Northeastern University
College of Computer and Information Science

Thesis Signature Page

Thesis Title: Robotic Grasping in Cluttered Scenes
Author: Matthew Cissaro
Department: Computer Science

Approved for Thesis Requirements of the Master of Science Degree

Thesis Advisor
Dr. Robert Platt

Thesis Committee Member or Reader
Dr. Tzahn Padir

Thesis Committee Member or Reader
Dr. Hanman Singh

GRADUATE SCHOOL APPROVAL:

COPY RECEIVED IN
GRADUATE SCHOOL OFFICE:
Robotic Grasping in Cluttered Scenes

A Thesis Presented

by

Matthew Corsaro

to

The Department of Computer Science

in partial fulfillment of the requirements

for the degree of

Master of Science

in

Computer Science

Northeastern University
Boston, Massachusetts

April 2017
Contents

List of Figures iii
List of Tables iv
List of Acronyms v
Abstract of the Thesis vi

1 Introduction 1

2 Background 2
  2.1 Motion Planning 2
    2.1.1 RRT 3
    2.1.2 TrajOpt 4
    2.1.3 OpenRAVE 4
  2.2 3D Sensing 5
    2.2.1 3D Sensors 5
    2.2.2 Point Cloud Processing 5
  2.3 ROS 6
  2.4 Machine Learning 7
    2.4.1 Regression 7
    2.4.2 Perceptron 8
    2.4.3 Neural Networks 9
    2.4.4 Deep Learning 10
  2.5 Related Work 11

3 Methods 13
  3.1 Baxter System 13
    3.1.1 Viewpoint Strategies 13
    3.1.2 Grasp Candidate Generation 15
    3.1.3 Grasp Classification 15
    3.1.4 Grasp Representation 16
    3.1.5 Training Data 16
    3.1.6 Grasp Selection 18
# List of Figures

3.1 Grasp Candidate Generation ........................................ 15
3.2 Grasp Representation ............................................... 17
3.3 Frictionless Antipodal Grasp ...................................... 18
3.4 Warthog with UR5 ..................................................... 24

4.1 Baxter Objects ....................................................... 26
4.2 YCB Objects .......................................................... 26
List of Tables

3.1 Baxter Grasp Success Rate from [4] . . . . . . . . . . . . . . . . . . . . . . . . . 20
4.1 YCB and Baxter Object Set Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
List of Acronyms

**CNN** Convolutional Neural Network. A neural network architecture containing layers that mimic the way a brain processes images.

**DoF** Degrees of Freedom. The total number of revolute and prismatic joints a robotic arm has.

**PCL** Point Cloud Library. A C++ library that provides a number of point-cloud-processing algorithms.

**ROS** Robot Operating System. Ubuntu-based toolkit that contains Python and C++ libraries that can be used to interface with various robots.

**RRT** Rapidly Exploring Random Tree. A search algorithm that randomly explores the state space.
Abstract of the Thesis

Robotic Grasping in Cluttered Scenes

by

Matthew Corsaro

Master of Science in Computer Science
Northeastern University, April 2017
Dr. Robert Platt, Advisor

Robotic grasping systems that can clear clutter from a surface have many possible applications. One of these grasping systems could be implemented on a mobile robot that performs household chores. A grasping system could just as easily be implemented on a vehicular mobile base in order to perform grasping tasks in an outdoor environment. The Helping Hands Lab at Northeastern had implemented a grasping system that detected grasps using a convolutional neural network. Because this system was implemented on a Baxter Research Robot, kinematic inaccuracies contributed significantly to the grasp failures that the system encountered. In order to reduce this error, the grasping system was ported to the UR5 robotic arm. Experiments have shown that the kinematic error prevalent in the Baxter arm has not occurred once with the UR5. The arm was later mounted on a mobile Warthog base in order to show how portable the system is.
Chapter 1

Introduction

Detecting objects and determining how to grasp them within a dynamic, cluttered environment is a difficult task for robots. The nature of the sensors available and the complex and varied shapes of objects are two of the main factors that contribute to the challenge. A grasp detection system used in a household setting must capable of detecting and grasping a variety of objects.

A grasp detection system has been implemented on the Baxter Research Robot by the CCIS Helping Hands Lab group. This system consists of a robotic arm with a wrist-mounted 3D sensor that picks objects from a cluttered pile. One of the major issues with the system is that the forward kinematic errors inherent in the Baxter arm cause a significant number of grasp failures during experiments.

While such a system has applications across numerous domains, one goal is to eventually mount a robotic arm to the base of a mobile robot, which would then detect and pick up litter in an outdoor environment. The two goals of this work were to reduce the occurrence of grasp failures due to kinematic errors, and to prepare the system to be mounted on the base of a mobile robot. Both of these tasks could be completed by porting system implemented on the Baxter robot to the more accurate and portable UR5 arm.
Chapter 2

Background

2.1 Motion Planning

Most robotic arms are comprised of kinematic chains, with the links in the chain connected by revolute or prismatic joints. The manipulators examined in this project consist entirely of revolute joints, which rotate about a single axis. In order for a robot to grasp an object, the position and orientation of its gripper, or end effector, must be known relative to some frame of reference fixed to the base of the robot. This pose could be represented as a $4 \times 4$ matrix, called a homogeneous transformation matrix. The upper-left $3 \times 3$ sub-matrix represents the end effector’s rotation relative to a base frame, while the upper-right $3 \times 1$ sub-matrix represents the relative displacement. This pose is a function of the robot’s joint configuration.

Consider a 1-DoF manipulator. The position and orientation of an end effector affixed to the end of the link are given by the homogeneous transformation matrix. In this case, the rotation matrix is dependent on the angle, and can be given by the Rodrigues rotation formula:

$$
\hat{R} = \hat{I} + (\sin \theta) \hat{K} + (1 - \cos \theta) \hat{K}^2,
$$

where $\hat{k}$ is the unit vector about which the joint rotates, and

$$
\hat{K} = \begin{bmatrix} 0 & -k_3 & k_2 \\ k_3 & 0 & k_1 \\ -k_2 & k_1 & 0 \end{bmatrix}
$$

The position for this point is then given by the matrix product of this rotation matrix and the vector from the origin of the base frame to the end effector in zero configuration, which describes the position of the arm when all joint angles are set equal to zero. The bottom row of this matrix contains three 0’s followed by a 1. The process of determining an $n$-DoF robot’s end effector pose is similar;
now, these operations are cascaded in series. The homogeneous transformation matrix that relates
the base frame to the end effector frame, frame $n$, can be defined with the following equation:
$$ T_{0,n} = T_{0,1}T_{1,2}...T_{n-1,n}. $$

Determining the end effector’s pose based on the robot’s current joint angles is called
forward kinematics. While forward kinematics are important for determining where the end effector
is, the opposite problem needs to be solved when executing a grasp. Grasps are represented as a
center point in $\mathbb{R}^3$ along with an approach axis. Given this point and orientation in the base reference
frame, it would be useful if a joint configuration could be determined such that the end effector was
centered at the given point and angled with the given orientation. This is the inverse kinematics
problem. Unlike the straightforward solution to forward kinematics, there is no simple closed-form
solution to the inverse kinematics problem. Numerical algorithms can be implemented to solve the
problem in real-time, but these solvers are often slow. One possible solution to this problem is to
solve the inverse kinematics problem only once for all possible positions and orientations in an arm’s
workspace. This is what the IKFast library does. After generating the large database entry using
a Collada file containing a model of the desired arm, the solution to any given inverse kinematic
problem becomes a simple database lookup.

Now that a desired end effector pose can be transformed into a set of joint configurations
that the robot can move to, an algorithm needs to be implemented to determine how the arm can do
this safely without colliding with any obstacles in its environment. There are several motion planning
algorithms that can be implemented to solve this problem.

### 2.1.1 RRT

The rapidly exploring random tree algorithm searches a large state space graph in order to
find a path from a beginning state to a goal state. The algorithm works by first growing a tree within
the state space. This is done iteratively by picking a random point at each iteration, and then finding
the closest point in the tree to this new point. If the distance between these two points is larger than
some threshold, the new point is selected to be the threshold distance away from this close point, in
the direction of the random point. This process is repeated until the goal state is added to the tree. At
this point, the algorithm traces backwards from the goal to the start configuration, and returns the
path taken. This can be implemented on a robot to find a path from an initial configuration to a goal
configuration. The possible states in this system are a discretized set of joint configurations where
the robot is not in collision with any obstacles.
CHAPTER 2. BACKGROUND

A variant of this algorithm is RRT*. As RRT* explores a state space and adds new points to a tree, it checks the surrounding points already connected within the tree. For any point within a surrounding neighborhood, if the cost of the path to the root node could be reduced by instead running through this newly added point, the tree is rewired to do so. This generates much cleaner solutions, though it isn’t the most computationally efficient motion-planning algorithm.

2.1.2 TrajOpt

The TrajOpt package is one trajectory generation package commonly implemented in robotic systems. Described in [5], TrajOpt uses sequential convex optimization to optimize a naive initial straight-line trajectory through configuration space. Non-convex optimization problems attempt to minimize some function of $x$ while satisfying several equality and inequality constraints, where either the objective function or some of the constraints are non-convex. In the domain of trajectory planning, the goal orientation and position are often modeled as equality constraints, while distance to obstacles, and joint position and velocity limits are modeled as inequality constraints. The function to minimize in this non-convex problem is the sum of squared displacements, which measures the displacements between consecutive trajectory points. By minimizing this function, the trajectory strays from the naive straight-line path as little as possible. Unfortunately, non-convex optimization problems are difficult to solve. One common method, however, is to model the problem as a series of convex sub-problems. These convex sub-problems can then be solved using a method that solves quadratic programs, such as extensions to the gradient descent algorithms used to solve linear programs. TrajOpt is superior to RRT planners in that it does not include elements of stochasticity that are responsible for creating complex motions in RRTs.

2.1.3 OpenRAVE

Motion planning algorithms generate plans through a robot’s configuration space. These trajectories must be collision free; the arm cannot come into contact with any part of the environment while it moves to a grasp. In order to check for collisions, a simulated model of the arm and environment can be used. The user generates an environment containing immovable objects for their particular scenario, and includes a model of the arm and any attachments. The OpenRAVE simulation environment can then be used to ensure that the robot is not in collision with an obstacle for a given joint configuration. Many useful algorithms such as RRT and other motion planners are included within OpenRAVE’s Python and C++ libraries. In addition, OpenRAVE contains a variety
CHAPTER 2. BACKGROUND

of visualization tools and a large database of robot models. Packages such as TrajOpt depend on OpenRAVE to perform collision checking.

2.2 3D Sensing

When localizing an object within a scene, typical two-dimensional images are difficult for robots to use because they do not include any information about the depth of the objects within the scene. In order to store depth information, scenes can be captured in three-dimensions and stored in a point cloud. A point cloud is a set of points in \( \mathbb{R}^3 \); this differs from a two-dimensional image, which is a bounded matrix of pixels. These points are distributed throughout the cloud, whose dimensions are not fixed. With this 3D information, various geometric operations can be performed on point clouds to analyze the data they contain.

2.2.1 3D Sensors

Point clouds cannot be captured using standard techniques from a 2D camera that captures one image at a time. Instead, a 3D sensor is typically used to capture a point cloud. One of the most commonly used 3D sensors is called a structured light sensor. These sensors illuminate a scene with a unique pattern of infrared light, invisible to the human eye. These light patterns are then detected by the sensor. The distortions in the patterns are analyzed in order to determine the depth of each point in the scene. The measurements from the distortions in the light pattern are sometimes complemented by other vision techniques. Depth from focus, for instance, relies on the fact that objects that appear blurry within a 2D image are farther away. Many sensors use this technology today, including Intel’s RealSense, Structure Sensors, and Microsoft’s Kinect.

A second type of commonly used depth camera is the time-of-flight sensor. These sensors work by sending out pulses of infrared light and measuring the time it takes for the light to return to a receiver. This time measurement is used with the speed of light to determine how far the light traveled before coming into contact with an object or obstacle. The PicoFlexx line of sensors from PMD are popular time-of-flight sensors.

2.2.2 Point Cloud Processing

Surface normal estimation is a common technique used in the analysis of point clouds. Each point can be assigned a surface normal that estimates the direction of a vector normal to the
CHAPTER 2. BACKGROUND

The surface of the actual object at that point. This is done by first selecting a group of neighboring points. The sample covariance matrix is calculated for the neighborhood in question, \( N \), using the following equation:

\[
\sum_{p \in N} (p - \bar{p})(p - \bar{p})^T,
\]

where \( \bar{p} = \frac{1}{|N|} \sum_{p \in N} p \). The eigenvectors of this matrix are then calculated. Finally, the eigenvector of the smallest length is used to represent the surface normal at the specified point. Surface normals are useful for better understanding the structure of an object captured in a point cloud.

In some cases, it is useful to fit a plane to a point cloud, and then remove any points that make up this plane. Plane filtering can remove a significant number of points that do not add any useful data to the point cloud besides the table location. A plane can be fit to a point cloud using RANdom SAmple Consensus, or RANSAC. This method can, in fact, fit a number of shapes to a point cloud. In this algorithm, for a certain number of iterations, three points from a point cloud are selected. These points are used to generate a candidate plane. This plane is then scored by summing the distances between each point in the cloud and the candidate plane. After all iterations complete, or if a certain threshold has been reached, all points associated with the highest-scoring plane can be filtered out.

2.3 ROS

ROS is the standard robot interface used throughout academia and industry. This open-source Ubuntu package contains numerous libraries and algorithms that can be called in both C++ and Python. These libraries allow developers to implement and combine different algorithms and techniques on a variety of robotic platforms. Some of the third-party libraries included with ROS are PCL, the OpenCV vision library, and the MoveIt motion planning library. ROS also contains a number of visualization and simulation tools, such as RViz and Gazebo. A ROS core instance is launched from a host machine, which may connect through a network to other machines. Different interconnected ROS nodes can be launched from the various machines to perform tasks. The tasks performed by these nodes can involve reading data from sensors, analyzing received data, and outputting commands to various motors and actuators on a robot. As a result, ROS packages are written to interface with numerous hardware devices. For instance, there are ROS drivers written for 3D sensors, the Baxter Research Robot, as well as a unique driver package for the UR5 and other UR robotic arms.
CHAPTER 2. BACKGROUND

2.4 Machine Learning

Once a series of valid candidate grasps is produced, the system must execute the one most likely to succeed. This grasp can be selected using a machine learning algorithm. Machine learning algorithms learn from a set of labeled examples, and use this knowledge to predict a novel example’s label. These examples are specified as a vector or matrix of features. Deciding how to represent grasps as a feature vector is a problem in and of itself. Selecting an optimal grasp is a supervised learning task, as the algorithm learns from a labeled training set. One of the most basic types of supervised machine learning problems is the classification problem. Examples can be assigned one of several labels from a known set of labels. Binary classifiers assign one of two labels, while multi-label classifiers assign one label from a set of more than two.

There are many strategies that can be applied to solve classification problems. The $K$-Nearest Neighbors algorithm finds the $K$ training points that are closest to a new example, and the example is given the label that the majority of these points have. In practice, it may not be practical to store and constantly search through large data sets. Additionally, the user must determine their optimal value for $K$, as well as the distance function to use.

2.4.1 Regression

Alternately, regression could be used to classify the data. Regression is typically used when the response is expected to be a continuous function of the input data. To solve a linear regression problem where the output is a linear function of the input, for instance, a weight vector must be chosen such that $y = f(\hat{x}) = \hat{w}^T \hat{x}$ best fits the training set. Note that the first element in this weight vector corresponds to the offset term, and the first element in each feature vector is therefore 1. A cost function for a particular weight vector in linear regression can be defined as:

$$C(\hat{w}) = \frac{1}{2} \sum_{i=1}^{n} (\hat{w}^T \hat{X}_i - y_i)^2$$

where $\hat{X}$ is a matrix of feature vectors for $n$ examples, with corresponding labels stored in $\hat{y}$. This least-squares cost function measures the amount that each predicted label in the training set differs from the actual label. In order to generate a model that estimates these training labels most accurately, this cost function should be minimized. The optimal weights can then be found using an optimization algorithm, such as gradient descent. In gradient descent, the gradient of the cost function is used to iteratively move towards an optimal value.
CHAPTER 2. BACKGROUND

Though linear regression itself cannot be used to solve our grasp classification problem, the concept could be transformed to fit the situation. Logistic regression is the form of regression commonly implemented to solve binary classification problems. The predictions made by standard forms of regression can range over all real numbers, which are difficult to translate into distinct classes. If the predictions instead ranged between 0 and 1, each output value $f(\hat{x})$ could correspond to the probability that $\hat{x}$ is a member of the positive class, while $1 - f(\hat{x})$ would correspond to the probability that $\hat{x}$ is a member of the negative class. $y$ could then be defined as a piecewise function that returns the most probable label:

$$y = \begin{cases} 
1 & f(\hat{x}) \geq 0.5 \\
0 & f(\hat{x}) < 0.5
\end{cases}$$

In order produce labels in this range, the form of the classification function must be changed. Instead of choosing $f(\hat{x})$ to be linear in $\hat{x}$, a sigmoid function, $s(z) = \frac{1}{1+e^{-z}}$, can be used. The sigmoid function has two useful properties: $\lim_{z \to \infty} s(z) = 1$ and $\lim_{z \to -\infty} s(z) = 0$. The hypothesis function can be written as a sigmoid function: $f(\hat{x}) = s(\hat{w}^T \hat{x})$. To train this classifier, the weight vector $\hat{w}$ must be selected. As was done in the linear regression case, $\hat{w}$ should be chosen to minimize a cost function. In the case of logistic regression, this cost function can be derived from the likelihood of the parameters, $P(y|\hat{x}; \hat{w}) = (s(\hat{w}^T \hat{x}))^y(1 - s(\hat{w}^T \hat{x}))^{(1-y)}$. Note that $P(y = 1|\hat{x}; \hat{w}) = s(\hat{w}^T \hat{x})$, and $P(y = 0|\hat{x}; \hat{w}) = 1 - s(\hat{w}^T \hat{x})$; this is consistent with the way the sigmoid function is used to produce the predictions for $y$. $\hat{w}$ should be chosen such that this likelihood of predicting the correct label is maximized. Instead of maximizing this likelihood function directly, any strictly increasing function of this likelihood can be maximized. In order to simplify the gradient calculations, the log of this likelihood can be maximized. As maximizing the log likelihood cost function is equivalent to minimizing the negative of the function, and many optimization algorithms minimize loss functions, the optimal weights can be selected by minimizing the negative log likelihood cost function:

$$C(\hat{w}) = - \sum_{i=1}^{n} y_i \log(\hat{w}^T \hat{X}_i) + (1 - y_i) \log(1 - \hat{w}^T \hat{X}_i)$$

2.4.2 Perceptron

Logistic regression could be modified to output the labels -1 or 1 directly, instead of inferring these labels from the produced likelihood function. This can be done using the perceptron
algorithm, an algorithm inspired by the anatomy of the brain. Neurons in the brain receive and combine electro-chemical signals from other neurons, and then send a signal of their own based on this concatenated input. Like neurons, perceptrons output a linear combination of their input. In the basic single-layer perceptron, each feature from an input feature vector feeds into a central perceptron. Each link between an input feature and the central perceptron is assigned a weight, often initialized to zero. Like the weight vector from logistic regression, these weights include a bias weight, and each feature vector includes a corresponding 1 as the first element. With each feature vector \( \hat{x} \) in the training data, the predicted label \( \sigma(\hat{w}^T \hat{x}) \) is used to update the weight vector, where \( \sigma(z) \) returns -1 or 1 depending on the sign of \( z \). If \( \sigma(\hat{w}^T \hat{x}) \) is the same as the true label \( y \), the weight vector remains the same. If the prediction differs from the true label, the weight vector is updated using the following equation: \( \hat{w} = \hat{w} + y \hat{x} \). This weight update procedure is repeated until all predictions in the training set are correct.

The major issue with the single-layer perceptron is that it classifies linearly separable data. Each point in a linearly separable set of data of dimension \( n \) can be correctly grouped using a separating hyperplane of dimension \( n - 1 \). However, data is not often linearly separable. This problem could be alleviated by mapping the data into a higher dimensional feature space using kernel functions.

Note that after being trained on \( n \) training samples, the weight vector can be written as \( \hat{w} = \sum_{i=1}^{n} \alpha_i y_i \hat{X}_i \), where \( \alpha_i \) is a count of the number of times feature vector \( \hat{X}_i \) was misclassified. The predicted label for any feature vector \( \hat{x} \) can then be written as \( (\sum_{i=1}^{n} \alpha_i y_i \hat{X}_i)^T \hat{x} = \sum_{i=1}^{n} \alpha_i y_i (\hat{X}_i \cdot \hat{x}) \). This is the dual form of the perceptron, and it is necessary to utilize kernel functions. Instead of mapping each training sample into a higher-dimensional feature space, a kernel function returns the inner product of two data points in this higher dimensional space. Any kernel function can replace the linear kernel function \( (\hat{X}_i \cdot \hat{x}) \) in the dual perceptron prediction equation. Common examples of kernel functions include Gaussian RBF and Polynomial kernels.

### 2.4.3 Neural Networks

The brain does not consist of a single neuron that accepts input from various sensory neurons and sends an output signal to a motor neuron, like the single-layer perceptron does. Instead, the brain consists of a large interconnected network of nearly 100 billion neurons. This concept can be used to extend the single-layer perceptron into a neural network consisting of several layers of neurons. Neural networks consist of an input layer that contains an input neuron for each feature in a...
feature vector. The neurons in this input layer than feed into the first hidden layer. A network can contain any number of hidden layers, where a different set of weights is assigned to the connections between layers.

Each neuron takes in a linear combination of signals from the neurons in the previous layer, sums these weighted values, and outputs this sum through an activation function to the neurons in the next layer. The activation function used in hidden layers is often the hyperbolic tangent function, which maps values to the range $(-1, 1)$, similar to the sigmoid function. The activation function for the final output node in a network used to perform binary classification is the sign activation function used in the single-layer perceptron. As with the single-layer perceptron, each hidden layer contains a bias neuron. Note that a sparsity can be introduced to the network by omitting links between neurons.

Stochastic gradient descent and backpropagation can be used to train this network. After the prediction for one feature vector in the training data is calculated by propagating the input signal through the network, the error calculated through the loss function can be propagated backwards through the network, where the weights have been initialized randomly. The resulting error associated with each neuron is then used to calculate the gradient of the loss function. This gradient is then used in gradient descent to determine the optimal values for the weights of the network. This process is repeated for all samples in the training set until the weights converge. An important property of neural networks is that much like the brain, as a network is trained, different clusters of neurons specialize in detecting certain features within a subset of the input feature vector.

### 2.4.4 Deep Learning

The standard feed-forward neural network is just one class of neural networks; in recent years, convolutional neural networks have become a popular approach to solving image classification problems. Images often contain three dimensions of data: pixels along the width and height of the image, as well as a color value for each pixel. As this differs from the one-dimensional input feature vectors used in the standard neural network, CNNs often accept three-dimensional input matrices. The structure of the CNN, therefore, consists of three-dimensional layers of neurons. CNNs can contain several additional types neuron layers including convolutional layers and pooling layers.

Convolutional layers consist of “filters” that examine small subsets of the previous layer’s width and height. These filters are convolved over the entire input to the layer, outputting a two-dimensional activation map. Each convolutional layer contains a set of these filters. The output of the layer is therefore a three-dimensional set of two-dimensional activation maps. In order to create
these convolutional filters using neurons, each neuron is connected to a small region of neurons from the previous layer. The pattern of connection, called the neuron’s receptive field, is typically the same between neurons within a convolutional layer.

Pooling layers, which serve to decrease the size of the data, typically follow convolutional layers in a network. Like adding noise to a network, pooling layers help to reduce overfitting. Computations are easier and faster to perform on the smaller amounts of data produced by a pooling layer. At each depth in the layer that provides input to a pooling layer, a filter slides over the width and depth of the input layer. Rather than use a linear combination of the connected weighted inputs, a pooling layer outputs the maximum or average weighted value.

In addition to these new layers, the rectified linear unit, or ReLu, activation function is also included in many CNNs. This function simply returns the value passed to it if the value is greater than 0, or 0 otherwise. Though this threshold function is easy to implement and can increase the network’s training speed, it can completely disable some of the neurons in the network if the learning rate is set too high.

2.5 Related Work

Many early works in object grasping relied on matching sensor data to known models of objects within an evaluation set. Jang et al. utilize a stereo camera to record point clouds in [6]. Objects in the scene are matched to objects in a 3D model database to facilitate grasping. Though this method can generate successful grasps, it is time consuming and relies too much on known models of objects.

[11] were able to use a learning algorithm to find grasp points on previously unseen objects using two images of the object. The grasp point in each of the 2D images is estimated, and then the corresponding 3D point is triangulated. They use simulated objects with labeled grasp points to train their model. The 2D images are divided into rectangular segments using a grid. This approach is known as a sliding window approach, since a window slides over each box and considers the segments individually. Features considered for each segment include edges, texture, and color. The model is trained using several Naive Bayes assumptions, and uses a Maximum a Posteriori estimation method to choose the best grasp point on new objects. Although this method is successful, it is not the current state of the art.

Convolutional Neural Networks have been trained to estimate optimal grasp points on objects in works such as [10]. When training their CNN, each object is labeled with several grasp
CHAPTER 2. BACKGROUND

points. Since the model trains on one grasp chosen at random over many trials, it learns that average of the valid grasp points for each object. By modifying the network, Redmon et al. were able to efficiently categorize objects and detect grasp points on their objects. Another network modification allowed them to divide an image into a rectangular grid, and find the optimal grasp point in each segment. This removed the error created by averaging the grasp points, which was done in the first model. The likelihood of success could then be predicted for the grasp predicted in each cell.

Similar approaches have been taken by [8] and [13]. [1] and [12] use deep learning to detect grasp points for humanoid robotic hands. [9] is among the works that does not utilize simulated training data; a physical robot is used to attempt to grasp objects for the training phase. The downside to this approach is that it takes 700 robot hours to train the model.
Chapter 3

Methods

3.1 Baxter System

The grasping system implemented by Prof. Platt’s group on the Baxter robot is discussed in [3]. This system consists of a Structure Sensor mounted to the wrist of a Baxter robot. A cluster of small objects is placed on a table in front of the robot. The robot then points the sensor towards the scene, and the sensor captures a point cloud. Depending on the viewpoint selection strategy chosen by the user, the arm may move to a different position in order to obtain a better view of the scene. Finally, an algorithm will choose an optimal grasp on the objects within the sensed point cloud that captures the current scene. The grasp detection strategy used here consists of three main components: generating a set of possible grasps, determining whether each of these grasps is likely to succeed, and choosing the best grasp to execute.

3.1.1 Viewpoint Strategies

In order to detect an optimal grasp on a cluster of objects, a point cloud of the scene must be captured. In [4], one of four strategies is used to capture a point cloud. These strategies are comprised of three types of viewpoints, with the first type occurring as the first step in each of the strategies. This first type of viewpoint is a constrained random viewpoint; the sensor must be a certain threshold away from the user-defined center of the workspace. In addition, the angle between the sensor’s line of sight and the direction of gravity must also be bounded. The distance constraint is added to the random view to ensure that the graspable objects are within the sensor’s view range. The angle of the view is bounded so the system views the objects from above, and eventually selects
CHAPTER 3. METHODS

a grasp that begins above the pile. This is desired because it is easier to pick an object from the top of a pile than it is to reach into the side of the pile.

After capturing a point cloud from the first random pose, an initial target grasp is selected using the grasp detection algorithm. The second type of viewpoint aims to view the initial target grasp from an optimal viewpoint. This optimal viewpoint is chosen based on a modeled distribution of good grasps. Azimuth and elevation angles relative to a grasp candidate are mapped to a relative density of positive viewpoints. Kernel density estimation is used to obtain this density function for several objects within the simulated BigBird dataset. The authors note that for rectangular objects, maxima in the densities occur when the azimuth angle relative to the candidate is \( \pi \). They hypothesize that a viewpoint parallel to the approach axis of the proposed grasp is not useful, while a viewpoint orthogonal to this approach axis is. This hypothesis is tested by selecting the viewpoint with the highest density of positives relative to a random ground truth positive grasp on objects in the BigBird dataset. The number of good grasps detected from this optimal viewpoint is detected to be several times larger than the number of grasps detected from a random viewpoint. Therefore, this method of choosing optimal viewpoints is appropriate. One issue with this strategy is that the density models used to choose optimal viewpoints are unique to the objects from which the distributions are generated. However, in practice, the optimal viewpoints obtained from density models generated from rectangular objects detect grasps well on non-rectangular objects. Therefore, the distributions generated from rectangular objects are used to determine the optimal viewpoint for all real objects.

Finally, the third viewpoint is selected using the target grasp from the previous step. The sensor’s orientation is aligned with this target grasp, and the final point cloud is captured. Candidate grasps are generated and classified based on this point cloud, but the final scoring step in the grasp selection algorithm is not called on the set of good candidates. Instead, the grasp closest to the target grasp from the previous step is executed. The objective of viewing the scene from this final viewpoint is to ensure that the solution returned from the second viewpoint is collision free. The optimal grasp is assumed to have been found at the previous viewpoint; this step serves to check the trajectory generated when performing this grasp.

The viewpoint strategy is chosen by the user upon starting the system. The first strategy views the scene from all three viewpoints discussed above. The second strategy assumes that the trajectory to the optimal grasp detected at the second viewpoint is clear, and the third viewpoint is not used. The third strategy assumes that the initial random viewpoint can detect a good grasp, and confirms this using the third viewpoint; the second viewpoint is omitted. In the final strategy, the grasp detected using the random viewpoint is executed. The authors hypothesize that viewing the
scene from all three viewpoints will result in the highest grasp success rate. The other strategies, however, would certainly be useful in a real-world setting if their success rates were comparable to the success rate of the 3-viewpoint method while the computation time was lessened.

3.1.2 Grasp Candidate Generation

After capturing a point cloud for which grasps will be calculated, a set of possible grasps must be generated. First, the algorithm randomly selects points within the dense sections of the point cloud, which are assumed to contain objects of interest. Next, the surface normal and axis of principal curvature for each sampled point are determined. As the angle of the optimal grasp is assumed to be orthogonal to the surface of the object, a grasp candidate is selected so that the palm of the hand is parallel to the object’s surface and the approach vector is orthogonal to this surface. A model of the robot’s hand is then inserted into the point cloud such that the model does not intersect any points in the cloud, and the hand is aligned with the calculated surface normal. The model of the hand is then moved towards the cloud until contact is made between the fingers and any points. The closing of the hand model around the point cloud is then simulated. If any points remain between the two fingers once the simulated grasp is complete, the grasp is considered feasible, and is added to the set of grasp candidates. Examples of simulated grasp candidates can be seen in figure 3.1.

![Figure 3.1: Grasp Candidate Generation](image)

3.1.3 Grasp Classification

Any one of a multitude of machine learning algorithms could be implemented in order to classify a grasp as good or bad. The recent related work discussed in chapter 2.5 shows that convolution neural networks are currently a popular approach to the grasp detection problem. Therefore, a convolution neural network is implemented in [3] to classify candidate grasps. After deciding to use a CNN, two aspects of the network implementation remain to be selected: the network architecture and the representation of grasp hypotheses to be passed to the network.
CHAPTER 3. METHODS

As noted in [3], the network architecture used to classify grasps is the common LeNet architecture. Originally used to classify hand-written digits, this network architecture is among the most popular across many disparate domains. Many related works, including [7], use LeNet as the architecture of their network. The LeNet network consists of six layers: two sets containing convolutional layers followed by pooling layers, an inner product layer with a ReLu activation function, and an inner product layer with a softmax activation function. The properties of each of these layers are discussed in detail in chapter 2.4.4.

3.1.4 Grasp Representation

Convolutional neural networks work accept three-dimensional feature matrices as input. As discussed in chapter 2.4.4, a two-dimensional color image can be represented as a three-dimensional feature matrix; these images are often used as input to a CNN. Therefore, the grasp candidates and their contexts must be transformed into this format. The most important information to encode within these representations is the geometry of an object that the end effector would come into contact with, as the contact between the object and the closing gripper determines whether a grasp is good. The authors achieve this by representing a grasp as a set of three images. Each of these images is related to the volume of points that fall within the bounds of the robot’s parallel gripper. The points in this rectangular cuboid are voxelized and then projected onto a plane parallel to the grasp approach axis and the gripper closing direction. In the first two of these three images, the color values assigned to each pixel relate to the average height of the occupied voxels in the corresponding column of the projection. The first image contains only non-occluded voxels from the partial cloud; these voxels would correspond to the areas of the object that the gripper makes contact with. The second image contains the occluded voxels from the original partial point cloud. The third image contains the voxels from the first image, but the 3 color values now relate to the 3 axes of the normal vector estimated at each voxel. This set of three images, like the one shown in figure 3.2 (c-e), represents a grasp well because the images include information about the points that the gripper would touch, those that it would not touch but may influence the success of the grasp, and the surface normals of the object.

3.1.5 Training Data

Before the grasp detection algorithm can be utilized, the neural network needs to be trained. This training process involves feeding batches of sample data points, each with a correct ground
CHAPTER 3. METHODS

Figure 3.2: Grasp Representation

truth label, into the network. Using the methodologies discussed in chapter 2.4, these training points are used to update the network’s weights. Training a CNN often requires hundreds of thousands of data points, and hand-labeling hundreds of thousands of grasps would take a very long time. Instead, simulated data can be used to train the network. This simulated data comes from the BigBird dataset, which contains mesh models of common household items, as well as partial point clouds of each object, which represent what a depth sensor would see if it viewed the mesh from a particular viewpoint.

In order to generate training data for the CNN, a grasp and its corresponding point cloud must first be generated. The point clouds are selected from the BigBird dataset, and then used to generate a set of grasp candidates using the procedure described in chapter 3.1.2. The point cloud representation of object with the gripper model in the grasp pose can then be converted into the set of images accepted by the network using the procedure defined in chapter 3.1.4. The problem of labeling the proposed grasp as good or bad still remains. However, the meshes given with the partial point clouds in the BigBird dataset can be used to solve this issue. These corresponding mesh models are used to generate the labels by analyzing whether the proposed grasps form a frictionless antipodal grasp on the object. A frictionless antipodal grasp occurs when the gripper closes around an object, and the surface normals at the points of contact on the object are parallel to the direction in which the grippers close, as shown in figure 3.3. By ensuring that a grasp is frictionless and antipodal, the chance of an object slipping out from the gripper as it closes are low. The CNN is therefore trained to detect whether a grasp is a frictionless antipodal grasp based solely on the partial point cloud.
CHAPTER 3. METHODS

Figure 3.3: Frictionless Antipodal Grasp

3.1.6 Grasp Selection

Once a set of grasp candidates have been generated and classified as good or bad, the best grasp from the subset of good grasps should be executed. Though some sort of machine learning algorithm could be implemented to score each grasp, the authors of [3] find that an intuitive heuristic can be implemented to assign a score to each grasp. The goal of this heuristic function is to assign higher scores to grasps that are more likely to succeed. Several desired features of the optimal grasp are analyzed in order to design this heuristic.

Before computing the heuristic value for each grasp classified as good, several additional filtering steps should be performed in order to ensure possibly infeasible grasps are not selected. For instance, if the end effector would need to move below the possibly unseen table plane, the grasp is not feasible; therefore, it is filtered out of the good set before the heuristic value is even calculated. In addition, grasps that require the any of the arm’s joints to move close to their joint limits is undesirable. If a grasp requires the arm to be in such a configuration, the grasp is filtered out.

Once the set of good grasps has been filtered, each grasp can be assigned a heuristic value calculated with the heuristic function outlined here. Grasps from the top of a pile of objects are the most likely to succeed, as objects at the top of a pile would presumably be less cluttered. Therefore, grasps that occur higher above a table are scored higher than those that are lower. A similar property holds for the approach angle of a grasp; grasps that approach an object from the top are more likely to succeed than grasps that approach the object from the side. A side grasp could be attempted on an object located underneath another object. Side grasps would be more likely to fail in this configuration because even a frictionless antipodal grasp on an object may not be enough to lift it from beneath a second object. Additionally, top grasps are more likely to succeed on rectangular objects. Recall that frictionless antipodal grasps are formed when the surface normals at the points
CHAPTER 3. METHODS

of contact are parallel to the direction in which the gripper fingers close. Even among clutter, one surface of a rectangular object will be parallel to the table. Therefore, frictionless antipodal grasps must come from directly above the object in this case. Side grasps could be practical in settings where a single tall object, such as a bottle, is standing upright, but they are not practical in a cluttered scene. Therefore, grasps with an approach vector closer to the downward vertical direction will be scored higher.

A grasp is also obviously more likely to succeed if the object fits better within the gripper’s fingers. Grasps on objects much smaller than the maximum gripper finger width are more robust to slight kinematic or sensor errors, and will be more likely to succeed. Therefore, the smaller the volume of occupied points between the gripper fingers, the higher the grasp should be scored. Finally, the optimal grasp should not be overly complicated to execute. Extraneous movements through the arm’s joint-space are time consuming and not desired. To this end, the closer a grasp is to the arm’s current configuration, the higher the heuristic should score it.

Within the heuristic function, four sub-functions, which correspond with each of the four concepts described above, are written to generate scores between 0 and 1. The final heuristic value for a grasp is the product of these sub-functions. After each good grasp is assigned a score, the system executes the grasp with the highest heuristic score. Note that this grasp detection algorithm makes no effort to identify individual objects within a cluster; the best grasp detected on the cluster as a whole is executed. This algorithm clears a cluttered scene of all objects, but as it is currently implemented, the user cannot indicated which object should be grasped next. The group has, however, investigated such a system in [2].

3.1.7 Results

Several experiments are performed in [4] and [3] in order to judge the effectiveness of the system. These experiments measure the grasp success rate when a group of 10 objects are clustered together on a table in front of the robot. The full experimental setup and procedure are detailed in chapter [4]. The interesting experiments occur in [4], where the efficacy of the four different viewpoint strategies is examined.

In [4], approximately 150 grasp attempts are executed using each of the four viewpoint strategies. As one would expect, the strategy that used all three viewpoints was the most successful, with a grasp success rate of 87%. As described in chapter [3.1.1], this strategy involves detecting a grasp from a random viewpoint, detecting an optimal grasp from an optimal viewpoint, and detecting
CHAPTER 3. METHODS

a collision-free grasp close to this optimal grasp. Somewhat surprisingly, the strategy that used a random viewpoint and the third collision-checking viewpoint outperformed the strategy in which the best grasp from the optimal viewpoint was executed without checking for collisions. The success rates were 83% and 78%, respectively. However, it is interesting to note that a majority of the errors in the strategy in which the grasp from the optimal viewpoint is executed come from forward kinematics. Of the 31 grasp failures in this case, 21 were forward kinematic errors. When the random and collision-checking viewpoints were used instead, only 9 errors were caused by the forward kinematics. This is most likely a result of the fact that the collision-checking viewpoint is located directly above the general area of the executed grasp. The optimal viewpoint, however, is orthogonal to the approach axis of the proposed optimal grasp. Therefore, the trajectory from the optimal viewpoint to the grasp can be longer than the trajectory from the collision-checking viewpoint to the grasp. As the authors state, the larger the configuration-space distance a trajectory has, the larger the kinematics error in the Baxter arm will be. It is worth noting that only 10 failures occurred when grasp returned from the optimal viewpoint was defective. 14 failures occurred because of defective grasps returned when only the random and collision-checking viewpoints were used. This shows that the optimal viewpoint is useful for decreasing grasp failures due to bad grasp predictions. The full results of these Baxter experiments can be seen in table 3.1.

<table>
<thead>
<tr>
<th></th>
<th>Attempts</th>
<th>Success Rate</th>
<th>FK</th>
<th>Grasp</th>
<th>Other</th>
<th>Total Failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Views 1-2-3</td>
<td>131</td>
<td>0.87</td>
<td>10</td>
<td>4</td>
<td>3</td>
<td>17</td>
</tr>
<tr>
<td>Views 1-2</td>
<td>141</td>
<td>0.78</td>
<td>21</td>
<td>10</td>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td>Views 1-3</td>
<td>154</td>
<td>0.83</td>
<td>9</td>
<td>14</td>
<td>3</td>
<td>26</td>
</tr>
<tr>
<td>Views 1</td>
<td>153</td>
<td>0.75</td>
<td>28</td>
<td>9</td>
<td>2</td>
<td>39</td>
</tr>
</tbody>
</table>

3.2 UR5 System

3.2.1 Motivation

As seen in chapter 3.1.7 the drawback to using the random and optimal viewpoints instead of the random and collision-checking viewpoints is that the number of grasp failures due to kinematic errors will increase. However, fewer grasps will fail due to poor grasp predictions under this optimal viewpoint strategy. If kinematic errors were somehow eliminated or significantly reduced, the
optimal viewpoint strategy would be superior to the collision-checking strategy. Though using random, optimal, and collision-checking viewpoints would result in a higher success rate than either of the two aforementioned strategies, an additional viewpoint would have to be analyzed, which would increase the time required to perform each grasp. The third view is useful for moving the arm closer to the grasp configuration and for detecting any obstacles that may have been occluded in a previous viewpoint, though it may not be entirely necessary.

In order to attempt to reduce the grasp failures caused by kinematic errors, it is important to understand why these errors occur in the first place. They are a result of the design of the Baxter arms, which have inaccurate forward kinematics. Baxter arms are compliant, as they are designed to be used safely around human collaborators. To accomplish this, the joints are springy. This causes the arm to jerk around the workspace rather than move in a smooth trajectory. If the system was instead implemented on a robotic arm with more accurate forward kinematics, these errors could be completely alleviated. For this reason, it was decided that the system should be implemented on the 6-DoF UR5 robot arm.

3.2.2 UR5 Port

In order to port system to the UR5, the general grasp detection algorithm would require very few modifications besides changing robot-specific constants such as the width of the gripper. Most of the changes necessary for porting the grasp detection system to a new robot involve planning and trajectory generation.

As described in chapter 2.1, the TrajOpt motion planner utilizes the OpenRAVE simulation package to create collision-free trajectories for a robot arm. A model of the environment and the inverse kinematics are required for each arm used here. In order to ensure that the arm does not collide with any of the immovable obstacles in its environment, the environment is modeled. The objects added to this environment include the table, the robot’s stand, and any surrounding walls. The environment file is an XML file that lists a set of rectangular prisms that represent the obstacles, specified by dimension and position. Additionally, a model of the arm, hand, and sensors must be added to the environment. This arm model, a Collada model stored in a .DAE file, is used when planning trajectories, and is parameterized over the arm’s joints. Though the UR5 arm model is available as only a .URDF model file, a .URDF model can easily be converted to a .DAE using a ROS package. This .DAE file is also used to generate the inverse kinematic solutions in the IKFast database. With the environment model for collision checking and the IKFast database for solving the
CHAPTER 3. METHODS

inverse kinematics, trajectories from a given joint configuration to a desired end effector pose can be generated using TrajOpt.

In order to interact with the UR5 arm, the ur_modern_driver ROS package can be utilized. This package provides an interface to send joint velocity commands to the UR5’s internal controller in order to move the arm. Since the trajectory planning algorithms return a set of joint configurations, this set of joint configurations must be converted to a set of joint velocity commands. This can be done using an on-line algorithm that relies on the current joint positions of the arm and the next goal point within the trajectory. While the arm is not at the desired trajectory point, the configuration-space distance between the current configuration and goal configuration is calculated. This error distance is used to calculate the velocity necessary to move to the next trajectory point by dividing the error distance by a time step-size constant. Before a joint velocity command is sent to the arm, several properties are first checked. Maximum velocity, acceleration, and force thresholds are passed in as parameters to the function that sends velocity commands to the arm. The magnitude of the new velocity command, acceleration required to move at these new joint velocities, and the force readings from the arm are used to determine if the velocity should be decreased. This allows the velocity of the arm and jerky movements between velocity commands to be reduced. This process of moving towards the next trajectory point is repeated for each point in the generated trajectory until the last configuration is reached, at which point the end effector will be in the desired pose.

3.2.3 UR5 Improvements

After successfully porting the grasping system to the UR5 arm, several improvements were made to increase the speed and grasp success rate of the system. The major improvements include changes to the trajectory planner, the way point clouds are captured, and the viewpoint system.

Though the TrajOpt planner generates efficient trajectories in a relatively short amount of time, it is often unnecessary to use this powerful optimization algorithm to execute the simple trajectories between viewpoints, or to grasp poses. Additionally, TrajOpt may fail to find a trajectory from a grasp pose in the center of the workspace to the box near the edge of the workspace, for instance. Both of these problems could be solved if a hierarchical planner was implemented to first attempt to find a trivial solution and then use a more complicated planning algorithm if this first one failed. The hierarchical planner implemented on the UR5 first checks the path from the starting configuration to the goal configuration, which is found using the arm’s inverse kinematics. If this plan, linear in configuration space, is collision-free, the planner would execute it. If not, the
CHAPTER 3. METHODS

planner attempts to find a trajectory using TrajOpt. If TrajOpt fails to find a solution, an RRT* planner is called. If this final planner is unable to find a valid path within 10 seconds of searching, the movement is considered to be impossible. This hierarchical planner speeds up the trajectory calculation in simple cases, and can find valid solutions even when TrajOpt fails.

Another major improvement implemented in the system on the UR5 is the inclusion of point cloud stitching. The three most successful viewpoint strategies capture at least two point clouds of the scene, though the information from the initial point clouds is thrown away once a new one is captured. The system could make better judgments about each potential grasp, as well as propose a more accurate set of candidate grasps, if point cloud information was saved from previous viewpoints. Concatenating multiple point clouds using PCL can be done with a simple command. However, it was found that the clouds did not align properly when using the RealSense SR300 sensor. Though this sensor issue could have been fixed using some sort of calibration routine, a Structure Sensor was mounted to the wrist instead. Point cloud alignment with this sensor was accurate to two millimeters, which was found to be sufficient. Initially, the concatenated clouds were only used during the candidate generation phase. After retraining the network using simulated data that included segments of an object’s partial point cloud captured from two viewpoints, the concatenated clouds were used to classify the grasps as well.

Stitching multiple point clouds helped to reduce the amount of data lost between views. Viewing the entire scene from multiple angles and storing the data in one point cloud on which to perform grasp detection could therefore be a reasonable viewpoint strategy. A fifth viewpoint strategy was introduced that used two hard-coded viewpoints in an attempt to capture the entire scene. These viewpoints were chosen to be 50 centimeters above the center of the workspace, 90 degrees apart from one another. This strategy proved to be faster than the other strategies that utilized at least two viewpoints because the point cloud was only recorded from the initial viewpoint; no processing or grasp predictions occurred at this step. There were, however, several drawbacks to this method. Though the views were centered around the workspace, objects that weren’t centered directly in front of the robot were undetectable. In addition, the views on the objects in the center of the workspace were sub-par. Even when the clouds combined, the data obtained from each viewpoint did not overlap much. Additional hard-coded views could be recorded around the object to increase the likelihood of seeing the entire scene. However, this would be no more efficient than the visual servoing method implemented in [3].

Besides these major extensions, many incremental improvements were made to the system on the UR5. These smaller improvements include enforcing stricter joint limits on the arm to avoid
CHAPTER 3. METHODS

self collisions, tuning the speed, acceleration, error, and force threshold parameters to create smoother trajectories, and adding force cutoff capabilities in case the arm collides with the table or an object.

3.2.4 Portability

Besides alleviating the forward kinematics error found in the Baxter system, migrating to the UR5 made the system more portable. Though the arms can be removed from a Baxter robot, mounting them elsewhere could prove to be difficult. A mounting system was designed by a member of the lab group in order to easily mount the UR5 arm elsewhere. With a portable arm, the group could test their grasping system at picking competitions, such as the Amazon Picking Challenge. The arm could also be mounted to some sort of mobile base in order to perform grasping tasks in an outdoor environment. The feasibility of this idea was demonstrated using a Warthog mobile robot. The UR5 and its custom mount were attached to the top of the Warthog, which was piloted outside. Though the system would ideally be straightforward to mount elsewhere, the environment did have to be remodeled to include portions of the Warthog that were in close proximity to the arm. The system was able to successfully grasp trash bags, boxes, and buckets from the ground. This system would eventually be used to pick up small gardening tools and various forms of litter in an outdoor environment. Eventually, this system could be implemented on a space rover and used to collect samples from the surface of a celestial body.

Figure 3.4: Warthog Mobile Robot with UR5 Arm
Chapter 4

Experiments

The experiments performed to evaluate the grasp success rate of the system with the UR5 arm are similar to the experiments implemented on the Baxter system. This allows us to compare the results between the two methods. The system’s performance is tested using two object sets: the object set used to perform experiments on the Baxter system, as well as the more difficult YCB set.

4.1 Setup

For each experimental trial, 10 objects are selected at random from a dataset. These objects are placed within a small cardboard box, and the box is shaken. The box is then turned upside-down over the table in front of the UR5 arm. The objects fall out into a cluttered pile. Next, the system is started, and the viewpoint method is specified. The system then grasps each object in the pile, and drops it in a box to the side of the table; this constitutes a successful grasp. The trial is over when the system has cleared the table of objects, fails to grasp the same object three times in a row, or fails to detect any suitable grasps three times in a row.

There are several classifications of errors that can occur in these experiments. Forward Kinematic errors are caused when the planned grasp should succeed, but the kinematics of the arm cause the grasp to fail. Bad grasp hypothesis errors occur when the grasp chosen by the algorithm is not good. Oftentimes, this error is a result of an occluded obstacle. Touched object before closed errors occur when the fingers of the gripper touch an object before reaching the final grasp configuration, causing the object to shift and the grasp to fail. Dropped after lifted errors occur when an object is initially successfully grasped, but the object falls out from between the gripper’s fingers on the way to the box.
CHAPTER 4. EXPERIMENTS

Two object sets are used throughout these experiments. The first is the same set of objects on which the Baxter system was evaluated in [4]. This set consists of 25 objects, which can be seen in figure 4.1. By recreating the experiments performed on the Baxter arm using the random and optimal viewpoints strategy, the relative success of the two systems can be compared.

![Baxter Objects](image)

Figure 4.1: Baxter Objects

The second set of objects comes from the YCB object set. As the Robotiq gripper attached to the UR5 was too large to grasp the smaller items, such as washers, they were excluded from the experimental set. Additionally, some of the heavier bottles and tools were excluded from the experimental set because they were too heavy for the arm to pick up. The complete YCB object set can be seen in figure 4.2. The YCB set is important because it contains a good representation of items that would typically be grasped in a picking challenge or competition.

![YCB Objects](image)

Figure 4.2: YCB Objects

4.2 Results

A summary of the UR5 results using the random and optimal viewpoints strategy (labeled Views 1-2 in [4]) on both the Baxter object set and the YCB object set can be seen in table 4.1.
CHAPTER 4. EXPERIMENTS

Table 4.1: YCB and Baxter Object Set Results

<table>
<thead>
<tr>
<th></th>
<th>YCB Dataset Views 1-2</th>
<th>Baxter Dataset Views 1-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attempts</td>
<td>108</td>
<td>112</td>
</tr>
<tr>
<td>Success Rate</td>
<td>0.89</td>
<td>0.86</td>
</tr>
<tr>
<td>FK</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Grasp</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>TOBC</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dropped</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Total Failures</td>
<td>12</td>
<td>15</td>
</tr>
</tbody>
</table>

4.3 Analysis

The most important fact to note in the first set of experiments is that zero failures occur due to the kinematics of the UR5. Porting the system to the UR5 did diminish the effects of the errors so prevalent in the Baxter system. Additionally, the grasp success rate was higher on the UR5 using two views 1 and 2 than it was on Baxter using view 1 and 3. As it was hypothesized that view 3 puts the arm in a configuration closer to the grasp goal, the kinematic errors don’t affect the results in these cases as much with Baxter. An 83% grasp success rate (154 attempts) was achieved on Baxter using views 1 and 3, but a grasp success rate of 85% (112 attempts) was achieved on the UR5 using only the first two viewpoints.

The most common type of failure on both object sets was due to bad grasp hypotheses. One common failure on both sets was due to the system attempting to grasp two objects at the same time. When two objects with narrow volumes were located next to one another, the system could identify the best grasp as being on these two objects since they fit within the maximum gripper width requirement of the heuristic. The system had no way of discriminating between different objects within the clutter pile.

Within the Baxter object set, 9 of the 15 failures occurred because of bad grasp hypotheses. These bad grasps could have been caused by missing information in a point cloud. If the sensor failed to detect some section of the surrounding objects or obstacles, a bad grasp could certainly be classified as a good grasp. Grasps that attempted to pick up a long object, such as the banana or the shoelaces, from one side sometimes failed. Though the grasps would have succeeded had they been executed near the center of the object, other objects within the clutter pile often prohibited this. Additionally, the heuristic rewarded grasps that encapsulated fewer points so the objects would better fit within the gripper width. This could have encouraged the system to grasp objects from their skinnier ends, rather than the thicker middle sections. Several of these grasps failed when a
side-grasp was attempted on spherical objects such as a ball or the plum. These grasps may have been selected when the heuristic favored smaller objects that fit better within the gripper, outweighing the penalty brought on by the side-grasp. Finally, during the last trial, 3 bad grasps caused issues, as did 2 attempted grasps on the screwdriver where it was dropped before being successfully put into the box. The grasp success rate was much lower during this trial, so this clutter case may have been particularly difficult.

Of 108 grasps attempted on the YCB object set, 12 grasp failures occurred. 10 of these failures were bad grasps, while the other two occurred when the arm dropped the object before making it back to the box safely. Perception was more of an issue with these more difficult objects. For instance, the overturned Duplo Lego blocks were not properly detected by the sensor because the bottom portion was hollow. The red cup also posed a problem; though the system could accurately detect and grasp the lip of the cup, the handle was often occluded. When a grasp was attempted on the portion of the lip near the handle, the arm would fail to grasp the cup. One interesting result occurred consistently with the bag of marbles. Though the sensor could not detect the reflective marbles or the mesh bag that held them, it was able to detect and grasp the cardboard tag at the top of the bag. The system also had particular trouble with the small spherical peach. Overall, the grasp success rate on the YCB set was 88.9%; this is higher than the grasp success rate obtained by the UR5 system on the Baxter object set.
Chapter 5

Conclusion

The systematic failure of the forward kinematics on the Baxter Research Robot can significantly decrease the grasp success rate of the experimental grasp systems developed in Northeastern’s CCIS Helping Hands Lab. In order to reduce the kinematic failures experienced during experiments using the group’s main grasping system, the system was ported to Universal Robots’ more accurate UR5 robotic arm. The grasp success rate of the system using this arm was found to increase, as no kinematic failures occurred. The accuracy of the system with the new arm was then evaluated on a new set of objects that were representative of the objects that may be found in a picking challenge. Grasp failures with this new system were found to be caused mainly by imbalances between the sub-sections of the heuristic, partial point clouds with missing information, and the classifier. In order to eventually solve some of these problems, instead of classifying a grasp as good or bad, a CNN could be trained to output the optimal grasp on an object directly. The portability of the new arm was demonstrated by mounting it to a Warthog mobile robotic base and successfully performing grasps. The UR5 grasping system will serve as the basis for future manipulation and grasping systems implemented by the lab group.


BIBLIOGRAPHY

