AN EXPERIMENTAL TEST BED FOR ROBOTIC GRASPING

By

John H. Behmer

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Approved:

____

Jeffrey C. Trinkle, Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

April 2011
(For Graduation May 2011)

© Copyright 2011
by
John H. Behmer
All Rights Reserved

CONTENTS

LIST OF TABLES . . . vi

LIST OF FIGURES . . . vii

ACKNOWLEDGMENT . . . viii

ABSTRACT . . . ix

1. INTRODUCTION . . . 1
   - 1.1 Background . . . 1
   - 1.2 Previous Work . . . 3
   - 1.3 Preliminary Setup . . . 5

2. CONTROL SYSTEM DESIGN . . . 8
   - 2.1 System Dynamics and Disturbance Rejection . . . 8
     - 2.1.1 Forward and Inverse Kinematics . . . 9
     - 2.1.2 Gravity Compensation . . . 13
     - 2.1.3 Friction Compensation . . . 16
     - 2.1.4 Velocity Feedback . . . 19
       - 2.1.4.1 Simple Moving Average . . . 21
       - 2.1.4.2 Exponential Moving Average . . . 21
   - 2.2 Survey of Controllers . . . 23
     - 2.2.1 PID Acceleration Controller . . . 24
     - 2.2.2 PID Torque Controller . . . 25
     - 2.2.3 Impedance Control . . . 26
   - 2.3 Trajectory Formation . . . 27
     - 2.3.1 Step Response . . . 28
     - 2.3.2 Trapezoidal Profile . . . 29
     - 2.3.3 Cartesian Space Control . . . 31
   - 2.4 Selecting PID Gains . . . 34
     - 2.4.1 Manual Selection . . . 35
     - 2.4.2 Using Inertia Parameters . . . 36
   - 2.5 Performance Analysis . . . 39
   - 2.6 Hand Controller . . . 42

3. CAMERA AND DATABASE DESIGN 44
3.1 Software Design 44
3.1.1 UDP Communication with Arm and Hand 45
3.1.2 Interfacing with Tracking System 46
3.2 Camera Calibration 48
3.2.1 Transforming Handedness of Coordinate Frames 49
3.2.2 Tracking Tools Calibration Procedure 50
3.2.3 Alignment of Coordinate Frames 50
3.2.3.1 Arc Formation about Base Frame 51
3.2.3.2 Non-Linear Least Squares Estimation 53
3.2.4 Extensions to Robot Parameter Estimation 56
3.3 Rigid Body Tracking 57
3.3.1 Conventions for Rigid Body Coordinate Frame 57
3.3.2 Transforming Handedness of the Camera Frame 58
3.3.3 Transforming Right Handed Data into the Robot Frame . . . 60
3.4 Grasp Acquisition Database 61
3.4.1 Implementation and Design 61
3.4.2 Data Stored 61
4. CONCLUSION 63
4.1 Future Work 63
4.2 Summary 65
REFERENCES 67
A. FORMULA DERIVATIONS 70
A.1 Backward Euler Velocity Error 70
A.2 Trapezoidal Profile Curve 70
A.3 Selecting Gains Based on Inertia 72
APPENDICES
B. TECHNICAL NOTES 74
B.1 Arm and Hand Controller 74
B.1.1 Denavit-Hartenberg 74
B.1.2 Manufacturer Provided Parameters 74
B.1.3 Gravity Compensation Equations 77

B.1.4 PID Gains 78
B.2 MATLAB and Simulink 78
B.2.1 Hardware Drivers 78
B.2.2 CAN Initialization 79
B.3 MATLAB Data Definitions 80
B.4 UDP Packet Definitions 83
B.4.1 Arm Command UDP Packet 83
B.4.2 Arm Feedback UDP Packet 84
B.4.3 Hand Command UDP Packet 85
B.5 Known Issues 86
B.5.1 Controller Issues 86
B.5.2 Barrett Hardware Issues 87
B.5.3 Barrett Hardware Operation 88
B.6 List of Repairs 89
C. MATLAB CODE 90
C.1 Symbolic Matrix Calculations 90
C.2 Maximum Inertia Calculations 93
C.3 Cartesian Space Simulation 99

LIST OF TABLES

B.1 Denavit-Hartenberg Parameters for the 4 DOF Barrett WAM 76

B.2 Link Masses and Center-of-Masses for the 4 DOF Barrett WAM 76

B.3 Joint Space PID Gains for the 4 DOF Barrett WAM 78

B.4 UDP Packet Definition: Arm Command 83

B.5 UDP Packet Definition: Arm Feedback 85

B.6 UDP Packet Definition: Hand Command 86

LIST OF FIGURES

1.1 Picture of Barrett Hand Demonstrating 4 DOF 6

2.1 Viscous Friction Estimate from Experiments 17

2.2 Connection of Negative and Positive Viscous Friction Plots 18

2.3 Plot of Lag Incurred by Using Backward Euler Method 20

2.4 Plot of Exponential Moving Average Weights 22

2.5 System Diagram of PID Acceleration Controller 24

2.6 System Diagram of PID Torque Controller 26

2.7 Sample Damping for Linear PID Systems 28

2.8 Plot of Trapezoidal Profile Position and Velocity Curves 30

2.9 System Diagram of Closed Loop PD with Inertia Term Compensated . 36

2.10 Sample Joint Space Move with Trapezoidal Profile 40

2.11 Sample Joint Space Move with Trapezoidal Profile (Triangle) 41

2.12 Sample Cartesian Space Move with Trapezoidal Profile 42

3.1 System Diagram of Connected Machines and Communication 45

3.2 Picture of Experimental Grasping Test Bed 47

3.3 Plot of Marker Locations for Calibration 52

3.4 Coordinate Frame Definition for a Trackable Object 57

B.1 Denavit-Hartenberg Diagram for the 4 DOF Barrett WAM 75

ACKNOWLEDGMENT

First I would like to express my thanks to family and friends, especially to my parents, who encouraged me to do this Master's even as I told them it was crazy to do this in just one extra year. I could not have done it without your constant support and motivation. I would also like to thank my brother and sister, Russ and Mandy, as well as my wonderful girlfriend Stasia, who will never admit she actually enjoyed those late night conversations about coordinate frames and unit quaternions. To the rest of my family – aunts, uncles, cousins, and grandmothers – and to all of those I spent time with in the lab these last two years – Jeremy Betz, Chris Jordan, Jed Williams, Will Macaluso, Hans Vorsteveld, Kane Hadley, Matt Dallas, Binh Nguyen, and Emma Zhang – it was a great two years. Best of luck to you all, and please keep in touch.

Next I would like to thank the members of the Computer Science department, especially Pam Paslow, Shannon Carothers, and Terry Hayden, for their outstanding kindness and for all of the paperwork, purchasing, and organizing they did on my behalf. To all of my professors, thank you for sharing such a wealth of knowledge with me both in and out of class, especially Alan Desrochers, John Wen, and David Hollinger. Your extraordinary teaching and support helped me not only to carry out this thesis, but to gain a genuine excitement for Computer Science and Systems Engineering.

Finally, above all, thank you to my outstanding graduate thesis advisor, Jeff Trinkle. I could not have asked for a more helpful, knowledgeable, and down-to-earth person to advise me over the course of the last two years. You shared an extraordinary amount of knowledge just about every time we talked, not only in Robotics but also in many other areas. Thank you for going out of your way to help me with each and every problem I faced along the way.

ABSTRACT

Humans have been on earth for hundreds of thousands of years. We have evolved traits and abilities that allow us to be one of the most sophisticated species ever to live. Each and every day, we sense, touch, grasp, and manipulate with our hands. We rely on highly dexterous and capable fingers as they are so vitally important to our interaction with the environment. The human hand is extremely well-tuned to these behaviors, yet the complexity of such actions currently prohibits robotic devices from doing the same. If robots are ever to become mainstream in aiding and cooperating alongside humans, having such skills will be crucial to their success.

The field of robotic grasping is inherently multidisciplinary, spanning topics in control theory, dynamics, physics, mathematics, computer science, and even artificial intelligence. The complexity of designing a robotic system capable of grasping comes from necessity to delve into several of these major research areas. We approach the grasping problem from the ground up by constructing an experimental grasping test bed with both a robotic manipulator and a means of accurately tracking objects in the workspace. We demonstrate the effectiveness of this system not only in theory, but also on the actual hardware. Precision and repeatability are crucial to running experiments and studying their results, so all aspects of the system carefully analyzed and documented.

This thesis paper addresses many of the core issues that are crucial to studying the area of robotic grasping. We begin by developing a control system for a robotic arm and hand using software that allows for rapid prototyping in a research setting. We discuss methods for dynamic disturbance rejection and then implement both joint space and Cartesian space PID controllers, touching on the concept of impedance control. Next we explore infrared marker and rigid body tracking, and introduce a very precise method for coordinate frame transformations and alignment. We aggregate all control and tracking data into single, well-defined experiments that can be stored and retrieved from a Grasp Acquisition Database. This database can later be used to teach, replay, or study the effectiveness of particular robotic grasps.

CHAPTER 1
INTRODUCTION

1.1 Background

The potential for robotic innovation in today's world is tremendous. In fact, for years now scientists and engineers have been predicting strong growth in the field (see [1]); however, in certain areas this potential is yet to be exploited. Robotics are used in a variety of areas in industry. Some examples include robots on assembly lines, those for bomb diffusion, space and sea exploration, and even for the military. However, many of the tasks given to robots are either highly repeatable or controlled by humans, via a remote control, for example. Teaching a robot the level of intelligence of a human in any respect is a very difficult task. This is why we see a large portion of robotics companies today still targeting research labs as opposed to the household consumer. We have yet to fully exploit that potential for robotic innovation.

One major reason prohibiting the introduction of robots into homes is the level of human-robotic interaction (HRI) that is required. The area of HRI is a field of its own, but it brings up certain questions that are interesting to other aspects of robotics. One of these areas is robotic grasping and control. The problem of programming a robot to sense an object and then to control it accurately and precisely enough to form a secure grasp, is very difficult. Humans can grasp objects with seemingly little to no reasoning at all. For robots, however, the process is much more complex. There are many issues in sensing and perception, object tracking, control theory, and dynamic disturbance rejection which must all be solved sufficiently well before we can even begin to approach the grasping problem from a high level.

If a robot were able to reliably grasp an object of arbitrary shape, orientation, texture, and size, it would be a huge step forward in the field. There are many applications both in current industrial robotics as well as room for growth in the field of prosthetics and humanoid robotics that could benefit from such a robust grasp

algorithm. Robots on the assembly line would not be restricted to monotonous, repeatable tasks. Those who are disabled and are missing arms and hands could be tremendously helped with semi-autonomous prosthetics. It would be a great leap forward for the field, yet it must start with a mastery of certain fundamental principles.

Much research has been done and many methods exist nowadays for building precise, reliable, repeatable control systems for robotic arms. Even with such a core foundation, however, one will find that every arm is distinctly different. From the kinematic structure and dynamic equations to the internal mechanics of the arm, each controller must be uniquely designed to meet certain performance requirements. Disturbance rejection is very important and is still a well-studied research area today. Friction, gravity, and other dynamic terms must be sufficiently accounted for in any such control system, and this is made more difficult by the fact that many of the dynamic terms are highly non-linear and hard to estimate.

Once a controller has been designed, there is the problem of trajectory formation. Manipulators with many degrees of freedom (DOF) have more flexibility in building a trajectory in the six-dimensional space of position and orientation. However, with more degrees of freedom also comes more complexity, especially in the kinematic equations. In addition, trajectory formation of the arm must be synchronized with that of the hand. Imagine yourself gracefully sweeping through a water bottle to grasp it. For humans this seems like a simple task, but applying such a behavior to a robotic manipulator becomes far more difficult. Few researchers have been successful at constructing such graceful grasps for arbitrary objects in the workspace. There is a large deal of physics and mathematics underlying such a task, in addition to object tracking and computer vision.

Object tracking is also essential to the study and effectiveness of robotic grasping. A good model of the position, orientation, and geometry of the object about to be grasped are all necessary when determining and executing the most appropriate grasp. Hardware synchronization between cameras and the robot is crucial, in the timing of receiving data and outputting control signals. Latency is not well-tolerated, and high sample rates significantly boost performance. In order to attain

high accuracy, consistent coordinate frame transformations and alignment must take place between the camera system and the robotic arm.

Finally, in order to have a successful experimental setup for grasping, one must piece together all of the aforementioned details into a single cohesive system. Feedback from cameras and sensors should drive the arm through a carefully planned trajectory, and data should then be recorded for offline use. We develop a Grasp Acquisition Database (GAD) which can be used to store this data. After hundreds and even thousands of experiments have been done, one can begin a careful examination of this database through data mining. By loading the data into a dynamic simulation tool, such as dVC3d[2] for example, we will be able to make strong quantitative comparisons between simulation and experiments. This will open the door to strong future research in robotic grasping and control.

1.2 Previous Work

Research in robotic grasping is not a new area, but it is ongoing field since it has yet to be solved sufficiently well for unpredictable environments. One common approach for teaching robotic grasping is through demonstration. This is an open-ended method, as there are a variety of ways to demonstrate proper grasping. For example, in [3], a human tries to teach a robot certain grasp configurations for specific objects. The human teacher wears a glove which captures the pose of the hand, and this data is recorded. In [4], they study motion capture data in a similar way, with the intent of accurately animating and replaying human grasps. [5] attempts to establish a of model human grasping accurately by placing magnetic sensors on a glove. The models can then be mapped to various robotic hands, and the accuracy of these models are then demonstrated in simulation. Another approach is known as kinesthetic learning, where a human directs the robotic arm and hand into a known good grasping configuration. In [6], they attempt to do this while allowing for perturbations in the input data; this makes for more generalized yet less effective grasps.

Though our ultimate long-term goal is robotic grasping, this thesis only touches the surface of the previously mentioned techniques. We must be aware of these

though, so that we can use them as design criteria for establishing the experimental test bed. A starting point for building this test bed is the design of a robotic control system. As previously mentioned, control of robotic systems becomes especially difficult due to disturbance rejection, and many have studied this area. Gravity compensation is well-understood and easy to model if the system parameters are known, such as the masses, center-of-masses, and distances between the links. If these are not well-known, offline computation can be used to estimate them, and we will discuss such a method later. Friction compensation is significantly more difficult.

Friction in joints is highly non-linear in position, velocity, and even load. Many controllers neglect frictional effects and instead use large PID gains which indirectly compensate for this. Many times this gives satisfactory performance, and we will assess several options for friction compensation in Section 2.1.3. In the past, there have been many studies using both offline and online friction identification. [7] implements an adaptive compensation scheme by inserting a friction term into the dynamic equation using state feedback and online estimation. [8] points out that many controllers assume the wrong dynamic model for friction, and rely on knowing certain system parameters. It also outlines a similar approach for identifying friction parameters in a controller.

With disturbance rejection in place, there are also a variety of controllers that can be designed to drive the robot. A generic PID controller typically works well, though PID controllers can take many forms depending on how the system is modeled. Many use position and velocity as the input to drive torques and neglect most of the system dynamics, opting instead to use very stiff gains to overcome disturbances. Others use dynamic disturbance calculation such as Recursive Newton Euler (RNE); in this case, the output from the PID controller is usually a vector of accelerations which are fed to RNE. This tends to yield superior performance, yet system parameters must be well-known. [9] builds on top of this approach, but adds actuator fault detection to account for hardware failures or collisions with objects in the workspace. Finally, another variant is the impedance controller described in [10]. Impedance control has strong promise in the field of grasping since it controls

stiffness in the six-dimensional Cartesian space as opposed to in the joint space. The end-tip acts as a spring and damper system with configurable stiffness gains. In Section 2.2, we will discuss in detail two variations of PID control that were studied, as well as introduce impedance control.

Tracking the position and orientation of rigid bodies can be achieved in many ways. Some have had success using magnetic sensors positions throughout the object, and the readings give the relative position and orientation of these objects to one another, as mentioned in [5]. In [11], they discuss a method of Kalman filtering for accelerometer readings which can then be used to determine position and orientation information; however, the double integration of acceleration has be known to cause drift in the sensor readings. Perhaps the most accurate methods use techniques in computer vision and multiple cameras to identify rigid bodies either using silhouetting or infrared marker tracking.

A final discussion of previous work stems from current grasp analysis software. Two major features that are necessary in the experimental analysis of grasping include: 1) some way of storing experiment data for later retrieval; and 2) some way of analyzing the data in simulation. [12] describes the Columbia Grasp Database, which stores the geometry for several hands, as well as many objects and grasping configurations. It is openly available and can be queried by anyone. The database itself was created with a software tool described in [13] known as GraspIt! It also was developed at Columbia University for dynamic simulation of grasps and for grasp planning and analysis. It is our future goal to collaborate with Columbia University on these tools, but for our preliminary work we will be using our dVC3d simulator and a local grasp database with a similar design.

1.3 Preliminary Setup

A sufficient test bed for exploring robotic grasping requires several key components. Obviously certain hardware is required, including but not limited to a robotic manipulator, an end effector, and some means of tracking objects to be grasped in the workspace. Several commercial tracking systems are available with included computer vision APIs for marker and rigid body tracking. There are also

many kinds of robotic manipulators and end effectors available today, the discussion of which is outside the scope of this thesis. Instead, one particular configuration of hardware and software will be discussed as well as its advantages and limitations.

The Barrett Whole Arm Manipulator (WAM) is the device that we will be using to study grasping. It is a highly dexterous robotic arm commonly used in research laboratories to study grasping and control. It is known for its very low friction, low mass, and natural back-drivability. The Barrett WAM itself has 4 DOF. In addition, an optional wrist can be purchased which adds 3 DOF typically used for the roll, pitch, and yaw orientation of the end-tip. One such end-tip that can be used to study robotic grasping is the Barrett Hand (see Figure 1.1). This hand contains three fingers as well as a spread joint. The spread joint can be moved anywhere from 0 to 180 degrees in order to position two of the fingers laterally across from the final finger, which acts as a thumb. Each finger is controlled by one motor which synchronously closes the inner and outer links. The two fingers controlled by the spread joint have an additional degree of freedom; however, since there is only one spread motor, these two fingers are always spread at opposite angles about the approach vector of the hand[1]. Therefore, the hand has a total of 4 DOF.

Figure 1.1: The Barrett Hand demonstrating two fingers spread apart, with the thumb shown on the right side of the image.

In order to track objects, we will use the OptiTrack Motion Capture system and Tracking Tools software. This system comes with six cameras and supports

[1]Here and for the remainder of the thesis, we refer to the 'approach vector' as the vector originating at the center of the palm and directed distal to the base, perpendicular to the plane of the palm.

up to 24 cameras total. After the calibration process, the cameras determine their locations in three-dimensional space and then provide streaming capability at a rate of up to 100 Hz. This system will allow us to track rigid bodies in the experimental test bed and if calibrated properly, can produce sub-millimeter accuracy for object locations. A description of the API and tools that we will use can be found in Section 3.1.

Finally, for a software setup we choose to use MATLAB, Simulink, and xPC Target[2]. MATLAB is a powerful numerical tool which contains built-in procedures for common tasks in matrix math and linear algebra. It is easy to learn and intuitive to use for both Computer Scientists and Engineers, which is of crucial importance to making steady progress in a research laboratory. Simulink provides block diagram programming of systems, and allows for rapid prototyping new control loops. Blocks exist for continuous and discrete time systems, hardware drivers, and many more areas. This also allows one to program more complicated functions, such as a gravity compensation routine, for example. Once a model has been designed and implemented in Simulink, it can be downloaded onto a real-time operating system using the xPC Target toolbox and then executed on the hardware. Our main motivation behind using this software is the rapid prototyping design and ease of use, which will allow for various grasping techniques to be tested efficiently. More details and technical information are available in Appendix B.2.

[2]It is worth noting here that a Ph.D. student Matt Courtney and his advisor Greg Starr previously developed a Simulink model at the University of New Mexico to control the WAM arm (refer to [14]). In this thesis, we build off of their gravity compensation model to build more complex controllers specialized for accurate control and robotic grasping.

CHAPTER 2
CONTROL SYSTEM DESIGN

An accurate robotic controller for both an arm and hand is arguably the most important aspect in the development of a grasping test bed. In this chapter, we introduce the fundamental principles behind dynamic disturbance rejection and the development of joint space and Cartesian space PID controllers. Along the way, we discuss methods of parameter estimation for gravity and friction. In addition, we analyze certain features that performed poorly and those that performed well. Over the course of the chapter, we work to design an accurate robotic controller and then conclude by assessing its overall performance.

2.1 System Dynamics and Disturbance Rejection

A robotic controller relies on well-known system dynamics in order to overcome external disturbances and perform in graceful, fluid movements. There are several key disturbances that must be cancelled to achieve this, including gravity, friction, Coriolis, and centrifugal terms. We denote the joint position, velocity, and acceleration vectors as q, $\dot{q}$, and $\ddot{q}$ and the gravity, friction, and Coriolis/centrifugal terms as $G(q)$, $B(q, \dot{q})$, and $C(q, \dot{q})$, respectively. Assuming these are all perfectly cancelled and the system is continuous[3], the robot will be in a sort of free-floating mode with zero acceleration of all of the links. In order to allow the robot to freely move and accelerate, we add an additional inertia term, $M(q)$. Any torques contributing to the inertia term will cause an acceleration of the links. This gives us the following equation describing the system dynamics:

$$\tau = M(q)\ddot{q} + C(q, \dot{q}) + G(q) + B(q, \dot{q}) \tag{2.1}$$

When building a PID controller to compensate these dynamic terms, we have several options. If we know or can estimate the system parameters accurately, we can

[3] Discrete time systems will introduce lag and quantization effects, causing an imperfect cancellation of the dynamics terms.

use a method such as the Lagrange Dynamic Model or Recursive Newton Euler Formulation to compute them on every iteration of the control loop (as discussed in [15]). In addition, our PID controller can output either joint torques or joint accelerations that will be added on to the rest of the dynamic terms to overcome the inertia of the links. If we output torques, then the PID gains should be scaled as a function of the inertia[4] of the links; if we output accelerations, however, we will achieve the same result by subsequently multiplying by the link inertias as a function of position. In either case, a fairly accurate knowledge of the link inertias is necessary to achieve sufficient tracking ability and to reach the desired location with a small steady-state error. In the following sections, we will discuss in detail the aforementioned controller designs and how one can select the gains to achieve good system performance.

It is worth noting that the sample rate is of great importance to the performance of the system. Throughout this thesis, we run the arm at 500 Hz, though boosting this to 1000 Hz or even 1500 Hz could lead to a substantial improvement. Currently we have had issues with WAM pendant faults by attempting to push the sample rate any higher than 500 Hz even though we have verified the efficiency of our code at such a sample rate. We note this here for completeness, and provide more details about this and other technical issues in Appendix B.5.

2.1.1 Forward and Inverse Kinematics

Before a controller can be built, it is important to understand the kinematics of the system since they will be used in several important calculations. The forward kinematics describe the coordinate frame transforms from some base frame along the rigid links all the way out to the end-tip. This gives us a concise representation of tool position and orientation in world coordinates[5], and is a function of the current joint positions. The inverse kinematics perform the opposite operation. The inverse kinematics take as input a desired six-dimensional tool configuration and yield the

[4] The inertia is a function of position so when computing the gains, we can either use the maximum inertia or mean value of the inertia over the entire joint space.

[5] Here we use world coordinates and base coordinates interchangeably because we choose to have these two coordinate frames exactly coincident. See Appendix B.1.1 for coordinate frame definitions.

appropriate joint angles necessary to reach such a configuration.

The forward kinematic structure is commonly computed for a robotic manipulator using Denavit-Hartenberg frames. Four parameters, namely a, d, α, and q are used to describe the kinematic structure of a series of rigid links connected by either revolute or prismatic joints. A detailed discussion is well-described in robotics textbooks (such as [15]) and thus is outside the scope of this thesis. However, a table of Denavit-Hartenberg parameters and a figure depicting the Denavit-Hartenberg frames for the Barrett WAM can be found in Appendix B.1.1. For reference, we include here the homogeneous transformation matrix defined in [15]. This is denoted T_{k-1}^{k}, as it is used to map frame k coordinates into frame $k-1$:

$$T_{k-1}^{k} = \begin{bmatrix} cos(\theta_k) & -cos(\alpha_k)sin(\theta_k) & sin(\alpha_k)sin(\theta_k) & a_k cos(\theta_k) \\ sin(\theta_k) & cos(\alpha_k)cos(\theta_k) & -sin(\alpha_k)cos(\theta_k) & a_k sin(\theta_k) \\ 0 & sin(\alpha_k) & cos(\alpha_k) & d_k \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

The T matrices can be multiplied out along the kinematic chain in order to find the transformation matrix of the tool in base coordinates, T_{base}^{tool}. Extracting a closed-form inverse kinematic solution from T_{base}^{tool} (as in [15], [16]) is often difficult for complex manipulators such as the Barrett WAM. Some use numerical methods instead, such as the Jacobian (refer to [17]). We shall introduce a concise representation of state of the end-tip in Cartesian space, namely the 6x1 tool configuration vector, u, which can be used in conjunction with the Jacobian to form a Cartesian space trajectory (see Section 2.3.3). In this thesis, we will use a superscript to depict the frame that u is represented in. For example, if we extract u from the T_{base}^{tool} matrix, we use u^0 to say that the vector is represented in base (frame 0) coordinates. With a couple of exceptions described below, there will always be a one-to-one mapping between this vector and T_{base}^{tool}.

The first three elements of u^i represent the position of the tool, and the final three elements represent the tool orientation (both in the ith frame):

$$u^i = \begin{bmatrix} x & y & z & \phi_x & \phi_y & \phi_z \end{bmatrix}^T \tag{2.3}$$

where x, y, and z define the origin position of the coordinate frame placed at the end-tip in frame i coordinates, and ϕ_x, ϕ_y, and ϕ_z define the orientation of that coordinate frame in the x, y, and z dimensions of frame i. This is concisely represented using the axis-angle method defined in [18]. More specifically, for any rotation matrix R_i^j (the upper left 3x3 matrix of T_i^j), we can extract some axis of unit magnitude, $\hat{K}^i$, and a rotation about that axis, θ. In order to get from frame i to frame j, a mobile frame i' (initially coincident with i) can simply be rotated about $\hat{K}^i$ by the amount θ, where $\hat{K}^i$ is represented in the fixed frame i. The 3x1 orientation vector in u^i is then defined as $\hat{K}^i\theta$. In summary, if we have a transformation matrix T_i^j with p represented in frame i (equations from [18]):

$$T_i^j = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

then $\hat{K}^i$ and θ can be extracted as shown here:

$$\theta = acos\left(\frac{r_{11} + r_{22} + r_{33} - 1}{2}\right) \tag{2.5}$$

$$\hat{K}^i = \begin{bmatrix} k_x \\ k_y \\ k_z \end{bmatrix} = \frac{1}{2sin(\theta)} \begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix} \tag{2.6}$$

The tool configuration vector is constructed as follows:

$$u^i = \begin{bmatrix} p_x & p_y & p_z & \theta k_x & \theta k_y & \theta k_z \end{bmatrix}^T \tag{2.7}$$

The tool configuration vector will be an important aspect to creating a Cartesian space controller, and for that reason it is introduced here. Notice that throughout the thesis, we will also refer to ν^i, which represents the linear and angular

velocity of the tool frame relative to frame i. This vector is computed as:

$$\nu^i = \begin{bmatrix} \dot{p_x} & \dot{p_y} & \dot{p_z} & k_x\dot{\theta} & k_y\dot{\theta} & k_z\dot{\theta} \end{bmatrix}^T \tag{2.8}$$

where the 'dot' implies the time derivative. The angular portion can be thought of as rotating the tool at some rotational velocity, $\dot{\theta}$, about the constant unit vector, $\hat{K}^i$.

In addition, using the constraint that $\hat{K}^i$ is a unit vector, we can easily extract $\hat{K}^i$ and θ given the 3x1 orientation vector of u:

$$\theta = \pm\sqrt{(\phi_x)^2 + (\phi_y)^2 + (\phi_z)^2} \tag{2.9}$$

$$\hat{K}^i = \frac{1}{\theta}\begin{bmatrix} \phi_x \\ \phi_y \\ \phi_z \end{bmatrix} \tag{2.10}$$

As discussed in [18], axis-angle representations are not unique; in fact, two equivalent representations can be extracted from any rotation matrix, specifically $(\hat{K}^i, \theta)$ and $(-\hat{K}^i, -\theta)$. This explains the choice of square root in Equation 2.9. In addition, as is also mentioned in [18], the axis-angle method has singularities at 0 and 180 degrees. We will see in Section 2.3.3 that this causes minor issues when attempting to maintain constant end-tip orientation during a Cartesian space move.

Next we can reconstruct the rotation matrix R_i^j, assuming $\hat{K}^i$ represents the rotation of frame j in frame i. As in [18], we have:

$$R_i^j = \begin{bmatrix} k_x k_x v_\theta + c_\theta & k_x k_y v_\theta - k_z s_\theta & k_x k_z v_\theta + k_y s_\theta \\ k_x k_y v_\theta + k_z s_\theta & k_y k_y v_\theta + c_\theta & k_y k_z v_\theta - k_x s_\theta \\ k_x k_z v_\theta - k_y s_\theta & k_y k_z v_\theta + k_x s_\theta & k_z k_z v_\theta + c_\theta \end{bmatrix} \tag{2.11}$$

where $v_\theta = 1 - c_\theta$.

The Jacobian is a 6xn matrix that maps joint motions to end-tip movements in Cartesian space, where n is the number of joints. The first three rows of J can be constructed by taking the partial derivatives of the position portion (top three

rows) of u, with respect to each joint angle, q_i (for $i = 1 \ldots n$). The final three rows can be constructed quite easily as described in [18]. Assuming all joints are revolute (as they are in the Barrett WAM), this forms the Jacobian matrix, J:

$$J_{1:3} = \begin{bmatrix} \frac{\partial w_1}{\partial q_1} & \frac{\partial w_1}{\partial q_2} & \cdots & \frac{\partial w_1}{\partial q_n} \\ \frac{\partial w_2}{\partial q_1} & \frac{\partial w_2}{\partial q_2} & \cdots & \frac{\partial w_2}{\partial q_n} \\ \frac{\partial w_3}{\partial q_1} & \frac{\partial w_3}{\partial q_2} & \cdots & \frac{\partial w_3}{\partial q_n} \\ (R_0^0)_z & (R_0^1)_z & \cdots & (R_0^{n-1})_z \end{bmatrix}_{6xn} \tag{2.12}$$

The Jacobian will be used in Cartesian space and impedance controllers introduced later in the thesis. In particular, it can be used in the equation $\nu = J\dot{q}$ to determine the joint motions that will yield a desired Cartesian space trajectory. Next we discuss dynamics compensation, and revisit the Jacobian once a joint space controller has been built.

2.1.2 Gravity Compensation

A logical first step in the design of a controller is verification of model parameters. Manufacturers typically provide technical data (possibly in Computer Aided Design, or CAD, models) which includes the link masses, link center-of-masses, link inertias, arm geometry, and Denavit-Hartenberg parameters. A simple way of verifying these numbers qualitatively is by building a gravity compensation model and running it on the arm. If the arm supports itself after being moved to a range of several different configurations, one can be fairly confident that these model parameters are correct[6]. However, in the following section we introduce a more precise method of solving for these parameters and then assess the benefits and drawbacks of such an algorithm.

In order to accurately compensate the force due to gravity, one must begin by forming a set of equations to describe these interactions. One common method (as described in [15]) is to use the Lagrange-Euler Dynamic Model. Here, we construct four equations (for the 4 DOF of the arm), each corresponding to the gravity torque

[6] Notice that the link inertias are not needed for Gravity Compensation and cannot be computed in this way. In Section 3.2.4, we briefly discuss another method which could be used to verify link inertias and other parameters.

on one of the four joints. If we neglect the rest of the system dynamics, we have the output torque to all four joints as simply a function of joint position, $\tau = G(q)$. Since the Barrett WAM is back-drivable, when we implement and run this we can move the arm around freely, compensating friction and the rest of the dynamic terms with our own strength. In doing so, we have completed the process of qualitative verification of the model parameters.

Before we discuss a more precise method for model verification, we should become more familiar with the gravity equations previously described. We find that the gravity acting on joint 1 is always zero since the axis of rotation of this joint is perpendicular to the ground plane. The gravity equations for joints 2 through 4 all have a similar arrangement of terms shown as follows[7]:

$$\begin{aligned} G_1(q) &= 0 \\ G_2(q) &= C_1\lambda_1 + C_2\lambda_2 + C_3\lambda_3 + C_4\lambda_4 + C_5\lambda_5 + C_6\lambda_6 \\ G_3(q) &= C_1\beta_1 + C_2\beta_2 + C_3\beta_3 + C_4\beta_4 \\ G_4(q) &= C_1\gamma_1 + C_2\gamma_2 \end{aligned} \tag{2.13}$$

where the C_i are coefficients made up of the unknown parameters we are solving for, and $G_j(q)$ represents the gravity torque acting on joint j as a function of joint position q. Each λ_k, β_k, and γ_k is composed of sines and cosines of the four joint angles. One important thing to note is that each C_i is consistent across all joints (i.e. C_1 in the $G_2(q)$ equation is equivalent to C_1 in the $G_3(q)$ equation).

Using these equations, we can form a linear system of equations ($Ax = b$) solved simply by a linear least squares regression, $A^TAx = A^Tb$. The system of

[7] These equations are listed in full in Appendix B.1.3.

equations above can be written out in matrix form as:

$$\begin{bmatrix} \lambda_1 & \lambda_2 & \lambda_3 & \lambda_4 & \lambda_5 & \lambda_6 \\ \beta_1 & \beta_2 & \beta_3 & \beta_4 & 0 & 0 \\ \gamma_1 & \gamma_2 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{bmatrix} = \begin{bmatrix} G_2(q) \\ G_3(q) \\ G_4(q) \end{bmatrix} \tag{2.14}$$

The A matrix is formed by evaluating the sines and cosines at various joint positions, and the b matrix is known from the applied joint torques. The joint torques are applied using a PD gravity compensation controller as described in [15]. Since for every set of joint positions there are six unknown coefficients and only three equations, we must position the arm in at at least two different configurations to have a unique solution[8]. Placing the arm in more than two distinct configurations will typically cause the system to become over-constrained; in this case, the least-squares solution with minimal error will be computed.

Model parameters for our system were verified first using the qualitative approach, and then using the least squares estimation discussed previously. Target positions were chosen at random by a human controller using a graphical user interface. The results were promising, and the coefficients computed were similar to those of the manufacturer, but due to the randomness of joint configurations and the number of data points collected at each configuration, the computed coefficients varied slightly between runs. In addition, this method fails to take into account forces due to static friction in the joints. A better approach to solving this would be to incorporate the full system dynamics and to use a non-linear least squares estimation over a set of configurations covering the entire joint space. We demonstrate the effectiveness of this approach for a similar application in camera calibration in Section 3.2.3.2.

[8]The configurations should be chosen such that the A matrix is well-conditioned. If multiple rows in A are linearly dependent, then A will become ill-conditioned and more than two configurations will be necessary for a unique solution.

2.1.3 Friction Compensation

Friction is another major disturbance which must be compensated to achieve accurate control of a robotic system. Friction in joints is primarily velocity dependent, though it can also be position or even load dependent. A very comprehensive explanation of various friction models can be found in [19]. For now, we focus on static and velocity dependent friction, given by the following equation:

$$B(\dot{q}) = f_c sgn(\dot{q}) + f_v \dot{q} \tag{2.15}$$

where f_c is the coefficient of static friction, f_v is the coefficient of viscous friction, and sgn is the signum function, defined as:

$$sgn(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases} \tag{2.16}$$

Velocity dependent friction can be easily analyzed using basic laws of physics. When the forces on a rigid body are balanced but the body is still moving, it is said to be in dynamic equilibrium. By running several experiments which place the arm into a state of dynamic equilibrium, we will be able to estimate the viscous friction coefficient, f_v, for each joint. If we send just one joint in motion, the Coriolis term is negligible, and joint configurations can also be chosen such that the centrifugal term is near zero. If possible, the joint of interest should be positioned such that the gravity torque on it is also zero[9], and all other joints should be programmed for gravity compensation mode. Applying a constant torque to this single joint will then cause that joint to accelerate (assuming the torque is greater than the coefficient of static friction, f_c) until the friction torque opposing it is exactly equal to the applied torque. At this point, the arm will be in dynamic equilibrium, traveling at a constant velocity. We record a data point $(\tau, \dot{q})$ and repeat this procedure for several different torques, τ, forming a scatter plot. Finally, as shown in Figure 2.1, a linear least squares fit through the points gives a line whose slope is f_v for this

[9]This configuration is not always possible but is desired because discretization effects will always cause an imperfect cancellation of gravity, especially at low sample rates.

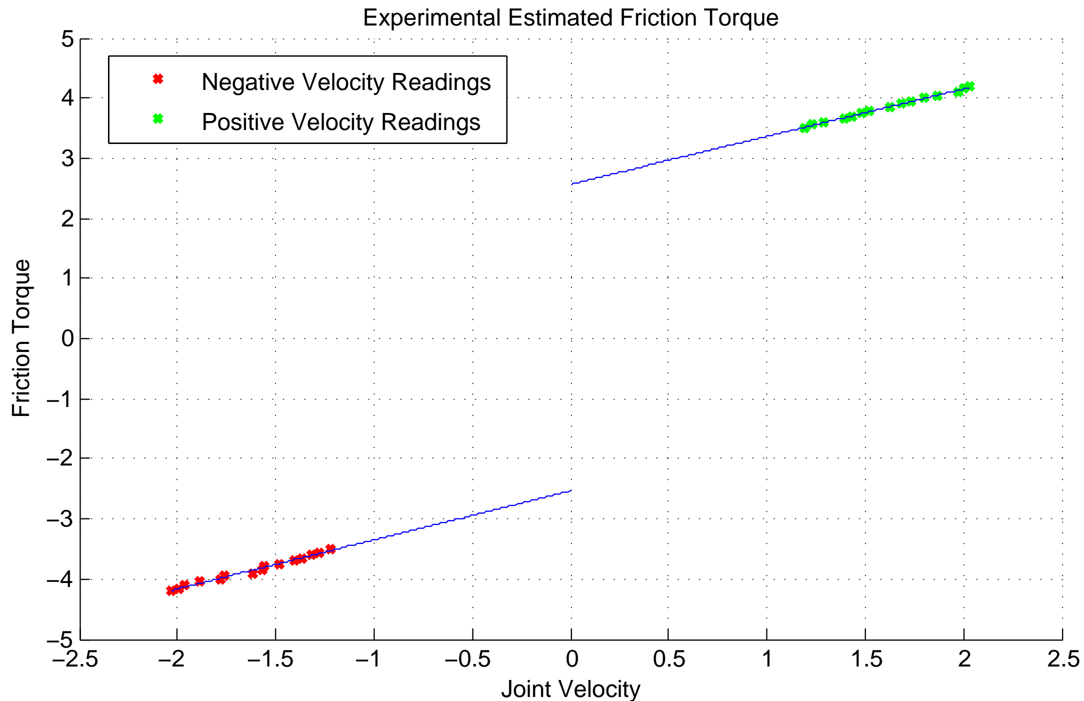
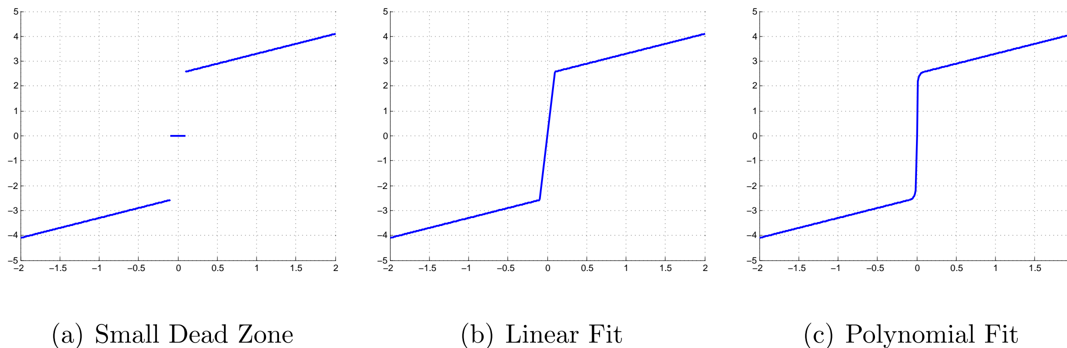


Figure 2.1: Viscous friction is calculated by applying constant torques to each joint and measuring the final velocity. A linear fit for both positive and negative velocities gives the line whose slope is f_v.

joint.

A crude approximation for the coefficient of static friction, f_c, can be found by applying gradually increasing torques until a joint is set into motion. By definition, the coefficient of static friction is the force at which a body transitions from a 'stick' to a 'slip' state, and the procedure described will give us a fair approximation to the actual value. However, actually incorporating the coefficient of static friction into the dynamic model for a robotic controller is much more difficult. If we attempt to model static and viscous friction very accurately, we will run into stability issues primarily due to poor velocity feedback (see Section 2.1.4 for more details). Vibration and oscillation are introduced by even the slightest imperfections in our friction estimation, and if static friction is included in the estimate, we find that performance is further degraded due to high torque compensation at very low velocities. For this reason, many researchers tend to neglect static friction and instead

interpolate between the positive and negative viscous friction plots (refer to [19]).



(a) Small Dead Zone (b) Linear Fit (c) Polynomial Fit

Figure 2.2: Several methods for connecting the two linear sections of a viscous friction plot in order to avoid joint oscillations and instability.

Connecting the positive and negative viscous portions of the friction graph is the final step in our friction estimation model. This can be done in several different ways, some of which are shown in Figure 2.2. In Figure 2.2(a), we add a small dead zone about zero to account for imperfections in our velocity signal. This is effective in removing vibrations but clearly fails to account for large static friction effects. In Figure 2.2(b) and Figure 2.2(c), linear and polynomial fits are respectively used to join the two ends of the viscous friction curve. We notice that the latter two methods do not remove as much instability as a simple dead zone, but nevertheless yield less vibration than if we were to include static friction in our estimate.

By building a gravity plus friction compensation model, we are able to qualitatively analyze the effectiveness of this approach. We found that a combined method using a linear blend and a small dead zone appeared to be most effective in cancelling friction. The dead zone spanned velocities with magnitude less than 0.1 rad/s, and the linear ramp spanned velocities with magnitude less than 0.5 rad/s. At sufficiently large velocities ($\dot{q} > 0.2$ rad/s), a human operator is able to move the joints smoothly, and with no added force the joint velocities are observed to be fairly constant, as expected.

Friction compensation is a very difficult area, and for that reason many research papers have been devoted to it. In this thesis, we have only touched the surface of various methods for friction compensation. While the friction on our par-

ticular WAM is unusually high and worth investigating, we find later that it is more effectively cancelled by a high-gain PID controller which does not introduce nearly as much instability into the system.

2.1.4 Velocity Feedback

Velocity feedback is also a crucial aspect toward achieving good system performance. If our velocity feedback is poor, then the damping term of our PID controller will perform errantly. Unfortunately, the WAM does not contain tachometers in its joints. This forces us to use the Backward Euler Method for estimating the joint velocities, namely the following:

$$\dot{q}(t) = \frac{q(t) - q(t-h)}{h} \tag{2.17}$$

where h is a small number. More specifically for our discrete case:

$$\dot{q}[k] = \frac{\Delta q}{\Delta t} = \frac{q[k] - q[k-1]}{\Delta t} \tag{2.18}$$

where Δt is approximately the sample time and k is the current step of the discrete time system. This method brings up issues of both numerical stability and a small lag in the computation. The computation of joint velocities tends to be numerically unstable for two reasons. The first reason is the quantization of joint positions fed back from the robot, which is inherent in the use of encoders. The second reason is the fact that we are dividing one very small quantized number (the change in joint position in radians) by another very small number (the sample time). This tends to cause a large amount of noise in the velocity computation. Finally, the estimated discrete derivative calculated by the Backward Euler Method effectively lags the actual derivative by one-half of the sample time. As shown in Figure 2.3, the velocity calculated at step k (time t) is actually a better estimate of the velocity at time $t - (\frac{\Delta t}{2})$, and not at time t. This lag is most pronounced in areas of large accelerations.

More advanced numerical methods exist for estimating the velocity; however, in the interest of time, those are left for future work. Instead we investigate the use

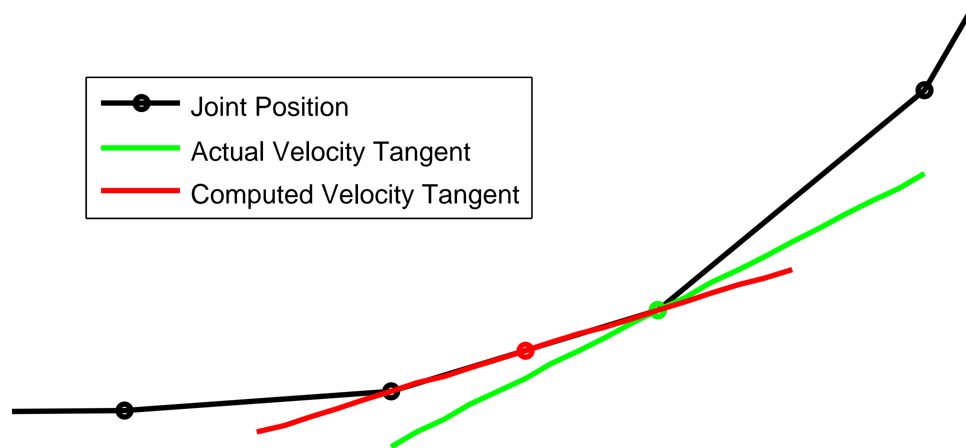


Figure 2.3: The Backward Euler Method for velocity estimation lags the true velocity by approximately one-half of the sample time. Whereas we are trying to find the tangent at time t, the computed tangent is more appropriate at time $t - \frac{\Delta t}{2}$.

of the Backward Euler Method along with a filtering method to attempt to deplete noise in the velocity readings. We experiment with two different types of filters, the simple moving average and the exponential moving average.

A moving average aids in smoothing out a noisy signal by taking the mean of the previous set of n readings, where n can be modified to meet performance requirements. It is inherently a low-pass filter and in addition introduces a lag, or delay, into the signal. Setting n to a large number significantly decreases the amount of noise, but also magnifies the effect of the low-pass filter and lag. Setting n small causes the signal to maintain its magnitude and to trail by a smaller amount, but at the same time, the signal will retain some noise. In this case, a generic moving average takes the following form:

$$\dot{q}_{generic}[k] = \sum_{i=0}^{n-1} w_i \dot{q}_{be}[k-i] \tag{2.19}$$

where w_i are the weights of individual samples, with the 0th sample being the most recent. The vector, $\dot{q}_{be}[k]$, is a set of velocity observations computed using the Backward Euler Method.

2.1.4.1 Simple Moving Average

For a simple moving average, all of the samples are given an equal weight of $w_i = \frac{1}{n}$. Expanded out, this looks like:

$$\dot{q}_{simple}[k] = \frac{1}{n}\Big\{\dot{q}_{be}[k] + \dot{q}_{be}[k-1] + \dot{q}_{be}[k-2] + \cdots + \dot{q}_{be}[k-(n-1)]\Big\} \quad (2.20)$$

Upon implementing a Simple Moving Average for the velocity readings, we see that this filter causes a strong vibration during movement of the joints. The vibration occurs even with $n = 2$ and becomes worse as n is increased, leading us to believe that too much lag is introduced into the signal by using this method. It is possible that increasing the sample rate could rectify this issue.

2.1.4.2 Exponential Moving Average

An exponential moving average is a variation of a moving average where the weights are not all equal to $\frac{1}{n}$. Instead they are distributed in an exponential fashion with values between 0 and 1, giving the most recent samples the highest weight. For this, we introduce a new parameter α which represents the rate of decline of the weights. The higher α is set, the more weight is given to the most recent samples (see Figure 2.4 for a sample plot of the weights). The exponential moving average tends to outperform the simple moving average by decreasing the amount of lag in the signal and by acting as less of a low-pass filter. The weights for an exponential moving average are computed as follows:

$$w_i = \begin{cases} \alpha(1-\alpha)^i & \text{for } 0 \leq i \leq (n-2) \\ (1-\alpha)^i & \text{for } i = (n-1) \end{cases} \quad (2.21)$$

When we expand the generic moving average with these new weights, we get:

$$\begin{aligned} \dot{q}_{exp}[k] = \alpha\Big\{\dot{q}_{be}[k] + (1-\alpha)\dot{q}_{be}[k-1] + (1-\alpha)^2\dot{q}_{be}[k-2] + \cdots \\ + (1-\alpha)^{n-2}\dot{q}_{be}[k-(n-2)]\Big\} + (1-\alpha)^{n-1}\dot{q}_{be}[k-(n-1)] \end{aligned} \quad (2.22)$$

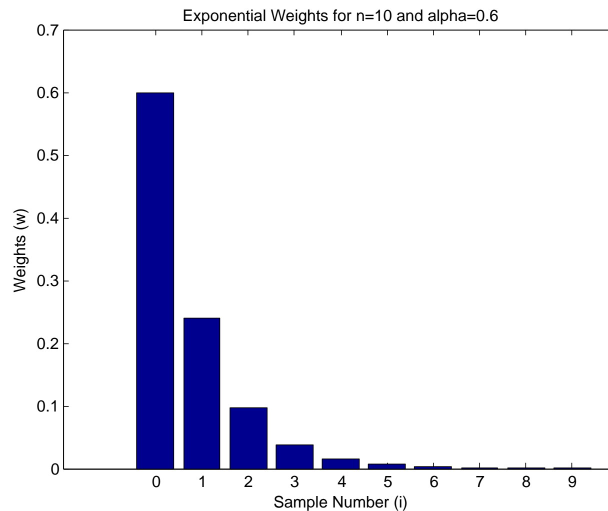


Figure 2.4: A plot of the weights for an Exponential Moving Average with $n = 15$ and $\alpha = 0.6$. The weights decline exponentially at a rate determined by α, giving the most weight to more recent samples.

This filter can also be efficiently computed in real-time. Notice that when computing in real-time, n essentially grows on each iteration as the total number of samples, and the weights are exponentially distributed over the entire runtime. However, the weights for the most distant samples (smallest i) become arbitrarily small and hardly contribute to the computation. An exponential moving average can be computed in real-time by using the following equation:

$$\dot{q}_{exp}[k] = \alpha\dot{q}_{be}[k] + (1 - \alpha)\dot{q}_{exp}[k - 1] \tag{2.23}$$

After implementing the real-time form of the Exponential Moving Average, we continue to experience the same vibration present in the Simple Moving Average. This vibration is also present when no filtering method is used though it is much less significant. A more advanced filter, such as the Kalman filter, is yet to be implemented and is left for future work. We expect that a Kalman filter could produce a much cleaner velocity signal by incorporating the system dynamics into the state estimation. For a complete list of future work, please refer to Section 4.1.

2.2 Survey of Controllers

Now that the fundamentals necessary for building a robotic controller have been introduced, we continue with a discussion of a joint space controller design. A robotic PID controller can be implemented in a variety of ways, differing mainly in the types of inputs and the way that the system dynamics are compensated. In the following sections we will discuss two such controllers, one which outputs joint accelerations and the other which outputs torques to be added on to the rest of the dynamic terms. In addition, we will introduce impedance control, which is currently being implemented by another student, but is a common and important controller for robotic grasping.

A generic PID controller is implemented in discrete time using the following equation with discrete integration and differentiation:

$$y[k] = K_p e[k] + K_d \dot{e}[k] + K_i \sum_{i=0}^{n-1} (e[k-i]\Delta t) \tag{2.24}$$

where y is the output signal, e is the error signal, $\dot{e}$ is the first derivative of the error signal, and K_p, K_d, and K_i are the proportional, derivative, and integral gains, respectively. In the case of the controller implemented on the WAM, we have the following:

$$e[k] = q[k] - q_{des}[k] \tag{2.25}$$

$$\dot{e}[k] = \dot{q}[k] - \dot{q}_{des}[k] \tag{2.26}$$

where q_{des} and $\dot{q}_{des}$ are the current desired position and velocity vectors. Notice that the velocity error signal can again be computed using the Backward Euler Method. A complete derivation for this can be found in Appendix A.1.

In this thesis, we implement both a PID Acceleration Controller and a PID Torque Controller, and the results and performance analysis will be described in the following sections.

2.2.1 PID Acceleration Controller

One variation of the generic PID controller is the PID Acceleration Controller. In this control method, the output from the PID controller is a vector of desired joint accelerations. Recall from Equation 2.1 that if we cancel all disturbances — gravity, friction, Coriolis, and centrifugal terms — we must only overcome the link inertias in order for the joints to accelerate. Therefore, if the output from the PID controller is a vector of joint accelerations, we will achieve the desired movement given the joint position and velocity error as input. If the current joint positions are far from the desired joint positions, for example, the accelerations output by the PID controller will be large. On the other hand, if we have nearly reached the target position and the joint velocities are greater than desired, the controller may output an opposing acceleration, decelerating the link to its target position and velocity.

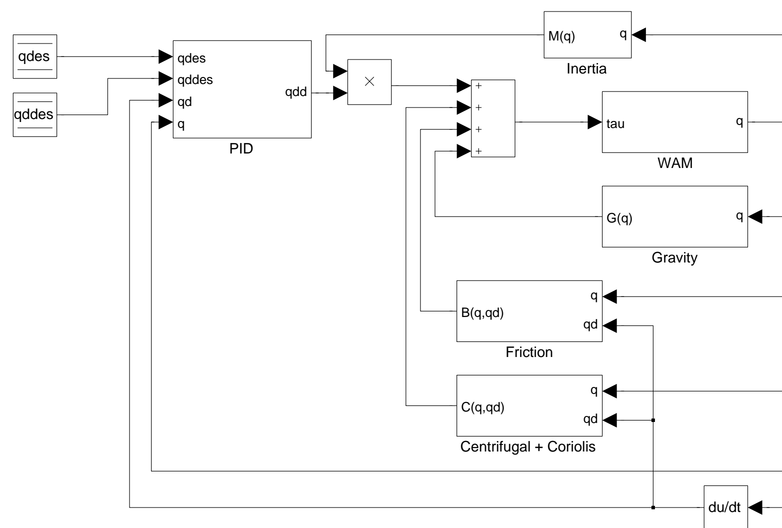


Figure 2.5: The complete system diagram for a PID Acceleration Controller. The current inertia of the links, $M(q)$, is computed on each time step and is incorporated into the system dynamics.

One method for computing the system dynamics commonly used in conjunction with this controller is the Recursive Newton Euler (RNE) formulation. This algorithm is described thoroughly in [15]; essentially, it uses forward iterations and then backward iterations to determine the torques acting on certain joints. Figure 2.5 shows the system diagram for a PID Acceleration Controller, where the

cancellation of dynamic terms (each shown separately in the figure) can all be computed using the RNE algorithm. Finally, notice also in the figure that the joint accelerations are multiplied by link inertias[10] computed on every cycle of the control loop. This multiplication produces joint torques necessary to accelerate the links, provided that the rest of the system dynamics have been accurately compensated.

The PID Acceleration Controller is a very accurate method used for robotic control, but it will behave poorly if the model parameters are incorrect or if the system dynamics have been improperly compensated. While boosting the PID gains should maintain stiffness in the joints throughout the motion, doing so is unnatural for compensating the non-linear effects of other dynamics terms. We find that due to uncertainties in inertia parameters and highly non-linear frictional effects, this controller is not well-suited for our purposes. Instead, we achieve better success with PID Torque Controller which does not rely on knowledge of the link inertias. For the remainder of this thesis, we will use and analyze the PID Torque Controller.

2.2.2 PID Torque Controller

A second variation of the generic PID controller is a PID Torque Controller. This is similar to PID Acceleration Controller in that the inputs, system dynamics, and desired behavior are all the same. However, in this case the controller outputs a vector of joint torques rather than joint accelerations. The joint torques which are produced correspond to the $M(q)\ddot{q}$ term of the system dynamics, but rather than computing the link inertias on every cycle of the control loop, we instead select gains proportional to the expected amount of inertia in the links. Notice that if the link inertias are unknown, these gains can be selected manually as in Section 2.4.1. If, however, the inertia parameters are known to be accurate, we can compute the maximum inertia that the joints will encounter and use this to select the PID gains (see Section 2.4.2 for more details).

If the link inertias are well-known, the PID Acceleration Controller should tend to outperform this controller, since $M(q)$ can be computed in real-time and included in the calculation. Nonetheless, the implementation of a PID Torque Controller

[10] The link inertias are provided in CAD diagrams along with other technical data, as described in Section 2.1.

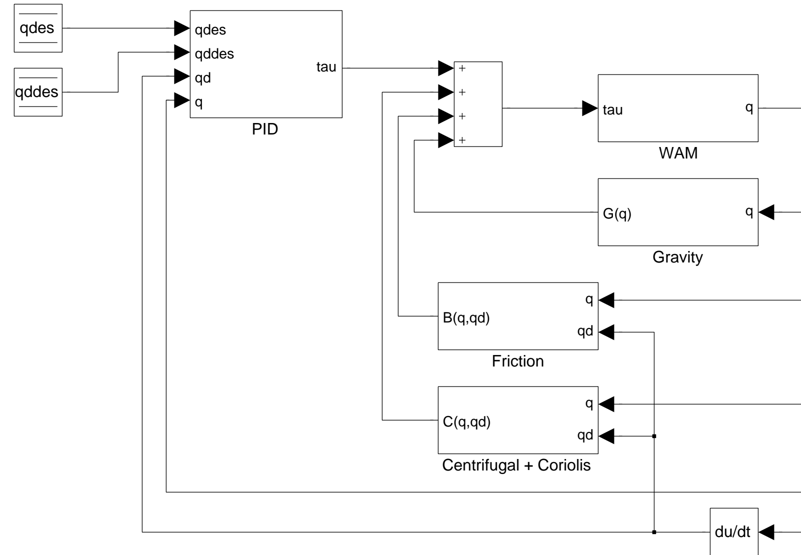


Figure 2.6: The complete system diagram for a PID Torque Controller. The gains can be selected manually, or based on the maximum inertia, $M_{max}(q)$, over the entire joint space.

tends to be more straightforward and we find that the results are substantially better, possibly due to improper cancellation of friction and inertia terms by the PID Acceleration Controller. If desired, we are again able to use the RNE method for cancelling disturbances. In this case, however, the acceleration vector which is fed to RNE is the set of all zeros[11], and the output of the PID controller is summed with the output of the RNE function before being sent to the WAM.

2.2.3 Impedance Control

We discuss a third and final controller, impedance control, due to its widespread acceptance in the field of robotic grasping. Impedance control is simply a form of Cartesian Space PD control, allowing the end tip to act as a spring and damper system in all six dimensions of position and orientation. It is powerful in the sense that stiffness is no longer controlled at the joint level, but at the end-tip instead. This allows for graceful interactions with the environment, either in unexpected collisions or in robotic grasps.

[11] Alternatively, we could just remove the inertia calculation from the RNE procedure, since we are computing that portion of the torque in the PID block instead.

Impedance control utilizes the Jacobian, J, and the tool configuration vector, u (both defined in Section 2.1.1). It is used for converting moments in Cartesian space to corresponding torques in joint space; in this way, it controls the stiffness of the tool by applying torques at the motors. The equation describing this relationship is shown here, as in [15]:

$$\tau = J^T(K_p e + K_d \dot{e}) \tag{2.27}$$

where τ is the torque produced by the impedance controller, e and $\dot{e}$ are the position and velocity error of the tool configuration vector, and K_p and K_d are as defined previously. In this case, we have a simple PD controller with no integral term, and the proportional and derivative gains are defined in Cartesian space, not in the joint space. The output of the PD controller (shown in parentheses in Equation 2.27) is the desired force at the end-tip in Cartesian space.

Impedance control will be an important aspect of robotic grasping experiments in the future, and for that reason this section is included for reference. As previously stated, another student is currently implementing an impedance controller and the completion of this controller will be left for future work.

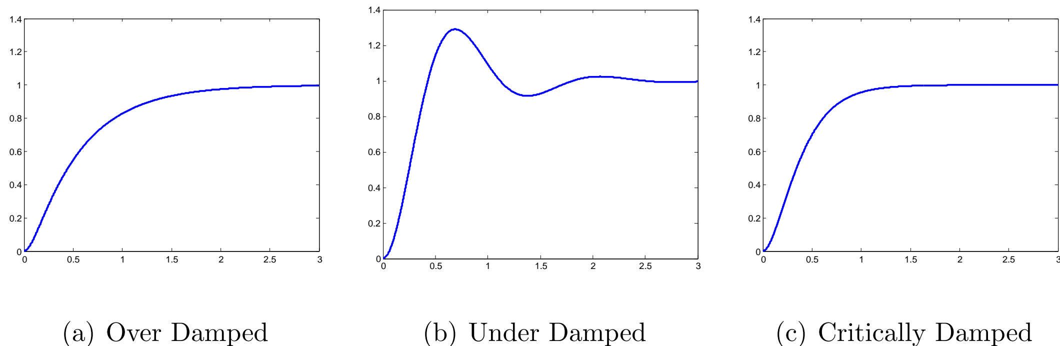
2.3 Trajectory Formation

As stated in the previous section, we find that a PID Torque Controller is best suited to our performance requirements for robotic grasping. Now that we have studied the effectiveness of several different controllers and methods for dynamic disturbance rejection, we move forward with this particular controller for the remainder of the thesis. More specifically, the Backward Euler Method will be used for joint velocity feedback and gravity compensation will be used in addition to the PID Torque Controller. For now, we omit friction compensation due to stability issues, and we also neglect the Coriolis and centrifugal terms since these will be fairly negligible at low joint velocities.

For satisfactory tracking performance, it is crucial to form a joint space trajectory through which the joints can move smoothly. In the following sections, we discuss the steps leading up to trajectory generation and the weakness of initial approaches to this problem.

2.3.1 Step Response

In linear system theory, the step response is a major criterion for assessing system performance. A step response begins with feeding a 'step' input of unit magnitude to the controller. In doing so, one can analyze the output behavior which, for a generic PID controller, tends to fall into one of several categories: over damped, under damped, or critically damped (see Figure 2.7 for sample plots of these responses). In robotic control, a critically damped system is most desired because the system will reach its target position in an optimal amount of time without any overshoot. The settling time is also of critical importance, and refers to the amount of time the controller takes to reach a resting state at the target position, generally with a small steady-state error.



(a) Over Damped (b) Under Damped (c) Critically Damped

Figure 2.7: Various step responses for linear PID systems. Altering the gains can produce drastically different response curves, but critical damping tends to be most desired in robotics and many other applications.

Throughout our initial work on the PID Torque Controller, we begin by feeding large step responses over only one control cycle. For example, if joint 1 of the arm is currently positioned at 0 radians and the desired joint position is 1 radian, we simply feed as input to the controller q_{des}=1 rad and $\dot{q}_{des}$=0 rad/s on the next time step. Of course, we would like the joint to be at 1 radian with a zero velocity, and in fact this is a perfectly valid step input. However, this approach has several drawbacks. First, since the position error has the potential to be very large, we are restricted to setting the proportional gain K_p quite low. If we set K_p too high, the output torque produced by the proportional term of the PID controller will be

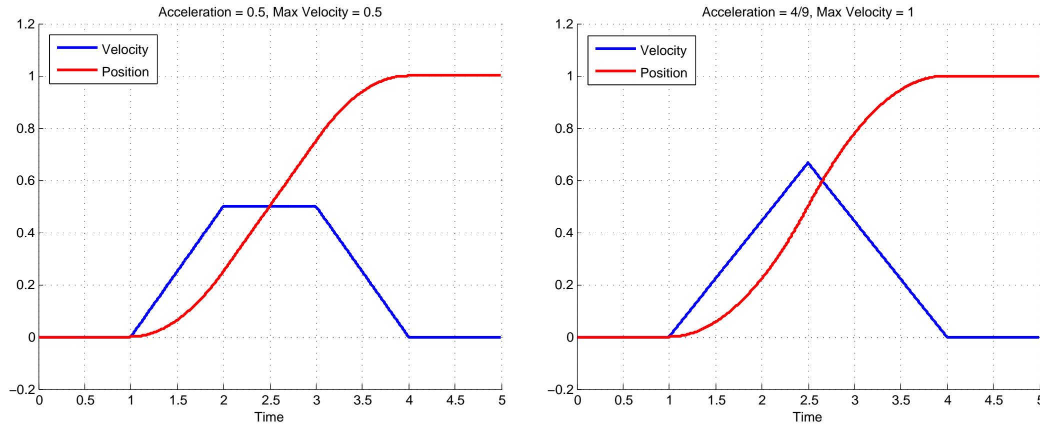
excessively large, and the safety pendants for the WAM will produce a fault. One could argue instead that there should be a saturation limit on the output torque, and as long as the position error is large, the output torque will be constant and limited by the saturation value. However, this too is a poor solution because it will introduce a non-linearity into the control loop that could otherwise be avoided. In addition, a low proportional gain leads to a very weak (non-stiff) response, especially for small position errors in the presence of uncompensated frictional effects. Finally, the combination of uncompensated dynamics terms along with small gains tends to lead to large steady state errors. Typically we would increase the integral gain to account for this, but with such large step inputs the integral term will accumulate quickly and lead to system instability.

Clearly, our initial approach will not yield the level of accuracy needed to perform grasping experiments. In the next section, we discuss one very effective and accurate approach to solving this problem, which is to form a joint space trajectory using a trapezoidal profile.

2.3.2 Trapezoidal Profile

One way of rectifying the issues of the previous section is to decompose the large step input into many smaller steps, where each smaller step typically takes place over one control cycle. This is a common form of trajectory generation, and it allows for drastically better accuracy and tracking performance. The advantages here are threefold. First, because the step sizes are significantly smaller we can boost the gains greatly, yielding more stiffness and accuracy in the joints throughout the motion. Second, because q tracks the input curve q_{des} with a very small error, we eliminate the accumulation problem of the integrator term. This yields improved system stability and allows us to boost the integral gain, K_i, to remove steady-state errors. Finally, with the smaller time steps we have much greater control over precisely what path should be followed in order to reach the target position. One such path is known as the trapezoidal profile.

The trapezoidal profile is an intuitive and graceful way of implementing this joint space trajectory, and consists of a period of acceleration, constant velocity,



(a) Constant Velocity Region

(b) No Constant Velocity Region

Figure 2.8: The trapezoidal profile gets its name from the shape of the velocity graph shown on the left. However, if the acceleration is too low or maximum velocity is too high, the profile will not reach its maximum velocity.

and then deceleration into the target position. This profile gets its name because of the shape of the velocity curve that it forms, as shown in Figure 2.8. Notice that in Figure 2.8(a), there exists a period of constant velocity in the curve, whereas this region is not present in Figure 2.8(b). The reason for this discrepancy comes from the values of acceleration and maximum velocity used to form the profile. In the former case, the acceleration was large enough to increase the joint velocity to its maximum value, which then remains constant so as to avoid exceeding this threshold. In the latter case, the acceleration was not large enough to increase the joint velocity to its maximum value, so instead we only have a region of acceleration and deceleration, each consuming one-half of the trajectory.

Each step of the trapezoidal profile can be computed using basic kinematic equations, and a full derivation the equations for this can be found in Appendix A.2. Implementation of this trajectory completes the joint space PID controller. However, building trajectories in Cartesian space is also vitally important to any experimental grasping setup. In the next section, we will see that the tool configuration Jacobian allows us to implement Cartesian space control on top of the joint space controller.

2.3.3 Cartesian Space Control

Now that we have developed a joint space controller, it is fairly trivial to construct a Cartesian space controller on top of this. Essentially we will form a desired trajectory in Cartesian space and then convert this into its corresponding joint space trajectory in one of two ways. The first and most straightforward solution is to use a closed-form inverse kinematic solution, but as we have seen this is not always possible. The second solution is to construct an approximation using the Jacobian. This approach is commonly used and works well, but it has several drawbacks which we will investigate further. In either case, implementing Cartesian space control in this way differs from impedance control in one major respect. While both methods are capable of forming Cartesian space trajectories, impedance control implements stiffness in Cartesian space, whereas the approach outlined next implements stiffness in the joint space.

In the Cartesian space controller designed here, Cartesian space commands are simply converted into joint space commands, and the stiffness gains are the same as those for the joint space controller (i.e. joint 3 has some stiffness). In an impedance controller, torques are clearly still applied to the motors, but these torques are instead computed using gains in the six-dimensional Cartesian space (i.e. the z-axis direction has some stiffness). This is a subtle but important distinction to point out.

Closed-form inverse kinematics are highly valuable in a robotic controller, though it is not always possible to devise such a solution. Recall that the six-dimensional tool configuration vector described in Section 2.1.1 defines the position and orientation of the tool frame in base coordinates. Since the desired tool configuration vector is known for an arbitrary via point, it can sometimes be inverted in closed-form to give the necessary joint positions to reach any Cartesian space configuration. In the case of high degree of freedom manipulators, however, the forward kinematic structure is often too complex to be inverted, and this is indeed the case with the 4 DOF Barrett WAM.

A common alternative to inverse kinematics is to use a Jacobian approximation. If we imagine a trajectory through the Cartesian space, we can picture this

as a six-dimensional arc through which the end tip will pass. Clearly this arc is a non-linear, continuous function composed primarily of sines, cosines, and robot parameters. In order to reach the final configuration, u^0_{final}, from the starting configuration, u^0_{start}, we first break the trajectory into smaller steps using the trapezoidal profile defined in Section 2.3.2, and then at each step we form a linear approximation to the trajectory using the Jacobian. The linear approximation is essentially tangent to the path through Cartesian space, and if the robot tracks such a path (avoiding singularities) it will end at approximately the final configuration, u^0_{final}. Worth noting is that the smaller the time steps, the more precise this approximation becomes[12], and this is yet another performance gain which can be had by boosting the sample rate.

Suppose the controller is fed a high level Cartesian command such as, 'move the tool to some Cartesian space state vector'. The state vector that is fed to the controller is u^0_{final}, at which point the controller is responsible for determining what joint motions will cause such a Cartesian space motion. The controller is also fed the positional accelerations and maximum velocities in x, y, and z, and the rotational acceleration and maximum velocity about the orientation axis. As introduced in Section 2.1.1, these are denoted $\ddot{\theta}$ and $\dot{\theta}$, respectively. At each subsequent time step, the trapezoidal profile generator computes the next desired Cartesian space position and velocity. The ν vector is formed using Equation 2.8. The positional difference is computed simply by subtracting the previous position in base coordinates from the current desired position in base coordinates. The rotational difference is computed as the axis-angle representation from the current rotation matrix (considered the zero configuration) to the next desired rotation matrix.

The only case in which this solution deteriorates is if the axis-angle representation is at a singularity. This occurs when we attempt to maintain a constant (or near constant) rotation of the tool frame. On each step, the desired rotation matrix will be the identity matrix, since we do not wish to change orientation. The computed angle in the axis-angle representation is zero, but the axis is ill-defined; the

[12]The approximation will become more and more accurate until it the time step is decreased below the precision of the computer; however, in a robotic controller, it is very unlikely that the sample times will ever be that small.

axis could in fact be any unit vector on the sphere of solutions and would still yield the correct solution (since the angle of rotation about the axis is zero). To solve for this case, we currently check the computed angle in the axis-angle representation, and if it is less than $1e^{-3}$ rad, we assume that the amount of rotation is negligible with respect to the accuracy of the controller. An arbitrary rotation axis is selected, the rotation angle is set to zero, and the tool maintains a constant orientation. A more robust method for dealing with this situation could be implemented using unit quaternions, as described in [20].

Given a Cartesian space trajectory, we can form its corresponding joint space trajectory by executing the following equations, which linearly approximate the path at each time step. To compute the joint space velocity vector for the next time step, $\dot{q}[k+1]$, given a desired next step Cartesian space velocity vector, $\nu[k+1]$, we have:

$$\dot{q}[k+1] = J^{-1}\nu[k+1] \tag{2.28}$$

where J is a function of the current joint position vector, $q[k]$. Since the system is discrete, we can also use this calculated joint space velocity to determine the distance the joints should travel over the next time step. This gives us the desired joint space position, q, as follows:

$$q[k+1] = \dot{q}[k+1]\Delta t \tag{2.29}$$

where Δt is approximately the sample time.

However, these equations only work when the robotic manipulator has at least 6 DOF and when J is invertible (i.e. the robot is not at a singularity). At a singularity, the manipulator loses at least one degree of freedom and therefore cannot control all six of the Cartesian space parameters. Since our WAM only has 4 DOF and we are attempting to control six, we must use a slight variation of this method. Our Jacobian matrix has dimension 6x4 and clearly not invertible since it is not square. Therefore, we have two main options to choose from. The first is to find the minimum error solution for controlling all 6 DOF. The second is to explicitly specify which of the 6 DOF we would like to control at each time step. In either case, both

of these options can be implemented using the Moore-Penrose pseudoinverse. For the first option, we have:

$$\dot{q}[k+1] = J^{+}\nu[k+1] \tag{2.30}$$

keeping our original Jacobian matrix, and simply using the pseudoinverse as opposed to the standard inverse operator. The latter option can be implemented in one additional step by first removing the two rows that we do not wish to control from the Jacobian matrix, J, and tool velocity vector, ν. This decreases the Jacobian matrix to a 4x4 square matrix, which will have full rank assuming the manipulator is not positioned at a singularity. Next, Equation 2.30 can be applied in this lower dimensional space. When the manipulator is not at a singularity, this approximation will be fairly accurate; however, if it passes through a singular configuration and drops ranks, the pseudoinverse will find the minimal error solution that closely satisfies the desired trajectory. In the Cartesian space controller implemented in this thesis, it is required that the three orientation parameters be controlled (a common requirement in grasping), and one additional position parameter can also be selected for control.

2.4 Selecting PID Gains

The final step in the development of a joint space controller is selecting the gains, K_p, K_d, and K_i. Of course with all of the gains set to zero, the PID controller outputs zero torques, and the only torques applied to the motors come from the rest of our dynamics compensation. However, carefully choosing gains can significantly impact system performance. In this section, we discuss two methods for gain selection. The first is a very basic method of increasing gains in a particular order until the desired step response is achieved. The second is a more precise method based on link inertias. For this case, however, the system parameters must be well-known and the system dynamics must be accurately compensated.

In this exploration of gain selection, our focus will be on three primary characteristics discussed previously in Section 2.3.1: the response time, the settling time, and a critically damped control behavior.

2.4.1 Manual Selection

The first and most simple method for gain selection is achieved by evaluating the system response as we gradually increase the gains. Each joint should be isolated, starting at the joint farthest from the base and moving inward. The number of gains we will find is generally $3n$ since we have K_p, K_d, and K_i for each of the n joints. In the case of the 4 DOF WAM, we will thus have $12 = 3(4)$ gains. This method of manual selection is commonly used and is included here for reference. Please see [21] for more details.

The first step is to slowly increase K_p while feeding a unit step as input, until the joint begins to oscillate with some constant amplitude. The proportional gain is used to control the stiffness of the controller and if set too high can cause oscillations and instability. At this point, the controller has only proportional gain and is said to be 'marginally stable' since it is oscillating about the target position without increasing or decreasing in amplitude over time. The next step is to use $K_p/2$ as the new proportional gain, and then to increase the integral gain K_i slowly (starting at zero) until the settling time is acceptable without introducing instability. The integral gain is most useful in decreasing steady state error and settling time. Finally, the damping gain, K_d, should be increased to prevent overshoot and achieve critical damping. If set too high, the damping gain can also cause vibrations and instability during joint motion.

Manually selecting the gains works well in the presence of non-linearities and uncompensated dynamic terms where a more systematic approach is not as applicable. Prior to using this approach, we explored the possibility of using inertia parameters to appropriately select the gains; however, manual gains selection proved to be more effective due to unknown friction and inertia parameters. In the interest of time, we use the recommended PID gains provided by Barrett, but in the next section we discuss a method of computing gains based on the maximum inertia of all the joints.

2.4.2 Using Inertia Parameters

Another solution which is quite effective when the system parameters are well-known is to select the gains based on the maximum inertia encountered by the joints. Since the output of the PID controller is a vector of torques that will accelerate the links, it makes sense that these torques should be proportional to the link inertias that must be overcome, as shown by the $M\ddot{q}$ term in the dynamics equation. Assuming in the ideal case that gravity, friction, Coriolis, and centrifugal terms are all perfectly cancelled and that there are no external disturbances, our PID control loop resembles that of Figure 2.9. Notice that the figure implies a continuous time system with PD control (no integral term). In actuality, the system is discrete and could just as easily be represented in discrete time, but for simplicity here we use a continuous time analysis[13]. Also, the integral term is theoretically not need if all other system dynamics are perfectly compensated, so it is not included in the diagram.

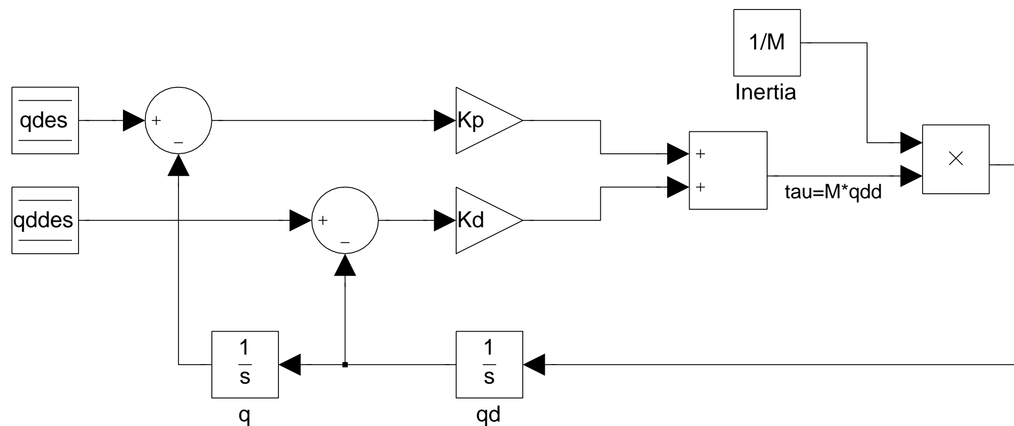


Figure 2.9: The PD control loop compensating only the inertia term demonstrates that gains can be selected accurately using the link inertia values, given that the rest of the dynamics are perfectly cancelled.

Having this system diagram in place, we can now solve for the closed-loop transfer function by setting $\dot{q}_{des}$ to zero and then analytically solve for the gains

[13] At high sample rates, discrete time systems are very similar in behavior to continuous time systems, so this is a reasonable comparison.

based on the maximum inertia values. A full derivation of the following equations can be found in Appendix A.3, and we include the result below:

$$\frac{\ddot{q}}{q_{des}} = \frac{\frac{K_p}{M}s^2}{s^2 + \frac{K_d}{M}s + \frac{K_p}{M}} \tag{2.31}$$

Next, we set the denominator equal to $s^2 + 2\zeta\omega_n s + \omega_n^2$ to perform a second order analysis in the continuous frequency domain. As explained in [22], ζ is known as the damping ratio and ω_n is the undamped natural frequency. After setting $\zeta = 1$ for critical damping, simple algebra shows that:

$$K_p = M\omega_n^2 \tag{2.32}$$

$$K_d = \sqrt{4MK_p} \tag{2.33}$$

We now have eight equations (Equations 2.32 and 2.33 applied to all four joints) and twelve unknowns, where the unknowns are K_p, K_d, and ω_n for each of the four joints. However, we enforce the additional constraint that the undamped natural frequency, ω_n, should be equal for all of the joints. This reduces the system to only nine unknowns; choosing any one of these arbitrarily will allow us to solve for the remaining eight. For example, we could set K_p for joint 1 to some desired value and solve for the rest of the unknowns. We are missing one vital piece of information necessary to solve these equations, however, which is the maximum inertia values M acting on each joint. Next we introduce a process for determining these inertia values by using a similarity transformation followed by the Parallel Axis Theorem. More details on this process can be found in [23] and [24].

As previously explained, the manufacturer typically provides link inertia data for each joint derived from CAD models. More specifically, the data provided is a set of 3x3 inertia matrices describing the dispersion of mass throughout each link, and centered at the origin of the its corresponding coordinate frame. We will denote the provided matrices M_i^i, to say that this is the inertia of frame i (subscript) represented in frame i (superscript). Also recall from the Denavit-Hartenberg representation in Section 2.1.1 that link k rotates about the z-axis of link $k-1$. In order to

determine the maximum inertia acting on each joint, we must first determine the joint configuration which will produce the maximum inertia and then transform these 3x3 link inertia matrices into the appropriate coordinate frame in order to calculate that inertia. As in [18], the inertia matrices are defined as follows:

$$M = \begin{bmatrix} M_{xx} & M_{xy} & M_{xz} \\ M_{yx} & M_{yy} & M_{yz} \\ M_{zx} & M_{zy} & M_{zz} \end{bmatrix} \tag{2.34}$$

Note, for example, that M_{xy} represents the moment of inertia about the y-axis when the link is rotated about the x-axis. This means that once we have consolidated the maximum inertia for each link into a single 3x3 matrix represented at the rotational frame of that joint, we can simply extract M_{zz}. This is the element we are interested in, since the axis of rotation defined by the Denavit-Hartenberg frames is the z-axis, and the inertia felt by the joint while trying to accelerate is also about the z-axis.

We begin by focusing on the problem transforming the inertia of link i from its own frame i into frame j. This can be done by applying a rotation followed by a translation. For the rotational part, we use a similarity transformation to rotate the body into frame j. Next, the Parallel Axis Theorem is used to shift the inertia of link i positionally from frame i into frame j. Both of these transformations can be combined into one equation, with the left half being the similarity transformation and the right half being Parallel Axis Theorem, as shown here:

$$M_i^j = R_j^i M_i^i (R_j^i)^T + m_i \begin{bmatrix} y^2+z^2 & xy & xz \\ xy & x^2+z^2 & yz \\ xz & yz & x^2+y^2 \end{bmatrix} \tag{2.35}$$

Notice that R_j^i is the rotation matrix mapping the orientation of frame i into frame j. The parameters x, y, and z compose the position vector from the origin of frame j to the origin of frame i represented in frame j coordinates[14]. At this point, we have a method for transforming a link inertia matrix from its own frame, i, to some

[14]This position vector can be extracted from the first three rows in the fourth column of the T_j^i matrix.

arbitrary other frame, j. Using Equation 2.35, the final step is to compute the maximum inertias acting on each joint.

The configuration for which a joint j will face its maximum inertia can be determined easily by inspection. One this configuration is known, all of the necessary coordinate frame transformations, T, can be computed, and the inertia matrices for all of the distal links can be brought into the rotational frame and summed. We sum the link inertias from link j through $n+1$, where the $(n+1)^{th}$ link represents the hand, if present. This gives the final equation used to calculate the maximum inertia of frame j, which we denote M_{tot}^{j-1} since the distal links as a whole rotate about the z-axis of frame $j-1$:

$$M_{tot}^{j-1} = \sum_{i=j}^{n+1} M_i^{j-1} \tag{2.36}$$

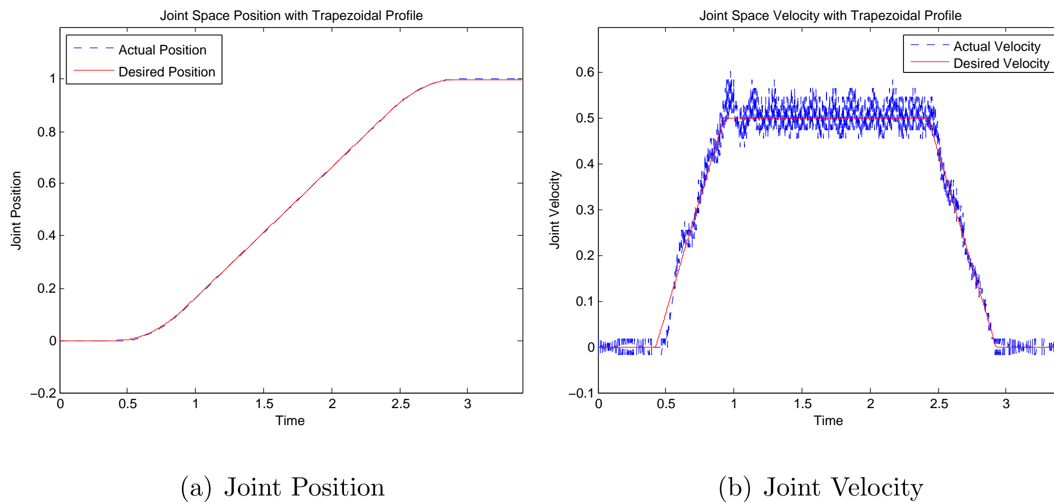
By combining the maximum inertia equations with the closed-loop controller equations, we are able to compute reasonable gains for a PD controller that should theoretically work, given near perfect disturbance rejection. However, as we have seen throughout this thesis, achieving near perfect disturbance rejection is very difficult in a robotic controller. A lot of effort was put into the method presented in this section, and for that reason it is included for reference. However, in the final controller we instead use manual gain selection as explained in Section 2.4.1.

2.5 Performance Analysis

In this section, we assess the performance of the finalized PID torque controller with trapezoidal trajectory formation, in both joint space and Cartesian space. In particular, we analyze the step response on isolated joints, focusing mostly on the tracking performance, settling time, and steady-state error. As explained in previous sections, the joint space PID controller is the basis for moving the arm not only in joint space but also through Cartesian space. Since its performance is vital to the success of the overall arm movement, we begin with two sample joint space responses and then conclude with a Cartesian space movement.

As the first sample joint space response, we feed joint 1 a unit step input,

starting at 0 rad and ending at 1 rad. The acceleration is set to 1 rad/s^2, and the maximum velocity is set to 0.5 rad/s. From a high level, we regard this as a step input; however, in reality the trapezoidal profile generator breaks this into a series of many smaller step inputs. Figure 2.10 shows the actual and desired position and velocity curves for this move. The mean and maximum tracking error for position are 0.0011 rad and 0.0053 rad, respectively, and for velocity are 0.0203 rad/s and 0.1027 rad/s. Overall, the position tracks quite well due to the high proportional gain. However, the velocity plot is very noisy as a result of having no tachometers on the joints and instead relying on the Backward Euler Method for discrete differentiation. While the desired velocity is maintained fairly well, the noise in the signal causes slight instability and vibration in the system. This was described in Section 2.1.4, and more advanced filtering methods are left for future work.



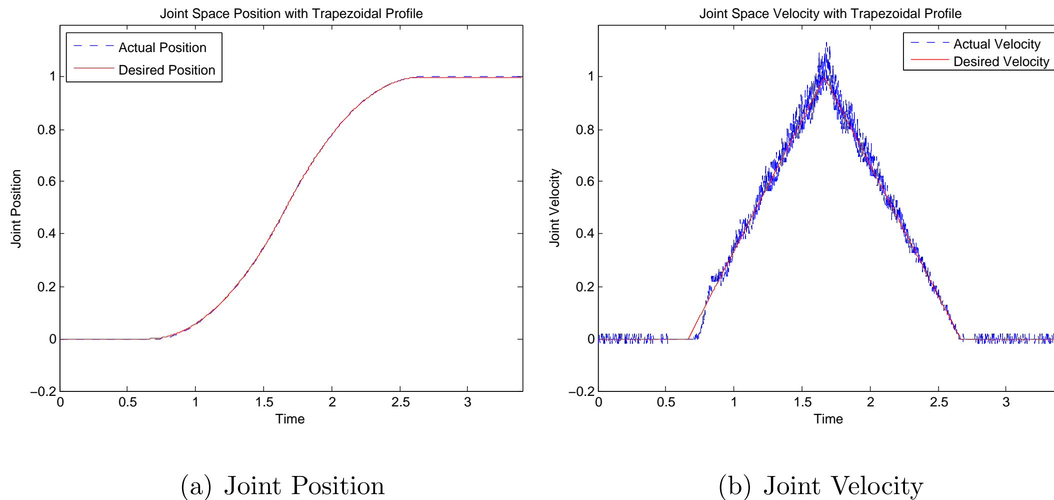
(a) Joint Position

(b) Joint Velocity

Figure 2.10: The system response is shown for a joint space command with target position of 1 rad and final velocity 0 rad/s. The acceleration is set to 1 rad/s^2, and the maximum velocity is set to 0.5 rad/s.

The second sample joint space response is quite similar to the first, but is included to demonstrate the trapezoidal profile formation when the joint space move does not reach its maximum velocity. Recall that in this case, the velocity plot is more so triangular than trapezoidal, as shown in Figure 2.11. Here, the only difference is that we set the maximum velocity to 1 rad/s rather than 0.5 rad/s. For

the unit step input that is fed to the controller, it happens to reach a velocity of 1 rad/s halfway through the trajectory, but then immediately starts decelerating in order to reach the target position without overshoot. In this case, either increasing the acceleration or decreasing the maximum velocity would introduce a region of constant velocity. The performance of this joint space move is similar to the last, with a mean and maximum position error of 0.0012 rad and 0.0054 rad, and a mean and maximum velocity error of 0.0218 rad/s and 0.1683 rad/s, respectively.

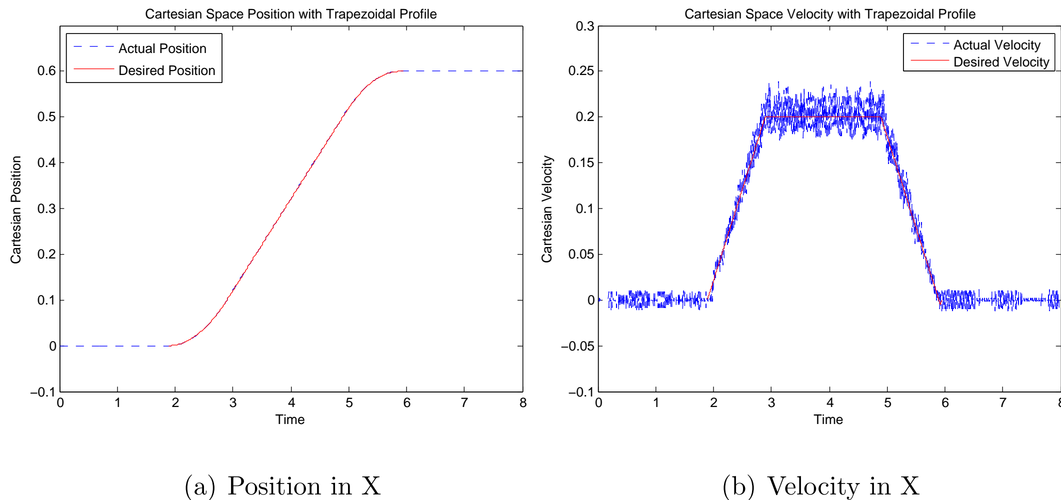


(a) Joint Position

(b) Joint Velocity

Figure 2.11: The system response is shown for a joint space command with target position of 1 rad and final velocity 0 rad/s. The acceleration is set to 1 rad/s^2, and the maximum velocity is set to 1 rad/s.

As a final example, we include a Cartesian space move for analysis. Recall that discretization effects due to the linear estimation of the Jacobian can introduce a slight drift as we evaluate a Cartesian space move, though this behavior improves at higher sample rates. We will analyze in particular the final position of the manipulator as compared to the target position. We begin by positioning the end-tip at an x-position of 0 m, and then attempt to move the end-tip on a straight line trajectory in x, maintaining a constant tool orientation. The desired x-position is 0.6 m, and the trajectory is known not to pass through singularities. In addition, the desired acceleration in x is 0.2 rad/s^2, and the desired maximum velocity in x is 0.2 rad/s. The choice of $\dot{\theta}$ and $\ddot{\theta}$ in this case is arbitrary since rotation of the tool is



(a) Position in X

(b) Velocity in X

Figure 2.12: The system response is shown for a Cartesian space straight line command with target x-position of 0 m and final velocity 0 m/s. The acceleration is set to 0.2 m/s^2, and the maximum velocity is set to 0.2 m/s.

not desired. After computing the Jacobian approximations at 500 Hz, the controller settles at the final desired x-position of 0.6000, which is quite accurate due to the sample rate and relatively slow velocity. The tool maintains its rotation to within the axis tolerance of $1e^{-3}$ rad. Finally, the end-tip overshoots by 0.3 mm and settles to within 0.2 mm in approximately 1.51 seconds. The position and velocity along x for this Cartesian space move are shown in Figure 2.12.

2.6 Hand Controller

Now that we have developed a robotic controller for the 4 DOF WAM, we shift focus to a controller for the Barrett Hand. Of course, robotic grasping relies on fluid motion of both the arm and hand, and accurate control of the hand is of utmost importance. Ideally, we would like the hand and arm to be controlled separately while at the same time moving in a coordinated fashion towards an accurate grasp. This requires that the controller has access to feedback data from both the arm and hand in real-time, and we discuss next how this can be done.

The Barrett Hand has two main control modes, real-time mode and supervisory control mode. In real-time mode, the control loop takes place in the software.

Just as we developed the arm controller using MATLAB and Simulink, we could develop a control law for the hand also implemented in software, and could use many of the same principles developed earlier in this chapter. Feedback from the controller gives joint positions, velocities, and strain[15], though in this case the output sent to the hand is a vector of desired velocities rather than torques. Internally, the hand uses a proportional gain on the joint velocities to output motor torques. In real-time mode, we receive feedback at a rate of approximately 30-33 Hz, much slower than the rate of the arm controller due to differences in CPU power and the transfer protocol used (serial versus CAN).

The difference with supervisory control mode is that the control loop takes place internally on a software chip in the hand. Rather than feeding desired velocities in real-time, high-level position commands are sent to the hand, which then uses a trapezoidal profile and some control law to output motor torques. While the fundamental principles are the same, Barrett has implemented this mode in their own software, and we are solely responsible to feeding commands to it. The drawback of using this mode is that feedback data can only be received once the joints have reached their final position or encountered an obstacle in the workspace.

In this thesis, we develop a large portion of the real-time controller for the hand, but it is unfinished and is left for future work. For now, we experiment using the supervisory control mode without real-time joint feedback, but in the future it will be vitally important to have a complete implementation of the real-time controller so that we can accurately analyze feedback from the hand. Technical details regarding high-level control commands for our supervisory mode controller can be found in Appendix B.4.3.

[15]This data that was not available in the controller from the arm. The fingers are not backdrivable, and we have access to strain gauge data for three out of the four joints in the hand.

CHAPTER 3
CAMERA AND DATABASE DESIGN

With a robotic controller in place for the WAM and Hand, our focus now shifts to a second fundamental topic of grasping. Of course, in order for a robot to grasp effectively, there must also be a means of locating objects in the workspace. In this chapter, we develop a system for combining image and tracking data with the high level controllers designed previously. We will introduce a method for accurately aligning and transforming coordinate frames, as a means of converting rigid body position and orientation into the base frame of the WAM. Finally, we will introduce the Grasp Acquisition Database and discuss how it can be used to aggregate large amounts of data from experiments.

3.1 Software Design

As mentioned previously, the controller for the arm and hand runs in a MATLAB model using Simulink, xPC Target, and Real-Time Workshop. These three toolboxes collectively allow for rapid prototyping of control systems using block diagrams. The block diagrams themselves are designed in Simulink, which is essentially a simulation environment for controls and signal processing. The Simulink models can be compiled into binary files using Real-Time Workshop, which then run on what is known as an xPC Target machine to control the actual hardware.

While block diagram programming provides an environment for rapidly testing new system designs, it is not generic enough to easily run grasping experiments. For this reason, we implement a suite of high level utilities in MATLAB code that interact with the xPC target machine as it runs the low level control code for the arm and hand. Essentially, the Simulink model executes everything discussed in Chapter 2, including gravity and dynamics compensation, the trapezoidal profile generator, PID control, and low-level hardware drivers for CAN and serial communication. The MATLAB suite then sends high-level joint space and Cartesian space commands to the controller over UDP. In addition, a modular environment

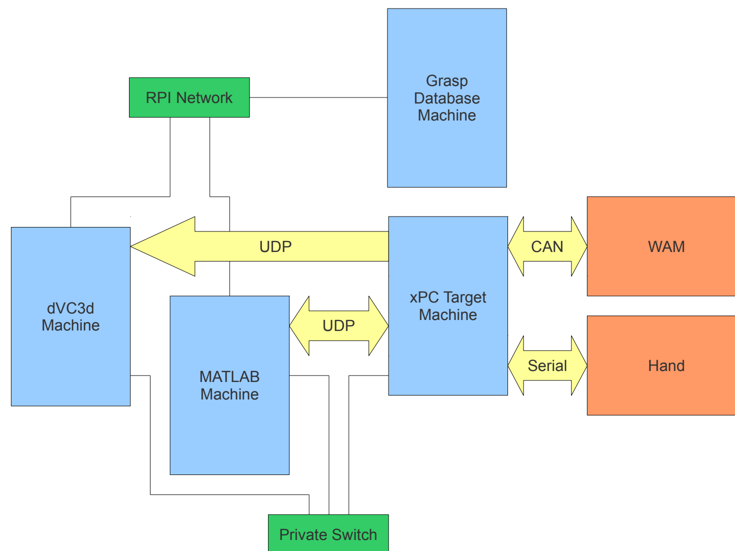


Figure 3.1: A system diagram showing the layout of how various machines are connected. UDP communication between the MATLAB and xPC Target machines drives experiments, which can then be stored on the Grasp Database.

for running experiments is implemented in the MATLAB code, which allows for capturing robot and tracking data in real-time. The data is processed and passed to a user-defined callback function, which then controls the arm throughout the experiment. After the experiment has been run, the suite also contains code for animating the recorded data and exporting it to the grasp database, if desired. Finally, experiments run previously can also be imported back into MATLAB for further animation or analysis.

In the next several sections, we discuss more details of the software and hardware setup, and then transition into the mathematics behind tracking system transformations. At the conclusion of this chapter, implementation of the experimental grasping test bed will be complete.

3.1.1 UDP Communication with Arm and Hand

Since the suite of MATLAB utilities is divided into a low-level controller and high-level experiment code, some means of communication between these two components is crucial. Of course, we would like to deliver high-level control commands

to the robot, but these commands will typically take several seconds to execute and thus high level commands need not be issued at 500 Hz. Communication in the opposite direction is also necessary. The high-level experiment code must have feedback from the robot in order to determine the current state and deliver control. In this case, it is desirable to have low-latency, rapid feedback so that the high-level callback function can respond quickly to events. Ideally, we should receive feedback from the controller on every control cycle.

As a lightweight option for satisfying these two objectives, we choose to use UDP communication over a dedicated Ethernet switch. On the side of sending control commands, this tends to work quite well, since packets are very rarely dropped. This then provides for a fairly reliable, connectionless communication protocol. If dropped packets become an issue in the future, connection-oriented TCP could be implemented as a more reliable alternative. On the receiving end, using UDP is almost required. As feedback is sent at 500 Hz, near real-time, low-latency delivery of packets is very important. This is vital not only to the execution of the experiment, but also to the accuracy of the recorded data. TCP could be used here, but as we will see the MATLAB experiment suite tends to drop about four out of five feedback packets, as each cycle takes close to 10 ms[16]. If TCP were used, the streaming data would be placed in a queue, causing a bottleneck.

The WAM data that is recorded includes the joint positions, velocities, and applied torques, as well as the Cartesian space state vector. The desired joint space or Cartesian space vectors are also recorded, if applicable. A detailed description of the MATLAB struct containing this data is available in Appendix B.3, and Appendix B.4 includes the format of the UDP packets.

3.1.2 Interfacing with Tracking System

The next important piece of the experimental grasping test bed incorporates the OptiTrack camera system introduced earlier and pictured in Figure 3.2. Since the camera system includes a software API[17] for marker-based rigid body tracking

[16]The inefficiency here is primarily caused by calls to the tracking system API and the lack of multi-threading capability in MATLAB.

[17]This piece of software was optionally purchased with the tracking system, and is known as OptiTrack Tracking Tools.

(refer to [25]), it would be quite beneficial for the experimental software suite to make calls to this API as a means of tracking objects to be grasped in the workspace. In fact, we are able to load the dynamic link library (DLL) directly from MATLAB, and we can use its utilities in the rest of the MATLAB experiment code.

Figure 3.2: The experimental grasping test bed, showing both the Barrett Arm and Hand as well as the six-camera overhead tracking system by OptiTrack.

The Tracking Tools software provides three major capabilities which we are interested in: camera calibration, marker detection, and rigid body tracking. The camera calibration procedure is explained in detail in Section 3.2.2, but it essentially allows the system to detect the location and orientation of all six cameras in the workspace. This is important not only because it gives the experimenter the ability to strategically place the cameras for grasping experiments, but it also allows the camera system to extract 3D marker locations from its raw images. The markers are easily located by the system, as they reflect infrared light transmitted by the cameras.

Calls to the API provide us access to 3D marker locations relative to the cameras base coordinate frame; however, the API somewhat surprisingly does not support individual marker tracking between frames. Instead, it provides the functionality for creating 'trackable' objects which can be identified and tracked. A

trackable object is created by placing three or more markers on some rigid body and allowing the camera system to record their relative distances apart. The API then identifies the 3D trackable position in camera base coordinates and the rotation relative to the starting orientation for each object. The markers must be positioned on the object such that they are not collinear; otherwise, the orientation of the object becomes ill-defined. The cameras are capable of a refresh rate of approximately 100 Hz.

We are able to incorporate this with the MATLAB grasp code in the following way. When the experiment starts, the user is prompted to activate the arm. Control in the Simulink model automatically sends a joint space command to the home position of the arm. As this is happening, UDP packets are transmitted at 500 Hz from the xPC Target machine to the MATLAB grasp code. The grasp code is blocking until it receives a UDP packet, at which point it requests data from the Tracking Tools API. As explained in the following sections, the camera data is then processed so that it is relative to the base frame of the WAM. Finally, a user-defined callback function is sent all of the robot and tracking data, which then drives the experiment and controls what data should be recorded. The process then repeats for the next UDP packet until the experiment is complete. The user deactivates the arm, and the data is recorded for further analysis.

This system provides a very modular, easy-to-use interface for the experimenter. However, since MATLAB does not support multi-threading, the program must be implemented in a single thread. While the UDP packets arrive at approximately 500 Hz, several are dropped while the camera system is processing image data at only 100 Hz. Most experiments record an average of about twenty percent of the robot data, yielding an overall capture rate of the slowest component, at about 100 Hz.

3.2 Camera Calibration

Camera calibration is a crucial aspect to achieving adequate tracking performance. There are two major steps that must be taken to calibrate the system. First, the Tracking Tools API provides a calibration procedure for identifying the camera

positions and orientations. If the calibration procedure is well-done, the system is capable of providing sub-millimeter tracking accuracy for rigid bodies. This metric is highly desirable and tends to improve by adding more cameras and carefully placing them in the workspace. This calibration procedure will be described in more detail in Section 3.2.2, and once complete, rigid body and marker locations returned by the API are quite accurate. Nevertheless, one additional step must be taken.

In order to easily interpret the tracking data, a second calibration procedure is run by the MATLAB grasp code. If we were to use the data provided directly by the API, it would be in terms of a left-handed coordinate frame defined by the tracking system. Since all of the robot data is relative to the right-handed base coordinate system of the WAM, we would ideally like for the tracking data to be in a consistent frame. In this way, the user callback function itself will not be required to perform a fairly complicated conversion each time it receives a set of data. Essentially, the homogeneous transformation matrix is found at the start of an experiment, and then marker and rigid body data is pre-processed using this transformation before being sent off to the callback function.

3.2.1 Transforming Handedness of Coordinate Frames

Before introducing the calibration procedure in more detail, we discuss the principles between right and left handed coordinate frame transformations. This will be important when we determine the transformation from the camera coordinate frame to the robot coordinate frame in Section 3.2.3. If a coordinate frame is represented as left-handed, it can be easily transformed to a right-handed system by reversing any of the primary axes, x, y, or z. In addition, any arbitrary rotation matrix preserves handedness between frames; that is, a rotation in some right-handed frame and an equivalent rotation in some left-handed frame will both yield the same rotation matrix[18].

For the transformations throughout the rest of this chapter, we choose to reverse the z-axis as a means of transforming left-handed frames to right-handed

[18] If we think of this in terms of Euler angles, using the right-handed angle convention in a right-handed frame is equivalent to using the left-handed angle convention in a left-handed frame, and will produce the same rotation matrix in both cases.

ones. For example, suppose a marker location $(2, 1, 3)$ is returned from the API. Since this is returned in terms of the left-handed camera frame, we simply define a new coordinate frame whose origin, x-axis, and y-axis are the same, but whose z-axis points in the opposite direction. Therefore, the marker location in this right-handed frame is $(2, 1, -3)$. The same principle is valid for transforming the position of a rigid body, and a method for transforming rigid body orientation is presented in Section 3.3.2.

3.2.2 Tracking Tools Calibration Procedure

The Tracking Tools API includes a calibration procedure for determining the position and orientation of all six cameras. In order to do this, it appears to use an iterative estimation process for determining camera parameters, but the details of this are obscure and unimportant. Instead, we are more interested in the method for running this calibration process and the final result which it yields.

Before running experiments, the Tracking Tools software should be run so that the calibration and preliminary setup can take place. The user marks the start of the calibration process using the software and then waves a 'wand' through the workspace, which contains markers of a known distance apart. Once a sufficient amount of data is recorded, the software will compute the result and set an arbitrary base frame for the cameras. The base frame can be translated or rotated as desired, but this is not necessary as we will transform all of the tracking data again anyway using the method in Section 3.2.3. Finally, trackable objects can be defined in the software and will be loaded from a saved project file in the MATLAB code.

3.2.3 Alignment of Coordinate Frames

One of the most crucial aspects to interfacing with the Tracking Tools software is transforming its data into a form that we are more familiar with. More specifically, we would like to translate and rotate marker locations and rigid bodies into the base frame of the robot. This will allow the arm to easily determine the relationship between its end-tip and the object to be grasped in the workspace. Ultimately, in this portion of the calibration process we would like to compute some homogeneous transformation matrix, T_{base}^{camera}, which precisely maps camera coordinates to robot

base coordinates. We can not realistically expect for this transformation to be more accurate than the camera system itself, but we will strive to achieve approximately the same accuracy as the cameras[19].

There are several obvious approaches to this problem, but most fail to produce sufficient accuracy. Recall that the Tracking Tools software allows the user to set the base frame as desired. Therefore, in theory one could construct a bracket with markers placed sturdily on the WAM, and then simply record the position of those points using the camera system. Since the points are then known in the camera frame and could be manually estimated in the base frame of the robot, the user could compute the transformation between the two frames. However, this approach is quite prone to human error and we find that the mean error in the estimation is about 5 mm.

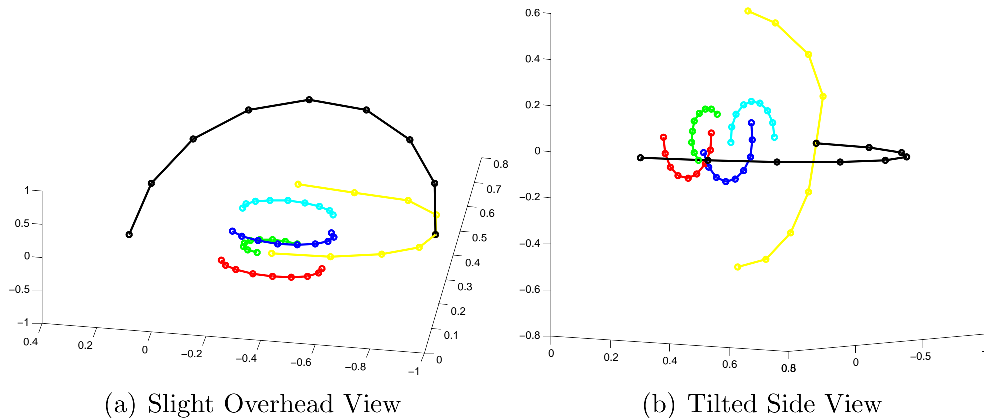
An alternative, yet more complicated approach uses a non-linear least squares estimation to determine the transformation. As we will see, this approach yields far greater accuracy and is the subject of the remainder of this section. The basic idea is the same as in the aforementioned process; however, we will remove the aspect of human error and add several other modifications which will make it far more precise.

3.2.3.1 Arc Formation about Base Frame

In order to accurately determine the coordinate frame transformation, we would like to select a set of joint configurations that will produce distinct and varied marker locations. Suppose we place several markers on link 1 of the robot and rotate joint 1. As defined by the Denavit-Hartenberg frames (see Appendix B.1.1), joint 1 rotates about z_0, the z-axis of the base frame. During the rotation, the marker locations are fixed in the first frame, but are changing in the base frame as defined by the transformation matrix T_0^1, which is a function of the first joint position, q_1. As q_1 rotates, the markers rigidly attached to link 1 'draw' arcs in 3D space, each forming a plane which is perpendicular to z_0. If joint 1 was rotated 360 degrees, the resulting set of arcs would appear as a set of platters stacked on top of one another.

[19] Recall that the camera system has the potential for sub-millimeter accuracy, and we would like to achieve this same accuracy after applying the transformation.

Each platter is a circle whose center is along the axis z_0, but all platters do not necessarily have the same radius (depending on where the markers are place on link 1). This would clearly define the z-axis of the base frame.



(a) Slight Overhead View (b) Tilted Side View

Figure 3.3: The marker locations, as captured by the camera system, are shown from two different perspectives. Arcs are formed through parallel planes revolving about z_0, and a final arc (black) comes up and over the robot to provide depth information.

Now suppose that as joint 1 is rotating, we record the marker locations from the camera system. The same set of platters described previously would be recorded, but this time the data would be in terms of the camera frame. If we were to plot the camera data, the platters would distinctly appear, but since they are in terms of the camera frame, they would be slightly tilted and translated relative to the camera frame definition. This translation and tilt (orientation) nearly defines the transformation from the camera frame to the base frame, with one exception. The data we have collected up to this point fully defines the orientation of the base frame relative to the camera frame, as well as the location of the origin along x_0 and y_0; however, the data does not give any indication of where the origin of the base frame is along z_0. In fact, an infinite number of translations along z_0 would satisfy the data. This last degree of freedom is unconstrained due to the fact that all of the platters form parallel planes, effectively restricting the translational data to two dimensions.

In order to capture the amount of translation of the origin along z_0, we must

form at least one additional arc in a plane not parallel to the others. To do this, we place one more marker on link 3 of the robot. By the same principle above, this marker is fixed in frame 3, and as joint 2 is rotated, the marker 'draws' an arc through a plane actually perpendicular to all the others. The matrix T_0^3 transforms the marker location into robot base coordinates, as a function of q_1, q_2, and q_3. This complete set of arcs is then shown from two different perspectives in Figures 3.3(a) and 3.3(b).

Notice in Figure 3.3 that we capture the marker locations for a discrete set of configurations along 180 degrees of rotation, so that the results are not weighted more heavily in any one dimension. More specifically, four markers attached to link 1 and one marker attached to link 3 are captured every $\frac{\pi}{8}$ rad while rotating joint 1 from $-\frac{\pi}{2}$ rad to $\frac{\pi}{2}$ rad (nine total configurations). For the final arc, the marker attached to link 3 is again captured at nine different configurations, specifically every $\frac{\pi}{8}$ degrees as joint 2 moves from $-\frac{\pi}{2}$ rad to $\frac{\pi}{2}$ rad. The markers attached to link 1 are not moving as joint 2 rotates, and so for this, we only capture the marker on link 3 to avoid redundant data.

As a final note, we require that the cameras be positioned such that every marker is visible at every configuration. Recall that the Tracking Tools API does not include functionality for tracking individual markers, so for this we use the shortest distance between marker locations over each frame as a primitive way of determining which marker is which. This is quite important for the non-linear least squares estimate as we will see in the next section.

3.2.3.2 Non-Linear Least Squares Estimation

We now introduce a method for estimating the transformation matrix, T_{base}^{camera}, given the data collected during the calibration process. The input to the function consists of a set of points for each configuration and also the joint positions recorded at each configuration. We will denote the position in the camera frame of the ith marker in the jth configuration as p_{ij}, and the position of the ith marker relative to its fixed frame as s_i:

$$p_{ij} = \begin{bmatrix} x_{ij} & y_{ij} & z_{ij} \end{bmatrix}^T \tag{3.1}$$

$$s_i = \begin{bmatrix} x_i & y_i & z_i \end{bmatrix}^T \tag{3.2}$$

In addition, we denote the vector of joint positions[20] for the jth configuration as q^j:

$$q^j = \begin{bmatrix} q_1 & q_2 & q_3 & q_4 \end{bmatrix}^T \tag{3.3}$$

For the estimation of T_{base}^{camera}, we will use a non-linear least squares estimate to determine the unknown parameters. This is an iterative process which attempts to converge on the solution vector, x, given some non-linear residual function, $f(x)$. The process starts with an initial guess, x_0, and with each subsequent guess for x, it attempts to minimize the residual function, $f(x)$. If the residual function is minimized within some tolerance, or if the solution fails to converge after some number of iterations, the estimation halts.

Next we define the unknown parameters and the set of non-linear equations to compute the residual. We define the number of configurations for joint 1 as C_1, the number of configurations for joint 2 as C_2, and the number of markers rigidly attached to the robot as M. The constant M includes markers which are rigidly attached to link 1 as well as link 3. For simplicity, let us assume that $C_1 = C_2$, and then we can define $C = C_1 = C_2$. We also assume that there is only one marker on the third link, and we call this the Mth marker. Altogether, this gives us a total of $(3CM + 3C) = 3C(M + 1)$ equations and $(6 + 3M)$ unknowns. There are three unknowns corresponding the x, y, and z location of each marker, s_i, in its fixed frame, plus the six-dimensional vector defining the transformation matrix, T_{base}^{camera}. In addition, there are three equations for each marker at each configuration. However, the set of joint 1 configurations captures all markers, whereas the set of joint 2 configurations only captures the marker fixed on link 3. The $3C$ in the expression defining the number of equations corresponds to each of x, y, and z for marker M when moving joint 2 through C configurations. The $3CM$ corresponds to each of x, y, and z for all M markers when moving joint 1 through C configurations.

[20] Here, we use a superscript for j so that the notation does not clash with the rest of the thesis, which uses subscript k to indicate the kth joint angle. Notice also that the method described never requires the fourth joint angle, since all computations are in terms of the first three links. Nonetheless, the angle is included in the definition of q^j for completeness.

In order compute the complete residual vector, $f(x)$, we must first define the residual vector for an individual marker, i, at some configuration, j. We denote this vector as r_{ij}, which can be computed as follows:

$$r_{ij} = \begin{bmatrix} r_{ij}^x \\ r_{ij}^y \\ r_{ij}^z \end{bmatrix} = \begin{cases} HT_{base}^{camera} \begin{bmatrix} p_{ij} \\ 1 \end{bmatrix} - HT_0^1(q^j) \begin{bmatrix} s_i \\ 1 \end{bmatrix} & \text{if } i < M,\ j \leq C \\ \\ HT_{base}^{camera} \begin{bmatrix} p_{ij} \\ 1 \end{bmatrix} - HT_0^3(q^j) \begin{bmatrix} s_i \\ 1 \end{bmatrix} & \text{if } i = M,\ j \leq 2C \end{cases} \tag{3.4}$$

In this case, H is simply a selection matrix defined to select the first three rows of T, as shown here:

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{3.5}$$

As throughout the rest of the thesis, the 0th frame is coincident with the base frame. Therefore, the residual function simply subtracts the guessed point, s_i, from the point in camera coordinates, p_{ij}, after they are transformed into robot base coordinates. p_{ij} is transformed using the guess for T_{base}^{camera} in the least squares estimate. As the solution begins to converge, the residual will approach zero since these are the same points in 3D space. Putting all of this together, we can define the residual vector, R_j, for a certain configuration, $j \leq C$:

$$R_j = \begin{bmatrix} (r_{1j})^T & \cdots & (r_{Mj})^T \end{bmatrix}^T \tag{3.6}$$

and for a certain configuration, $C < j \leq 2C$:

$$R_j = r_{Mj} \tag{3.7}$$

Using these, we now construct the complete input and residual vectors, x and $f(x)$,

respectively:

$$x = \left[\begin{array}{cccccccccc} p_x & p_y & p_z & \Omega_z & \Omega_y & \Omega_x & (s_1)^T & (s_2)^T & \cdots & (s_M)^T \end{array} \right] \tag{3.8}$$

$$f(x) = \left[\begin{array}{ccc} (R_1)^T & \cdots & (R_{2C})^T \end{array} \right] \tag{3.9}$$

where the 3x1 orientation vector, Ω, is a set of z-y-x Euler angles defining the rotation of the camera frame relative to the base frame, and the 3x1 position vector, p, defines the translation to the camera frame in robot base coordinates.

The non-linear least squares method described produces a transformation matrix, T_{base}^{camera}, which is quite precise in transforming camera coordinates to robot base coordinates. Typically, this method produces results with mean and maximum marker errors of about 1.3 mm and 4 mm, respectively[21]. The mean error tends to be approximately in line with the calibration error reported by the Tracking Tools software during wanding, and we should not expect to achieve better results than the preliminary wanding calibration. Therefore, the non-linear least squares method seems to be quite appropriate for this application and should be expected to yield very precise results for grasping experiments.

3.2.4 Extensions to Robot Parameter Estimation

The non-linear least squares method previously introduced could be easily extended to estimation of other robot parameters, such as the inertia terms, masses, center-of-masses, and Denavit-Hartenberg parameters. These terms could be added to the x vector and could then be computed appropriately in the residual vector, $f(x)$. In fact, the arm could be moved throughout the joint space at various velocities and accelerations while recording the applied torques, joint positions, accelerations, and velocity estimates. If the full set of dynamic equations was incorporated into the residual vector, the non-linear least squares estimate could theoretically solve for the minimal error solution to all of these parameters. The procedure described is not implemented in this thesis, but would be an interesting extension for future work.

[21] These errors are found by computing the residual distances, $\sqrt{(r_{ij}^x)^2 + (r_{ij}^y)^2 + (r_{ij}^z)^2}$.

3.3 Rigid Body Tracking

Now that we have discussed a method for transforming all camera data into the base frame of the robot, we implement one final feature necessary for performing grasping experiments. In order to grasp rigid bodies in the workspace, it is crucially important that the experimenter have access to the current position and orientation of objects at all times. Of course, it is also important that one has knowledge of the object geometries; however, extracting this information from camera data is a research topic of its own, and is outside the scope of this thesis. Instead we will discuss how the transformation found previously can be used to convert rigid body data into the base frame of the robot. This is generally a straightforward process, but is made more difficult here due to the fact that the tracking data is defined in a left-handed coordinate system.

3.3.1 Conventions for Rigid Body Coordinate Frame

In order to transform rigid body position and orientation from one coordinate frame to another, we need to understand how the object is defined by the tracking system. Recall from Section 3.1.2 that we initialized rigid bodies as trackable objects

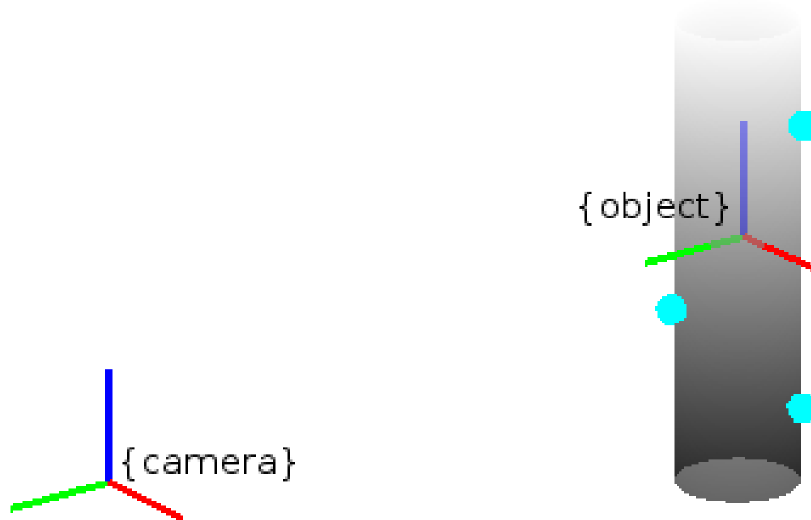


Figure 3.4: The coordinate frame rigidly attached to a trackable object is initially defined at the centroid of its markers and with its axes aligned with the left-handed camera frame.

in the Tracking Tools software. We did so by allowing the software to record the relative positions of three or more markers, as long as they were placed on the object in a non-collinear fashion. When each trackable object is created, the camera system stores these marker positions relative to a coordinate frame rigidly 'attached' to the object. Therefore, the marker positions are fixed in the object frame, and we will refer to them in this frame as the 'marker base positions'. The object coordinate frame is initially defined with origin at the centroid of all of its markers, and with axes aligned with the axes of the camera system. Recall that since the camera coordinate frame is left-handed, so is the coordinate frame attached to the object. This coordinate frame definition is shown more clearly in Figure 3.4.

3.3.2 Transforming Handedness of the Camera Frame

The Tracking Tools API uses unit quaternions[22] to describe orientation information for trackable objects. Each unit quaternion is a vector of four parameters collectively describing a three-dimensional rotation; the additional degree of freedom is constrained by the restriction that the vector must have unit magnitude. These are described well in [18], and the equations are included here as well, for reference. A unit quaternion, ϵ, can be thought of in terms of the axis-angle representation introduced in Section 2.1.1:

$$\epsilon = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \end{bmatrix} = \begin{bmatrix} k_x sin(\frac{\theta}{2}) \\ k_y sin(\frac{\theta}{2}) \\ k_z sin(\frac{\theta}{2}) \\ cos(\frac{\theta}{2}) \end{bmatrix} \tag{3.10}$$

Given that this represents the orientation of an object in the workspace, we would like to modify the camera and object coordinate systems from left to right handed, so as to be consistent with the robot coordinate frames. If we reverse the z-axis on both the camera and object frames, the rotation matrix between them is still the same, as described in Section 3.2.1. Furthermore, if we think of the quaternion in terms of the axis-angle representation, flipping the z-axis of the camera frame

[22] These are alternately known as Euler parameters.

definition also requires flipping the sign of the k_z component of the unit axis vector, since that vector is defined in terms of the left-handed camera frame. Therefore, if we denote the left-handed camera frame as l and the right-handed camera frame as r, then the new quaternion defining trackable orientation in terms of the right-handed camera frame is simply:

$$\epsilon^r = \begin{bmatrix} \epsilon_1^l & \epsilon_2^l & -\epsilon_3^l & \epsilon_4^l \end{bmatrix}^T \tag{3.11}$$

The unit quaternion, ϵ^r, can then be used to compute the rotation matrix between a right-handed camera frame and a right-handed object frame. The generalized equation in [18] helps us to do this, and is included here for reference:

$$R(\epsilon) = \begin{bmatrix} 1-2(\epsilon_2)^2-2(\epsilon_3)^2 & 2(\epsilon_1\epsilon_2-\epsilon_3\epsilon_4) & 2(\epsilon_1\epsilon_3+\epsilon_2\epsilon_4) \\ 2(\epsilon_1\epsilon_2+\epsilon_3\epsilon_4) & 1-2(\epsilon_1)^2-2(\epsilon_3)^2 & 2(\epsilon_2\epsilon_3-\epsilon_1\epsilon_4) \\ 2(\epsilon_1\epsilon_3-\epsilon_2\epsilon_4) & 2(\epsilon_2\epsilon_3+\epsilon_1\epsilon_4) & 1-2(\epsilon_1)^2-2(\epsilon_2)^2 \end{bmatrix} \tag{3.12}$$

Given an arbitrary rotation matrix, we can also extract the unit quaternion representation as in [18], where each r is as defined in Equation 2.4:

$$\begin{aligned} \epsilon_1 &= \frac{r_{32}-r_{23}}{4\epsilon_4} \\ \epsilon_2 &= \frac{r_{13}-r_{31}}{4\epsilon_4} \\ \epsilon_3 &= \frac{r_{21}-r_{12}}{4\epsilon_4} \\ \epsilon_4 &= \frac{1}{2}\sqrt{1+r_{11}+r_{22}+r_{33}} \end{aligned} \tag{3.13}$$

Finally, all that is left is to transform the position from the left-handed camera frame to the right-handed one. This is very easy to do, as described in Section 3.2.1. If we are given the trackable position in left-handed coordinates, p^l, then the same position represented in the right-handed camera frame can be denoted as p^r:

$$p^r = \begin{bmatrix} p_x^l & p_y^l & -p_z^l \end{bmatrix}^T \tag{3.14}$$

We can use all of this information to construct the final transformation matrix, T_{camera}^{object}, which describes the position and orientation of the object frame in the

camera frame, both of which are now right-handed:

$$T_{camera}^{object} = \left[\begin{array}{ccc|c} & R(\epsilon^r) & & p^r \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{3.15}$$

3.3.3 Transforming Right Handed Data into the Robot Frame

Now that we have the transformation from the object frame to the camera frame and the transformation from the camera frame to the base frame, the final step is to compute the transformation for this object in the robot frame. Notice that this easily scales to any number of trackable objects, and will provide the current position and orientation information of each trackable in the coordinate system used by the robotic controller. The final transformation is quite simple, as shown here:

$$T_{base}^{object} = T_{base}^{camera} T_{camera}^{object} \tag{3.16}$$

We also collect two additional pieces of data as a way of assessing the accuracy of the transformation, the expected position of all trackable markers and the actual position of all trackable markers. We will denote these as m_e and m_a, respectively. Given the base marker positions in the right-handed object frame, m_b, we can compute m_e as follows:

$$m_e = H T_{base}^{object} m_b \tag{3.17}$$

where H is a selection matrix on the first three rows of T, as defined in Equation 3.5. Next, given m_e for each marker as well as the set of all raw marker locations, we correlate m_e and m_a by their minimum distance apart. This gives us a set of trackable marker locations (actual and expected) in robot base coordinates for each visible trackable, and for each frame that is recorded. The marker error, m_{err}, is then computed as the Euclidean distance between m_e and m_a:

$$m_{err} = \sqrt{(m_e^x - m_a^x)^2 + (m_e^y - m_a^y)^2 + (m_e^z - m_a^z)^2} \tag{3.18}$$

3.4 Grasp Acquisition Database

To bring together all of the data that has been collected from the robotic controller and the tracking software, we implement the final portion of the experimental grasp test bed, the Grasp Acquisition Database. The database, as shown in the system diagram in Figure 3.1, is located on the RPI network and is accessible remotely as well (http://grasp.robotics.cs.rpi.edu/). In this section, we discuss the implementation of this database and its potential for storing and retrieving grasping experiments.

3.4.1 Implementation and Design

The Grasp Acquisition Database is designed to store the raw data from grasping experiments, which can later be loaded for analysis, or replayed on the robot or in simulation. There is also a strong potential for data mining here, which will open the doors to grasp analysis and artificial intelligence. For example, once hundreds or even thousands of experiments were performed on the robot, the researcher could search for finger velocity data and relate it to how the arm was positioned and how the object moved. Comparing similar experiments to one another may lead to a set of observations about what makes a 'successful' grasp.

The database is implemented on a Linux server running MySQL software. The tables are all set up in a relational fashion, with each piece of data corresponding to a single grasping experiment. In the library of grasping utilities, we have implemented a function for importing and exporting experiments to and from MATLAB. In addition, another student is currently working on a front-end website which can be used to access grasping data and replay experiments, though the completion of this interface is left for future work.

3.4.2 Data Stored

Virtually all of the data we can capture during an experiment can be recorded and saved in the grasp database. Here we will introduce exactly what data can be stored, yet a more detailed description of the MATLAB struct definitions is provided in Appendix B.3. Recall from Section 3.1.2 that WAM and tracking data is

recorded at approximately 100 Hz, due to the single-threaded nature of the software. Therefore, at about 100 Hz, we have access to the raw marker locations, trackable marker locations (estimated, actual, and error), trackable positions and orientations, robot joint positions, velocities, and torques. In addition, we have access to the commanded joint positions and velocities, as well as the corresponding data for the Cartesian space controller, if applicable[23].

In addition to the data described, we also record the current experiment time at each step. Accompanying each frame is a flag to indicate whether camera data was captured on that frame. Since the system typically requests tracking data faster than the refresh rate of the cameras, we do not expect to receive tracking data on every frame. If tracking data is not captured on a particular frame, the flag indicates that this is so, and the data recorded for that frame is simply copied from the last available update. Finally, we record one set of meta data for each experiment, including the time stamp when the experiment was inserted, the type of experiment, a description of the experiment, and the calibration data from camera to robot base. There is also potential in the future to record a high-definition video clip of each experiment, but such a feature is not implemented in this thesis.

[23]On every cycle of the control loop, there is a set of desired joint positions and velocities. However, if the controller is not in Cartesian space mode, the set of desired Cartesian space positions and velocities are not computed, and are therefore not recorded.

CHAPTER 4
CONCLUSION

4.1 Future Work

In this thesis, we have implemented the basis for a robotic grasping test bed, but there is still quite a bit of research that can be done to build off of this further. In this section, we will list some of the topics for future work and briefly describe how they may be incorporated into the current system. The following is a list of potential topics:

- Velocity Feedback – Poor velocity feedback seems to be the cause of some accuracy issues in the controller, including joint vibrations. In Section 2.1.4, we saw that discrete differentiation produces a noisy velocity signal, while moving averages introduced a lag and low-pass filtering effects. A Kalman Filter may be the best alternative, as it could take into account the system dynamics and predict joint accelerations. The biggest challenge in implementing this would be unknown dynamic terms, such as high friction.

- Sample Rate Improvement – As described in Appendix B.5, the sample rate of the arm controller is currently limited to about 500 Hz. It seems that this is either a minor issue in the configuration of the Simulink model or a limitation in the Barrett electronics. It may or may not be possible to resolve this issue, but it is worth looking into for its potential performance gains.

- Camera Synchronization – While the system synchronization works well currently, it is limited by the single-threaded nature of MATLAB. Splitting the camera capture code into a separate thread or process may allow us to record controller data much closer to the actual sample rate of the WAM.

- Trapezoidal Profile Blends – The trapezoidal profile is currently limited to 'fixed' trajectories. In other words, it receives and processes one high-level command at a time, without blending the acceleration and velocity curves

between moves. This means that the arm decelerates to zero velocity at every via point, when it could ideally move at a non-zero velocity through the via point on its way to the next desired position.

- Object Avoidance – Avoiding obstacles in the workspace is a well-researched area and would be a worthy improvement to the current arm controller. This would prevent the arm from crashing into the table, for example, if an erroneous command was sent to the robot.

- Real-Time Hand Controller – As explained in Section 2.6, the hand controller currently operates in supervisory control mode. Implementing a real-time controller would allow the user to collect position and velocity data at about 30-33 Hz, which could then be paired nicely with the data from the arm and stored in the GAD. There are certain challenges here, in that the hand is velocity-controlled (as opposed to torque-controlled), the fingers are not back-drivable, and some means of torque switch detection is necessary for the Barrett Hand.

- Quaternion Orientation Control – Cartesian space orientation control using the axis-angle representation has limitations due to its singularities at 0 and 180 degrees. Recall that we avoided this problem by declaring a zero rotation if θ was less than $1e^{-3}$ rad. In [20], a more robust approach is introduced, in which all four quaternion parameters can be used to control the tool orientation.

- Grasping Experiments – The ultimate goal of this research is to study robotic grasping. Performing grasping experiments on the physical hardware and in dVC3d simulations should provide some very interesting comparisons. While adding the wrist to the WAM would give it three extra degrees of freedom, grasping experiments can still be performed through careful selection of trajectories.

- Grasp Database – The first revision of the grasp database has been specially designed to store data from our grasping test bed. However, it has also been designed with flexibility in mind, and a future improvement would be to ex-

pand this to different robotic arms, hands, and object geometries. A front end interface to the database is currently in progress, and this will surely aid in the implementation of these goals.

4.2 Summary

Throughout this thesis, we have worked towards the development of a robotic grasping test bed. We began by evaluating the forward and inverse kinematics as well as the system dynamics of the Barrett WAM and Hand. We evaluated gravity compensation and introduced a method for verifying manufacturer specifications. The method was successful in verifying the model parameters, but there was some area for improvement in the area of static friction. We also analyzed the position and velocity dependent friction on each of the joints, which was found to be quite non-linear and larger than expected. We discussed several approaches to PID control and the selection of gains, and then built Cartesian space control as a wrapper around joint space control using the Jacobian. We implemented a hand controller in supervisory control mode, but a real-time controller is not yet complete.

Having finished the basis for a robotic controller, we developed a suite of software tools to drive the arm and hand, while at the same time collecting control and tracking data. UDP control packets and the general software and system design was discussed. A non-linear least squares estimation was explained in detail for transforming camera coordinates to robot base coordinates. Rigid body transformations were implemented so that all data is stored in one consistent coordinate frame. Finally, a Grasp Acquisition Database was developed to store calibration, control, and tracking data from experiments.

In this thesis, we have only begun the work on our robotics grasping test bed, and there is still much room for expansion and future work. Improvements to the controller, artificial intelligence and data mining, as well as comparisons between physical experiments and simulation are all potential areas for future research. After performing many experiments in robotic grasping, we will be able to analyze a large wealth of data and will continue to make strides towards careful robotic manipulation and cooperation with humans. Robotics has and always will be an exciting

field, especially in areas of consumer electronics, prosthetics, and automation, and we hope that this research will advance it further towards its full potential.

REFERENCES

[1] Shimon Y. Nof. *Handbook of Industrial Robotics*, volume 1. John Wiley & Sons, 2nd, illustrated edition, 1999.

[2] Binh Nguyen and Jeffrey C. Trinkle. dvc3D: a three dimensional physical simulation tool for rigid bodies with contacts and Coulomb friction. *The 1st Joint International Conference on Multibody System Dynamics*, May 2010.

[3] Charles de Granville and Andrew H. Fagg. Learning grasp affordances through human demonstration. *Autonomous Robots*, 2008.

[4] Nancy S. Pollard and Victor B. Zordan. Physically based grasping control from example. *ACM SIGGRAPH Symposium on Computer Animation*, 2005.

[5] Staffan Ekvall and Danica Kragic. Interactive grasp learning based on human demonstration. *ICRA Robotics and Automation*, April 2004.

[6] Micha Hersch, Florent Guenter, Sylvain Calinon, and Aude Billard. Dynamical system modulation for robot learning via kinesthetic demonstrations. *IEEE Transactions on Robotics*, December 2008.

[7] Jen-Shi Chen and Jyh-Ching Juang. A robust adaptive friction control scheme of robot manipulators. *ICGST- ARAS*, 5, January 2006.

[8] Craig T. Johnson and Robert D. Lorenz. Experimental identification of friction and its compensation in precise, position controlled mechanisms. *IEEE Transactions on Industry Applications*, 28(6), December 1992.

[9] Alessandro De Luca and Lorenzo Ferrajoli. A modified Newton-Euler method for dynamic computations in robot fault detection and control. *IEEE International Conference on Robotics and Automation*, May 2009.

[10] Ernest D. Fasse and Jan F. Broenink. A spatial impedance controller for robotic manipulation. *IEEE Transactions on Robotics and Automation*, 13(4), August 1997.

[11] Dan Simon. Kalman filtering. *Embedded Systems Programming*, pages 72–79, June 2001.

[12] Corey Goldfeder, Matei Ciocarlie, Hao Dang, and Peter K. Allen. The Columbia grasp database. *IEEE International Conference on Robotics and Automation*, May 2009.

[13] Andrew T. Miller and Peter K. Allen. GraspIt! A versatile simulator for robotic grasping. *IEEE Robotics & Automation Magazine*, 11(4):111–122, December 2004.

[14] Greg Starr and Matt Courtney. How to control the WAM using MATLAB. http://www-mep.unm.edu/WAM/. Date Last Accessed, Mar 28, 2011.

[15] Robert J. Schilling. *Fundamentals of Robotics: Analysis and Control*. Prentice-Hall, Inc., 1990.

[16] Bruno Siciliano and Oussama Khatib. *Springer Handbook of Robotics*. Springer, 2008.

[17] Bill Baxter. Fast numerical methods for inverse kinematics. http://billbaxter.com/courses/290/html/. Date Last Accessed, Mar 7, 2010.

[18] John J. Craig. *Introduction to Robotics: Mechanics and Control*. Pearson Education, Inc., 3rd edition, 2005.

[19] H. Olsson, K.J. Astrom, C. Canudas de Wit, M. Gafvert, and P. Lischinsky. Friction models and friction compensation. *European Journal of Control*, 4(3):176–195, November 1998.

[20] Tony Barrera, Anders Hast, and Ewert Bengtsson. Incremental spherical linear interpolation. *SIGRAD*, 13:7–13, 2004.

[21] University of Michigan. Control tutorials for MATLAB: PID tutorial. http://www.engin.umich.edu/group/ctm/PID/PID.html. Date Last Accessed, Mar 16, 2011.

[22] Alan V. Oppenheim, Alan S. Willsky, and S. Hamid Nawab. *Signals & Systems*. Pearson Education, 2nd edition, 1997.

[23] Young-Hoo Kwon. Transformation of the inertia tensor. http://kwon3d.com/theory/moi/triten.html. Date Last Accessed, Mar 7, 2010.

[24] Young-Hoo Kwon. Inertia tensor. http://kwon3d.com/theory/moi/iten.html. Date Last Accessed, Mar 7, 2010.

[25] NaturalPoint. NaturalPoint tracking tools users manual: Getting started. Version 2.0, April 2010.

[26] Barrett Technology Inc. BA4-310 system user manual. Version 1.1, pages 68-69.

[27] Barrett Technology Inc. WAM arm: Inertial specifications. Document: D1005, Version: AA.00, 2006.

[28] Barrett Technology Inc. Example WAM configuration file. http://web.barrett.com/svn/btclient/tags/LATEST/config/WAM4.conf. Date Last Accessed, Feb 26, 2011.

[29] The MathWorks. xPC Target 4.4: Supported hardware. http://www.mathworks.com/products/xpctarget/supportedio.html. Date Last Accessed, Feb 26, 2011.

[30] Barrett Technology Inc. WAM arm: CAN property reference. Revision 39.

[31] Barrett Technology Inc. WAM arm: User's guide. Document: D1001, Version: AA.00, 2005.

[32] Barrett Technology Inc. BarrettHand: BH8-255 user manual. Version 1.0, July 1999.

APPENDIX A
FORMULA DERIVATIONS

A.1 Backward Euler Velocity Error

We show that using the Backward Euler method to compute the velocity error is equivalent to subtracting $\dot{q}_{des}[k]$ from $\dot{q}[k]$.

$$\begin{aligned}\dot{e}[k] =& \frac{e[k] - e[k-1]}{\Delta t} \\ =& \frac{q[k] - q_{des}[k]}{\Delta t} - \frac{q[k-1] - q_{des}[k-1]}{\Delta t} \\ =& \frac{q[k] - q[k-1]}{\Delta t} - \frac{q_{des}[k] - q_{des}[k-1]}{\Delta t} \\ =& \dot{q}[k] - \dot{q}_{des}[k]\end{aligned}$$

A.2 Trapezoidal Profile Curve

We show how to form the trapezoidal profile given the initial position x_0, target position x_f, time elapsed t, positive acceleration a_{pos}, and positive velocity v_{pos}. First, we begin by calculating a and v adjusted for the direction of motion:

$$\begin{aligned}a =& a_{pos} * sgn(x_f - x_0) \\ v =& v_{pos} * sgn(x_f - x_0)\end{aligned}$$

Next we define several variables. The points (t_1, x_1) and (t_2, x_2) denote the time and position on the curve at the point immediately after acceleration and immediately before deceleration on the curve. In addition, we define x_{ramp} as the distance traveled during the acceleration phase, t_{mid} as the time spent at constant velocity, v_{mid} as the velocity at the midpoint of the trapezoid, and t_f as the time the target position is reached. We compute t_1 and x_{ramp} assuming the curve has a

constant velocity section:

$$t_1 = \frac{v}{a}$$
$$x_{ramp} = \frac{1}{2}at^2$$

Now using these values, we can check if the curve has a constant velocity section. If so, $|2x_{ramp}|$ will be less than $|x_f - x_0|$, and we compute t_{mid} and v_{mid} as follows:

$$t_{mid} = (x_f - x_0 - 2x_{ramp})/v$$
$$v_{mid} = v$$

Otherwise, we modify t_1 and x_{ramp}, then set t_{mid} and v_{mid} as follows:

$$x_{ramp} = (x_f - x_0)/2$$
$$t_1 = \sqrt{(2x_{ramp})/a}$$
$$t_{mid} = 0$$
$$v_{mid} = at_1$$

Whether or not we have a constant velocity section, these general equations apply and are computed next:

$$t_2 = t_1 + t_{mid}$$
$$t_f = 2t_1 + t_{mid}$$
$$x_1 = x_0 + x_{ramp}$$
$$x_2 = x_1 + t_{mid} * v_{mid}$$

Finally, the following equations can be used to determine the current position and velocity on the curve. Notice that if there is no constant velocity region, the equa-

tions above dictate $t_1 = t_2$, and the second condition will never evaluate to true:

$$x_{des} = \begin{cases} x_0 + \frac{1}{2}at^2 & \text{if } 0 \le t \le t_1 \\ x_1 + v(t - t_1) & \text{if } t_1 < t < t_2 \\ x_2 + v_{mid}(t - t_2) - \frac{1}{2}a(t - t_2)^2 & \text{if } t_2 \le t \le t_f \end{cases}$$

$$\dot{x}_{des} = \begin{cases} at & \text{if } 0 \le t \le t_1 \\ v_{mid} & \text{if } t_1 < t < t_2 \\ v_{mid} - a(t - t_2) & \text{if } t_2 \le t \le t_f \end{cases}$$

A.3 Selecting Gains Based on Inertia

We show the derivation of transfer function for the closed-loop PID block in Figure 2.9, and then use it to extract the gains necessary for critical damping.

$$\begin{aligned} K_d(\dot{q}_{des} - \frac{1}{s}\ddot{q}) + K_p(q_{des} - \frac{1}{s^2}\ddot{q}) &= M\ddot{q} \\ K_d\dot{q}_{des} - K_d\frac{1}{s}\ddot{q} + K_p q_{des} - K_p\frac{1}{s^2}\ddot{q} &= M\ddot{q} \\ \cancelto{0}{K_d\dot{q}_{des}} + K_p q_{des} &= \ddot{q}(M + \frac{1}{s}K_d + \frac{1}{s^2}K_p) \end{aligned}$$

Setting $\dot{q}_{des}$ to zero, then solving for the transfer function $\frac{\ddot{q}}{q_{des}}$,

$$\begin{aligned} \frac{\ddot{q}}{q_{des}} &= \frac{K_p}{M + \frac{K_d}{s} + \frac{K_p}{s^2}} \\ &= \frac{K_p s^2}{Ms^2 + K_d s + K_p} \\ &= \frac{\frac{K_p}{M}s^2}{s^2 + \frac{K_d}{M}s + \frac{K_p}{M}} \end{aligned}$$

Finally, setting the denominator equal to $s^2 + 2\zeta\omega_n s + \omega_n^2$ with $\zeta = 1$ for critical damping,

$$s^2 + \frac{K_d}{M}s + \frac{K_p}{M} = s^2 + 2\zeta\omega_n s + \omega_n^2$$

We solve for ω_n,

$$\omega_n^2 = \frac{K_p}{M}$$
$$\boldsymbol{\omega_n = \sqrt{K_p/M}}$$

and also for K_p,

$$\mathbf{K_p = M\omega_n^2}$$

and lastly for K_d,

$$2\omega_n = 2\sqrt{K_p/M} = \frac{K_d}{M}$$
$$4\frac{K_p}{\cancel{M}} = \frac{K_d^2}{M^{\cancel{2}}}$$
$$K_d = \sqrt{4MK_p}$$
$$\mathbf{K_d = 2\sqrt{MK_p}}$$

Notice that setting any one of the unknowns K_p, K_d, or ω_n constrains this set of equations for each joint.

APPENDIX B
TECHNICAL NOTES

In this appendix, we provide technical data and notes pertaining to the thesis. The lab website contains more technical information and can be found at http://www.cs.rpi.edu/twiki/view/RoboticsWeb/WebHome.

B.1 Arm and Hand Controller

B.1.1 Denavit-Hartenberg

Figure B.1 shows the Denavit-Hartenberg frames for the 4 DOF Barrett WAM, the notation of which is described in [15]. These frame definitions are also listed in [26], though we have manually verified them as well. The arm is in the zero position when it is extended perpendicular to the ground. The intersection of each x_i and z_i axis is the origin of the ith frame, with y_i (not shown) forming a right-handed coordinate system. Axes z_1 and z_3 are directed into the page, and dashed lines represent that two frames have the same origin. Due to the triangular structure of the elbow joint (joint 4), frame 4 is inserted as a reference point in computing the transform between frame 3 and frame 5. The origin of the tool frame (frame 5) is located at the intersection of the plane of the palm and the axis pointing directly through the center of the last cylindrical link. Finally, in the figure, each ith frame rotates about the z_{i-1} axis.

Table B.1 lists the Denavit-Hartenberg parameters for the 4 DOF Barrett WAM. Notice that if the hand is not attached, $d_5 = 0.35$, and the origin of the tool frame is shifted proximally along the last cylindrical link to align with the plane of the cylinder where the hand would normally attach.

B.1.2 Manufacturer Provided Parameters

Table B.2 contains the link masses and link center-of-masses for links 1 through 5, where link 5 refers to the hand. We also list the mass inertia matrices (kg·mm^2) for these links. Center-of-mass and inertia parameters are not provided for the

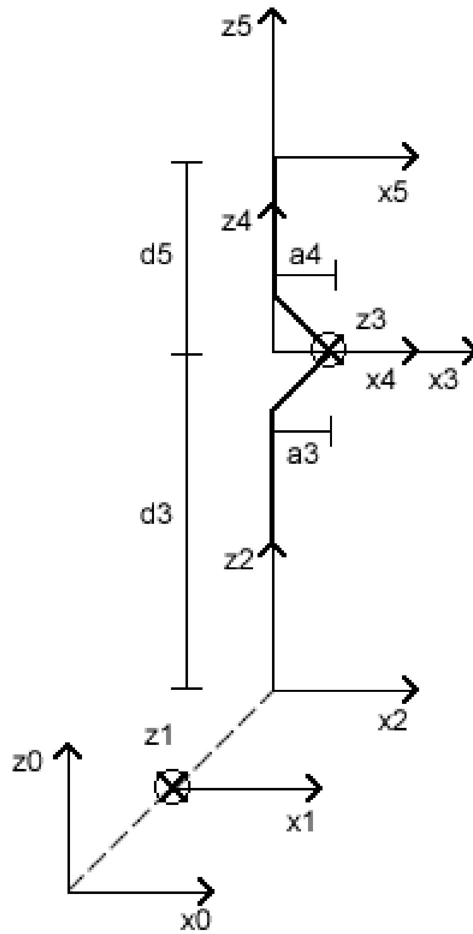


Figure B.1: This figure shows the Denavit-Hartenberg frames for the 4 DOF Barrett WAM, starting at the base frame and extending out to the tool frame. We define the world frame coincident to the base frame (frame 0), since the base is a non-moving frame.

hand, and Barrett has told us that they do not have those numbers available. Each center-of-mass is in terms of their own frame (i.e. (x_1, y_1, z_1) is represented in frame 1). The same is true for the inertia matrices. Please refer to [27] for more details.

$$M_1^{(1)} = \begin{bmatrix} 242919.8516 & 636.5542 & -93.1687 \\ 636.5542 & 92036.7003 & -262.6542 \\ -93.1687 & -262.6542 & 207050.7372 \end{bmatrix}$$

$$M_2^{(2)} = \begin{bmatrix} 32413.1852 & -20.2663 & -143.7579 \\ -20.2663 & 21649.2574 & -38.6737 \\ -143.7579 & -38.6737 & 25026.5101 \end{bmatrix}$$

$$M_3^{(3)} = \begin{bmatrix} 138575.0448 & -16890.4832 & 6.5293 \\ -16890.4832 & 5749.2209 & -7.1669 \\ 6.5293 & -7.1669 & 141310.5180 \end{bmatrix}$$

$$M_4^{(4)} = \begin{bmatrix} 39049.0802 & -2.2101 & 21.9187 \\ -2.2101 & 39649.6217 & 1.6243 \\ 21.9187 & 1.6243 & 2177.6044 \end{bmatrix}$$

Frame	a	d	α	q
1	0	0	$-\pi/2$	q_1
2	0	0	$\pi/2$	q_2
3	0.045	0.55	$-\pi/2$	q_3
4	-0.045	0	$\pi/2$	q_4
5	0	0.44*	0	0

Table B.1: This table lists the Denavit-Hartenberg parameters for the 4 DOF Barrett WAM for each of the 5 coordinate frames attached to its rigid links. All distances are measured in meters, and all angles are measured in radians. *The table assumes the hand is attached; if it is not, then $d_5 = 0.35$.

Link	m	x	y	z
1	8.3936	0.3506	132.6795	0.6286
2	4.8487	-0.2230	-21.3924	13.3754
3	1.7251	-38.7565	217.9078	0.0252
4	1.0912	11.7534	-0.1092	135.9144
5	1.2	N/A	N/A	N/A

Table B.2: This table lists the link masses (kg) and center-of-masses (mm) for each of the 5 rigid links in the 4 DOF Barrett WAM. Link 5 refers to the Barrett Hand.

B.1.3 Gravity Compensation Equations

The gravity compensation equations for joints 1 through 4 collectively have six coefficients in common. The coefficients are denoted C_i, where i simply represents the ith coefficient. In addition, m_j represents the mass of the jth link, and c_{jx}, c_{jy}, and c_{jz} represent the x, y, and z center-of-mass values of the jth link. Denavit-Hartenberg parameters, a and d, are also subscripted with their corresponding frames.

$$
\begin{aligned}
C_1 &= 9.81(m_4c_{4z} + m_5d_5 + m_5c_{5z}) \\
C_2 &= 9.81(m_4a_4 + m_4c_{4x} + m_5a_4 + m_5c_{5x}) \\
C_3 &= 9.81(m_3c_{3z} + m_4c_{4y} + m_5c_{5y}) \\
C_4 &= 9.81(m_3a_3 + m_3c_{3x} + m_4a_3 + m_5a_3) \\
C_5 &= 9.81(m_2c_{2z} + m_3d_3 - m_3c_{3y} + m_4d_3 + m_5d_3) \\
C_6 &= 9.81(m_2c_{2x})
\end{aligned}
$$

These coefficients are multiplied by sequences of sines and cosines to determine the gravity torques. The sets of sines and cosines are as follows, where s and c correspond to the sine and cosine of their subscripted joint angle:

$$
\begin{array}{lll}
\lambda_1 = -c_4s_2 - c_2c_3s_4 & \beta_1 = s_2s_3s_4 & \gamma_1 = -c_3c_4s_2 - c_2s_4 \\
\lambda_2 = s_2s_4 - c_2c_3c_4 & \beta_2 = c_4s_2s_3 & \gamma_2 = c_3s_2s_4 - c_2c_4 \\
\lambda_3 = c_2s_3 & \beta_3 = c_3s_2 & \\
\lambda_4 = -c_2c_3 & \beta_4 = s_2s_3 & \\
\lambda_5 = -s_2 & & \\
\lambda_6 = -c_2 & &
\end{array}
$$

Finally, the gravity torque equations are constructed by multiplying the expressions listed above:

$$
\begin{aligned}
G_1(q) &= 0 \qquad \text{(B.1)} \\
G_2(q) &= C_1\lambda_1 + C_2\lambda_2 + C_3\lambda_3 + C_4\lambda_4 + C_5\lambda_5 + C_6\lambda_6 \\
G_3(q) &= C_1\beta_1 + C_2\beta_2 + C_3\beta_3 + C_4\beta_4 \\
G_4(q) &= C_1\gamma_1 + C_2\gamma_2
\end{aligned}
$$

Joint	K_p	K_i	K_d
1	900	1250	10
2	2500	2500	20
3	600	1250	10
4	500	250	2.5

Table B.3: This table lists the joint space PID gains for the each of the joints of the 4 DOF Barrett WAM. The Cartesian space controller is simply a wrapper on joint space control, and therefore uses these same gains.

B.1.4 PID Gains

Listed in Table B.3 are the proportional, integral, and derivative gains we are using for the joint space controller. These were given in [28]. The gains are listed for each of the four joints of the WAM. Notice that the derivative gains are noticeably less than the others, but this is to be expected since it is multiplying on a different scale; specifically, it multiplies on units of rad/s as opposed to rad.

B.2 MATLAB and Simulink

Several details of the Simulink model are slightly obscure, and for that reason they are clarified further here. For the most part, the code is well-commented, and documentation blocks can be found within each subsystem in the Simulink model. In addition, more technical details are provided on the lab website and in the Barrett user manuals.

B.2.1 Hardware Drivers

xPC Target is essentially a very minimal real-time operating system developed by The Mathworks. Since it is intended to be highly optimized, there is only a small set of hardware drivers that are supported. The xPC Target machine set up in the lab has a Softing CAN-AC2-PCI card for CAN communication and an Intel PILA8460M 10/100 Mbps Ethernet card for network traffic. xPC Target does not support the onboard Ethernet card in this machine, so it is very important that the network cable be connected from the switch to the card in the PCI slot. The Mathworks has a detailed list of supported hardware on their website in [29].

B.2.2 CAN Initialization

In the main subsystem of the Simulink model, there is a block labeled 'CAN Setup.' This block is executed when the model begins running on the xPC Target machine and is used to initialize communication between xPC Target and the control board in the base of the WAM. There are several properties that are set during the initialization, including the velocity warning limit (VL1), velocity fault limit (VL2), current status (STAT), and maximum allowed motor torques (MT); it is very important that these be sent in the order listed here, or the WAM will not be initialized properly. The values in parentheses correspond to the abbreviated names of these parameters in the Barrett command reference, [30], which is posted on the lab website. The data format for CAN messages tends to stay the same for different WAM firmware versions, but the actual data values are firmware specific. At the time of this writing, the firmware version for the WAM is 0x270080, and the monitor version is 0x60080. Specific details on how to determine the firmware version are available on the lab website.

Each CAN message has two important pieces of information, the message identifier and the actual data. The message identifier is an 11-bit binary value which describes where the message originated and where its destination is. As described in the CAN Communication Specifications in [31], a message identifier of 1024 (1 00000 00000) corresponds to a group broadcast (1) from the host (0) to all motors (0). To send a message to the safety module, we use the message identifier 10 (0 00000 01010). In a similar way, this corresponds to a directed message (0) from the host (0) to the safety module (10). The fact that the safety module has identifier 10, is not listed in the Barrett documentation, and this information was obtained by contacting Barrett technical support.

As a simple example, we discuss the maximum torque packet that is part of the CAN initialization. Other packets will have a similar structure, and this is described more thoroughly in the Barrett documentation. To set the maximum torques of all actuators, we use the CAN message identifier 1024 described previously. This will send a group broadcast from the host to all motors. The data of the message consists of four bytes, describing exactly what the message will do. In the case of

the maximum torque limit, we would like to set the maximum torque property to 4000, and in order to do this we send the unsigned byte values, [171 0 160 15]. The first byte is 10101011 in binary (171), where the first bit indicates we would like to set a value (1), and the rest of the bits indicate which property identifier we would like to set. The property identifiers are listed in tabular form in the Barrett command reference on the lab website. As shown in the table, the property identifier for MT is 43 (0101011). The second byte is always set to zero. The third byte is 10100000 in binary (160), and represents the low-order byte of the property value. The fourth byte is 00001111 in binary (15), and represents the high-order byte of the property value. Together, the last two bytes give the desired property value $4000 = 15(256) + 160$.

B.3 MATLAB Data Definitions

In this section, we discuss the format of the primary structs used in the experiment code. There are three major structs, one for calibration data, one for recorded experiment data, and one for frame data to be sent to the user-defined callback function. The struct definitions are shown below. Specific details about these variables can be found on the lab website.

The following lists the fields of the calibration struct:

- stage
- step
- waitStart
- markerFailures
- markerFailures
- marker3
- q
- pcam

- trackedMarkers
- tformData
  - Tcamwam
  - errmean
  - errmax
  - residual

The following lists the fields of the experiment struct:

- state
- handedness
- frameCount
- frameTimes
- wamData
  - q
  - qd
  - qdes
  - qddes
  - wdes
  - wddes
  - tau
- trackData
  - frameUpdated
  - rawMarkers
  - trackables

        * name
        * lastTrack
        * mbase
        * mapos
        * mepos
        * merr
        * tpos
        * tqtr

The following lists the fields of the frame data struct:

- recvTime
- wamDataRcvd
    - q
    - qd
    - qdes
    - qddes
    - wdes
    - wddes
    - tau
- trackDataRcvd
    - rawMarkers
    - trackables
        * tracked
        * mapos
        * mepos
        * merr
        * tpos
        * tqtr

B.4 UDP Packet Definitions

There are three different types of UDP packets used currently to control and receive feedback from the arm and Hand. These UDP packets transmit data back and forth between the MATLAB machine and the xPC Target machine. The controller for the arm receives high-level control commands and provides real-time feedback via UDP. The controller for the hand is currently only capable of the former, since a controller has not yet been implemented in real-time mode (as discussed in Section 2.6). The format of these packets will be described in detail in this section, and throughout the description, double-precision numbers are represented using the IEEE 754 floating-point format.

B.4.1 Arm Command UDP Packet

The arm command UDP packet is used for sending high-level joint and Cartesian space commands to the arm. Currently it only supports these two control modes, but there is room for expansion so that other modes can be implemented (impedance control, low-level torque mode, etc.). The packet data is essentially an array of 13 double precision numbers; therefore, the total payload is $13(8) = 104$ bytes as shown in Table B.4.

id (1)	pos (4)	acc (4)	vel (4)

Table B.4: This table shows the data format for the UDP packet containing high-level arm commands. It consists of $1 + 4(3) = 13$ doubles, and therefore has a $13(8) = 104$ byte payload.

In the table, the first value is the packet identifier describing the control mode and associated options. Though this parameter is cast as a double, it should really have a value corresponding to an 8-bit unsigned integer[24]. The three highest order bits are currently unused, the next two bits define the position mask, and the final (lowest order) three bits define the control mode. The control mode for a joint space move is zero, and the control mode for a Cartesian space move is one. Since three

[24] The reason for doing this is so that we can easily pass the whole packet through the Simulink model, which requires vectors all having the same data type.

bits are designated for the control mode, up to eight modes can be defined in total. Finally, the position mask is only used in Cartesian space control, and defines which of x, y, or z will be controlled. The Cartesian space controller currently requires that all three orientation parameters be controlled, which allows one position parameter to be controlled by the final degree of freedom. The theory behind this is described in Section 2.3.3.

For a joint space move, the control mode is zero, and the rest of the bits are unused. Therefore, the first double value in the packet can always be set to zero. For a Cartesian space move, the first double value must be either 9 (01 001), 17 (10 001), or 25 (11 001), to control either the x, y, or z position, respectively.

The last three parameters in the packet are each double vectors of length four, corresponding to the desired position, acceleration, and maximum velocity of the move. For a joint space move, these values correspond to joint space parameters in rad, rad/s, and rad/s^2, respectively. For a Cartesian space move, this is slightly more complicated. The position vector uses all four elements, where the first is the desired position in base coordinates along the selected axis (x, y, or z), in units of meters. The final three doubles contain the axis-angle representation, $\hat{K}\theta$, for the desired tool orientation represented in base coordinates. For the velocity and acceleration elements, only two doubles are required for each vector. The first double represents the velocity and acceleration along the selected tool position axis, in m/s and m/s^2, respectively. The second double in each set represents the rotational velocity ($\dot{\theta}$) and acceleration ($\ddot{\theta}$) about the axis defined by the axis-angle representation. The final two elements are unused, and should be set to zero in order for the trapezoidal profile to work properly (see the Simulink model for details).

B.4.2 Arm Feedback UDP Packet

A UDP feedback packet is sent from the xPC Target machine to the MATLAB machine on each control cycle, which essentially contains a snapshot of all current controller values. This is not only useful for recording data, but also for determining what high-level control commands should be executed next. The packet data contains one unsigned 32-bit integer and $7(4) = 28$ doubles; therefore, the total payload

is $4 + 28(8) = 228$ bytes as shown in Table B.5.

num (1 int)	wdes (4)	wddes (4)	qdes (4)	qddes (4)	q (4)	qd (4)	tau (4)

Table B.5: This table shows the data format for the UDP packet containing feedback from the arm. It consists of one unsigned 32-bit integer and $7(4) = 28$ doubles, and therefore has a $4{+}28(8) = 228$ byte payload.

The format of this packet is naturally quite straightforward. The first four bytes contain an unsigned 32-bit integer 'block number'. This simply contains the current cycle count of the control loop, starting at one. It is useful for ensuring that the UDP packets are received in the correct order. Using a 32-bit integer is sufficient here, and a simple calculation shows the controller would need to be running for about 99 straight days before the integer overflows[25]. The remaining seven fields are each vectors of length four, containing the following values, respectively: desired Cartesian space positions and velocities (for the four selected fields out of six), desired joint space positions and velocities, current joint positions and velocities, and current applied joint torques. If the controller is not in Cartesian space mode, all of the fields for Cartesian space position and orientation will contain NaN (not a number).

B.4.3 Hand Command UDP Packet

The hand command UDP packet is similar in function to the arm command UDP packet, but is formatted in a much different way. Since we have not yet developed code to interact with the real-time hand controller, we instead allow the MATLAB experiment code to send UDP packets containing raw data to be sent over the serial port. This provides all of the functionality needed to execute joint space control with a trapezoidal profile, but is thought of as a temporary solution until the real-time controller is complete. More details regarding the current implementation of the hand controller are available in Section 2.6.

As shown in Table B.6, the data simply consists of a 32-byte message to be

[25]This is calculated by $2^{32}\ bits\ /\ 500\ Hz\ /\ 60\ \frac{sec}{min}\ /\ 60\ \frac{min}{hr}\ /\ 24\ \frac{hr}{day} \approx 99.42 days$.

command (31 bytes)	13 (1 byte)

Table B.6: This table shows the data format for the UDP packet containing raw data to be sent to the hand over the serial port. It consists of 32 ASCII characters, where the last is the carriage return.

sent over the serial port. The final byte must be the carriage return character (13), as this signifies the end of a command in supervisory control mode. Specific details on these serial commands can be found in [32], which is posted on the lab website.

B.5 Known Issues

In this section, we discuss known issues with in the controller design and Barrett hardware operation.

B.5.1 Controller Issues

There are two known issues with the controller for the arm. It is possible that both may be fixed in future work.

- Sample Rate – The sample rate of the arm controller is currently limited to about 500 Hz (this is also the default sample rate at which the Barrett C code runs). If the sample rate is increased to 1000 Hz via the Simulink software, then the WAM pendant produces a heartbeat failure, typically within the 60 seconds of execution. Sample rates between 500 Hz and 1000 Hz have not been tested. This problem began happening after Barrett replaced the electronics board in the base of the WAM, as part of an upgrade from optical to magnetic encoders. Prior to that upgrade, the controller ran well up to about 1250 Hz, and the heartbeat failure never occurred. The only limitation previously was code performance, and pushing the sample rate any higher than about 1250 Hz would cause a 'CPU overloaded' error on the xPC Target machine. Barrett technical support was unsure why we were having this issue and said that while they typically run the WAM at 500 Hz, we should be able to increase sample rate further without any problem.

- Velocity Feedback – Since the WAM does not contain tachometers in its joints, we rely on discrete differentiation to obtain a velocity signal. As described in Section 2.1.4, this approach yields a very noisy velocity signal and seems to be causing joint vibrations. Some sort of filtering method (such as a Kalman filter) may improve this in the future.

B.5.2 Barrett Hardware Issues

In this section, we list several oddities we have encountered in the Barrett hardware.

- Static Shock – When the arm is powered on and running, many times it releases a large static shock. This happens, for example, when the arm is in gravity compensation mode and is being moved around by the user. This has been an ongoing problem, but while Barrett technical support said it was unusual, they did not offer an explanation on how to fix it.

- High Friction – The Barrett WAM is known for having very low friction, but we have noticed relatively large, position-dependent friction in our particular WAM. Barrett looked into this issue and was able to duplicate our findings, but said they found nothing out of the ordinary with our hardware. They mentioned that they do not collect friction data and therefore had little to compare to, but would expect about 0.8-1.0 Nm of friction on joints 1, 2, and 4, with slightly higher friction on joint 3. They said it is possible we need to replace the ball bearings or other equipment, but this would be very difficult to diagnose.

- Resistive Braking – The WAM has resistive braking in its joints as a safety precaution, but we have noticed this is not always activated as expected, leaving the joints of the arm very loose. In fact, when the resistive braking is off, the arm has noticeably lower resistance, but we are unsure if this is related to the friction issue above.

B.5.3 Barrett Hardware Operation

In this section, we list several known issues in operating the WAM, most of which are also described in detail in the Barrett manuals.

- Shift Idle – When the arm is 'Shift-Idled' (hold 'Shift' on the pendant, then press 'Idle'), it is very important that it be in the home position. Shift-Idling takes place when the arm is first powered on or after a pendant fault occurs during operation. It essentially checks for any errors and sets the current encoder positions as the 'zero position'. In the Simulink code, we add an offset in radians to describe the actual angular position of the arm when it is in home position[26]. If the arm is Shift-Idled somewhere other than the home position, the computed angular positions will be incorrect, and the applied torques could be very erratic.

- Hand Firmware – The hand firmware resides in a RAM chip inside the hand, which loses its charge after about two days of being powered off. It is therefore convenient to leave the hand powered on at all times. If it is ever left off for more than about two days, the firmware will need to be reloaded. In addition, each time the firmware is loaded, a 'hand initialize' command (HI) must be sent over the serial port before the motors will work properly. The initialization procedure moves the fingers through a set of pre-configured movements, and strange finger vibrations should be expected during this process.

- Cable Tensioning – After repeated use of the arm, the steel cables that move its joints will begin to stretch out and will need to be re-tensioned. Barrett has C code to do this automatically, and the instructions for this can be found on their website or in documentation manuals. The auto-tensioning procedure applies an appropriate amount of torque in the cables, but is incapable of decreasing cable tension. After looking at technical documents on the auto-tensioning mechanism, one will see why cable tension cannot be released without re-cabling the joints.

[26]The actual zero position is where the arm points straight up at the ceiling, while home position is where it folded up and resting on the rubber joint stop.

B.6 List of Repairs

The following is a list of all known repairs since our Barrett WAM and Hand were purchased in 2005:

- October 2005 – The refurbished arm and hand were purchased from Barrett. Order information for the arm includes: auto-tensioners have new leaf-spring design; original puck molds; low voltage operation only; revision 1 pendants.
- February 2008 – Motor 4 of the arm was moved over motor 2. This was a design change implemented in newer WAMs.
- July 2010 – Barrett investigated high friction in the arm and suggested ball bearings might need replacing (but these were not replaced). Some of the electronics failed while they had it, so the main electronics board was replaced, optical encoders were removed, and magnetic encoders were installed. New electronics boards were also installed in the pendants, and the WAM was re-cabled. The hand was re-tensioned, and more lubricant was applied to gears in the hand.
- December 2010 – Strain gauges were installed in the hand. The motor for finger 2 failed shortly after this upgrade, so the hand was sent back to Barrett and the finger was repaired.

APPENDIX C
MATLAB CODE

C.1 Symbolic Matrix Calculations

This function can be used to symbolically compute the forward kinematics, homogeneous transformation matrices, Jacobian, and gravity loading terms. The full equations can then be inserted into the dynamic model for the controller.

```
% computeWamVars.m
% Computes symbolic forward kinematics, Jacobian, and gravity terms

% Define symbolic variables
syms a3 a4 d3 d5 t1 t2 t3 t4 t5;
syms c1x c1y c1z c2x c2y c2z c3x c3y c3z c4x c4y c4z c5x c5y c5z;
syms m1 m2 m3 m4 m5;

% Denavit-Hartenberg parameters
a = [0 0 a3 a4 0];
d = [0 0 d3 0 d5];
t = [t1 t2 t3 t4 0];
b = [-pi/2 pi/2 -pi/2 pi/2 0];

% Center-of-Masses
c = [c1x c2x c3x c4x c5x;
     c1y c2y c3y c4y c5y;
     c1z c2z c3z c4z c5z;
     1 1 1 1 1];
cs = [c1x c2x c3x c4x c5x c1y c2y c3y c4y c5y c1z c2z c3z c4z c5z];

% Masses
m = [m1 m2 m3 m4 m5];  % m5 = mass of hand
```

```
% Define actual variables
a_act = [0 0 .045 -.045 0];
d_act = [0 0 .55 0 .44];  % d_act(5) = 0.35, if no hand

c_act = zeros(3,5);
c_act(:,1) = 1e-3*[  0.3506  132.6795   0.6286];
c_act(:,2) = 1e-3*[  0.2230  -21.3924  13.3754];
c_act(:,3) = 1e-3*[-38.7565  217.9078   0.0252];
c_act(:,4) = 1e-3*[ 11.7534   -0.1092 135.9144];
c_act(:,5) = [0 0 -0.04];

m_act = [8.3936 4.8487 1.7251 1.0912 1.2];  % m_act(5) = 0, if no hand

g = 9.81;

% Create symbolic matrices
T = sym(zeros(4,4,5));   % T(k to k-1)
Tb = sym(zeros(4,4,5));  % T(k to base)
cf = sym(zeros(3,5));    % Center of masses in base frame
A = sym(zeros(3,5,5));   % Partial differentials of center of masses

% Iterate through links
for i=1:5
    % Avoid tiny errors in symbolic math
    if (abs(b(i)) == pi/2)
        cb = 0;
    else
        cb = 1;
    end
```

```
    if (b(i) == pi/2)
        sb = 1;
    elseif (b(i) == -pi/2)
        sb = -1;
    else
        sb = 0;
    end

    % Compute forward kinematics
    T(:,:,i) = [ cos(t(i)), -sin(t(i))*cb,  sin(t(i))*sb, a(i)*cos(t(i));
                 sin(t(i)),  cos(t(i))*cb, -cos(t(i))*sb, a(i)*sin(t(i));
                 0, sb, cb, d(i);
                 0, 0, 0, 1 ];
    if (i==1)
        Tb(:,:,i) = T(:,:,i);
    else
        Tb(:,:,i) = Tb(:,:,i-1)*T(:,:,i);
    end

    % Convert center of mass to base frame
    cf(:,i) = Tb(1:3,:,i)*c(:,i);
    for j=1:5
        A(:,j,i) = diff(cf(:,i),t(j));
    end
end

% Compute gravity load on each joint
h = sym(zeros(4,1));
for i=1:4
    for j=i:5
        h(i) = h(i) + g*m(j)*A(3,i,j);
```

```
    end
end

% Compute symbolic tool position and Jacobian
upos = Tb(1:3,4,5);
V = [[diff(upos,t1) diff(upos,t2) diff(upos,t3) diff(upos,t4)];
     [0; 0; 1] squeeze(Tb(1:3,3,1:3))];

% Substitute in actual variables
usubs = subs(upos,[a d],{[a_act d_act]});
Vsubs = subs(V,[a d],{[a_act d_act]});

Tsubs = subs(T,[a d],{[a_act d_act]});

hsubs = subs(h,cs,{c_act});
hsubs = subs(hsubs,[a d m],{[a_act d_act m_act]});

% Display computed matrices
digits(4);
vpa(Tb(:,:,5))
vpa(Vsubs)
vpa(h)
```

C.2 Maximum Inertia Calculations

This function can be used to compute the maximum inertia on each joint. It then uses the method described in Section 2.4.2 to calculate PID gains, given the desired K_p value for joint 1.

```
% calc_jmax.m
% Computes maximum inertia and PID gains for each joint

function jmax = calc_jmax(kp1)
```

```
% Configurations of maximum inertia for each of the four joints
qs = [0 -1.57 0    0 0 ;
      0 -1.57 0    0 0 ;
      0 -1.57 0 1.57 0 ;
      0 -1.57 0    0 0];

% Denavit-Hartenberg parameters
a = [ 0 0 .045 -.045 0 ];
d = [ 0 0 .55 0 .44 ];  % d(5) = 0.35, if no hand
b = [ -pi/2 pi/2 -pi/2 pi/2 0 ];

% Mass of each link
m = [8.3936 4.8487 1.7251 1.0912 1.2];  % m(5) = 0, if no hand

% COM of each link in its own Lk
c = zeros(3,5);
c(:,1) = 1e-3*[  0.3506  132.6795   0.6286];
c(:,2) = 1e-3*[  0.2230  -21.3924  13.3754];
c(:,3) = 1e-3*[-38.7565  217.9078   0.0252];
c(:,4) = 1e-3*[ 11.7534   -0.1092 135.9144];
c(:,5) = [0 0 -0.04];

% Link Inertia of link k about its COM, and in Lk
Db = zeros(3,3,5);
Db(:,:,1) = 1e-6*[95157.4294    246.1404    -95.0183;
                    246.1404  92032.3524   -962.6725;
                    -95.0183   -962.6725  59290.5997];
Db(:,:,2) = 1e-6*[29326.8098    -43.3994   -129.2942;
                    -43.3994  20781.5826   1348.6924;
                   -129.2942   1348.6924  22807.3271];
Db(:,:,3) = 1e-6*[56662.2970  -2321.6892      8.2125;
```

```
                  -2321.6892   3158.0509    -16.6307;
                      8.2125    -16.6307  56806.6024];
Db(:,:,4) = 1e-6*[18890.7885     -0.8092  -1721.2915;
                     -0.8092  19340.5969     17.8241;
                  -1721.2915     17.8241   2026.8453];
Db(:,:,5) = diag([0.006 0.006 0.006]);  % estimation assuming sphere

% Transformation Matrices from Lk to Lk-1
T = zeros(4,4,5);

% Link Inertia of link k about its COM, and in L0
Df = zeros(3,3,5);

fprintf('\n');

% Apply Parallel Axis Theorem to get Df
% Df is the inertia of frame k about the origin of frame k
for k=(1:1:5)
    Df(:,:,k) = parallelAxisTheorem(Db(:,:,k), m(k), c(:,k));
end

% Motor inertia is negligible
% For cylinder, Jzz = (mr^2)/4
% m = 0.044, r = 0.0175
% Jm = (3.36875e-6)*ones(1,4);

% Build gear reduction matrix
% N = [42 28.25 28.25 -18];
% n3 = 1.68;
%
% M = [ -N(1)     0        0    0 ;
```

```
%           0  N(2) -N(2)/n3    0 ;
%           0 -N(2) -N(2)/n3    0 ;
%           0     0        0 N(4)];

% Max inertia for each link
%jmax = (M.^2)*Jm';

jmax = zeros(1,4);

% Calculate for each link
for link=(1:1:4)
    q = qs(link,:);

    TT = eye(4);
    J = zeros(3,3);

    for k=(link:1:5)
        cb = cos(b(k));
        sb = sin(b(k));

        % Compute forward kinematics, T(k to k-1)
        Tk = [ cos(t(k)), -sin(t(k))*cb,  sin(t(k))*sb, a(k)*cos(t(k));
               sin(t(k)),  cos(t(k))*cb, -cos(t(k))*sb, a(k)*sin(t(k));
               0, sb, cb, d(k);
               0, 0, 0, 1 ];
        T(:,:,k) = Tk;

        % Compute T(k to link-1)
        TT = TT*T(:,:,k);

        % Similarity transform from frame k to frame link-1 coords
```

```
        Dl = TT(1:3,1:3)*Df(:,:,k)*TT(1:3,1:3)';

        % Apply Parallel Axis Theorem to move inertia to frame link-1
        Dl = parallelAxisTheorem(Dl, m(k), TT(1:3,4));

        % Add the inertia from frame k
        J = J+Dl;
    end

    % We want Izz
    jmax(link) = jmax(link) + J(3,3);

    fprintf('%10.4f kg/mm^2\n',jmax(link)*10^6);
end

% Get the gains
[kp, kd] = getGains(kp1,jmax);

% Print out proportional gains
fprintf('\nKp Values\n');

for i=(1:4)
    fprintf('Joint %d => %10.4f\n', i, kp(i));
end

% Print out derivative gains
fprintf('\nKd Values\n');

for i=(1:4)
    fprintf('Joint %d => %10.4f\n', i, kd(i));
end
```

```
    fprintf('\n');
end


% Compute Kd given Kp and the inertia
function [Kd] = getKd(Kp, Jmax)
    Kd = (4*Jmax*Kp)^(1/2);
end


% Compute the gains
function [Kp,Kd] = getGains(Kp1,Jmax)
    wn = sqrt(Kp1/Jmax(1));
    fprintf('\nwn = %10.4f\n', wn);

    Kp = Jmax*wn^2;
    Kd = (4*Kp.*Jmax).^(1/2);
end


% Transform inertia using parallel axis theorem
function [Dout] = parallelAxisTheorem(Din, m, p)
    x = p(1);
    y = p(2);
    z = p(3);

    Dout = Din + m*[y^2+z^2, x*y, x*z ;
                    x*y, x^2+z^2, y*z ;
                    x*z, y*z, x^2+y^2];
end
```

C.3 Cartesian Space Simulation

This function can be used to simulate any Cartesian space move of the arm using the Jacobian. It is useful to run this simulation prior to sending commands to the actual arm. In doing so, one can be sure that there is no numerical instability in the trajectory and the arm will accurately reach its desired position.

```
% jacobSim.m
% Simulates Cartesian space trajectory given any desired move

computeWamVars;

% T(tool to base) computation
computeT = @(q)  % Insert symbolic T matrix here

% Jacobian Computation
computeV = @(q)  % Insert symbolic J matrix here

% Set parameters for Cartesian space move
run_time = 1;
sample_time = .002;
steps = run_time/sample_time;
q = [-pi/4 -pi/2 pi/2 3*pi/4]';
mask = [1 4:6];

% Compute initial T matrix
Tstart = computeT(q);

% Define translation and rotation for the move
xdist = 0.2;
Rstart = Tstart(1:3,1:3);
Rend = Rstart*getEuler2Rot(deg2rad([20 5 -10]));
```

```
% Compute tool configuration vectors
wstart = [Tstart(1:3,4); zeros(3,1)];
wdes = [Tstart(1:3,4)+[xdist 0 0]';
        Rstart*getRot2AxisAngle(Rstart'*Rend)];

% Compute tool velocity and desired T matrix
wdot = (wdes-wstart)./run_time;
Tdes = [Rend wdes(1:3); 0 0 0 1];

% Prepare simulation variables
wdot = wdot(mask);
traj = zeros(6,steps);

% Simulate for run_time
for i=1:steps
    % Compute corresponding joint velocities for this step
    Vcurr = computeV(q);
    qdot = pinv(Vcurr(mask,:))*wdot;

    % Step joint position
    q = q + qdot*sample_time;

    % Compute new T(tool to base) and extract via point
    Tcurr = computeT(q);
    traj(:,i) = [Tcurr(1:3,4);
                 Rstart*getRot2AxisAngle(Rstart'*Tcurr(1:3,1:3))];
end

% Compute the final T matrix and tool configuration vector
Tend = computeT(q);
wend = traj(:,end);
```

```
% Display simulation variables
Tstart
Tdes
Tend
wstart = wstart(mask)
wdes = wdes(mask)
wend = wend(mask)


% Compute rotation matrix given axis-angle vector
function [ R ] = getAxisAngle2Rot(w)
    thK = sqrt(sum(w.^2));
    if(abs(thK) < 1e-3 || abs(pi-thK) < 1e-3)
        thK = 0;
        K = [0 0 1];
    else
        K = w./thK;
    end

    Kx = K(1);
    Ky = K(2);
    Kz = K(3);

    ct = cos(thK);
    st = sin(thK);
    vt = 1-cos(thK);

    R = [Kx*Kx*vt+ct, Kx*Ky*vt-Kz*st, Kx*Kz*vt+Ky*st;
         Kx*Ky*vt+Kz*st, Ky*Ky*vt+ct, Ky*Kz*vt-Kx*st;
         Kx*Kz*vt-Ky*st, Ky*Kz*vt+Kx*st, Kz*Kz*vt+ct];
```

```
end


% Compute axis-angle vector given rotation matrix
function [w] = getRot2AxisAngle(R)
    theta = acos((R(1,1)+R(2,2)+R(3,3)-1)/2);
    if(abs(theta) < 1e-3 || abs(pi-theta) < 1e-3)
        % Doesn't matter which axis we pick
        K = [0 0 1]';
        theta = 0;
    else
        K = [R(3,2)-R(2,3); R(1,3)-R(3,1); R(2,1)-R(1,2)]./(2*sin(theta));
    end

    w = K*theta;
end


% Compute rotation matrix given Euler angles
% z-y-x notation (yaw(z), then pitch(y), then roll(x))
function R = getEuler2Rot(ypr)
    R = [ cos(ypr(2))*cos(ypr(1)), ...
         -cos(ypr(3))*sin(ypr(1))+sin(ypr(3))*sin(ypr(2))*cos(ypr(1)), ...
          sin(ypr(3))*sin(ypr(1))+cos(ypr(3))*sin(ypr(2))*cos(ypr(1)); ...
          cos(ypr(2))*sin(ypr(1)), ...
          cos(ypr(3))*cos(ypr(1))+sin(ypr(3))*sin(ypr(2))*sin(ypr(1)), ...
         -sin(ypr(3))*cos(ypr(1))+cos(ypr(3))*sin(ypr(2))*sin(ypr(1)); ...
         -sin(ypr(2)), ...
          sin(ypr(3))*cos(ypr(2)), ...
          cos(ypr(3))*cos(ypr(2))];
end
```