THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

CONNECTING VIRTUAL ROBOTICS TO AN EXPERIMENTAL PLATFORM

by
Callum Rohweder

School of Information Technology and Electrical Engineering,
The University of Queensland.

Submitted for the degree of
Bachelor of Engineering
in the field of Mechatronics

June & 2018.

52 KENNIGO STREET
SPRING HILL, QLD, 4000
Tel. 0404 639 174

June 8, 2018

Professor Michael Bruenig
Head of School
School of Information Technology and Electrical Engineering
The University of Queensland
St Lucia, Q 4072

Dear Professor Michael Bruenig,

In accordance with the requirements of the degree of Bachelor of Engineering in the division of Mechatronic Engineering, I present the following thesis entitled 'Connecting Virtual Robotics To An Experimental Platform'. This work was performed under the supervision of Dr Surya Singh.

I declare that the work submitted in this thesis is my own, except as acknowledged in the text and footnotes, and has not been previously submitted for a degree at The University of Queensland or any other institution.

Yours sincerely,

Callum Rohweder.

Acknowledgments

I specifically would like to thank my fellow students in the school of ITEE for assisting me during this project; for listening to the software issues I faced and giving useful insight and direction.

The product developed for this thesis was created in complete self-sufficiency, with functionality specified by my supervisor.

Abstract

Simulation platforms give the ability to test the movements and operations of robotic manipulators without causing damage to the device or surrounding objects. It was proposed that a simulation environment, V-REP, could be used to simulate the realistic movements of the Kinova Jaco arm. Thus, creating the ability to detect and correct for collisions and undesirable arm configurations, where-after, the movements can be passed on to the physical Jaco arm to move without error.

This thesis looks into the development of a program that uses the V-REP remote API to control the robotic arm. V-REP and the created program offer different methods of controlling the movement of the Jaco arm's gripper, where the background and integration of these methods are presented. The inputs to the developed program can other be from an external gaming controller, or keyboard entries, where the commands change the position of the end-effector in small increments. A forward kinematics mode for the program was also developed, so the used can individually change the angles of joints.

It was found that V-REP's inverse kinematis solver set to Damped Least Squares was the superior method of moving the end-effector of the Jaco arm compared to V-REP's Pseudo Jacobian method; as it was able to smoothly move the end-effector in less iterations. Additionally, the created program came with a forward kinematics mode, and three inverse kinematics modes; inverse using relationships between joints, a control approach to reducing the error in end-effector position by changing the joint angles, and calculating the inverse Pseudo Jacobian. Due to design errors in integrating latter two in to the developed program, the analytical approach was proven to be the most successful. This was because it was able to correctly move the end-effector given commands from keyboard or joystick, in real-time.

Contents

Acknowledgments **v**

Abstract **vii**

List of Figures **xii**

List of Tables **xiii**

1 Introduction **1**

2 Programming Robots to Move **5**

2.1 MATLAB Toolboxes 5

2.1.1 Robotics Toolbox 5

2.1.2 Robotic Arm Models 7

2.1.3 Adaptive Neuro-Fuzzy Inference System 9

2.2 Programmed Robotic Arms 10

2.3 Simulation Platforms and Physics Engines 13

2.3.1 V-REP 13

2.3.2 Alternative Simulation Tools 18

3 Theory **20**

3.1 Child Processes and Multi-Threading 20

3.2 Regulators 22

3.3 Forward Kinematics 24

3.4 Inverse Kinematics 27

3.4.1 By Matrix Manipulation Approach 27

3.4.2 Analytical Approach 28

3.4.3 Jacobian 30

3.4.4 Damped Least Squares 32

3.4.5 Newton-Raphson Method 33

4 Design Approach **36**
4.1 Client Program Functionality Requirements and Design 36
4.2 Project Development 38
4.2.1 Set Up 38
4.2.2 Forward Kinematics Implementation 40
4.2.3 V-REP Inverse Kinematics 42
4.2.4 Joystick Integration 45

5 Kinematics Approach **48**
5.1 Forward Kinematics 48
5.2 Inverse my Matrix Manipulation 48
5.3 Inverse by Analytical Approach 49
5.4 Inverse by Control Loops 55
5.5 Inverse by Jacobian 60

6 Results Summary and Discussion **64**
6.1 Jaco Arm Control 64
6.2 Joystick Implementation and Processing Time 66
6.3 Client Program Inverse Kinematics 67
6.4 Client Program Jacobian 68
6.5 Kinova SDK 69
6.6 Data Streaming Implementation 69
6.7 Communication 70

7 Conclusions **72**
7.1 Summary and Conclusions 72
7.2 Developed Files 73
7.3 Possible Future Work 74

Appendices **76**

A Tables and Figures **77**

B Additional Findings **86**
B.1 Joystick Applications 86
B.1.1 Objectives 86
B.1.2 Windows API 86
B.1.3 HTML5 Gamepad API 87
B.1.4 Libusb and xusb.c 87
B.1.5 Simple Direct Layer (SDL) Source Code 88

B.2 Alternate Inverse Kinematics Solutions 88

C Program Listings **90**

C.1 V-REP Client Program 90

C.2 SDL 90

C.3 MATLAB 91

C.4 Additional Files on Disk 91

Bibliography **93**

List of Figures

1.1 Flow diagram describing the key features of this thesis 4

2.1 Motor Torque Control System 6
2.2 smrobot Simscape Model 7
2.3 smrobot Simscape Model Visualisation 8
2.4 Simulink Model of Motor Control Loop 8
2.5 Simulink Model of Motor Controllers 9
2.6 Flow Diagram of using Jacobian in Programming 12
2.7 Flow Diagram of the V-REP Simulation 14
2.8 Sample Lua Code for Connecting V-REP to a port 15

3.1 Feedback Control System with Disturbance 22
3.2 Physical representation of D-H Parameters between two links 25
3.3 Revolute Joint Arm 28
3.4 The Newton-Raphson Method applied to Inverse Kinematics 34

4.1 The Jaco arm in V-REP with joints set to IK mode, with a small error in target to tip position 43
4.2 Left: the Jaco arm using the Pseudo Inverse Jacobian with 50 iterations. Right: the Jaco arm using DLS of damping 0.05 and 5 iterations 44

5.1 Dimensions and angles of Jaco Arm 49
5.2 Simplification of Jaco arm geometry 50
5.3 Simplification of Jaco arm wrist geometry 51
5.4 Control loop for approximating the joints angles to move the end-effector using the position of the fourth joint 53
5.5 Response of Chosen Controller Given a Change in End-Effector Position Command (3mm in the x-y plane) 54
5.6 Error being corrected in control loop 56
5.7 The desired and actual positions of the end-effector with each joint under control 57
5.8 The desired and actual angular positions of the end-effector 58

5.9 Standard Regulator Block Diagram 58
5.10 Block Diagram of integral action, giving the output (y) a reference input (r) 59
5.11 Block Diagram of a Standard Cascade Regulator 59
5.12 Starting position of the Jaco arm when giving IK inputs to a FK scene 62

6.1 Client Program on Start-Up 65
6.2 Client Program given incorrect arguments on start-up 66

A.1 Simple V-REP client program in Python Flow Chart 77
A.2 V-REP Client Makefile 78
A.3 Lua function in V-REP main script, for retrieving object names . . . 78
A.4 Example of a text file created by the V-REP client 79
A.5 Client program in Forward Kinematics mode 80
A.6 D-H Parameters axis for each joint on Jaco arm 81
A.7 An example of the terminal output of the joystick process when the left joystick is toggled on 82
A.8 The calculation method for Inverse Kinematics in V-REP 83
A.9 V-REP Client Program Control Loop Error 84
A.10 Plot of using Jacobian Approximation to move the end-effector to a desired position 85

List of Tables

2.1 V-REP remote API operation modes 16

4.1 Classical D-H Parameters of the Jaco Arm 41

A.1 Client program IK input commands 81
A.2 Client program IK input commands 82
A.3 Chosen controller coefficients for each joint 83

Chapter 1

Introduction

Remotely controlling a robot or robotic manipulator is a desirable objective with large complications to still be overcome. It is proposed that doing this through a virtual environment can eliminate collisions or undesirable actions that may incidentally occur due to the nature of long distance control, whilst providing the benefits of virtual simulations. Specifically, this thesis focuses on the development a remote interface to a 'Virtual Robotics Environment Platform', VREP. VREP can be used to simulate robotic arms and processes, and includes the commonly used wheel-chair based Kinova Jaco arm which is available at the University of Queensland. The remote interface, otherwise known as the 'client program' (given that it treats VREP as the server), was designed to take input from a user by either keyboard or joystick, and have the motion played out in VREP, corrected for any obstructions, and then control the movement of the Jaco arm through the Kinova Software Development Kit.

In all engineering disciplines, virtual simulations allow one to view and interact with an environment or process in a non-destructive manner. Simulations can provide a realistic rendering of an event, whilst providing further detail into physical phenomena that establish design constraints, and optimization techniques. Robotics makes use of lumped electromechanical components to interact with an environment in a desirable manner. Thus, virtual robotics is a necessary field for growth in engineering, as it allows the testing of interactions with an environment whilst giving unforeseen insight.

Companies such as those in manufacturing, technical experts in the fields of medicine and surgery, and persons with disabilities all benefit from the capability of robotic manipulators. In most circumstances it is expected that these manipulators can be controlled by a user in real time, however this ability is restricted by inherent delay in the process of receiving an input, calculating an action, and actuating. Further expectations of robotics include optimality and customisation; where it may be desirable for movements of a robot to minimize the energy used in a given

process or a robotic arm to pick up a glass a in a certain manner. This provides interconnected layers of desired functionality for a robot; a layer dedicated to moving the manipulator, a layer designed to create movements that meet the user's needs whilst minimizing design criteria, and a monitoring layer that focuses on aspects such as physical constraints and robotic learning.

With the invention of the internet to provide long range data-resourcing, came a desire to move the control of manipulators and processes in a remote location. The concept of remote robotics is no different to the typical method of controlling manipulators, an input has a desired output; however at some time in between, data is processed and sent through the internet. Given factors such as time to send, packet loss, processing time, and internet traffic, a significant amount of undesirable delay is added and decreases the satisfaction of real-time control. This produces large complications in areas such as remote robotic surgery, where reaction delay may have harmful effects.

Virtual robotic environments can simulate the true movements of a manipulator given its physical attributes. VREP in particular can calculate the joint angles required to be able to move the hand from one position to another using an inverse kinematics method and accuracy of choice. It is believed that allowing a virtual environment to compute the movements required by a remote user will increase the accuracy in movement and speed in calculating movements.

Although this thesis does not go into large detail on the testing of the complete product, remote user to physical robotic arm movement, it does go through the design of the remote interface. The problems faced in controlling a robotic arm, design strategies tested, and the future improvements of the client program are presented. This interface was crucial to the success of the whole system, and it was important to refine before interfacing with a physical arm. Included in this document is:

- A comparison of different approaches for moving a robotic arm, along with the calculations and methods for refining accuracy and decreasing processing time.
- An overview of VREP, and how its features were used to replicate the true movements of the Jaco arm. This includes a comparison of Inverse Kinematics (IK) functionality.
- Validity of using VREP as a real-time simulation tool, looking into aspects such as rendering delay and communication techniques.
- The features included in the client program and how its functionality was shaped, and the limitations associated with a real-time program performing

large amounts of calculations based off user inputs and a remote server.

- An overview of the Jaco Kinova arm and it's software development kit, which can be used to control the Jaco arm from VREP or the client program.

To accomplish the task of remotely controlling the Jaco arm with collision avoidance, the project was broken into four main stages. The long-term objectives to achieve the aforementioned goals are:

- To configure the external simulation control of VREP and the Jaco arm within. This included retrieving all of the arm's joint information on start-up, so the correct joints can be referenced during connection. The central purpose of this objective was to achieve control of the arm in simulation, getting it to move its gripper to desired coordinates.

- Using the arm'ss information in VREP, and the current joint angles from the physical Jaco arm using the SDK, achieve movement replication between the the physical and virtual systems. The purpose of this was that it would allow tuning of the VREP simulation to mimic the way the physical arm moves and responds to commands.

- With VREP tuned appropriately, allow the movement of the physical arm by controlling movement in the simulation. It was predicted that movement in the simulation would come from keyboard commands or on-screen mouse presses.

- Detect collision, move objects in the laboratory and interact with the environment in simulation, with the Jaco arm replicating the motion.

Due to unforeseen circumstances, the objectives presented in this project were changed to focus on the different ways of interacting with the virtual arm through VREP's remote API. The primary focus is to test the validity of using a remote client to interact with VREP, and whether a good interface with the aforementioned requirements can be created. The objectives of this thesis were to:

- Create a user friendly interface for controlling a robotic arm in VREP, robust to arm location within a scene and mode of the virtual arm (forward kinematics or inverse kinematics modes)

- Allow the user to give input types as they desire, such as moving the Jaco arm gripper from point to point, or controlling the joint angles individually; with keyboard input or joystick input.

- Experiment with different mathematical solutions and physics engines to achieve accurate and fast movement of the virtual robotic arm
- Experiment and outline the different functionality VREP has to offer, which may be utilised in the development of the over-arching objectives.

The required functionality of this project can be seen in the figure below. A user controls the VREP client program, which communicates with the virtual robotic arm for the interest of collision avoidance. From this, the client program can interface with the program designed to communicate with the Kinova API to control the physical arm. The movements of the physical arm are a mimic of the virtual arm's movements without collisions. This process is illustrated in Figure 1.1, below.

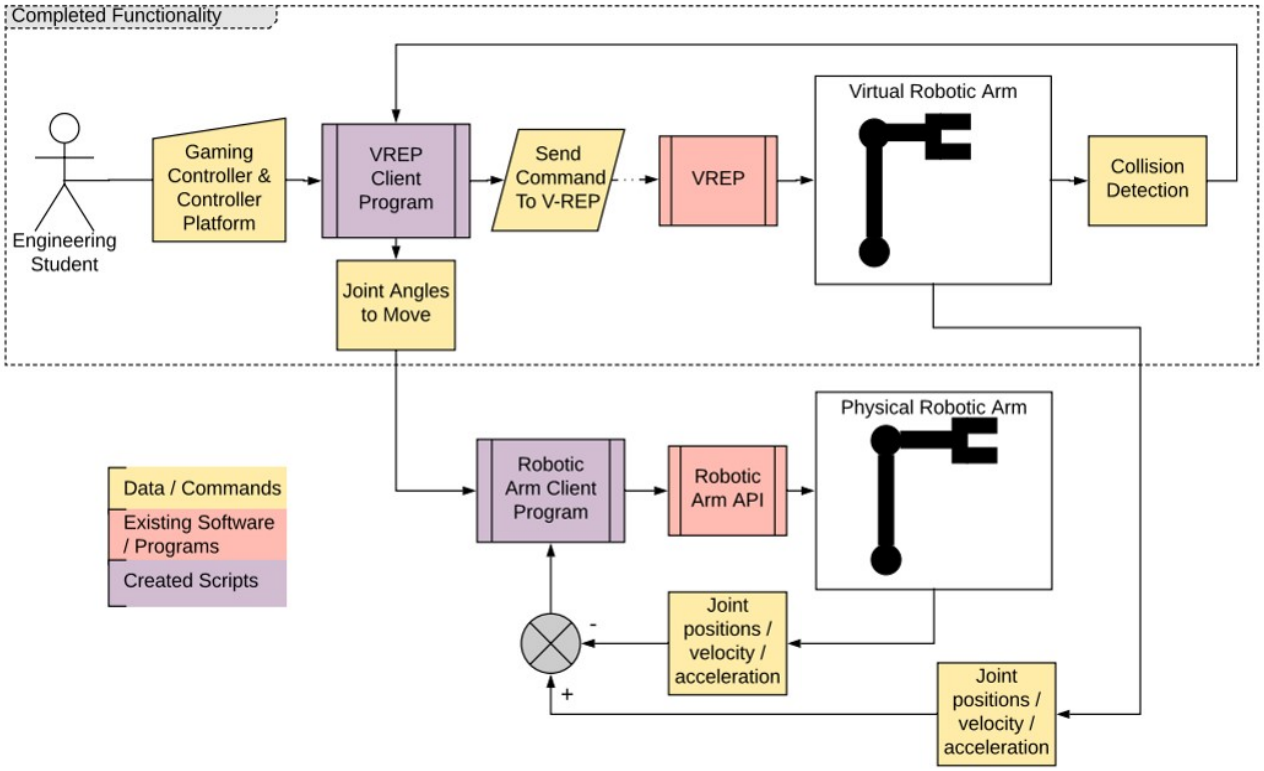

Figure 1.1: Flow diagram describing the key features of this thesis

From here, a variety of challenges will be pondered as mathematical formulas are derived to describe the motion of a 6 degree of freedom (DOF) arm, and a remote interface is created for allowing real-time movement within VREP.

Chapter 2

Programming Robots to Move

In order to move a robotic arm, two layers of control are required, the motor position control for each joint, and the system that depicts how much the joints need to change angle when moving from one arm configuration to another. Following are some case studies that present methods of controlling a robotic arm, where the maths behind them are left for the theory section of this report.

2.1 MATLAB Toolboxes

Matlab provides an extraordinary amount of support for calculating the important properties required for the control of a robotic manipulator. Below is a sample of some of the useful libraries available that were, and further could be further, beneficial to creating an interface for controlling the virtual Kinova Jaco arm remotely.

2.1.1 Robotics Toolbox

Professor Peter Corke, the Director of the Australian Centre for Robotic Vision (ACRV), has designed a MATLAB specific toolbox for computing the mechanics associated with robotic manipulators. Included is easy to use functions for creating robotic arms, and calculating parameters such as forward and inverse kinematics, trajectory generation and control systems for motor control. Within the Queensland University of Technology's Robot Academy is 'Master Classes' on using the toolbox through examples of simple manipulators and the theory behind the complex calculus being performed.

There are dozens of object files containing the parameters of robotic manipulators; including the 6 DOF puma560 and limited information on the 6DOF Kinova Jaco arm. The parameters within these files contain the link times of the arm, the Denavit-Hartenberg parameters (DH parameters - theory section of this report), and

can contain inertial and angle properties for each link. These properties are helpful in computing the forward kinematics, inverse kinematics, and dynamic equations that represent the motion and torques on the arm.

With this, the toolbox allows the viewing of the manipulator in desired angles, the movement of the arm between two desired end-effector positions, and the building of the control system controlling the torque at each motor given the current arm configuration. The latter of which is crucially important in providing accurate control of the arm, in ensuring the appropriate power is provided to all of the joint links as it moves over a given trajectory. Unfortunately, it is extremely difficult to compute, and for each degree of freedom contains 10 constants that need finding; mostly inertia terms which for each link depends on its orientation and position relative to the arm's base; thus relying on all link angles between the base and the link of interest. The mechanics of this is presented in the theory section of the report, but the Robotics Toolbox provides functions which compute the inertial, coriolis and gravitational torques acting on the arm. Professor Corke also provides a method of motor control for each link given the aforementioned parameters are known. The method uses the typical motor position control strategy of a Proportional Derivative Controller (details in the theory section) to compute the required motor torque from the error in angle position and velocity, however also includes a feed-forward component of the calculated disturbance torque due to the coupled dynamics between links, as presented in Figure 2.1.

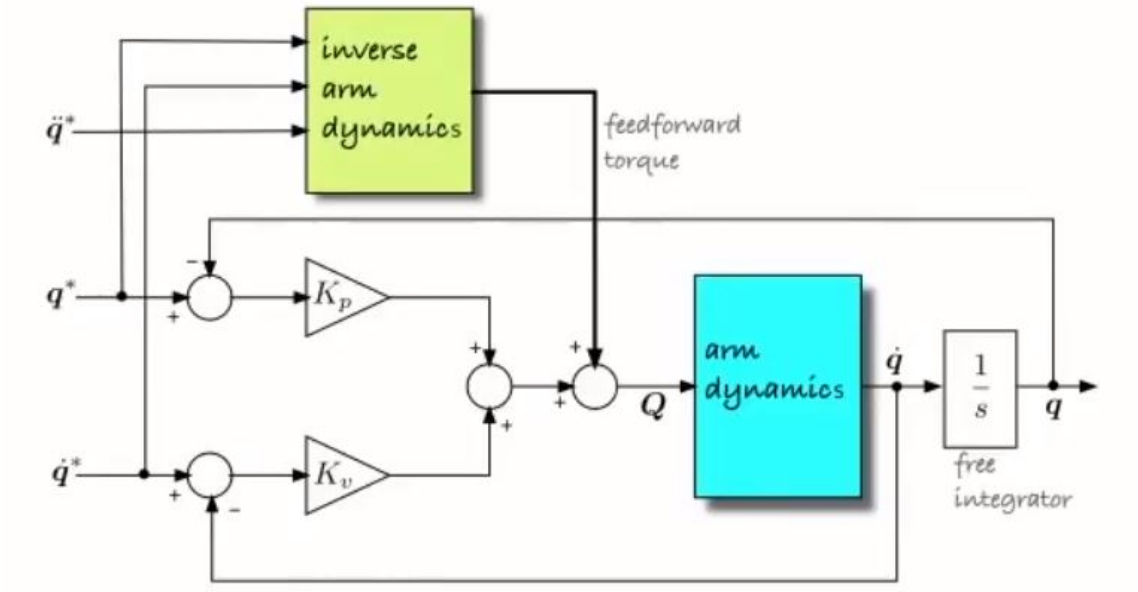

Figure 2.1: Motor Torque Control System

Thus the torque provided at each link can be expressed as:

$$\tau = K_p(q^* - q) + K_v(\dot{q^*} - \dot{q}) + M(q)\ddot{q^*} + C_j(q, \dot{q})\dot{q} + g_j(q)$$

M denotes the inertial matrix, C the coriolis matrixm and gravitational torques is denoted by g. K_p and K_d are the proportional and derivative gains.

The change in angle required by each joint when moving the end-effector from one coordinate to another can be computed using the variation of the 'ikine' function for the given robotic arm; the forward kinematics can be computed similarly using the 'fkine' function.

2.1.2 Robotic Arm Models

Separate to the custom designed toolbox of Professor Corke, MATLAB has its own robot manipulator modelling sub-program and alternative methods of control. Simscape allows the creation of physical systems within MATLAB's simulation tool Simulink. An example of 6 DOF robotic arm is presented in Figure 2.1, below; the base of the arm represents the origin, and the lines span out from this with rigid bodies followed by links. Each rigid body can be shaped within MATLAB or imported from a CAD model to produce a replica of the real robotic arm; where properties such as material and inertia can be defined.

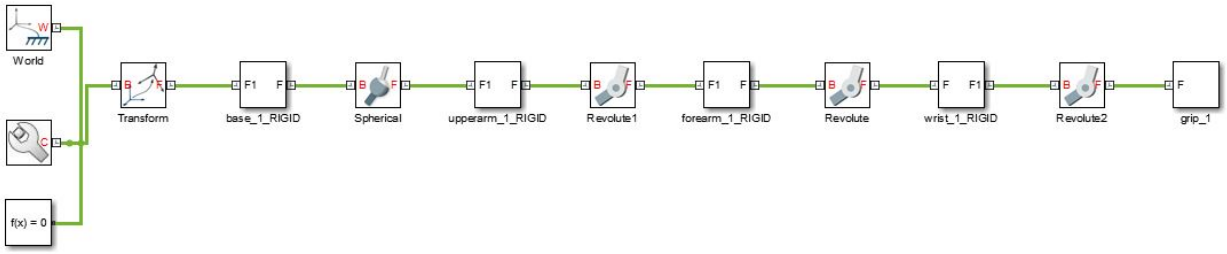

Figure 2.2: smrobot Simscape Model

Simulating the model shown above will provide a visualisation of the arm, as shown in Figure 2.3, and any motion set on it by external forces or motor torques; not included in the above model.

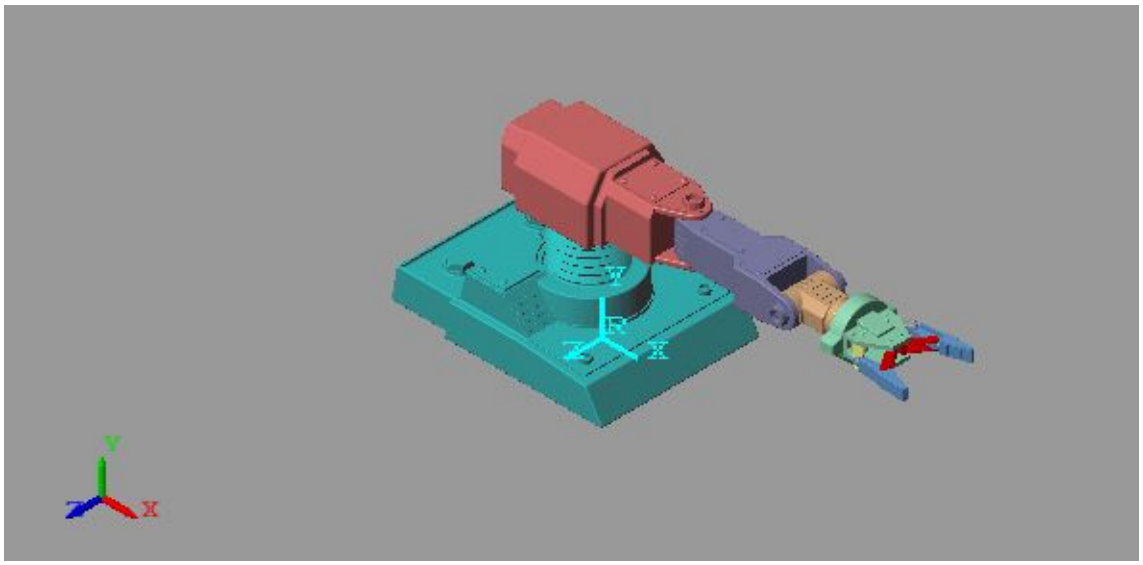

Figure 2.3: smrobot Simscape Model Visualisation

From here, it is a matter of controlling the robotic arm to move from one joint configuration to another. For a custom robotic arm, any of the strategies outlined in the theory section of this report can be used, however MATLAB provides another means which is described in the section below. In order to move the joints from one position to the next, a strategy similar to that shown in Figure 2.1 is required. In simulink, the layer of controlling the motor position can be formatted as illustrated in Figure 2.4.

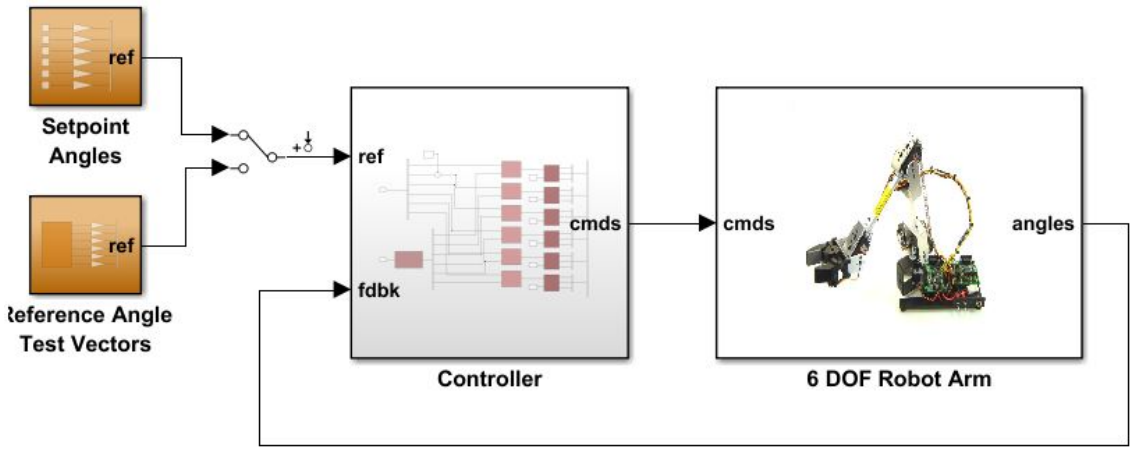

Figure 2.4: Simulink Model of Motor Control Loop

Where nested within the controller is the tuned PID controllers for each motor which drives it to the desired position; an example of this is presented in figure 2.5. This is similar to Peter Corke's method but without pre-emptive knowledge of the disturbance torque due to the arm's dynamic coupling.

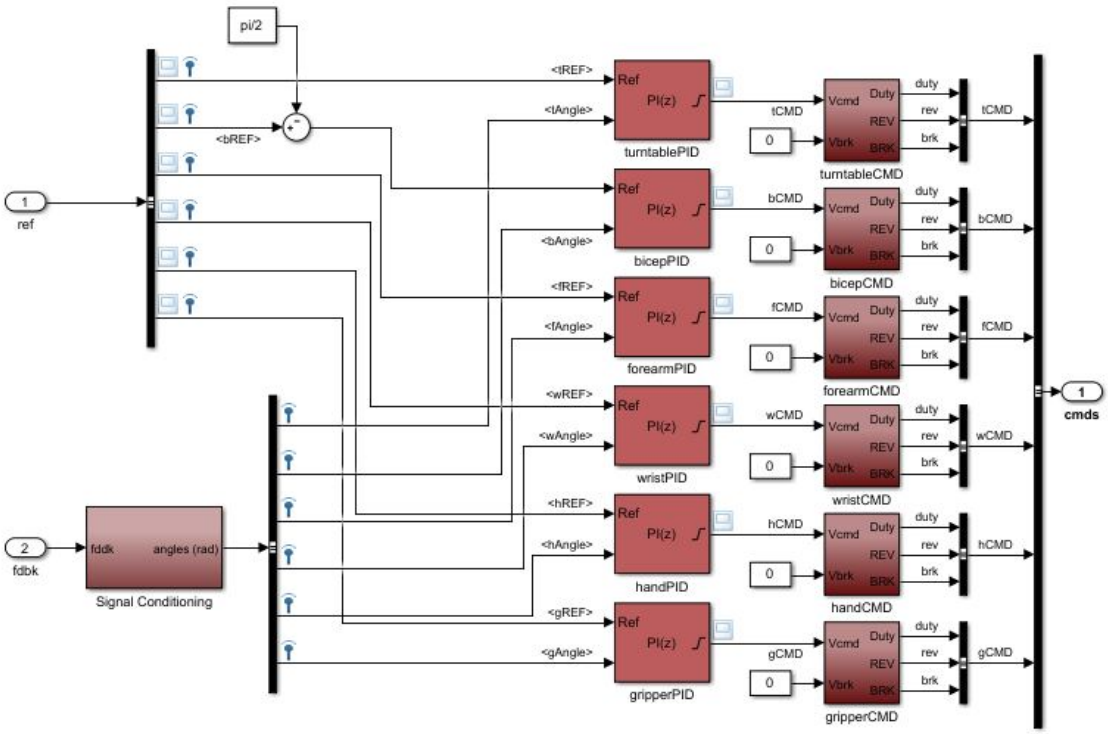

Figure 2.5: Simulink Model of Motor Controllers

It will be seen that VREP operates in the same way, as it enables the importing of manipulator models and setting of each joint's controller coefficients. The figure above gives a good representation of the mechanics behind what the parameters are doing in terms of regulating each joint's error to a desired angle. A significant advantage of MATLAB is the auto-tuning functionality of the PID controllers for each joint, however a controller on it's own would not be as robust as that described in the Robotics Toolbox section.

2.1.3 Adaptive Neuro-Fuzzy Inference System

An Adaptive Neuro-Fuzzy Inference System (ANFIS) network can be used to deduce the joint angles required for an arm's end-effector to reach a desired 3-D position. Calculating the required angles analytically for an arm of greater than 4 DOF can become challenging and result in infinitely many solutions. Like any neural network,

with a sufficiently large dataset the internal weights will learn what angles will result in the desired end-effector position. Thus, in training an ANFIS to control each joint for a training dataset, it is able to interpolate angles to sufficient accuracy and move the joints to reach a desired end-effector position.

To train the network, input to output mapping data is required. Thus, given the forward kinematics for a robotic manipulator is known, a dataset of angles to end-effector position can be created over the space that it is desired for the end-effector to move. After the network gains its own understanding of the mapping, any point within the testing region can be given and the network return the required joint angles. In a MATLAB example, each joint is given its own network to be trained such that for each joint an angle will be provided given a desired position. Outside the range of training, the manipulator will not respond promisingly, and thus for a large control range, a sufficiently large dataset is required. Overall neural networks respond quickly to providing results once trained, however the training process can be time consuming.

2.2 Programmed Robotic Arms

Volume 58 of the 2012 International Journal of Computer Applications, titled Software Development for an Inverse Kinematics of Seven-Degrees of Freedon Newly Designed Articulated Inspection Robot, details on previous software development tools used to control robotic manipulators. It is detailed that the inverse kinematics can be solved using an Adaptive Neuro Fuzzy Network (ANFIS), a neural network trained to reduce error in end effector position, non-linear optimization methods, geometric approaches, and through the manipulation of the transformation matrices. In the ANFIS approach, it was noted that MATLAB was used, however C / C++ programming has been used to develop programs that compute the more mathematics based inverse kinematics methods. A program is proposed in this paper which uses Visual-Basic to allow the calculation of forward and inverse kinematics, and trajectory planning.

A thesis paper from the University of South Florida comments on the fact that programming in a low level language such as C is closer to the hardware level which reduces computation time in processing and communicating with external systems (sensors and motors). High-level languages, such as MATLAB, provide an user-friendly interface for solving complex maths operations, however lack capabilities found in C / C++ such has multi-threading and easy communication between concurrently running processes. It is explained that the complexity in configuring MATLAB to communicate with a separate C++ program lead to failure of the

wheel-chair mounted robotic arm's program on multiple occasions. The software system in this paper refers to using a C++ program used to communicate with sensors and other hardware, whilst using MATLAB for the control algorithm computations, and simulation. Due to the aforementioned advantages of C++, a new C++ program, with an external library for matrix manipulation, was created to replace the MATLAB program. The concluding results being that the new software system reduced delays in the control of the robotic arm, leading to better stability and control.
The primary purpose of using MATLAB in this experiment was to simulate a virtual model of the robotic arm, and using the angles of this to control the physical robotic arm through the C++ platform. The complexity and delay produced by MATLAB lead to slow and unreliable control.

The Kinova Jaco arm's Software Development Kit (SDK) is written in C++, with the handling of communication and calculations stored in an human-unreadable dynamic link library. There is no documentation into how the SDK calculates the joint angles required for a desired gripper position, however there are usage examples in C++ of all the available functions; one of which is used to setting the gripper position and angle in cartesian coordinates. This SDK allows direct tuning of crucial arm parameters such as maximum applied torque, PID coefficients and velocity control. Predefined structures can be filled with this information and sent to the arm's FIFO buffer to interpret and act accordingly, making the control of the arm from the developer's perspective effortless.
Controlling the robotic arm in this way however, means the arm needs to be completely set-up in order to use the SDK. It may also be hard to understand how the parameters affect the movement of the arm; for example, the coordinate system isn't well defined and will take some trial and error to understand. This could be done with the help of the 'get' functions, which can return torque, velocity and position values of the arms current configuration. To further understand how the settings can affect the arm, Kinova have supplied the 'Development Center' with the SDK (which back-end C++ code is accessible, and can provide knowledge into how the arm can be controlled) what allows the user to move the arm, set trajectories, and review current parameters.

The Robot Operating System (ROS) is a modular collection of commonly used software in low-level device control, hardware abstraction and functionality seen in robotics. ROS can be used in Python, C++ and Lisp programming languages to build a framework around robotic software kits that directly control a robotic actuator. Thus, it has already been used as a cover of the Kinova SDK for the Jaco

arm. Further, simulation platforms such as Gazebo and MoveIt can be controlled using ROS, with the software for controlling a Jaco arm in Gazebo being available. Together, ROS can be used to control a virtual model of the Jaco arm in Gazebo, as well as the physical arm through the Kinova SDK. Control of robotic manipulators in VREP is also available using ROS through the VREP remote API.
ROS offers a package called Rvis, which provides a method of interacting with imported models in 3D. Similar to a robotics simulation package, Rvis can provide information about the centre of mass of objects, collisions, and joints of robotic manipulators and objects in a scene. It however lacks the real-world modelling that comes with simulation programs such as V-REP or Gazebo (running a physics engine). Thus, it is typically used as a wrapper for these programs, such that the user can retrieve key information about objects in a scene whilst simulating properties such as gravity and friction.

In Professor Peter Corke's Robot Academy series, he goes through an intuitive explanation for programming robotic arms using the arm's Jacobian matrix (referenced in Theory). As the Jacobian is a function of joint angles, it only provides a truthful reflection of the small change in angles required for a small change in end effector position, local to the current arm configuration. Thus, the Jacobian matrix can give a solution for small angle changes, but after the arm moves to this new position, it must be recalculated. This can be done in programming using discrete time steps as outlined in Figure 2.6. Unfortunately, as this is an approximation technique, if any of the spacial or time parameters are too large, the result will no longer hold, and thus it should be used with caution.

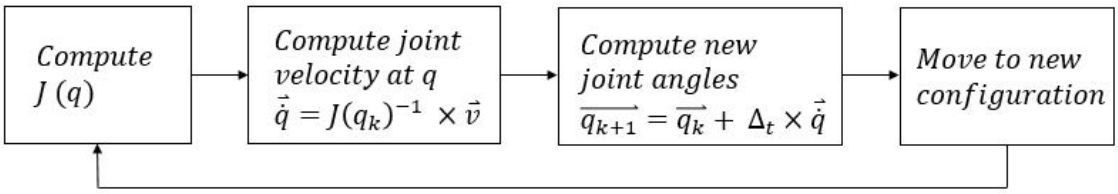

Figure 2.6: Flow Diagram of using Jacobian in Programming

A similar technique that can reduce the error between the end effector's position and a small change in position is the PID controller. With the appropriate weightings for each joint, the joints can be moved to reduce the error in end effector position. Note that there is no control of the joint-space in this case, and could lead to undesired joint configurations. This gives the Jacobian an advantage, as it is non-invertible at a joint singularity; when the arm looses a degree of freedom.

2.3 Simulation Platforms and Physics Engines

There are numerous platforms available for simulating the dynamics and interactions of robot's with a scene. V-REP is particularly useful for this thesis due to its ability to remotely control the simulation and the objects within, whilst providing a realistic environment for said interactions. Other popular robotic platforms have similar functionality, but may come at a cost, include Webots, Robot Studio, ARGoS, labVIEW, Visual Components, Virtual Robotics Toolkit, Gazebo, Actin Simulation and Workspace. Although the objective of these platforms is to simulate a robotic manipulator in a scene, they are customized in the robots that can be simulated, the physics engine they use to simulate the dynamics, and the requirements by the users using the product. A good example of this is the Robot Virtual Worlds platform, which has comprehensive capabilities in simulating and programming LEGO robotics. On the contary, Workspace is used for simulating industrial production lines with multiple robotic manipulators acting together; and is used by large companies such as ABB and Mitsubishi. The scope of this thesis focuses on the use of free simulators that provide realistic and programmable simulations for educational use. Thus, simulation platforms such as Webots, Gazebo and V-REP are of particular interest as they equally provide the required features to meet the objectives.

An online survey was conducted by researchers Serena Ivaldi, Vincent Padois and Franceso Nori to determine which simulation platform and dynamics solver was most preferred. The survey was completed by 119 participants with 62% possessing a PhD, and of the remaining, 32% had a university level degree. It was found that 39% of the participants hadn't heard of V-REP, compared to 27% and 15% for Webots and Gazebo respectively. However, it was found that V-REP was the best rated for usability, documentation and set-up guides, whereas Gazebo was the most used. From the findings, it was noted that V-REP was the most likely for applicants to try, and keep using, compared to Webots and Gazebo.

A full comparison between Gazebo and V-REP was conducted by Lucas Noguira from the Universidade de Campinas. This article goes through implementing features common to both platforms and the process required for setting them up to do a given task; i.e. connect a sensor to a robot and read it through ROS. It was found that V-REP offered a much better interface for adding components, and used less computation time when compared.

2.3.1 V-REP

The robotics simulation platform used in this thesis is the Virtual Robotics Experimental Platform (V-REP) by Coppelia Robotics. V-REP offers a means of

realistically simulating mobile robotics and actuators with an easy to use interface. In a simulation, V-REP continuously reads an internal 'main' Lua script, that may read other scripts describing properties associated with robotics in the scene, which are customisable by the user. On the side pane is a large list of available robots that can be imported, and once they are, a script customised by Coppelia is loaded to the list of running scripts; i.e. in the case of the Jaco arm, a script is imported that moves the arm around in the scene to a pre-configured trajectory. The simulation process is described by the Figure 2.7, where it can be seen that the main flow of code goes between actuation, reaction, sensing and control, with monitoring collisions being the main priority at each stage.

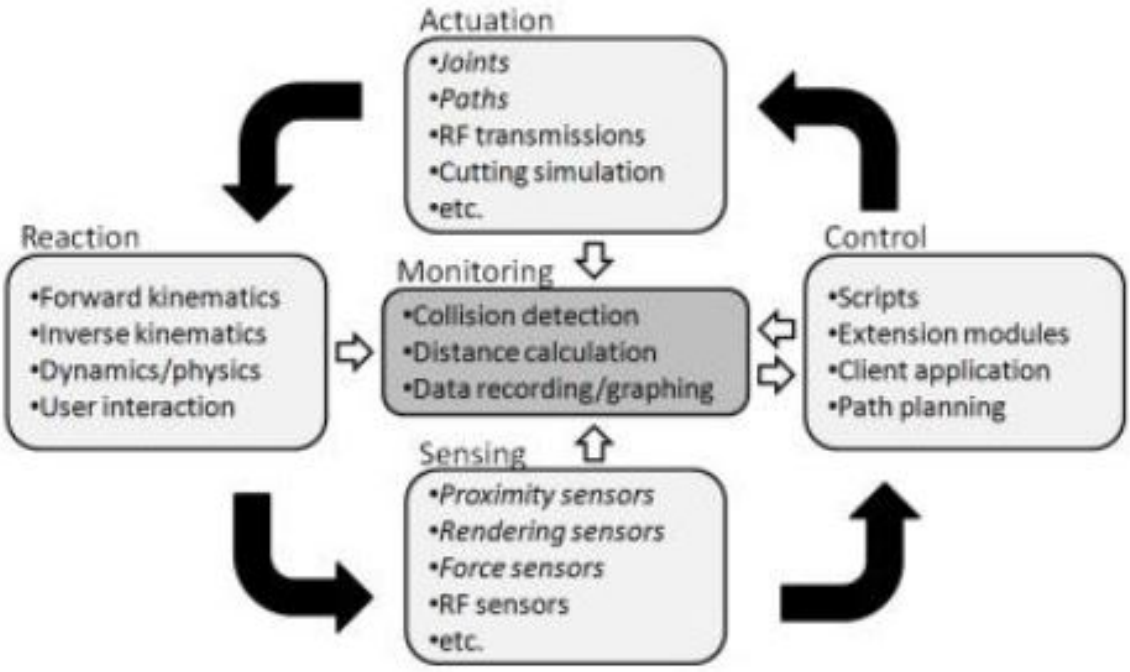

Figure 2.7: Flow Diagram of the V-REP Simulation

Communication

Fortunately, the parameters within these scripts and objects within the scene can be controlled using the remote API, which is available for C/C++, Python, Java, Matlab, Octava, Lua and Urbi. The documentation for the remote API contains extensive descriptions of over 100 functions, where the usage is available for all supported languages. In addition to the remote API and internal Lua scripts, there are 4 other methods of controlling a scene in V-REP. The most obvious is plugins, which can provide custom Lua commands for interacting with the internal Lua

scripts, it is effectively an external library of functions. Similarly, ROS can provide a wrapper of V-REP, enabling control from any ROS supported coding language, and further can enable the control of an external robot through a ROS node, using the information from V-REP; which is the aim of this thesis to a large extent.
There is no limitations to utilizing V-REP's capabilities between using a remote API client and a ROS node, as seen in the connection methods table on the V-REP website; both will contain lag due to communication over the internet through a socket connection. Other methods of controlling V-REP include through the use of an add-on, or a BlueZero node.

Communication using the remote API is through a socket connection between the client API and VREP. To enable connection, V-REP needs to be configured to allow external connections through a specified port; there is no restriction to the number of open ports, however it does induce simulation lag. As scripts are associated with objects within the scene, this configuration is typically set in a threaded child script of an invisible dummy object in the V-REP scene; an example of this can be seen in Figure 2.8.

```
threadFunction=function()
end


simSetThreadSwitchTiming(2) -- Default timing for automatic thread switching

simExtRemoteApiStart(19999) -- port of which remote API server will run

-- Here we execute the regular thread code:
res,err=pcall(threadFunction)
if not res then
    simAddStatusbarMessage('lua runtime error: '..err)
end
```

Figure 2.8: Sample Lua Code for Connecting V-REP to a port

On the client side, connecting to V-REP waits for a given length of time for a client identification number. Once this number is obtained, the number is used for all communication thereafter in a manner noted by Coppelia for the language chosen. As an example, the functions could return parameters such as joint angles, velocity and sensor outputs, alternatively the functions could also be used to set or change these parameters.

Communication between the client program and V-REP is undertaken in four different modes; blocking function calls, non-blocking function calls, data streaming and synchronous operation. A description of these is presented in Table 2.1.

Operation Mode	Summary
Blocking	When a command is sent in blocking mode, the client will wait until the V-REP API response before moving to the next line of code.
Non-blocking	When data needs to be updated in the client program, that isn't required straight away. The client can make a call for the data to be copied to a particular address and move on to the next line. The API will respond in due time.
Data Streaming	When the same data is constantly required from V-REP, the data-streaming method allows a way for V-REP to send updates of a particular value without calling for it each time. For instance, joint angles of a robotic arm may be required by the client program with large frequency, and could be updated without making direct calls and waiting for a response each time once a data-streaming call is given. This is a continuous mode, and the client will be stuck receiving data.
Synchronous	When it is desired to have the simulation and client program run concurrently in time, such that the simulation advances with the client. Synchronous mode allows the client to control the advancement of the simulation.

Table 2.1: V-REP remote API operation modes

To be able to interact with a particular object in V-REP, one must refer to it in function calls using its 'object handle'. The object handle is a unique number given to every object in a scene, typically in order of when the object was added to the scene, known by its appearance in the scene hierarchy; a list containing all of the objects in the scene, starting with objects like the default camera, floor, lights, XYZ coordinate frame, and then any objects that were placed in the scene after. Once an object has been deleted from the middle of the hierarchy, the other objects retain their object handle; thus once an item is added its object handle won't change. When calling a missing object in a scene, NULL will be returned.

V-REP Features

Models can be imported into V-REP as a mesh (.stl) from any 3D modelling software (such as AutoDesk). The models can either be static objects to be placed in the scene, or make up parts of a robotic manipulator. When importing models, it is important to consider the amount of triangles used to create the component, when this is too large (more than 20,000), manipulating the object in V-REP will come with

noticeable lag. When building a robotic arm from a CAD drawing, the entire model can be imported, and then subdivided to form an interaction 'group' of meshes. When the Denavit-Hartenberg parameters of the arm are known, these can be used to form the relationship between the joints. The interactions between parts can be made by adding translational and/or rotational relationships, with parameters such as maximum angle and starting position specified. All of the objects in the scene can be modified out of simulation to be dynamic or static (be influenced by external forces), and responsive or non-responsive (have a reaction when a collision with the object occurs). It should be noted that the object doesn't need to be part of a robot, but can be an object in the scene either as a feature or an interactable object.

On start-up, V-REP comes with 21 pre-assembled non-mobile robotic manipulators (arms), including the Kinova Jaco and Mico arms. Other V-REP supported models can be loaded into a scene, given the appropriate .ttm format. The model browser also includes a selection of pre-made models, such as arm grippers, sensors, furniture, household appliances, people, vehicles and mobile robots. Using the model browser alone, it is possible to replicate a room or scene that a real robot would interact with. The robots can also be placed on any surface in the scene, and be provided a given trajectory or motion path for interacting with objects.

Collision detection is an important feature required for this thesis, as the objective is to detect collisions within the scene. V-REP has a collision detection feature, which will register collisions between collidable entity-pairs. Thus, objects that may collide within a scene need to be set prior to simulation; a robotic arm can be set to 'Enable all collision detections', which will flag when the robotic arm collides with any object. From a remote API client's prospective, this isn't yet helpful as V-REP will not flag a collision to the client automatically. Instead, the client program will need to ask for the execution of a custom Lua function within V-REP, which will use the simCheckCollision function to check for collisions within the scene, and then stream the response back to the client. Thus, this could be used in a data-streaming method, where the client is continuously checking for data from V-REP being send from the custom function. Alternatively, the remote API contains a simxReadCollision function which can be called given an object's collision handle (i.e. a link of a robotic arm pre-registered to potentially collide), where the response will indicate a collision or lack there of.

The most useful feature of V-REP's for this thesis is the kinematics solver, where the arm can be set to either forward kinematics or inverse kinematics mode. By default, one is able to give joints desired positions, and the PID controller representing

the motor dynamics actuates the joint to produce a realistic resulting position at that of which is desired; unless the motor actuation saturates, resulting in steady-state error from the goal position. From the V-REP client, one is able to send desired position, velocity and actuation force to the joints of a robotic manipulator at any given moment. Conversely, when the position of a joint in space is desired, the calculation of the angles to produce this goal position is required. V-REP is able to solve this inverse kinematics problem, given a desired position and the links requiring movement are known; and the desired position is in reach of the arm, i.e. the solution is solvable.
To set a robotic arm to inverse kinematics mode such that the end-effector reaches a goal position, a dummy object representing the 'Target' for the gripper must be added. It may be desirable to also create a 'Tip' dummy object that moves with the gripper, at a position defined as the tip by the forward kinematics transformation for the arm. This tip object is required to be nested under the gripper parent on the scene hierarchy, such that it moves with the gripper, however its world coordinate can be set to that of the forward kinematics solution at any moment, and this offset to its parent will be maintained.
The 'Scene Object Properties' for the dummy objects depict the kinematics mode and the links to other objects in the scene. In this window, the objects would be required to be set to non-collidable, non-measurable and non-detectable, so they are merely position orientated objects. Further, once the inverse kinematics link between the objects is made, V-REP offers the user to select between two inverse kinematics solutions in the 'Calculation Module Properties' window. These are the pseudo-inverse and damped least squares methods which are discussed in the theory section. Depending on the chosen method, the damping and number of loop iterations on evaluating an accurate solution can be set.

2.3.2 Alternative Simulation Tools

Webots

Webots is a robotics platform that allows the simulation of popular robot models, that can be interacted with using external software scripts, of which can be then programmed onto real robots; given they are compatible. Similar to V-REP, one can communicate with the simulation using TCP/IP from a program written in C/C++, MATLAB, Python or Java. The simulation also utilises Open Dynamics Engine (ODE) to give a realistic representation of physical interactions between objects. Motors and sensors can be placed in the scene or within robots, and tuned to mimic their real-time capabilities; maximum torque, PI control of servos, and

sampling of sensors to name a few parameters that can be varied. There is little information on the comparison between this platform and V-REP, but from the information found on Webots and in the survey outlined above, it appears to have equivalent functionality, if not less, as it relies only on one physics engine whereas V-REP offers four. Webots could be used for this thesis, as it provides a means of real-time (or even speed-up) interaction with a scene and the objects within. The use of ODE is no downfall, as outlined in the ODE section, it provides a realistic rendering of collisions within a scene.

Gazebo

Gazebo offers a very similar platform to V-REP, where a model can be placed in a scene and have the simulation controlled by a remote API. As an open source platform, full integration with ROS has been created, where the internal parameters of a simulation can be controlled with ROS. It is offered on any OS with a command line interface to help control simulations. Simulations are saved to an .xml script, which can be edited, and rerun; a feature V-REP lacks with it's .ttt saving format. A thorough comparison was made between Gazebo, V-REP and ARGoS, which demonstrates that they all contain the key features required for this thesis. It can be seen however, that V-REP has more positive features when compared to the other two. Gazebo on start-up is built with ODE, however can be built from source with any physics engine.

Bullet Physics Library

Open Dynamics Engine

Vortex Dynamics

Newton Dynamics

Chapter 3

Theory

In order to ensure the client program can function responsively, control the robotic arm in real-time and not over-use the computer's CPU, a variety of theory had to be researched before commencing the project. In particular, the theory examined covered three main areas; the mathematics associated with the dynamics of non-linear systems, control theory, and software development. As this is a Mechatronics thesis, key theory of these topics will be explained for the interest of the reader.

3.1 Child Processes and Multi-Threading

Processes have their own memory-space and are are completely separate to one another. They are able to interact with their virtual memory and create threads to undertake separate tasks, such as getting inputs from a user, handling the graphical interface, writing to external hardware. A process is effectively the application or program as a whole, and threads are the internal functionality that coordinate information that make up the program. Threads have the appearance to be working in parallel, but synchronously; for instance, one thread may be checking for a particular sensor reading, and once this has been received it can signal to another thread, that is waiting for the signal, to conduct calculations or react in particular manner. Threads have their own stack and priority, and so variables local to one thread don't affect another's, however as noted, they are able to signal each other, as well as share the process's data (i.e. global variables). This is a vital difference between processes and threads, processes cannot effect each other's memory, they have allocated space and interact with Kernel on how much memory it can use, but it cannot access another process's variables. All application processes have the same priority also, whereas threads can have a variety on different priorities depending on CPU time and importance to the overall functionality. As a rule of thumb, threads that make important calculations and run quickly have high priority,

whereas slower threads doing large tasks can have a lower priority, because compared to their overall CPU time blocking for a fast thread won't greatly effect the runtime.

Typically, when developing an application in a coding language such as Python, MATLAB and C, the flow of code is singular as there is one process and one thread. The main thread in a program can create other threads, which are given a unique thread identification. In the same way, a process can spawn another process with a unique process identification number, however obtains a parent process's identification number. In this way, the parent process can monitor the status of the child process, and send signals, similar to threads.
In Linux, the child process is created by duplicating the parent, and giving the new process it's own resources such as memory-space and timers. The processes can communicate through reading and writing to files in memory and signalling that messages are ready; this is called piping, where an output from one process serves as in input to another. When using the Windows API however, the set-up is slightly different, in that everything is a 'Window', and Windows send messages to other Windows. The intuition behind this is that a Window can be interacted with by the user at any time, with key strokes or mouse clicks, and from operating system events. Thus, when a Window is created, it is set-up to receive signals through a queue that will hold messages, so it is ready when these different events occur, enabling it to react accordingly. Between operating systems, the overall outcome and the way processes interact is in essence the same, however when programming these differences become noticeable as each has their own checks, structure and signalling.

The CPU is a shared resource between processes, where the CPU usage for a given process is calculated as a percentage of the amount of clock cycles dedicated to running a particular task/thread. As aforementioned, threads can be allocated priorities, and thus if the threads of a process are constantly requiring run-time, say being stuck in an infinite loop, the CPU will be held servicing the one process given no task of higher priority is scheduled, thus consuming 100% of the CPU usage. On a program level, if tasks are of equal priority and in no way get suspended, the CPU will schedule the run-time of the tasks equally between them.
To avoid having a program use up all of the CPU's available usage, by taking the CPU whenever it becomes available, threads are suspended whilst they wait for information. Suspended threads can be resumed by events such as receiving a signal from another thread, or a given amount of time has passed. It is in this way that the CPU is able to run multiple programs concurrently, threads will block until they are required to be serviced, and until then the processor can run other tasks. An example of this is the USB ports in a computer, the processor will run programs, and

occasionally stop and check the hardware states of the computer, before re-storing the memory of a suspended program and resuming. When a USB is connected, the hardware task will signal that one of the ports is in use, and begin a chain of events, taking priority over the program that was running.
Threads can be suspended in programs typically by a function called sleep, which is given a specific amount of time to sleep for. The function for resuming suspended tasks will also exist, which typically requires a handle to the thread in memory.

3.2 Regulators

The fundamental definition of a linear system is a system that when given a harmonic input, produces a harmonic output. Due to the frequency characteristics of the system, the output may be scaled and phase shifted, however the harmonic remains. In this sense, the input shapes the output. Regulators are a class of control scheme that use this property to shape a desired output given a reference input, with the use of feedback and a compensator.

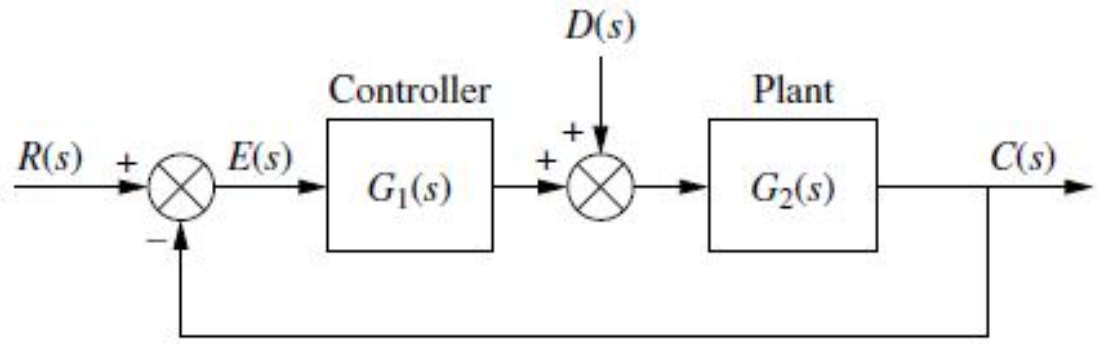

Figure 3.1: Feedback Control System with Disturbance

Presented in the figure above is a simple control system which contains a plant that takes input from a controller, which is driven by an input defined by the reference input R(s) and output C(s); although the laplace notation is used in this figure, R and C could the state vectors of the system. Ignoring the disturbance at this the system in this thesis is virtual, it is known that the gain for this type of system is defined as;

$$\frac{C(s)}{R(s)} = \frac{G_1(s)G_2(2)}{1+G_1(s)G_2(s)}$$

and the error is;

$$E(s) = \frac{R(s)}{1+G_1(s)G_2(s)}$$

where controllers $G_1(s)$ are designed to minimize $E(s)$,such that the steady-state error is zero;

$$e(\infty) = \lim_{s\to 0} s * E(s) = 0 \tag{3.1}$$

A typical example of a controller has already been mentioned in this paper, and that is the PI (proportional gain plus integral action controller) which was proposed for regulating the servo positions inside a robotic manipulator's joints to a desired joint position. In that circumstance, the controller is defined by the function;

$$G_1(s) = \frac{K_P s + K_I}{s} \tag{3.2}$$

where K_P and K_I are the proportional and integral gains respectively. In this case, the controller takes in error in servo angle and outputs a voltage that goes to the motor to produce the desired reference on the output. The reason a PI controller is used in this case is due to the integral action's ability to reduce steady state error. This is easily analogous when considering a mechanical system being moved to a desired position. The proportional gain multiplies by the current error to produce a force that moves the system, however there is no judging whether this force is enough to overcome the opposing forces in the system, and may result in an equilibrium position less than that of which is desired. The integral of the error can thus be used, as the error accumulates and gets multiplied by K_I, the force grows to overcome the disturbances until the desired location is reached. This can be seen when considering the steady-state error formula, 3.1, with the referenced controller, 3.2, and a mechanical plant with inertial and linear components (thus, at minimum be defined by $[M]s^2 + [K]$);

$$\lim_{s\to 0} s * E(s) = s * \frac{R(s)}{1+\frac{K_P s+K_I}{s}G_2(s)}$$

$$\rightarrow e(\infty) = \lim_{s\to 0} s\frac{sR(s)}{s + (K_P s + K_I)G_2(s)} = \lim_{s\to 0} \frac{0}{K_I G_2(0)} = 0 \tag{3.3}$$

Thus integral gain eliminates steady-state error given sufficient time, hence its use as a regulator in servo motors. The use of controllers cannot be mistreated however, as it may have been realised in the mechanical example above, the gains will change the dynamics of the system. Thus, when using controller's it is required that they are tuned to give the desired damping, rise-time and settling time of which is required for the system. In general, the addition of K_I increases overshoot and settling time, which can be counter-acted by the inclusion of a velocity term K_D, which for a second order system will decrease overshoot, but incorporate delay into the system. Another negative with using a derivative controller as a regulator in the feed-forward path is that it can lead to over-saturation of the actuator when the reference input is changed between two consequent time samples (step change), producing a 'infinite' gain being multiplied through the plant. The proportional gain is similar to K_I, in that if sufficiently large can cause overshoot of the desired position, however, will always result in steady-state error.

This thesis looks into the use of controllers to correct for the error between the desired end-effector position, and the actual. It is proposed that regulating the co-ordinate space of the end-effector can be achieved by varying the joint angles of the arm with a controller based off the joints' sensitivity in moving the end-effector in a given direction. The premise behind the methodology chosen is backed by the theory presented in this section.

3.3 Forward Kinematics

As previously mentioned, the forward kinematics of a robotic chain is mathematics that defines the relationship between known joint angles and the end-effect position, such that;

$$\vec{P} = f(\vec{q}) \tag{3.4}$$

where $\vec{P}$ is the position vector of the end-effector relative to the base of the arm and $\vec{q}$ is a vector of joint angles θ_i or prismatic joint displacements d_i.

This function, $f(\vec{q})$, is found by the translational components of the transformation matrix, T, that describes the sequence of twists, rotations and displacements between end-effector position and base of the chain. T can be found by multiplying the transformation between the base link and the next due to q_1, by the transformation between the next link and that after due to q_2, and so on until the end of the arm, denoted as link number n, such that;

$$T = A_0^i = A_0^1 A_1^2 ... A_{n-1}^n \tag{3.5}$$

The homogenous transformation between two links, A_i^{i+1}, is created using the product of the rotational transformation about the (rotation axis) z-axis, the translation transform along the z-axis a displacement d_i (the offset in the z-axis from one link to another), the translation transform along the x-axis a displacement a_i, and finally the rotational transformation describing the rotation from one x-axis of first joint to the same of the next by an angle denoted α_i. These terms are the Denavit-Hartenberg Parameters for the robotic chain, and can be further understood in Figure 3.2, provided by Mark W. Spong's Robot Modeling and Control textbook;

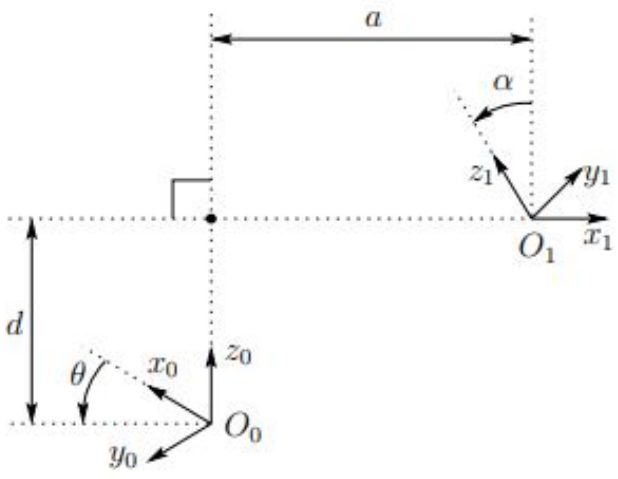

Figure 3.2: Physical representation of D-H Parameters between two links

Where it can be seen that two criteria must be met for the transformation to exist, z_0 intersects x_1, and axis x_1 is perpendicular to axis z_0.

Thus, A_i^{i+1} is defined as;

$$A_i^{i+i} = Rot_{z,\theta_i} Trans_{z,d_i}, Trans_{x,a_i} Rot_{x,\alpha_i} \tag{3.6}$$

where

$$Rot_{z,\theta_i} = \begin{pmatrix} c_{\theta_i} & -s_{\theta_i} & 0 & 0 \\ s_{\theta_i} & c_{\theta_i} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, Trans_{z,d_i} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$Trans_{x,a_i} = \begin{pmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, Rot_{x,\alpha_i} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c_{\alpha_i} & -s_{\alpha_i} & 0 \\ 0 & s_{\alpha_i} & c_{\alpha_i} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Thus, the transformation between joints can be described as;

$$A_i^{i+1} = \begin{pmatrix} c_{\theta_i} & -s\theta_i c_{\alpha_i} & s_{\theta_i} s_{\alpha_i} & a_i c_{\theta_i} \\ s_{\theta_i} & c_{\theta_i} c_{\alpha_i} & -c\theta_i s\alpha_i & a_i s_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.7}$$

It can be seen that the homogenous transformation A_i^{i+1} contains the 3 dimensional rotation transformation matrix from i to $i+1$, and the column vector representing the translation transformation. Equation 3.7 and therefore be written as;

$$A_i^{i+1} = \begin{pmatrix} & Rot_i^{i+1} & & Trans_i^{i+1} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.8}$$

Thus, as $T = A_0^1 A_1^2 ... A_{n-1}^n$, T will too have this form. In addition to this, the end-effector position, for arbitrary joint angles $\vec{q}$, can be found by extracting the translational column vector from equation 3.8, such that;

$$\vec{P} = f(\vec{q}) = Trans_0^n(\vec{q}) \tag{3.9}$$

3.4 Inverse Kinematics

The inverse kinematics of a robotic manipulator is the mathematics that defines the angles of the robotic given the position of the end-effector relative to the base of the arm. Using the formula from the previous section, the inverse kinematics can be written as;

$$\vec{q} = f^{-1}(\vec{P}) \tag{3.10}$$

Given the derivation from the last section, this computation isn't simple to compute, as the angle vector $\vec{q}$ can contain any number of joint angles, whereas the position vector of the end effector $\vec{P}$ is defined in $\mathbb{R}^3$. For a 6 joint arm, this gives 6 unknowns and three equations.

There are numerous methods that present inverse kinematics solutions to any serial chain of joints; even when there is no known closed form solution. A paper by Gregory Z. Grudic, the University of British Columbia, called Iterative Inverse Kinematics with Manipulator Configuration Control and Proof of Convergence, lists many of the different methods in some detail, including the Tsai and Morgan's solution for a 6 revolute joint arm. A summary of popular solutions is provided in the appendix

In this thesis, the most common approaches are used to determine the joint angles required to move the Jaco arm's end effector.

3.4.1 By Matrix Manipulation Approach

As previously stated, the transformation that describes the end-effector relative to the base for a set of joint angles can be represented by equation 3.5. From this, for a 6 joint arm, it can be determined that;

$$(A_0^1)^{-1}T = A_1^2 A_2^3 A_3^4 A_4^5 A_6^6 \tag{3.11}$$

In the Introduction to Robotics: Analysis, Systems, Applications textbook by Saeed B. Niku, it was proposed that mathematical relationships between joints can be found by multiplying both sides of the transformation equation described in eq1, by the inverse transformation between two joints, as seen in eq2. In doing this, it is

known that the left hand side must equal the right hand side, however some entries between the left and right may not appear the same, despite being equivalent. From this fact, two entries can be equated together to write trigonometric relationships between joints. It was found that this property is most easily seen where angled twists appear between joints. By iteratively multiplying through by the inverse of a transformation matrix between two joints and inspecting the right and left side's elements, the equations for each joint angle was determined.

3.4.2 Analytical Approach

A popular approach to solving the inverse kinematics of simple serial chains is by analytically solving the relationship between joints, from the base to the end-effector. This is easiest seen through example by considering the 3 revolute joint arm in Figure 3.3, provided by Lung-Wen Tsai's Robot Analysis textbook.

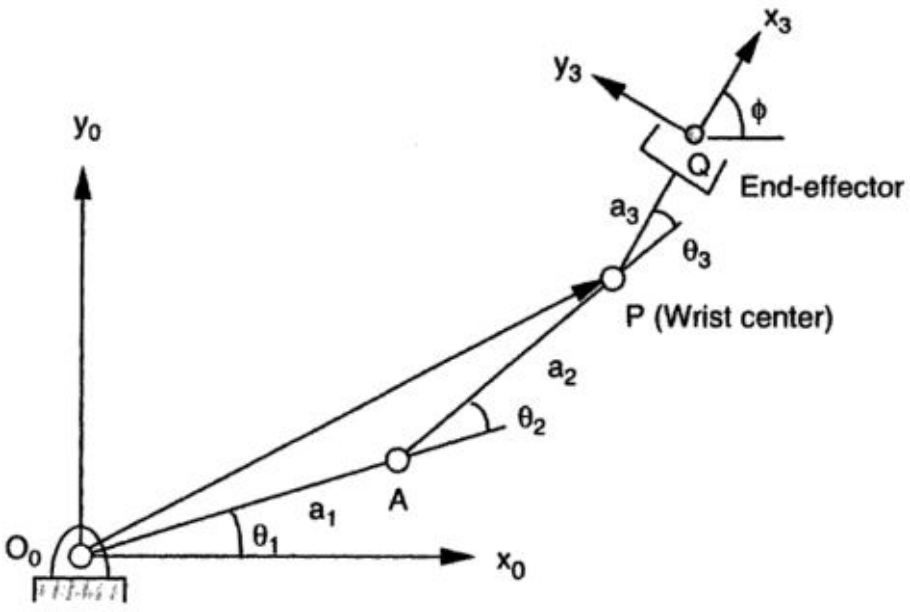

Figure 3.3: Revolute Joint Arm

It can be seen in the figure above, that the end-effector is described by the coordinate vector $\vec{Q}$, with a joint 3 being represented in space by vector $\vec{P}$. It can also be seen that lengths a_1 and a_2 form a triangle with vector $\vec{P}$, dependent on the varying angles of θ_1 and θ_2. In addition to this, it can be seen that the four equations can be derived between the points;

$$p_x = a_1 c\theta_1 + a_2 c\theta_{12}$$
$$p_y = a_1 s\theta_1 + a_2 s\theta_{12}$$
$$p_x = q_x - a_3 c\sigma$$
$$p_y = q_y - a_3 s\sigma$$

where $\sigma = \theta_1 + \theta_2 + \theta_3$

From the cosine rule, the triangle can be defined by;

$$p_x^2 + p_y^2 = a_1^2 + a_2^2 - 2a_1 a_2 c\theta_2 \tag{3.12}$$

which can be rearranged for θ_2 in terms of point P and the arm lengths such that;

$$\theta_2 = cos^{-1}(\frac{p_x^2 + p_y^2 - a_1^2 - a_2^2}{2a_1 a_2}) \tag{3.13}$$

which will have two solutions depending on the evaluation of the fraction within the inverse cosine, and the arm is fully stretched when this equates to 1.

The equations describing p_x, p_y in terms of θ_1, θ_2 above can be rearranged for $cos(\theta_1), sin(\theta_1)$ such that;

$$c\theta_1 = \frac{p_x(a_1 + a_2 c\theta_2) + p_y a_2 s\theta_2}{a_1^2 + a_2^2 + 2a_1 a_2 c\theta_2} \tag{3.14}$$

$$s\theta_1 = \frac{p_y(a_1 + a_2 c\theta_2) - p_x a_2 s\theta_2}{a_1^2 + a_2^2 + 2a_1 a_2 c\theta_2} \tag{3.15}$$

and θ_1 can be computed by;

$$\theta_1 = atan2(s\theta_1, c\theta_1) \tag{3.16}$$

Using the desired end-effector position Q, the third angle can thus be solved from its relationship to point P, described above. This method can be repeated working

up the joints, were the position of each link can give the required joint angles. The equations outlined in this section were used in the final product of this thesis.

3.4.3 Jacobian

The Jacobian is a linearised representation of the dynamics of a system for a particular set of states. For example, the system for a robotic arm can be computed by Lagrange Equations, to have the form;

$$[\tau] = [M(q)]\ddot{q} + [C(q, \dot{q})]\dot{q} + [G(q)]$$

Where τ is a vector of the applied joint torques, q describes the joint angles such that $\dot{q}$ is the joint velocity vector and $\ddot{q}$ is the joint acceleration vector. As the arm moves, the perpendicular distance to gravity from the arm's base each joint will vary, and thus so will the torque due to gravity; this dynamics is covered in the matrix G, which depends on q. The matrix C contains the Coriolis and centripetal terms that make up the gyroscopic torques experienced as the arm swings around, which is a function of velocity and angle. Finally, the matrix M represents the torques experienced due to the inertia of the arm's links, which is a function of joint accelerations, and varies with joint angles.
One is able to control the arm using a supplied torque once these dynamics are known, however for a 6 DOF arm, there will be 60 parameters that need to be found to accurately described the motion. By taking the Taylor Series expansion of the equation of motion described above, about a linearised set of joint angles, the small change in angles required for a change in end-effector position can be determined; it is this matrix result that is termed the Jacobian. This can be described below, where $\vec{X}$ represents the change in end-effector position. Where τ is described by the equation above, the force F of the end effector for a small change in end-effector position, can be used to find the change in angles;

$$\tau \delta \vec{q} = F \delta \vec{X}$$

$$\tau = J^T F$$

An alternative approach to using the dynamics of the system to compute the Jacobian matrix, it to derive it from the forward kinematics transformation matrix; which already describes a relationship between joint angles and end effector position.

From the transformation matrix, it was found that the position of the end effector $\vec{P}$ can be found by equation 3.4;

$$\vec{P} = f(\vec{q})$$

The problem came when considering the inverse of $f(\vec{q})$, which is a 3x6 matrix for a 6 joint arm; as this matrix is rank deficient it cannot be inverted. Three more states that describe the position of the end-effector is required to construct a square matrix, and these are the end-effector's angular positions relative to the base of the arm, as described by the rotation matrix within the transformation matrix T.

From the transformation matrix, it can be seen that a small change in angle q_i can cause a small change in rotation and translation of the end effector such that;

$$\frac{\delta T}{\delta q_i} = \begin{pmatrix} & \frac{\delta R}{\delta q_i} & & \frac{\delta t}{\delta q_1} \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{3.17}$$

In three dimensions, angular velocity can be represented by the Skew matrix;

$$S(\omega) = \begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix} \tag{3.18}$$

Thus, the angular velocity vector $(\omega_x, \omega_y \omega_z)^T$ can be found from forming the Skew matrix by equating coefficients, using the result described in 3.18 for each angle;

$$S(\omega) = \frac{\delta R}{\delta q_i} R^T \dot{q_i} \tag{3.19}$$

From this, the Jacobian for a 6 DOF arm can be described as;

$$J = \begin{pmatrix} \frac{\delta x}{\delta q_1} & \frac{\delta x}{\delta q_2} & \frac{\delta x}{\delta q_3} & \frac{\delta x}{\delta q_4} & \frac{\delta x}{\delta q_5} & \frac{\delta x}{\delta q_6} \\ \frac{\delta y}{\delta q_1} & \frac{\delta y}{\delta q_2} & \frac{\delta y}{\delta q_3} & \frac{\delta y}{\delta q_4} & \frac{\delta y}{\delta q_5} & \frac{\delta y}{\delta q_6} \\ \frac{\delta z}{\delta q_1} & \frac{\delta z}{\delta q_2} & \frac{\delta z}{\delta q_3} & \frac{\delta z}{\delta q_4} & \frac{\delta z}{\delta q_5} & \frac{\delta z}{\delta q_6} \\ \frac{\delta \theta_x}{\delta q_1} & \frac{\delta \theta_x}{\delta q_2} & \frac{\delta \theta_x}{\delta q_3} & \frac{\delta \theta_x}{\delta q_4} & \frac{\delta \theta_x}{\delta q_5} & \frac{\delta \theta_x}{\delta q_6} \\ \frac{\delta \theta_y}{\delta q_1} & \frac{\delta \theta_y}{\delta q_2} & \frac{\delta \theta_y}{\delta q_3} & \frac{\delta \theta_y}{\delta q_4} & \frac{\delta \theta_y}{\delta q_5} & \frac{\delta \theta_y}{\delta q_6} \\ \frac{\delta \theta_z}{\delta q_1} & \frac{\delta \theta_z}{\delta q_2} & \frac{\delta \theta_z}{\delta q_3} & \frac{\delta \theta_z}{\delta q_4} & \frac{\delta \theta_z}{\delta q_5} & \frac{\delta \theta_z}{\delta q_6} \end{pmatrix} \tag{3.20}$$

Thus, the change in angles required for a change in end-effector position;

$$\frac{dX}{dt} = J\frac{dq}{dt} \tag{3.21}$$

and the change in end-effector position that results from a change of angles is;

$$\frac{dq}{dt} = J^{-1}\frac{dX}{dt} \tag{3.22}$$

In the case that the Jacobian isn't invertible the Pseudo Inverse Jacobian can be used; when one of the angular velocities doesn't change with angle, which occurs when there are less than 6 DOF in the arm. Thus, the objective of this method is to make the Jacobian full rank, by multiplying the Jacobian by its transpose, resulting in;

$$\frac{dq}{dt} = J^T(JJ^T)^{-1}\frac{dX}{dt} \tag{3.23}$$

3.4.4 Damped Least Squares

The damped least squares method provides a more stable set of change in angles required for small errors in end-effector position close to arm singularities, however takes longer to compute. This method looks to minimize this error by the following quantity;

$$\Delta\theta = (J^T J + \lambda^2 I)^{-1} J^T \Delta X \tag{3.24}$$

given a non-zero damping constant λ. There are numerous methods for finding a good quantity for λ, however in general it should be large enough such that the solution is stable near singularities, but not too large or else accuracy is reduced.

3.4.5 Newton-Raphson Method

The Newton-Raphson method is a numerical methods technique on homing-down on a better approximation than one given. It was found that, once an approximation gives a result with small error from the expected, a better solution can be made by reusing formula with the result of the original guess, where convergence on an accurate solution can be retrieved given enough iterations in the local space to the original guess.

from the Taylor Series, it is known that the true position x can be given by a previous position x_0 plus h, for small changes in position h, ;

$$f(x) = f(x_0 + h) \approx f(x_0) + hf'(x_0) \tag{3.25}$$

and thus;

$$h \approx -\frac{f(x_0)}{f'(x_0)} \tag{3.26}$$

therefore, the true position can be approximated by;

$$x = x_0 + h \approx x + 0 - \frac{f(x_0)}{f'(x_0)} \tag{3.27}$$

using this property in iteration, the true position x will be converged on, such that the next estimate for iteration number n is given by;

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{3.28}$$

It has already been described that the Jacobian can be used to approximate the change in joint angles required for a change in end-effector position, thus, a better approximation of these angles can be determined using the Newton-Raphson method. Given the Taylor Series expansion of a desired end-effector position x_d;

$$x_d = f(\theta_d) = f(\theta^k) + \frac{\Delta f}{\Delta \theta}|_{\theta^k}(\theta_d - \theta^k) + (higher\ order\ terms) \tag{3.29}$$

it is known that

$$J(\theta^k) = \frac{\Delta f}{\Delta \theta}|_{\theta k} \tag{3.30}$$

and

$$\Delta\theta = (\theta_d - \theta^k) \tag{3.31}$$

Equations 3.30 and 3.31 together give;

$$x_d - f(\theta^k) = J(\theta^k)\Delta\theta$$

$$\Delta X = J(\theta^k)\Delta\theta \tag{3.32}$$

Thus, this matches the form used in the Newton-Raphson method, and the iteration method can be used to create a better approximation of ΔX given a better approximation of $\Delta\theta$. This process is described in the figure below;

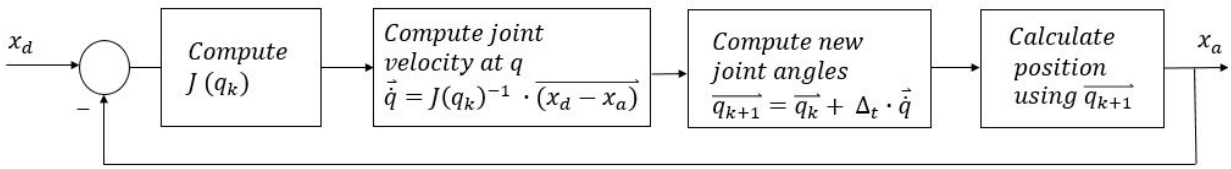

Figure 3.4: The Newton-Raphson Method applied to Inverse Kinematics

V-REP uses this approach for its inverse kinematics solver; except with the Pseudo-Inverse Jacobian, as it intrinsically provides the smallest 2-norm among all solutions, even when the arm is near singularities. When a set of joints is in inverse

kinematics mode, either the Pseudo Inverse or Damped Least Squares method can be chosen, with an iteration number; as seen in Appendix A, Figure A.8. This iteration number is the number of times V-REP will perform the above loop before using the evaluated change in angles to move the joints. Thus, the number of iterations determines the accuracy of these methods, where the more iterations the more processing time is required to narrow down on a more accurate solution.

Due to V-REP's knowledge on the position of joints within a serial chain, using the inbuilt D-H solver the transformation matrix for the arm is determined. Thus, from this the Jacobian, and then the Pseudo-Jacobian matrices are computed for a given configuration, and used to calculate the change in angles required for the 'Tip' object to reach the 'Target' object; given a solver type as specified above. This method will be mimicked in this thesis such that the client program can have the ability to control the joint angles of both V-REP and the physical Kinova Jaco arm.

Chapter 4

Design Approach

This chapter looks into the design of the client program, the decisions and functionality required to complete the program's objectives. Firstly the overall objectives are defined, and then the procedures for completing these are provided. The results chapter will go into detail of how the functionality was implemented, and the successes and failures which arose.

4.1 Client Program Functionality Requirements and Design

Using V-REP's remote API, a client program is required to be developed to allow the external control of the virtual Jaco arm. This program must have the following functionality:

- Be able to control a scene with the Jaco arm with joints either in Forward or Inverse Kinematics modes
- Allow input commands come from the user's keyboard, or from an external joystick
- Allow the control of the Jaco arm by calculating its own inverse kinematics
- Be able to connect to V-REP through a specified IP address and port number
- Be easily able to convert between implemented kinematics modes
- Have data-steaming functionality to enable collision detection
- Be able to quickly retrieve the object handles for the joints of the Jaco arm, for any given V-REP scene before processing

- Be able to identify when the Jaco arm is in an undesirable configuration and have automatic adjustments; an undesirable configuration is when the gripper is facing upwards, as if to grab something from above, instead of out in front.
- Allow real-time manipulation of the virtual robotic arm and V-REP scene

Considering there was a distinct lack of background experience with V-REP and using remote API's, the development of the client program was broken down into small stages. These stages together allowed the full development of the program:

1. Import the Jaco arm into V-REP, add the gripper; additionally, learn how to import models into the scene and create a robotic arm for the interest of familiarising oneself with V-REP
2. Use a familiar coding language to interface with the remote API to communicate with V-REP
3. Determine a method for getting the object names within the scene, to make a mapping between object handles and object names (i.e. it is known that the fourth joint's name will contain the word 'joint' and the number 4, but that doesn't give a way of communicating with V-REP, as the remote API requires object handles.
4. Make the IP address, port number, and scene mode input arguments to the program for quick management
5. Develop the client program to be able to send and receive joint angles from the Jaco arm in V-REP
6. Create an interface for communicating with the client, to be able to transfer input commands to the client; for getting and moving joint angles
7. Create a V-REP scene with the Jaco joints in Inverse Kinematics Group, such that the end-effector tracks a target object
8. Using the mapping for object names to object handles, get the target and tip objects' handle
9. Control the position of the target during a simulation, so the Jaco arm follows the commands.
10. Design an interface for giving keyboard inputs to move the target in the scene
11. Find a way of getting the input from a gaming controller via USB; i.e. 'X' button pressed.

12. From a separate program, create a child process to run the program for getting the controller inputs, and send them to the parent process
13. Integrate this functionality into the V-REP client program, so when the scene mode is Inverse Kinematics, the joystick application process is created and communicates with the client of any input changes.
14. Using MATLAB, form the forward and inverse kinematics solutions using the D-H parameters, analytical inverse method, a control system for reducing angle error using regulators, and the Jacobian matrix for a given position
15. Adjust the tip dummy object to mimic the end-effector position from the forward kinematics
16. Integrate forward kinematics functionality into the client program
17. Integrate the analytic inverse solution into the client program
18. Integrate the control scheme solution into the client program
19. Integrate the jacobian inverse method into the client program
20. Tune the solutions individually
21. Devise a method of quickly swapping between inverse kinematics solutions
22. Optimize the code and program

Through the development of the program, additional stages were added, such as enabling the child process terminate when there wasn't a joystick connected, and allowing the program to then accept inverse kinematics commands through keyboard input. Later, it was discovered that threads cause blocking of other tasks, and shouldn't be kept running. These additional developments will be outlined in the approach section below, as the development of the program is explained.

4.2 Project Development

4.2.1 Set Up

Originally, the program was written in python as it allowed easy manipulation of variables, and passing of information to V-REP. There is plenty of example code for setting up a connection to V-REP from a python program. This allowed an understanding of how the connection was established; V-REP needed to be set to read from a given port, and then from the client program, a call to connect was made

which returned a client identification number, which is used for every interaction thereafter. In the python version, a call to 'import vrep' was made, which handles the remote connection and allows the use of the python library. The script was really basic, it created a connection, got the object handles given the name of the joints of interest, set some parameters and moved the joints. This process is descibed in Appendix A, Figure A.1.

At this stage, a better definition of the problem scope was provided by Dr Singh, which included many different ideas than that expected. Dr Singh made mention of having the scene in V-REP be projected onto a table using a homography, and then interacting with the virtual arm by moving in-front of the projection. From this, I began to add sensors and cameras into the V-REP scene, so a fixed point can be projected; significant delay came with the addition of a camera in the scene whilst moving the arm, and thus this design concept wasn't pursued further. The inclusion of a joystick also became a requirement for this task, in that the end-effector of the arm should be controlled with a joystick. From this, it was decided that to allow interfacing with peripherals, the client program should be written in C. The idea behind this was that an executable can create child processes that are separate to the VREP client, which handle tasks such as reading from the joystick, or projecting the environment onto a table (if that was to be required).

From previous studies, it was known how to develop a C application in the Linux environment, and thus a virtual machine was made. The program was then rewritten in C, which had it's own problems associated with including the external API files in the correct order, and the generation of the makefile; there was limited instruction on which of the many files should be included. The makefile is presented in Appendix A, Figure A.2 for documentation purposes.

The client then became under development, with the first task being to extract the object names and handles from V-REP. Unfortunately there wasn't a remote API function for getting object names, however there was one for handles. Thus, an internal V-REP function needed to be created to return object names given an object handle. This was done in the V-REP main script, and was called in the remote API using the simxCallScriptFunction(). This remote API function expects 4 types of information to be returned from V-REP, (an integer, float, string and a buffer), thus the Lua function returned two empty tables, one containing the object name, and an empty buffer; a copy of this function is available in Appendix A, Figure A.3. Using the blocking communication technique, this function was able to be called for each object handle in the scene, and the name stored.

This process caused a slight delay on start up, whilst the client program retrieved the names corresponding to important joints of the Jaco arm, and mapping them to their respective object handles; all of the information for objects in the V-REP scene

were stored in an struct. Thus, it was desirable to write the mapping for a regularly used scene to a text file once it is retrieved from V-REP, allowing the text file to be used on start-up instead of communication to V-REP when the simulation is run again. To further develop this, the client program will look for a given text file for a scene, and if it does not exist it will get the information from V-REP, and write it to the file for next time; otherwise it will fill the struct with the information from the text file; a copy of which is available in Appendix A, Figure A.4. As the important information from V-REP is the names and object handles of joints, motors and the Target object, this is the only information saved to the file, making it a much faster avenue than requesting from V-REP and sorting.

Now that the client program was able to get the joint handles in the scene, and send desired joint positions. Merging this functionality together, the client became able to automatically associate a joint number with an object handle and then set or get the joint angle from V-REP. To further progress this, the command-line interface was developed to take in forward kinematics inputs, of the form, 'joint number' 'angle to move', with enter being used to store the request. From this, the program was able to set a joint angle of the Jaco arm in V-REP, and return the new angle. This process is outlined in Appendix A, Figure A.5.

From this, it was important to ensure the forward kinematics can be calculated, to ensure the Jaco arm's end-effector position was in the expected position when returned from V-REP. Furthermore, it would be required for the inverse kinematics derivation and implementation.

4.2.2 Forward Kinematics Implementation

Kinova Robotics supplied documentation that assisted with the forward kinematics of the arm. In their $JACO^2$ 6 DOF Advanced Specification Guide, is the D-H parameters, joint lengths and offset angles for the robotic arm. Expressed in the table below are the numerical values for the D-H parameters, for the given axis positions along the arm. It can be seen that the axis for joint 4 isn't at the physical motor position, rather at the angled point in the link, this is to simplify the parameters as is at the bend, pointing towards joint 3, and is merely powered by the motor at joint 4. This occurs again at joint 5, where links 4 and 5 have a bend of 60 degrees, moving the axis of rotation away from the joints. The parameters used in this case are the 'Classical D-H Parameters' from the Kinova documentation, illustrated in Appendix A, Figure A.6. There are other sets of parameters for the Jaco arm; for

instance, the zeroth axis can be turned such that α_0 is 0, and similarly for α_3, where α_2 and α_4 can be $-pi/2$. The classical parameters were used as it matches the Jaco documentation, Peter Corke's Robotic Toolbox, and the set of angles to make the arm stand straight up could be easily determined. Table 4.1 presents the Classical D-H parameters for the Jaco 2, 6 DOF Kinova arm.

Joint	α_{i-1}	a_{i-1}	d_i	θ_i
1	pi/2	0	275.5	q1
2	pi	410.0	0	q2
3	pi/2	0	-9.8	q3
4	pi/3	0	-250.08	q4
5	pi/3	0	-85.563	q5
6	pi	0	-202.78	q6

Table 4.1: Classical D-H Parameters of the Jaco Arm

Using the theory outlined for deriving the end-effector position from a set of joint angles (forward kinematics), the transformation matrix in terms of symbolic joint angles $\vec{q}$ for this arm was calculated using MATLAB. The script for this is provided with this paper, where it can be seen that the transformation matrix is denoted by the variable A. From the figure of the Jaco arm provided, it can be seen that it is in folded over position, whereas in V-REP the Jaco arm begins in a straight up configuration. Thus, there is a difference between the mapping for angles in V-REP compared to the transformation matrix, meaning, for them to be transparent the angles of the transformation matrix need to be offset to mimic that of the virtual arm. Correctly, for the transformation matrix to provide an arm configuration with the gripper at it's tallest point, like V-REP, the angles need to be substituted into the transformation matrix as;

$$\vec{q} = [pi, pi/2, -pi/2, 0, -pi, pi]^T$$

resulting in an end-effector position;

$$\vec{X} = [0, -64.299, 1181.145]^T$$

Thus, for changes in joint angles relative to V-REP's starting position, the following substitution had to be made;

$$\vec{q} = [pi - q_1, \frac{pi}{2} + q_2, -\frac{pi}{2} + q_3, q_4, -pi + q_5, pi + q_6]^T$$

Thus q_i is the offset in angle between one joint and the next, as per the forward kinematics convention provided in the theory section of the report. To separate the V-REP communication and terminal input handling to the mathematics associated with forward and inverse kinematics, functions for these were placed in its own kinematics files (called kinematics.c/.h). Once the forward kinematics solution was correct in MATLAB, the mathematics that described the end-effector position in terms of joint angles was inserted into the file, and given an input command that ran the function and printed the expected end-effector position. This was able to prove that the forward kinematics model and the tip position agreed; after some slight tweaking of the tip position relative to joint 6.

4.2.3 V-REP Inverse Kinematics

After the forwards kinematics was correct, the inverse kinematics methods can be written in MATLAB and derived, to be integrated into the client program. For the integration, the client program had to be set-up to take inverse kinematics inputs, and manipulate the joint angles with V-REP accordingly. In addition to this, the client program needed to be able to control the target object in a inverse kinematics V-REP scene. Thus, the next objective to complete was to get V-REP to recognise the Jaco arm serial chain and compute the joint angles such that the tip follows the target, moved by the client program.

This part was straight forward, an 'IK Group' needed to be created that contained the Target and Tip objects, and the reference frame to be set to that of the Jaco's base. The solver can have constraints on the x, y, z, θ_x, θ_y and $theta_z$; thus the tip can be constrained to the target's position and orientation. The desired precision can also be specified, which is how close the solver's solution can place the tip to the target's position before finishing the interations (before the maximum iteration number is reached). The settings chosen where 0.5mm in translational coordinates, and 0.1 radians for the angles, no restriction on θ_z so the arm always points away from its base, and a maximum number of steps to be 5. The Damped Least Squares solver was selected with damping of 0.052, this provided a smooth transition towards a desired path; whereas the Pseudo Jacobian Method can have significant error around the target before getting a precise enough approximation,

causing scattered movements. More specifically, for a small change in tip to target position of 1.5mm in the y, and 7.6mm in the z, the Pseudo Jacobian method with 5 (and even 50) iterations wasn't able to change the base angle enough for joint 2 and 3 to move into the correct plane to allow the error in y and z be limited. This problem has occurred as V-REP has the Jaco model placed up-right, thus there are singularities with joints 2 and 3 being aligned. The Damped Least Squares method performs better close to singularities, where 5 iterations in this mode can effortlessly move the end-effector into position. Below, Figure 4.1 presents the Jaco arm in inverse kinematics mode with small error in target to tip position. Following is a figure of the results between using the DLS and Pseudo inverse methods. Due to the arm being in inverse kinematics mode, the joint angles cannot be set to place them out of singularities on start up; as V-REP is constantly trying to make the target and tip error fall below the threshold.

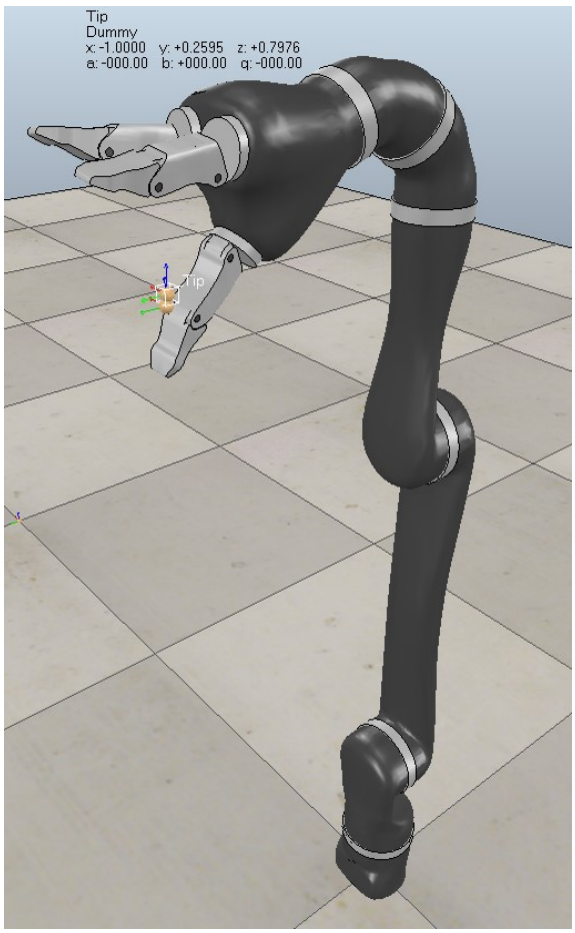

Figure 4.1: The Jaco arm in V-REP with joints set to IK mode, with a small error in target to tip position

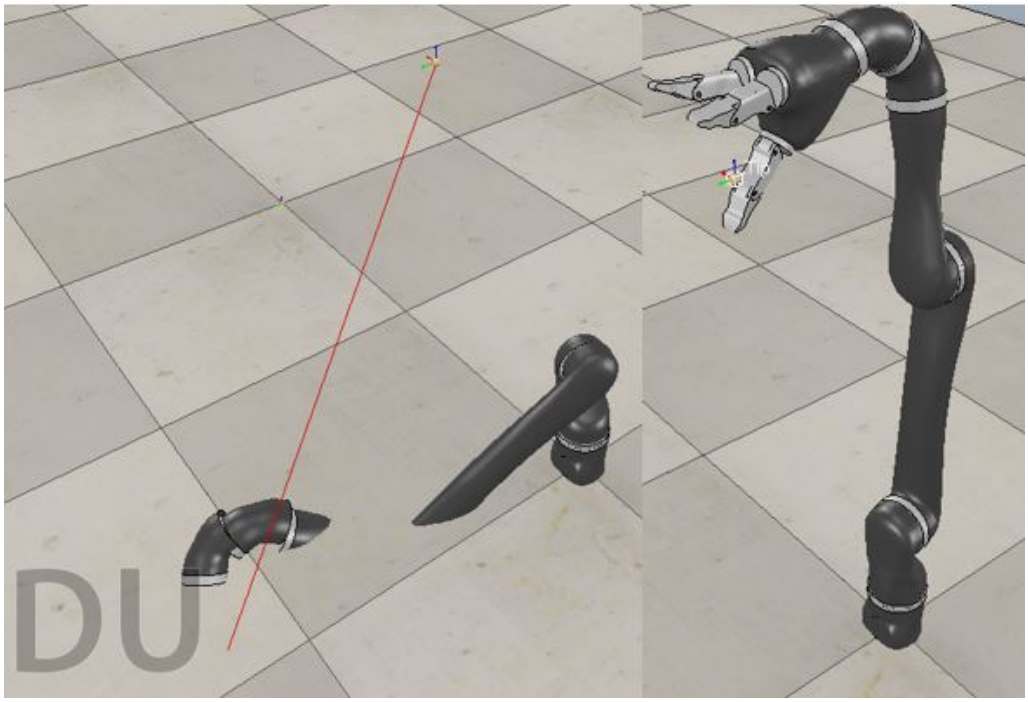

Figure 4.2: Left: the Jaco arm using the Pseudo Inverse Jacobian with 50 iterations. Right: the Jaco arm using DLS of damping 0.05 and 5 iterations

For this robotic manipulator it was found that the DLS method was the best to used with V-REP, and the small iteration number didn't come with any noticeable delay in the program; thus satisfying the real-time objective.

The client program was set up for interpreting inverse kinematics inputs, being keyboard commands; depicted in Appendix A, Table A.1. Given the command, presented in the table below, the target position will be retrieved from V-REP, incremented on according to the command, and then the world position of the target would be reset; moving the arm.

The maths for calculating the new coordinates of the target were quite simple, where $\vec{P_t}$ denotes the position of the tip relative to the base such that;

$$\vec{P_t} = Tip_{world}\vec{C}oorinates - JacoBase_{world}\vec{C}oordinates$$

the 'w' command would have the following computations;

$$radius = \sqrt{P_{t_x}^2 + P_{t_y}^2}$$

$$radius = radius + \Delta radius$$

$$angle = atan2(P_{t_y}, P_{t_x})$$

$$P_{t_x} = radius \times cos(angle)$$

$$P_{t_y} = radius \times sin(angle)$$

It was found that the program could seamlessly control the target in V-REP,and the end-effector would follow.

4.2.4 Joystick Integration

The next objective was to add a joystick as an input source for moving the end-effector. It was assumed that the physical Jaco arm would be set-up to a Windows computer in the Robotics Design Lab at UQ, and as a port is necessary for creating a processes from the client program to read the joystick, it was decided that the client program was required to be developed on a Windows computer; rather than a virtual machine were it cannot directly access USB ports. The task of moving to Windows was time consuming, Linux compilers were first downloaded to Windows such that the program can compile in command-line. Unfortunately the networking on Windows is different to Linux, and so there wasn't libraries for converting Linux network code to a form understandable by Windows. It was decided that a new client program should be created in Visual Studio, and then the client program's custom files have the code copied over.

Only Windows has the ability to control the Human Interface Drivers, however the Windows API allows one to communicate with this via functions such as GetRawInputData(); available with hidsdi.h. Unfortunately a program using Windows API can only communicate with another Window through Window messaging (queues). The client program was a console application, which had the ability to communicate with other applications through a method similar to piping in Linux. Given it was desirable to find an existing joystick interface as an executable which can easily be run by a creating a child process, many platforms were tested to determine the easiest to interface with. The notes taken during this time is provided in the appendix, where Windows API, HTML Gamepad API, LibUSB, and Simple Directmedia Layer (SDL) source code were tested. It was found that SDL provided the best means in interfacing with the Joystick, as the library offered the source code that was editable from Visual Studio. Initially the project, testjoystick, was designed to display information about the joystick, and show which buttons were

being pressed in a Window. It was also discovered that the application had it's own handle to the console, using SDL_log to print to terminal. This meant that usual handles to the terminal such as stdout and stdin had no effect on the console window. Thus, changes were made to the program (it should be noted that SDL is open-source as long as it's referenced and changes made are commented on);

- The code that created and controlled the window was removed; so the application only printed to the control via SDL_log
- The handle to stdout and stdin were redefined; such that stdin pointed to the STD_INPUT_HANDLE, which was opened for reading.
- The code was made to print to stdout "No Joysticks Attached" and exit; so the child process isn't left open when there isn't a joystick.

A new project was made to attempt to spawn a child process using the Microsoft Developer Network (MSDN); libraries accessible from Visual Studio. With reference to the example code on the CreateProcess function, information successfully flowed between the testjoystick application and the new project file. This code was then copied into the client program, in the file joystick.c. This file was used to store all code relevant to the joystick, such as creating the process, reading from the process, interpreting the button presses and joystick positions, adding the relevant information to the input buffer, and the handing of passing commands from the buffer to the main client thread. A total of 3 threads were used, with two made to service the commands of the joystick; these are detailed in Appendix B, Table B.1.

It can be seen in the below figure that the output of the joystick process (SDL_log prints that were provided, but same information that is sent to the client program) shows the axis being toggled as a number between 0 and 3 (two axis per joystick) with 16-bit signed integer representing the amount it's pushed on that axis. Thus, when a joystick axis is held and not returned to zero, the output of the joystick process only shows that it's reached 32797, but doesn't keep printing to show that it's still held there; this can be seen in Appendix A, Figure A.7. For this reason an additional thread was created to check the joystick buffer, and re-add commands after a period without change. It was found that 750 ms was the optimal time to wait between adding to the buffer, as it took on average 600 ms for a command to be interpreted sent to V-REP. This average is CPU speed dependent, and thus changes from computer to computer; it is similar to the sensitivity of the joystick.

With the joystick successfully moving the tip position in V-REP, the main objective of having an external tool control the robotic simulation was complete. It was now a manner of getting different inverse kinematics methods working with the V-REP client program. This was to enable the V-REP scene can be in forwards

kinematics mode, and the client program move the joint angles to provide a desirable end-effector position. The chosen methods for the client program were, the analytical approach as this would require the least number of calculations, a control systems approach, and using the inverse Jacobian with Newton-Raphson's method. Three were chosen to determine which was the best given the design objectives.

The client program was further developed during this stage, to incorporate commands for getting the world coordinates of objects, pausing communication between V-REP and the client to allow all joint angles to be sent at one time, streaming mode was trialled with the objective of updating the client's knowledge of the joint angles, and setting up the interface having a forward kinematics scenes with inverse kinematics inputs, and a way of quickly flicking between the different inverse kinematics solutions.

Chapter 5

Kinematics Approach

5.1 Forward Kinematics

As aforementioned, the forward kinematics was derived using the D-H Parameters provided by Kinova and by following the theory section of this report for the transformation matrix. Due to the amount of symbolic expressions, MATLAB didn't ignore numbers close to zero. Thus, before using the transformation matrix, each coefficient in each index was filtered and extracted if it was smaller than 0.000001. The matrix expressions are still too long to be displayed here, however it is available in an attached MATLAB scripted titled forwardKinematics.m and in the kinematics.c file for the client program. It was found that the transformation could accurately determine the position of the end-effector given a set of angles; this can be checked by typing 'fk' as an input command for the client program, and checking the coordinates of the tip object in V-REP.

5.2 Inverse my Matrix Manipulation

In order to take the strain off V-REP doing all the computations, and to give the control of either forward or inverse kinematics control to the client program, three inverse kinematics methods were integrated into the client program. The first attempt at an inverse kinematics solution was to derive the relationship between joints through manipulations of the transformation matrix, as represented in equation 3.11 from section 3.4.1.

After rearranging the matrices between the left and side and right hand side, and inspecting the indexes of both sides for equalities, no fruitful results were found. For instance, when observing the results of both $A_4^5 * A_5^6$ compared to $(A_0^1 * A_1^2 * A_2^3 A_3^4)^{-1} A$, each index of both matrices were mathematically identical; any variances could be corrected with trigonometric identities.

5.3 Inverse by Analytical Approach

To begin this approach, the Jaco arm was redrawn in different configurations to determine how the varying each joint can change the end-effector position. Following the approach outlined in the theory section of this report, it can be seen that the second link forms a triangle with point $\vec{P}$, formed by the joint angles q_2, q_3 and the joining joint lengths. An angle of 60 degrees is always present between the rotational axis of q_4 and link $\vec{PR}$, similarly with q_5 and link $\vec{RS}$. In Figure 5.1, provided below, $q_4 = \frac{\pi}{2}$ and $q_5 = \pi$, which leads to the end-effector remaining in the x-z plane; the angles contribute no y-component.

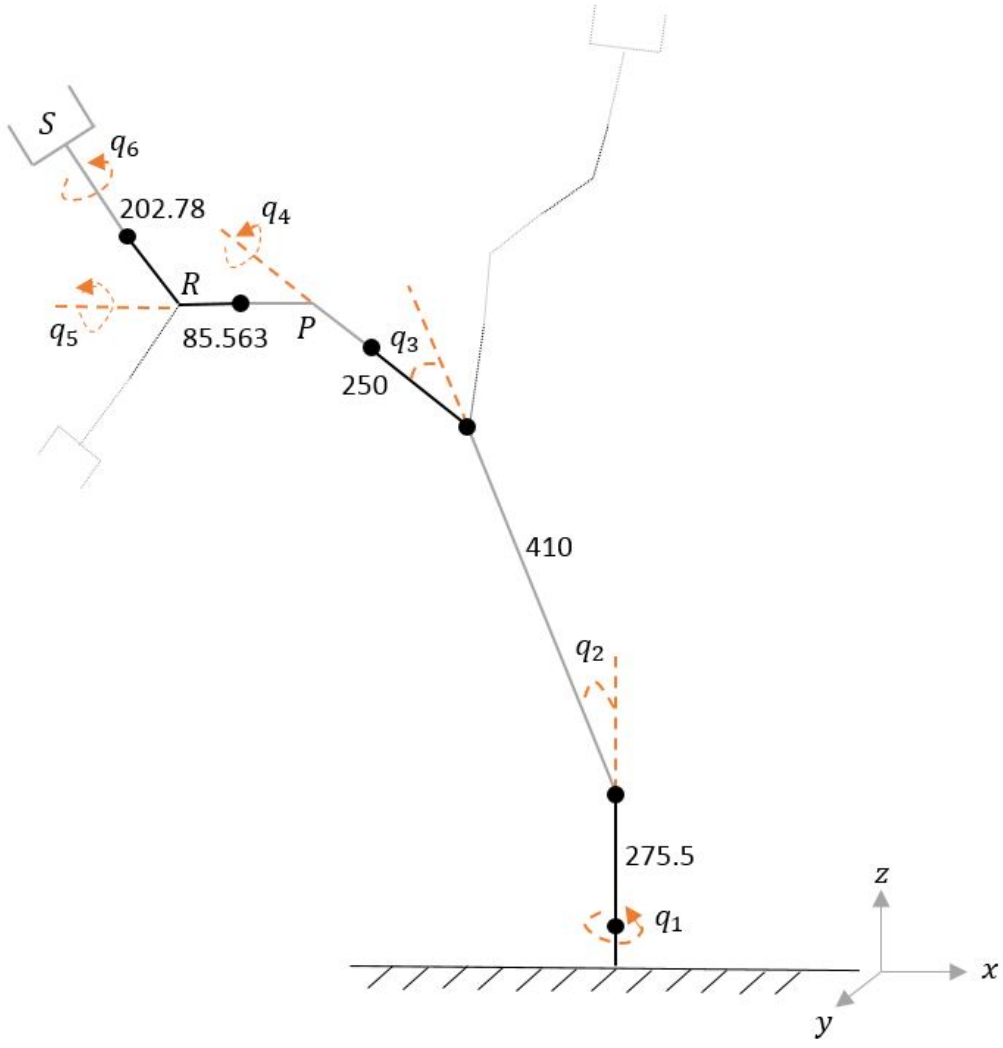

Figure 5.1: Dimensions and angles of Jaco Arm

Varying these two angles in any means will result in a decrease in angle visible in this figure, between the rotational axes and the links; as part will form into the

y-z plane. This is better illustrated in Figure 5.2, where these angles are denoted α and β respectively. Instead of considering point $\vec{P}$, the axial coordinate due to the 60 degree links were added to the link prior, moving point $\vec{P}$ to $\vec{P'}$.

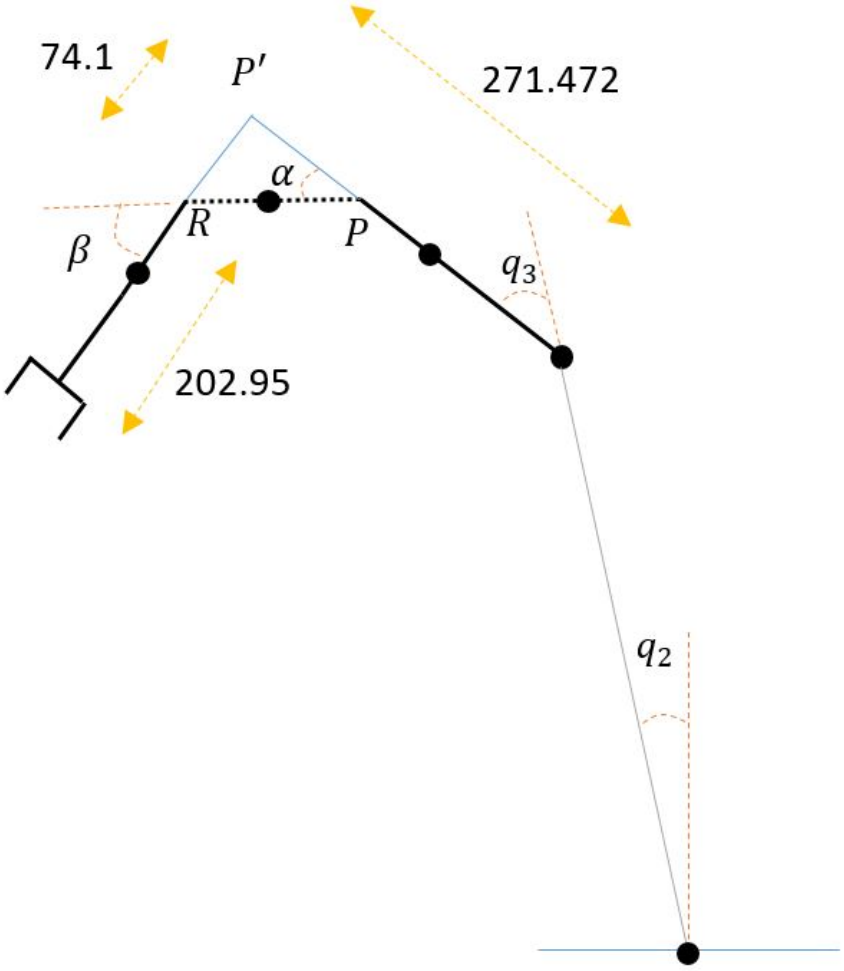

Figure 5.2: Simplification of Jaco arm geometry

Limiting α and β to 60 degrees, results in the following relationships;

$$\alpha = \frac{q_4}{6}$$

$$\beta = \frac{q_5 - \frac{\pi}{2}}{6}$$

Focusing more on the wrist, the lengths of the proposed vectors can be determined; note that D_4 and D_5 are known from the D-H parameters of the arm.

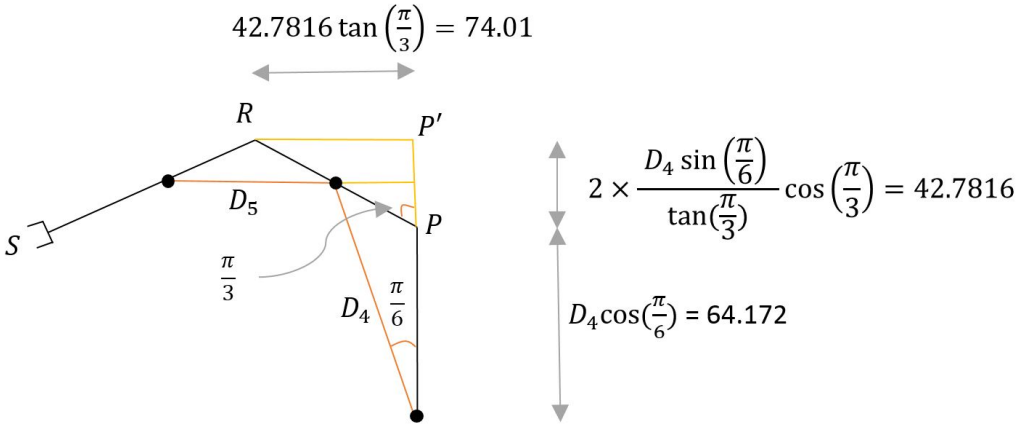

Figure 5.3: Simplification of Jaco arm wrist geometry

From Figure 5.3 it was found that $\vec{PR} = 85.563$

The point $\vec{R}$ can thus be written in terms of point $\vec{P'}$ and the affect of the arm angles contributing length in the x-z plane such that;

$$R_x = P_x + 85.563sin(q_2 + q_3 + \alpha) \tag{5.1}$$

$$R_y = P_y + 85.563sin(q_2 + q_3 + (\frac{pi}{3} - \alpha))$$

$$R_z = P_z + 85.563cos(q_2 + q_3 + \alpha)$$

Similarly for the end-effector position $\vec{S}$ in terms of $\vec{R}$

$$S_x = R_x + 202.95sin(\beta + \alpha + q_2 + q_3) \tag{5.2}$$

$$S_y = R_y + 202.95sin((\frac{\pi}{3} - \beta) + (\frac{\pi}{3} - \alpha) + q_2 + q_3)$$

$$S_z = R_z + 202.95cos(\beta + \alpha + q_2 + q_3)$$

From the theory section, equation 3.2, it is known that;

$$P_{xy}^2 + (P_z - D_1)^2 = 410^2 + 250^2 - 2 \times 410 \times 250 \times cos(q_3)$$

$$\rightarrow q_3 = cos^{-1}(\frac{P_{xy}^2+(P_z-D_1)^2-410^2-250^2}{2\times410\times250})$$

and from equations 3.4, 3.5, 3.6,

$$cos(q_2) = \frac{P_{xy}(410+250sin(q_3))+250sin(q_3)(P_z-D_1)}{410^2+250^2+2\times410\times250cos(q_3)}$$

$$sin(q_2) = \frac{-P_{xy}\times250sin(q_3)+(P_y-D_1)(410+250cos(q_3))}{410^2+250^2+2\times410\times250cos(q_3)}$$

$$\rightarrow q_2 = atan2(\frac{sin(q_2)}{cos(q_2)})$$

The angular position of $\vec{P}$ in the x-y plane is dependent on q_1, and the radial position from the base in x-y coordinates is dependent on the angles of q_2, q_3, as seen above. Thus, the expression for q_1 in terms of $\vec{P}$ will have the form (a constant offset of angle occurs do to the arm's orientation in V-REP);

$$q_1 = atan2(P_y, P_x)$$

Therefore, given a desired position for $\vec{P}$, joint angles q_1, q_2 and q_3 can be determined. From this position and a known end-effector position, $\vec{S}$, the angles α, β, can be determined and thus q_4, q_5;

Substituting out $\vec{R}$, by equating equations 5.1, 5.2, it is known that;

$$sin(q_2+q_3+\alpha)85.563 + P_x = S_x - 202.95sin(\beta+\alpha+q_2+q_3)$$

This can be repeated for the y and z coordinates, however it can be seen that the relationship between $\vec{P}$ and $\vec{S}$ depends on all the angles being known, or, both $\vec{P}$ and $\vec{S}$ being known; not only $\vec{S}$, unless α, β are known, or more relationships between the joints are found. This is not a dead-end however, as it is known that the relationship between $\vec{P}$ and q_1, q_2 and q_3 exists, and if α, β are static between movements, it can be found that;

$$\Delta\vec{P} \propto \Delta\vec{S}$$

Thus, knowing the change in position required at the end-effector, the change in coordinates of $\vec{P}$ can be approximated to move the end-effector to the desired position. Discrepancies will depend on the angle of the arm joints, however this method will bring the end-effector close to the desired point. Any variation in position can be corrected using feedback control on the approximated angles q_2, q_3, calculated as per the above formulas. Figure 5.4 presents the controller that was used in design;

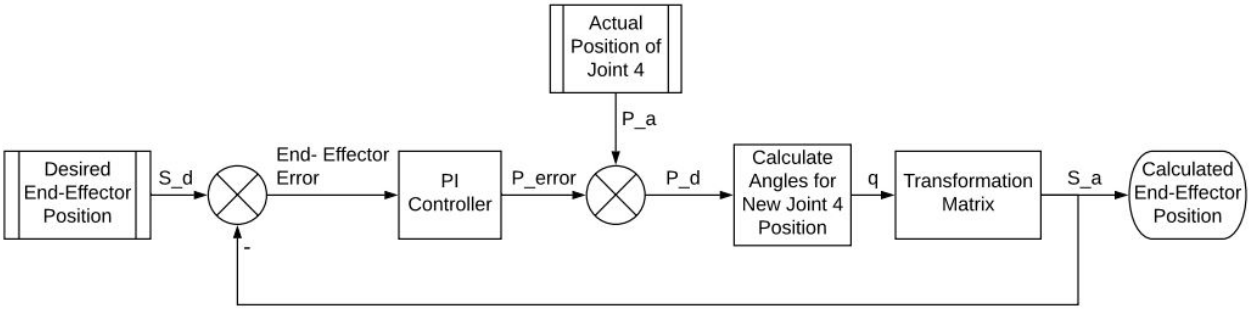

Figure 5.4: Control loop for approximating the joints angles to move the end-effector using the position of the fourth joint

The controller gains are smaller than 1 to reduce the error being added to the position of the fourth joint, such that a solution is approached with each loop iteration. Integral control allows the error to be added with each loop, reducing steady state error, whilst the proportional gain allows the required state be reached quickly.

In the MATLAB script, inverseKinematics.m, the steps for calculating the required angles for a desired end-effector position is presented; following the maths presented above. After verification of this process, the code was written in the client program to tune the gains of the controller, and add any angle offsets present between theoretical calculations and the joints of the arm in V-REP. The PI controller was able to be tuned by writing the calculated end-effector position to a spread sheet (stored in programming/client/x62/Debug) each loop iteration; allowing the plotting to view the response. For this design, an accuracy of 6mm in each component of $\vec{S}$ was required, with a maximum loop iteration of 20. It was decided that the K_P and K_I gains for each axis should be equal, as it is desirable for the arm to weigh each axis equivalently. The response with the lowest overshoot and steady-state error was found to have gains of $K_P = 0.08$ and $K_I = 0.2$, with the response of each axis being presented in Figure 5.5; and further in the appendix. It can be seen that the responses converge to the desired values in practice. In code, bounds were placed to ensure the control loop was not forcing the arm to reach an unreachable end-effector position.

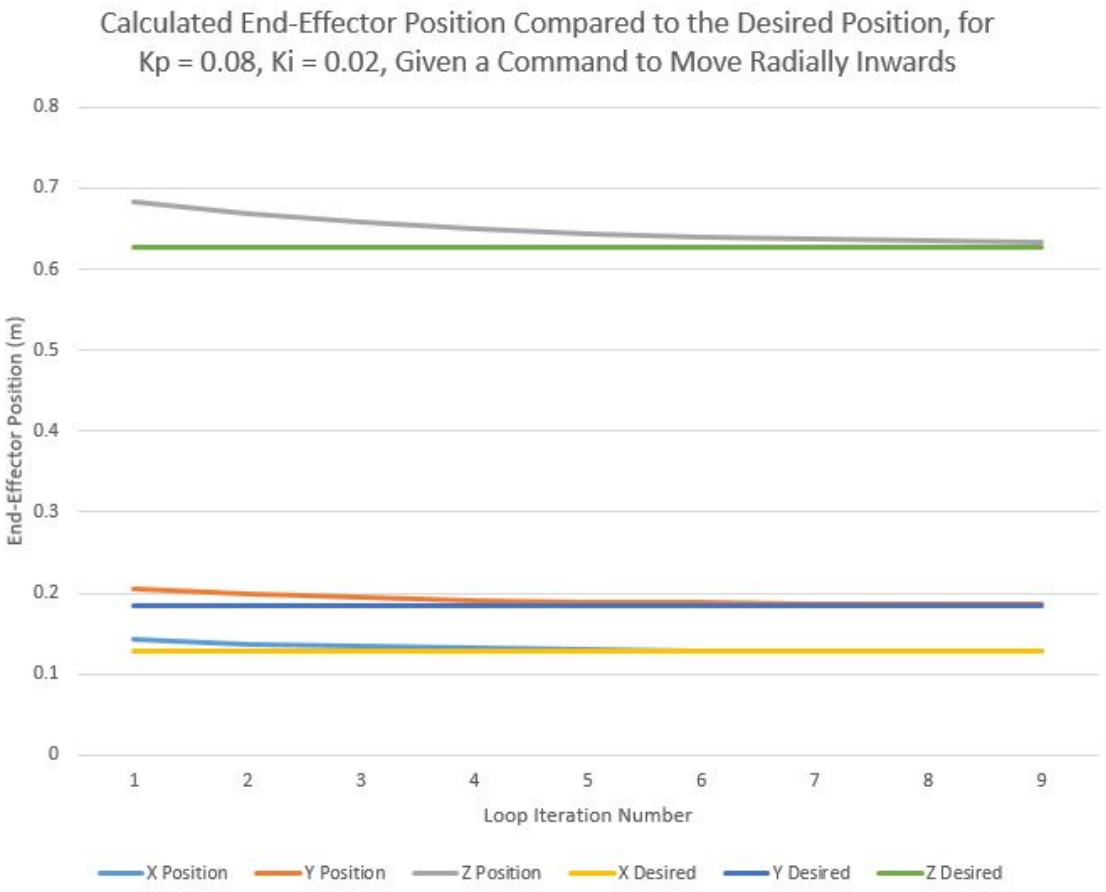

Figure 5.5: Response of Chosen Controller Given a Change in End-Effector Position Command (3mm in the x-y plane)

5.4 Inverse by Control Loops

The success in the former section called for a trial of using control loops to govern the change in angles required for reducing the error in end-effector position. A key problem with this form of control is that there is no method of checking for undesirable joint configurations; such as the arm folding onto itself. Due to this system being non-linear, the controller response will depend on position, and thus will only be used for correcting small errors in end-effector position. In MATLAB Simulink the original model was created, where the inputs were the desired end-effector coordinates. To give the system a localized approximation, the first 3 angles were calculated using the inverse kinematics described in the previous section. To make a start, the 6 joint angles had a PI controller with proportional gain being a fraction of their following link lengths. The purpose of this was that, for example, the second link is the longest at 410mm, and thus small angles of q_2 will effect the end-effector position more than q_3, which moves a link of length 200mm; thus the gains for controlling q_3 are more than that for q_2. The error being corrected was not the difference in x, y, z coordinates between the desired position S_d and actual position S_a of the end effector. Instead, the change in angle between these locations in the x-y plane, and ground-z plane were used; this was to decrease the probability of having inaccurate error from changing between quadrants. The use of angles also limits the error to between 0 and 2π, which is the same range as the angles for the robotic arm, whereas if displacements were used the error would be linear and large, differing significantly to joint angles. The angular displacement error, θ_z, θ_{xy} are illustrated in the Figure 5.6.

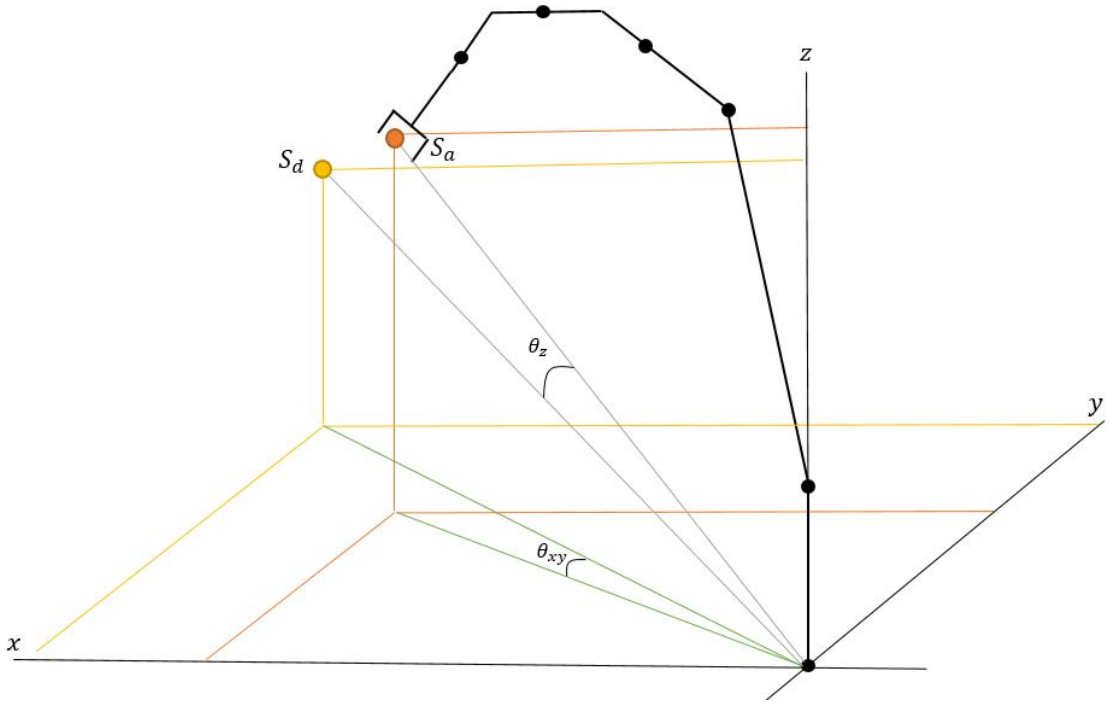

Figure 5.6: Error being corrected in control loop

It is known that q_1 does not effect the z coordinate of the end-effector position, and is the largest contributor to motion in the x-y plane. The dominant response was desired to be the elimination of error θ_z, where θ_{xy} will be corrected thereafter. To control q_1, a high integral gain was used, such that the end-effector position will slowly be regulated for steady-state error in the x-y plane.

The error inputs to the PI controllers for each joint was determined by the amount of which the joint would contribute in that angle. For instance, joints 4 and 5 equally contribute to the error of θ_z and θ_{xy}, thus the error into the controller is the summation of these errors. Angles q_2 and q_3 were given the largest proportional gains, as it is desired that these joints move the most.

It was quickly found that the system could become unstable given the incorrect parameters for one of the controllers, but the design process itself was a good starting point.

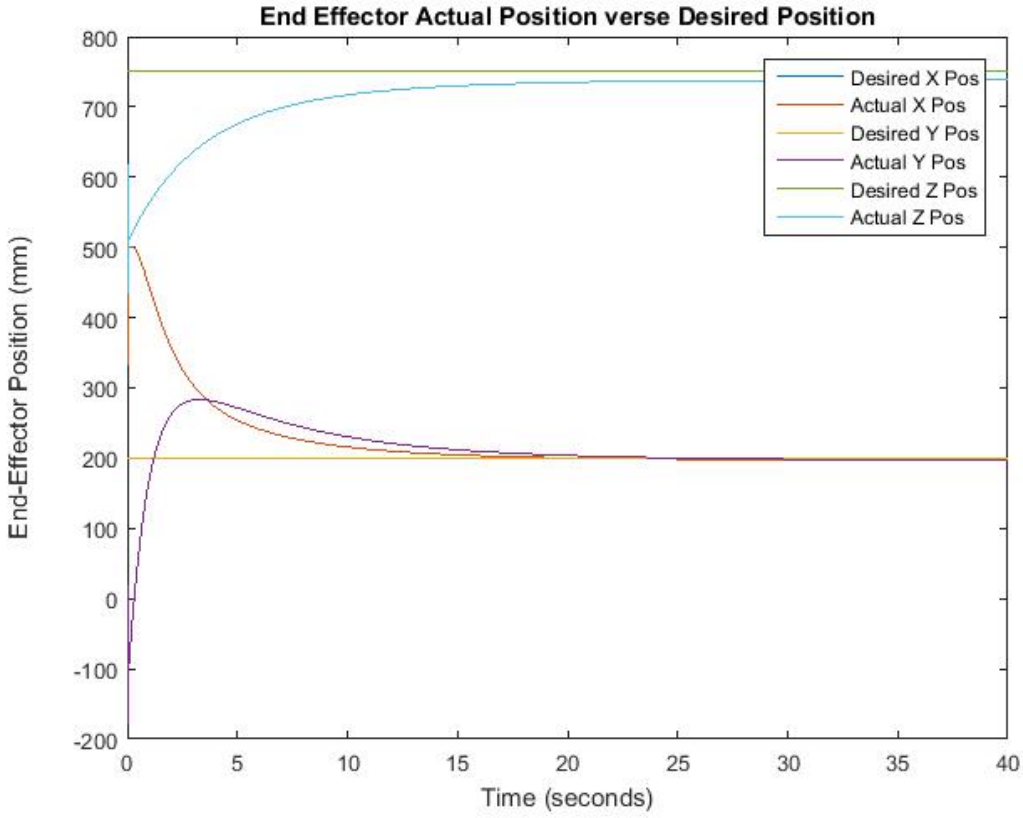

Figure 5.7: The desired and actual positions of the end-effector with each joint under control

It can be seen in Figure 5.7 that the error in end-effector position was correctly adjusted, moving the arm from its reset position defined by the D-H parameters to the desired location of $S_d = [200, 200, 750]^T$. Unfortunately, as the control is on two degrees of freedom, and the end-effector's position is in 3 dimensional space, there is uncorrected steady-state error in the z-coordinate.

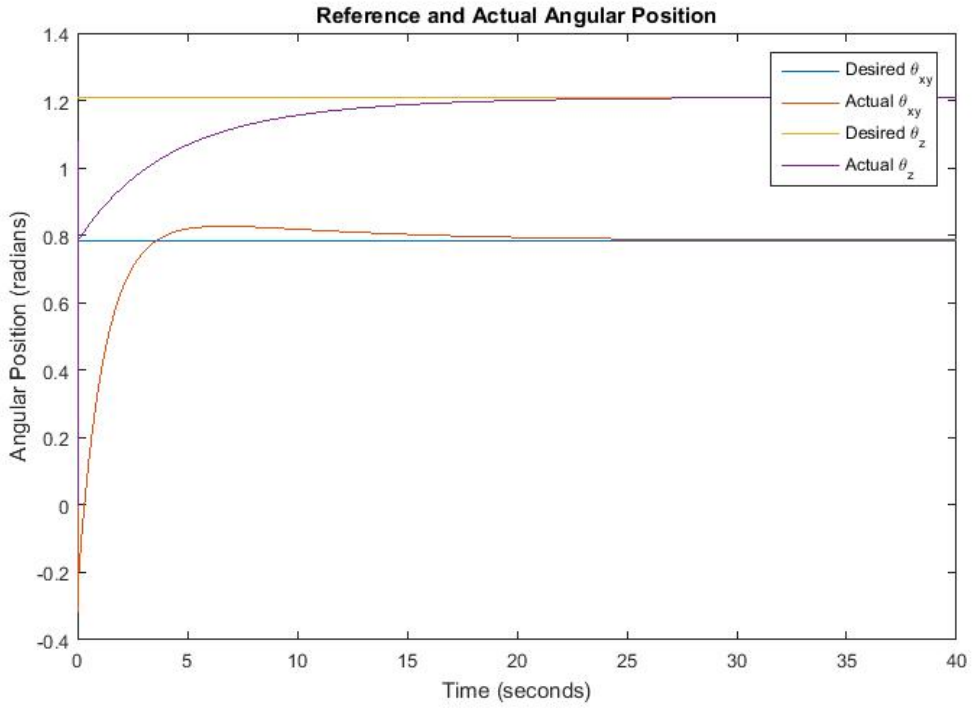

Figure 5.8: The desired and actual angular positions of the end-effector

It can be seen in Figure 5.8, above, that the desired angular positions were achieved, however, from this prospective there isn't control on radial distance, which lead to stead-state error; thus error in at least three states, angular or linear displacements, need to be used used. Another issue with this controller is that the convergence of θ_{xy} is faster than θ_z, which is the opposite to that of which is desired, and is a direct consequence of incorrect implementation a reference position correction with integral gain. A high-level representation of the controller used to vary each angle is presented in Figure 5.9.

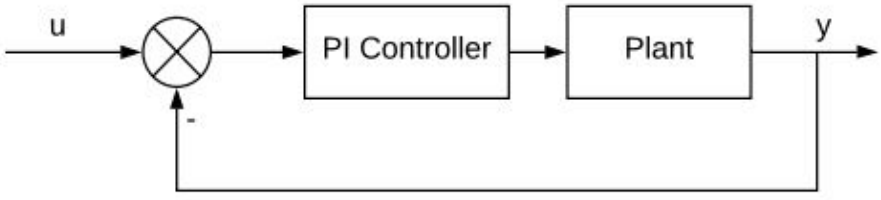

Figure 5.9: Standard Regulator Block Diagram

Instead, it would be desirable to use 'integral action', shown in Figure 5.10, as it provides a reference position that eliminates steady-state error, and is robust to disturbances.

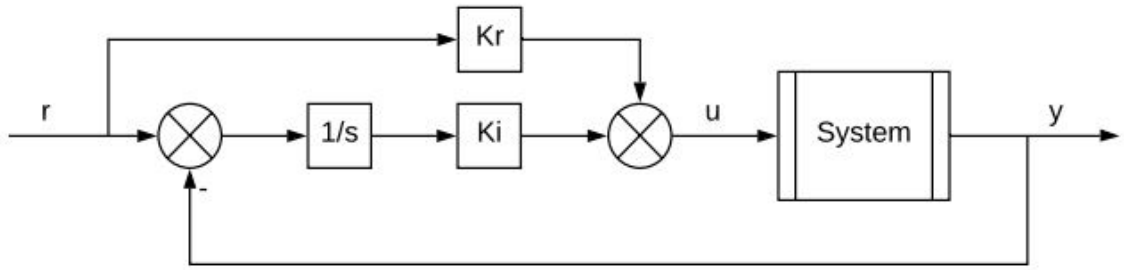

Figure 5.10: Block Diagram of integral action, giving the output (y) a reference input (r)

A separate method to achieving the desired response would be to have a cascade loop, regulating θ_z in the faster inner loop, and θ_{xy} in the outer-loop, as presented in Figure 5.11.

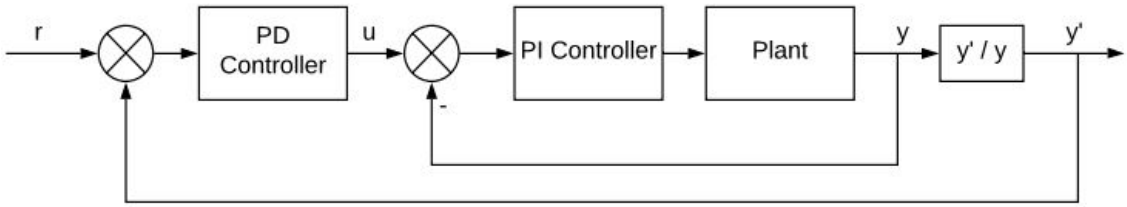

Figure 5.11: Block Diagram of a Standard Cascade Regulator

The proposed control system was incorporated into the V-REP client program, and the parameters further tuned by exporting the position of the end-effector to a spread sheet, which allowed plotting of the responses. Provided in Appendix A, Figure A.9, is a graph showing the error of the controller that was designed within the client program, where an increase in radial distance of the end-effector by 3 mm was desired. It can be seen that steady-state error occurs in θ_z, as found in the simulation, however θ_{xy} reaches and stays at exactly 0.

Appendix A, Table A.3, presents the chosen controller gains, and the percentage of error in angle contributed to the total error seen by each controller.

5.5 Inverse by Jacobian

Within this theory section of this report is was found that the Jacobian can be found from the transformation matrix for a robotic manipulator. Following the creation of the transformation matrix for the Kinova Jaco arm, the Jacobian was approach was trialled; in the MATLAB file 'jacobianInverseKinematics.m'.

The first step was to retrieve the expressions that describe the change in linear and rotational positions of the end-effector due to a change in each angle.

1. Firstly, the rotational and translational matrices were extracted from the transformation matrix
2. The components of the Jacobian representing the change in translational co-ordinates was filled by deriving the three position states by each joint angle
3. From this, the rotational matrix can be derived with respect to each angle, creating $\frac{\delta R}{\delta q_i}$
4. The Skew matrix was created, by $S(\omega) = \frac{\delta R}{\delta q_i} \times R^T$
5. ω_x, ω_y and ω_z were extracted from the skew matrix; representing the change in rotation of the end-effector due to a change in angle q_i
6. The rotational components were added to the required position within the Jacobian matrix.
7. With the Jacobian matrix complete and in symbolic form, substitution of angles of the Jaco arm in a known, not singular, position was substituted into the Jacobian.
8. The Jacobian could now be inverted; note that the symbolic representation of the Jacobian is overly complex, a symbolic representation of the inverse Jacobian could not be computed.
9. Substituting these same angles into the transformation matrix leads to the end-effector location in this configuration known.
10. A vector representing $\frac{dX}{dt}$ needed to be created, the arbitary position $[0, 0, 3, 0, 0, 0]^T$ was used, representing only a change in height by 3 mm; it was found that small changes in position lower than 1 took more iterations to get an accurate approximation.
11. The change in angles were calculated $\frac{dq}{dt} = J^{-1}\frac{dX}{dt}$.

12. This change in angle should be multiplied by the change in time (0.025 was used); in a sense this is a proportional gain that restricts over-shooting the desired end-effector position by incrementing in too large of angles. Thus the change in angle is known to be: $\Delta q = \Delta t J^{-1} \frac{dX}{dt}$

13. The approximated angles for the desired end-effector position could be calculated by $q_{k+1} = q_k + \Delta q$

14. Substituting q_{k+1} into the transformation matrix could provide the new location of the end-effector.

15. Re-calculation of the desired change in position, $\frac{dX}{dt}$, is required to narrow down on an accurate approximation of $\vec{q}$ as per the Newton-Raphson Method; $\frac{dX}{dt}$ = starting position + desired change in position - new calculated position of the end-effector.

16. Steps 7 to 15 were repeated 50 times to narrow down on an accurate approximation of joint angles for the desired end-effector position.

The results of this process is illustrated in Appendix A, Figure A.10, where it can be seen that the z-coordinate of the end-effector converges towards the desired position. It can also be noted that as the error in the x and y positions starts to diverge, the method begins to return it to the desired locations. Considering the displacements are in millimeters, this approximation of the end-effector position is usefully accurate for this thesis.

Unfortunately however, as aforementioned, the symbolic representation of the Jacobian calculated for this robotic arm is lengthy and consists of harsh trigonometric expressions. A very small sample of the sixth row, first column, of the Jacobian is provided below; which can be seen to have conjugates of angles.

$$\left(1.0\,\cos\left(\overline{\mathrm{q6}}\right)\,\left(0.75\,\sin\left(\overline{\mathrm{q4}}\right)\,\left(\cos\left(\overline{\mathrm{q1}}\right)\,\cos\left(\overline{\mathrm{q2}}\right)\,\cos\left(\overline{\mathrm{q3}}\right)+\cos\left(\overline{\mathrm{q1}}\right)\right),\right)\right)$$

Unfortunately these expressions cannot be written in C; unless code for conjugate is written. Thus, this Jacobian matrix cannot be easily integrated into the client program. To simplify the Jacobian, the rotational states of the end-effector were excluded from the matrix, reducing the amount of constraints forming the solution. This leads to the Jacobian being rank deficient, which can be overcome with the use of the Pseudo-Inverse Jacobian method; outlined in theory. This method forms a square matrix of size 6, which describes an under-actuated robotic arm. The same process as outlined above was used, again, before inverting the symbolic variables were substituted with the joint positions. It was found that this method did not require step 16, as in the approximation did not require re-iteration to provide a

better result, as the end-effector position was within the required accuracy. This significantly reduced the computation time of the solution, and the symbolic Pseudo Jacobian was less complicated than that previously found.

For a change in position $\Delta X = [0, 0, 3, 0, 0, 0]^T$, from an end-effector position of $S_a = [344.4863, 219.7989, 488.1665]^T$, the calculated position was found to be $S_{calc} = [344.4755, 219.79189, 491.1696]^T$. The change in angles for this new end-effector position was;

$$\Delta q = [0.0001232, -0.0024076, -0.00240756, 0.000061, 0.00020, 0]^T$$

Therefore, it was found that this provided an adequate solution to the inverse kinematics problem, that could be recreated in the client program; MATLAB can convert expressions to C code given they are simple enough. From this, the symbolic representation of JJ^T was copied into the client program as a 2-D array of doubles.

When the client program is working with a forward kinematics scene, it moves the Jaco arm into a configuration without singularities; this configuration is presented in Figure 5.12.

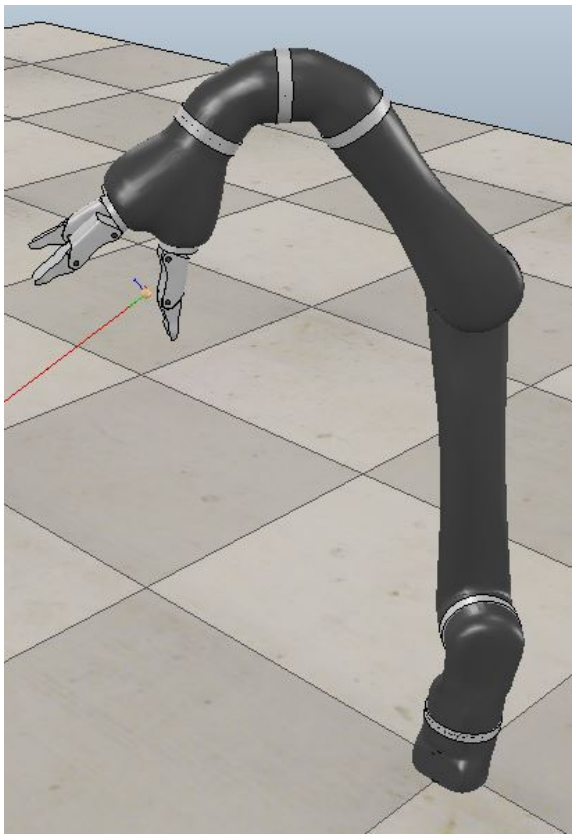

Figure 5.12: Starting position of the Jaco arm when giving IK inputs to a FK scene

This configuration corresponds to the angles, referenced to the D-H configuration, of

$$q = [-8.6394, 1.74581, 10.47175, 4.71239, 0.000001, \pi]^T$$

Substituting these angles into the matrix elements gives a result equivalent to that from MATLAB for JJ^T.

$JJ^T =$

$$\begin{pmatrix} 1.4889e5 & -3.81914e3 & -1.3779e2 & -9.6326e4 & 6.77427e4 \\ -3.81615e3 & 3.0065e5 & -1.15784e5 & 4.24957e-1 & 3.113135e-1 \\ -1.37792e2 & -1.15784e5 & 9.901883e4 & -1.488897e-1 & -7.42055e-2 \\ -9.6325e4 & 4.24857e-1 & -1.488897e-1 & 6.235711e4 & 4.3853e4 \\ -6.77426e4 & 3.11313e-1 & -7.42055e-2 & 4.38533e4 & 3.08403e4 \end{pmatrix}$$

As per the theory, the next step is to inverse JJ^T, and then multiply this by J, resulting in $J(JJ^T)^{-1}$. The inverse of this matrix is computed by calculating the matrix of minors, matrix cofactors, adjugate, and then dividing through by $1/det(JJ^T)$; the general process for calculating the inverse of a nxn matrix in C. Before starting this, the determinant of JJ^T is calculated to ensure the matrix is invertible.

In the V-REP client program, the determinant is calculated to be -1.723682e-29 for the starting configuration of the arm. For this same position, MATLAB calculates the determinant to be -6.90106775e-43, thus the calculations are off by a power of -13 before the inverse is calculated. As the determinant is not equal to 0, the inverse was calculated, resulting in a matrix with coefficients magnitudes different to that computed in MATLAB for the same angles. Occasionally, the determinant is calculated in the client program to be $-inf$ despite the arm not being moved. This leads to the belief that the smallest of variances in angles can lead to incorrect results; a direct result of the precision in calculations between MATLAB and C. In the client program, the code has been written for this implementation, but was found to be unsuccessful when it came to computing the inverse of the Pseudo Jacobian, and thus was unable to be used; the client program can enter the functions for these calculations, and display the results, when receiving IK inputs and 'Mode 3' is active.

Chapter 6

Results Summary and Discussion

6.1 Jaco Arm Control

The Jaco arm was successfully loaded into the V-REP scene, and was able to be set to both inverse and forward kinematics modes. In the inverse kinematics mode, it was found that control over all six position states was enabled; the tip object could be set to mimic the target object in both translational and rotational position. The focus was to create a means of moving the end-effector from one set of coordinates to another, which was implemented by changing the target's translational position; with the only restriction on angle being that the gripper pointed in the same direction as the target is in the x-y plane. No means of changing the orientation of the end-effector by providing an input was integrated into the client program, however there is a function within the client program designed to vary orientation of objects within V-REP. It may be beneficial to set the joystick up to swap between position control and orientation control, such that the joysticks can either toggle a change in position (as current) or toggle a change in target orientation. To switch between modes, a button on the controller could be pressed; already an output from the joystick process.

It was found that there is no way to change joint angles once V-REP identifies a set of links as an inverse kinematics group. By default, the arm is configured in the upright position with singularities occurring at joints 2 and 3, the Pseudo-Inverse kinematics method for moving the end-effector to the target position could not be used. V-REP's Damped Least Squares inverse kinematics solver however was able to correctly move the Jaco arm despite this, and within 5 iterations always stably moved the end-effector to the target position, in real-time. Using this method with the joystick proved that the Jaco arm was able to be controlled using a joystick seamlessly.

Once the transformation matrix for the arm was determined, the tip object from

the inverse kinematics method was able to be used to indicate the position calculated by the forward kinematics. Thus, it was found that the forward kinematics defines the movement of a point slightly above the last joint of the gripper's 'thumb'. By typing 'fk' as input into the client program, the translational component of the transformation matrix is used to compute the tip position, which is accurate to 0.5 of a millimetre in each coordinate. The transformation matrix in this case is formed using the classic D-H parameters from the Jaco arm documentation, and the translational components were extracted from MATLAB, however there is a separate set of D-H parameters included in the client program which are used to calculate the transformation matrix. With the former method producing accurate results, the calculation of the transformation matrix in the client program is not used.

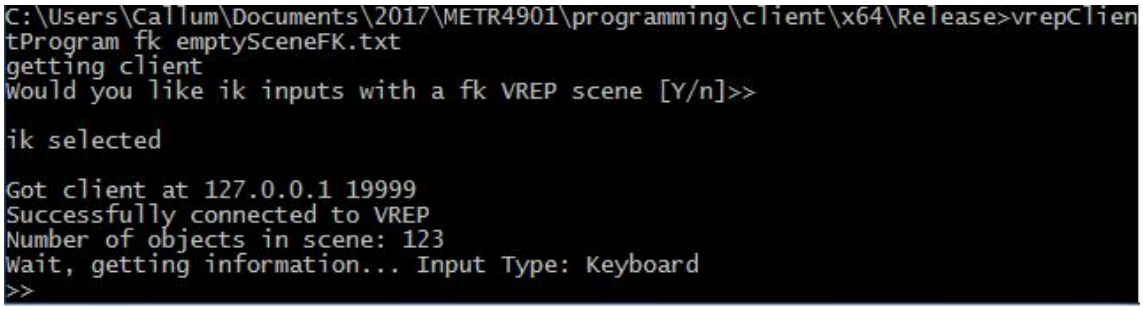

Figure 6.1: Client Program on Start-Up

The input arguments of the client program are;

vrepClientProgram.exe fk/ik [object filename] [IP Address] [port]

On start-up, the client will connect to local host on port 19999 unless a IP address and port is specified in the program usage. Once connected, it will prompt for the user to wait while it creates the joystick process and determines if a joystick is connected. When a joystick is connected, it will flag that it found the joystick and is ready for commands, otherwise it will prompt for a keyboard input; this process is shown in Figure 6.1. The input commands can either be for forwards kinematics for moving a joint angle a given amount of degrees, or inverse kinematics commands that move the end-effector set distances radially from or circularly around the base of the arm. If the user doesn't specify an object filename, it will use the hard-coded object handles for the joints; which may be different if a new scene is created and the Jaco arm isn't the first object to be imported, however this process is shown in Figure 6.2. When an input argument to the client specifies that the 'scene' (Jaco arm in V-REP) is in inverse kinematics mode (specified by 'ik'), the target object will be moved and

V-REP will calculate the angles to follow. If a scene isn't specified or the input arguments are incorrect, the user will be prompted for each argument individually. When 'fk' is specified as an input argument, the user will be asked whether they desire input arguments of type inverse or forwards kinematics. The separation of these is due to the separation of giving forwards kinematics inputs when the scene is in forwards kinematics mode compared to when the scene in in inverse kinematics mode. Additionally, when the program is set to take inverse kinematics inputs but the scene is in forwards kinematics mode, the Jaco arm needs to be moved into a usable configuration for the inverse methods used; without singularities and with joint 4 of the Jaco arm in the same quadrant as the gripper, with the gripper further out radially.

```
C:\Users\Callum\Documents\2017\METR4901\programming\client\x64\Release>vrepClien
tProgram
getting client
Usage: vrepClientProgram.exe fk/ik [object filename] [IP Address] [port]
Please specify a scene Kinematics mode (ik/fk)>> fk
Would you like ik inputs with a fk VREP scene [Y/n]>>n

fk selected

Got client at 127.0.0.1 19999
Successfully connected to VREP
Number of objects in scene: 123
H = 18
H = 21
H = 24
H = 27
H = 30
H = 33
Wait, getting information... >> Input Type: Keyboard
```

Figure 6.2: Client Program given incorrect arguments on start-up

6.2 Joystick Implementation and Processing Time

Several methods of getting the commands from the joystick were trialled before the SDL solution was found. A comparison of these is found in the appendix, although it is believed that this method provided the best solution as it was a console based application which was easily modifiable in Visual Studio. This program was changed to send only the information necessary for moving the end-effector to the V-REP client program, which communicated through stdin and stdout. It was found that the application wouldn't exit if there wasn't a joystick available. Thus, the application was made to send whether or not there is a joystick to connect to on start up, and exiting if there isn't, allowing the client program to take inputs from keyboard instead of joystick. The joystick application originally used 17% CPU

usage when reading form the joystick. Upon closer inspection, it was found that the program was continuously searching for events on the joystick; it wasn't going to sleep to allow other programs to run more often. MSDN has a function called 'sleep' which will set the thread to wait state, which was used for 50ms within this application. It was found that this figure reduced the CPU usage to below 1% whilst not taking away from the responsiveness of the application to joystick events.

Upon further inspection of the CPU usage whilst using the V-REP client, it was found that the client program consumed 25% of the CPU on average. Considering it consisted of one thread for calculations and communication with V-REP, one for reading and interpreting from the joystick process, and one for adding to the buffer when the joystick hasn't been moved and the last log was non-zero, this figure was understandable, however unnecessary. The latter thread hangs in a loop until 750 ms (varies computer to computer depending on CPU speed, but this number was found to be good for the computer this thesis was constructed on) has passed, where it checks the buffer and adds a new command if needed. Thus, it was desirable to make this thread terminate if there isn't a joystick to read from, and when there is, sleep for 50ms to allow other threads to run (meaning a new command is added every 750 - 800 ms, which is an unnoticeable change for the user. A similar outcome was made for the thread that communicates with the joystick process. The main thread was also put into wait state, for 20ms, when it's held waiting for an input command. Overall, it was found that these implementations decreased the CPU usage to around 1%. It was also evident that the inverse kinematics calculations were performed at around 500 ms (depends on computer CPU), as opposed to the prior average of 750 ms; these results can be seen in the inverse kinematics spread-sheets. Thus, adding waiting delays within the threads allowed the important computations happen quicker, which led to a faster client program.

6.3 Client Program Inverse Kinematics

With setting up the client to be able to interpret inverse kinematics commands, it was found that errors commonly occurred when the end-effector moved between quadrants. For instance, with the implementation of the proposed analytic inverse kinematics solution (judging the change in end-effector position by moving position of the fourth joint), scenarios occurred where the fourth joint was in one quadrant, and the end-effector in another. This would produce an error when the client program attempts to align the position of the fourth joint with the end-effector, and when it bases calculations for the new end-effector position off a joint which is in a different quadrant. To overcome this problem, when the angle of end-effector to the x-axis is found to be offset from that of the fourth joint (pointing straight upwards,

in a different quadrant to the fourth joint, or just offset from previous movements) the client program will move the fourth joint to align them in the x-y plane. This is done by reducing the error in angle between the fourth joint and the tip, before starting the inverse kinematics. A problem with this method is that errors can still occur when the gripper sits on the negative x-axis. This is because the angle of the fourth joint could be close to 2π, and the tip be close to 0. Thus, there is a large error that causes undesirable movements where the client program recognises a difference in quadrants, and tries to align something that is visually aligned, by flipping the wrist around. As with the physical Jaco arm, it is recommended that this virtual arm only moves within one quadrant (however it can operate fine without coming near the negative x-axis).

In order to demonstrate the implementation of the three inverse-kinematics methods the client program can provide, commands for swapping between the methods was developed. The input syntax for this is, 'mode (mode number) (boolean value)'; for instance, turning mode 1 off would require the command 'mode 1 0'. Typing only the word 'mode' will display the usage for this command, and show the active modes. The mode numbers are as presented in this paper, the first mode is the analytical solution, the second is a control systems method, and the third is for using the Jacobian inverse.

Extracting data from the control loops to a spread-sheet allowed the tuning of the PI controllers. It could be seen that adjusting certain gains would produce effects in steady-state error, oscillation and setting time. In addition to this, it could be seen which commands the solutions struggled with most. For instance, it was found that the control system implementation was accurate in changing the x-y position of the end-effector, however had high oscillation and steady-state error in the z. This was explained to be due to a lack of constraint on the radial position of the end-effector, and due to a high integral gain on q_1 which controls the x-y angular position; thus eliminating stead-state error faster than the error in the z-axis is corrected. To overcome these issues, more than two forms of error should be reduced, as the end-effector moves in 3 dimensional space, and a cascade loop should be used so the responses do not fight.

6.4 Client Program Jacobian

As found in the Kinematics Approach section, the Jacobian inverse kinematics method was unsuccessfully implemented. In the client program, the determinant of JJ^T would equal 0 when using double-precision floating point numbers, and would equal an incorrect value when using standard floats. For double-precision, the spacing, or error in rounding to a representable number, can be found to be

2^{n-52}, for a number between 2^n and 2^{n+1}, giving a maximum possible error of 2^{-53}, which rounds to $1.11022e-16$. Thus, the differences in answers is a consequence of rounding between the finite values available with floating point numbers, and thus is also dependent on the order of arithmetic. An example of this error is shown in the calculations below;

$e = 1 - 3 * (4/3 - 1)$

results in

$e = 2.2204e - 16$

which is evidently twice 1.11022e-16, and is not equal to 0 as it is supposed to. For the computation of the determinant, MATLAB uses the product of diagonals of the matrices from the LU decomposition; where the LU decomposition matrices are calculated by Gaussian elimination. In the documentation for MATLAB's lu function (calculates the lower and upper triangle matrices such that A = L × U), it is noted that factorisation is a key step in obtaining the determinant, for reasons described above. Both, an LU decomposition method and a standard algebraic way of solving smaller determinants within the matrix, were implemented into the client program to solve for the determinant, with results either approaching -inf or 0. Even when comparing the LU matrices from the client, they are different to that from MATLAB, leading to the conclusion that MATLAB's implementation is structured to minimise the error in calculations, which is lacking in the client program.

6.5 Kinova SDK

Development with the Kinova SDK was not trialled as the initialisation functions require connection to the physical Jaco arm, and thus no testing could be conducted. There are plenty of examples within the SDK that show how to code for moving to a new end-effector position, or changing the target positions for the joint angles. From the examples, it is evident that the program designed to use the SDK and control the physical arm would be hardly dissimilar to the developed client program. It would be required to take in a joint number and a desired angle, where internally it sends the commands to the Jaco arm to work with. As the client program now uses significantly less CPU, and can control the virtual arm in real-time, it is evident that processing to the development of the physical arm's interface can begin.

6.6 Data Streaming Implementation

To be able to detect collisions, the client program comes with a synchronous mode, where it continuously retrieves the Jaco arm's angles from V-REP. In order to do this, 6 threads were created, one for each joint; as only one value can be streamed

constantly. Thus, 6 connections are made to V-REP over 6 ports, with the data being streamed every 50 ms (variable). Once a new joint angle arrives to one of the clients, it updates the struct containing the arm parameters, which is accessible by all threads; mutexes make this thread safe, the main thread will only look at the angles when making movement calculations. This mode was deactivated for general use of the client program, as it was found to significantly reduced simulation time (due to V-REP's increased background processing). Another finding was that the simulation time depended on the amount of objects in the scene, as well as the CPU's capabilities; moving to a different computer, it was found that moving the Jaco arm was 5 times faster. Thus, there is potential that the streaming method can work with the client program operating as normal, given an empty scene and a good CPU. To reduce the load on the CPU, the threads sleep for 10 ms whenever data has not arrived from V-REP. This was found to again to decrease processing time, however not enough to implement velocity / acceleration checking; but may be a good option on a better computer.

6.7 Communication

As aforementioned, the speed of calculations were found to be CPU dependent; where the client's method of calculating the inverse kinematics was found to take 100 ms on average on one computer, and 500 ms on another. Communication speed between V-REP and the client program is dependent on the networks both run on. When connecting to local host, the network adapters are completely missed, so communication happens within 1 ms. When run on different computers connected via a LAN connection, sending an inverse kinematics command to V-REP consisted of three data transfers from the client to V-REP, where the packets contained 100 data bytes, taking 80 ms to complete. When the client is in IK mode and V-REP is in FK mode, the client sends one packet containing all of the desired joint angles; with a data length of 282 bytes. The transfer and acknowledgement in this case took 20 ms. Thus, the time between the client program taking an input from a user and V-REP receiving the joint commands is approximately 120 ms, given the computers are directly connected. It is unknown as to how long V-REP will take to interpret this command, and act, and thus this information doesn't provide evidence that V-REP calculating the inverse kinematics, or the client program, is faster. It is expected however that sending all of the data for the joint angles in one packet will reduce processing time on the V-REP side, as it only needs to interpret one packet instead of multiple. A useful finding was that the the remote API cannot connect to computers on a private network; having the client at the University of Queensland, and V-REP on a home network, it was found that the two were unable to connect.

There was however no problem when the two were on the same network.

Chapter 7

Conclusions

7.1 Summary and Conclusions

The primary objective of this thesis was to create a client program which uses the V-REP remote API to control a virtual Kinova Jaco arm. Upon forming a method for reliable control, different techniques used for calculating the joint angles required to move the end-effector of a 6 joint robotic arm were studied. Experimentally, the validity of three approaches were tested, to determine which was the most robust in accurately controlling the robotic arm.

The first approach used relationships between joints to calculate the angles required to move the arm into an approximate position, and then used control loops to reduce the error in end-effector position. This was found to be successful in most arm configurations, where the joint angles were correctly calculated in 500 ms on an old computer, and an average of 100 ms on a modern laptop. Unfortunately this method led to errors when the arm tried to reach points too close to the z-axis, as some components of the arm were in different quadrants to others.

A control systems approach was wrongly implemented into the client program, which allowed the calculation of joint angles which produced an undesirable amount of steady-state error in the height of the end-effector from the base. After reviewing the model used in for this approach, it was found that there is a lack of control on the radial position of the arm, which contributed to the steady-state error, along with having a high integral gain on the control of the x-y axis. Alternative models were proposed which would overcome these problems.

The Jacobian was also integrated into the client program after providing good results in MATLAB. Due to calculation differences, and numerical precision, the

inverse Jacobian for a given set of joint angles was unable to be calculated with accuracy in the client program. The discrepancy between MATLAB and the client program came from the more sophisticated method of calculating the determinant and inverse of a square matrix in MATLAB; which takes into account the order of operations such that the error due to the rounding of (double) floating point numbers is reduced.

Despite these draw-backs, the client program successfully acheived crucial design objectives.

- An easy to use program was created which allowes the control of the Jaco arm given forward or inverse kinematics commands.
- The end-effector position can be controlled with the use of an external joystick (similar to a PS3 controller).
- Mathematical solutions to the forward and inverse kinematics of the Jaco arm are integrated into the V-REP client and produce accurate results.
- Collision avoidance can be implemented using the data-streaming capabilities of the client program given a fast enough CPU.
- Through V-REP's inverse kinematics solver, set to Damped Least Squares, or the analytical solution provided by the client program, the position of the end-effector of the Jaco arm can be controlled in real time.
- The client program created has functions within for further development, such as orientation control of the end-effector, detecting collisions, and sending joint information to other programs.
- The client program was optimized for CPU usage and computational efficiency. It can also be quickly modified to allow the printing of usage information associated with different areas of the program, and to allow data-streaming.

7.2 Developed Files

In total, the client program contains three .c files and .h files. The main file was designed to do calculations and communicate with V-REP; the flow diagram for which is provided with this document. All kinematics calculations were kept in a separate file to communication, and is used for control loop, forward / inverse kinematics, and computing the inverse Jacobian. The final file, with two threads, is dedicated to controlling the commands coming from the joystick; one for getting and

interpreting commands and another for servicing the command buffer, which added commands to the buffer when the joystick is held in a non-zero position. When the threads are waiting for an input, from user, joystick or buffer, the threads are set to a waiting state to limit the CPU usage of the program, which was found to decrease computation time, creating a more responsive system between joystick command and V-REP Jaco arm movements.

All kinematics were trialled in MATLAB before implementing into the client program. This relieved the need for the client program to manipulate complex matrix operations for the forward kinematics. Instead, the calculations were made symbolically in MATLAB and copied over for integration; for instance, in calculating the Pseudo Jacobian, 10 complex expressions describing the elements of JJ^T were able to be copied from MATLAB instead of having the client program compute this relationship each time. The MATLAB files are provided with this document, which include forward kinematics, inverse kinematics, Jacobian inverse and Pseudo inverse calculations, and a Simulink model simulating the control system implemented in the client program.

7.3 Possible Future Work

As noted, unsolved problems occurred in the closing of this thesis. It is believed that with time all of which can be fixed, giving a more complete client program for controlling the virtual robotic arm. As it stands, the client program is able to control the virtual arm in both forward kinematics and inverse kinematics modes, using V-REP to calculate joint angles or using the proposed analytic solution. However, it is evident from the MATLAB script on the inverse Jacobian, that this method can produce accurate results in approximating the change in joint angles required for a small change in end-effector position. This method can be further implemented in two ways;

- Using the symbolic JJ^T matrix in the client program for the Jaco arm, construct a method for computing the determinant and inverse in a similar means to MATLAB, to ensure transparent results across platforms; MATLAB uses LU decomposition solved by Gaussian elimination, whereas the client program uses a simplified version of this which is susceptible to rounding errors.

- V-REP has an internal function, simComputeJacobian, which provides the last linearised Jacobian matrix of an inverse kinematics group. This functionality, and more, are available in the External Kinematics Library, which is an auxiliary API not directly a part of V-REP. Example code for using this library is provided with V-REP on download, and is written in C++. With an

original lack of knowledge of how these functions work, and what the matrices represent, this avenue was not greatly explored. With more knowledge now, it can be seen to be a useful library to incorporate into the client program.

Additionally, it was found that the control system implementation was able to converse two of the three axis of the tip to a desired end-effector position. The steady-state error in the z-axis can be eliminated by making the following design considerations;

- Having control over the error in three axis, polar or Cartesian, instead of two; which were the polar angles in the x-y plane and xy-z plane. Reducing the control to two, being the x-y and z angles, reduces the control of the end-effector in 3-Dimensional space.

- It was known that q_1 would have a large influence on correcting the error in the x-y plane, and the other joints would have a coupling of all three axis. A high integral gain was set to control q_1, with the expectation that the other joints would position the end-effector close to the desired position, with any error in the x-y plane being slowly corrected by the control of q_1. Instead, the high integral gain created a faster response than the other joints. Reflecting on the model used, this is understandable as integral gain intrinsically increases rise-time and overshoot. Outlined in the Kinematics Approach section is the two models which were supposed to be used to meet the design goals. These are to implement integral action, or similarly a cascade loop, which brings q_1 to a reference position with a slow enough response that it does not effect the regulation of the other joint angles.

Data-streaming was proven to be successful, however V-REP became slow with communicating with each joint angle to the client continuously. For future improvements of this, it is recommended to try the data-streaming mode of the client program on a computer with a better CPU. Additionally, an internal LUA script can be created to detect collisions using V-REP's collision detection feature, and communicate with a thread in the client program when there is a collision. Thus, this would change the number of threads required for collision detection to 1 instead of 6 plus acceleration calculations; which may result in false positives. It was found that communicating with V-REP with over 7 threads can cause the program to crash, and thus the number of threads connecting to V-REP should be limited.

With these developments, and the current state of the client program, it is believed that the application for controlling the physical Jaco arm can be created.

Appendix A

Tables and Figures

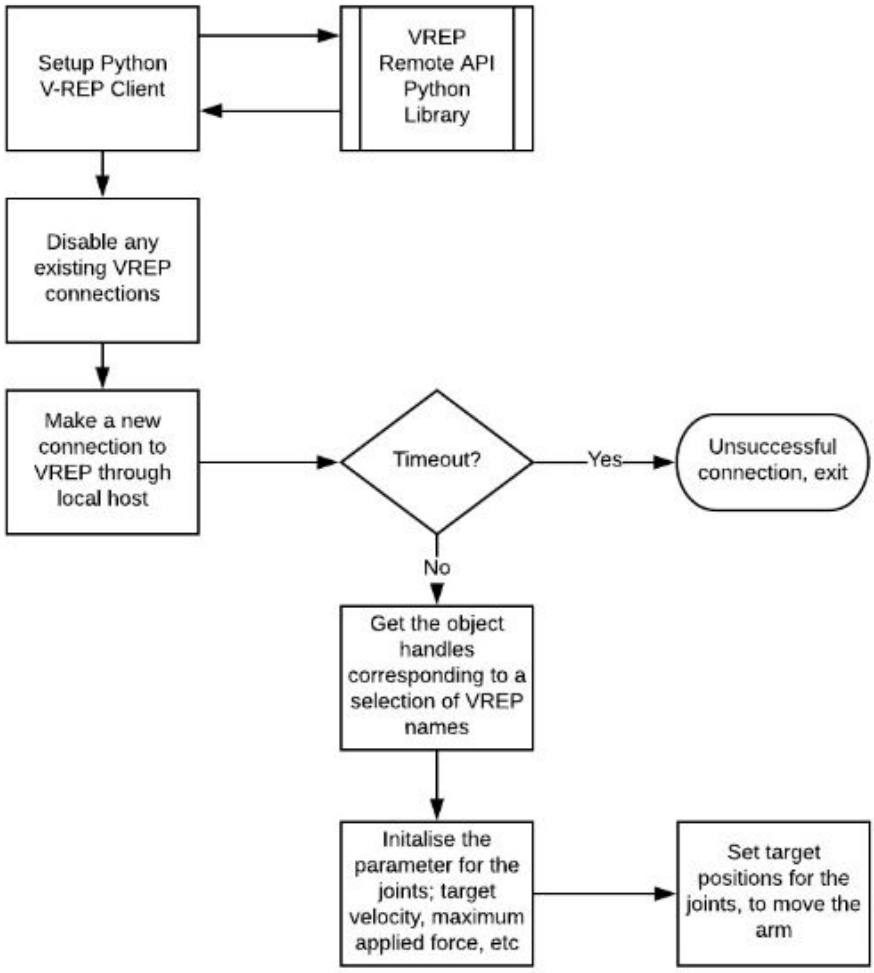

Figure A.1: Simple V-REP client program in Python Flow Chart

```
# Specify compiler
CC=cl.exe
# Specify Linker
LINK=link.exe
# Flags

all:
    gcc -Wall -DNON_MATLAB_PARSING -DMAX_EXT_API_CONNECTIONS=255 -fPIC -std=gnu99 -c vrep_main.c -o vrep_main.o
    LINK $(CFLAGS) -c extApi.c -o extApi.o
    LINK $(CFLAGS) -c extApiPlatform.c -o extApiPlatform.o
    CC extApi.o extApiPlatform.o vrep_main.o -o vrepClient -lpthread
```

Figure A.2: V-REP Client Makefile

```
-- Lua function for returning object names, given
-- given an object handle.
function get_object_names(objectHandle)
    objectName=simGetObjectName(objectHandle[1]*1)
    simAddStatusbarMessage(objectName)
    returnList={objectName}
    --returnList[1] = objectName

    return {},{},{objectName}, ''
end
```

Figure A.3: Lua function in V-REP main script, for retrieving object names

```
18 Jaco_joint1
21 Jaco_joint2
24 Jaco_joint3
27 Jaco_joint4
30 Jaco_joint5
33 Jaco_joint6
38 JacoHand_joint1_finger1
41 JacoHand_joint2_finger1
46 JacoHand_Spherical_joint7
48 JacoHand_Prismatic_joint2
50 JacoHand_Spherical_joint8
54 JacoHand_Spherical_joint1
56 JacoHand_Prismatic_joint1
58 JacoHand_Spherical_joint4
60 JacoHand_joint1_finger2
65 JacoHand_joint2_finger2
70 JacoHand_Spherical_joint5
72 JacoHand_Prismatic_joint4
74 JacoHand_Spherical_joint10
76 JacoHand_Spherical_joint
78 JacoHand_Prismatic_joint8
80 JacoHand_Spherical_joint2
82 JacoHand_joint1_finger3
87 JacoHand_joint2_finger3
92 JacoHand_Spherical_joint6
94 JacoHand_Prismatic_joint3
96 JacoHand_Spherical_joint9
98 JacoHand_Spherical_joint0
100 JacoHand_Prismatic_joint5
102 JacoHand_Spherical_joint3
104 JacoHand_fingers12_motor1
108 JacoHand_fingers12_motor2
112 JacoHand_finger3_motor1
115 JacoHand_finger3_motor2
122 Target
```

Figure A.4: Example of a text file created by the V-REP client

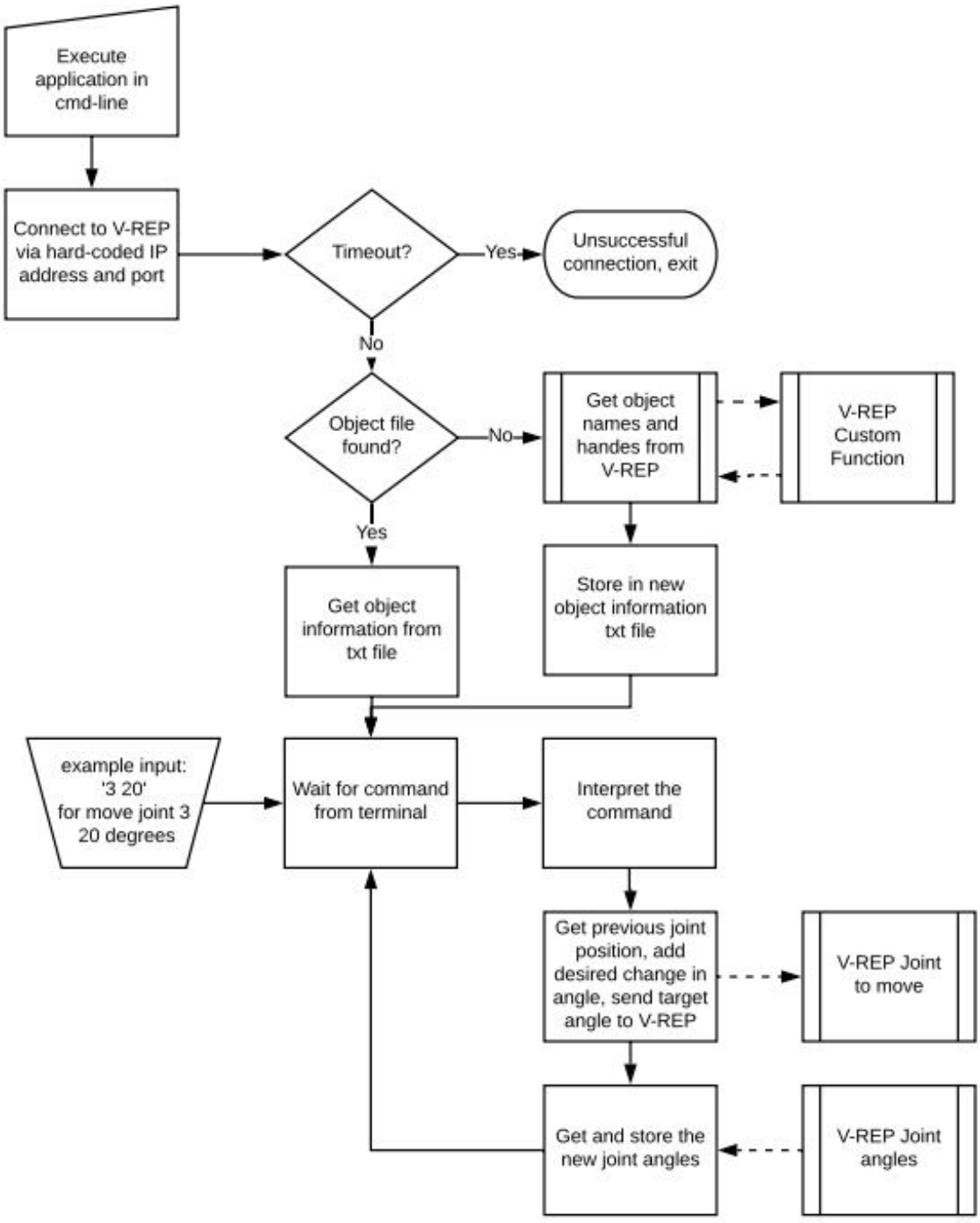

Figure A.5: Client program in Forward Kinematics mode

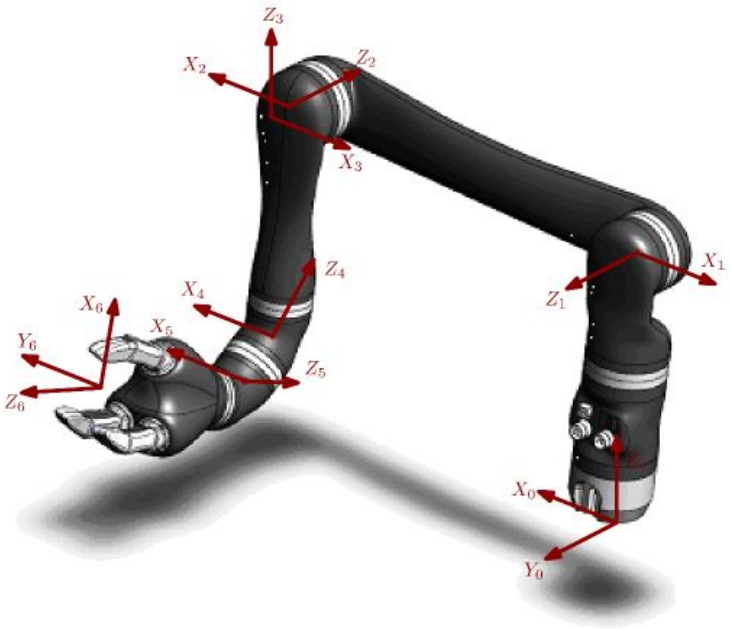

Figure A.6: D-H Parameters axis for each joint on Jaco arm

Keyboard Command	Change in Tip Position	Description
'w'	3 mm	radially out (horizontal) from the base of the arm
's'	3 mm	radially in (horizontal) towards the bass of the arm
'a'	3 mm	anti-clockwise (horizontal) from the current tip position
'd'	3 mm	clockwise (horizontal) from the current tip position
'-'	3 mm	change in height of the tip downwards
'+'	3 mm	change in height of the tip upwards

Table A.1: Client program IK input commands

Thread	Description
Main	Handled input commands from the joystick buffer or from the terminal, and communicated with V-REP to produce the desired outcome
Joystick Interpretation	Reads from stdout of the joystick process, interprets the information, and adds a (w/s/d/a/-/+) command to the joystick command buffer with decimal value corresponding to the amount that the joystick was pushed for that command (0 for no press, 1 for 100% pressed).
Joystick Add Command	This task keeps track of the last command to be added to the buffer, when the push value is greater than 0, and 750ms has passed, a copy of the last command is added to the buffer as the joystick is still pressed down.

Table A.2: Client program IK input commands

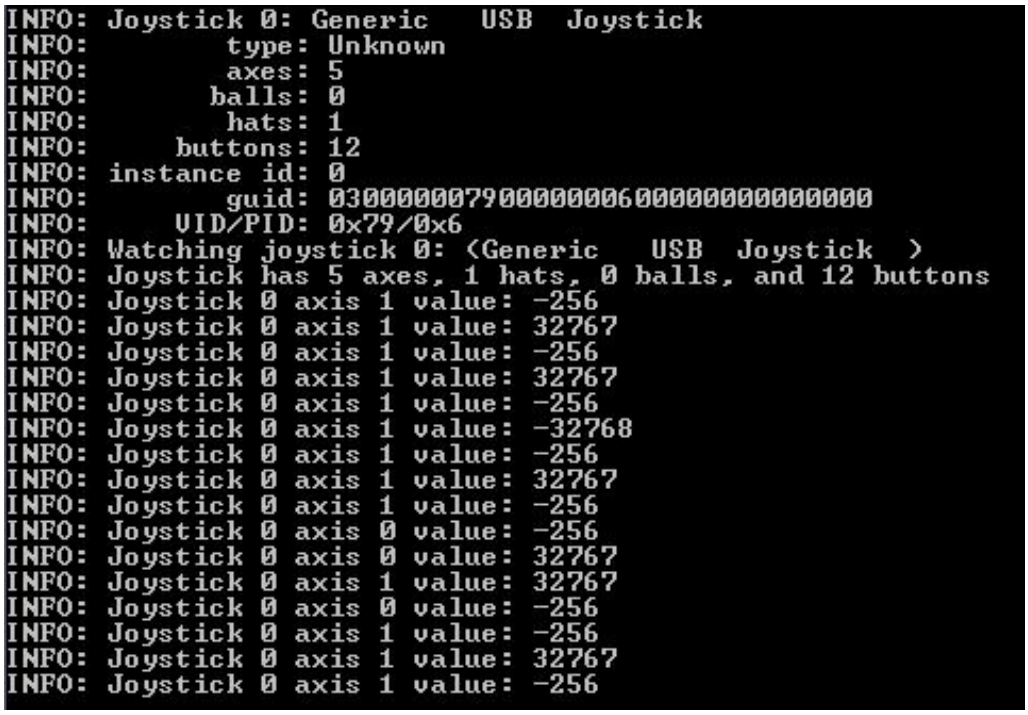

Figure A.7: An example of the terminal output of the joystick process when the left joystick is toggled on

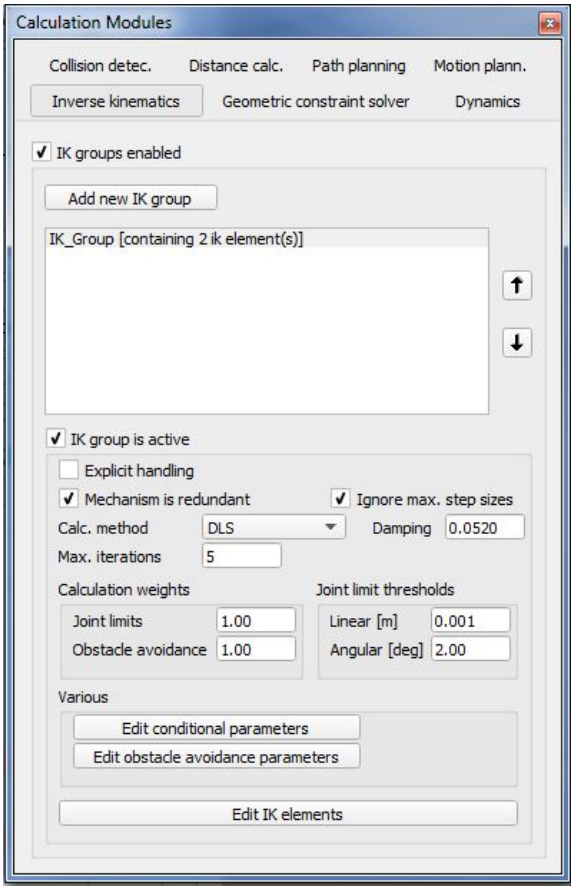

Figure A.8: The calculation method for Inverse Kinematics in V-REP

Joint	K_p	K_I	% XY	% Z	Arm Length	Angle Correction
1	0.05	0.99	100	0	197.13	$-q_1 - \pi$
2	0.65	0.55	25	75	410	$q_2 - \frac{\pi}{2}$
3	1.17	1.11	25	75	207.3	$q_3 + \frac{3\pi}{2}$
4	0.1	0.02	50	50	74.1	q_4
5	0.15	0.15	50	50	74.1	$q_5 - \pi$
6	0.15	0.15	0	0	160	q_6

Table A.3: Chosen controller coefficients for each joint

Figure A.9: V-REP Client Program Control Loop Error

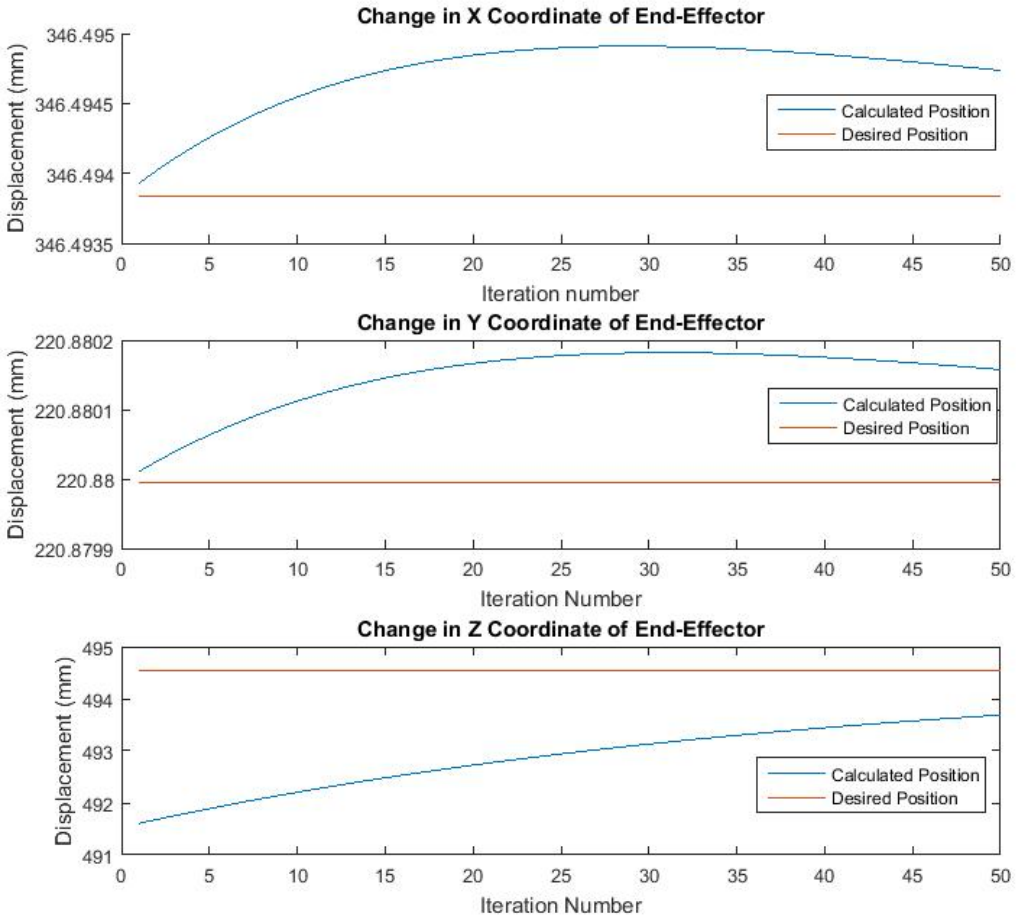

Figure A.10: Plot of using Jacobian Approximation to move the end-effector to a desired position

Appendix B

Additional Findings

B.1 Joystick Applications

B.1.1 Objectives

Retrieve real time data of the joystick states; button presses and joystick position from a Generic Joystick Controller with USB input. Have the states printed to stdout in real time using an executable that can be spawned from a parent process (VREP main client). The process must be killable from the parent process as well, to avoid unnecessary hanging and infinite forking, which could damage computer memory.

B.1.2 Windows API

Windows has its own API for building window executables. This API is formatted differently to standard C programs, and contains very different properties and types. For example, main is run from [1]: int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd) The syntax and libraries from Windows.h, with hidsdi.h allowes direct communication with Human Interface Devices (HID). Windows are generated first, and messages are sent through this thread and the HID, thus to GetRawInputData(), a window must be created to generate the message struct [1] and the lParam argument [2]. Message windows can be created by setting the hWndParent parameter of CreateWindow() to HWND_MESSAGE; this will create a hidden window that can only be terminated using Task Manager. The primary problem with the Windows API is that it is separate to the console, thus writing to stdout is ineffective, and the only way to create an output is to write to a file with FILE* f = fopen(), and fprintf(f, text). Setting the linking configuration from Windows to Console will cause the program build to be unsuccessful. Despite these problems, Windows API provides the lowest

level of talking directly to the HID drivers, and manipulation of USB data from the controller.

Open-source code for generating a window containing a joystick GUI with real-time button presses is available from [4]. Building RawInputJoystickSource/Raw Input.vcxproj in Visual Studio will generate an executable Raw Input.exe in Raw-InputJoystickSource/Debug, which can be run from command prompt. Adding print statements or writing to stdout won't produce anything on command prompt, however the executable will create a window containing the joystick visualisation. Re-writing the source code from the Windows API was unsuccessful as getting the raw input requires the message struct, generated using a window thread. Created text files however will appear in /Debug.

B.1.3 HTML5 Gamepad API

Code for creating joystick support with browser (HTML5) games is available from [5] with information at [6]. The latter contains instructions for getting a PS4 controller working in Chrome and Firefox. As this isnt useful for application based programs, it would be a useful alternative for games that are ran using .html.

B.1.4 Libusb and xusb.c

A library used for retrieving information from USB, with an example executable and solution, xusb.c, which demonstrates this for Generic, PS3 and Xbox joysticks [7]. The library functions have documentation available from [7], however, apart from the xusb example there is limited documentation on how to implement these functions or format code. Thus, manipulating the code for the Generic USB Joystick to work with xusb.c, to see the button inputs is difficult. If the file is ran with xusb.exe s vid:pid the device information and reading input report can be generated, if buttons are being pressed the hexadecimal output will reflect this. It is possible to loop the Reading Input Report and print to stdout in the same way as current. The inputs will then have to be mapped to buttons, which can create difficulty with interpreting the state of the joysticks. A PS3 controller code did not work (exited with Input/Output error), as the inputs werent printed to terminal, but it is expected that the code format for processing the input, to show which buttons were pressed, will be similar to that for the Generic USB controller. The code can be executed from libusb-1.0.21/examples/bin64, with code kept in libusb-1.0.21/examples/xusb.c.

B.1.5 Simple Direct Layer (SDL) Source Code

Written in C, a cross platform library can be download for the use of getting data from keyboard, mouse and joysticks, from the SDL website [9]. If the source code is modified it must be marked as so, otherwise it is free to be used and manipulated to repeatedly display input data from devices; originally set to only print device information, however input manipulation functions are provided. The joystick project can be built by loading SDL2-2.0.7/VisualC/SDL into Visual Studio and building from testjoystick.c. This file initialises the joystick and retrieves data from the SDL_joystick.c file, where it can be executed in a console from SDL2-2.0.7/VisualC/Win32/Debug. The test files are setup to produce a GUI, however the code for this can be commented out, and input manipulation moved to main. 'Printf' can be included by creating a new stream pointing to standard out, and joystick information sent to it (by default, SDL uses a different handler (not stdout) to point information to the command line, and so an external process cannot read this information).

B.2 Alternate Inverse Kinematics Solutions

Manipulators with 6 rotational joints can reach any position within its stretched length, where arms with more joints are termed as over-actuated robotic actuators and are capable of changing their joint configurations without changing the end-effector position. It is due to the aforementioned reason that 6 joint robotic arms are so widely used, and the inverse kinematics is of such high interest.
Raghavan and Roth's solution to the inverse kinematics of any 6 joint robotic arm is identified in many research papers and textbooks. It takes the transformation matrices between joints of the arm from the forward kinematics, and rearranges them to make a system of equations about joint three. From here, there are 6 unknowns and 5 equations, where by substituting out the trigonometric products form a linear system with 16 unknowns and 6 equations. With more substitutions they arrived at a 12 by 12 matrix written in terms of variables representing trigonometric expressions. The determinant of this matrix gives a 16-degree polynomial referenced to the middle joint, of which can be solved for using a root solver, and the other joint angles found by back substitution. From this, it was concluded that there is 16 possible solutions to the inverse kinematics, however solving a 16-degree polynomial made this method too slow for real-time implementation. The Manocha Method avoids solving the determinant by forming the equations into an eigenvalue problem, where in solving for the eigenvectors gave three of the arm's joint angles with less operations. This was the first real-time inverse kinematics method to be

implemented, with methods since being variations of the two outlined above. The Raghavan and Roth method is briefly outlined in appendix C of Lung-Wen Tsai's textbook Robot Analysis: The Mechanics of Serial and Parallel Manipulators. Further, it is worked through along with the Manocha Method in an article from the 14th Triennial World Congress, 1999, A Faster Algorithm for Calculating the Inverse Kinematics of General 6R Manipulator for Robot Real Time Control.
It is believed that although new methods derived from those described above form solutions to the inverse kinematics problem in this thesis, as the Jaco arm has 6 revolute joints, the mathematics knowledge and tools for computing the eigenvalues of the complex equations is insufficient.

Appendix C

Program Listings

C.1 V-REP Client Program

The V-REP client program was developed in C for Windows console, written using Visual Studio 2017 on Windows 7. This program links with the V-REP Remote API, with the relevant includes already with the source files; so no extra libraries should be required other than those provided. Three source files are used in the client program, with the main one titled 'vrep_main.c', which contains initialisations, the communication and input handling for the program. All kinematics handling is stored in 'kinematics.c', and communicating with the joystick process is contained within 'joystick.c'.

Visual studio can build the project, creating an executable in programming/client/x64/Release/vrepClientProgram.exe

The usage for which is presented in this paper.

Within the programming directory the remote API folders and files can be found, further, in programming/client the client program's source files are present.

C.2 SDL

A copy of SDL-2.0.7 is present in programming/SDL2-2.0.7, where the executable for the joystick is contained within programming/SDL2-2.0.7/VisualC/Win32/Debug/testjoystick.exe. The contents of the SDL2-2.0.7 folder is excessive, this is because it contains the full library so the program can work. The source code for this application can be accessed through /programming/SDL2-2.0.7/VisualC/tests/ testjoystick/testjoystick.vcxproj.

It will be mentioned again, the SDL is not work of my own, however testjoystick.exe has been modified for the use of this thesis.

When the V-REP client program starts up, it will use the current working di-

rectory to navigate to the folder containing joystick.exe, so these folders should not be changed.

C.3 MATLAB

The forward kinematic calculations are included in forwardKinematics.m. This MATLAB script uses the Classic D-H parameters of the Jaco arm to calculate the transformation matrix. From there, the translational components are extracted and the offset in angles between the D-H configuration and the upright position are substituted in. Thus, after running this file, the transformation between each link noted by A_i, the full transformation matrix A, and the coordinates for the Jaco arm in the upright position A_up will be included to the workspace.

An analytical and matrix manipulation solutions to the inverse kinematics problem is presented in inverseKinematics.m. Where the transformation matrix is computed as earlier, and then rearranged to determine relationship between matrix elements; as described in the theory section of this paper. Following this, the analytical solution is provided for a known end-effector position and corresponding joint angles. As previously noted, this approach was unsuccessful after joint 4 of the Jaco arm.

The calculations for the inverse Jacobian using the Newton-Raphson method and the Pseudo Inverse Jacobian is presented in jacobianInverseKinematics.m. Firstly the transformation matrix is calculated, and then the translational components were derived with respect to each angle and the Skew matrix was determined, with respect to each angle. Following, the Jacobian was constructed and substituted in the start-up position of the arm when it's in inverse kinematics mode (no singularities). From this, it was found that the inverse Jacobian accurately repositions the arm within 50 iterations, to a change in z coordinate of 3mm for the end effector. The matrices used for this were complex, and so three of the states were made redundant and the Pseudo inverse Jacobian was calculated, which provided, again, accurate results. Running this should be done in sections, as broken up already. V will contain the final end-effector position of the two Jacobian sections, were the vector x denotes the starting position given by the angles q.

A Simulink model, Control_Model_v3_1.slx is also included. This model presents the control systems approach to adjusting the joints to reach an end-effector position.

C.4 Additional Files on Disk

The disk (.zip file) will contain the excel spread-sheets used when tuning the control loops in the V-REP client program; appropriately named. In addition to this, it

will include V-REP scenes set-up for running the client program. These will be included in the scenes directory, where it is recommended that emptySceneFK.ttt and emptySceneIK.ttt should be used when running the program; as they are empty scenes and the client has already made text files containing the object names and object handles, located in the Release folder of the program.

If the program were to be run, for easy start up the usage for emptySceneFK.ttt is:

vrepClientProgram.exe fk emptySceneFK.txt

Similarly if the IK scene was to be used, then the fk/FK are replaced with ik/IK.

Bibliography

[1] L. Lamport, *LaTeX: A Document Preparation System*, 2nd ed. (Addison-Wesley, 1994).

[2] REFERENCE 2

[3] Etc.