a member of a class of features called shape histograms. In cylindrical coordinates, the spin image projects points into a histogram along the azimuthal axis and bins them according to their vertical and radial coordinates. This both reduces the dimensionality of the input space and generates a feature vector of uniform length for any point cluster. Ankerst [1999] Refers to the bins in the radial direction as shell bins, and the bins in the azimuthal direction as sector bins, and proposes a combined "spider-web" model that is very similar to the spin image, differing only by the axis along

which the points are projected in cylindrical coordinates. Rather than projecting azimuthally, the points are projected vertically. The shell, sector, and spider-web approaches and histogram samples are shown in Figure 2-5.

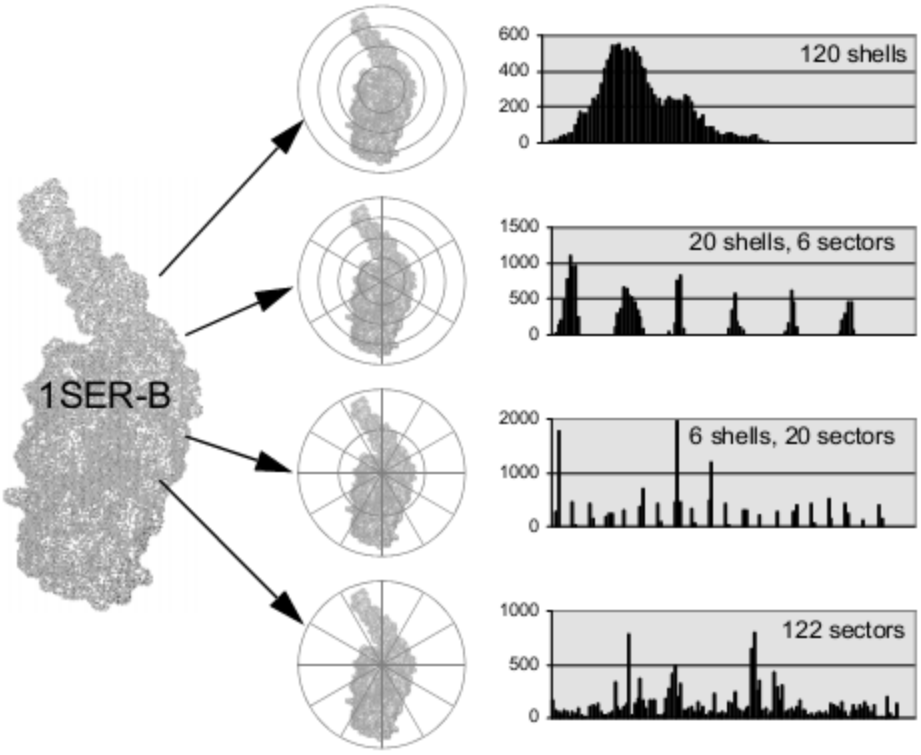


Figure 2-5. Different 3D binning schemes and the resulting histograms. [Ankerst, 1999]

Another type of histogram feature called the Point Feature Histogram (PFH) does not directly bin the points, but instead describes local shape by binning relationships between the surface normals of points within a neighborhood [Rusu, 2008]. The feature is constructed by the following algorithm illustrated by Figure 2-6:

- For each point *p*, select all of *p*'s neighbors that lie within a sphere of radius *r*.

- For each pair for points p_i, and p_j with estimated surface nomals n_i and n_j (i ≠ j),
    - Assign a Darboux *uvn* frame to one of the points.
        - $u = n_i$
        - $v = (p_j - p_i) \times u$
        - $w = u \times v$
    - Compute angular variations as follows
        - $\alpha = v \cdot n_j$
        - $\varphi = \left(u \cdot (p_j - p_i)\right) / \|p_j - p_i\|$
        - $\theta = arctan2(w \cdot n_j, u \cdot n_j)$
- Bin the α,φ, and θ values for each pair of points in the neighborhood into a histogram.

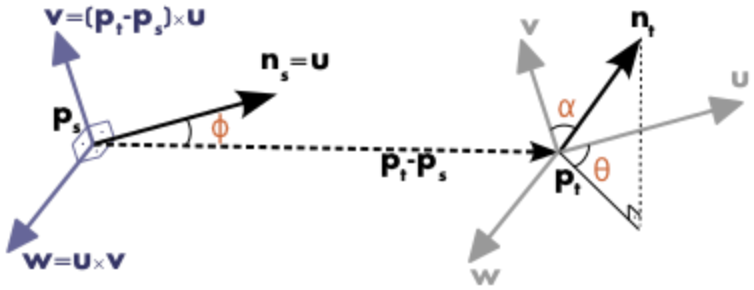


Figure 2-6. Point Feature Histogram normal differencing scheme [Rusu, 2009].

Unlike Ankerst's shape histograms, PFH and its faster variant Fast PFH (FPFH) [Rusu, 2010] are local, pose invariant features that are used with a sample consensus algorithm to perform alignment of point clouds. In that sense, they are directly analogous to SIFT features, but in shape space rather than appearance space. As such, they could be

used in a manner similar to Lowe's correspondence grouping approach [2009], or with the bag-of-words model discussed in section 2.2.3.

In order to take advantage of the ability to isolate point clusters belonging to the same object (see section 2.2.1), Rusu has developed an extension of FPFH called the Viewpoint Feature Histogram (VFH) that is computed globally on a point cluster rather than locally on a few neighboring points. VFH also has the interesting property of *not* being invariant to pose. All of the features discussed previously are invariant to pose because a good object detector should detect an object regardless of its orientation in space. Determining the pose of the object would then require additional downstream processing. VFH, however, can be used to not only recognize an object, but also determine its orientation in space. This ability to perform pose estimation is essential to robotic object manipulation.

Like FPFH, the *extended FPFH component* of VFH bins relative pan, tilt, and roll angles between points. However, this is done on different sets of point and using a different reference frame. Instead of computing a histogram on a point and its k neighbors, the extended FPFH computes the histogram on the angular differences between the surface normal of *all* points in the cluster and the *central* point in the cluster. This makes the feature global on the point cluster. This global nature of the extended FPFH feature is shown in, Figure 2-7.

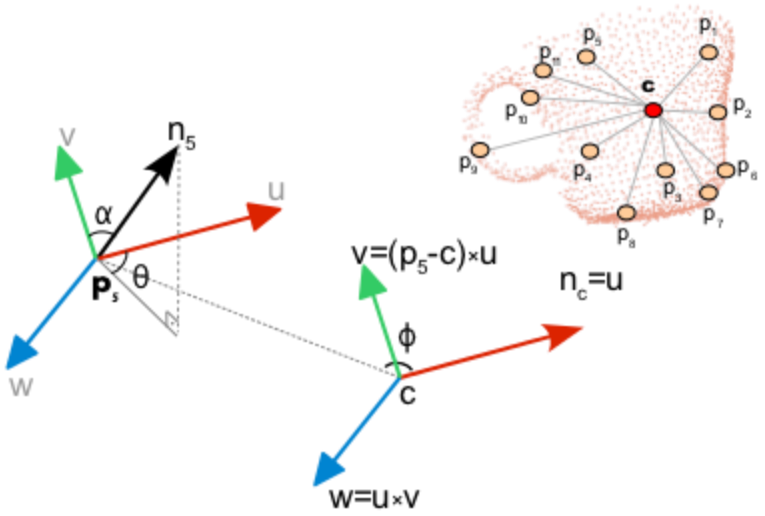


Figure 2-7. Computation of the extended FPFH component of the VFH feature [Rusu 2010].

The extended FPFH component of VFH captures the global shape of the cluster, but it still does not depend directly on the viewpoint direction. To achieve viewpoint variance, additional statistics are computed between the *viewpoint direction* and the surface normal at each point. This *viewpoint component* is then concatenated with the extended FPFH component to give the full VFH descriptor. The viewpoint component is computed by binning the angles that the viewpoint direction makes with the surface normal at each point in the cluster. The viewpoint direction refers to the direction of the camera's central axis translated to each point in the cloud, *not* the direction of the vector between the camera and each point. This formulation assures that the resulting feature will remain invariant to scale and translation. The computation of the viewpoint component of VFH is shown in Figure 2-8.

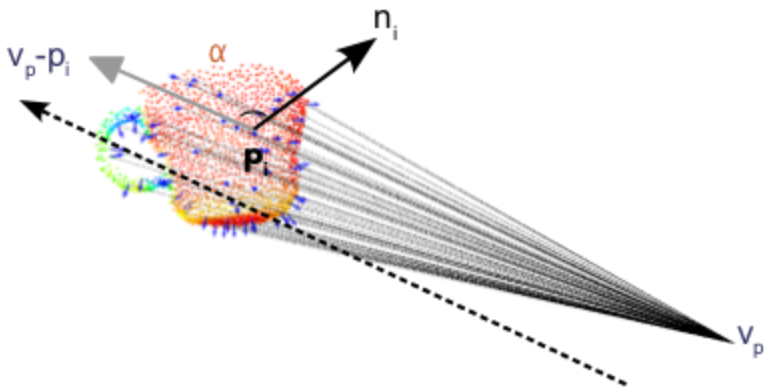


Figure 2-8 Computation of the viewpoint component of the VFH feature [Rusu 2010].

The extended FPFH and the viewpoint components are collected in a single histogram as shown in Figure 2-8.

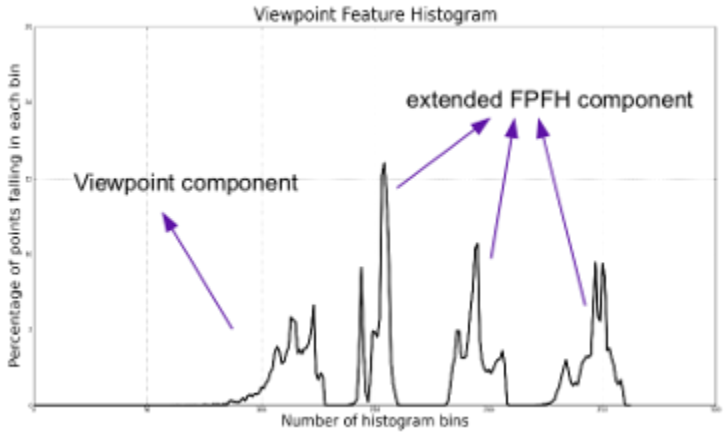


Figure 2-9. Full VFH feature [Rusu, 2010].

In 2D imagery, it is difficult to discern whether an object is large and far away, or small and close. For that reason, scale invariance is generally considered a requirement in an image feature. However, when trying to capture 3D geometry, it is important to differentiate between scale invariance and size invariance. The VFH feature described above is formulated to be invariant to scale. This means the feature vector will be the

same regardless of its size in the image (or alternatively, its distance from the camera.) However, by being invariant to scale, it is also invariant to size. In other words, the feature will be identical for two objects of identical shape, but different size. This is a significant problem if the target dataset contains such objects.

Also, while the VFH feature itself in $O(n)$ in the number of points in the cluster, it requires that the surface normal be estimated beforehand. At best, this requires an SVD matrix decomposition of $O(d^3)$ in the number of matrix columns [Klasing, 2009].

2.2.3 Classification

The previous section discussed various features that can be described to represent the content of an image. This section examines how the features extracted from an image can then be used to classify the object(s) in an image.

Lowe [2004] performs classification by finding correspondences between SIFT features in a test image and SIFT features in a training image. Each matched feature has a keypoint with known 2D location, orientation, and scale. If the object is assumed to be rigid, each matched feature therefore suggests a hypothesis about the object's pose and scale. If three or more keypoints agree upon the pose of the object, the probability is high that the hypothesis is correct. The affine parameters of the object are then computed and the spatial consistency is checked against other matched points.

More recently, classifiers from the machine learning community have become popular in visual recognition. A classical approach to classification is the Bayesian classifier. [Duda 2001] The basic idea in Bayesian classification is to use training data to estimate the probability distribution of each class across the feature space. Given a test case, you extract the feature vector, and classify it as the most probable class. If the probability distributions for each class are known exactly, the Bayes classifier is optimal.

In the case where the prior class probabilities are unknown, the method is called Naïve Bayes. Bayes classifiers are widely implemented, but are typically outperformed by other methods. [Caruna, 2006]

A very intuitive approach to classification is the K-Nearest Neighbor (KNN) algorithm. With KNN, a metric such as the Euclidean or Mahalanobis distance is used to describe the similarity between feature vectors. Given a test vector, the k nearest vectors in the training set vote on the classification. KNN is easy to implement and is surprisingly powerful given the simplicity of the algorithm. [Caruna, 2006] In fact, for very large classification spaces, it has been demonstrated to work as well or better than state-of-the art machine learning methods. [Deng, 2010] Another advantage of KNN is that training is trivial, requiring only that the training samples vectors be stored with their class labels.

A popular state-of-the-art classifier is the Support Vector Machine (SVM). [Caruna, 2006] In a 2D feature space, it is easy to classify two classes of data if a line can be drawn that separates all instances of one class from another. In 3D, this decision boundary would be a plane. However, if the data are at all interspersed in the original feature space, such a line, plane, or hyperplane in dimensions greater than 3, cannot be found. The idea behind a support vector machine classifier is to map the original feature vectors into a higher-dimensional space where a separating hyperplane exists. [Duda, 2001] This mapping is done through a kernel function, φ. Popular choices include linear, polynomial, and Radial Basis Function (RBG) kernels. Ideally, the separating hyperplane maximizes the margin between itself and the nearest training instances. Training the SVM requires finding a set of weights for the hyperplane function that maximize this margin. A disadvantage of SVMs is they only handle two classes. Multi-class classification must be done by training several SVM classifiers. This can be done by

making the classification between each class and all other classes (1 vs. all), or making classifications between each pair (1 vs. 1) of classes.

All of the classifiers here presume that an image can be represented by a single vector of uniform length. From the earlier discussion of features, it is not clear how this can be done. The most common approach for transforming a set of visual features into a feature vector that can be used by a classifier is the Bag of Words (BoW) method borrowed from text document classification. [Sivic, 2003] Under this model, the features (such as SIFT or HOG) extracted from all of the training data are vector-quantized into a codebook or visual vocabulary of fixed size. A visual word in the vocabulary is a region of feature space that contains similar features extracted from training images. An image can then be described by a frequency histogram across all of these visual words. This histogram is referred to as the image's Bag of Words, and is a very compact and rich description of the image content. By describing images this way, every image has a feature vector of uniform length that can be used for classification.

2.2.4 Previous recognition work at NRG

Previous work in object recognition from depth imagery has approached the problem of recognition as a contour matching problem. In this work, *Procrustes analysis* [Dryden, 1998] is used to obtain an optimal transformation between contours in translation, rotation, and scale. The current work compares an extracted vertical cross-section of an object with a library of templates generated from training data. K-means clustering is used to group the training data into several template shapes for each object of interest. In this template clustering step, $k=3$.

At test time, a vertical cross section it taken from a segmented point cluster representing the object. The *partial Procrustes distance* is used as the shape similarity

metric. In order to discriminate between objects of similar shape but different size (such as the small and large cylinders shown in Figure 1-5 in Chapter 1). The partial Procrustes distance is computed by omitting the he scale component of the Procrustes transformation. A KNN classifier (k=1) is used to determine the class of the detected object. Figure 2-10 shows test cross-sections together with the matched template shape.

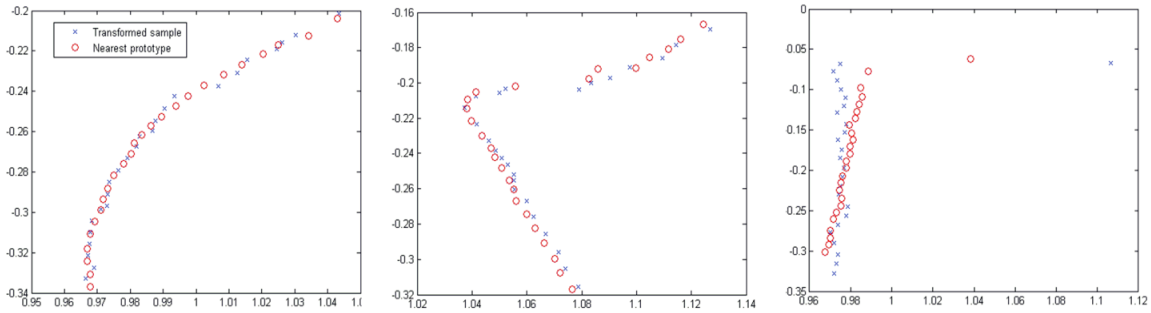


Figure 2-10. Procrustes matching results

Table 1 shows the classifier performance presented as a confusion matrix. The correct classification rates are reasonably high for each of the three objects in the dataset.

Table 2-1. Classifier confusion matrix

Object	Hemisphere	Small cyl.	Large cyl.
Hemisphere	0.80	0.20	0.00
Small cyl.	0.00	1.00	0.00
Large cyl.	0.06	0.00	0.94

The results presented in Table 2-1 suggest that the shape information contained in depth imagery can be discriminative enough for object classification even in the presence of specular reflections and poor surface texture.

However, the contour matching approach has several drawbacks that prevent it from being useful for recognition in a glovebox environment:

- The correct classification rates are not good enough for autonomous operation in a glovebox.
- The size of the dataset is too small to determine how well the method would scale.
- The 3D shape is reduced to two dimensions cross section extraction, completely discarding a significant amount of shape data.
- Uniformity in feature length is artificially achieved by under/oversampling the original data rather than by using a feature that is inherently uniform in length.

These drawbacks force the conclusion that Procrustes contour matching is not well suited to the problem at hand.

2.2.5 Visual recognition discussion

This section discussed several important considerations for design of a visual recognition system. A generic approach to recognition follows these steps:

1. Perform image segmentation
2. Extract features
3. Generate image or segment representation
4. Classifier training

This review has shown that segmentation can be done either *temporally* if video is available, or *spatially* on individual images or frames. Temporal segmentation requires that the background be relatively static, or that it have a small number of possible states. Spatial segmentation requires that objects in the image are reliably discernible from the background. Either approach could be applied to the problem of recognition in a glovebox. The background is reasonably static, making temporal segmentation possible.

The use of depth imagery and the fact that objects of interest in a glovebox lie on a horizontal planar surface enable reliable spatial segmentation.

The above sections have also discussed many types of image features that can be used for recognition, classification, and pose estimation. The features computed on depth data are of the most relevant to the current work. 2D contours were shown be a poor representation. Table 2-2 provides a comparison between the remaining point cloud features discussed above.

Table 2-2. Point cloud feature comparison.

Feature	Shape Hist.	Spin Images	FPFH	VFH
Scale Invariance	Y	Y	Y	Y
Global/Local	Global	Either	Local	Global
Captures Pose?	No	No	No	Yes
Captures Size?	No	No	No	No
Comp. Speed	Fast	Med	Med	Med

All of the features are scale invariant, and must be in order to be useful. Shape histograms, Spin Images, and VFH can all be computed on a segmented cluster of points to describe global object shape. None of the features captures object size. The global features lose the ability to capture size in order to achieve invariance to scale. Only VFH is capable of determining viewpoint direction from the feature itself. Spin images, FPFH, and VFH all require estimation of surface normals, so while the features themselves compute quickly, the requirement of an additional processing step slows them down considerably. None of these features has all of the qualities desired in a feature for recognition and pose estimation from depth imagery. VFH is the closest as it is the only feature discussed here that is even capable of describing both object class and pose. It has the added benefit of being fully implemented in PCL, and (through PCL), ROS.

Selection of a classifier is somewhat difficult because different classifiers seem to perform better under different circumstances. There is no single "best" classifier. This problem is often addressed by trying several for a given application and choosing the one that performs the best. To differentiate between specific object instances, K-nearest neighbor is intuitive and shows good results across several problems. For category-level recognition, SVM is popular, widely implemented, and also provides good results across several applications.

2.3 RELATED WORK DISCUSSION

This section has presented literature to probabilistic modeling and visual recognition. Several methods for encoding pose into a probabilistic representation are available. The choice of an appropriate method is based on computational considerations and the nature of the underlying probability distributions. For pose estimation, a parametric representation of probability is appropriate. Filtering operations for Gaussians are computationally simple, and the position of an object in a glovebox is well-represented by a Gaussian. It is unlikely that a position measurement from depth data would require more the more accurate modeling of the posterior provided by particle-based methods.

For visual recognition, traditional texture-based features are not likely to perform well in a glovebox environment due to poor surface texture on the objects of interest. Shape-based features computed on point clouds are likely to yield better results. The general approach to recognition is similar regardless of the type of imagery, but features derived from point cloud data naturally capture different information than those derived from appearance imagery. Several point cloud features exist, but none of them simultaneously achieves scale invariance while capturing size, shape, and viewpoint

invariance. Therefore none are well suited to simultaneous feature-based object recognition and pose estimation, particularly if the target dataset contains objects of similar shape but different size.

The next chapter will present a novel feature that *does* possess all of the qualities required for the problem of object recognition and pose estimation in a nuclear materials glovebox.

Chapter 3: Methods

Chapter 1 described the problem of visual recognition and pose estimation in gloveboxes and concisely stated some important assumptions and limitations which are restated here:

- The environment is uncluttered; objects can be readily segmented by Euclidean clustering or temporal back-ground subtraction.
- The objects have little surface or visual texture.
- The objects may be lightly reflective such as matte-finished stainless steel.
- We wish to differentiate between objects of similar shape but different size.
- We desire real time performance.

Chapter 2 reviewed related work in deterministic and probabilistic modeling and found that probabilistic modeling allows the uncertainty in visual measurements to be quantified and reduced by filtering results over several frames. It also examined several visual recognition techniques including a variety of point cloud features used for object recognition and pose estimation. None of these features addressed all of the assumptions and limitations above.

This chapter discusses the probabilistic object model used in the current work, and introduces a new and novel point cloud feature that addresses some of the shortcomings of the features discussed in Chapter 2.

3.1 Probabilistic Object Model

This section describes the probabilistic object model used in the current work, and the statistical filtering techniques used to update the model.

3.1.1 The model

An object model, M_o consists of its label, $\boldsymbol{l}$ and pose, S. The label is modeled by the discrete probability mass function across N possible classes, C_j, where $\sum_{j=1}^{N} P(C_j) = 1$, The object pose is modeled as a continuous multivariate Gaussian. In this work, it is assumed that all objects are vertical with respect to a planar surface, leaving 4 degrees of freedom in the pose: *xyz* position and rotation about the vertical axis, θ. This assumption reasonably assumes that objects are constrained to sitting on a planar work surface. The object model is described concisely by equations 3-1 through 3-3 where $\boldsymbol{\mu}$ is the mean of the 4D Gaussian and Σ is the covariance.

$$M_o = [\boldsymbol{l}, S] \tag{3-1}$$

$$\boldsymbol{l} = [P(C_1), P(C_2), P(C_3) \cdots P(C_N)] \tag{3-2}$$

$$S = [\boldsymbol{\mu}, \Sigma] \tag{3-3}$$

At any given moment, the best estimate of the object's label is $argmax_i(l_i \in \boldsymbol{l})$, and the best estimate of its 4D pose is $\boldsymbol{\mu}$.

3.1.2 Bayesian update

This posterior is updated by Bayes' rule where $P(C_k|z)$ is the conditional probability of class j given visual recognition result z:

$$P(C_j|z) = \frac{p(z|C_j)P(C_j)}{\sum_{k=1}^{N} p(z|C_k)P(C_k)} \tag{3-4}$$

To use a Bayesian update of the posterior, a sensor model is created for the visual recognition system to determine $p(z|C_j)$. This is accomplished by classifying many images of each class to generate the prior probability distributions. For previously unobserved objects, a uniform prior is assumed across all possible classes.

In the first observed image, the mean of the Gaussian is set to the measured position and the covariance is set very high. At each subsequent frame, the pose estimate

is updated by filtering the sensor measurement into the current Gaussian estimate of the object's pose by applying Equation 3-4.

While the model formulation permits the pose estimate to be updated by Kalman filtering, we assume a quasi-static object pose and omit the prediction step of the Kalman filter algorithm. This assumption is reasonable in our problem domain, where by design nothing moves rapidly. Update of the modeled pose proceeds as follows:

$$K_t = \Sigma_{t-1} C_t^T (\mathrm{C}_t \Sigma_{t-1} C_t^T + Q_t)^{-1} \quad 3\text{-}5$$

$$\mu_t = \mu_{t-1} + K_t(z_t - C_t \mu_{t-1}) \quad 3\text{-}6$$

$$\Sigma_t = (I - K_t C_t)\Sigma_{t-1} \quad 3\text{-}7$$

Equation 3-5 shows the computation of the Kalman gain, the value of which describes the relative contributions of the previous state estimate and the sensor measurement to the new state estimate. C_t is a matrix that describes the sensor model and Q_t is the covariance of the sensor noise distribution. These parameters will be determined by measuring object poses with the methods described against ground truth positions. Equations 3-6 and 3-7 compute the new mean and covariance of the object's position from the previous estimates and the Kalman gain.

3.1.2 Sensor model

In order to use equations 3-5 through 3-7, a sensor model for the visual recognition system must be in place. The sensor model requires three parameters:

- $p(z|C_j)$, The probability of obtaining recognition result z given Class C_j
- C_t, the pose sensor model.
- Q_t, the pose sensor noise covariance.

For each class, $p(z|C_j)$ is estimated by classifying several instances of the class and determining the fraction of instances yielding each possible result. For the pose

sensor model, it is assumed that the true pose is measured by the system, subject to noise. For the noise model, it is assumed that the covariance is independent:

$$Q_t = \begin{bmatrix} \sigma_x & 0 & 0 & 0 \\ 0 & \sigma_y & 0 & 0 \\ 0 & 0 & \sigma_z & 0 \\ 0 & 0 & 0 & \sigma_\gamma \end{bmatrix} \quad \text{3-8}$$

We then compute the variance in each dimension independently. This is important because γ, the rotation about the vertical axis, is determined in the classification step, whereas the xyz position is determined directly from range data. Therefore we expect far more noise in the rotational component of the pose. Each of the four variances is computed by classifying tens of thousands of images. This completes the sensor model necessary to use equations 3-5 through 3-7 to update the model.

3.2 THE CYLINDRICAL PROJECTION HISTOGRAM

This section presents the Cylindrical Projection Histogram (CPH), a new and novel feature suitable for both recognition and pose estimation under the assumptions and limitations laid out in Chapter 1.

Chapter 2 explored several point cloud features that can be used for object recognition. Of those, the VFH feature is most suitable given the limitations and assumptions above. It requires an uncluttered and easily segmented environment, and does not require significant image texture. However, it does not directly capture object size without sacrificing its scale invariance, and the pre-processing required for surface normal estimation may exclude its use when real-time performance is desired.

3.2.1 CPH shape histogram

The CPH feature itself is most similar to the shape histograms presented by Kazhdan [2003], although the target application is closer to that of VFH. Like those

features, CPH features are computed on clusters of points that are considered to belong to a single object. The approach is to project each point in the cluster outward from the cluster's centroid and bin the points as they pass through the surface of a cylinder of arbitrary radius. The resulting histogram captures the variation in point density as the cluster is viewed from different angles. The orientation of the cylinder into which the points are projected is fixed relative to the camera, meaning the baseline histogram is *not* invariant to viewpoint direction or object scale. The lack of invariance to viewpoint direction allows object pose to be estimated from the feature. The lack of scale invariance is an undesirable property, and is addressed by a modification to the baseline histogram that achieves scale invariance while preserving size information. This modification is discussed in Section 3.2.2.

The projection cylinder's height is equal to the vertical extent of the point cluster on which the feature is computed. Its vertical axis is oriented vertically with respect to the camera, normal to the viewing angle and intersecting the centroid of the point cloud. This orientation of the projection cylinder is shown in Figure 3-1.

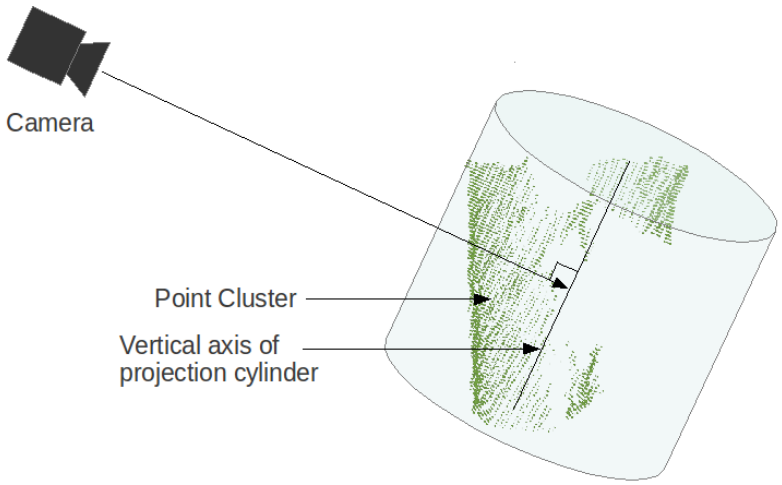


Figure 3-1. Orientation of projection cylinder

To compute the histogram, the projection cylinder's surface is quantized into M vertical bins and N azimuthal bins. Every point in the cluster is then projected into the cylinder and the bin into which it is projected is incremented by 1. The result is a 2D histogram that represents the spatial distribution of points in the cluster Figure 3-2 shows a conceptual example of this process where M=5 and N=8.

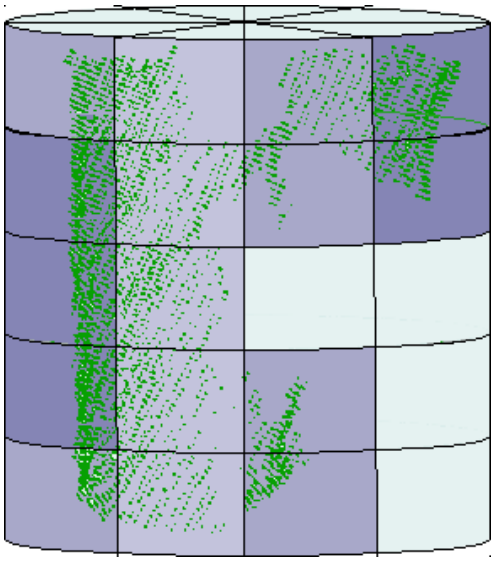


Figure 3-2. Example of histogram construction.

Another similarity of the CPH histogram to those constructed by Kazhdan's shape histograms and spin images in that all three features collapse points in 3D space into a 2D representation. Each feature performs the collapse in θ,r,h cylindrical coordinates, but the features vary in the axis along which they collapse the cluster. Assuming a global spin image where the support width and support angle include all of the points on a cluster, a spin image collapses all the points azimuthally, and bins them according to their r and h components. Kazhdan's spider-web shape histogram collapses the points vertically and bins them according to their θ and r components. CPH projects the points radially and

bins them according to their θ and h components. For a cluster of points $p_i \in P$, the CPH histogram is constructed as follows:

1. Start with an empty histogram vector:

$$\boldsymbol{h} = [h_0, h_1, h_2 \cdots h_{M \times N}] \tag{3-9}$$

2. Compute the centroid, c of the point cloud.

$$\begin{bmatrix} c_x \\ c_y \\ c_z \end{bmatrix} = \frac{1}{2} \begin{bmatrix} argmax_i(p_{ix} \in P_x) + argmin_i(p_{ix} \in P_x) \\ argmax_i(p_{iy} \in P_y) + argmin_i(p_{iy} \in P_y) \\ argmax_i(p_{iy} \in P_z) + argmin_i(p_{iz} \in P_z) \end{bmatrix} \tag{3-10}$$

3. For each point $p_i \in P$, compute vertical (y) and azimuthal (θ) bins and increment the correct bin in $\boldsymbol{h}$.

$$y = floor \left[\frac{p_{iy} - argmin_j(p_{jy} \in P_y)}{\left(argmax_i(p_{iy} \in P_y) - argmin_i(p_{iy} \in P_y)\right)/M} \right] \tag{3-11}$$

$$\theta = floor \left[\frac{\pi + atan2(p_{iz} - c_z, p_{ix} - c_x)}{2\pi/N} \right] \tag{3-12}$$

$$h_{y*N+\theta} = h_{y*N+\theta} + 1 \tag{3-13}$$

4. Terminate when all points have been binned.

An example of a histogram computed from the algorithm above is shown in Figure 3-3. The histogram in Figure 3-3 provides a unique signature for a specific view of a single object instance. However, because the total area of the histogram is proportional to the number of points in the cloud, it is not invariant to image scale.

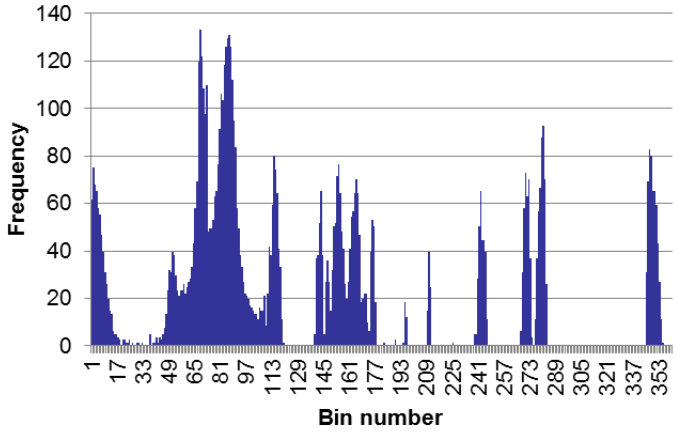


Figure 3-3. Baseline histogram for CPH feature.

3.2.2 Spatial Extents

The histogram described in the previous section is not invariant to image scale. If an object is larger or closer to the camera, there will be more points in the resulting cluster. This means that the cluster feature cannot differentiate a larger object from one that is just closer to the camera. We desire a feature that is invariant to image scale, but not invariant to the object size. Therefore it is insufficient to normalize the histogram because this would make the feature invariant to both image scale and object size.

To accomplish scale invariance while retaining information about object size, two modifications are made to the baseline CPH feature.

1. Compute the spatial extents, $\boldsymbol{s}$, of the cluster in Cartesian Space:

$$\begin{bmatrix} s_x \\ s_y \\ s_z \end{bmatrix} = \begin{bmatrix} argmax_i(p_{ix} \in P_x) - argmin_i(p_{ix} \in P_x) \\ argmax_i(p_{iy} \in P_y) - argmin_i(p_{iy} \in P_y) \\ argmax_i(p_{iy} \in P_z) - argmin_i(p_{iz} \in P_z) \end{bmatrix} \quad 3\text{-}14$$

2. Rescale the shape histogram, $\boldsymbol{h}$, such that the highest peak is equal in magnitude to the largest spatial extent:

$$\boldsymbol{h}^* = \frac{argmax(s_x, s_y, s_z)}{argmax_i(h_i \in \boldsymbol{h})} \cdot \boldsymbol{h} \quad 3\text{-}15$$

3. Append the spatial extents, $\boldsymbol{s}$, to the end of the rescaled histogram, $\boldsymbol{h}^*$. The result is the full CPH feature.:

$$\boldsymbol{f}_{CPH} = [\boldsymbol{h}^*, \boldsymbol{s}^T] \quad 3\text{-}16$$

The full CPH feature histogram is illustrated in Figure 3-4. The histogram in Figure 3-4 is computed on the same cluster as the histogram depicted in Figure 3-3. For both of these features, M=5 and N=72. The modifications to the baseline shape histogram make the feature invariant to image scale, but allow the vector magnitude to vary between objects of similar shape but different size. It also provides additional discriminative information through the spatial extents that is useful for both recognition and pose estimation.

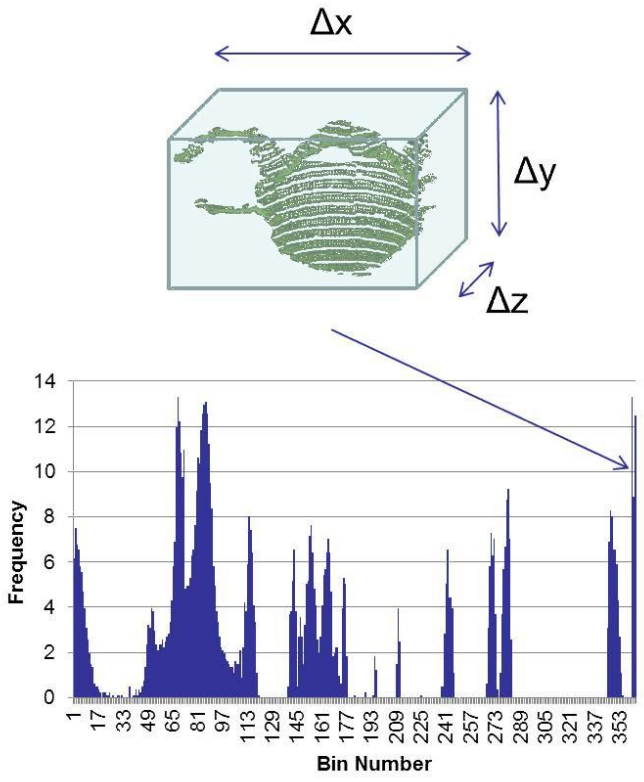


Figure 3-4. Full CPH feature on a coffee cup, showing the addition of spatial extents and rescaling.

Comparing Figure 3-3 and Figure 3-4, it appears that the histogram has been rescaled by about a factor of 10. This is a result of the largest spatial extent being a little more than 10 cm and is purely coincidental. Adding the spatial extents directly encodes information about the actual object size (not scale) into the feature. Rescaling the histogram achieves scale invariance because the spatial extents are computed in physical space which is always invariant to image scale. The resulting feature vector is of dimension $MxN+3$. The vector's direction in this hyperspace is determined by the object's shape, and its magnitude is determined by the object's size. This quality is demonstrated clearly in Figure 3-5.

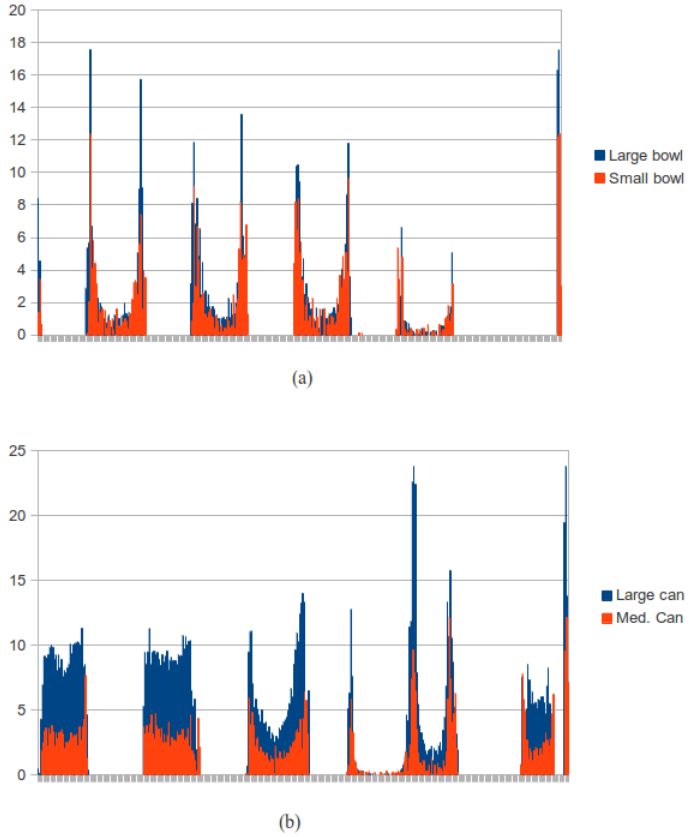


Figure 3-5. Comparison of CPH features for objects of different sizes. (a) Large and medium cans. (b) Large and small bowls.

3.2.3 Comparison to VFH

Figure 3-6 demonstrates some important differences between the VFH and CPH features. The first clear trend is that the VFH feature tends to be much sparser with many bins having a zero value. The sparseness is troubling because there are many bins that carry no information about the object. The CPH features also have some empty bins, the problem is much less pronounced, with the exception of the pliers.

The peaks in the VFH feature are narrow and tall. While several tall peaks are preferable to an overly uniform distribution, if they are too narrow, it will be difficult to match any clusters to each other because a shift of the peaks by only a bin or two will place the histograms too far apart in feature space. This could be a real problem in the presence of noise. The CPH peaks on the other hand, are tall and discriminative without being too narrow. Most of the peaks have what appears to be a normal statistical distribution, which indicates that the feature should hold up well under noisy conditions.

Also, the VFH peaks appear to fall around similar bins regardless of the object. There are four distinct peaks in similar locations among each of the objects. This calls their discriminative ability into question. Ideally, different objects should have very different feature signatures. The VFH features definitely have identifiable similarities. The CPH features, on the other hand, are much more distinct between objects.

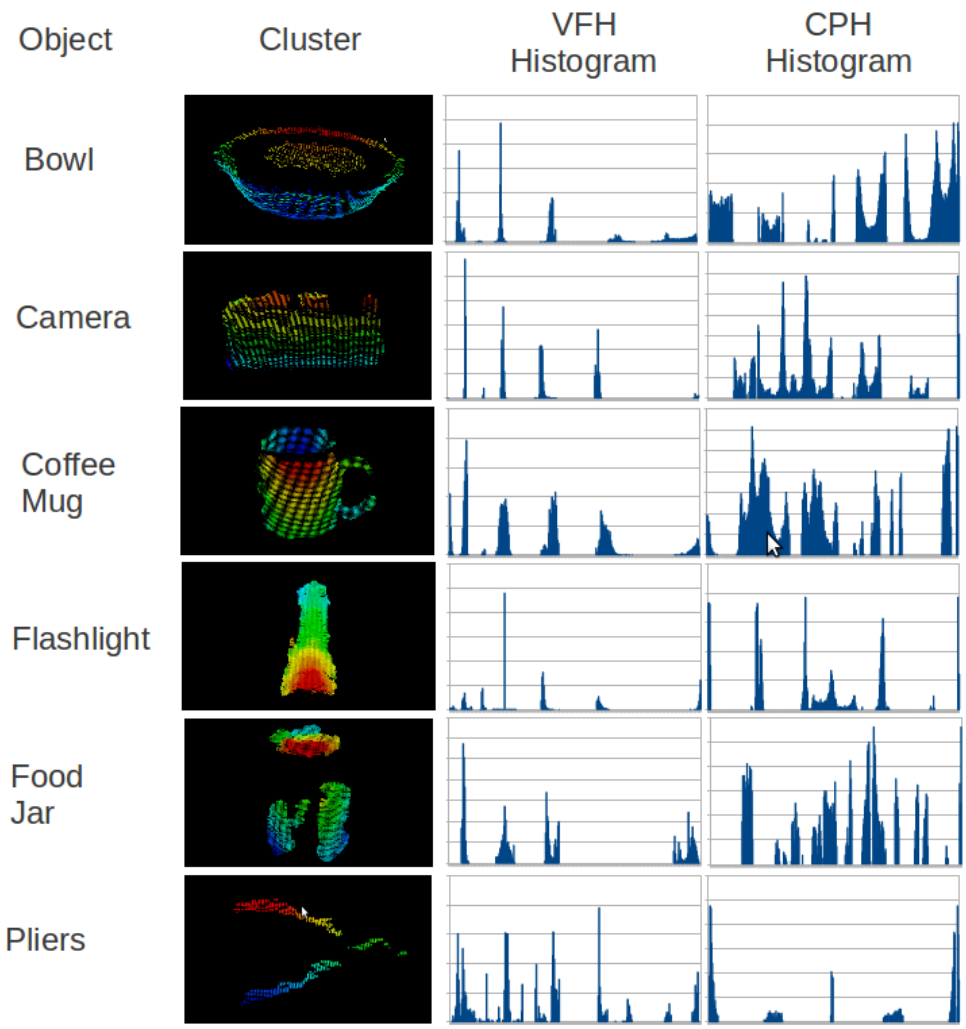


Figure 3-6. Samples of features extracted on different objects.

3.3 DISCUSSION OF METHODS

The object model defined in this chapter is an important advance in NRG's ability to perform manipulation in unstructured environments. By having access to quantitative uncertainty information, the control system and the human operator can make informed decisions about the appropriate level of autonomy for tasks that involve contact interaction. The probabilistic object model also permits reduction in uncertainty by combining multiple measurements though Bayesian filtering. In the current work,

multiple measurements from a single sensor are filtered frame-by-frame. However, the model presented in this chapter paves the way for fusion of information from multiple sensors in the future.

The CPH feature presented in this chapter is a novel approach to the idea of a shape histogram for point cloud cluster recognition. It captures both object shape and size while maintaining invariance to image scale. Chapter 5 presents experiments that demonstrate the discriminative power of CPH and also provides a side-by-side comparison of the performance of CPH, and the VFH feature reviewed in Chapter 2.

Chapter 4: System Implementation

The previous chapter discussed a probabilistic object model and a novel point cloud feature that can be used for cluster recognition. This chapter describes how these methods are implemented and integrated into a complete object recognition and pose estimation system. This system is trained and tested on two different datasets under varying noise conditions. Results are provided for CPH-based recognition, and VFH-based results are provided as a basis of comparison.

4.1 SYSTEM DESCRIPTION

Some of the experiments done in Chapter 5 are performed on datasets that have been stored on disk and some are done on "live" data from a vision sensor. The recognition systems for these two types of experiments are nearly identical, but they do have some subtle and important differences. This chapter presents the overall recognition system and also discusses some of the slight differences in implementation between using live vs pre-collected data.

4.1.1 Classifier training

Training refers to the process of learning different classes from a labeled set of images. In this work, we use a K nearest neighbor (KNN) classifier with K=1. This classifier assigns the class of the nearest training sample in feature space to the input image. In this work, each view of an object is considered to be its own class for classification purposes, so the input image gets assigned the same label and pose of the nearest training sample in feature space. We use a fast implementation of KNN called the Fast Library for Approximate Nearest Neigbors (FLANN) [Muja, 2009]. FLANN requires that the training set be organized into a k-d tree, so training proceeds in two steps:

1. Feature extraction: The point cloud feature must be computed for each image in the training set and the features written to disk. Because there are several thousand training images, this step can take some time and is performed offline.
2. K-d tree construction: K-d tree construction is computationally fast and is performed at runtime prior to any classification. Performing this step at runtime allows the system operator to select which objects the classifier should consider at runtime, allowing fast customization of the recognition system to a particular task.

Once the k-d tree is built, the classifier is ready to accept input images. At test time, input images are run through a recognition pipeline to obtain recognition and pose estimation results.

4.1.2 Recognition pipeline

Between image acquisition and recognition result there is a serial sequence of processing steps referred to as an image pipeline. Figure 4-1 shows the image pipeline used for the experiments in this chapter.

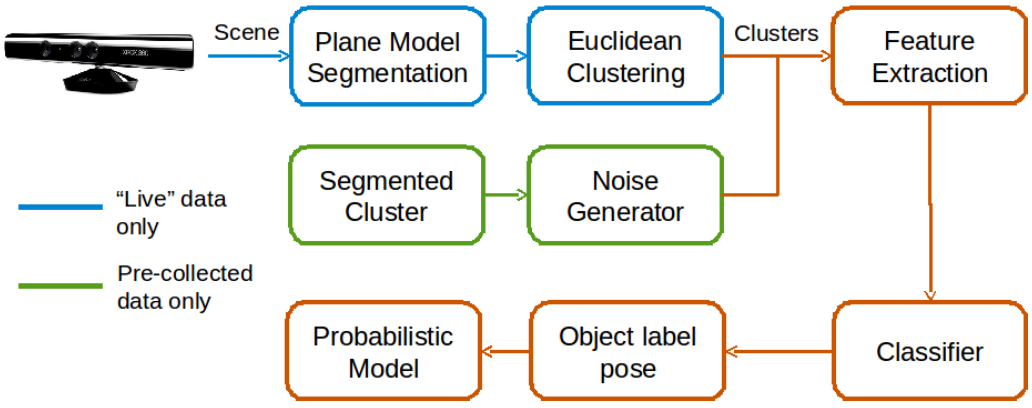


Figure 4-1. Recognition pipeline showing the difference between the "live" and pre-collected preprocessing steps.

Plane model segmentation

The recognition pipeline uses Rusu's Euclidean clustering scheme, previously discussed in Chapter 2. One requirement of this algorithm is that the objects of interest be separated from each other by some threshold distance. In practice, most objects of interest will lie on a common surface (such as the floor of a glovebox). If the common surface is not removed from the point cloud prior to Euclidean clustering, the algorithm will find a single cluster that consists of the surface and anything adjoining it. For this reason, it is necessary to estimate the dominant planar surface in the input image and remove it from downstream processing.

This plane model segmentation is done by RAndom SAmple Consensus (RANSAC). RANSAC is an iterative algorithm that randomly selects sets of three points and fits a plane to those points and then searches to see how many of the remaining points agree fit the model within some threshold. This is repeated iteratively and the model with the best consensus is retained. The algorithm runs as follows:

- Begin with an empty list of model inliers, *I*, and a minimum number of points required for consensus, *d*.
- For *k* iterations,
    - Randomly select three points, fit a planar model, *M* to them, and add them to *I*.
    - For each point $p_i \notin I$, If the point lies within distance *t* of the model, add p_i to *I*.
    - If the number of points in $I > d$:
        - Re-fit the planar model to all points in *I*.
        - Compute fit quality metric, q.
        - If $q > best\_fit$:

- $best\_fit = q$
- $best\_model = M$

- End algorithm

After the RANSAC fit, all points lying within a threshold distance of the planar model are removed from the image. The remaining points are sent downstream to the Euclidean clustering algorithm.

Euclidean Clustering

The recognition pipeline used in this work uses Rusu's Euclidean clustering algorithm presented in section 2.2.1. This is a region growing algorithm that looks for sets of points that are grouped in Euclidean space. Each cluster is considered to belong to the same object. Figure 4-2 shows an example of the plane model segmentation and Euclidean clustering steps in the pipeline.

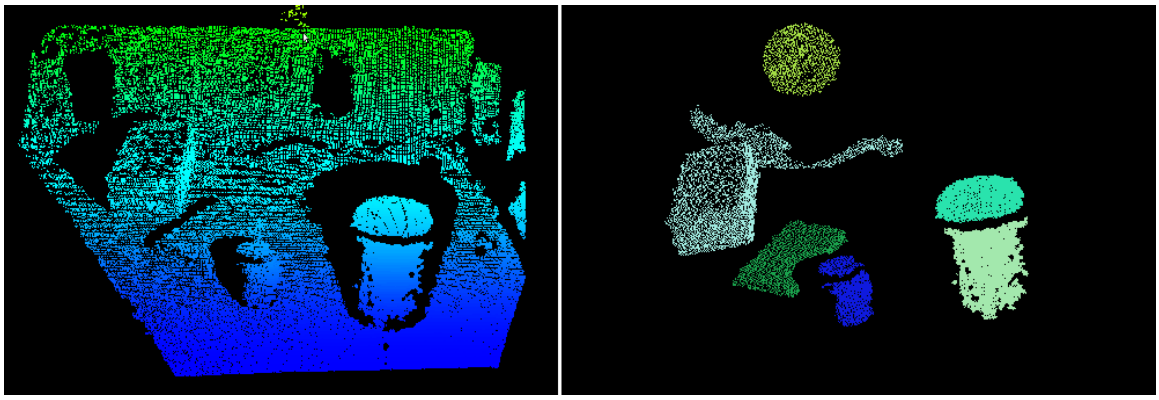


Figure 4-2. Results of plane model segmentation and Euclidean clustering. (left) Raw depth image inside glovebox. (right) Result after removing the dominant planar surface and clustering.

The image in Figure 4-2 was captured in a cluttered environment. The glovebox is typically not in this state of clutter, but this image was used to demonstrate the

capabilities and limitations of the plane model segmentation and Euclidean clustering steps in the recognition pipeline. Several objects are correctly segmented out of the image, including a glove port that is almost indistinguishable in the original image (the yellow circle). However, the importance of the light clutter assumption is demonstrated by the white segment in the left image of Figure 4-2. The white segment is a box that appears to have a strange "tail". That "tail", is a loose glove that is laying inside the glovebox and makes contact with the box. This type of segmentation failure would almost certainly result in a failure of the system to recognize the box.

Another possible failure in Figure 4-2 is the large cylindrical object which is an open container. The segmentation splits the container into two clusters: one representing the outside front surface (light green), and another representing the inside back surface (light blue). This type of failure may, however, not lead to a recognition failure. Because the algorithm is trained on actual camera images segmented by the same algorithm, It is likely that one of these two clusters will still get matched correctly at test time.

Feature Extraction

Every cluster found by the Euclidean clustering step is passed downstream in the pipeline for recognition. The previous steps are pre-processing steps that obtain the set of clusters that the system will attempt to recognize. The first step in the actual recognition portion of the pipeline is to compute the feature to be used for recognition. In the experiments in the next Chapter, the feature is either a VFH feature or a CPH feature. For a complete description of the features and how they are computed, refer to Chapters 2 and 3 respectively.

Classifier

Once features have been extracted on each of the point clusters, the feature is matched to its nearest neighbor in feature space. The classifier uses the chi-square distance between the input feature $\boldsymbol{f}_t$, and each of training features $\boldsymbol{f}_r$.

$$\chi_{\boldsymbol{f}_t,\boldsymbol{f}_r} = \sqrt{\sum_{j=1}^{L} \frac{1}{c_j}\left(f_{tj} - f_{rj}\right)^2} \qquad 4\text{-}1$$

Where c_j denotes the average fraction of the histogram contained in bin j across the entire training set. The classifier implementation used in these experiments is the FLANN library for approximate nearest neighbors [Muja, 2009].

The classifier returns the file name where the feature is stored on disk. The object's label and rotational pose are encoded into the filename and are extracted when the nearest neighbor is located. The file name format is `<label>_<pose>.<file_ext>`. The label and pose are then used to update the probabilistic object model as discussed in section 3.1.2.

Offline data pipeline

The recognition system is designed for "live" imagery collected in real time. However, to validate the system and to collect statistics for the Bayesian model update, it is necessary to classify a large number of images. In the current work, around 125,000 images are classified for system validation and statistics collection. The advantage of using pre-collected data is that it allows the use of a publicly available dataset.This permits meaningful validation and peer review because results can be validated and examined by anyone on exactly the same data. For these reasons, we provide a description of an alternative pre-processing pipeline for use with data that have been collected offline. These steps are shown in green in Figure 4-1.

The biggest difference between the "live" and "pre-collected" pipelines is that we assume that offline data has already been properly segmented and that we are working with only a single point cluster. The plane model segmentation and Euclidean clustering steps are therefore omitted. One advantage of using spatial segmentation as opposed to temporal background subtraction is that it can be done on individual images. If a dataset were provided that was not already segmented, the plane model segmentation and Euclidean clustering could be re-added to the pipeline without the need to develop a temporal background model.

One challenge that arises when using multi-view datasets is that typically only one image is provided for each view. The dataset would grow unnecessarily large if tens or hundreds of images were collected at each view in the dataset. This creates a challenge when trying to separate samples into a training set and a test set. We cannot consider our results valid if we train and test on the same data, but any division of the data into training and testing groups reduces the pose resolution in the training set. Also, testing would occur only on views that aren't present in the training set. This is not an accurate representation of the live system where you would train on all 120 views and would fully expect to find similar views in the test set.

For example, assume a data set contains 120 images of an object collected at three degree intervals. If 80 images are reserved for training, the resolution of the classifier increases from 3 to 4.5 degrees. Also, the test images will be at least three degrees away from the nearest training instance and thus the accuracy of the pose estimation will suffer in a completely artificial way. To address this issue, test images are artificially generated by adding Gaussian noise to the *z* (depth) channel of the training images. In addition to providing a more realisitic test, this also allows the robustness of the system to noise to be evaluated. Because the target application involves objects with surface properties that

are likely to be noisy due to variations in reflectivity and infrared emittance, robustness to noise is an important consideration.

4.3 SOFTWARE IMPLEMENTATION

The previous section discussed the recognition system at a high\ level to provide an overview of the pipeline and the basic methodology. This section gets into some finer implementation details and covers how the system is implemented in the Point Cloud Library (PCL), and the Robot Operating System (ROS). The high-level ROS implementation is covered first, as it closely reflects the pipeline discussed above. Following the system overview, each of the ROS nodes is discussed in more detail, including how various elements of PCL are used to implement the various steps of the pipeline discussed in 4.1.2.

Figure 4-3 shows the ROS nodes in the recognition system and how they interact with each other. Black arrows indicate topic subscriptions and point from the publisher to the subscriber. Green arrows indicate that during object recognition, at least one service call is made between nodes. The arrow points from the client node to the server node. All of the service calls use custom service definitions. The .msg and .srv files are provided in Appendix A. It may be worthwhile to review the basic ROS ideas and terminology from Chapter 1.

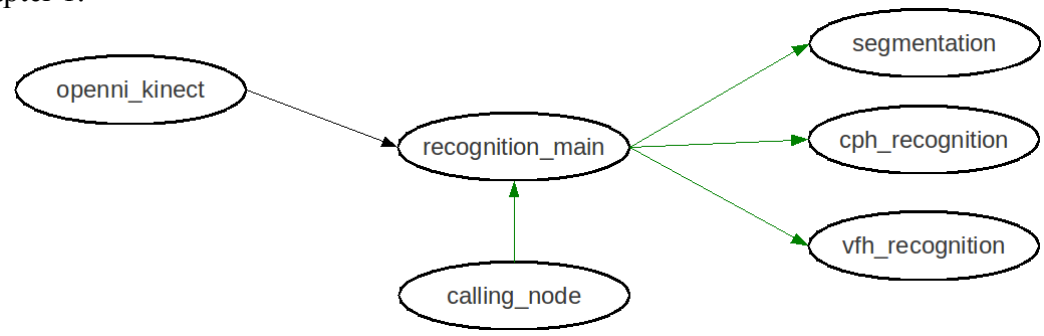


Figure 4-3. Graph of the ROS recognition system implementation.

The `/calling_node` is any node uses the recognition system. In practice, `/calling_node` will be a robot application that requires object recognition or a node generated from the command line with `rosservice call`. The remaining nodes and their services are discussed below. Particular attention is given to how PCL classes and functions are used to implement the pipeline discussed in the previous section. For additional implementation detail, the code used to generate the node executables is provided in Appendix A.

openni_kinect

The openni_kinect node is shown as a single node in Figure 4-4 for clarity. In reality is a large system of nodes created by the ROS launch file `openni.launch` from the openni_launch stack in ROS Fuerte. The openni_launch nodes publish the point cloud data and rgb image data from the Kinect on several topics. The topic to which the recognition_main node subscribes is /camera/depth_registered/points. This topic is published as a `sensor_msgs::PointCloud2` message, the standard format for point cloud data in ROS.

recognition_main

The main node that manages the recognition system is called recognition_main. The main node offers two services, `/live_test`, and `/run_test`. The services use custom defined service calls. Table 4-1 shows each node from Figure 4-4, and explains the service definitions in detail.

The `/live_test` service is called to perform a series of tests to evaluate the classifier's performance on live data from the Kinect. The service request contains the following information:

- The ground truth label of the object being tested

- The feature to use for classification (VFH or CPH)
- The number of views to examine
- The number of images to take at each view

The service response provides performance data from the test. It contains the following information:

- The raw accuracy of the classifier
- The average standard error in the pose estimate
- The filtered accuracy of the classifier
- The filtered error of the pose estimate

For each image at each view, the segmentation node performs plane model segmentation and Euclidean clustering as described in Section 4.1.2. The service request includes a `sensor_msgs::PointCloud2` array that contains each cluster found in the scene. Each of these clusters is passed in a service request to either the cph_recognition_node or vfh_recognition node depending upon the desired feature type. The recognition result is passed back in the service response. The response consists of the object label and rotational pose of the nearest training sample in feature space. The results are filtered frame over frame with the current label and pose estimate distributions stored within the recognition_main node.

The `/run_test` service is used for testing on offline datasets. It is used to recognize many images at one time and return aggregated results. The service request consists of the name of the ground truth object, the feature to use for recognition (VFH or CPH), the level of noise to use in the test, and the number of noisy samples that should be run for each image in the original test set. The service response contains the exact same information as it did for the `/live_test` service. The raw results from the `/run_test` service are used to collect statistics for the Bayesian filters and to

characterize the classifer performance. Therefore until the statistics have been collected, the `/run_test` filtered results will not be meaningful.

When `/run_test` is called, the test images are loaded from .pcd files stored on disk. Recognition and statistical filtering are done in exactly the same manner as in the `/live_test` service, except the clusters loaded from file are passed directly to cph_recognition_node or vfh_recognition node without first calling the segmentation service on the segmentation node. This pre-processing is unnecessary since the point clouds stored on disk are already segmented.

All of the statistical filtering happens in the recognition_main node. To do the statistical filtering, the node needs $P(C|z)$ for each class. This data is stored in a table where entry (m,n) is the a priori probability that an unknown object is of class m given recognition result n. The node also needs the per-class pose standard deviations. These are stored in a separate table with a single column. All of this data is loaded at runtime from a file called classes.list. Each object in the training set must have a row entry in classes.list. Each row contains the object name, the pose standard deviation, and the $P(C|z)$ values. An example of a classes.list file is provided in Appendix B.

It is not uncommon that some entries in $P(C|z)$ are zero when the statistics are collected. However, zeros in $P(C|z)$ can cause unnecessary failures in the improbable event that a misclassification occurs that never occurred during statistics collection. This is because propagating a zero through the Bayesian update may set the probability of the true class to zero. When this happens, no amount of additional evidence for the true class will ever raise the probability back above zero. To prevent this, if any entry in $P(C|z)$ is less than .001, it is replaced with .001. This assures that very rare individual failures don't cause complete failure of the algorithm.

segmentation

The segmentation node offers a single service. The service request contains a `sensor_msgs::PointCloud2` representing an entire scene in which objects of interest rest on a common planar surface. The service response is a `sensor_msgs::PointCloud2` array in which each entry is a point cluster belonging to a single object. The service request also takes several filter parameter arguments that are detailed in Table 4-1. These parameters allow a region of space to be defined outside of which no clusters are processed. This prevents additional unnecessary processing on clusters that are located outside the region of interest. The segmentation node implementation relies heavily on functions from PCL. When the segmentation service is called, the scene is processed as follows:

- The PointCloud2 object from the service request is converted from a ROS `sensor_msgs::PointCloud2` into a PCL `pcl::PointCloud<PointXYZ>` using the PCL function `pcl::fromROSMsg()`.
- The scene is filtered using a voxel grid filter with a leaf size of 1 mm (`pcl::VoxelGrid<pcl::PointXYZ>`).This divides space into 1 mm cubic regions and allows only a single point per region. In the current work, where only a single sensor is used and the distance from the camera is constrained by the glovebox, this filter is probably unnecessary. With the small leaf size, few if any points are actually filtered. The filter is retained however, because if an additional sensor were used, the clouds would grow artificially dense where the images overlap.
- All points out of the region of interest defined in the service request are discarded.

- Plane model segmentation is performed using the RANSAC algorithm described in 4.1.2 (`pcl::SACSegmentation<pcl::PointXYZ>`) The distance threshold, t is 2 cm and the maximum number of iterations is 100.
- Euclidean segmentation is performed using Rusu's method described in 2.2.1. The method is implemented in PCL as `pcl::EuclideanClusterExtraction<pcl::PointXYZ>`. The threshold distance d_{th} is 2 cm. The minimum and maximum number of points per cluster are set according to the segmentation service request.
- The resulting clusters are of `type pcl::PointCloud<pcl::PointXYZ>`. They are converted to `sensor_msgs::PointCloud2` types `using pcl::toROSMsg()` and pushed onto a vector of point cluster that is return through the service response.

vfh_recognition and cph_recognition

The vfh_recognition node and the cph_recognition nodes are nearly identical in their implementation and function. Each node loads the training samples at runtime. Recall that the features are extracted from the training samples offline. The node need only load the pre-computed features. This operation takes no perceivable time when the node is created.

Each node offers a single service, either `/vfh_recognition` or `/cph_recognition`. Each of these services takes the same service definition. The point cluster is passed in as a `sensor_msgs::PointCloud2`, and a maximum chi-square distance threshold is passed in as well. If no training sample lies within this value,

no result is returned. The two values in the service response are a string with the object label and a custom-defined pose message that contains the 4D pose. When the recognition service is called, processing proceeds as follows:

- The centroid of the cluster is computed with `pcl::compute3DCentroid()`. The centroid x,y, and z values are stored in the pose of the service response.
- In the vfh_recognition_node only, the surface normals are estimated using `pcl::NormalEstimation<pcl::PointXYZ>`.
- The feature (either VFH or CPH) is extracted. The VFH feature is implemented in PCL as `pcl::VFHEstimation<pcl::PointXYZ, pcl::Normal, pcl::VFHSignature308>`. The feature is passed to the FLANN nearest-neighbor classifier, which returns the filename of the nearest training feature.
- The filename is parsed to obtain the object label and rotational pose which are written to the service response.

The code that implements all of the above nodes was developed for the current work and is available on the ROS-Industrial repository and in Appendix A. The CPH implementation is designed to be a direct replacement in any code that currently uses another PCL feature type. The functional conventions are retained with functions like `setInputCloud()` and `compute()`, and the CPH object works on PCL PointCloud data types. The CPH feature is parameterized by the number of bins in the vertical and azimuthal directions, and the CPH class constructor allows these to be customized. However, 5 vertical bins and 72 azimuthal bins are used throughout. These choices give high resolution in the angular pose and keep the length of the feature similar to VFH for valid comparison.

Table 4-1. Nodes and the services they offer.

Node	Services Offered	Description	Service definition
recognition_main	run_data	Performs recognition and collects statistics on offline datasets	run_data.srv
	get_object_list	Processes a scene and returns a list of recognized objects	nrg_recognition.srv
segmentation	segmentation	Performs plane model segmentation and Euclidean clustering. Returns list of object clusters	segmentation.srv
vfh_recognition_node	vfh_recognition	Recognizes an object cluster and estimates the pose with VFH features.	recognition.srv
cph_recognition_node	cph_recognition	Recognizes an object cluster and estimates the pose with CPH features.	recognition.srv

4.3 IMPLEMENTATION REVIEW AND DISCUSSION

This chapter provided an overview of the image processing pipline implemented in the current work. Section 4.2 described how the pipeline takes a point cloud of an entire scene, locates objects through spatial filtering and clustering, and then recognizes objects and estimates their pose by comparing them to known training samples by nearest-neighbor matching. The pipeline makes heavy use of PCL functions for type conversions, filtering operations, and segmentation. This is done in a modular way, so that the methods and algorithms within the pipeline are interchangeable for alternatives. This makes the implementation flexible, allowing for fast integration of future research efforts.

The modularity is enhanced by the use of ROS for the high level implementation. The different portions of the pipeline are implemented as individual ROS nodes. Not only does this make the use of alternative methods easy, it also assures a fair comparison between CPH and VFH by ensuring that only the actual recognition portion of the pipeline changes without inadvertently making other small changes that may affect the overall results. ROS integration also allows the recognition system to plug seamlessly into a robot's control system for the target application of unstructured robotic manipulation.

Chapter 5: Experiments

Chapters 3 and 4 describe two variations on a complete object recognition and pose estimation methodology. The two variations are identical except for the actual feature used for training and classification. The first feature is the Viewpoint Feature Histogram (VFH). VFH is a surface normal based feature built into PCL that is specifically designed for simultaneous recognition and pose estimation. The second feature is the Cylindrical Projection Histogram (CPH). CPH is novel feature also specifically developed for simultaneous recognition and pose estimation. CPH is a fundamentally different type of feature than VFH. Whereas VFH is a histogram built from surface properties, CPH is based on a shape histogram derived directly from the spatial point distribution. This chapter presents experimental results that demonstrate the superiority of CPH in the target application.

Chapters 3 and 4 also describe a method for using probabilistic modeling techniques to reduce uncertainty in the object model by statistically combining several measurements through Bayesian filtering. The modeling and filtering methods are independent of the type of feature used for recognition and pose estimation. However the performance of the statistical update depends heavily on the quality of the classifier's statistics. For example, in the case of a completely random classifier, no amount of statistical filtering will improve the recognition rate. Likewise, in the case of a perfect classifier, no amount of statistical filtering will improve the recognition rate above 100%. Therefore the modeling results presented in this chapter are presented for each of the two classifier variations (VFH and CPH) independently. The experiments presented in this chapter show that probabilistic modeling improves overall performance, but the improvement is marginal.

The recognition system presented in this document addresses the problem of instance-level recognition. This problem assumes that all members of an object class have identical appearance. The majority of the experimental results here are instance-level results However, to fully exercise the CPH feature, its performance is evaluated on the category-level problem where the assumption of identical appearance is removed. The results show that VFH does not perform well at the category level. CPH shows some promise at the category level, but requires further investigation.

5.1 DATASETS

The experiments in this chapter are performed on two datasets. The first is a subset of images from the RGB-D dataset [Lai, 2010]. The second dataset was collected as part of the current research and is intended to be representative of objects typically found inside gloveboxes for nuclear materials processing at Los Alamos National Laboratory (LANL). Each is discussed in detail below. The subset of the Lai dataset will be referred to as the RGB-D dataset, and glovebox-specific dataset will be referred to as the LANL dataset for the remainder of this document.

5.1.1 RGB-D dataset

The complete RGB-D dataset developed by Lai *et al.* contains 300 instances of objects in 51 categories. For each object, segmented color and depth images are provided in the form of short 30 Hz videos of the objects rotating through 360 degrees on a turntable. Figure 5-1 shows some example images form the RGB-D dataset. The current work uses only a subset of this data for two reasons.

1. Classifier performance depends on dataset size [Deng, 2010]. Because the number of objects in a glovebox task is much smaller than the 300, we limit the number of objects to a similar size as the LANL dataset.

2. The images in the complete RGB-D dataset are not annotated with the pose at collection time. The pose estimation methodology used in the pipeline from Chapter 4 requires that every training image be annotated with the pose. This is a laborious process to do after the fact.

Seven instances from the full RGB-D dataset were selected for inclusion in our RGB-D dataset. These objects were selected because they were the most geometrically similar to objects in the LANL database. Only the depth imagery is used; the RGB imagery is discarded. Figure 5-2 shows the seven objects included in our RGB-D dataset. Only images the leftmost object in Figure 5-2 are used for all instance-level experiments. The other objects in Figure 5-2 are used only for the category-level experiments.

Figure 5-1. Example images from Lai's RGB-D dataset [2011].

Figure 5-2. Objects from the RGB-D dataset used for the experiments in this chapter. (a) Bowl. (b) Camera. (c) Coffee mug. (d) Dry battery. (e) Flashlight. (f) Food jar. (g) Pliers.

These seven objects were selected because most of them have little surface texture and are therefore representative of objects in our target application. Because we assume that objects can be readily segmented from the scene, we use the segmented depth images provided by Lai et al. For category-level recognition, we use the same seven object classes, but we use all available instances and test by Leave-One-Out Cross Validation (LOO-CV).

Because we would like to characterize the pose estimation performance, the training samples must be annotated with pose information. While Lai's RGB-D dataset provides images from around 600 different viewpoints per instance, it does not provide any ground truth information about the pose of the objects relative to the camera. The method of acquisition is to capture frames from video while the object is rotated on a turntable through approximately one rotation. This is done at three different elevation

angles, although we use only a single elevation. To estimate the ground truth poses in the dataset, we manually annotate the pose of each frame by the following procedure.

1. Manually locate frame numbers corresponding to 0°, 45°, 90°, 135°, 180°, 225°, 270°, and 315°
2. Linearly interpolate frame numbers between the manually annotated frames in step 1.

This is done only on the images from the lowest elevation because our pose estimation is limited to a single rotational degree of freedom. The complete RGB-D dataset can be obtained from the authors of [Lai, 2011]..

5.1.2 LANL dataset

To evaluate our methods on a relevant dataset, we have created our own pose-annotated dataset for LANL glovebox applications. The objects in the dataset were determined through discussions with engineers at Los Alamos National Laboratory. Each object is an item that a flexible glovebox automation system may need to interact with. In addition to testing our methods on relevant objects, this dataset provides some challenging objects for any recognition algorithm. Several of the objects have almost no texture and are made from matte-finished steel which is moderately reflective. This data is therefore useful to anyone wishing to recognize these types of objects, so the dataset has been made publicly available by contacting the author (brian.erick.oneil@gmail.com). Figure 5-3 shows the objects in the dataset.

Figure 5-3. Objects in the LANL dataset. (a) broom. (b) t-fitting. (c) lathe tool. (d) large can. (e) large bowl. (f) medium can. (g) scale. (h) small bowl. (i) small can. (j) sealing tape.

Figure 5-4 shows examples of data collected for the LANL dataset. The clusters shown in Figure 5-4 indicate the difficulty of the objects in the dataset. The shiny metallic objects almost all give noisy and incomplete point clusters. The incompleteness can be explained in the following way. Where there are highlights on the objects due to specularity, the depth imager does not register points. This leads to discontinuities in the point cluster that interfere with the Euclidean segmentation algorithm. Instead of a single cluster representing the entire object, parts of the object end up in separate clusters. Because the recognition system requires a one-to-one object/cluster correspondence, points detected on the object are discarded as belonging to different objects.

The T-fitting and the small can are missing large patches due to missing data on some reflective surfaces. However, the missing data is reasonably consistent, so reasonable recognition performance can still be achieved. As long as clusters collected at test time resemble those collected for training, correct recognitions are possible even if with incomplete data.

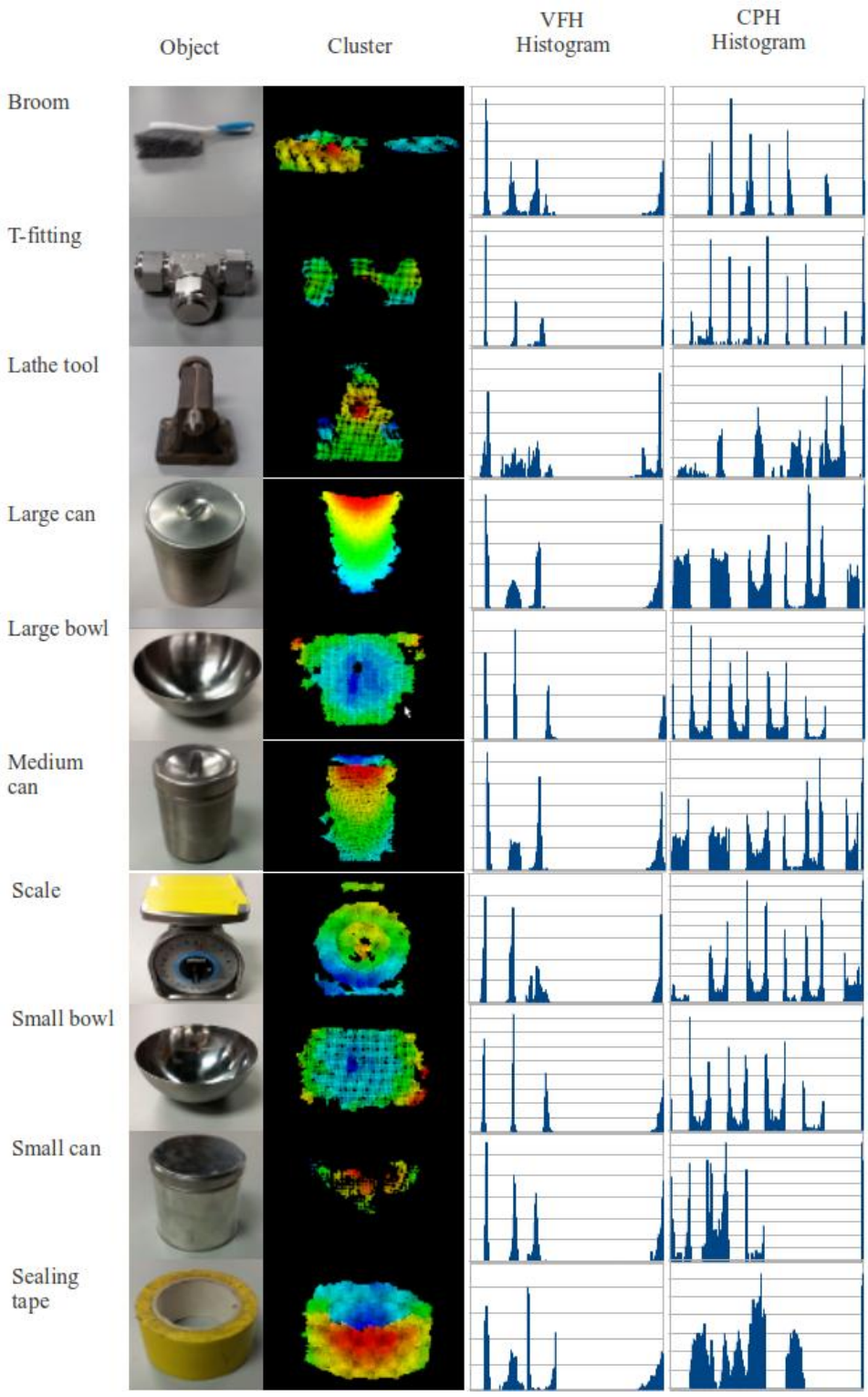


Figure 5-4. Samples of CPH features extracted on different objects.

Many of the same observations can be made in Figure 4-2 can also be made in this data. The VFH histograms tend to be sparse and clustered around a few distinct peaks that do not vary much between objects. The CPH histograms are well populated, with distinct structures that vary among the objects.

In order to facilitate the easy *in situ* collection of pose-annoated object images, we assembled a custom data collection system. The data collection hardware is a servo-driven pan/tilt table. The table's servos are controlled by an Arduino microcontroller. The table is shown in Figure 5-5. Full specifications on the data collection system and the code required to run the system are provided in Appendix C.

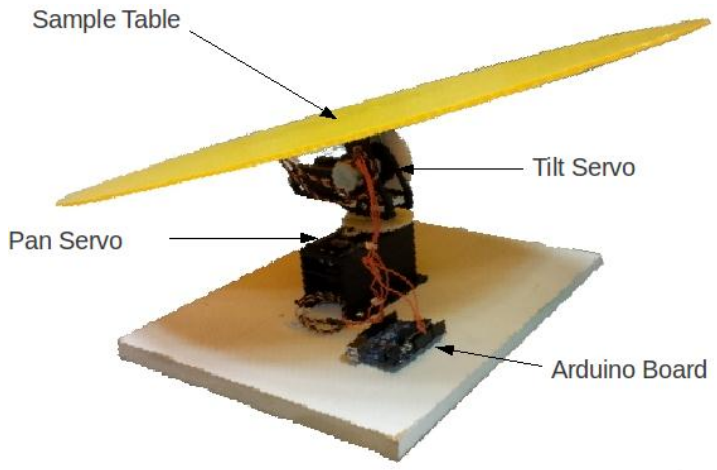


Figure 5-5. Servo-driven pan/tilt table for data collection

The data collection software is implemented in ROS. A lightweight version of the base ROS libraries called rosserial_arduino is installed on the Arduino microcontroller. A full ROS system runs on a Linux PC that includes a rosserial_python node that looks for any nodes running on serial-connected devices and includes those nodes in the ROS

system running on the PC. Figure 5-6 is a graphical depiction of the ROS data collection software.

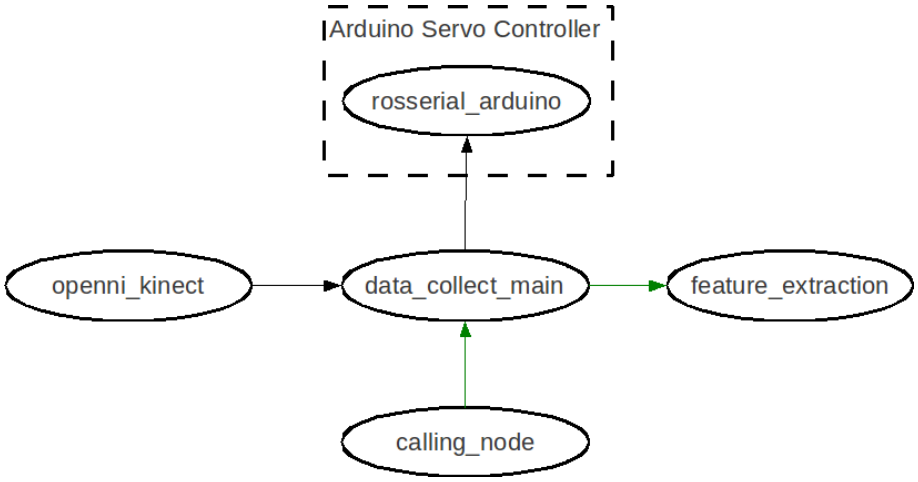


Figure 5-6. ROS data collection software diagram.

The ROS system allows easy integration of image acquisition and analysis software with the pan/tilt control software. The rosserial_arduino node subscribes to pan and tilt command topics. The main data collection node can then set the servo position prior to collecting an image. This allows the system to automatically collect multi-viewpoint data as specified by user-supplied parameters. The following is a description of each node in the data collection system. The .srv definition files for the service requests are provided in Appendix A.

openni_kinect

The openni_kinect node is the same set of nodes used in the recognition pipeline. It publishes the point cloud data for the entire scene.

rosserial_arduino

The rosserial_arduino node runs the rosserial libraries for arduino. It subscribes to the pan/tilt servo command topics given in degrees. When new commands are received, it converts them to pulse width commands and controls the two servo motors on the pan/tilt mechanism to achieve the desired object view.

data_collect_main

The data_collect_main node accepts a service request from the calling node. The request contains the label associated with the object on the pan/tilt table, and the angular resolution desired in the training set. When a training service request is made, data_collect_main does the following:

- For each requested viewpoint:
    - Publish servo commands to the pan/tilt mechanism.
    - Pass most recent depth image from Kinect to the feature_extraction node for processing.
- End processing when all viewpoint images complete.

feature_extraction

The feature_extraction node accepts a service request from the data_collect_main node. The entire scene from the Kinect is passed in the service request. The feature extraction node performs plane model segmentation and Euclidean cluster extraction within a small bounding box around the pan/tilt table to isolate the portion of the scene that belongs to the training object. This cluster is written to disk as a .pcd file following the naming convention in 4.1.2. It then extracts a VFH feature and a CPH feature and writes these to file as well. The original point cloud is not necessary for the recognition

algorithm. However, retention of the raw data allows fair comparison of future methods, and the raw clouds can be used when visualizing data.

5.2 RECOGNITION AND POSE ESTIMATION EXPERIMENTS

Recognition, pose estimation, and probabilistic filtering experiments are performed on both datasets. There is one significant difference between the experiments performed on the RGB-D dataset and those performed on the LANL dataset. For the RGB-D dataset, the images used in the test phase are the same as the images used in the training phase, but Gaussian noise added to the z channel of each point in the cluster. Two different noise levels are tested. For the LANL dataset, the imaging system and the original objects are available for live testing. Therefore no artificial noise is added; frames are grabbed directly from the Kinect for processing.

PCL is a popular package in for processing depth images, and its inclusion in ROS makes it the *de facto* standard for point cloud processing in robotics research. For this reason, CPH is compared side-by-side with VFH, the PCL feature that is designed to address the same problem. The experiments presented in this section address the following questions:

- Recognition performance
    - Rate: What fraction of images are correctly classified?
    - Confusion: How do the failures vary among classes?
    - Receiver Operating Characteristics: How well do the classifiers reject false positives?
    - Susceptibility to noise: Does the recognition performance degrade with increasing image noise?
- Pose estimation performance

- Standard deviation: How accurate is the estimated pose compared to the ground truth?
- Susceptibility to noise: Does the pose estimation performance degrade with increasing image noise?

- Computation time: Does the feature offer real-time performance?

5.2.1 RGB-D dataset results

The first set of experiments evaluates the classifier's raw performance without any statistical filtering. For each of the 1254 original images in the dataset, 100 noisy samples are generated and classified giving a total of 125,400 test images. For each object, the fraction of samples assigned to each class is computed to generate a confusion matrix, and to provide prior statistics that can be used for statistical filtering in later experiments.

Data are collected in this manner for both the VFH feature and the CPH feature. The methods used for classification between the two features are identical with the exception of feature extracted from the cluster. The first set of experiments is done with Gaussian noise in the z channel of $\sigma = 1\,mm$. Tables 5-1 and 5-2 show the confusion matrices for the VFH-based classifier and CPH-based classifier respectively.

In Tables 5-1 and 5-2, the left column is the ground set truth object, and the numbers in the table show the fraction of instances of that object classified as the object listed in the top row. These confusion matrices give a concise description of the classifier performance and also provide some insight into what kind of failures one can expect. Both tables show that their respective features perform well on the object recognition task, however, the performance of CPH is dramatically better, correctly classifying all 125,400 noisy images in the test set.

Table 5-1. Confusion matrix for VFH feature on RGB-D dataset, σ_{noise} = 1 mm.

	Bowl	Camera	Coffee Mug	Battery	Flashlight	Jar	Pliers
Bowl	1.000	0.000	0.000	0.000	0.000	0.000	0.000
Camera	0.000	0.805	0.114	0.041	0.030	0.010	0.000
Coffee Mug	0.000	0.000	0.760	0.000	0.061	0.179	0.000
Battery	0.000	0.004	0.001	0.971	0.024	0.000	0.000
Flashlight	0.000	0.009	0.048	0.114	0.829	0.000	0.000
jar	0.000	0.041	0.003	0.000	0.000	0.956	0.000
Pliers	0.000	0.000	0.011	0.000	0.000	0.006	0.983

Table 5-2. Confusion matrix for CPH feature on RGB-D dataset, σ_{noise} = 1 mm

	Bowl	Camera	Coffee Mug	Battery	Flashlight	Jar	Pliers
Bowl	1.000	0.000	0.000	0.000	0.000	0.000	0.000
Camera	0.000	1.000	0.000	0.000	0.000	0.000	0.000
Coffee Mug	0.000	0.000	1.000	0.000	0.000	0.000	0.000
Battery	0.000	0.000	0.000	1.000	0.000	0.000	0.000
Flashlight	0.000	0.000	0.000	0.000	1.000	0.000	0.000
jar	0.000	0.000	0.000	0.000	0.000	1.000	0.000
Pliers	0.000	0.000	0.000	0.000	0.000	0.000	1.000

The VFH feature's performance is the best on the bowl and the pliers. The bowl images have more point on average than images from the other objects, so it is possible that the normal estimation is better because there is more data on which to compute the normal. The pliers, however, have fewer points than most of the other object. They do have significantly more surface texture than the remaining objects, and because the VFH feature is a surface descriptor, it stands to reason that it would perform better on objects with more surface texture.

In addition to the recognition statistics, the standard deviation from the true pose is computed by Equation 5-1 where θ_c is the pose determined by the classifier and θ_t is the ground truth pose.

$$\sigma = \sqrt{\sum_{i=1}^{N} \frac{1}{N} (\theta_{ci} - \theta_{ti})^2} \qquad 5\text{-}1$$

Table 5-3 gives the standard deviation in the pose estimate for each object. No data is given for the bowl because it is axially symmetric and the pose estimate is meaningless. The same is true in theory for the jar. However, as shown in the sample image in Figure 4-2, The imaging system fails to capture data on the exposed glass of the jar, and only the label is visible in the depth image. Because of the optical properties of the material, descriptive pose estimation information is present in the image even though the object itself is axially symmetric.

Table 5-3. Standard deviation in pose estimate (measured from ground truth) for VFH and CPH classifiers, σ_{noise} = 1 mm.

Object	VFH σ (degrees)	CPH σ (degrees)
Bowl	NA	NA
Camera	81.7	10.5
Coffee Mug	104.0	10.8
Battery	104.0	10.8
Flashlight	69.0	11.7
jar	55.3	10.2
Pliers	40.0	10.2

Because the object recognition and pose estimation are both performed by the nearest neighbor classifier, we expect that pose estimation performance would be closely correlated with object recognition performance. The feature type that is most consistent between the training set and the test set will give the best recognition result and the best

pose estimation result. This is reflected in the results presented in Table 5-3. The error in the pose estimate from the CPH feature is dramatically lower and appears to depend less upon the object than does the error from the VFH feature.

The poor performance of the VFH feature is surprising. Rusu [2010] does not report standard deviation, but instead reports that 98.52% of test images match to the correct pose. Those results are reported on a much larger dataset (60 objects), and with much greater resolution in the training set. Nevertheless, in the current work we fail to replicate their results on the RGB-D dataset. The current approach uses a recognition pipeline very similar to that of [Rusu, 2010] to permit a valid comparison between features. Because the RGB-D dataset contains pre-segmented images, the results reported here are independent of any minor differences in the preprocessing steps in the pipeline.

The most likely cause of the VFH performance discrepancy is that in this work, we artificially generate the test set by adding Gaussian noise to the training set. It is not clear in Rusu how their dataset is separated into training and test data. Even though Rusu's results could not be replicated, the experiments in this chapter demonstrate that CPH-based recognition outperforms even Rusu's VFH performance, albeit on a much smaller dataset.

The same experiments were conducted at a higher noise level of σ=*5 mm.* The actual noise in the Kinect depth imagery depends upon the location of the pixel in the field of view. It is non-Gaussian and difficult to model. Testing on two noise levels is an attempt to bracket the performance in the presence of noise. Tables 5-4 and 5-5 show confusion matrices for both features at the higher noise level, and Table 5-6 shows the pose estimation results at the of σ=*5 mm* noise level. The recognition results at the higher noise level show that VFH recognition performance degrades in the presence of additional noise.

Table 5-4. Confusion matrix for VFH feature on RGB-D dataset, σ_{noise} = 5 mm.

	Bowl	Camera	Coffee Mug	Battery	Flashlight	Jar	Pliers
Bowl	1.000	0.000	0.000	0.000	0.000	0.000	0.000
Camera	0.000	0.629	0.209	0.054	0.101	0.005	0.002
Coffee Mug	0.000	0.000	1.000	0.000	0.000	0.000	0.000
Battery	0.000	0.002	0.000	0.986	0.012	0.000	0.000
Flashlight	0.000	0.014	0.122	0.238	0.626	0.000	0.000
jar	0.000	0.101	0.050	0.000	0.000	0.849	0.000
Pliers	0.000	0.000	0.034	0.011	0.006	0.000	0.949

Table 5-5. Confusion matrix for CPH feature on RGB-D dataset, σ_{noise} = 5 mm.

	Bowl	Camera	Coffee Mug	Battery	Flashlight	Jar	Pliers
Bowl	1.000	0.000	0.000	0.000	0.000	0.000	0.000
Camera	0.000	1.000	0.000	0.000	0.000	0.000	0.000
Coffee Mug	0.000	0.000	1.000	0.000	0.000	0.000	0.000
Battery	0.000	0.000	0.000	1.000	0.000	0.000	0.000
Flashlight	0.000	0.000	0.000	0.000	1.000	0.000	0.000
jar	0.000	0.000	0.000	0.000	0.000	1.000	0.000
Pliers	0.000	0.000	0.000	0.000	0.000	0.000	1.000

Table 5-6. Standard deviation in pose estimate (measured from ground truth) for VFH and CPH classifiers, σ_{noise} = 5 mm.

Object	VFH σ (degrees)	CPH σ (degrees)
Bowl	NA	NA
Camera	96.6	10.5
Coffee Mug	100.1	10.9
Battery	110.9	13.8
Flashlight	89.9	11.7
jar	69.5	10.2
Pliers	72.6	10.7

The overall recognition rate for the VFH-based classifier drops from 90.1% to 86.3%. The worst case recognition rate among all objects drops from 81% to 62.5%. While this performance is unacceptable for most applications, the noise level is arbitrarily high. These results demonstrate the level of robustness to noise, not the expected performance in a real system. The CPH-based classifier maintains a 100% recognition rate even at the more extreme noise level.

The pose estimation results are significantly degraded in the presence of additional noise. The standard pose error increased by an average of 20.6° for the VFH classifier and 0.6° for the CPH classifier. At either noise level, the VFH pose estimates cannot be used reliably for object grasping. The CPH pose estimates are excellent at both noise levels, although they do exceed the ~3° angular resolution in the training data.

5.2.2 LANL dataset results

Performance of the VFH and CPH classifiers is evaluated on the LANL dataset in a similar manner as the RGB-D dataset. However, the results for the LANL dataset more accurately represent the performance expected on a deployed system. Whereas test images for the RGB-D dataset were artificially produced by the addition of Gaussian noise, the fact that we have the objects from the LANL dataset in hand allows us to test the classifier on live data, using the full system implementation described in Chapter 4. The results in this section are over 100 images from each of 120 object views for a total of 12,000 images per object.

Tables 5-7 and 5-8 show the results for the raw classifiers presented as confusion matrices. As was the case with the low-noise results on the RGB-D dataset, the overall performance of both classifiers is good. The overall recognition performance of the VFH classifier is 89.6%, with the CPH classifier performing significantly better at 97.2%. Pose

estimation error on the LANL dataset is done by the same methods as for the RGB-D dataset. Out of the ten objects in the dataset, six are axially symmetric about the vertical axis, so pose estimation errors are only reported for the remaining four objects. The results are presented in Table 5-9.

Table 5-7. Confusion matrix for VFH classifier on LANL dataset.

	Broom	Fitting	Lathe Knob	Large Bowl	Large Can	Medium Can	Scale	Small Bowl	Small Can	Tape Roll
Broom	0.990	0.004	0.002	0.000	0.000	0.000	0.002	0.000	0.002	0.000
Fitting	0.049	0.781	0.031	0.000	0.000	0.002	0.115	0.000	0.022	0.000
Lathe Knob	0.000	0.002	0.993	0.000	0.000	0.001	0.000	0.000	0.004	0.000
Large Bowl	0.000	0.000	0.000	0.827	0.000	0.000	0.000	0.173	0.000	0.000
Large Can	0.000	0.000	0.000	0.000	0.802	0.198	0.000	0.000	0.000	0.000
Medium Can	0.000	0.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000
Scale	0.007	0.023	0.008	0.000	0.000	0.000	0.962	0.000	0.000	0.000
Small Bowl	0.000	0.000	0.000	0.393	0.000	0.000	0.000	0.607	0.000	0.000
Small Can	0.000	0.001	0.000	0.000	0.000	0.003	0.001	0.000	0.995	0.000
Tape Roll	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000

Table 5-8. Confusion matrix for CPH classifier on LANL dataset.

	Broom	Fitting	Lathe Knob	Large Bowl	Large Can	Medium Can	Scale	Small Bowl	Small Can	Tape Roll
Broom	0.983	0.013	0.000	0.000	0.000	0.000	0.001	0.003	0.000	0.000
Fitting	0.003	0.964	0.000	0.000	0.000	0.000	0.000	0.000	0.033	0.000
Lathe Knob	0.000	0.002	0.998	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Large Bowl	0.000	0.000	0.000	0.934	0.000	0.000	0.000	0.066	0.000	0.000
Large Can	0.000	0.000	0.000	0.000	0.969	0.031	0.000	0.000	0.000	0.000
Medium Can	0.000	0.000	0.000	0.000	0.001	0.999	0.000	0.000	0.000	0.000
Scale	0.006	0.056	0.008	0.000	0.000	0.001	0.929	0.000	0.000	0.000
Small Bowl	0.000	0.000	0.000	0.040	0.000	0.001	0.000	0.957	0.002	0.000
Small Can	0.000	0.012	0.000	0.000	0.000	0.000	0.002	0.000	0.986	0.000
Tape Roll	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000

Table 5-9. Standard deviation in pose estimate on LANL dataset.

Object	VFH σ (degrees)	CPH σ (degrees)
Broom	16.9	24.2
Fitting	88.9	93.8
Lathe Knob	16.4	12.6
Scale	27.4	28.9

The ability of the CPH feature to capture object size is apparent in the results for the different sized bowls and different sized cans. The VFH classifier classifies the large can as a medium can 19.8% of the time. The CPH classifier is only confused between these objects 3.1% of the time. Similarly, the VFH classifier classifies the large bowl as a small bowl 17.3% of the time, and the small bowl as a large bowl 39.3% of the time. The CPH classifier's confusion rates between the same objects are 6.6% and 4.0% respectively.

The results in table 5-9 are surprising in that The VFH classifier performs dramatically better pose estimation on the live LANL data, and the CPH classifier performs significantly worse. The better performance of VFH is likely attributable to the fact that the four objects in table 5-9 are more conducive to feature-based pose estimation than those in Tables 5-3 and 5-6 from the RGB-D dataset. That is, the battery, camera, and flashlight all have bilateral symmetry across at least one axis, which can lead to pose confusion. The fact that the coffee mug's handle may be partially or fully occluded means there exists a range of ambiguous poses. The only object from the RGB-D dataset with a strong, non-symmetrical pose signature is the pliers, and for the pliers, the pose error is similar to what is observed in the LANL dataset. The degradation of the CPH pose estimation performance is not as surprising. Because the live images are not derived directly from the training data as in the RGB-D experiments, we would expect an exact pose match to be less common, and indeed the result reflect this.

The T-fitting results for both objects show that the pose estimate is not much better than random. The fitting is much smaller than the other objects, and its images tend to be noisy and incomplete. This object also gives the poorest recognition from the VFH classifier, although it is interesting to note that the CPH classifier gives very good recognition performance, although the test image is seldom matched to the correct pose.

This suggests that the recognition success is mostly attributable to the size component of the CPH feature.

5.3 PROBABILISTIC MODELING RESULTS

The experiments in the previous section examined the raw performance of two classifiers. All of the results in that section are from analysis on single image frames. This section demonstrates how the use of probabilistic modeling affects classifier performance. The data tables in this section look very similar to the data tables presented in the previous section. However, these results are computed accumulating and filtering results over 10 frames by the methods presented in Section 3.1. Filtering more than ten frames becomes computationally prohibitive. At this level, performance is already less than real time. However, some latency in the recognition performance is acceptable if the results are significantly improved.

The Bayesian filtering operations require a sensor model. The sensor model is constructed from the data collected in the previous section. We assume a zero mean error in the pose estimate giving $C_t = I$. The standard deviations in the pose estimates from the previous section are used to construct the noise covariance model matrix, Q_t. The class-conditional probabilities, P(z|C) are generated from the confusion matrices presented in the previous section in the following way. Assuming confusion matrix *F,* each element in a class-conditional probability matrix *P* is constructed accoding to:

$$P_{z,c} = \frac{F_{z,c}}{\sum_{i=1}^{N} F_{i,c}} \qquad 5\text{-}2$$

The result is a matrix *P* where $P_{i,j}$ = P(z_i|C_j), the probability of obtaining recognition result *i*, given that the object is of class *j*. This matrix, along with the standard pose deviation for each object is loaded from file when the `recognition_main` node is created.

5.3.1 RGB-D dataset results

As in the previous section, individual frames for the RGB-D test data are generated by adding Gaussian noise to the z channel of the point cloud. The distributions used to model the object are updated frame over frame by Equations 3-4 to 3-7. The final reported object class is the most probable class in the discrete distribution representing the object. the object. Tables 5-10 and 5-11 show confusion matrices for the final recognition result after 10 frames of filtering for VFH and CPH respectively.

Table 5-10. Confusion matrix filtered over 10 frames for VFH classifier, σ_{noise} = 1 mm.

	Bowl	Camera	Coffee Mug	Battery	Flashlight	Jar	Pliers
Bowl	1.000	0.000	0.000	0.000	0.000	0.000	0.000
Camera	0.000	0.893	0.066	0.011	0.030	0.000	0.000
Coffee Mug	0.000	0.000	0.766	0.000	0.060	0.174	0.000
Battery	0.000	0.000	0.000	0.973	0.027	0.000	0.000
Flashlight	0.000	0.005	0.023	0.088	0.884	0.000	0.000
jar	0.000	0.010	0.000	0.000	0.000	0.990	0.000
Pliers	0.000	0.006	0.000	0.000	0.000	0.000	0.994

Table 5-11. Confusion matrix filtered over 10 frames for CPH classifier, σ_{noise} = 1 mm.

	Bowl	Camera	Coffee Mug	Battery	Flashlight	Jar	Pliers
Bowl	1.000	0.000	0.000	0.000	0.000	0.000	0.000
Camera	0.000	1.000	0.000	0.000	0.000	0.000	0.000
Coffee Mug	0.000	0.000	1.000	0.000	0.000	0.000	0.000
Battery	0.000	0.000	0.000	1.000	0.000	0.000	0.000
Flashlight	0.000	0.000	0.000	0.000	1.000	0.000	0.000
jar	0.000	0.000	0.000	0.000	0.000	1.000	0.000
Pliers	0.000	0.000	0.000	0.000	0.000	0.000	1.000

The results for the CPH classifier, predictably, are still 100%. The statistical filtering cannot improve upon the 100% raw classification rate. The results for the VFH classifier, however, do demonstrate an improvement in the classification rates over the

raw VFH classifier. The overall recognition performance improves from 90.1% to 92.9%. The improvement is not dramatic, but does indicate that there is some value in collecting statistics and accumulating evidence to avoid misclassifications.

The reported pose is the mean of the 4D Gaussian model after filtering over 10 frames. Table 5-12 shows the filtered pose estimation results. Overall, there is a 17.4% reduction of the error in the pose estimate. These results show that probabilistic modeling and Bayesian update make a significant improvement in the pose estimates. Some of the VFH results are only improved marginally.

Table 5-12. Standard deviation in pose estimate (measured from ground truth) for filtered VFH and CPH classifiers, σ_{noise} = 1 mm.

Object	VFH σ (degrees)	CPH σ (degrees)
Bowl	NA	NA
Camera	75.4	0.0
Coffee Mug	100.5	0.0
Battery	96.5	0.4
Flashlight	59.1	0.0
jar	41.7	0.0
Pliers	21.2	0.6

For objects such as the coffee mug, the raw standard deviation is high, thus there is not much improvement after 10 frames. However, where the standard deviation is lower, such as with the pliers, the error in the pose estimate is reduced by 50%. This is also true for the CPH results. Because the raw standard deviations were all fairly small at around 10°, the statistical filtering rapidly reduces the error in the pose estimate. For each object, the error in the pose estimate drops to less than 1°.

The same statistical filtering experiments were performed on the σ_{noise} = 5 mm data. Table 5-13 shows the confusion matrix for the VFH classifier. Because of the

perfect raw recognition rate of the CPH classifier, the confusion matrix is not shown for the statistically filtered case, but the filtered recognition results were performed, and confirm that the filtered result remains perfect. The filtered VFH results improve the overall recognition rate from 86.3% to 91.7%. Table 5-14 shows the filtered pose estimation results for both classifiers at σ_{noise} = 5 mm. Unsurprisingly, the results show that the pose estimation deteriorates in the presence of more noise.

Table 5-13. Confusion matrix filtered over 10 frames for VFH classifier, σ_{noise} = 5 mm.

	Bowl	Camera	Coffee Mug	Battery	Flashlight	Jar	Pliers
Bowl	1.000	0.000	0.000	0.000	0.000	0.000	0.000
Camera	0.000	0.720	0.000	0.142	0.128	0.005	0.005
Coffee Mug	0.000	0.000	1.000	0.000	0.000	0.000	0.000
Battery	0.000	0.000	0.000	1.000	0.000	0.000	0.000
Flashlight	0.000	0.017	0.006	0.094	0.883	0.000	0.000
jar	0.000	0.116	0.000	0.015	0.000	0.869	0.000
Pliers	0.000	0.005	0.000	0.022	0.017	0.011	0.945

Table 5-14. Pose estimation results for both classifiers at σ_{noise} = 5 mm.

Object	VFH σ (degrees)	CPH σ (degrees)
Bowl	NA	NA
Camera	96.7	0.2
Coffee Mug	101.5	1.1
Battery	114.8	10.3
Flashlight	90.9	0.2
jar	72.2	0.0
Pliers	70.1	1.2

5.3.2 LANL dataset results

The Probabilistic modeling results for the LANL dataset are collected in the same way, again with the exception that live data are used instead of artificially noisy data. The object model is updated over 10 frames, filtered according to the methods described in Chapter 3. Tables 5-15 and 5-16 show the filtered recognition results. Filtering over 10 frames improves the overall accuracy of the VFH classifer from 89.6% to 94.6%. The CPH recognition rate improves from 97.2% to 98.5%, with six out of the ten objects achieving 100% correct recognition.

Table 5-17 shows the filtered pose error results for the LANL dataset. Filtering improves the overall accuracy of the VFH classifier by 20.4% and the CPH classifier by 17.6%. This translates to an absolute improvement of 4.8° and 6.3° respectively.

Table 5-15. Confusion matrix for LANL datset filtered over 10 frames for VFH classifier.

	Broom	Fitting	Lathe Knob	Large Bowl	Large Can	Medium Can	Scale	Small Bowl	Small Can	Tape Roll
Broom	0.992	0.008	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Fitting	0.000	0.917	0.000	0.000	0.000	0.000	0.050	0.000	0.033	0.000
Lathe Knob	0.000	0.000	0.993	0.000	0.000	0.001	0.000	0.000	0.006	0.000
Large Bowl	0.000	0.000	0.000	0.992	0.000	0.000	0.000	0.008	0.000	0.000
Large Can	0.000	0.000	0.000	0.000	0.958	0.042	0.000	0.000	0.000	0.000
Medium Can	0.000	0.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000
Scale	0.000	0.025	0.000	0.000	0.000	0.000	0.975	0.000	0.000	0.000
Small Bowl	0.000	0.000	0.000	0.367	0.000	0.000	0.000	0.633	0.000	0.000
Small Can	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.000	1.000	0.000
Tape Roll	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000

Table 5-16. Confusion matrix for LANL dataset filtered over 10 frames for CPH classifier.

	Broom	Fitting	Lathe Knob	Large Bowl	Large Can	Medium Can	Scale	Small Bowl	Small Can	Tape Roll
Broom	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Fitting	0.000	0.975	0.000	0.000	0.000	0.000	0.000	0.000	0.025	0.000
Lathe Knob	0.000	0.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Large Bowl	0.000	0.000	0.000	0.975	0.000	0.000	0.000	0.025	0.000	0.000
Large Can	0.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000	0.000
Medium Can	0.000	0.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000
Scale	0.006	0.000	0.052	0.000	0.000	0.000	0.942	0.000	0.000	0.000
Small Bowl	0.000	0.000	0.000	0.040	0.000	0.001	0.000	0.957	0.002	0.000
Small Can	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000
Tape Roll	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000

Table 5-17 Standard deviation in pose estimate on LANL dataset filtered over 10 frames.

Object	VFH σ (degrees)	CPH σ (degrees)
Broom	12.9	19.1
Fitting	84.6	82.5
Lathe Knob	11.0	12.1
Scale	21.8	19.3

5.4 CATEGORY-LEVEL RECOGNITION RESULTS

The CPH feature is intended to enable recognition of specific instances of an object. In fact, it is intended to recognize a specific *view* of a specific instance of an object. The classification results so far consider each view of each object a class, and there is an implicit constraint that every instance of a class is identical in appearance. If this constraint is removed, the problem becomes more difficult. As an example, a classifier that must recognize "chairs" would have to correctly assign the objects in Figure 5-8 to the same class. This problem is known as *category-level* recognition as opposed *to instance-level* recognition.

Figure 5-7. Example of a challenging category-level problem.

Although a human immediately recognizes all of the images in Figure 5-7 as chairs, it would clearly be a much more difficult problem for a computer. Even though CPH is not intended for use on the category-level problem, its performance is evaluated here to get a sense of how well a global feature might perform if the identical appearance assumption were removed.

Lai's RGB Dataset provides several instances of the categories that are used in this work. In the experiments so far, only a single instance, imaged from several viewpoints has been used. To approach the category-level problem, the remaining

instances of each objects are added to our dataset. Figure 5-2 shows all object instances used for the category-level problem. The recognition methodology is modified in the following ways:

- The test data is no longer generated by adding artificial noise. Instead, we use Leave One Out – Cross Validation (LOO-CV) Where we train on all images from all instances except one. All images of the remaining instance are then used for testing. Under this method, no images of the test object match are present in the training data.
- The KNN classifier is replaced by a Support Vector Machine (SVM) classifier. The classifier is LibSVM with a posterior probability output. [Chang, 2011].

Table 5-18 shows the category-level results for the seven classes, and Figure 5-8 shows the ROC curves for each object. On six out of the seven categories, the category-level performance of the CPH/SVM classifier is excellent. However, the 0.0% recognition rate on the pliers is troubling. An important observation of the object instances in Figure 5-2 is that the shape of the objects does not vary dramatically between the instances. The category-level problem attempted here is nothing like the one suggested by figure 5-8. The pliers have more shape variation among the instances in the class then the other objects, and the performance suffers as a result. The good performance overall suggests that the feature can perform well on easier category-level problems, but if the identical appearance assumption is relaxed too far, the CPH feature ceases to be a good discriminator.

Table 5-18. Category-level recognition rates for CPH feature with SVM classifier.

Object	Recognition Rate
Bowl	100.0%
Camera	90.6%
Coffee Mug	93.4%
Dry Battery	92.5%
Flashlight	100.0%
Food Jar	100.0%
Pliers	0.0%

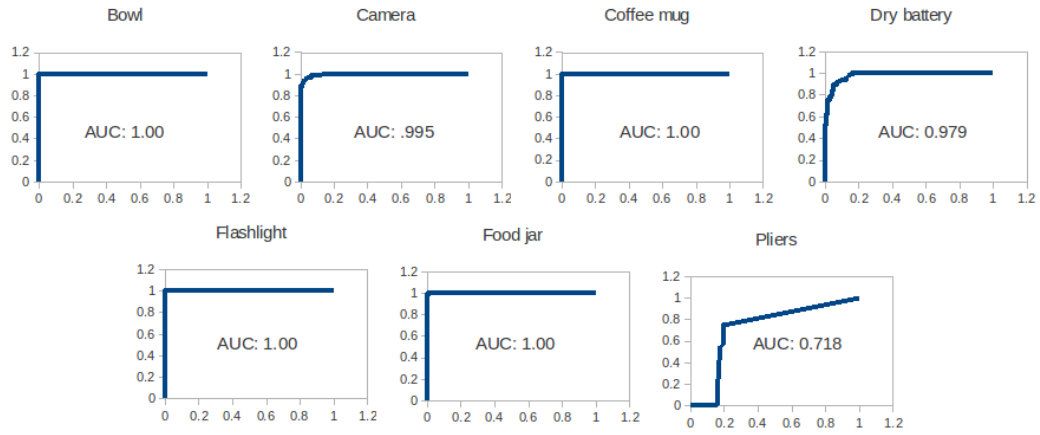


Figure 5-8. Receiver Operating Characteristics for category-level recognition with CPH feature and SVM classifier.

5.5 COMPUTATIONAL PERFORMANCE

Both the CPH and VFH features are intended to be used in real-time recognition systems. Therefore the ability to compute the feature in real time on a standard PC is important. The CPH feature has a computational advantage in that it does not require surface normal estimation on the cluster. Table 5-19 shows the computational performance for each object in the RGB-D dataset, and Figure 5-9 plots the computation time against the number of points in the cluster. The average feature computation times are reported for the CPH features whereas the actual computation times are plotted for the

VFH feature. This is because for the CPH features, the actual computation times are often close to machine zero, so they must be accumulated to get a meaningful number. Nevertheless, the data clearly demonstrate that the computational performance of CPH is real time whereas for VFH it is not.

These data were collected on a Lenovo notebook computer with an Intel Core i7 processor and 8GB of RAM running Ubuntu Linux 11.10. The times for the VFH feature include both the feature computation time and the time required to estimate the surface normal. On average, the CPH feature computes 753 times faster than the VFH feature.

Table 5-19. Average feature computation time on the RGB-D dataset.

Avg. Size (points)	VFH Comp. Time (ms)	CPH Comp. Time (ms)
10814	1734	2.500
1207	481.4	0.625
6335	1160	1.375
457	25.81	0.250
3284	698.1	0.500
2742	364.4	0.563
671	21.75	0.138

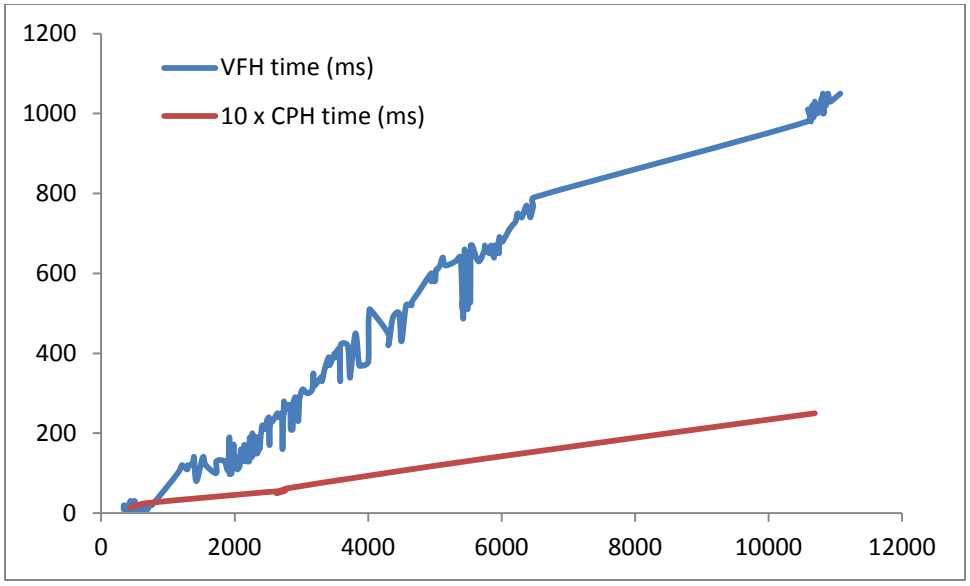


Figure 5-9. Feature computation time vs. cluster size. The CPH values shown are scaled by a factor of 10.

Figure 5-10 shows that both features require time approximately linear in the number of points, the slopes are dramatically different. For larger point clouds, it can take almost two seconds to compute a single VFH feature. The CPH feature, on the other hand, computes in milliseconds on even the largest clusters, making it suitable when real-time performance is required.

5.6 SUMMARY AND DISCUSSION OF EXPERIMENTS

5.6.1 Chapter 5 summary

The experiments discussed in this chapter were intended to evaluate the performance of the CPH feature descriptor for use in 3D point cluster recognition and pose estimation. To do this, a modular recognition pipeline was implemented using PCL and ROS. This allowed the evaluation of CPH as an alternative to VFH, the feature implemented in PCL for use in point cluster recognition and pose estimation. Several experiments quantified the performance of each feature in the following areas.

- Raw multi-class recognition rate.
- Raw rotational pose error.
- Bayesian filtered multi-class recognition rate using probabilistic object model.
- Pose error after Kalman filter update of probabilistic object model.
- Category-level recognition performance
- Computational performance.

The remainder of this chapter discusses the results from each of these areas and evaluates them in the context of the general object recognition problem. It also addresses the specific problem of simultaneous object recognition and pose estimation for nuclear materials handling.

5.6.2 Multi-class recognition and pose estimation.

The first set of recognition and pose estimation experiments was done on the RGB-D dataset discussed in Section 5.1.1. The results are summarized in Table 5-20.

Table 5-20. Recognition and pose estimation summary (RGB-D data)

		VFH		CPH	
		σ_{noise} = 1mm	σ_{noise} = 5mm	σ_{noise} = 1mm	σ_{noise} = 5mm
Bowl	Rec. Rate	100%	100%	100%	100%
	σ_{pose} (deg)	NA	NA	NA	NA
Camera	Rec. Rate	80%	63%	100%	100%
	σ_{pose} (deg)	88.7	102.3	0.0	5.7
Coffee Mug	Rec. Rate	100%	100%	100%	100%
	σ_{pose} (deg)	76.8	102.6	0.0	1.8
9V Battery	Rec. Rate	100%	99%	100%	100%
	*σ_{pose} (deg)	101.4	112.3	0.4	22.1
Flashlight	Rec. Rate	88%	63%	100%	100%
	σ_{pose} (deg)	62.2	95.2	0.0	0.5
Food Jar	Rec. Rate	98%	82%	100%	100%
	σ_{pose} (deg)	42.9	85.0	0.0	1.0
Pliers	Rec. Rate	99%	93%	100%	100%
	σ_{pose} (deg)	18.0	70.3	0.1	2.7

The RGB-D dataset results suggested that the CPH feature performed much better than the VFH feature in its ability to match a test object to a specific view of a specific object. Furthermore, the results in Table 5-20 show that the recognition performance of VFH degrades in the presence of significant noise. However, because the test images were artificially generated from the training set, these results demonstrate performance much better than would be expected in a deployed recognition system.

In order to better characterize the true performance of the CPH feature, and also to test it on application-relevant objects, the LANL dataset was developed. This allowed live testing of the VFH and CPH features under conditions similar to what is expected for a deployed system. Table 5-21 summarizes the performance.

Table 5-21. Recognition and pose estimation summary (LANL data)

		VFH	CPH
Broom	Rec. Rate	99%	98%
	σ_{pose} (deg)	16.9	24.2
T-fitting	Rec. Rate	78%	96%
	σ_{pose} (deg)	88.9	93.8
Lathe knob	Rec. Rate	99%	100%
	σ_{pose} (deg)	16.4	12.6
Large bowl	Rec. Rate	83%	93%
	σ_{pose} (deg)	NA	NA
Large can	Rec. Rate	80%	97%
	σ_{pose} (deg)	NA	NA
Medium	Rec. Rate	100%	100%
	σ_{pose} (deg)	NA	NA
Scale	Rec. Rate	96%	93%
	σ_{pose} (deg)	27.4	28.9
Small bowl	Rec. Rate	61%	96%
	σ_{pose} (deg)	NA	NA
Small can	Rec. Rate	100%	99%
	σ_{pose} (deg)	NA	NA
Tape roll	Rec. Rate	100%	100%
	σ_{pose} (deg)	NA	NA

The results for the LANL set demonstrate much better recognition performance from the CPH feature. One of the goals of CPH was to define a feature that would capture object size. As expected, the performance advantage of CPH is significantly greater when differentiating between objects of similar shape but different size. The VFH feature is confused between large and small bowls and large and small cans. It is interesting to note that VFH is not confused by the small can, despite its similarity in shape to the medium and large cans. This can be explained by examining Figure 5-6. The small can's surface has particularly high specular reflection, and so the segment representing it is missing most of the can's surface, resulting in a point cluster whose shape is much different than the large or medium can.

Using the live data caused an expected drop in pose estimation performance for the CPH classifier. Surprisingly, however, the pose estimation performance of the VFH classifier improved considerably. This is attributable to the pose ambiguities of the objects selected for the RGB-D dataset making pose estimation by surface normal binning very difficult.

5.6.3 Filtered recognition and pose estimation

By collecting statistics on classifier performance, a sensor model was developed and used to filter results over 10 frames. The results for the RGB-D dataset are summarized in Table 5-22. While the raw performance of CPH on this data is too good to draw any conclusions about the benefits of using a probabilistic model, the VFH results do show improvement in recognition and pose estimation results. However, the improvements in pose estimates are modest and neither feature offers performance suitable to use in a deployed system.

Table 5-22. Filtered results summary for RGB-D data.

		VFH		CPH	
		σ_{noise} = 1mm	σ_{noise} = 5mm	σ_{noise} = 1mm	σ_{noise} = 5mm
Bowl	Rec. Rate	100%	100%	100%	100%
	σ_{pose} (deg)	NA	NA	NA	NA
Camera	Rec. Rate	89%	72%	100%	100%
	σ_{pose} (deg)	74.5	96.7	0.0	0.2
Coffee	Rec. Rate	76%	100%	100%	100%
	σ_{pose} (deg)	100.5	101.5	0.0	1.1
9V Battery	Rec. Rate	97%	100%	100%	100%
	σ_{pose} (deg)	96.5	114.8	0.4	10.3
Flashlight	Rec. Rate	88%	88%	100%	100%
	σ_{pose} (deg)	59.1	90.9	0.0	0.2
Food Jar	Rec. Rate	99%	87%	100%	100%
	σ_{pose} (deg)	41.7	72.2	0.0	0.0
Pliers	Rec. Rate	99%	95%	100%	100%
	σ_{pose} (deg)	21.2	70.1	0.0	1.2

Table 5-23. Filtered results summary for LANL data.

		VFH	CPH
Broom	Rec. Rate	99%	100%
	σ_{pose} (deg)	12.9	19.1
T-fitting	Rec. Rate	92%	97%
	σ_{pose} (deg)	84.6	82.5
Lathe knob	Rec. Rate	99%	100%
	σ_{pose} (deg)	11.0	12.1
Large bowl	Rec. Rate	99%	98%
	σ_{pose} (deg)	NA	NA
Large can	Rec. Rate	96%	100%
	σ_{pose} (deg)	NA	NA
Medium can	Rec. Rate	100%	100%
	σ_{pose} (deg)	NA	NA
Scale	Rec. Rate	98%	94%
	σ_{pose} (deg)	21.8	19.3
Small bowl	Rec. Rate	63%	100%
	σ_{pose} (deg)	NA	NA
Small can	Rec. Rate	100%	100%
	σ_{pose} (deg)	NA	NA
Tape roll	Rec. Rate	100%	100%
	σ_{pose} (deg)	NA	NA

Table 5-23 provides a summary of the filtered recognition and post estimation results on the LANL dataset. This table makes a strong case for the use of a probabilistic object model. The pose estimates are significantly improved with both features. Much of the large/small object confusion in the VFH classifier is mitigated by statistical filtering, although recognition performance for the small bowl remains very poor. While the overall performance of CPH is still better than VFH, the advantage is not as great on live data with the use of the probabilistic model. However, the computational advantage of CPH must be considered along with these results. On large point clusters, the VFH feature may take almost two seconds to compute, so filtering over 10 frames means the total time require to achieve the performance in Table 5-23 is nearly 20 seconds for some objects. CPH, on the other hand, computes extremely quickly, and filtering over 10 frames still occurs in less than a second.

5.6.4 Category-level performance

Category-level recognition is a harder problem than instance-level recognition. Neither VFH nor CPH are designed with this problem in mind, but the performance on this problem was examined experimentally using an SVM classifier instead of the KNN classifier used in the instance-level experiments. While the classifier performed well on some classes, the failure of CPH to recognize any instances of one class of objects suggest that it is not suitable to this problem.

5.6.5 Computational performance

CPH is a much simpler feature than VFH from a computational perspective. This is because of the need to estimate point surface normals in order to compute the VFH descriptor. CPH computes about 3 orders of magnitude faster than VFH. This is the

difference between computing in a couple milliseconds versus a couple of seconds. Because of this, CPH is much better suited to real-time recognition.

Overall, the performance of CPH is comparable to or better than VFH in every area of comparison. The filtered recognition performance is sufficient for use on datasets of around ten objects, even in applications that require high recognition accuracy. However, the pose estimation results are generally not good enough for autonomous grasping in unstructured environments.

Chapter 6: Application demonstrations

Previous chapters have provided substantial detail on the technical aspects of a visual cluster recognition system. This chapter shows how the technical accomplishments discussed previously can be used in deployed systems. The first demonstration shows how CPH recognition enables autonomous pick and place tasks in an unstructured workcell. The second demonstration shows how information from the probabilistic object model can be displayed to a system user to give a real-time idea of model quality. These two demonstrations show how the work presented in this document support unstructured manipulation by advancing autonomous capabilities and supporting notions of shared human-robot initiative and transitional autonomy.

6.1 AUTOMATED SORTING WORKCELL

The CPH feature is part of the ROS-Industrial packages for industrial manipulation in ROS. It was recently used at the core of a vision-based demonstration to showcase ROS-Indsustrial at a major robotics conference and trade show [Automate 2013]. This demonstration was developed in collaboration with Southwest Research Institute, the North American Center of Excellence for ROS-Industrial. This chapter describes the demonstration and how it relates to nuclear materials handling.

All of the grasp and motion planning in the following demonstration is performed in ROS using sensor information from Microsoft Kinect depth cameras. ROS-Industrial provides the ROS bindings for the manipulators in the workcell. The demonstration is intended to show the following:

- How ROS and ROS-Industrial simplify integration of complex robotic systems under a common operating framework.

- How using ROS for industrial automation allows incorporation of recent research advances into deployed systems.

CPH features form the crux of the ability to demonstrate the second point. The demonstration described below shows that robust and reliable object recognition can be achieved on a deployed system suitable for unstructured industrial automation.

6.1.1 Demonstration workcell

Figure 6-1 shows an exterior view of the demonstration workcell. Figure 6-2 shows a schematic layout of the demonstration area. The workcell contains two manipulators, a Motoman SIA-20D and a Universal Robotics UR working in close proximity. The robots share a workspace within a confined environment. Parts of the workspace are partitioned off according to the task specification, but the environment is unstructured in that the positions of objects are uncertain and no fixtures are used.

Figure 6-1. Demonstration workcell, exterior view.

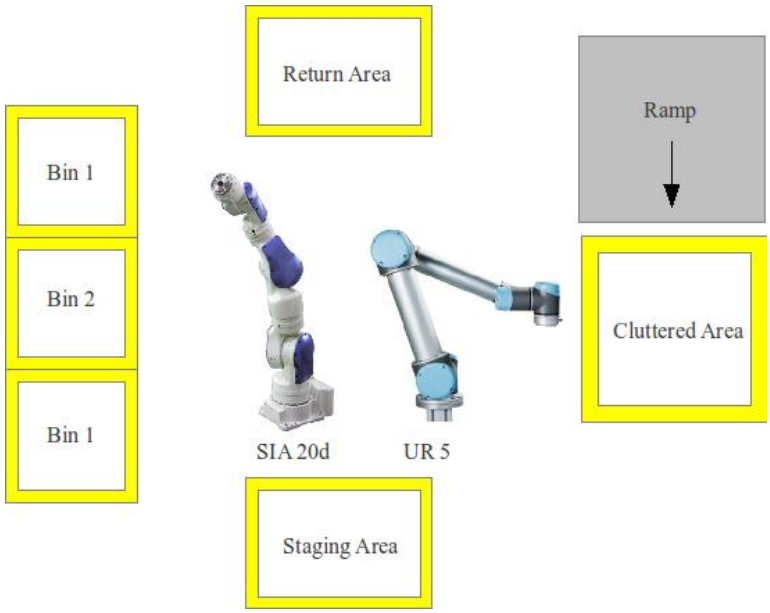


Figure 6-2. Schematic workspace layout.

6.1.2 Task description and execution

The demsontration task is to sort a cluttered group of objects into a set of bins. The bins in this demonstration are not physical bins, but areas partitioned out of the workspace for item storage. The demonstration begins with four objects of each of three type randomly distributed in the cluttered area. The sorting proceeds as follows:

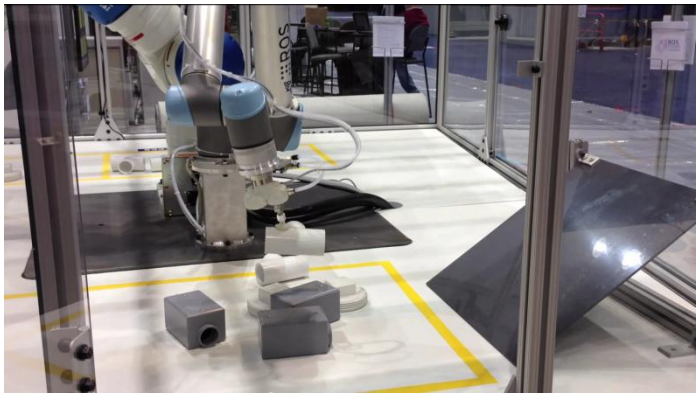
1. **Object singulation.** – The UR 5 Robot moves a single object from the cluttered area to the staging area. The object is selected from the cluttered area at random after image segmentation. Because of the clutter, CPH would be unreliable for recognition in the cluttered area.
2. **Recognition and sorting.** – Once a single object has been moved to the staging area, the processing pipeline described in Chapter 4 is used to identify the object. When it has been identified, the SIA-20d robot moves the object to the appropriate bin location.

3. **Return.** - When all objects have been sorted, the SIA-20d moves them each individually to the return area. The UR 5 robot retrieves them and drops them onto the ramp where they tumble back into the cluttered area. The demonstration then begins again.

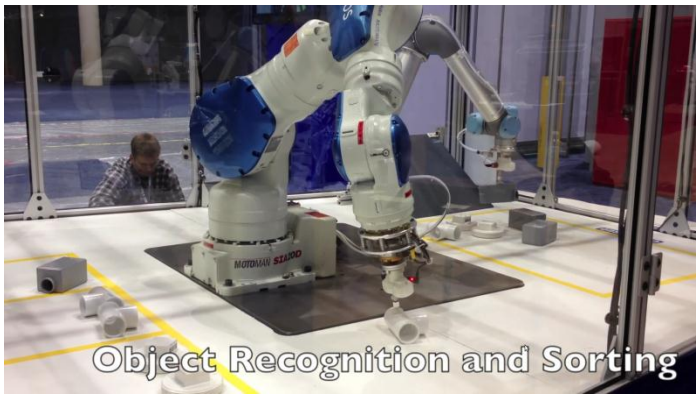
Figure 6-3 shows the demonstration in progress. A video of the demonstration is available at http://www.youtube.com/watch?v=dGPwHodET8s&feature.

6.1.3 Demonstrated capability

The demonstration described in this section shows how the work presented in this document helps solve the problem of manipulation in unstructured environments. The demonstration task shares much in common with the nuclear materials handling problem introduced in Chapter 1. The objects have little surface texture, the environment is confined and unstructured, but also contains little clutter. All of this is also true in a nuclear materials glovebox. The problem of light clutter is addressed by separating individual objects prior to additional segmentation and recognition. For a system operating under dynamic autonomy, this demonstration shows how reliable cluster recognition allows the system to accept a high level instruction from the operator ("Put away these objects") and translate that isntruction into a series of low-level tasks that can be completed without further input from the operator.



(a)



(b)

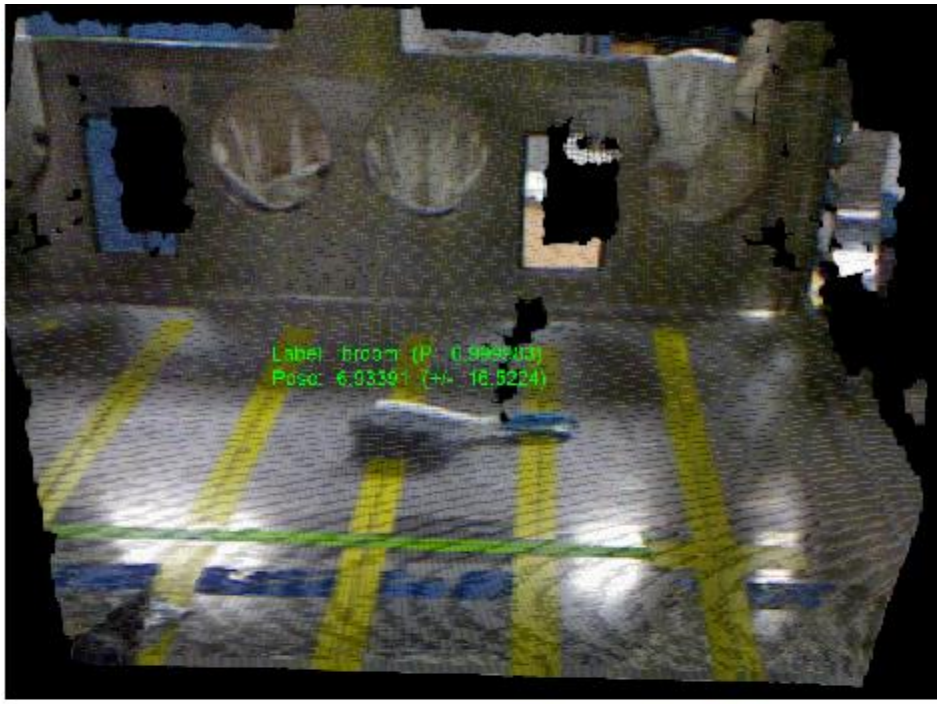
(c)

Figure 6-3. Application demonstrion. (a) Object singulation with UR 5 robot. (b) Recogition and object grasp. (c) Object sorting.

6.2 SYSTEM USER DISPLAY DEMONSTRATION AT UT AUSTIN

Perhaps the most important aspect of a transitional autonomy scheme is the ability of the system to determine the quality of its internal world model and communicate that to the system operator. This allows the operator to know when the model is degraded and anticipate the need for more direct human involvement in system operation. This section shows a simple user display where information about objects detected and modeled by the perception system is displayed for the user.

Figure 6-4 shows two screen captures of the user dislay. The image is annotated with the object and model information. In addition to the object's label and pose, probabilisitic model information is output to the screen adjacent to the perceived object. Figure 6-4 (a) was captured immediately after the new object was acquired. Figure 6-4 (b) was acquired after a few frames. The improvement in the model resulting from fitering is apparent between the images. The probability of the object being a broom reaches a maximum of 1 and the pose uncertainty is reduced from 16.5 degrees to less than a degree. Figure 6-5 demonstrates the ability of the classifier to handle multiple objects in light clutter. Images are shown with two and four objects present respectively.

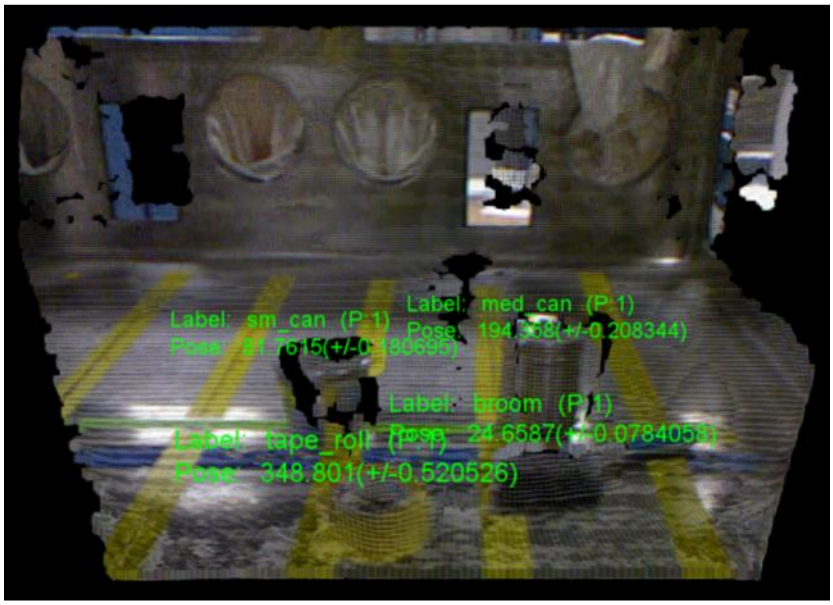


(a)

(b)

Figure 6-4. User display showing label with probability estimate and pose with deviation estimate. (a) shortly after initial acquisition. (b) several frames later.

(a)



(b)

Figure 6-5. User display showing light clutter (a) two objects present. (b) four objects present.

6.3 COMMENTS ON DEMONSTRATIONS

The demonstrations described in this chapter show how the recognition pipeline developed in Chapters 3 and 4 enables certain autonomous tasks in unstructured environments. The most fundamental manipulator task is to move an object from one place to another. The automated sorting workcell shown in Section 6.1 demonstrates this capability within a difficult unstructured environment. The majority of manipulator tasks in a glovebox would be simiar pick-and-place operations.

The user display shown in Section 6.3 gives a system operator a real-time notion of the quality of the system's internal model. This allows the operator to know whether or not a particular task can be completed automously under high-level direction, or if it must be completed through teleoperation. These demonstrations show that a CPH-based recognition system with probabilistic object modeling supports transitional autonomy system by enabling autonomous unstructured pick-and-place and by providing the model data necessary to determine an appropriate level of autonomy.

Chapter 7: Conclusion

The last decade has seen rapid expansion of autonomous robotics technologies. However, very few of these technologies are successfully crossing the chasm between research labs and commercial or industrial success. There are many technical and economic reasons for this that can all be concisely summarized as follows: Autonomous robotic systems must be able to do the things humans do more safely, more reliably, and at lower cost, and in approximately the same amount of time This safer/better/cheaper axiom is the reason that robots are not used for many exceptionally hazardous tasks at the national laboratories, despite the fact that they have access to some of the best trained scientists and engineers in the world.

This chapter summarizes the material presented in previous chapters and suggests several avenues for further research that would improve the work presented here and extend its impact.

7.1 SUMMARY

Chapter 1 introduced the problem of unstructured manipulation in the context of nuclear materials handling in DOE glovebox facilities. It proposed the transitional autonomy as a way to achieve the safety and robustness required to better deploy and utilize advanced robotics. This idea acknowledges that the current state of the art in fully autonomous systems is insufficient to satisfy the safer/better/cheaper axiom and opts instead to provide a framework in which the level of system autonomy changes in response to environmental uncertainty, incomplete information, or user input. This requires a visual perception system to perceive the environment, and a probabilistic modeling framework to track the quality of the model. This chapter also discussed ROS, an exciting development in robotics research that improves the probability of system

deployment by opening up a wealth of state-of-the-art algorithms to any system implemented in ROS. It also briefly discussed the Microsoft Kinect, and how it has rapidly transformed the field of robot vision and perception by offering inexpensive access to coupled range and color imagery.

Chapter 2 examined and reviewed previous work in the areas of robot world modeling, computer vision, and robot perception. The clear trend in world modeling is toward probabilistic models. This trend is driven in large part to the need of mobile systems to localize themselves from noisy sensor data, a problem that most manipulation systems do not have. However as manipulation systems move into increasingly uncertain environments, and are integrated with a wider range of noisy sensors, they too benefit from the use of probabilistic modeling.

Many of the well-developed techniques for computer vision do not translate well to depth imagery. This is particularly true for segmentation algorithms and image feature descriptors which tend to rely on strong gradients. With the advent of sensors like the Kinect, there has been a rapid expansion into segmentation algorithms and feature descriptors that work well on depth imagery. One of these, the Viewpoint Feature Histogram, is used as a benchmark for the current work.

Chapter 3 presented the methods that form the backbone of the current work. A probabilistic object model is used that treats an object's label as a discrete probability distribution over all possible objects. Objects are assume to sit upright on a planar surface, restricting the pose to four dimensions: x, y, z and θ. The model is a 4D Gaussian in these dimensions. This object model is updated by discrete Bayesian filtering, and Kalman filtering for the label and pose, respectively.

Chapter 3 also introduced a novel feature descriptor, the Cylindrical Projection Histogram (CPH). CPH is a member of a class of shape descriptors called shape

histograms that also includes the Spin Image. Its major advantages compared to other point cluster descriptors are its fast computation, and its ability to encode object size while maintaining invariance to image scale.

Chapter 4 delved into the details system implementation. This chapter described how the methods presented in Chapter 3 became part of a functional perception system, implemented in ROS and PCL. This chapter included a detailed description of the entire processing pipeline to take a depth frame from raw image to a description of the objects in the scene.

Chapter 5 uses the system presented in Chapter 4 to perform several experiments that test and validate the methods proposed in Chapter 3. Experiments were conducted on a publicly available dataset in the literature, and on a custom dataset that captures the operational requirements of glovebox manipulation at Los Alamos National Laboratory. Because the methods presented here require pose-annotated training data, a custom data-collection system was built to simplify collection of datasets with objects imaged at around 360° with 1° angular resolution.

The experiments occurred in two phases. In the first phase a VFH and a CPH classifier performed classification and pose estimation on approximately 12,000 images for each object. The raw results give a good indication of each feature's capabilities. The results from this phase are also used to generate *a priori* probability tables and pose statistics that inform the statistical update algorithms. In the second phase, each test image is filtered over 10 frames, and the resulting label and pose are reported. This is done for both datasets, the only difference being that the RGB-D test images were synthetically generated by adding Gaussian noise to the training set. The raw results demonstrate that the raw classification performance of CPH is superior to that of VFH, which a very strong advantage in its ability to discriminate between similarly shaped

objects of different size. This is done with a computational performance advantage of about three orders of magnitude.

Chapter 6 presented two application demonstrations show how the work presented here supports autonomous unstructured manipulation under transitional autonomy. Manipulation was performed on hardware systems, and model information communicated with a human operator for the purposes of determining whether or not a task can be completed autonomously.

7.2 RECOMMENDATIONS FOR FUTURE WORK

This document demonstrates the contribution of the CPH feature and probabilistic modeling techniques to the problem of unstructured manipulation. However, there remains significant work to be done in the areas of robot vision, world modeling, and transitional autonomy. The work presented here may serve a springboard for further exploration. Below is a list of avenues for revision, extension, and application of the techniques presented in this thesis.

- **Generalize feature scaling** – One of the most powerful aspects of the CPH feature is the direct encoding of size information without sacrificing scale invariance. However, it is likely that the addition of spatial extents and histogram re-scaling could be performed on any point cluster feature. This includes VFH and its derivatives. Researchers pursuing other types of point cluster features should consider these techniques to improve performance on objects of similar size or appearance but different size.
- **Investigate other projections** – The CPH feature projects points into a cylindrical surface. A cylinder was used as an intuitive shape to capture variation in viewpoint as an object on a plane rotates about a central axis. However, the

shape histogram component of CPH conceivably could be computed by projecting into any surface. It is possible that another type of projection, such a spherical one, could yield as good or better results, and may perform better in the general 6 DOF case.

- **Incorporate appearance** – All of the work presented here uses only depth data. However, in many applications, both a depth image and an appearance image are available. The CPH descriptor, or some variant of it, may be enhanced if used in conjunction with an appearance based method. Hybrid shape/appearance features have been shown to be effective in some real-time recognition applications. [Hinterstossier, 2012]. Possible approaches might include a concatenated global color histogram or vector of average chromaticity values for the points in each bin of the CPH shape histogram. Incorporation of appearance information may, however, have a deleterious effect on CPH's performance on reflective objects.
- **Evaluate on larger datasets** – Both datasets in this work are relatively small in order to keep the scale of the problem relevant to the DOE/LANL mission. However, it has not been determined whether or not the methods presented here would scale well to larger datasets. In order to evaluate the scalability, the LANL dataset should be expanded to include more objects. A good approach would be to create a dataset similar to Lai's RGB-D dataset, but to annotate each image with the geometric transformation between the camera frame and a local reference frame on the object. Evaluating on larger datasets may also motivate work in how to handle the hierarchical nature of object classes. For example, a multi-stage classifier might first determine whether the object is large or small, then determine a broad category, then determine a specific class or instance. This hierarchical approach has shown promise in works such as Lai.

- **Use multiple imagers** – All results in here use a single depth camera as a sensor with the result that the data is not fully 3D. The depth image is still captured by a 2D pixel array, so more than half of an object's geometry is typically occluded. However, the methods presented here should generalize well to a more complete 3D description of the object that could be captured by adding an additional depth camera opposite the first.
- **Sensor Fusion** – One of the strengths of the probabilistic model is that it permits model refinement from any sensor for which a sensor model exists. In order to further reduce model uncertainty, the robot could measure a sensed object's position via tactile sensing. The object's size could be measured by grasping with a force or current-limited gripper. These measurements could then inform the Gaussian position and the discrete distribution across object labels.
- **Downstream pose refinement** – The pose estimation results reported here are as good or better than feature-based pose estimation results with VFH. However, when using live data, the results are insufficient for dexterous grasping. For example, it would be unlikely that a robot reliably grasp a mug by its handle, or correctly wield a tool. However, iterative alignment techniques such as RANSAC or Iterative Closest Point (ICP) algorithms could be performed after obtaining a classification and gross pose estimation from the CPH classifier. The pose from the CPH classifier could serve as an initial guess for the iterative pose estimation algorithm, improving the likelihood of convergence and reducing the time required for convergence.

7.3 CONCLUDING REMARKS

The research presented in this document advanced object recognition and gross pose estimation capabilities on challenging objects. The major contributions of this work are:

- The **Cylindrical Projection Histogram** – A scale invariant point cluster descriptor that encodes shape, size, and pose for object recognition and pose estimation.
- ROS-compatible CPH code that is publicly available from Southwest Research Institute's ROS repository: swri-ros-pkg. It will also be included with the unstable trunk of PCL.
- The **probabilistic object model** – A non-deterministic model that quantifies the uncertainty in the robot's internal model of the environment and permits statistical refinement from the robot's sensor suite.
- The **LANL object dataset** – A small but challenging dataset that contains reflective objects, and includes several objects with similar shapes and different sizes. This dataset is publicly available by contacting the UT Nuclear Robotics Group.
- A prototype GUI interface that communicates information about the model quality to the system operator. This allows the operator to make decisions regarding the appropriate level of autonomy.

These contributions lay the groundwork for a formalized decision making framework for transitional autonomy by providing the ability to quantify environmental uncertainty. And while this document ends without *fully* solving the robot vision problem, it does provide a strong foundation upon which the UT NRG can continue to develop advanced perception capabilities that support deployment of manipulators for unstructured

materials handling tasks in DOE facilities. It also provides for public consumption a new avenue for the vision community to further explore solutions for autonomous object recognition and pose estimation.

Appendix A: CPH recognition ROS package code

The following code is also available on the NRG code repository as well as Southwest Research Institute's ROS repository.

cph.h

```
#include <pcl/point_types.h>
#include <pcl/point_cloud.h>
#include <pcl/common/transforms.h>
#include <pcl/io/pcd_io.h>
#include <iostream>
#include <flann/flann.h>
#include <math.h>

#define PI 3.14159265

class CPHEstimation{

public:

  CPHEstimation(){
   num_ybins = 5;
   num_cbins = 72;
   hist.resize(num_ybins * num_cbins,0);
   centroid.resize(3,0);
  };

  CPHEstimation(int num_y, int num_c){
   num_ybins = num_y;
   num_cbins = num_c;
   hist.resize(num_ybins * num_cbins,0);
   centroid.resize(3,0);
  };

  bool setInputCloud(pcl::PointCloud<pcl::PointXYZ>::Ptr inputCloud){
   cloud = inputCloud;
   return 1;
  };

int compute(std::vector<float> &result){
  //compute bouding box size:
  float x_max=0, x_min=100, y_max=0, y_min = 100, z_max=0, z_min=100;
  for(unsigned int i=0; i<cloud->size(); i++){
    if(cloud->points.at(i).x > x_max)
      x_max = cloud->points.at(i).x;
    if(cloud->points.at(i).x < x_min)
      x_min = cloud->points.at(i).x;
    if(cloud->points.at(i).y > y_max)
```

```
      y_max = cloud->points.at(i).y;
    if(cloud->points.at(i).y < y_min)
      y_min = cloud->points.at(i).y;
    if(cloud->points.at(i).z > z_max)
      z_max = cloud->points.at(i).z;
    if(cloud->points.at(i).z < z_min)
      z_min = cloud->points.at(i).z;
  }

  x_size = (x_max - x_min);
  y_size = (y_max - y_min);
  z_size = (z_max - z_min);

  float max_size = x_size;
  if(y_size > max_size)
    max_size = y_size;
  if(z_size > max_size)
    max_size = z_size;
  max_size*=100;

  centroid.at(0) = x_min + x_size/2;
  centroid.at(1) = y_min + y_size/2;
  centroid.at(2) = z_min + z_size/2;

  //compute feature
  hist.clear();
  hist.resize(num_cbins*num_ybins, 0.0f);
  int y,c;
  float dy = y_size/num_ybins;
  float dc = 2*PI/num_cbins;

  for(unsigned int i=0; i<cloud->size(); i++){
    //bin z component
    y = floor((cloud->points.at(i).y-y_min)/dy);
    c = floor((PI+atan2(cloud->points.at(i).z-centroid.at(2),cloud-
>points.at(i).x-centroid.at(0)))/dc);
    if(y*num_cbins+c < hist.size())
      hist.at(y*num_cbins+c)+=1.0f;
  }

  //Find tallest peak
  float max_peak = 0;
  for(unsigned int i=0; i<hist.size(); i++){
    if(hist.at(i) > max_peak)
     max_peak = (float)hist.at(i);
  }

  //Rescale cph to largest spatial extent:
  float scaleFactor = max_size/max_peak;
  for(unsigned int i=0; i<hist.size(); i++){
    hist.at(i)*=scaleFactor;
```

```
  }

  //Concatenate epatial extents
  hist.push_back(x_size*100);
  hist.push_back(y_size*100);
  hist.push_back(z_size*100);

  result.clear();
  result = hist;
  return result.size();
};

private:

  std::vector<float> hist;
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud, result;
  int num_ybins, num_cbins;
  float x_size, y_size, z_size;
  std::vector<float> centroid;
};
```

cph_recognition_node.cpp:

```
#include <pcl/point_types.h>
#include <pcl/point_cloud.h>
#include <pcl/common/transforms.h>
#include <pcl/console/parse.h>
#include <pcl/console/print.h>
#include <pcl/io/pcd_io.h>

#include <iostream>
#include <fstream>

#include <flann/flann.h>

#include <boost/filesystem.hpp>
#include <boost/random/mersenne_twister.hpp>
#include <boost/random/normal_distribution.hpp>
#include <boost/random/variate_generator.hpp>

#include <nrg_object_recognition/recognition.h>

#include "ros/ros.h"
#include "sensor_msgs/PointCloud2.h"
#include "cph.h"



typedef std::pair<std::string, std::vector<float> > cph_model;
std::vector<cph_model> models;
flann::Matrix<int> k_indices;
```

```
flann::Matrix<float> k_distances;
flann::Matrix<float> *data;
sensor_msgs::PointCloud2 fromKinect;
ros::Publisher pub;
std::ofstream outFile;
int segCount = 0;
int num_ybins = 5; int num_rbins = 72;
int histSize = num_ybins*num_rbins+3;

inline void
nearestKSearch (flann::Index<flann::ChiSquareDistance<float> > &index, const
cph_model &model,
                int k, flann::Matrix<int> &indices, flann::Matrix<float>
&distances)
{
  // Query point
  flann::Matrix<float> p = flann::Matrix<float>(new float[model.second.size
()], 1, model.second.size ());
  memcpy (&p.ptr ()[0], &model.second.at(0), p.cols * p.rows * sizeof (int));

  indices = flann::Matrix<int>(new int[k], 1, k);
  distances = flann::Matrix<float>(new float[k], 1, k);
  index.knnSearch (p, indices, distances, k, flann::SearchParams (512));
  delete[] p.ptr ();
}

bool
loadHist (const boost::filesystem::path &path, cph_model &cph)
{
  //path is the location of the file being read.
  std::ifstream featureFile;
  featureFile.open(path.string().c_str(), std::ifstream::in);
  cph.second.clear();
  float value;
  for(unsigned int i=0; i<histSize; i++){
   featureFile >> value;
   cph.second.push_back(value);
  }
  cph.first = path.string ();

  return (true);
}

void
loadFeatureModels (const boost::filesystem::path &base_dir, const std::string
&extension,
                   std::vector<cph_model> &models)
{
  if (!boost::filesystem::exists (base_dir) &&
!boost::filesystem::is_directory (base_dir))
    return;
```

```
  for (boost::filesystem::directory_iterator it (base_dir); it !=
boost::filesystem::directory_iterator (); ++it)
  {
    if (boost::filesystem::is_directory (it->status ()))
    {
      std::stringstream ss;
      ss << it->path ();
      pcl::console::print_highlight ("Loading %s (%lu models loaded so
far).\n", ss.str ().c_str (), (unsigned long)models.size ());
      loadFeatureModels (it->path (), extension, models);
    }
    if (boost::filesystem::is_regular_file (it->status ()) &&
boost::filesystem::extension (it->path ()) == extension)
    {
      cph_model m;
      if (loadHist (base_dir / it->path ().filename (), m))
        models.push_back (m);
    }
  }
}


bool recognize_cb(nrg_object_recognition::recognition::Request &srv_request,
                  nrg_object_recognition::recognition::Response &srv_response)
{
  //create knn index
  flann::Index<flann::ChiSquareDistance<float> > index (*data,
flann::LinearIndexParams ());
  index.buildIndex ();

  pcl::PointCloud<pcl::PointXYZ>::Ptr cluster (new
pcl::PointCloud<pcl::PointXYZ>);
  pcl::fromROSMsg(srv_request.cluster, *cluster);

  // Create cph estimation.
  CPHEstimation cph(num_ybins,num_rbins);

  //Hold results:
  std::string label, angleStr;
  int angle;
  Eigen::Vector4f translation;

  //Demean the cloud.
  Eigen::Vector4f centroid;
  pcl::compute3DCentroid (*cluster, centroid);
  srv_response.pose.x = centroid(0);
  srv_response.pose.y = centroid(1);
  srv_response.pose.z = centroid(2);
```

```
//Compute cph:
  cph.setInputCloud (cluster);
  std::vector<float> feature;
  cph.compute(feature);

  int k = 1; //number of neighbors
  cph_model histogram;
  histogram.second.resize(histSize);

  for (size_t i = 0; i < histSize; ++i)
  {
      histogram.second[i] = feature.at(i);
  }

  //KNN classification
  nearestKSearch (index, histogram, k, k_indices, k_distances);

  //determine label and pose:
  if(k_distances[0][0] < srv_request.threshold){
    //Load nearest match
    std::string cloud_name = models.at(k_indices[0][0]).first;

    cloud_name.erase(cloud_name.end()-3, cloud_name.end());
    angleStr.assign(cloud_name.begin()+cloud_name.rfind("_")+1,
cloud_name.end());
    std::string label;
    label.assign(cloud_name.begin()+5,
cloud_name.begin()+cloud_name.rfind("_"));
    angle = atoi(angleStr.c_str());
    srv_response.label = label;
    srv_response.pose.rotation = angle;
  }
  return(1);
}


int main(int argc, char **argv)
{
  ros::init(argc, argv, "cph_recongition_node");
  ros::NodeHandle n;

  loadFeatureModels (argv[1], ".csv", models);
  pcl::console::print_highlight ("Loaded %d VFH models. Creating training
data\n",
      (int)models.size ());

  // Convert data into FLANN format
  data = new flann::Matrix<float> (new float[models.size () *
models[0].second.size ()], models.size (), models[0].second.size ());

  std::cout << "data size: [" << data->rows << " , " << data->cols << "]\n";
```

```
  for (size_t i = 0; i < data->rows; ++i)
    for (size_t j = 0; j < data->cols; ++j)
      *(data->ptr()+(i*data->cols + j)) = models[i].second[j];

  pcl::console::print_error ("Training data loaded.\n");

  ros::ServiceServer serv = n.advertiseService("/cph_recognition",
recognize_cb);

  ROS_INFO("cph_recognition_node ready.");

  ros::spin();

  return(1);
}
```

vfh_recognition_node.cpp

```
#include <pcl/point_types.h>
#include <pcl/point_cloud.h>
#include <pcl/common/transforms.h>
#include <pcl/console/parse.h>
#include <pcl/console/print.h>
#include <pcl/io/pcd_io.h>
#include <pcl/features/vfh.h>
#include <pcl/features/normal_3d.h>

#include <iostream>
#include <fstream>

#include <flann/flann.h>
#include <boost/filesystem.hpp>
#include <boost/random/mersenne_twister.hpp>
#include <boost/random/normal_distribution.hpp>
#include <boost/random/variate_generator.hpp>

#include <nrg_object_recognition/recognition.h>
#include "ros/ros.h"
#include "sensor_msgs/PointCloud2.h"


typedef std::pair<std::string, std::vector<float> > vfh_model;
std::vector<vfh_model> models;
flann::Matrix<int> k_indices;
flann::Matrix<float> k_distances;
flann::Matrix<float> *data;
//ros::Publisher recognized_pub;
sensor_msgs::PointCloud2 fromKinect;
ros::Publisher pub;

inline void
```

```
nearestKSearch (flann::Index<flann::ChiSquareDistance<float> > &index, const
vfh_model &model,
                int k, flann::Matrix<int> &indices, flann::Matrix<float>
&distances)
{
  // Query point
  flann::Matrix<float> p = flann::Matrix<float>(new float[model.second.size
()], 1, model.second.size ());
  memcpy (&p.ptr ()[0], &model.second[0], p.cols * p.rows * sizeof (float));

  indices = flann::Matrix<int>(new int[k], 1, k);
  distances = flann::Matrix<float>(new float[k], 1, k);
  index.knnSearch (p, indices, distances, k, flann::SearchParams (512));
  delete[] p.ptr ();
}

bool
loadFileList (std::vector<vfh_model> &models, const std::string &filename)
{
  std::ifstream fs;
  fs.open (filename.c_str ());
  if (!fs.is_open () || fs.fail ())
    return (false);

  std::string line;
  while (!fs.eof ())
  {
    getline (fs, line);
    if (line.empty ())
      continue;
    vfh_model m;
    m.first = line;
    models.push_back (m);
  }
  fs.close ();
  return (true);
}

bool
loadHist (const boost::filesystem::path &path, vfh_model &vfh)
{
  int vfh_idx;
  // Load the file as a PCD
  try
  {
    sensor_msgs::PointCloud2 cloud;
    int version;
    Eigen::Vector4f origin;
    Eigen::Quaternionf orientation;
    pcl::PCDReader r;
    int type;  int idx;
```

```
    r.readHeader (path.string (), cloud, origin, orientation, version, type,
idx);

    vfh_idx = pcl::getFieldIndex (cloud, "vfh");
    if (vfh_idx == -1)
      return (false);
    if ((int)cloud.width * cloud.height != 1)
      return (false);
  }
  catch (pcl::InvalidConversionException e)
  {
    return (false);
  }

  // Treat the VFH signature as a single Point Cloud
  pcl::PointCloud <pcl::VFHSignature308> point;
  pcl::io::loadPCDFile (path.string (), point);
  vfh.second.resize (308);

  std::vector <sensor_msgs::PointField> fields;
  pcl::getFieldIndex (point, "vfh", fields);

  for (size_t i = 0; i < fields[vfh_idx].count; ++i)
  {
    vfh.second[i] = point.points[0].histogram[i];
  }
  vfh.first = path.string ();
  return (true);
}

void
loadFeatureModels (const boost::filesystem::path &base_dir, const std::string
&extension,
                   std::vector<vfh_model> &models)
{
  if (!boost::filesystem::exists (base_dir) &&
!boost::filesystem::is_directory (base_dir))
    return;

  for (boost::filesystem::directory_iterator it (base_dir); it !=
boost::filesystem::directory_iterator (); ++it)
  {
    if (boost::filesystem::is_directory (it->status ()))
    {
      std::stringstream ss;
      ss << it->path ();
      pcl::console::print_highlight ("Loading %s (%lu models loaded so
far).\n", ss.str ().c_str (), (unsigned long)models.size ());
      loadFeatureModels (it->path (), extension, models);
    }
```

```
    if (boost::filesystem::is_regular_file (it->status ()) &&
boost::filesystem::extension (it->path ()) == extension)
    {
      vfh_model m;
      if (loadHist (base_dir / it->path ().filename (), m))
        models.push_back (m);
    }
  }
}

bool recognize_cb(nrg_object_recognition::recognition::Request &srv_request,
                  nrg_object_recognition::recognition::Response &srv_response)
{

  //create knn index
  flann::Index<flann::ChiSquareDistance<float> > index (*data,
flann::LinearIndexParams ());
  index.buildIndex ();

  pcl::PointCloud<pcl::PointXYZ>::Ptr cluster (new
pcl::PointCloud<pcl::PointXYZ>);
  pcl::fromROSMsg(srv_request.cluster, *cluster);

  // Create vfh estimation.
  pcl::VFHEstimation<pcl::PointXYZ, pcl::Normal, pcl::VFHSignature308> vfh;

  //Hold results:
  std::string label, angleStr;
  int angle;
  Eigen::Vector4f translation;

  //Demean the cloud.
  Eigen::Vector4f centroid;
  pcl::compute3DCentroid (*cluster, centroid);
  srv_response.pose.x = centroid(0);
  srv_response.pose.y = centroid(1);
  srv_response.pose.z = centroid(2);

  vfh.setInputCloud (cluster);
  //Estimate normals:
  pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> ne;
  ne.setInputCloud (cluster);
  pcl::search::KdTree<pcl::PointXYZ>::Ptr treeNorm (new
pcl::search::KdTree<pcl::PointXYZ> ());
  ne.setSearchMethod (treeNorm);
  pcl::PointCloud<pcl::Normal>::Ptr cloud_normals (new
pcl::PointCloud<pcl::Normal>);
  ne.setRadiusSearch (0.03);
  ne.compute (*cloud_normals);

  //VFH estimation
```

```
  vfh.setInputNormals (cloud_normals);
  pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new
pcl::search::KdTree<pcl::PointXYZ> ());
  vfh.setSearchMethod (tree);
  pcl::PointCloud<pcl::VFHSignature308>::Ptr vfhs (new
pcl::PointCloud<pcl::VFHSignature308> ());
  vfh.compute (*vfhs);

  //Load histogram
  vfh_model histogram;
  histogram.second.resize(308);
  for (size_t i = 0; i < 308; ++i)
  {
    histogram.second[i] = vfhs->points[0].histogram[i];
  }

  //Algorithm parameters
  int k = 1; //number of neighbors
  nearestKSearch (index, histogram, k, k_indices, k_distances);

  //determine label and pose:
  if(k_distances[0][0] < srv_request.threshold){
    //Load nearest match
    std::string cloud_name = models.at(k_indices[0][0]).first;
    cloud_name.erase(cloud_name.end()-8, cloud_name.end()-4);
    std::string recognitionLabel, viewNumber;
    recognitionLabel.assign(cloud_name.begin()+5,
cloud_name.begin()+cloud_name.rfind("_"));
    viewNumber.assign(cloud_name.begin()+cloud_name.rfind("_")+1,
cloud_name.end()-4);
    srv_response.label = recognitionLabel;
    srv_response.pose.rotation = atof(viewNumber.c_str());
  }
  return(1);
}


int main(int argc, char **argv)
{

  ros::init(argc, argv, "vfh_recongition_node");
  ros::NodeHandle n;

  loadFeatureModels (argv[1], ".pcd", models);
  pcl::console::print_highlight ("Loaded %d VFH models. Creating training
data\n",
      (int)models.size ());

  // Convert data into FLANN format
  data = new flann::Matrix<float> (new float[models.size () *
models[0].second.size ()], models.size (), models[0].second.size ());
```

```
  std::cout << "data size: [" << data->rows << " , " << data->cols << "]\n";
  for (size_t i = 0; i < data->rows; ++i)
    for (size_t j = 0; j < data->cols; ++j)
      *(data->ptr()+(i*data->cols + j)) = models[i].second[j];

  pcl::console::print_error ("Training data loaded.\n");

  ros::ServiceServer serv = n.advertiseService("/vfh_recognition",
recognize_cb);

  ROS_INFO("vfh_recognition_node ready.");

  ros::spin();

  return(1);
}
```

euclidean_segmentation.cpp

```
#include "ros/ros.h"
#include "nrg_object_recognition/segmentation.h"

#include <pcl/ModelCoefficients.h>
#include <pcl/point_types.h>
#include <pcl/io/pcd_io.h>
#include <pcl/filters/extract_indices.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/features/normal_3d.h>
#include <pcl/kdtree/kdtree.h>
#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/segmentation/extract_clusters.h>

ros::Publisher plane_pub;
ros::Publisher bound_pub;
ros::Publisher cluster_pub;

bool segment_cb(nrg_object_recognition::segmentation::Request &seg_request,
            nrg_object_recognition::segmentation::Response &seg_response)
{
  // Read in the cloud data
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new
pcl::PointCloud<pcl::PointXYZ>), cloud_f (new pcl::PointCloud<pcl::PointXYZ>);
  std::cout << "segmenting image..." << std::endl;
  pcl::fromROSMsg(seg_request.scene, *cloud);
```

```
// Create the filtering object: downsample the dataset using a leaf size of
1cm
  pcl::VoxelGrid<pcl::PointXYZ> vg;
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new
pcl::PointCloud<pcl::PointXYZ>), cloud_filtered_0 (new
pcl::PointCloud<pcl::PointXYZ>);
  vg.setInputCloud (cloud);
  vg.setLeafSize (0.001f, 0.001f, 0.001f);
  vg.filter (*cloud_filtered_0);

  // Create the segmentation object for the planar model and set all the
parameters
  pcl::SACSegmentation<pcl::PointXYZ> seg;
  pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
  pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_plane (new
pcl::PointCloud<pcl::PointXYZ> ());
  pcl::PCDWriter writer;
  seg.setOptimizeCoefficients (true);
  seg.setModelType (pcl::SACMODEL_PLANE);
  seg.setMethodType (pcl::SAC_RANSAC);
  seg.setMaxIterations (100);
  seg.setDistanceThreshold (0.01);
  //Spatial filter.
  cloud_filtered->resize(0);
  //Parameters
 float min_x = seg_request.min_x, max_x = seg_request.max_x;
 float min_y = seg_request.min_y, max_y = seg_request.max_y;
 float min_z = seg_request.min_z, max_z = seg_request.max_z;

 for(pcl::PointCloud<pcl::PointXYZ>::iterator position=cloud_filtered_0-
>begin(); position!=cloud_filtered_0->end(); position++){
   if(position->x > min_x && position->x < max_x && position->y >
(2.145*position->z - 3.2) && position->y < (-.466*position->z + .500))
     cloud_filtered->push_back(*position);
  }
  sensor_msgs::PointCloud2 cloud_filtered_pc2;
  pcl::toROSMsg(*cloud_filtered, cloud_filtered_pc2);
  cloud_filtered_pc2.header.frame_id = "/camera_depth_optical_frame";
  bound_pub.publish(cloud_filtered_pc2);

  std::cout << "num points in spatially filtered cloud: " << cloud_filtered-
>points.size() << std::endl;
  int i=0, nr_points = (int) cloud_filtered->points.size ();
  while (cloud_filtered->points.size () > 0.5 * nr_points)
  {
    // Segment the largest planar component from the remaining cloud
    seg.setInputCloud (cloud_filtered);
    seg.segment (*inliers, *coefficients);
    if (inliers->indices.size () == 0)
    {
```

```
      std::cout << "Could not estimate a planar model for the given dataset."
<< std::endl;
      break;
    }

    // Extract the planar inliers from the input cloud
    pcl::ExtractIndices<pcl::PointXYZ> extract;
    extract.setInputCloud (cloud_filtered);
    extract.setIndices (inliers);
    extract.setNegative (false);

    // Publish dominant plane
    extract.filter (*cloud_plane);
    sensor_msgs::PointCloud2 plane_pc2;
    pcl::toROSMsg(*cloud_plane, plane_pc2);
    plane_pc2.header.frame_id = "/camera_depth_optical_frame";
    plane_pub.publish(plane_pc2);

    // Remove the planar inliers, extract the rest
    extract.setNegative (true);
    extract.filter (*cloud_f);
    cloud_filtered = cloud_f;
  }

  pcl::toROSMsg(*cloud_filtered, cloud_filtered_pc2);
  cloud_filtered_pc2.header.frame_id = "/camera_depth_optical_frame";
  cluster_pub.publish(cloud_filtered_pc2);

   std::cout << "Number of points in remaining clusters: " << cloud_filtered-
>points.size()  << std::endl;
  // Creating the KdTree object for the search method of the extraction
  if(cloud_filtered->points.size() > 50){
    pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new
pcl::search::KdTree<pcl::PointXYZ>);
    tree->setInputCloud (cloud_filtered);

    std::vector<pcl::PointIndices> cluster_indices;
    pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;
    ec.setClusterTolerance (0.03); // 2cm
    ec.setMinClusterSize (75);
    ec.setMaxClusterSize (25000);
    ec.setSearchMethod (tree);
    ec.setInputCloud (cloud_filtered);
    std::cout << "extracting clusters...\n";
    ec.extract (cluster_indices);

    std::cout << "length of cluster_indices: " << cluster_indices.size() <<
std::endl;
    int j = 0;
    for (std::vector<pcl::PointIndices>::const_iterator it =
cluster_indices.begin (); it != cluster_indices.end (); ++it)
```

```
    {
      pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_cluster (new
pcl::PointCloud<pcl::PointXYZ>);
      for (std::vector<int>::const_iterator pit = it->indices.begin (); pit !=
it->indices.end (); pit++)
       cloud_cluster->points.push_back (cloud_filtered->points[*pit]); //*
      cloud_cluster->width = cloud_cluster->points.size ();
      cloud_cluster->height = 1;
      cloud_cluster->is_dense = true;
      std::cout << "writing cluster to service response. It has " <<
cloud_cluster->points.size() << " points.\n";
      sensor_msgs::PointCloud2 tempROSMsg;
      pcl::toROSMsg(*cloud_cluster, tempROSMsg);
      seg_response.clusters.push_back(tempROSMsg);
      j++;
    }
  }
  return (1);
}
int main(int argc, char **argv)
{

  ros::init(argc, argv, "segmentation_node");
  ros::NodeHandle n;

  plane_pub = n.advertise<sensor_msgs::PointCloud2>("/dominant_plane",1);
  bound_pub = n.advertise<sensor_msgs::PointCloud2>("/bounded_scene",1);
  cluster_pub = n.advertise<sensor_msgs::PointCloud2>("/pre_clustering",1);
  ros::ServiceServer serv = n.advertiseService("/segmentation", segment_cb);

  ros::spin();

}
```

main_dataset_node.cpp

```
#include "ros/ros.h"
#include "std_msgs/UInt16.h"
#include <visualization_msgs/Marker.h>

#include <pcl/point_types.h>
#include <pcl/point_cloud.h>
#include <pcl/common/transforms.h>
#include <pcl/console/parse.h>
#include <pcl/console/print.h>
#include <pcl/io/pcd_io.h>

#include <iostream>
#include <fstream>
#include <math.h>
#include <sstream>
```

```
#include <flann/flann.h>

#include <boost/filesystem.hpp>
#include <boost/random/mersenne_twister.hpp>
#include <boost/random/normal_distribution.hpp>
#include <boost/random/variate_generator.hpp>

#include "cph.h"
#include "nrg_object_recognition/run_data.h"
#include "nrg_object_recognition/recognition.h"
#include "nrg_object_recognition/segmentation.h"

float subtract_angle(float angle_1, float angle_2);

ros::ServiceClient cph_client, vfh_client, seg_client;
std::vector<std::vector<float> > probTable;
std::map<std::string, int> classMap;
std::vector<float> pose_dev;
int k;

ros::Publisher pan_pub;
ros::Publisher rec_pub;
ros::Publisher text_pub;
sensor_msgs::PointCloud2 cloud_to_process;

void kinect_cb(sensor_msgs::PointCloud2 fromKinect)
{
cloud_to_process = fromKinect;
}


bool test_cb(nrg_object_recognition::run_data::Request &main_request,
          nrg_object_recognition::run_data::Response &main_response)
{
  std::string objectName = main_request.object_name;
  std::stringstream fileName;
  std::vector<float> pcc_row(k,0); //will hold class conditional
probabilities. P(c|testObject)
  std::vector<float> filtered_result(k,0); //holds filtered result.

  nrg_object_recognition::recognition rec_srv;
  int num_objects = 0;
  std::string angleStr;
  float angle=0, pose_err = 0, cum_err=0, filtered_pose_err=0, cum_filt_dev=0;

  //Noise generator:
  boost::mt19937 mers;
  mers.seed(static_cast<unsigned int>(std::time(0)));
  boost::normal_distribution<float> dist(0, main_request.noise_level);
```

```
  boost::variate_generator<boost::mt19937 , boost::normal_distribution<float>
> noise(mers,dist);

  std::cout << "test running...\n";
  int avgSize = 0;
  int size_idx = 0;
  //For each test file:
  for(boost::filesystem::directory_iterator it ("test"); it !=
boost::filesystem::directory_iterator (); ++it){
    if (boost::filesystem::is_regular_file (it->status ()) &&
boost::filesystem::extension (it->path ()) == ".pcd")
    {
      //Set up probabilistic filters:
      std::vector<float> classProb;
      Eigen::Matrix4f K, Sigma, Q, I;
      Eigen::Vector4f pose;
      Q(0,0) = .005f; Q(1,1) = .005f; Q(2,2) = .005f; Q(3,3) = 0.1;
      I.setIdentity();

      //start with uniform prior
      classProb.clear();
      classProb.resize(7,(float)1/7.0f);

      //Start with high covariance:
      Sigma(0,0) = 1000; Sigma(1,1) = 1000; Sigma(2,2) = 1000; Sigma(3,3) =
1000;

      //Hold results:
      std::string label, angleStr;
      int angle, z;
      Eigen::Vector4f translation;

      int cloudSize=0;

      //Iterate through noisy samples
      for(unsigned int j=0; j<main_request.num_samples; j++){

       //Read .pcd file.
       fileName.str("");
       fileName << "test/" << it->path().filename().c_str();
       pcl::PointCloud<pcl::PointXYZ>::Ptr cluster (new
pcl::PointCloud<pcl::PointXYZ>);
       pcl::io::loadPCDFile (fileName.str(), *cluster);

       //Add noise to the z channel.
       for(size_t idx = 0; idx < cluster->points.size(); idx++){
         cluster->points[idx].z += noise();
       }

       pcl::toROSMsg(*cluster, rec_srv.request.cluster);
```

```
        rec_srv.request.threshold = 10000; //(nearest neighbor no matter how
far.)

        if(main_request.method == 0){
         cph_client.call(rec_srv);
        }
        else if(main_request.method == 1){
         vfh_client.call(rec_srv);
        }
        num_objects++;

        //Increment appropriate row.
        z = classMap[rec_srv.response.label];
        pcc_row.at(z)++;

        //compute error in pose estimate, and accumulate squared error.
        std::string fileNameString = fileName.str();
        angleStr.assign(fileNameString.begin()+fileNameString.rfind("_")+1,
fileNameString.end()-3);
        angle = atof(angleStr.c_str()); //angle is ground truth angle
        translation(0) = rec_srv.response.pose.x; translation(1) =
rec_srv.response.pose.y; translation(2) = rec_srv.response.pose.z;
        pose_err = subtract_angle(angle, rec_srv.response.pose.rotation);
        cum_err += pow(pose_err, 2);

        //Filter results:
        //Bayes filter for class probability:
         //Compute denominator:
         float bayesDen = 0;
        for(unsigned int ck=0; ck<7; ck++){
          bayesDen += probTable.at(ck).at(z)*classProb.at(ck);
        }
        //Update P(C|z)
        for(unsigned int cj=0; cj<7; cj++){
         classProb.at(cj) = probTable.at(cj).at(z)*classProb.at(cj)/bayesDen;
        }

        //Filter the pose estimate info in:
        if(j==0){
          pose(3) = rec_srv.response.pose.rotation;
        }

        else{
          Q(3,3) = pose_dev.at(z);
          std::cout << "Q: " << Q << std::endl;
          K = Sigma*(Sigma + Q).inverse();
          std::cout << "K: " << K << std::endl;
          float newAngle = pose(3) +
K(3,3)*subtract_angle(rec_srv.response.pose.rotation, pose(3));
          std::cout << "newAngle: " << newAngle << std::endl;
          if(newAngle < 0)
```

```
          newAngle += 360;
        pose = pose + K*(translation-pose);
        if(newAngle > 360)
          newAngle -= 360;
        pose(3) = newAngle;
        Sigma = (I-K)*Sigma;
        std::cout << "Sigma: " << Sigma << std::endl;
       }
       std::cout << "pose: " << pose(3) << std::endl;
      }
      //increment bin of label
      for(unsigned int class_it = 0; class_it < 7; class_it++){
       if(classProb.at(class_it) > classProb.at(z))
        z = class_it;
      }
      filtered_result.at(z)++;
      cum_filt_dev += pow(subtract_angle(pose(3),angle),2);//Sigma(3,3);
    }
  }
  std::cout << "done.\n";
  main_response.sigma_pose = pow(cum_err/num_objects, .5);
  for(unsigned int i=0; i<pcc_row.size(); i++){
    pcc_row.at(i) = pcc_row.at(i)/num_objects;
  }
  for(unsigned int i=0; i<pcc_row.size(); i++){
    filtered_result.at(i) =
main_request.num_samples*filtered_result.at(i)/num_objects;
  }

  main_response.rec_rate = pcc_row.at(classMap[objectName]);
  main_response.prob_dist = pcc_row;
  main_response.sigma_filtered =
pow(main_request.num_samples*cum_filt_dev/num_objects,.5);
  main_response.filt_dist = filtered_result;

  return(1);
}

bool live_cb(nrg_object_recognition::run_data::Request &main_request,
          nrg_object_recognition::run_data::Response &main_response)
{
  std_msgs::UInt16 command;
  command.data = 0;
  ros::Rate loop_rate(.1);
  pan_pub.publish(command);
  loop_rate.sleep();

  std::string objectName = main_request.object_name;
  std::vector<float> pcc_row(k,0); //will hold class conditional
probabilities. P(c|testObject)
  std::vector<float> filtered_result(k,0); //holds filtered result.
```

```
  nrg_object_recognition::recognition rec_srv;
  nrg_object_recognition::segmentation seg_srv;
  int num_objects = 0;
  std::string angleStr;
  float angle=0, pose_err = 0, cum_err=0, filtered_pose_err=0, cum_filt_dev=0;

  std::cout << "test running...\n";
  //For each view
  for(unsigned int i=0; i<main_request.num_images; i++){
      //Set up probabilistic filters:
      std::vector<float> classProb;
      Eigen::Matrix4f K, Sigma, Q, I;
      Eigen::Vector4f pose;
      Q(0,0) = .005f; Q(1,1) = .005f; Q(2,2) = .005f; Q(3,3) = 0.1;
      I.setIdentity();

      //start with uniform prior
      classProb.clear();
      classProb.resize(k,(float)1/7.0f);

      //Start with high covariance:
      Sigma(0,0) = 1000; Sigma(1,1) = 1000; Sigma(2,2) = 1000; Sigma(3,3) =
1000;

      //Hold results:
      std::string label, angleStr;
      int angle, z;
      Eigen::Vector4f translation;

      //Take several images.
      for(unsigned int j=0; j<main_request.num_samples; j++){
        ros::spinOnce();
       //Call segmentation service...
       seg_srv.request.scene = cloud_to_process;
       seg_srv.request.min_x = -.75, seg_srv.request.max_x = .4;
       seg_srv.request.min_y = -5, seg_srv.request.max_y = .5;
       seg_srv.request.min_z = 0.0, seg_srv.request.max_z = 1.15;
       seg_client.call(seg_srv);

       //could iterate through all clusters here in future app.
       if(seg_srv.response.clusters.size() > 0){
         rec_srv.request.cluster = seg_srv.response.clusters.at(0);
         rec_srv.request.threshold = 10000;
         //Call recognition service...
         if(main_request.method == 0){
         cph_client.call(rec_srv);
         }
         else if(main_request.method == 1){
         vfh_client.call(rec_srv);
         }
```

```
        num_objects++;

        //Increment appropriate row.
        z = classMap[rec_srv.response.label];
        pcc_row.at(z)++;

        //compute error in pose estimate, and accumulate squared error.
        angle = command.data; //angle is ground truth angle

        translation(0) = rec_srv.response.pose.x; translation(1) =
rec_srv.response.pose.y; translation(2) = rec_srv.response.pose.z;
        pose_err = subtract_angle(angle, rec_srv.response.pose.rotation);
        std::cout << "label: " << rec_srv.response.label << std::endl;
        cum_err += pow(pose_err, 2);

        //Visualization:////////////////////////////////////////////////////
        //build filename.
        std::stringstream fileName;
        fileName << "data/" << rec_srv.response.label << "_" <<
rec_srv.response.pose.rotation << ".pcd";
        //Load and convert file.
        pcl::PointCloud<pcl::PointXYZ>::Ptr trainingMatch (new
pcl::PointCloud<pcl::PointXYZ>);
        sensor_msgs::PointCloud2 rosMsg;
        pcl::io::loadPCDFile(fileName.str(), *trainingMatch);
        //Translate to location:
        Eigen::Vector4f centroid;
        pcl::compute3DCentroid(*trainingMatch, centroid);
        pcl::demeanPointCloud<pcl::PointXYZ> (*trainingMatch, centroid,
*trainingMatch);

        Eigen::Vector3f translate;
        Eigen::Quaternionf rotate;
        translate(0) = rec_srv.response.pose.x;
        translate(1) = rec_srv.response.pose.y;
        translate(2) = rec_srv.response.pose.z;
        rotate.setIdentity();
        pcl::transformPointCloud(*trainingMatch, *trainingMatch, translate,
rotate);

        pcl::toROSMsg(*trainingMatch, rosMsg);
        //Add transform to header
        rosMsg.header.frame_id = "/camera_depth_optical_frame";
        //Publish to topic /recognition_result.
        rec_pub.publish(rosMsg);

        ///end visualization////////////////////////////////////////////

        float bayesDen = 0;
        for(unsigned int ck=0; ck<7; ck++){
          bayesDen += probTable.at(ck).at(z)*classProb.at(ck);
```

```
        }
        //Update P(C|z)
        for(unsigned int cj=0; cj<k; cj++){
          classProb.at(cj) = probTable.at(cj).at(z)*classProb.at(cj)/bayesDen;
        }

        //Filter the pose estimate info in:
        if(j==0){
          pose(3) = rec_srv.response.pose.rotation;
        }

        else{
          Q(3,3) = pose_dev.at(z);
          std::cout << "Q: " << Q << std::endl;
          K = Sigma*(Sigma + Q).inverse();
          std::cout << "K: " << K << std::endl;
          float newAngle = pose(3) +
K(3,3)*subtract_angle(rec_srv.response.pose.rotation, pose(3));
          std::cout << "newAngle: " << newAngle << std::endl;
          if(newAngle < 0)
            newAngle += 360;
          pose = pose + K*(translation-pose);
          if(newAngle > 360)
            newAngle -= 360;
          pose(3) = newAngle;
          Sigma = (I-K)*Sigma;
          std::cout << "Sigma: " << Sigma << std::endl;
        }
        std::cout << "pose: " << pose(3) << std::endl;

         //Put result in rviz marker for visualization
         visualization_msgs::Marker object_text_marker;
         object_text_marker.header.frame_id = "/camera_depth_optical_frame";
         object_text_marker.header.stamp = ros::Time();
        object_text_marker.type =
visualization_msgs::Marker::TEXT_VIEW_FACING;
       object_text_marker.action = visualization_msgs::Marker::ADD;
        object_text_marker.pose.position.x = rec_srv.response.pose.x;
        object_text_marker.pose.position.y = rec_srv.response.pose.y-.10;
        object_text_marker.pose.position.z = rec_srv.response.pose.z;
        object_text_marker.pose.orientation.w = 1.0;

        object_text_marker.scale.x = .03;
        object_text_marker.scale.y = .03;
        object_text_marker.scale.z = .03;

       object_text_marker.color.a = 1.0;
       object_text_marker.color.r = 0.0;
       object_text_marker.color.g = 1.0;
       object_text_marker.color.b = 0.0;
```

```
        std::stringstream object_text_ss;
        object_text_ss << "Label: " << rec_srv.response.label << "\nP: " <<
classProb.at(z) << "\nPose: " << pose(3);
        object_text_marker.text = object_text_ss.str();
        text_pub.publish(object_text_marker);

      }
      } //end sample iterator
    //increment bin of label
    for(unsigned int class_it = 0; class_it < k; class_it++){
      if(classProb.at(class_it) > classProb.at(z))
       z = class_it;
    }
    filtered_result.at(z)++;
    cum_filt_dev += pow(subtract_angle(pose(3),angle),2);//Sigma(3,3);

    command.data += 360/main_request.num_images;
    pan_pub.publish(command);
    loop_rate.sleep();
  }//end image iterator
  std::cout << "done.\n";
  main_response.sigma_pose = pow(cum_err/num_objects, .5);
  for(unsigned int i=0; i<pcc_row.size(); i++){
    pcc_row.at(i) = pcc_row.at(i)/num_objects;
  }
  for(unsigned int i=0; i<pcc_row.size(); i++){
    filtered_result.at(i) = filtered_result.at(i)/main_request.num_images;
  }

  main_response.rec_rate = pcc_row.at(classMap[objectName]);
  main_response.prob_dist = pcc_row;
  main_response.sigma_filtered =
pow(main_request.num_samples*cum_filt_dev/num_objects,.5);
  main_response.filt_dist = filtered_result;

  return(1);
}


int main(int argc, char **argv)
{

  ros::init(argc, argv, "main_dataset_node");
  ros::NodeHandle n;

  ros::ServiceServer offline_serv = n.advertiseService("/run_test", test_cb);
  ros::ServiceServer live_serv = n.advertiseService("/live_test", live_cb);
  ros::ServiceServer marker_test = n.advertiseService("/marker_test",
marker_cb);
```

```
  cph_client =
n.serviceClient<nrg_object_recognition::recognition>("cph_recognition");
  vfh_client =
n.serviceClient<nrg_object_recognition::recognition>("vfh_recognition");
  seg_client =
n.serviceClient<nrg_object_recognition::segmentation>("segmentation");

  ros::Subscriber kin_sub = n.subscribe("/camera/depth_registered/points", 1,
kinect_cb);
  pan_pub = n.advertise<std_msgs::UInt16>("/pan_command",1);
  rec_pub = n.advertise<sensor_msgs::PointCloud2>("/recognition_result",1);
  text_pub = n.advertise<visualization_msgs::Marker>("/object_text", 1);

  std::ifstream objectListFile;
  objectListFile.open(argv[1], std::ios::in);
  std::string tempName, objectLine;
  k = 0;
  std::vector<float> probRow;

  std::cout << "reading file...\n";
  while(std::getline(objectListFile, objectLine)){
    std::istringstream objectSS(objectLine);
    //Record object name and add to map.
    objectSS >> tempName;
    std::cout << tempName << std::endl;
    classMap.insert(std::pair<std::string, int>(tempName, k));
    //Get pose sigma
    objectSS >> tempName;
    pose_dev.push_back(atof(tempName.c_str()));

    //populate probability table (line k;)
    while(objectSS >> tempName){
      probRow.push_back(atof(tempName.c_str()));
    }
    probTable.push_back(probRow);
    probRow.clear();
    k++;
  }
  std::cout << "done\n";
  if(k != probTable.at(0).size())
    std::cout << "Error: Probability table is not square!" << std::endl;

  for(unsigned int i=0; i<k; i++){
   for(unsigned int j=0; j<k; j++){
     std::cout << probTable.at(i).at(j) << " ";
   }
   std::cout << std::endl;
  }

  std::cout << probTable.size() << " objects in set.\n";
```

```
  ROS_INFO("main_dataset_node ready!");
  ros::spin();
}
float subtract_angle(float angle_1, float angle_2){
 if(angle_1-angle_2 > -180 && angle_1-angle_2 < 180)
   return(angle_1-angle_2);
 else if(angle_1-angle_2 < -180)
   return(angle_1-angle_2+360);
 else
   return(angle_1-angle_2-360);
}
```

pose.msg

```
float32 x
float32 y
float32 z
float32 rotation
```

recognition.srv:

```
sensor_msgs/PointCloud2 cluster
float32 threshold
---
string label
nrg_object_recognition/pose pose
```

segmentation.srv

```
sensor_msgs/PointCloud2 scene
float32 min_x
float32 max_x
float32 min_y
float32 max_y
float32 min_z
float32 max_z
---
sensor_msgs/PointCloud2[] clusters
```

run_data.srv

```
string object_name
uint16 method
uint16 num_samples
uint16 num_images
float32 noise_level
---
float32 rec_rate
float32 sigma_pose
float32[] prob_dist
float32 sigma_filtered
float32[] filt_dist
```

Appendix B: Input probability tables

The tables listed below contain the statistical data necessary for probabilistic filtering on the LANL dataset. They contain the prior probabiliiteis P(c|z), as well as the standard devition of the pose rotation. The tables are slightly modified from the actual data so that they do not contain any zeros. This prevents the posterior probabilities from going to zero in the event of misclassifications not observed during statisitcs collection.

cph.list

```
broom 16.8 .987 .006 .002 .001 .001 .001 .001 .001 .001 .001
fitting 63.2 .003 .977 .001 .001 .001 .001 .001 .001 .001 .001
lathe_knob 9.7 .001 .005 .983 .001 .001 .001 .001 .001 .001 .001
lg_bowl 94.8 .001 .001 .001 .958 .001 .001 .001 .013 .001 .001
lg_can 105.1 .001 .001 .001 .001 .991 .136 .001 .001 .001 .001
med_can 64.6 .002 .001 .008 .001 .001 .835 .001 .001 .001 .001
scale 18.6 .001 .004 .001 .001 .001 .021 .991 .001 .038 .001
sm_bowl 82.8 .001 .001 .001 .034 .001 .001 .001 .979 .009 .001
sm_can 99.6 .001 .003 .001 .001 .001 .002 .001 .001 .931 .001
tape 117.7 .001 .001 .001 .001 .001 .001 .001 .001 .001 .991
```

vfh.list

```
broom 15.5 .803 .006 .002 .001 .001 .001 .001 .001 .001 .001
fitting 74.9 .036 .879 .019 .001 .001 .002 .054 .004 .009 .002
lathe_knob 11.6 .001 .002 .905 .001 .001 .001 .001 .001 .001 .001
lg_bowl 78.3 .001 .001 .001 .807 .001 .001 .001 .235 .001 .001
lg_can 109.9 .001 .001 .003 .001 .991 .050 .001 .001 .001 .001
med_can 103.9 .001 .001 .008 .001 .001 .934 .001 .001 .001 .001
scale 28.0 .010 .050 .005 .001 .001 .001 .918 .001 .001 .001
sm_bowl 101.4 .005 .001 .001 .185 .001 .001 .001 .754 .001 .001
sm_can 103.0 .142 .058 .063 .001 .001 .008 .022 .001 .983 .001
tape 96.3 .001 .001 .001 .001 .001 .001 .001 .001 .001 .990
```

Appendix C: Training data collection

The following ROS and Arduino code was used for data collection. This code is available in the NRG repository, but not on any public ROS repository. Because CPH and VFH features are not invariant to viewpoint, data collection must be done *in situ*. Therefore this code is not general and serves only as an example of how one might go about collecting data. Table C-1 is a bill of materials for constructing an inexpensive data collection table.

Table C-1. Pan/tilt Bill of Materials

Item	Qty	Vendor	Vendor P/N
Arduino Uno microcontroller	1	Adafruit tech.	50
Tilt mechanism	1	servocity.com	SPT400
Pan mechanism	1	servocity.com	SPG400A-BM-360
Servo	2	servocity.com	HS-7955TG

pan_tilt.pde (Arduino code)

```
#include <Servo.h>
#include <ros.h>
#include <std_msgs/Int32.h>


Servo tilt;
Servo pan;
int tiltHome = 59, panHome = 234;
int tiltCommand, panCommand;
int pos = 0;    // variable to store the servo position


ros::NodeHandle n;

void pan_cb(const std_msgs::Int32 & panAngle){
  if(panAngle.data+panHome >= 0 && panAngle.data+panHome <=360)
  {
    panCommand = panAngle.data+panHome;
  }
  else if(panAngle.data+panHome < 0)
  {
    panCommand = panAngle.data+panHome + 360;
  }
```

```
  else if(panAngle.data+panHome > 360)
  {
    panCommand = panAngle.data+panHome-360;
  }
  pan.write(panCommand/2);
}

void tilt_cb(const std_msgs::Int32 & tiltAngle){
  if(tiltAngle.data+tiltHome > 0 && tiltAngle.data+tiltHome < 360)
  {
    tiltCommand = tiltAngle.data+tiltHome;
  }
  else if(tiltAngle.data+tiltHome < 0)
  {
    tiltCommand = tiltAngle.data+tiltHome + 360;
  }
  else if(tiltAngle.data+tiltHome > 360)
  {
    tiltCommand = tiltAngle.data+tiltHome-360;

  }


  tilt.write(tiltCommand);
}

ros::Subscriber<std_msgs::Int32> pan_sub("/pan_command", &pan_cb );
ros::Subscriber<std_msgs::Int32> tilt_sub("/tilt_command", &tilt_cb );


void setup()
{
  tilt.attach(8,1000,1500);
  pan.attach(10,1050,1950);
  n.initNode();
  n.subscribe(pan_sub);
  n.subscribe(tilt_sub);

}

void loop()
{
  n.spinOnce();
  delay(1);
}
```

feature_extraction.cpp

```
#include <iostream>
#include <fstream>
```

```
#include <ros/ros.h>
#include <sensor_msgs/PointCloud2.h>
// PCL specific includes
#include <pcl/ros/conversions.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl/features/vfh.h>
#include <pcl/features/normal_3d.h>

#include "data_collection/process_cloud.h"
#include "euclidean_segmentation.h"
#include "nrg_object_recognition/segmentation.h"
#include "cph.h"

  ros::ServiceClient seg_client;

bool cloud_cb(data_collection::process_cloud::Request &req,
              data_collection::process_cloud::Response &res)
{
  //Segment from cloud:
  std::vector<pcl::PointCloud<pcl::PointXYZ>::Ptr> clouds;
  nrg_object_recognition::segmentation seg_srv;

  seg_srv.request.scene = req.in_cloud;
  seg_srv.request.min_x = -.75, seg_srv.request.max_x = .4;
  seg_srv.request.min_y = -5, seg_srv.request.max_y = .5;
  seg_srv.request.min_z = 0.0, seg_srv.request.max_z = 1.15;
  seg_client.call(seg_srv);

  pcl::PointCloud<pcl::PointXYZ>::Ptr cluster (new
pcl::PointCloud<pcl::PointXYZ>);
pcl::fromROSMsg(seg_srv.response.clusters.at(0), *cluster);
  std::cout << "cluster has " << cluster->height*cluster->width << "
points.\n";

  //Write raw pcd file (objecName_angle.pcd)
  std::stringstream fileName_ss;
  fileName_ss << "data/" << req.objectName << "_" << req.angle << ".pcd";
  std::cout << "writing raw cloud to file...\n";
  std::cout << fileName_ss.str() << std::endl;
  pcl::io::savePCDFile(fileName_ss.str(), *cluster);
  std::cout << "done.\n";

  //Write vfh feature to file:
  pcl::VFHEstimation<pcl::PointXYZ, pcl::Normal, pcl::VFHSignature308> vfh;
  vfh.setInputCloud (cluster);
  //Estimate normals:
  pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> ne;
  ne.setInputCloud (cluster);
  pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new
pcl::search::KdTree<pcl::PointXYZ> ());
```

```
  ne.setSearchMethod (tree);
  pcl::PointCloud<pcl::Normal>::Ptr cloud_normals (new
pcl::PointCloud<pcl::Normal>);
  ne.setRadiusSearch (0.03);
  ne.compute (*cloud_normals);
  vfh.setInputNormals (cloud_normals);
  //Estimate vfh:
  vfh.setSearchMethod (tree);
  pcl::PointCloud<pcl::VFHSignature308>::Ptr vfhs (new
pcl::PointCloud<pcl::VFHSignature308> ());
  // Compute the feature
  vfh.compute (*vfhs);
  //Write to file: (objectName_angle_vfh.pcd)
  fileName_ss.str("");
  std::cout << "writing vfh descriptor to file...\n";
  fileName_ss << "data/" << req.objectName << "_" << req.angle << "_vfh.pcd";
  pcl::io::savePCDFile(fileName_ss.str(), *vfhs);
  std::cout << "done.\n";

  //Extract cph
  std::vector<float> feature;
  CPHEstimation cph(5,72);
  cph.setInputCloud(cluster);
  cph.compute(feature);
  //Write cph to file. (objectName_angle.csv)
  std::ofstream outFile;
  fileName_ss.str("");
  fileName_ss << "data/" << req.objectName << "_" << req.angle << ".csv";
  outFile.open(fileName_ss.str().c_str());
  std::cout << "writing cph descriptor to file...\n";
  for(unsigned int j=0; j<feature.size(); j++){
      outFile << feature.at(j) << " ";
  }
  outFile.close();
  fileName_ss.str("");
  std::cout << "done.\n";
  res.result = 1;
}

int main(int argc, char **argv)
{

  ros::init(argc, argv, "feature_extractor");

  ros::NodeHandle n;


  //Offer services that can be called from the terminal:
  ros::ServiceServer joint_test_serv = n.advertiseService("process_cloud",
cloud_cb );
```

```
  seg_client =
n.serviceClient<nrg_object_recognition::segmentation>("segmentation");

  ros::spin();

  return 0;
}
```

data_collection.cpp

```
//code contributed by Alexandria Gallagher
#include "ros/ros.h"
#include "std_msgs/UInt16.h"
#include "std_msgs/String.h"
#include "data_collection/dataCollect.h"
#include "data_collection/process_cloud.h"
#include <sensor_msgs/PointCloud2.h>_
#include <sstream>

ros::Publisher pan_pub;
ros::ServiceClient pan_client;
sensor_msgs::PointCloud2 cloud_to_process;

void kinect_cb(sensor_msgs::PointCloud2 fromKinect)
{
cloud_to_process = fromKinect;
}

bool rotate_cb(data_collection::dataCollect::Request &req,
data_collection::dataCollect::Response &res)
{
std_msgs::UInt16 command;
command.data=0;
data_collection::process_cloud srv;

srv.request.objectName = req.objectName;

while(command.data<360)
{
ros::Rate loop_rate(.2);

srv.request.in_cloud = cloud_to_process;
srv.request.angle = command.data;
pan_client.call(srv);
srv.response.result = 1;

pan_pub.publish(command);
ros::spinOnce();
loop_rate.sleep();
command.data += req.delta;
}
```

```
res.result = 1;
std::cout << "Status: " << srv.response.result << "\n";
return 1;
}

int main(int argc, char **argv)
{
ros::init(argc, argv, "pan360_data_collect");
ros::NodeHandle n;

pan_pub = n.advertise<std_msgs::UInt16>("/pan_command",1);
ros::ServiceServer pan_serv = n.advertiseService("/pan360_data_collect",
rotate_cb);
ros::Subscriber pan_sub = n.subscribe("/camera/depth_registered/points", 1,
kinect_cb);
pan_client = n.serviceClient<data_collection::process_cloud>("process_cloud");

ROS_INFO("Ready!");
ros::spin();

return 0;
}
```

process_cloud.srv

```
sensor_msgs/PointCloud2 in_cloud
float32 angle
string objectName
---
int32 result
```

dataCollect.srv

```
string objectName
int32 delta
---
int32 result
```

Appendix D: ROS recognition quickstart tutorial

The following tutorial describes how to replicate the recognition results achieved in this dissertation using the ROS system developed as part of this work. This tutorial also appears on the NRG wiki page.

Preliminaries

Before proceeding, make sure you are using a computer that has ROS installed, as well as the openni_camera and openni_launch packages. To do this, check out the ROS page on the wiki. Then, make sure that the ROS directory of your local NRG repository is listed in the ROS_PACKAGE_PATH variable of ~/.bashrc. Your ~/.bashrc file should have some lines that look kind of like this:

```
export ROS_PACKAGE_PATH=:$ROS_PACKAGE_PATH:~/nrg/ROS
export ROS_WORKSPACE=~/nrg/ROS
```

If you have recently cloned the NRG repository to your local machine, you will need to make sure that the ROS package management tools can find the data_collection and nrg_object_recognition packages, and that the packages have been built. From a terminal:

```
rospack profile
rosmake data_collection
rosmake nrg_object_recognition
```

Before continuing, you should also be fairly familiar with basic ROS concepts like nodes and services. If not, you should take some time to familiarize yourself with the tutorials.

nrg_object_recognition Quickstart

Object recognition happens in two phases:

- Training: The recognition system learns what interesting objects look like
- Test: The recognition system locates and identifies previously learned objects in the scene.

Training:

The data_collection package is used to train the classifier. You will need to use the servo-driven data collection table pictured below. Verify that the Kinect and the Arduino on the data collection table are plugged in. For each object you wish to recognize, you must collect multi-view training images. To do this start each of these from a terminal:

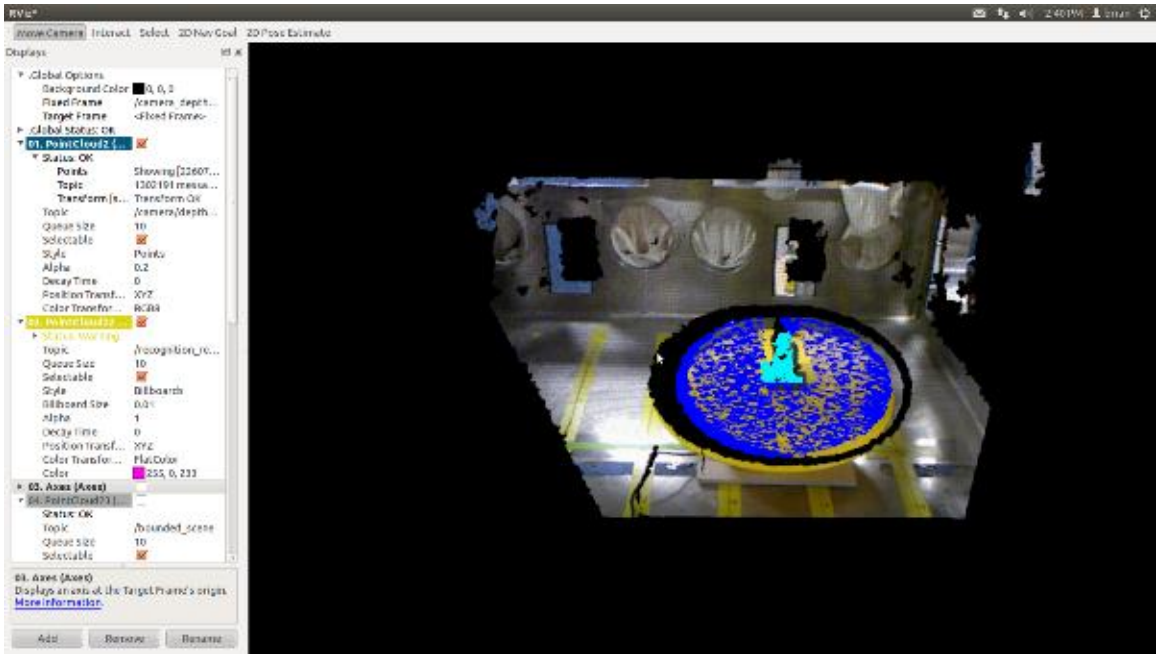
```
roslaunch openni_launch openni.launch
rosrun data_collection pan360_data_collect
rosrun data_collection feature_extraction
rosrun nrg_object_recognition segmentation_node
rosrun rosserial_python /dev/ttyACM0
rosrun rviz rviz
```

Note that all ROS nodes run as their own processes so you will need to do each of the above from a different terminal (ctrl+shift+T to get a new tab in a terminal window). Once all of the above nodes are running, you call a service from the command line. It takes two arguments: the object name and the angular resolution you want in the training data. Here is an example:

```
rosservice call pan360_data_collect test_object 1
```

This will take some time. The object will now be imaged at 1 degree increments. If you have rviz running, subscribe to the PointCloud2 published by the segmentation node. You should see something like Figure D-1

Figure D-1. Rviz display during data collection.



The object is segmented out from the scene and feature files are written to <current directory>/data.

Testing:

Currently, nrg_object_recognition/main_dataset_node makes good example code. It was written to measure performance of the classifier, but it provides a convenient way to experiment with the object recognition package and to get a feel for how it works.

First, get the ROS system running:

```
roslaunch openni_launch openni.launch
rosrun rosserial_python /dev/ttyACM0
rosrun nrg_object_recognition segmentation_node
rosrun nrg_object_recognition cph_recognition_node
rosrun main_dataset_node
rosrun rviz rviz
```

The main_dataset_node offers some services that can be requested from a terminal. For this quickstart, we will use the live_test service. Place an object on the data_collection table. Then, from a terminal:

```
rosservice call live_test <object_name> 0 1 8 0
```

The arguments in the service request are the object's name, the method (0 for CPH), the number of images to take at each angle (1), and the number of angles at which to take the images. What should happen now is that the system will attempt to recognize the object on the data collection table at 8 different angles. The main_dataset_node will print the recognition result at each angle to the screen. The point cloud of the training sample to which the test image was matched is published on a ROS topic and displayed in rviz. In rviz, you should see something like this:

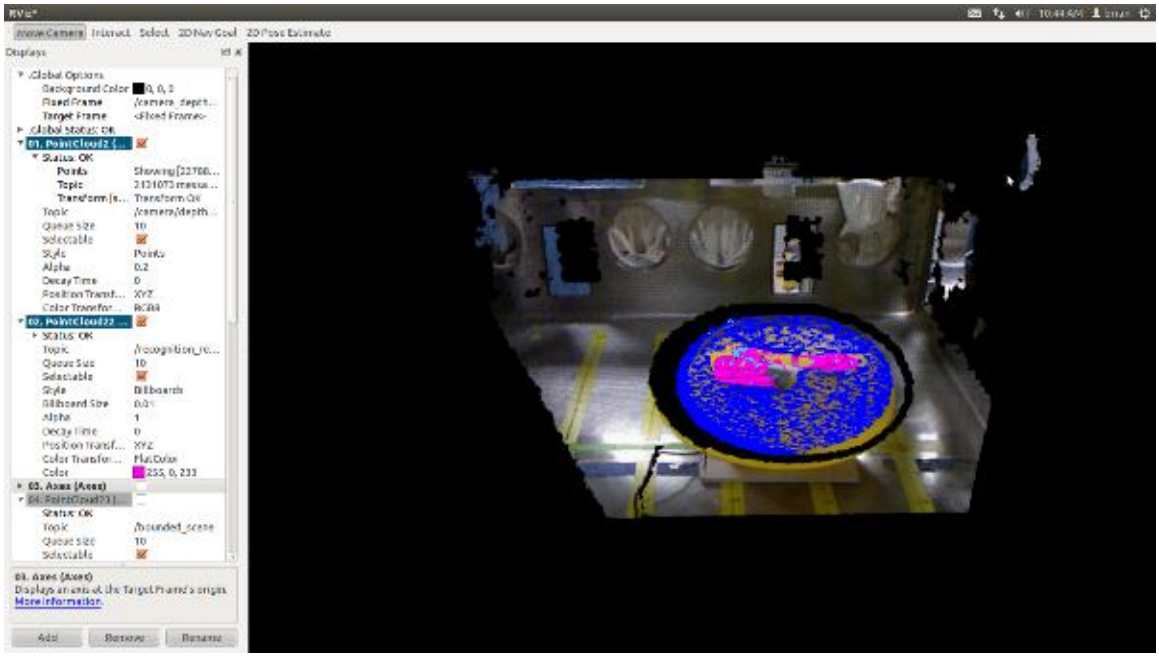


Figure D-2. Rviz display during testing

In this image, the recognition/pose estimation result is shown in magenta. Congratulations! you are now recognizing objects.

Appendix E: PCL code

This version of CPH can be built with the pcl trunk. The file cph.cpp should be placed in the features/src directory. The cph.h file should be placed in features/include, and the cph.hpp file should be placed in features/include/impl. This implementation separates the CPH feature from ROS, and more tightly integrates it with PCL for use in cluster recognition applications.

The following disclaimer must accompany *each* of the files below. For brevity, it is only printed here once.

disclaimer

```
/*  Point Cloud Library (PCL) - www.pointclouds.org
 *  Copyright (c) 2012- Brian O'Neil
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of Willow Garage, Inc. nor the names of its
 *     contributors may be used to endorse or promote products derived
 *     from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 */
```

cph.cpp

```
#include <pcl/point_types.h>
#include <pcl/impl/instantiate.hpp>
#include <pcl/features/cph.h>
#include <pcl/features/impl/cph.hpp>

// Instantiations of specific point types
#ifdef PCL_ONLY_CORE_POINT_TYPES
  PCL_INSTANTIATE_PRODUCT(CPHEstimation,
((pcl::PointXYZ)(pcl::PointXYZI)(pcl::PointXYZRGB)(pcl::PointXYZRGBA)))
#else
  PCL_INSTANTIATE_PRODUCT(CPHEstimation, (PCL_XYZ_POINT_TYPES))
#endif
```

cph.hpp

```
#ifndef PCL_FEATURES_IMPL_CPH_H_
#define PCL_FEATURES_IMPL_CPH_H_

#include <pcl/features/cph.h>
#include <pcl/common/common.h> //May not need this


//////////////////////////////////////////////////////////////////////////////
template<typename PointInT, typename PointOutT> void
pcl::CPHEstimation<PointInT, PointOutT>::compute (PointCloudOut
&output)
{

  // Copy the header
  output.header = input_->header;

  // Resize the output dataset
  // Important! We should only allocate precisely how many elements we
will need, otherwise
  // we risk at pre-allocating too much memory which could lead to
bad_alloc
  // (see http://dev.pointclouds.org/issues/657)
  output.width = output.height = 1;
  output.is_dense = input_->is_dense;
  output.points.resize (1);

  // Perform the actual feature computation
  computeFeature(output);

  Feature<PointInT, PointOutT>::deinitCompute ();

}
//////////////////////////////////////////////////////////////////////////////

template <typename PointInT, typename PointOutT> void
pcl::CPHEstimation<PointInT, PointOutT>::computeFeature (PointCloudOut
&output)
```

```
{
  output.points.resize (1);
  output.width = 1;
  output.height = 1;

  //compute bouding box size:
  float x_max=0, x_min=100, y_max=0, y_min = 100, z_max=0, z_min=100;
  for(unsigned int i=0; i<input_->size(); i++){
    if(input_->points.at(i).x > x_max)
      x_max = input_->points.at(i).x;
    if(input_->points.at(i).x < x_min)
      x_min = input_->points.at(i).x;
    if(input_->points.at(i).y > y_max)
      y_max = input_->points.at(i).y;
    if(input_->points.at(i).y < y_min)
      y_min = input_->points.at(i).y;
    if(input_->points.at(i).z > z_max)
      z_max = input_->points.at(i).z;
    if(input_->points.at(i).z < z_min)
      z_min = input_->points.at(i).z;
  }

  x_size_ = (x_max - x_min);
  y_size_ = (y_max - y_min);
  z_size_ = (z_max - z_min);

  float max_size = x_size_;
  if(y_size_ > max_size)
    max_size = y_size_;
  if(z_size_ > max_size)
    max_size = z_size_;
  max_size*=100;

  centroid_(0) = x_min + x_size_/2;
  centroid_(1) = y_min + y_size_/2;
  centroid_(2) = z_min + z_size_/2;

  //compute feature
  hist_.clear();
  hist_.resize(nr_bins_vert_*nr_bins_azmt_, 0.0f);
  int y,c;
  float dy = float(y_size_)/float(nr_bins_vert_);
  float dc = float(2.0*PI)/float(nr_bins_azmt_);

  for(unsigned int i=0; i<input_->size(); i++){
    //bin z component
    y = int(floor((input_->points.at(i).y-y_min)/dy));
    c = int(floor((PI+atan2(input_->points.at(i).z-centroid_(2),input_-
>points.at(i).x-centroid_(0)))/dc));
    if(y*nr_bins_azmt_+c < hist_.size())
      hist_.at(y*nr_bins_azmt_+c)+=1.0f;
  }
```

```
  //Find tallest peak
  float max_peak = 0;
  for(unsigned int i=0; i<hist_.size(); i++){
    if(hist_.at(i) > max_peak)
     max_peak = (float)hist_.at(i);
  }

  //Rescale cph to largest spatial extent:
  float scaleFactor = max_size/max_peak;
  for(unsigned int i=0; i<hist_.size(); i++){
    hist_.at(i)*=scaleFactor;
  }

  //Stick size onto the end and BAM! scale variance.
  hist_.push_back(x_size_*100);
  hist_.push_back(y_size_*100);
  hist_.push_back(z_size_*100);

  //Populate output:
  for(unsigned int i=0; i<hist_.size(); i++){
    output.points[0].hist[i] = hist_.at(i);
  }
}
#endif    // PCL_FEATURES_IMPL_CPH_H_
```

cph.h

```
#ifndef PCL_FEATURES_CPH_H_
#define PCL_FEATURES_CPH_H_

#ifndef PI
#define PI 3.14159265
#endif

#include <pcl/point_types.h>
#include <pcl/features/feature.h>

struct CPHSignature
  {
    float hist[363];
  };

  POINT_CLOUD_REGISTER_POINT_STRUCT(CPHSignature,
                            (float[363], hist, hist))
namespace pcl
{

  /** \brief CPHEstimation estimates the <b>Cylindrical Projection
Histogram (VFH)</b> descriptor for a given point cloud

   * \author Brian O'Neil
    * \ingroup features
    */
```

```
  template<typename PointInT, typename PointOutT = CPHSignature>
  class CPHEstimation : public Feature<PointInT, PointOutT>
  {
    public:
      using Feature<PointInT, PointOutT>::feature_name_;
      using Feature<PointInT, PointOutT>::getClassName;
      using Feature<PointInT, PointOutT>::indices_;
      using Feature<PointInT, PointOutT>::input_;

      typedef typename Feature<PointInT, PointOutT>::PointCloudOut
PointCloudOut;
      typedef typename boost::shared_ptr<CPHEstimation<PointInT,
PointOutT> > Ptr;
      typedef typename boost::shared_ptr<const CPHEstimation<PointInT,
PointOutT> > ConstPtr;


      /** \brief Empty constructor. */
      CPHEstimation () : nr_bins_vert_ (5), nr_bins_azmt_ (72)
      {
        feature_name_ = "CPHEstimation";
      }

      /** \brief Overloaded computed method from pcl::Feature.
        * \param[out] output the resultant point cloud model dataset
containing the estimated features
        */
      void
      compute (PointCloudOut &output);

    private:
      void computeFeature (PointCloudOut &output);
      int nr_bins_vert_, nr_bins_azmt_;
      std::vector<float> hist_;
      //pcl::PointCloud<pcl::PointXYZ>::Ptr cloud, result;
      float x_size_, y_size_, z_size_;
      Eigen::Vector3f centroid_;

    protected:

  };
}

#ifdef PCL_NO_PRECOMPILE
#include <pcl/features/impl/cph.hpp>
#endif

#define PCL_INSTANTIATE_CPHEstimation(T) template class PCL_EXPORTS
pcl::CPHEstimation<T>;

#endif  //#ifndef PCL_FEATURES_CPH_H_
```

References

AdaFruit Industreis. (n.d.). The Open Kinect Project. Retrieved November 1, 2012, from http://www.adafruit.com/blog/2010/11/04/the-open-kinect-project-the-ok-prize-get-1000-bounty-for-kinect-for-xbox-360-open-source-drivers/

Ankerst, M., Kastenmuller, G., Kriegel, H. P., & Seidl, T. (1999). 3D shape histograms for similarity search and classification in spatial databases. *Symposium on Advances in Spatial Databases*, 207-226.

Automate 2013, Chicago, IL. Jan 21-24 2013. http://www.automate2013.com

Austin, D. J., & Jensfeldt, P. (2000). Using multiple Gaussian hypotheses to represent probability distributions for mobile robots. Proceedings of the IEEE Inernational Conference on Robotics and Automation (ICRA).

Bradski, G. (2000). *The OpenCV Library.* Dr. Dobb's Journal of Software Tools.

Bruemmer, D. J., Marble, J. L., McKay, M. D., & Dudenhoeffer, D. D. (2002*). Dynamic-Autonomy for Remote Robotic Sensor Deployment*. Spectrum 2002.

Brooks, R. A. (1986). *A robust layered control system for a mobile robot*. IEEE Journal of Robotics and Automation, 2, 14-23.

Canny, J. (1987). *The complexity of robot motion planning*. Cambridge, MA: MIT Press.

Carreira, J., & Sminchisescu, C. (2008a). Constrained parametric min-cuts for automatic object segmentation. *IEEE Intl. Conference on Computer Vision and Pattern Recognition (CVPR)*.

Carreira, J., & Sminchisescu, C. (2010b). Object recognition as ranking holistic figure-ground hypotheses. *IEEE Intl. Conference on Computer Vision and Pattern Recognition (CVPR)*.

Caruna, R., & Niculescu-Mizil, A. (2006, June). An empirical comparison of supervised learning algorithms. *International Conference on Machine Learning (ICML)*, 161-168.

Chang, C., & Lin, C. (2011). LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology, 2*(3), 27:1-27:27.

Coonley, M. S. (Ed.). (1st/2nd Quarters 2008). ARIES turns 10. *Actinide Research Quarterly*.

Dalal, N., & Triggs, B. (2005, June). Histograms of oriented gradients for human detection. *IEEE International Conference on Computer Vision and Pattern Recognition (CVPR)*, 886-893.

Dellaert, F., Burgard, W., & Thrun, S. (1999). Monte Carlo localization for mobile robots. *Proceedings of the International Conference on Robotics and Automation (ICRA)*.

Deng, J., Berg, A., Li, K., & Fei-Fei, L. (2010). What does classifying more than 10,000 image categories tell us? *European Conference on Computer Vision (ECCV)*.

Dryden, I. L., & Mardia, K. V. (1998). *Statistical Shape Analysis*. John Wiley & Sons.

Duda, R., Hart, P., & Stork, D. (2001). *Pattern Classification* (2nd ed.). New York: Wiley-Interscience.

Felzenszwalb, P. F., & Huttenlocher, D. P. (2004, September). Efficient graph-based image segmentation. *International Journal of Computer Vision, 59*(2), 167-181.

Forssen, P. E., & Lowe, D. G. (2007). Shape descriptors for maximally stable extremal regions. *IEEE Intl. International Conference on Computer Vision (ICCV)*.

Glover, J., Rus, D., & Roy, N. (2008). Probabilistic models of object geometry for grasp planning. *Procedings of Robotics: Science and Systems (RSS 2008)*.

Gould, S., Rodgers, J., Elidan, G., & Koller, D. (2008). Multi-class segmentation with relative location prior . *International Journal of Computer Vision (IJCV)*.

Harris, C. G., & Stephens, M. J. (1988). A combined corner and edge detector. *Proceedings of the Fourth Alvey Vision Conference*, 147-151.

Hinterstossier, S., Cagniart, C., Ilic, S., Sturm, P., Navab, N., & Lepetit, V. (2012, May). Gradient response maps for real-time detection of texture-less objects. *IEEE Trans. on Pattern Analysis and Machine Intelligence, 34*(5), 876-888.

Hoey, J., & Poupart, P. (2005). Solving POMDPs with continuous or large discrete observation spaces. *Proceedings of the Intl. Joint Conference on Artificial Intelligence*.

Hoiem, D., Efros, A., & Hebert, M. (2005). Geometric context from a single image. *Proceedings of the IEEE Intl. Conference on Computer Vision (ICCV)*.

Johnson, A., & Hebert, M. (1999, May). Using spin images for efficient object recognition in cluttered 3D scenes. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI), 21*(5), 433-449.

Kaebling, L. P., Cassandra, A. R., & Kurien, J. A. (1996). Acting under uncertainty: Discrete Bayesian models for mobile robot navigation. *Proceedings of the IEEE/RSJ Conference on Robotics and Systems (IROS)*.

Kalman, R. E. (1960). A new approach to linear filtering and prediction problems. *ASME Journal of basic engineering, 82*, 35-45.

Kavraki, L., Latombe, J. C., & Overmars, M. (n.d.). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation, 12*, 566-580.

Kazhdan, M., Funkhouser, T., & Rusinkiewics, S. (2003, June). Rotation invariant spherical harmonic representation of 3D shape descriptors. *Symposium on Geometry and Processing*.

Kim, K., Chalidabhongse, T. H., & Harwood, D. (2004). Background modeling and subtraction by codebook construction. *IEEE Intl. Conference on Image Processing (ICIP)*.

Klasing, K., D. Althoff, D., Wollherr, D., & Buss, M. (2009, May). Comparison of surface normal estimation methods for range sensing applications. *Intl. Conf. on Robotics and Automation*, 3206-3211.

Knoll, J. A. (2007). *Complete workcell modeling for robot control and coordination* (Unpublished master's thesis). University of Texas, Austin, TX.

Lai, K., Bo, L., Ren, X., & Fox, D. (2010). A large-scale hierarchical multi-view RGB-D dataset. *IEEE International Conference on Robotics and Automation (ICRA)*.

Lai, K., Bo, L., Ren, X., & Fox, D. (2011, August). A scalable, tree-based approach for joint object and pose recognition. *AAAI Conference on Artificial Intelligence (AAAI)*.

Liu, T., Sun, J., Zheng, N., Tang, X., & H, Shum. (2007). Leaning to detect a salient object. *IEEE Intl. Conference on Computer Vision and Pattern Recognition (CVPR)*.

Lowe, D. G. (1999). Object recognition from local scale-invariant features. *IEEE Intl Conference on Computer Vision (ICCV)*.

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision (IJCV), 60*(2), 91-110.

Matas, J., Chum, O., Urban, M., & Pajdla, T. (2002, September). Robust wide baseline stereo from maximally stable extremal regions. *13th British Machine Vision Conference (BMVC)*, 384-393.

Maybeck, P. S. (1990). The Kalman Filter: An introduction to concepts. In I. J. Cox & G. T. Wilfong (Eds.), *Autonomous robot vehicles*. Springer Verlag.

Mesa Imaging. (2012). *Swiss Ranger SR 4000*. Retrieved from http://www.mesa-imaging.ch/profview4k.php

Mikolajczyk, K., Tuytelaars, T., Schmid, C., Zisserman, A., Matas, J., Schaffalitzky, F., . . . Van Gool, L. (2005). A comparison of affine region detectors. *International Conference on Computer Vision (IJCV), 65*(1), 43-72.

Muja, M., & Lowe, D. G. (2009). Fast approximate nearest neighbors with automatic algorithm configuriation. *Intl. Conf. on Computer Vision Theory and Applications*.

Navalpakkam, V., & Itti, L. (2006). An integrated model of top-down and bottom-up attention form optimizing detection speed. *IEEE Intl. Conference on Computer Vision and Pattern Recognition (CVPR)*, 2049-2056.

O'Neil, B., Pryor, M., & Landsberger, S. (2011). A graph-based modeling approach for automating neutron radiography experimentation. In *ANS EPRRSD - 13th Robotics & remote systems for hazardous environments*. Knoxville, TN.

Point Grey Research. (n.d.). *Bumblebee 2*. Retrieved from http://www.ptgrey.com/products/bumblebee2/bumblebee2_stereo_camera.asp

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., . . . Wheeler, R. (2009). ROS: an open-source Robot Operating System. *International Conference on Robotics and Automation*.

Reif, J. H. (1979). Complexity of the Mover's Problem and Generalizations. *Proceedings of the 20th IEEE Symposium on Foundations of Computer Science*, 421-427.

Rusu, R. (2009). *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments* (Doctoral dissertation, Technical University Munich, Munich, Germany). Retrieved from http://files.rbrusu.com/publications/RusuPhDThesis.pdf

Rusu, R. B., Beetz, M., & Bloodow, N. (2009, May). Fast Point Feature Histograms for 3D registration. *International Conference on Robotics and Automation*.

Rusu, R. B., Bradski, G., Thibaux, R., & Hsu, J. (2010, October). Fast 3D recognition and pose using the Viewpoint Feature Histogram. *23rd IEEE/RSJ International Conference on Robotics and Systems*.

Rusu, R. B., & Cousins, S. (2011, May 9). 3D is here: Point Cloud Library (PCL). *International Conference Robotics and Automation*.

Rusu, R. B., Marton, Z. C., Blodow, N., & Beetz, M. (2008). Learning informative point classesfor the acquisition of object model maps. *10th Intl. Conference on Control Automation, Robotics, and Vision*.

Shi, J., & Malik, J. (2000, August). Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 22*(8), 888-905.

SICK Sensor Intelligence. (2010, April 15). *LMS-200 Data Sheet* [Brochure].

Sivic, J., & Zisserman, A. (2003). Video google: A text retrievla approach to object matching in videos. *IEEE Intl. Conference on Computer Vision (ICCV)*.

Smith, R. C., & Cheeseman, P. (1986). On the representation of spatial uncertainty. *International Journal of Robotics Research, 52*, 56-68.

Stauffer, C., & Grimson, W. E. (1999). Adaptive background mixture models for real-time tracking. *IEEE Conference on computer vision and pattern recognition (CVPR)*.

Thrun, S., Burgard, W., & Fox, D. (2006). *Probabilistic Robotics*. Cambridge, MA: MIT Press.

Velodyne Lidar, Inc. (n.d.). *Velodyne HDL-64E Laser Rangefinder*. Retrieved 2012, from http://velodynelidar.com/lidar/hdlproducts/hdl64e.aspx

Wu, T. F., & Lin, C. J. (2004). Probability estimates for multi-class classification by pairwise coupling. *Journal of Machine Learning Research, 5*, 975-1005.

Wu, Z., & Leahy, R. (1993, November). An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 15*(11), 1101-1113.

Vita

Brian Erick O'Neil was born in Tempe, AZ in 1978 to his parents Pat and Barb O'Neil. Brian graduated from Arizona State University with a B.S. in Political Science in 2001. He then worked as a professional flight instructor and airline pilot before returning to Arizona State where he earned a second B.S. degree in Mechanical Engineering. Brian entered the graduate school at The University of Texas in 2008 studying nuclear engineering and robotics. He earned his master's degree at UT in Mechanical Engineering in 2010. Brian currently resides in Austin with his wife, Emily, son, Henry, and daughter, Ellen.

Permanent email: brian.erick.oneil@gmail.com

This dissertation was typed by the author