LOW-COST 4-DOF ROBOTIC ARM USING ROS: COLLISION-FREE PATH PLANNING

Jacob Blanck

Bachelor of Engineering
Mechatronic Engineering Major

MACQUARIE
University
SYDNEY·AUSTRALIA

Department of Engineering
Macquarie University

November 6, 2017

Supervisor: Subhas Mukhopadhyay

ACKNOWLEDGMENTS

I would like to acknowledge several people for supporting me and helping me through the duration of my undergraduate degree and thesis. Firstly, I would like to dedicate this project to God. He is always there, giving me wisdom and opportunity through my circumstances and my family. He has given me everything in life and I pray that I can continue to live my life in His image and to achieve what He has set out for me. Thank you to my beautiful wife, Rachel, for all the love and support, guiding and encouraging me through all the stressful times and long hours. Without her, I would not be the man I am today, she always inspires me to work harder and believe that what I am working towards is achievable. Subhas Mukhopadhyay, thank you for your guidance, support and assistance through my thesis project. I am very grateful for the attention and time you have given throughout this project.

STATEMENT OF CANDIDATE

I, Jacob Blanck, declare that this report, submitted as part of the requirement for the award of Bachelor of Engineering in the Department of Engineering, Macquarie University, is entirely my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualification or assessment at any other academic institution.

Student's Name: Jacob Blanck

Student's Signature: [illegible]

Date: 06/11/2017

ABSTRACT

Robotic arm control and path planning is a highly important aspect of research due to the rise in robotic arm applications and advancements in their capacity to work accurately and safely. ROS software has been used to achieve this collision-free path planning and robotic arm control previously in research, however these robotic arms are often expensive and supply ready-made SDK packages that function within ROS. In this project, ROS software is used to simulate the "AX-12A Smart Robotic Arm" from CrustCrawler. This low-cost robotic arm has limited open-source ROS software associated with it, in addition to no accessible simulations of the robotic arm. The aim of this project is to form a collision-free static path planning model within ROS, similar to packages available for more expensive robotic arms. The ROS software generates a random goal state and plans a collision-free path for the robotic arm to follow, achieving the desired state. This is achieved through a MoveIt! configuration generated from SRDF and URDF files created for the robotic arm. The results show that the URDF functions correctly and the MoveIt! configuration path plans successfully. The simulation is able to identify configurations that include self-collision and will not path plan to this collision. This research is significant for the use of low-cost robotic arms within both research and industrial industries whilst accurately executing path planning to a goal state and operating without collision.

Contents

List of Figures

List of Tables

Abbreviations

3D	Three-Dimensional
DOF	Degrees of Freedom
ID	Identifier
IGES	Initial Graphics Exchange Specification
IK	Inverse Kinematics
JTAC	Joint Trajectory Action Controller
OMPL	Open Motion Planning Library
PR	Personal Robot
RAM	Random-Access Memory
ROS	Robot Operating System
SDK	Software Development Kit
SRDF	Semantic Robot Description Format
tf	Transform
URDF	Unified Robot Description Format
USB	Universal Serial Bus
XML	Extensible Markup Language

Chapter 1

Introduction

Path planning is the process of establishing a collision-free movement from the current configuration of the robotic arm, to the final configuration of the robotic arm [4]. The final configuration is generally determined by a goal end-effector position and the remaining links positions determined through inverse kinematics or similar. There are two basic types of robotic arm path planning; static and dynamic. Static path planning is based on the full knowledge of the robots workspace, in which the ideal path is planned to achieve the goal position without collisions [5, 6]. Dynamic path planning however is based off constantly evaluating the workspace and predicting hazards, due its environment constantly changing [4]. These changes can be work related, such as sorting objects that are brought into the workspace at random, or unpredictable obstacles such as humans. As a result, the end-effector position and path planning process are essential in any robotic arm within its workspace to ensure collision-free movement. The methodology of controlling these robotic arms has changed over time, with sensors, human controls and "repetitive configuration moves" all featuring over the years. In recent times, controlling robotic arms through software such as ROS has been of academic interest for non-dynamic and dynamic environments [7, 8]. This is explored further in Chapter 2 with similar ROS controlled projects described in Section 3.4.

1.1 Project Objectives

In this project, ROS will be used to implement a software simulation model of the AX-12A Smart Robotic Arm from CrustCrawler. The simulation will be able to achieve collision-free path planning from a start configuration to a goal configuration. The workspace will be static such that every obstacle in the workspace will be known. The simulation presented will function identically to many other robotic arms supported by ROS, but for a much cheaper robotic arm in comparison. At present, there is limited research into such a low-cost robotic arm completing this task. The project research can then be built onto in future to implement software-to-hardware integration, end-effector control, image processing to identify objects and demand the end-effector to reach the objects and sort them, as well as increasing the complexity of the workspace into a dynamic workspace.

1.2 Project Goals

- Implement a virtual machine which runs the Ubuntu 16.04 operating system complete with ROS Kinetic and its relevant tools and packages outlined in Section 3.2.
- Transform a 3D drawing of the AX-12A Smart Robotic Arm from CrustCrawler into a URDF file which includes details on all joints locations, axis and limits as well as links mass, inertial values and origin.
- Create a MoveIt! configuration for the robotic arm based of the SRDF and URDF files which demonstrates the realism of the software demonstration through realistic joint limits and similarity to the physical robotic arm.
- Demonstrate the MoveIt! configuration's ability to plan a path from its present state to a goal state without failure or self-collision.

1.3 Project Validation

When examining the research into ROS control and collision-free path planning, the most similar robotic arm with packages available in the ROS library is for the "Robotis Manipulator-H". This robotic arm has 6 DOF using the H-series Dynamixel Pro servo motors and use RS-485 communication and USB interface. The cost however of these robotic arms is $18,900 USD [3], compared to the cost of the AX-12A Smart Robotic Arm from CrustCrawler, which is $700 USD [1]. The CrustCrawler robotic arm has 5 DOF using 7 AX-12A Dynamixel servo motors and has the RS-485 communication, USB interface and feedback. These motors are both ROBOTIS servo motors with the same communication type, however the AX-12A are considerably cheaper, costing $44.95 USD each [3] compared to the more expensive H-series Dynamixel Pro costing $1,490 USD [3]. Therefore, it is the aim of this project to achieve the same path planning software control possible on the ROBOTIS MANIPULATOR-H, on the AX-12A Smart Robotic Arm from CrustCrawler instead. This would create a low-cost demonstration of collision-free path planning and lead into the possibility to build the controllers needed to transfer this software movement into the robotic arm itself using the Dynamixel SDK.

Chapter 2

Literature Review

2.1 Introduction

This chapter presents the research conducted prior to the undertaking of the project, including related work and similar projects. Section 2.2 compares the design decision to avoid obstructions against moving each obstruction between the robotic arm and the goal object. Section 2.3 presents research into safety concerns and solutions associated with robotic arm workspaces and shows how a controlled workspace can be a static workspace or a dynamic workspace without unpredictable obstacles. Section 2.4 evaluates relevant projects including implementing robotic arm manipulation, ROS control and path planning.

2.2 Collision-free vs. Moving Obstacles

There are many types of dynamic workspaces which a robotic arm can be designed to navigate. In some cases, robotic arms are used to search for one or more specific colours [9, 10] or complex materials/bodies or differing shapes/sizes [11, 12]. The robotic arm identifies the desired object, either by colour, size, etc, then plans a collision-free path to navigate the end-effector from it's current position to the goal position. The identification of colours and shapes is performed with the use of image processing, which also can assist in mapping the workspace of the robotic arm. When an object is presented which is blocking the robotic arm from reaching the desired object, a design decision must be made either to path plan around the obstacle [4] or to move the obstacle [12]. Both designs result in the same desired position, however the moving of multiple objects to reach the desired object is time-consuming and presents more challenges than path planning collision-free around obstacles, therefore making the collision-free path planning around obstacles approach preferable.

2.3 Workspace Safety

Robotic arms are often used in the industry due to their repeatability and reliability compared to human labour, as well as their ability to work with hazardous conditions which are not appropriate for humans [13]. When working with robotic arms in this way, the workspace must be clearly defined and regulated to ensure safety for all aspects, including the robotic arm, the products being handled by the arm and any human presence. The four main categories of robotic incidents are; Impact or collision accidents, crushing and trapping accidents, mechanical part accidents (a breakdown in mechanical parts) and other accidents as a result of working with the robotic arm, such as environmental hazards, tripping hazards and hazard caused by extensions of the robotics, like power supplies or hydraulic lines [13]. Incidents and hazards can be avoided through safeguards which are implemented for the robotic arms workspace. Relevant safeguards include risk assessments and safeguard devices such as presence-sensing devices and fixed barriers to prevent contact with moving parts or human presents within the workspace. Perimeter fencing are designed to keep humans from entering the "danger zone", while presence-sensors detect human movement within the workspace and shut down movement to ensure the safety of the human [13].

By implementing perimeter fencing into a workspace in which the environment is fully known, static path planning can then be used as outside, unpredictable obstacles such as humans can be eliminated from the workspace. Additionally, once a workspace changes from static to dynamic by means of variable goal locations, such as when picking up select items from a pile of objects, then the corresponding dynamic path planning can be accomplished without dynamic obstacle avoidance of unpredictable obstacles such as humans, but rather navigating around stationary objects to select the correct/desired object.

2.4 Relevant Projects

2.4.1 Cheap Chess Robot

This project develops a robot capable of playing chess using ROS packages and image processing. The robotic arm used is the same robotic arm used within this project, the AX-12A Smart Robotic Arm from CrustCrawler [14]. This project had the benefit of using packages, namely "ua-ros-pkg", which are not currently available, which assisted with implementation from software-to-hardware, as well as displaying results and communicating with the Dynamixel motors. The result of the project was very positive, achieving the goal of a functioning chess playing robotic arm with less restrictions on the game itself, however the project did not use MoveIt! which is part of the goal of this project.

2.4.2 Collision-Free Manipulation

Vision, manipulation and planning is researched for the benefit of sorting objects where all objects must reach their goal locations. The solution is focused around combining ROS, MoveIt! and OMPL through the Cyton Gamma 300 which uses single MX-28 Dynamixel servo motors at each joint of the 7 DOF robotic arm [15]. This project is supplied a 3D model where a URDF, MoveIt! configuration and OMPL code is created, though this robotic arm is simpler to create OMPL files for due to its single-actuated joints, compared to the 2 dual-actuated joints in the AX-12A Smart Robotic Arm from CrustCrawler. The results of this project are positive in that the project achieves similar aims to this project, however the robotic arm used is at a much higher price point than the robotic arm used in this project, solidifying the goal of a low-cost solution using the AX-12A Smart Robotic Arm from CrustCrawler.

2.4.3 Planning for Tabletop Clutter

A low cost custom 5 DOF robotic arm is manipulated within a cluttered tabletop environment. The approach of this project, as discussed in Section 2.2, is to clear the collision area between the robotic arm and the desired object by moving obstacle objects aside. The robotic arm consists of 5 Dynamixel servo motors (2 MX-64T, 2 AX-12A and a single MX-106T) and costs $1788.2USD at time of writing [12]. The results of the project are very positive, achieving manipulation of the robotic arm and manipulation of simple geometries due to the simple end-effector gripper. The project does however take a different approach to this project, as discussed in Section 2.2 and is still not as low cost as the AX-12A Smart Robotic Arm from CrustCrawler.

2.4.4 Object Sorting in Cluttered Environments

A pipeline is presented which is used for the perception and manipulation of the viewing area to accurately sort objects by a specified property, such as size, colour or shape. Two motion primitives are used to manipulated the scene and the results are shown through ROS implementation onto the PR2 robot from Willow Garage [16]. The results of this project are overly positive, demonstrating the benefits of robotic manipulation in sorting clutter and motion planning. This project however does use the PR2 robot for manipulation and implementation through ROS, which packages are currently available and as described in Section 3.4.2, is considerably more expensive than the AX-12A Smart Robotic Arm from CrustCrawler used in this project.

Chapter 3

Background

3.1 Introduction

This chapter provides background information relating to the project, which is necessary to understand the implementation described in Chapter 4 and is organised as follows. Section 3.2 introduces the key software concepts and technologies used throughout the project. Section 3.3 describes the key hardware and components featured in the scope of the project. Section 3.4 lists robots related to this projects through their similarity and higher costs.

3.2 Software

3.2.1 Robot Operating System (ROS)

ROS is a framework for writing software for robotic applications. These sets of tools and libraries are open-source and encourage a collaborative approach to create robotic control and manipulation. Ubuntu is a open-source computer operating system based on Linux distribution and is fully supporting of ROS.

Key ROS Concepts

- The *tf* (transform) library manages coordinate transform data for robotic systems used within ROS. The library supports the defining of both static and dynamic transforms resulting in the tracking of robotic arm joints within any frame of reference [17].

- *catkin* Build System creates a *catkin workspace* with three folders. The original source code is contained within the "src/" folder, whilst the build space is within "build/" and the development space is within "devel/". Both the build/ and devel/ folders are created using the *catkin make* command and changes based on the user edited code and installed packages within the src/ folder.

- *roslaunch* is a tool used to launch multiple ROS nodes. The XML files defines which nodes to launch based of the *.launch* extension [18]. *roslaunch* is summoned within the command line inside the workspace "src/" folder as follows;
  `$ roslaunch <package_name> <launch_file_name>.launch`

Key ROS-Related Tools

- *MoveIt!* is an open-source software for the motion planning, control, kinematics, navigation and manipulation of robots [19].

- *rviz* is used for the 3D visualisation of URDF described robots and functions with *MoveIt!* for 3D simulation and motion planning.

- *rqt_tools* allows for graphical representations of ROS nodes, topics and messages through navigation stacks.

- URDF (Unified Robot Description Format) is a format for describing a robot in a machine-readable language. The .urdf file describes the physical properties of the robotic arm, including details on all joints and links that form a robot. Joints contain information such as locations, axis and limits. Links detail information such as mass, inertial values, origin and visual appearance. This file is then used with the *tf* library, rendering in 3D for visualisation, simulations and motion planning using *rviz* and *MoveIt!* [17].

- *Gazebo* is a 3D real-time simulator which can display URDF described robots similar to *rviz*. This tool can also be used to test the implementation of robotic controllers before implementing to hardware [20].

3.2.2 RoboPlus and "Mixcell"

RoboPlus is a program from ROBOTIS which is used to manage the Dynamixel servo motors using the USB2Dynamixel described in Section 3.3. The "Dynamixel Wizard" is used to modify the ID, baud rate, angle/motor limits and joint/wheel mode of Dynamixel servos. RoboPlus is compatible with Windows, however is not so useful when using Ubuntu, therefore "Mixcell" [21] is a beneficial open-source program which can be used as substitute. "Mixcell" is used for the same purpose of "Dynamixel Wizard", however functions within Ubuntu.

3.2.3 SolidWorks-to-URDF Exporter

The SolidWorks-to-URDF exporter [22] is an open-source SolidWorks add-in which enables a URDF file to be created using a pre-existing 3D drawing. A SolidWorks assembly can be built into links using mesh shapes and joints to create an URDF file. Additionally, since each SolidWorks part has mass, each links moment of inertia in all frames can be automatically calculated. This is extremely useful and saves the need for manual calculations for all links in all frames.

3.3 Hardware

3.3.1 AX-12A Smart Robotic Arm by CrustCrawler

The 5 DOF, 7 actuator robotic arm used throughout this project is the "AX-12A Smart Robotic Arm" from CrustCrawler. The material of this robotic arm is 6061 aluminium with a black hard anodize finish. The actuators are AX-12A Dynamixel servo motors with two DOF joints are dual motor controlled. The motors are connected in daisy chain and provide feedback for position, velocity, current, voltage and temperature. The joint configuration is shown in Figure 3.1 with the "gripper motor" controlling the end-effector pinch motion through gears and dependant joints. The side view shown in Figure 3.2 originates from a 3D drawing of the robotic arm [1] which shows the dimensions of the robotic arm in millimetres.

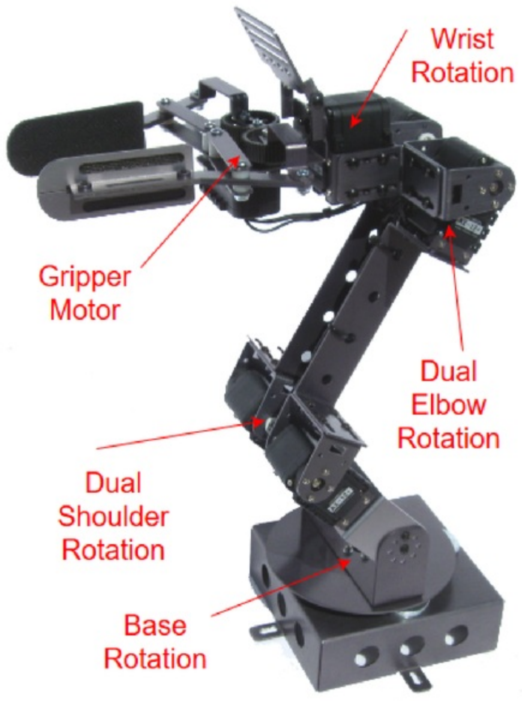


Figure 3.1: Robotic Arm Joint Configuration [1]

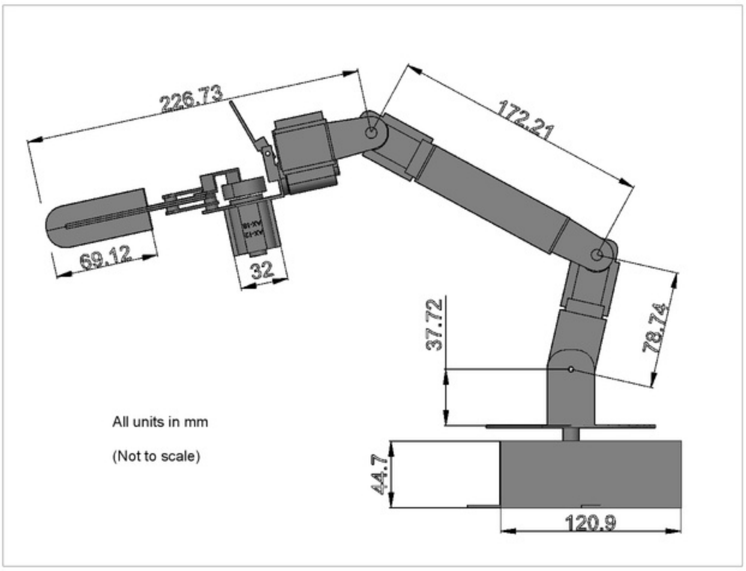


Figure 3.2: AX-12A Smart Robotic Arm Side View

3.3.2 USB2Dynamixel

The USB2Dynamixel shown in Figure 3.3 connects the Dynamixel servo motors to a PC for customisation through RoboPlus/"Mixcell", as described in Section 3.2.2. The device communicates via TTL communication (via the function selection switch) and uses the 3-pin port, as shown in Figure 3.3.

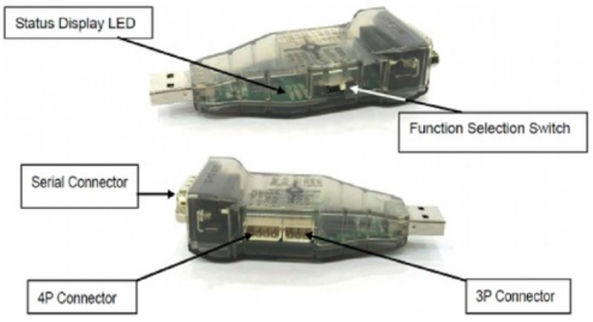


Figure 3.3: USB2Dynamixel [2]

3.4 Related Robots Running ROS

3.4.1 Robotis Manipulator-H

Robotis Manipulator-H is a 6 DOF robotic arm which features the H-series "Dynamixel Pro" servo motors, costs $18,900 USD [3] and has open-source packages available for ROS control and manipulation [23]. This includes a SDK for software-to-hardware control, MoveIt! path planning and end-effector control. The URDF file of the robotic arm is shown in Figure 3.4.

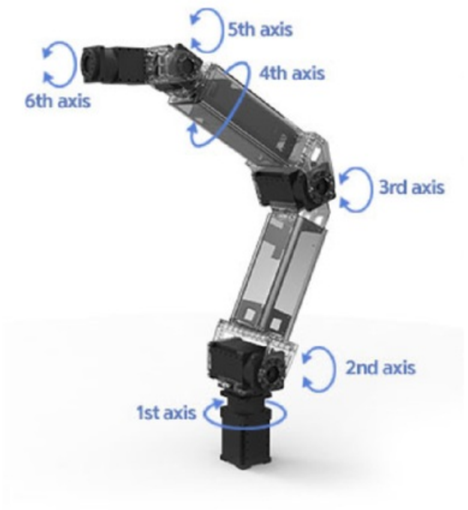


Figure 3.4: Robotis Manipulator-H [3]

3.4.2 Personal Robot 2 (PR2)

PR2 from Willow Garage is a robotics research and development platform with built software designed for ROS application. This innovative device is priced at $280,000 (plus taxes and shipping) [24] and is leading for tutorials and open-source code for ROS applications, such as MoveIt! which is applied in this project [25].

3.4.3 Cyton Epsilon 300

The Cyton Epsilon 300 is a robotic arm from ROBAI which provides packages for ROS control and manipulation. ROS control includes end-effector control and joint control. The robotic arm has 7 DOF and advertises prices starting under $5000 [26].

Chapter 4

Implementation

4.1 Introduction

This chapter provides the key information regarding the implementation of the project and is organised as follows. The remainder of Section 4.1 outlines the setup of the software, installing of packages required for the project. Section 4.2 explains the process behind manipulating the supplied 3D drawing into the framework to convert the drawing into a URDF file. Section 4.3 describes the key elements in the created the URDF file, including link meshes and inertia tensors. Section 4.4 details key details in the MoveIt! configuration created from the URDF file.

4.1.1 Software Setup

Ubuntu 16.04.3 operating system was implemented on a virtual machine (VMware Workstation 12) and ROS Kinetic was installed. This version of ROS was chosen for it's full compatibility with Ubuntu 16.04.3, rather than the newer version Lunar.
Note: The decision to run the Ubuntu operating system on a virtual machine rather than on the primary machine is due to the lack of a spare computer and the convenience of running both report and project on one computer.
A workspace named "catkin_ws" was created within Ubuntu using the *catkin* build system. The folder tree that was created is shown in Figure 4.1.

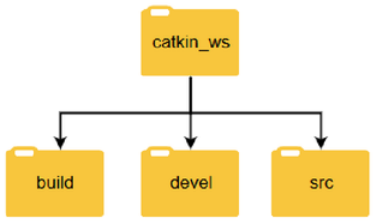


Figure 4.1: Folder tree for *catkin* workspace

Some packages suggested by tutorials were installed, including "python-rosinstall" and "python-catkin-tools", before `$ /catkin_ws/devel/setup.bash` was executed to configure the ROS environment.

4.2 3D Model

A 3D drawing (.iges file) of the AX-12A Smart Robotic Arm by CrustCrawler [1] was used for this section of the project as the base work to form a URDF file in Section 4.3. This was preferred over creating a URDF file using primitives and the dimensions of the robotic arm as this was more presentable, accurate and already existed free of charge.

4.2.1 Materials

The first step of improving the 3D model was adding material to each part of the robotic arm, which allows for an accurate calculation of the mass moment of inertia needed in the URDF file in Section 4.3. The plastic gears attached to the last motor on the end-effector of the robotic arm are plastic while the small dependant joints have small rubber tubes. All of the metal frame is 6061 aluminium which only left the Dynamixel servo motors. These motors clearly are not made from one solid material, so its mass and volume was used to calculate an appropriate material with similar density. Using $p = m/V$ where $p =$ density, $m =$ mass (55g) and $V =$ volume (32 x 50 x 40mm) [27], we find $p = 859.375$ kg/m^3 and use a material with similar density within SolidWorks.

4.2.2 SolidWorks Parts

Simplifying the SolidWorks assembly was required to streamline the SolidWorks-to-URDF process, as instead of highlighting all parts that combine to make one link. Rather, its possible to combine all small parts to one large link before the process is undertaken. As a result, 46 parts was reduced to 6 parts where each part is a link between joints defined in the URDF file in Section 4.3. Additionally, the base part in the 3D model [1] is not square but rather slightly askew. Therefore, this was adjusted to simplify the assembly and the following URDF default pose.

4.2.3 Reference Geometry

This robotic arm has 5 DOF, each of which being a revolute joint featuring at least one actuator. Each of these joints must be defined within the URDF file and therefore their co-ordinate systems and axis' must be referenced within SolidWorks. A plane was drawn down the centre of the robotic arm, parallel to the viewing angle in Figure 4.2. Each co-ordinate system was drawn at the point of actuation, with the z-axis pointing perpendicular to the joint's corresponding actuator. A fixed joint has also been defined at the adjustable joint on the robotic arm, as this allows for adjusting in software to suit the hardware setting, increasing flexibility and workability of the model. Additionally, reference axis were drawn along the same z-axis' for the SolidWorks-to-URDF conversion. The location for each joint and its co-ordinate system is shown in Figure 4.2.

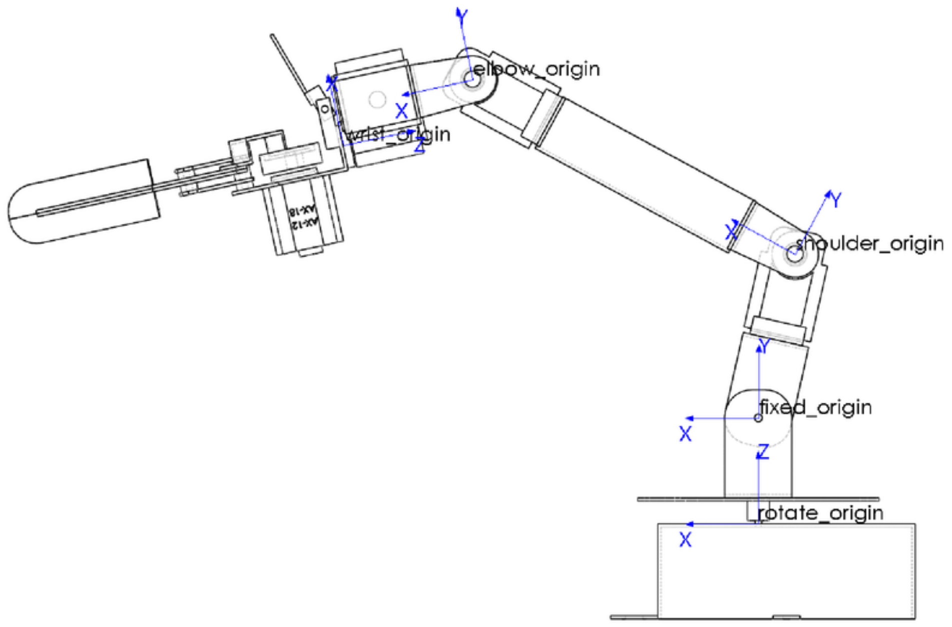


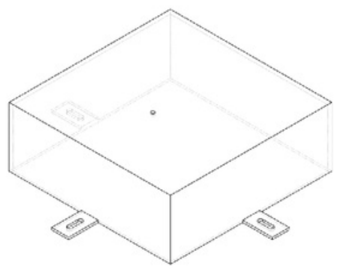
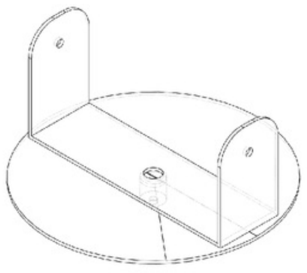
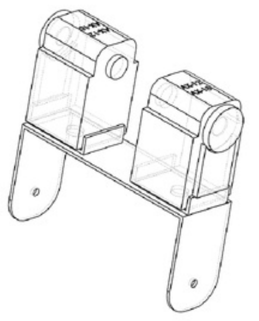
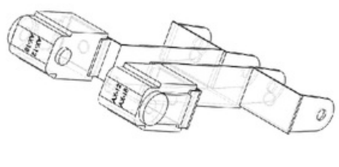
Figure 4.2: Co-ordinate Systems at each joint

4.3 URDF File

Creating the URDF file from the SolidWorks assembly is completed using the SolidWorks-to-URDF add-in [22]. The co-ordinate systems and reference axis' described in Section 4.2 are stated for each joint, while each link is given its corresponding SolidWorks part.

4.3.1 Link Meshes

Link meshes are the defined shape of each part which links two joints together. The robotic arm URDF functions as a chain of parent-to-child links, which is shown correspondingly from top-down in Table 4.1. The link names correlates to the two joints that correspond to each one, for example, "link_shoulder2elbow" is the link between the joints named "joint_shoulder" and "joint_elbow". The parent-to-child order of the robotic arms joints is shown in Table 4.2 and 4.3.

Link Name	Link Mesh	Controlling Joint
base_link		no actuation
link_rotate2fixed		"joint_rotate" Type = revolute
link_fixed2shoulder		"joint_fixed" Type = fixed
link_shoulder2elbow		"joint_shoulder" Type = revolute

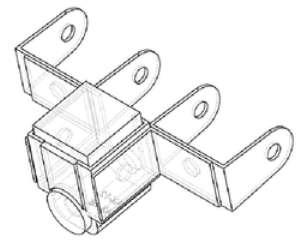
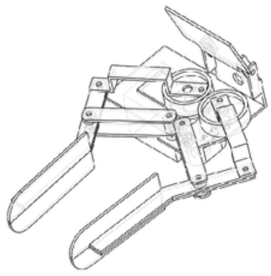
link_elbow2wrist		"joint_elbow" Type = revolute
link_end_effector		"joint_wrist_rotate" Type = revolute

Table 4.1: Link Meshes

4.3.2 Link Properties

Each link is individual in its shape, size, mass and as a result has a corresponding mass moment of inertia. This is calculated by SolidWorks technology, identifying the density, mass, volume, surface area, centre of mass and then performing the calculations internally. This information can be viewed using **Mass Properties** before using the SolidWorks-to-URDF exporter. This exporter uses the same SolidWorks technology to calculate the mass moment of inertia for each link, which is far too advanced to do without computer assistance given the complexity of most links. The formula for mass moment of inertia (inertia tensor) for each link is shown in Equation 4.1.

$$I = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} \tag{4.1}$$

Joint Name	**Position (x, y, z) m**
joint_rotate	(0, 0, 0)
joint_fixed	(0, 0, 0.049784)
joint_shoulder	(-0.017186, 0.077011, 0)
joint_elbow	(0.17221, 0, 0)
joint_wrist_rotate	(0.066193, -0.017501, 0)

Table 4.2: Joint Origin Position Transforms

4.3.3 Joint Transforms and Limits

The origin of each joint is the change from the previous joint, such that the origin of "joint_elbow" is the change from the origin of "joint_shoulder" to that location. These origin position transforms are shown in Table 4.2 and their origin rotation transforms are shown in Table 4.3.

Joint Name	**Rotation (x, y, z) rad**
joint_rotate	(0, 0, 0)
joint_fixed	(1.5708, 0, 0)
joint_shoulder	(0, 0, 0.7096)
joint_elbow	(0, 0, -0.7096)
joint_wrist_rotate	(-1.5708, 0, 1.5708)

Table 4.3: Joint Origin Rotation Transforms

Excluding "joint_fixed" which is a "fixed" joint, all joints named on the robotic arm are "revolute" joints. These joints are controlled in hardware by AX-12A Dynamixel servo motors and have limits whilst set in "joint mode". Joint mode results in a 300 degree rotation radius, from 30 degrees to 330 degrees as shown in Figure 4.3.

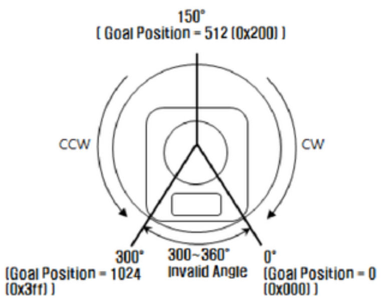


Figure 4.3: Joint Mode Limits

Additionally to these motor limits, the URDF must have more specific limits as the robotic arm should not collide with itself. While the MoveIt! configuration in Section 4.4 does not allow for self-collision, the URDF also defines limits for each joint to avoid such a situation. A lower and upper limit is defined for each joint and is based on the default pose of the motor. These values are shown in Table 4.4.

Joint Name	lower (rad)	upper (rad)
joint_rotate	-2.618	2.618
joint_shoulder	-3.3	1.3247
joint_elbow	-2.7	1.23
joint_wrist_rotate	-2.618	2.618

Table 4.4: URDF Joint Limits

4.3.4 Navigation Stack

Following the competition of the URDF file, it can be displayed using the following command:

```
$ roslaunch robot_arm display.launch
```

Whilst running this display, the navigation stack can be sourced using *rqt_graph*, which outputs as Figure 4.4.

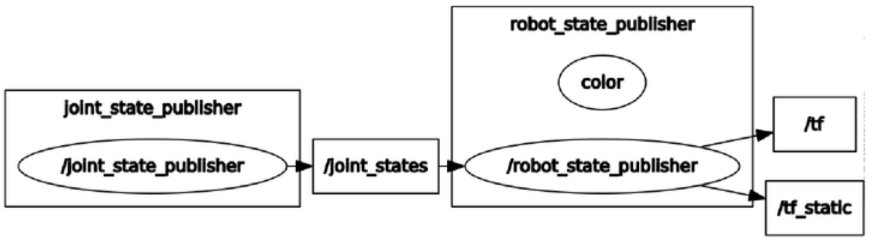


Figure 4.4: Navigation Stack of URDF

4.4 MoveIt! Configuration

The MoveIt! configuration file was created using the open-source MoveIt! setup assistant using the following command:

```
$ roslaunch moveit_setup_assistant setup_assistant.launch
```

By loading the URDF file, the setup assistance contains 8 steps to follow to form the completed MoveIt! configuration file and accompanying SRDF file. A full tutorial using the PR2 robot described in Section 3.4.2 was used which described each step [25] and was extrapolated onto the robot_arm URDF file and its MoveIt! configuration.

4.4.1 Self-Collision Matrix

By default, a MoveIt! configuration assumes that any link could come into collision with any other link in the robotic arm, resulting in a self-collision. A tag called "disable_collisions" can be used to eliminate checking some collisions between links when they are known to be impossible to occur. Table 4.5 shows all link pairs which have disabled collisions in the MoveIt! configuration and the reason for their non-collision. "Never" means that the link cannot collide with the other within its joint constraints, whilst "adjacent" means that links are attached to each other directly and cannot collide with each other. This information is stored as code in the SRDF file, which is used to generate the MoveIt! configuration.

Link 1 Name	**Link 2 Name**	**Reason**
base_link	link_fixed2shoulder	Never
base_link	link_rotate2fixed	Adjacent
base_link	link_shoulder2elbow	Never
link_elbow2wrist	link_end_effector	Adjacent
link_elbow2wrist	link_fixed2shoulder	Never
link_elbow2wrist	link_shoulder2elbow	Adjacent
link_end_effector	link_fixed2shoulder	Never
link_end_effector	link_shoulder2elbow	Never
link_fixed2shoulder	link_rotate2fixed	Adjacent
link_fixed2shoulder	link_shoulder2elbow	Adjacent
link_rotate2fixed	link_shoulder2elbow	Never

Table 4.5: Link Pairs with Disabled Collision Checks

4.4.2 Navigation Stack

Following the competition of the MoveIt! configuration file, it can be displayed using the following command:

```
$ roslaunch moveit_config demo.launch
```

Whilst running this display, the navigation stack can be sourced using *rqt_graph*, which outputs as Figure 4.5 for all active nodes and topics, whilst Figure 4.6 outputs nodes only, making for a simpler diagram.

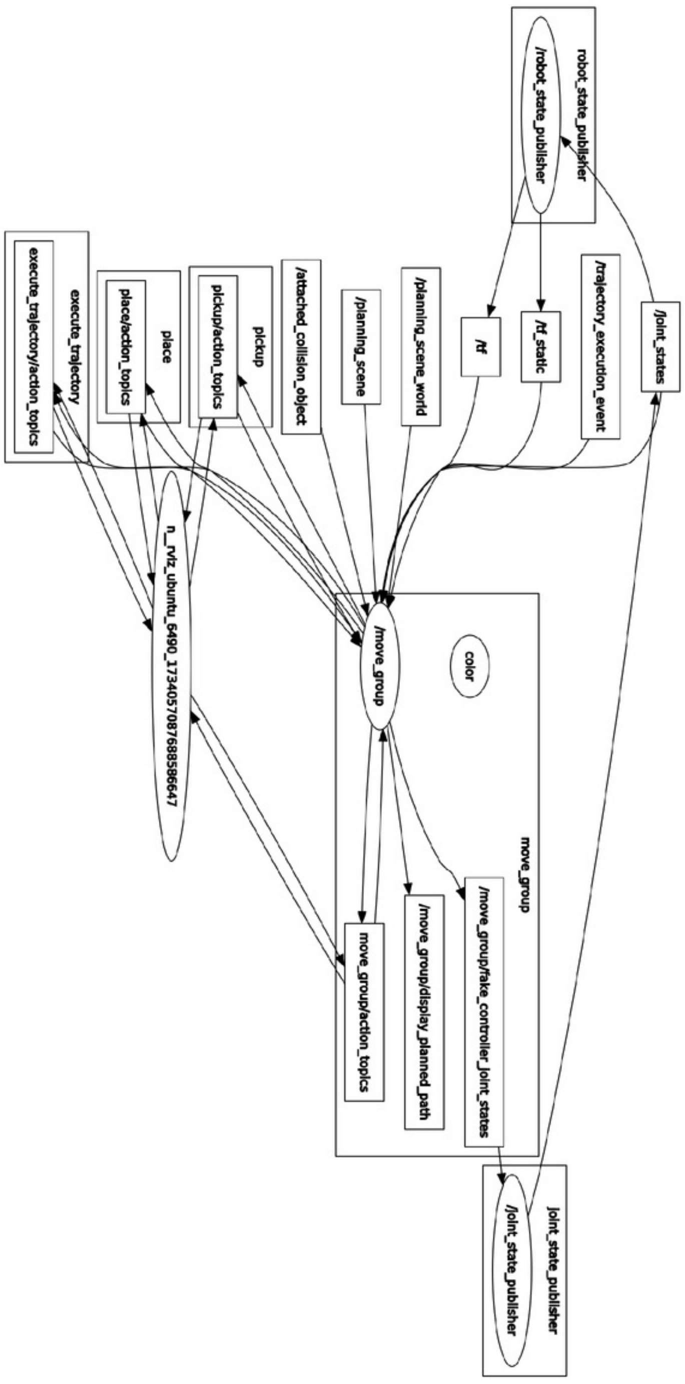


Figure 4.5: Navigation Stack of Active Nodes and Topics

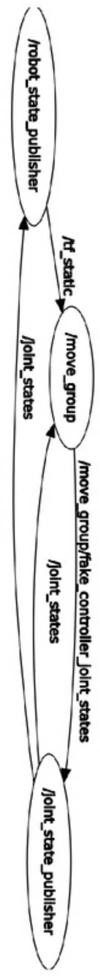


Figure 4.6: Navigation Stack of Nodes Only

Chapter 5

Results

5.1 Introduction

This chapter presents how the URDF and MoveIt! configuration outlined in Chapter 4 were tested and the results that were obtained. Section 5.2 displays the results gathered from the created URDF file. Section 5.3 provides the output results of the MoveIt! configuration file as well as the results from path planning and success of collision-free movement. Section 5.4 shows the collision detection within the MoveIt! configuration for both self-collision and collision with known objects. Section 5.5 describes the detailed workspace of the robotic arm.

5.2 URDF Display

The results from the creation of a URDF file which replicates the AX-12A Smart Robotic Arm from CrustCrawler was successful. The major difference between the URDF file and the hardware is the end-effector. This model has taken the end effector as a single link, with no movement due to the joint complexity required in software to perform its realistic movement. The result of the URDF is shown in *rviz* in Figure 5.1. Each joint is represented by their axis, in which each z-axis is perpendicular to the actuator controlling the joint.

5.3 MoveIt! Path Planning

5.3.1 Path Planning in MoveIt!

The process of path planning is successful for all "random valid" goal states of the robotic arm with a goal tolerance set to "0". The "Planning" tab within *rviz* allows the user to update the goal state to "random valid" or "random" configuration and then select "Plan" for the robotic arm's path. The path planning process is consistently performed in less than 0.2 seconds with an average of less than 0.1 seconds, as described in Section 5.3.4

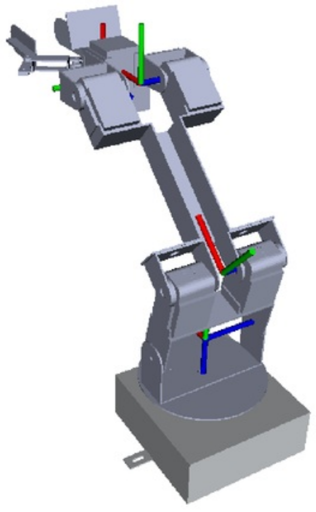


Figure 5.1: URDF displayed in *rviz*

and the simulation can then display the planned path. The starting state is displayed in grey and the goal state is displayed in orange as shown in Figure 5.2.

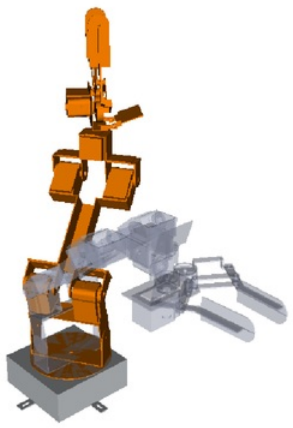


Figure 5.2: Start Position (grey) to Move Goal (orange)

5.3.2 Default Pose to Random Goal Configuration

The starting state has been set to "default" whilst the goal state has been set to a "random" configuration. Figure 5.3 and Figure 5.4 show the planned path through a path trail, with Figure 5.4 showing the trail more frequently.

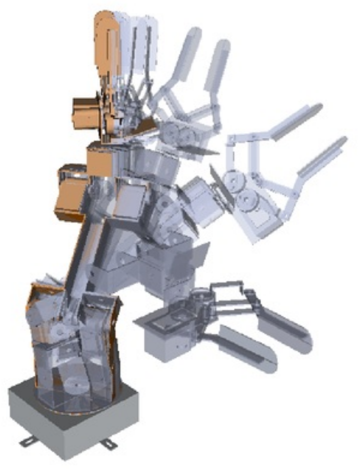


Figure 5.3: Default Start Position Path Planning

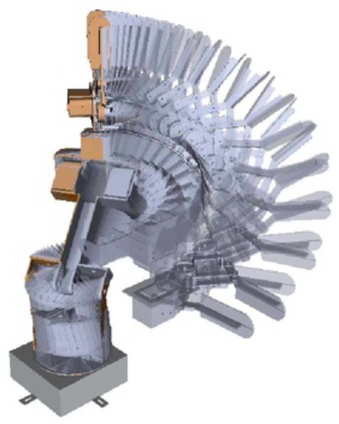


Figure 5.4: Default Start Position Path Planning with more frequent trail

5.3.3 Random Start to Random Goal Configuration

The starting state has been set to "random" whilst the goal state has also been set to a "random" configuration. Figure 5.5 and Figure 5.6 show the planned path through a path trail, with Figure 5.6 showing the trail more frequently.

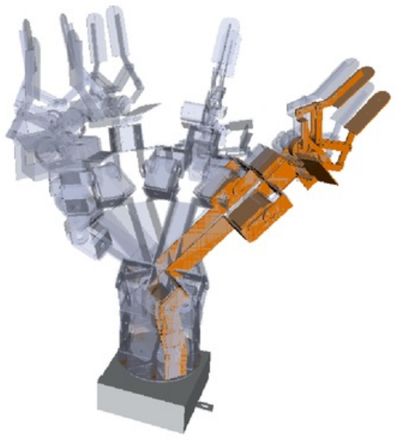


Figure 5.5: Random Start Position Path Planning

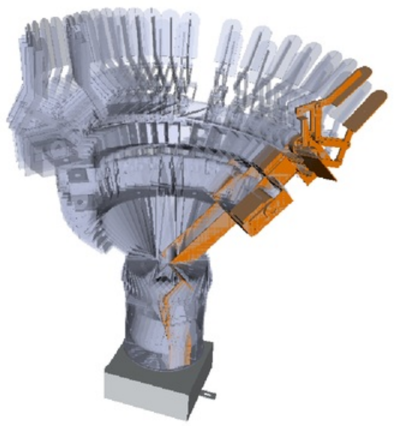


Figure 5.6: Random Start Position Path Planning with more frequent trail

5.3.4 Average Time Taken to Establish Planned Path

The time taken for the simulation to plan a collision-free path is a important result to the project. This does depend on the amount of processing power the computer contains, with this virtual machine running 8GB of RAM and 2 processing cores. A test was conducted by planning a path for a goal state 10 times and taking the average. This was then repeated 3 times for 3 different goal states and the results are shown in Table 5.1.

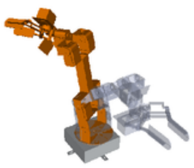
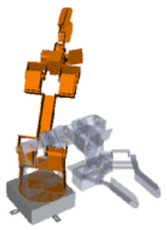
	 Goal State 1	 Goal State 2	Goal State 3
Attempt #	Time to Plan (s)	Time to Plan (s)	Time to Plan (s)
1	0.082	0.149	0.058
2	0.044	0.067	0.083
3	0.037	0.056	0.091
4	0.049	0.073	0.074
5	0.124	0.075	0.063
6	0.054	0.035	0.08
7	0.085	0.1	0.074
8	0.044	0.064	0.108
9	0.029	0.056	0.046
10	0.057	0.037	0.051
Average	**0.0605**	**0.0712**	**0.0728**

Table 5.1: Average Time Taken to Establish Planned Path

5.4 Collision Detection

Path planning within a static workspace involves providing the software with all known obstacles contained which could cause a collision. The two key types of obstacles that can cause a collision is the self-collision of the robotic arm and a collision with items within the workspace. The MoveIt! configuration provides us with results for the first type of collision, self-collision, as shown in Figure 5.7 where the "random" goal state was a state which involved a self-collision, shown in red as a collision between "base_link" and "link_end_effector". As a result, the path will not be planned as the software recognises that this is an "invalid" state and would cause a collision.

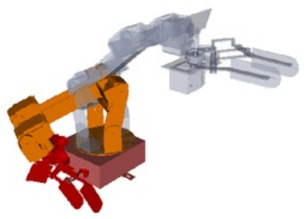


Figure 5.7: Self-Collision Detection in *rviz*

5.5 Defined Workspace

The workspace of the robotic arm is an important result to consider when performing static path planning safely as discussed in Section 2.3. When placed on a flat surface such as a table, the robotic arm has a set workspace in which it can function, as shown in Figure 5.8. It should be noted that the workspace is not a simple sphere about any single joint. Due to the configuration of the robotic arm, the workspace spans a reach of 391.72mm from the "shoulder" joint of the robotic arm, which rotates about the base rotating joint. The joint is limited from 30-330 degrees, such that the remaining 60 degrees is smaller in span. The current rotation of the base is at 180 degrees and the highest point of the workspace is slightly off-centre due to the angle of the "fixed" joint.

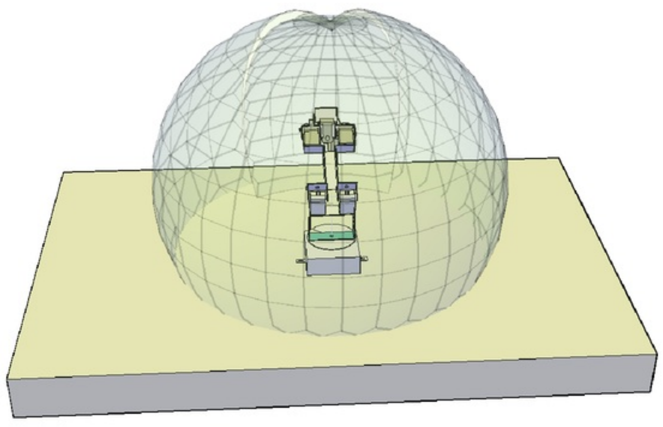


Figure 5.8: Robotic Arm Workspace when placed on flat surface

Chapter 6

Discussion

6.1 Assessment of Project

All goals for the project as described in Section 1.2, were implemented and have supplied the foundation for future work to further the research into a low-cost robotic arm used for path planning within ROS. There are some shortcomings for the project which are discussed in Section 6.2 and while this can be perceived as future work, it would be beneficial to implement these improvements to the project in future. Additionally, the implementation to hardware for this project would be greatly beneficial, as discussed in Section 6.3. This project builds the foundation for future improvements in ROS control for the low-cost robotic arm, with more rigid simulations, tests and ROS control possible with the future of the project as discussed in Section 6.4.

6.2 Limitations of Project

6.2.1 4 DOF Simulation of a 5 DOF Robotic Arm

During this project, the end-effector link featuring the 7th Dynamixel servo motor controlling the "pinch joint", is considered as one link with no functioning joint. This is due to the complexity of the end-effectors motion and the challenge of re-creating this movement in software. The link mesh as shown in Table 4.1 named "link_end_effector" consists of one Dynamixel servo motor on the underbelly of the link which is connected to a parent gear and a child gear of equal size and teeth. When the motor moves, the end effect joint moves forward and the two "fingers" diverge, or the end effector joint retreats and the "fingers" separate. Implementing this motion in ROS is challenging and for simplicity in the project, was neglected. This movement is similar to the movement of a prismatic joint for the sake of simulation, however the movement in hardware and the values of the motor that this corresponds to is a more challenging task to understand and implement into the URDF, SRDF models and the MoveIt! configuration.

6.2.2 Obstacles within Robotic Arm Workspace

While this project demonstrates the MoveIt! configuration's ability to plan a path from its present state to a goal state without self-collision, it does not supply an example of avoiding collision with another object/entity within the simulation. This would have involved placing an object within the scene of the MoveIt! configuration and within the workspace of the robotic arm. Between the two states, an object would have been placed which would have collision conditions such that the path planning would have been demonstrated to avoid a collision with the object.

6.2.3 End-effector Control

Rather than a goal state for the robotic arm to path plan to, an end-effector co-ordinate would have been preferable. This would use open-source ROS MoveIt! IK packages which run the inverse kinematics to allow the robotic arm to determine a viable configuration to achieve the end-effector location within the workspace and without collision with self or other objects within the workspace. The main issue with this is that the IK packages supplied by MoveIt! are for 6 DOF robotic arms and above and do not support lower DOF robotic arms to my knowledge at present.

6.3 Implementation to Hardware

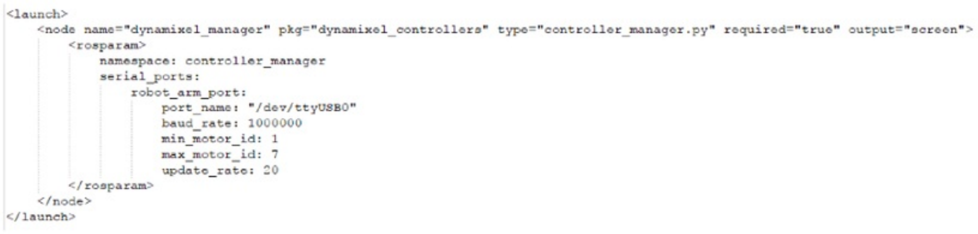
Implementing the simulation of static path planning of the robotic arm onto hardware would be very beneficial to the project, as this would allow for a greater gather of results, testing accuracy and goal tolerance when applied to hardware, rather than the goal tolerance of 0 and complete accuracy that the simulation provides. The implementation for software-to-hardware integration within MoveIt! and ROS for Dynamixel servo motors is performed through a Joint Trajectory Action Controller (JTAC).

6.3.1 Joint Trajectory Action Controller

A JTAC takes a list of joint positions, velocities, accelerations and efforts for each joint of the robotic arm over a period of time as an input. This can be used to implement the simulation of the robotic arm in MoveIt! onto the robotic arm. A desired goal configuration of the robotic arm is sent to MoveIt! using a *rviz* GUI plugin, or through the command-line interface. MoveIt! generates the planned path in the form of a **trajectory_msgs/JointTrajectory** message, before this is sent to the JTAC by connecting the **follow_joint_trajectory** topic. The steps involved to achieving this JTAC is detailed below, using open-source software referenced in code [20].

Connect to Dynamixel Motors

Create a *.launch* file to manage the controller that connects the motors and publishes raw feedback data at a specified rate. An example of this for the project is shown in Figure 6.1.

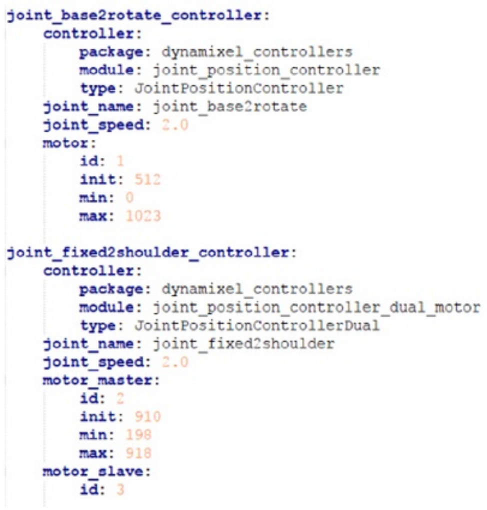


```
<launch>
    <node name="dynamixel_manager" pkg="dynamixel_controllers" type="controller_manager.py" required="true" output="screen">
        <rosparam>
            namespace: controller_manager
            serial_ports:
                robot_arm_port:
                    port_name: "/dev/ttyUSB0"
                    baud_rate: 1000000
                    min_motor_id: 1
                    max_motor_id: 7
                    update_rate: 20
        </rosparam>
    </node>
</launch>
```

Figure 6.1: Example code for controller_manager.launch

Create a Meta Controller

A meta controller is an action server that allows individual servo motors to be associated with each joint and control this by an action client. A joint position controller should be connected to each joint, an example for this project is shown in Figure 6.2. The AX-12A Smart Robotic Arm from CrustCrawler features both single-actuated joints and dual-actuated joints, which is represented in Figure 6.2 through the "joint_position_controller" and "joint_position_controller_dual_motor" respectively.

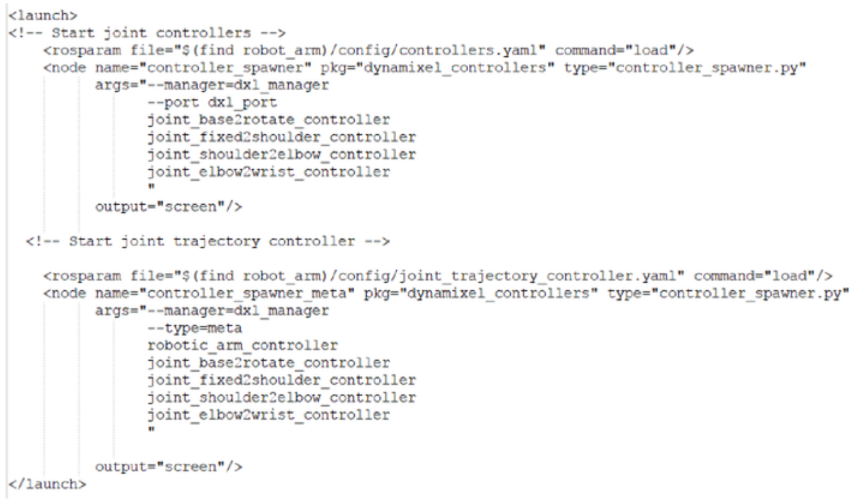


```
joint_base2rotate_controller:
    controller:
        package: dynamixel_controllers
        module: joint_position_controller
        type: JointPositionController
    joint_name: joint_base2rotate
    joint_speed: 2.0
    motor:
        id: 1
        init: 512
        min: 0
        max: 1023

joint_fixed2shoulder_controller:
    controller:
        package: dynamixel_controllers
        module: joint_position_controller_dual_motor
        type: JointPositionControllerDual
    joint_name: joint_fixed2shoulder
    joint_speed: 2.0
    motor_master:
        id: 2
        init: 910
        min: 198
        max: 918
    motor_slave:
        id: 3
```

Figure 6.2: Example code for controllers.yaml

A configuration file for JTAC should also be created which specifies the minimum velocity and constraints such as goal time. Finally, a *.launch* is required to load the controller parameters, for which an example for this project is shown in Figure 6.3.



```
<launch>
<!-- Start joint controllers -->
    <rosparam file="$(find robot_arm)/config/controllers.yaml" command="load"/>
    <node name="controller_spawner" pkg="dynamixel_controllers" type="controller_spawner.py"
          args="--manager=dxl_manager
                --port dxl_port
                joint_base2rotate_controller
                joint_fixed2shoulder_controller
                joint_shoulder2elbow_controller
                joint_elbow2wrist_controller
                "
          output="screen"/>

  <!-- Start joint trajectory controller -->

    <rosparam file="$(find robot_arm)/config/joint_trajectory_controller.yaml" command="load"/>
    <node name="controller_spawner_meta" pkg="dynamixel_controllers" type="controller_spawner.py"
          args="--manager=dxl_manager
                --type=meta
                robotic_arm_controller
                joint_base2rotate_controller
                joint_fixed2shoulder_controller
                joint_shoulder2elbow_controller
                joint_elbow2wrist_controller
                "

          output="screen"/>
</launch>
```

Figure 6.3: Example code for start_meta_controller.launch

This example code shown in Figure 6.1, 6.2 and 6.3 are only samples of some written code and therefore are not a solution to this specific project and as such should not be taken as functional code. While this method should work correctly for Dynamixel servo motors since it sources the Dynamixel SDK for dynamixel_controllers and follows similar steps to functioning open-source examples [20], it has not been implemented in this project.

6.4 Future Work

This project begins to build the foundation for future works between the AX-12A Smart Robotic Arm from CrustCrawler and ROS. Additionally, this project demonstrates the use of static path planning using ROS and performing this task using a low-cost robotic arm. This can be used as the foundation for any number of related goals and projects, as described in this section.

Implementing the End-effector Motion

The end-effector in this project is taken as a single link with no actuation, as discussed in Section 6.2.1. There is an opportunity for future work to implement the correct motion for this robotic arm, giving the simulation 5 DOF as the robotic arm has, rather than the 4 DOF URDF presented in the project.

End-effector Control

As discussed in Section 6.2.3, the project does not use IK to path plan to a goal end-effector location. This would be extremely beneficial to the project and its research as it would enable the model to explore channels such as object sorting and further explore path planning as discussed in Section 2.4.

Implementation to Hardware

Discussed in Section 6.3, a JTAC could be build using Dynamixel SDK to implement the MoveIt! configuration software simulation onto the robotic arm. This would enable tests such as the accuracy and goal tolerance of the hardware, as well as open multiple avenues for demonstrations and applications.

Image Processing

There are many applications for using image processing in conjunction with a ROS controlled robotic arm, such as the projects discussed in Section 2.4 including object sorting based on multiple factors, such as size or colour. Furthermore, there are many open-source packages available in ROS for such image processing and can be integrated with the existing structures within this project.

Dynamic Path Planning

This project focuses on static path planning within a non-changing and known workspace of the robotic arm. As mentioned in Section 2.2, robotic arms can be used to search for one or more specific colours [9,10] or complex materials/bodies or differing shapes/sizes [11,12] within a dynamic workspace. Using ROS to navigate the a dynamic workspace with collision-free path planning would be very beneficial for future research, further ensuring the safety of workers as discussed in Section 2.3 as well as advancing the model.

Chapter 7

Conclusions

Collision-free static path planning has been simulated using ROS to control a low-cost 4 DOF robotic arm. The benefit of the robotic arm used is the fractional cost compared to robotic arms used in similar projects and in similar ROS control applications. The AX-12A Smart Robotic Arm from CrustCrawler used has limited open-source ROS software associated with it, in addition to no accessible simulations of the robotic arm within ROS. The implementation of this simulation required a URDF file, SRDF and MoveIt! configuration of the robotic arm to be created, which together formed the AX-12A Smart Robotic Arm from CrustCrawler in software, however lacked the "pinch" motion of the 5th DOF. The simulation demonstrated the expected results of collision-free path planning from any valid position to a demanded goal state with average path planning time of less than 0.1 seconds and a goal tolerance of zero. The simulation identified configurations that include self-collision and did not path plan to this collision, which demonstrated the collision-free path planning within an empty workspace. Further simulations are required to demonstrate the collision-free path planning about present obstacles within the static workspace to further the research conducted. This research was significant for the use of low-cost robotic arms within both research and industrial industries, with the capability of accurately executing path planning to a goal state and operating without collision.

Appendix A

URDF file (robot_arm.urdf)

```
<robot
name="robot_arm">
<link
name="base_link">
<inertial>
<origin
xyz="0 0 0"
rpy="0 0 0" />
<mass
value="0.14802" />
<inertia
ixx="0.00031882"
ixy="2.4634E-10"
ixz="2.1467E-06"
iyy="0.00031355"
iyz="1.0027E-10"
izz="0.00056542" />
</inertial>
<visual>
<origin
xyz="0 0 0"
rpy="0 0 0" />
<geometry>
<mesh
filename="package://robot_arm/
meshes/base_link.STL"/>
</geometry>
<material
name="">
<color
rgba="0.89804 0.91765 0.92941 1" />
</material>
</visual>
<collision>
<origin
xyz="0 0 0"
rpy="0 0 0" />
<geometry>
<mesh
filename="package://robot_arm/
meshes/base_link.STL"/>
</geometry>
</collision>
</link>
<link
name="link_rotate2fixed">
<inertial>
<origin
xyz="0 0.0588478918345656 0"
rpy="0 0 0" />
<mass
value="0.025446" />
<inertia
ixx="3.2059E-05"
ixy="6.786E-08"
ixz="-7.1623E-09"
iyy="1.6385E-05"
iyz="-1.9678E-08"
izz="4.1799E-05" />
</inertial>
<visual>
<origin
xyz="0 0 0"
rpy="0 0 0" />
<geometry>
<mesh
filename="package://robot_arm/
meshes/link_rotate2fixed.STL"/>
</geometry>
<material
name="">
<color
rgba="0.79216 0.81961 0.93333 1" />
</material>
</visual>
<collision>
<origin
xyz="0 0 0"
rpy="0 0 0" />
<geometry>
<mesh
filename="package://robot_arm/
meshes/link_rotate2fixed.STL"/>
</geometry>
</collision>
</link>
<joint
name="joint_rotate"
type="revolute">
<origin
xyz="0 0 0"
rpy="0 0 0" />
<parent
link="base_link" />
<child
link="link_rotate2fixed" />
<axis
xyz="0 0 1" />
<limit
lower="-2.618"
upper="2.618"
effort="0"
velocity="3.33" />
</joint>
<link
name="link_fixed2shoulder">
<inertial>
<origin
xyz="0.137339568094239 0 0"
rpy="0 0 0" />
<mass
value="0.1155" />
<inertia
ixx="0.00018406"
ixy="6.0679E-06"
ixz="4.0292E-12"
iyy="0.00015855"
iyz="2.271E-12"
izz="4.5553E-05" />
</inertial>
<visual>
<origin
xyz="0 0 0"
rpy="0 0 0" />
<geometry>
<mesh
filename="package://robot_arm/
meshes/link_fixed2shoulder.STL"/>
</geometry>
<material
name="">
<color
rgba="0.79216 0.81961 0.93333 1" />
</material>
</visual>
<collision>
<origin
xyz="0 0 0"
rpy="0 0 0" />
<geometry>
<mesh
filename="package://robot_arm/
meshes/link_fixed2shoulder.STL"/>
</geometry>
```

```
</collision>
</link>
<joint
name="joint_fixed"
type="fixed">
<origin
xyz="0 0 0.049784"
rpy="1.5708 0 0" />
<parent
link="link_rotate2fixed" />
<child
link="link_fixed2shoulder" />
<axis
xyz="0 0 -1" />
</joint>
<link
name="link_shoulder2elbow">
<inertial>
<origin
xyz="0.13734 0 0"
rpy="0 0 0" />
<mass
value="0.13341" />
<inertia
ixx="0.00021458"
ixy="-3.5965E-06"
ixz="1.1269E-11"
iyy="0.00047665"
iyz="-3.0558E-12"
izz="0.00028193" />
</inertial>
<visual>
<origin
xyz="0 0 0"
rpy="0 0 0" />
<geometry>
<mesh
filename="package://robot_arm/
meshes/link_shoulder2elbow.STL"/>
</geometry>
<material
name="">
<color
rgba="0.79216 0.81961 0.93333 1" />
</material>
</visual>
<collision>
<origin
xyz="0 0 0"
rpy="0 0 0" />
<geometry>
<mesh
filename="package://robot_arm/
meshes/link_shoulder2elbow.STL"/>
</geometry>
</collision>
</link>
<joint
name="joint_shoulder"
type="revolute">
<origin
xyz="-0.017186 0.077011 0"
rpy="0 0 0.7096" />
<parent
link="link_fixed2shoulder" />
<child
link="link_shoulder2elbow" />
<axis
xyz="0 0 -1" />
<limit
lower="-3.3"
upper="1.3247"
effort="0"
velocity="3.33" />
</joint>
<link
name="link_elbow2wrist">
<inertial>
<origin
xyz="0.042421 0 0"
rpy="0 0 0" />
<mass
value="0.069218" />
<inertia
ixx="3.6022E-05"
ixy="8.149E-07"
ixz="3.0449E-12"
iyy="3.8788E-05"
iyz="7.7853E-13"
izz="2.499E-05" />
</inertial>
<visual>
<origin
xyz="0 0 0"
rpy="0 0 0" />
<geometry>
<mesh
filename="package://robot_arm/
meshes/link_elbow2wrist.STL"/>
</geometry>
<material
name="">
<color
rgba="0.79216 0.81961 0.93333 1" />
</material>
</visual>
<collision>
<origin
xyz="0 0 0"
rpy="0 0 0" />
<geometry>
<mesh
filename="package://robot_arm/
meshes/link_elbow2wrist.STL"/>
</geometry>
</collision>
</link>
<joint
name="joint_elbow"
type="revolute">
<origin
xyz="0.17221 0 0"
rpy="0 0 -0.7096" />
<parent
link="link_shoulder2elbow" />
<child
link="link_elbow2wrist" />
<axis
xyz="0 0 -1" />
<limit
lower="-2.7"
upper="1.23"
effort="0"
velocity="3.33" />
</joint>
<link
name="link_end_effector">
<inertial>
<origin
xyz="-0.015137 -0.019755 -0.037903"
rpy="0 0 0" />
<mass
value="0.087747" />
<inertia
ixx="0.00014586"
ixy="-1.7692E-05"
ixz="1.4942E-05"
iyy="0.00012169"
iyz="1.9931E-05"
izz="8.1825E-05" />
</inertial>
<visual>
<origin
xyz="0 0 0"
rpy="0 0 0" />
<geometry>
<mesh
filename="package://robot_arm/
meshes/link_end_effector.STL"/>
</geometry>
<material
name="">
<color
rgba="0.79216 0.81961 0.93333 1" />
</material>
</visual>
<collision>
<origin
xyz="0 0 0"
rpy="0 0 0" />
<geometry>
<mesh
filename="package://robot_arm/
meshes/link_end_effector.STL"/>
</geometry>
</collision>
</link>
<joint
name="joint_wrist_rotate"
type="revolute">
<origin
xyz="0.066193 -0.017501 0"
rpy="-1.5708 0 1.5708" />
<parent
link="link_elbow2wrist" />
<child
link="link_end_effector" />
<axis
xyz="0 0 -1" />
<limit
lower="-2.618"
upper="2.618"
effort="0"
velocity="3.33" />
</joint>
</robot>
```

Bibliography

[1] "Ax 12 a smart robotic arm." [Online]. Available: http://www.crustcrawler.com/products/AX12ASmartRoboticArm/

[2] "Robotis." [Online]. Available: http://support.robotis.com/en/product/auxdevice/interface/usb2dxl_manual.htm

[3] "Robotis." [Online]. Available: http://www.robotis.us/

[4] Y. Li, B. Mac Namee, and J. Kelleher, "Expecting the unexpected: Measure the uncertainties for mobile robot path planning in dynamic environment," in *Conference Towards Autonomous Robotic Systems*. Springer, 2013, pp. 363–374.

[5] D. Roy, "Algorithmic path planning of static robots in three dimensions using configuration space metrics," *Robotica*, vol. 29, no. 2, p. 295315, 2011.

[6] W. Parvez and S. Dhar, "Path planning of robot in static environment using genetic algorithm (ga) technique," *International Journal of Advances in Engineering & Technology*, vol. 6, no. 3, p. 1205, 2013.

[7] V. Tlach, I. Kuric, D. Kumičáková, and A. Rengevič, "Possibilities of a robotic end of arm tooling control within the software platform ros," *Procedia engineering*, vol. 192, pp. 875–880, 2017.

[8] T.-Y. Li and J.-C. Latombe, "On-line manipulation planning for two robot arms in a dynamic environment," *The International Journal of Robotics Research*, vol. 16, no. 2, pp. 144–167, 1997.

[9] R. Szabo and I. Lie, "Automated colored object sorting application for robotic arms," in *Electronics and Telecommunications (ISETC), 2012 10th International Symposium on*. IEEE, 2012, pp. 95–98.

[10] M. Nkomo and M. Collier, "A color-sorting scara robotic arm," in *Consumer Electronics, Communications and Networks (CECNet), 2012 2nd International Conference on*. IEEE, 2012, pp. 763–768.

[11] M. Gupta and G. S. Sukhatme, "Using manipulation primitives for brick sorting in clutter," in *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. IEEE, 2012, pp. 3883–3889.

[12] A. Ozgür and H. L. Akın, “Planning for tabletop clutter using affordable 5-dof manipulator.”

[13] D. Zhang, B. Wei, and M. Rosen, “Overview of an engineering teaching module on robotics safety,” in *Mechatronics and Robotics Engineering for Advanced and Intelligent Manufacturing*. Springer, 2017, pp. 29–43.

[14] B. Okal and O. Dunkley, “Design of a cheap chess robot: Planning and perception,” 2011.

[15] A. Fekete, “Robotic grasping and manipulation of objects,” 2015.

[16] M. Gupta, J. Müller, and G. S. Sukhatme, “Using manipulation primitives for object sorting in cluttered environments,” *IEEE transactions on Automation Science and Engineering*, vol. 12, no. 2, pp. 608–614, 2015.

[17] “Core components.” [Online]. Available: http://www.ros.org/core-components/

[18] V. S. Lindrup, “Robotic maintenance and ros-appearance based slam and navigation with a mobile robot prototype,” Master’s thesis, NTNU, 2016.

[19] “Moveit! motion planning framework.” [Online]. Available: http://moveit.ros.org/

[20] A. Gaddipati and A. Sadananda, “eysip-2017,” Sep 2017. [Online]. Available: https://github.com/eYSIP-2017/

[21] C. C. Filho, “clebercoutof/mixcell,” Aug 2017. [Online]. Available: https://github.com/clebercoutof/mixcell

[22] “Wiki,” Jun 2017. [Online]. Available: http://wiki.ros.org/sw_urdf_exporter

[23] “Robotis git.” [Online]. Available: https://github.com/ROBOTIS-GIT

[24] “Order a pr2.” [Online]. Available: http://www.willowgarage.com/pages/pr2/order

[25] 2017. [Online]. Available: http://docs.ros.org/kinetic/api/moveit_tutorials/html/index.html

[26] “Cyton epsilon 300.” [Online]. Available: http://www.robai.com/robots/robot/cyton-epsilon-300/

[27] “Dynamixel ax-12a robot actuator.” [Online]. Available: http://www.trossenrobotics.com/dynamixel-ax-12-robot-actuator.aspx